CMOR 421/521 Assignment 2: OpenMP

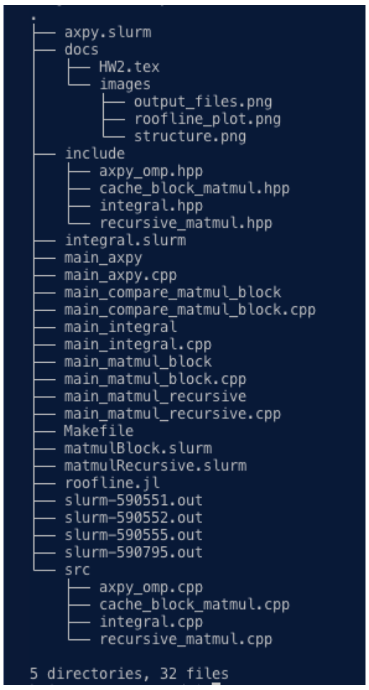
Yuhao Liu

March 17, 2025

Contents

1 Directory Structure

Below is my file organization for this assignment. My final zip file follows this structure (`docs/` for LaTeX, `src/` for source files, and `include/` for header files):



```
.
├── axpy.slurm
├── docs
│   ├── HW2.tex
│   └── images
│       ├── output_files.png
│       ├── roofline_plot.png
│       └── structure.png
├── include
│   ├── axpy_omp.hpp
│   ├── cache_block_matmul.hpp
│   ├── integral.hpp
│   └── recursive_matmul.hpp
├── integral.slurm
├── main_axpy
├── main_axpy.cpp
├── main_compare_matmul_block
├── main_compare_matmul_block.cpp
├── main_integral
├── main_integral.cpp
├── main_matmul_block
├── main_matmul_block.cpp
├── main_matmul_recursive
├── main_matmul_recursive.cpp
├── Makefile
├── matmulBlock.slurm
├── matmulRecursive.slurm
├── roofline.jl
├── slurm-590551.out
├── slurm-590552.out
├── slurm-590555.out
├── slurm-590795.out
└── src
    ├── axpy_omp.cpp
    ├── cache_block_matmul.cpp
    ├── integral.cpp
    └── recursive_matmul.cpp

5 directories, 32 files
```

Figure 1: structure

- The `main_axpy.cpp`, `main_matmul_block.cpp`, `main_compare_matmul_block.cpp`, `main_matmul_recursive.cpp`, and `main_integral.cpp` are include the `main` functions for the parallelized `AXPY`, `Cache Block Matrix Multiplication`, `Recursive Matrix Mult` and `Integral Pi`

- The `roofline.jl` plot the roofline figure, and calculate the percentage of peak performance.

- The `main_axpy`, `main_matmul_block`, `main_matmul_recursive`, `main_compare_matmul_bloc` and `main_integral` are execution file which already by compiled. You can directly run these file in your local computer as long as you have 16 threads in your computer. If you want to compile each of them, you can using the command in the next section.

- The `axpy.slurm`, `matmulBlock.slurm`, `matmulRecursive.slurm`, and `integral.slurm` are script file to help run the execution file in NOTXs. I already set 16 cores for each of task. But its were made by my personal account. You can directly using the command in the next section to run the file after changed the account.

- The folder `docs/` is to store the Latex file and images.

- The folder `include/` is the place for `.hpp` files. The `axpy_omp.hpp`, `cache_block_matmul.hpp`, `recursive_matmul.hpp`, and `integral.hpp` are in there.

- The folder `src/` is the place for `axpy_omp.cpp`, `cache_block_matmul.cpp`, `recursive_matmul.cp` and `integral.cpp` which are used to implemented all different methods for matrix transpose, matrix multiplication, and the timing analysis functions.

2 How to Build and Run the Code (In NOTXs)

- **Build Instructions:**
  - For all executable file, you can directly using

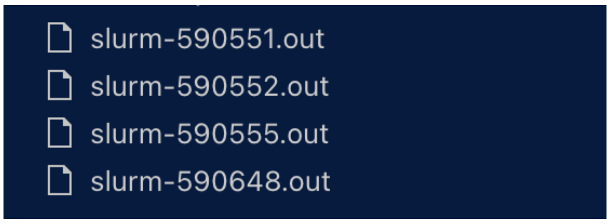
    ```
    make
    ```

    to build all executable file.

- **Running Instructions:** Once you build the execution file, it will appear as driver files. You can use the following command to run the code in NOTXs:

  ```
  sbatch axpy.slurm
  sbatch matmulBlock.slurm
  sbatch matmulRecursive.slurm
  sbatch integral.slurm
  ```

- **Execution:** When you execute the program, in the terminal you can see the output file in the drive folder:



```
slurm-590551.out
slurm-590552.out
slurm-590555.out
slurm-590648.out
```

Figure 2: Output files

These output file will storage the output for each task.

If you want to clean all executable files, using

```
make clean
```

3 Analysis

In this section, I will show you the analysis for these results. All the parallel *efficiency* for strong scaling, I followed the Amdahl's law:

$$E(n) = \frac{S(n)}{n}$$

where

$$S(n) = \frac{T(1)}{T(n)}$$

For the weak scaling, I only compute the parallel speed-up with

$$S(n) = \frac{T(1)}{T(n)}$$

3.1 AXPY

For the `AXPY` method in `version_1` and `version_2`, from the following table (Strong Scaling):

AXPY Version	**Threads**	**Runtime**	**Efficiency**
Version 1	1	5.94938	1
	2	1.72952	1.71995
	4	1.26784	1.17313
	8	1.16535	0.638155
	16	1.84623	0.201403
Version 2	1	2.67886	1
	2	1.45018	0.92363
	4	0.746254	0.897436
	8	0.464915	0.720256
	16	0.432303	0.387295

Table 1: Runtime and Efficiency of AXPY Versions

We can find that for `version_1`, after we increased the number of threads, the runtime without keeping decreasing instead of increasing, like `Threads = 16`. The efficiency increased at first and then decreased. On the other hands, we can see that the `version_2` is more general since it's runtime keeping decreased and the efficiency reduced normally.

3.1.1 Strong and Weak scaling

In this part, I will perform strong and weak scaling studies for up to 16 threads. The reults table is as following:

Scaling Experiment	AXPY Version	Threads	Runtime (s)	Efficiency (Speed Up)
Strong Scaling	Version 1	1	5.68555	1.000000
		2	2.45827	1.156410
		4	1.88886	0.752513
		8	2.24959	0.315922
		16	1.68469	0.210928
	Version 2	1	3.13006	1.000000
		2	1.55607	1.005760
		4	0.849503	0.921146
		8	0.509333	0.768178
		16	0.305709	0.639919
Weak Scaling	Version 1	1	0.360786	1.000000
		2	0.813838	0.443314
		4	2.11619	0.170489
		8	6.72802	0.053624
		16	21.7201	0.016611
	Version 2	1	0.190484	1.000000
		2	0.241768	0.787878
		4	0.246050	0.774168
		8	0.294072	0.647746
		16	0.406239	0.468897

Table 2: Strong and Weak Scaling Results for AXPY Operation

From the table 2, we can find the result of runtime and parallel efficiency for each type of scaling.

I think **version 2** clearly scales better than Version 1 in both the strong-scaling and weak-scaling experiments.

- In strong scaling, version 2 starts at about 3.13s (1 thread) and goes down to about 0.31 s (16 threads). That is roughly a 10× speedup, with an efficiency of around 64% on 16 threads. Version 1, on the other hand, shows a speedup of only about 3.4× at 16threads (down from 5.69s to 1.68s), for an efficiency around 21%.
- In weak scaling, the ideal is to keep runtime nearly constant as you add threads and proportionally increase the problem size. Version 2 remains under a second up to 16 threads (0.41s at 16threads), while Version 1 increases to over 21s at 16 threads. Accordingly, Version 2's efficiency in the weak-scaling section (0.47 at 16 threads) is substantially higher than Version 1's (0.011 at 16 threads).

3.1.2 Roofline Model(CMOR 521)

The figure of Roofline is:

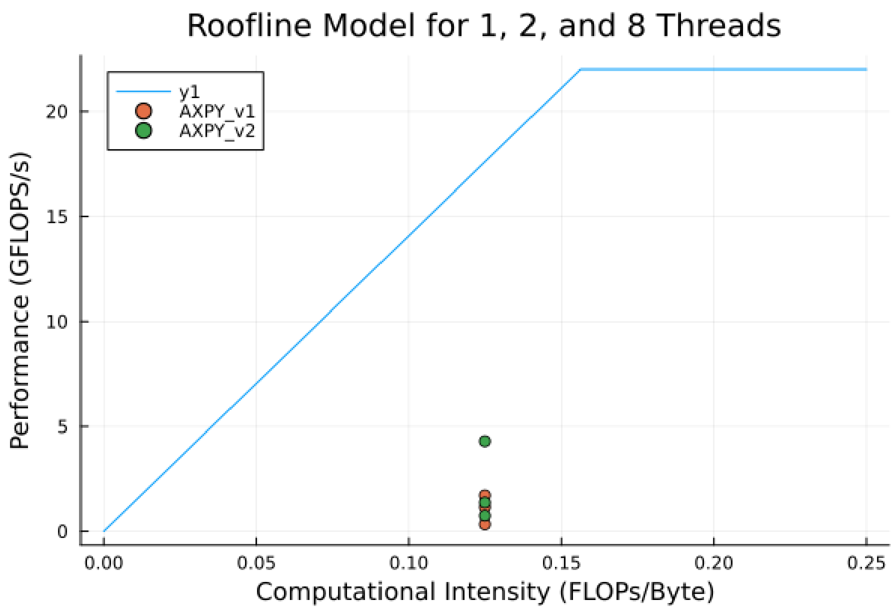


Figure 3: Roofline Model

The percentage of peak performance do you achieve for each case are as following:

Threads	AXPY Version	Performance (% of Peak)
1	v1	1.53%
2	v1	2.63%
8	v1	0.98%
1	v2	3.39%
2	v2	3.13%
8	v2	2.44%

Table 3: Achieved performance compared to theoretical peak performance

3.2 Matrix Multiplication (Cache Block)

3.2.1 Compare ”#omp parallel for” with ”#omp parallel for collapse(...)”

In this part, firstly, we need to compare the ”#omp parallel for” with ”#omp parallel for collapse(...)” and determined the optimal collapse parameter. Since the cache block matrix method can only stand for 2 or 3 collapse parameter, and the result for these three different parallelization way in the following table:

Version	Threads	Matrix Size	Runtime	Efficiency
MatMul Standard	1	2048	27.06997	1.00000
	2		13.48462	1.00373
	4		6.76744	1.00001
	8		3.89171	0.86947
	16		4.00292	0.42266
MatMul Collapse(2)	1	2048	28.53727	1.00000
	2		13.47711	1.05873
	4		6.76001	1.05537
	8		3.91194	0.91186
	16		4.01510	0.44422
MatMul Collapse(3)	1	2048	27.69317	1.00000
	2		13.52404	1.02385
	4		6.78140	1.02092
	8		3.86833	0.89487
	16		3.78207	0.45764

Table 4: Strong Scaling Experiment (Matrix Size = 2048 $\times$ 2048)

Based on the results, collapse(3) appears to be the optimal variant. Since all three variants have similar runtimes with one thread (Standard: 27.07s, collapse(2): 28.54s, collapse(3): 27.69s), so the difference isn’t significant when parallelism isn’t leveraged. At 2 and 4 threads, all versions achieve nearly identical performance, with efficiencies around or slightly above 1 (even a bit over 1 can happen because of improved cache reuse or other micro-architectural effects). As the thread count increases, the differences become clearer:

- With 8 threads, Collapse(3) shows the lowest runtime (3.86833s) compared to Standard (3.89171s) and Collapse(2) (3.91194s).
- At 16 threads, Collapse(3) delivers a runtime of 3.78207s and an efficiency of 0.45764, which is noticeably better than Standard (4.00292s, 0.42266 efficiency) and Collapse(2) (4.01510s, 0.44422 efficiency).

Therefore collapsing three loops provides a larger iteration space.

3.2.2 Strong and Weak scaling

In this part, I will perform strong and weak scaling studies for up to 16 threads. The reults table is as following:

Experiment	Version	Threads	Matrix Size	Runtime (s)	Efficiency (Speed Up)
Strong Scaling	MatMul Standard	1	2048	27.10687	1.00000
		2	2048	13.54723	1.00046
		4	2048	6.76382	1.00191
		8	2048	3.83429	0.88370
		16	2048	3.85602	0.43936
	MatMul Collapse(2)	1	2048	28.20344	1.00000
		2	2048	14.06791	1.00240
		4	2048	7.03822	1.00180
		8	2048	4.14558	0.85041
		16	2048	4.28617	0.41126
Weak Scaling	MatMul Standard	1	128	0.00623	1.00000
		2	256	0.02621	0.11878
		4	512	0.10443	0.01490
		8	1024	0.44713	0.00174
		16	2048	3.84999	0.00010
	MatMul Collapse(2)	1	128	0.00677	1.00000
		2	256	0.02709	0.12497
		4	512	0.10847	0.01560
		8	1024	0.50719	0.00167
		16	2048	4.29759	0.00010

Table 5: Strong and Weak Scaling Experiments for Matrix Multiplication

3.3 Recursive Matrix Multiplication

In this part, I will perform strong and weak scaling studies for up to 16 threads. The reults table is as following:

Scaling Experiment	Threads	Matrix Size	Runtime (s)	Efficiency (Speed Up)
Strong Scaling	1	2048	24.50431	1.00000
	2	2048	12.29185	0.99677
	4	2048	6.47467	0.94616
	8	2048	3.93354	0.77870
	16	2048	3.63948	0.42081
Weak Scaling	1	128	0.00589	1.00000
	2	256	0.02456	0.23989
	4	512	0.10145	0.05806
	8	1024	0.48898	0.01205
	16	2048	3.64802	0.00161

Table 6: Recursive Matrix Multiplication: Strong and Weak Scaling Results

3.4 (For CMOR 521) Integral Pi

In this part, I re-implement the integration code used in class using atomic operations and compared the performance of atomic operations to the reduction-based implementation by performing a strong scaling study for each implementation.

This part is the integration code used in class using atomic operations

```c
// Function that computes pi using atomic updates
double integral_atomic(int num_steps) {
    double step = 1.0 / num_steps;
    double sum = 0.0;
    #pragma omp parallel for
    for (int i = 0; i < num_steps; i++) {
        double x = (i + 0.5) * step;
        double temp = 4.0 / (1.0 + x * x);
        #pragma omp atomic
        sum += temp;
    }
    return step * sum;
}
```

This part is the integration code with the reduction-based implementation

```c
// Function that computes pi using the reduction clause
double integral_reduction(int num_steps) {
    double step = 1.0 / num_steps;
    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < num_steps; i++) {
        double x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    return step * sum;
}
```

The strong scaling result table is as following:

Implementation	Threads	Time (s)	Speedup	Efficiency	Pi
Atomic	1	0.964693	1.000000	1.000000	3.141593
	2	5.435353	0.177485	0.088742	3.141593
	4	6.622583	0.145667	0.036417	3.141593
	8	6.744995	0.143024	0.017878	3.141593
	16	5.788789	0.166649	0.010416	3.141593
Reduction	1	0.235411	1.000000	1.000000	3.141593
	2	0.123600	1.904615	0.952307	3.141593
	4	0.063234	3.722870	0.930718	3.141593
	8	0.033579	7.010662	0.876333	3.141593
	16	0.032575	7.226819	0.451676	3.141593

Table 7: Strong Scaling Study for PI Integration (num_steps = 100,000,000)