COMP SCI 557 Homework 4

Compile Environment

- CPU: `11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz`
- Memory: `16 GB`
- Compiler: `icc (ICC) 19.0.5.281 20190815`
- OS: `Ubuntu 22.04.5 LTS (Jammy Jellyfish)`
- MKL: `/s/intelcompilers-2019/amd64_rhel6/compilers_and_libraries_2019.5.281/linux/mkl`
- Compile commands (See `Makefile` for details):

```
$ make -j
  icc main.cpp MatMatMultiply.cpp Utilities.cpp -o build/main_2048_32
-Wall -O2 -qopenmp -DMATRIX_SIZE=2048 -DBLOCK_SIZE=32 -Wl,--start-
group /s/intelcompilers-
2019/amd64_rhel6/compilers_and_libraries_2019.5.281/linux/mkl/lib/inte
l64/libmkl_intel_lp64.a /s/intelcompilers-
2019/amd64_rhel6/compilers_and_libraries_2019.5.281/linux/mkl/lib/inte
l64/libmkl_core.a /s/intelcompilers-
2019/amd64_rhel6/compilers_and_libraries_2019.5.281/linux/mkl/lib/inte
l64/libmkl_intel_thread.a -Wl,--end-group -liomp5 -lpthread -lm -ldl
```

Runtime Environment

- CPU: `AMD Ryzen 7 7735HS with Radeon Graphics`
- CPU Cache:
  - L1d: `256 KiB (8 instances)`
  - L1i: `256 KiB (8 instances)`
  - L2: `4 MiB (8 instances)`
  - L3: `16 MiB (1 instance)`
- Memory: `Configured Memory Speed: 4800 MT/s, dual channel`
- OS: `Ubuntu 24.04.2 LTS (Noble Numbat)`
- Required Libraries: `libomp.so.5`
  - Install command: `sudo apt install libomp-dev`
- Runtime Environment Selection Explanation: Since CSL machine cannot output steady timing result, I chose to use my own server to run the program. Since I only have machines with AMD CPU, the improvement of replacing hand-code kernel call with MKL library call might not be too obvious.

Performance Analysis

Matrix Size	Block Size	Base (16T, ms)	Base (1T, ms)	Base Speed-Up	MKL (16T, ms)	MKL (1T, ms)	MKL Speed-Up
1024	16	12.77	64.53	5.06	9.74	44.54	4.57
1024	32	12.66	70.70	5.58	9.70	44.70	4.61
1024	64	17.51	84.90	4.85	9.69	46.49	4.80
2048	16	92.02	705.47	7.66	75.32	358.00	4.75
2048	32	101.47	648.36	6.39	64.77	360.38	5.56
2048	64	146.25	710.84	4.86	70.67	355.28	5.03
4096	16	1072.74	7881.43	7.35	547.94	2849.22	5.20
4096	32	821.90	5056.83	6.15	530.11	2816.41	5.31
4096	64	1197.39	5736.21	4.79	516.84	2764.39	5.35

NOTE: Since `MKL` version cannot change the block size, the block size should be ignored for `MKL` version.

Commentary on Performance Analysis

> Does operating at certain (large or small) matrix sizes seem to make it less important what the size of blocks is (for performance)?

Observing the trends of the performance from above, block size is still quite important for large matrix. In fact, a poorly tuned block size (e.g., 64 for a 4096×4096) can hurt performance badly compared to a better choice (like 32).

For small matrices, block size is also important. For example, for 1024×1024, smaller block sizes (16 or 32) are better than 64. If the block size is too large, there might not be enough work to spread across threads, The overall sensitivity to block size remains significant across all tested dimensions, but which block size is best can shift as the matrix grows.

> Are you seeing the same changes in parameters such as block size helping performance in some resolutions and hindering performance in others?

Yes. In the above table, we can see that:

- For 2048×2048, the best performance at 16 threads is with a block size = 16 (92.02ms). Increasing block size to 32 actually makes it much slower (101.47ms).
- But for 4096×4096, block size 16 is no longer the best; block size 32 becomes faster (821.90 ms vs. 1072.74s with block size 16).
- In fact, if we use more matrix size and block size for comparation, we might find that for certain size of small matrix, a block size of 16 can give better performance; while for sufficiently large matrix, a block size of 32 give better performance. This can be seen in table.md

Therefore, for different matrix sizes, the optimal block size can vary. A single choice can help at one matrix size or shape and hinder at another.

> For some of the resolutions/block sizes in your experiment, are you witnessing speed-up that is not quite proportional with the number of cores used when using all vs. one thread?

Yes. The measured speed-ups (4x - 7x) are all below the ideal scaling of 16x for 16 threads, or below the ideal scaling of 8x for 8 physical cores. One of the reasons could be memory bandwidth, which becomes significant as more cores access memory simultaneously, or the block size becomes larger. Also, the overhead of synchronization, management of threads, and potential load imbalance between threads can also account for it. Even though the use of block can improve cache locality and reduces cache misses, selecting a correct block size is very important, especially when the total size of matrix cannot fit entirely inside the cache.

Bonus

Matrix Size	Block Size	Bonus (16T, ms)	Bonus (1T, ms)	Bonus Speed-Up
(1024x4096) * (4096x1024)	16	51.62	336.24	6.52
(1024x4096) * (4096x1024)	32	51.38	317.57	6.18
(1024x4096) * (4096x1024)	64	71.51	350.48	4.90
(2048x2048) * (2048x2048)	16	87.50	652.68	7.46
(2048x2048) * (2048x2048)	32	100.88	601.73	5.96
(2048x2048) * (2048x2048)	64	128.64	703.04	5.46
(4096x1024) * (1024x4096)	16	229.69	1519.67	6.61
(4096x1024) * (1024x4096)	32	199.28	1263.68	6.34
(4096x1024) * (1024x4096)	64	274.26	1417.37	5.17

Commentary on Performance Analysis for Bonus

As we can see, the runtime of the matrix size `(2048x2048) * (2048x2048)` is very close to the runtime in the base version, with the `Discrepancy between candidate kernel and reference kernel` being less than `0.0003` for most of the time, this verifies that the implementation of the bonus version is correct.

Observed from the above, even with the same number of multiplications, with different shapes of the matrices and different block sizes, the performance varies. One possible reason for the behavior is that the resulting matrices. For example, for `(1024x4096) * (4096x1024)`, the resulting matrix is `1024x1024`, while for `(2048x2048) * (2048x2048)`, the resulting matrix is `2048x2048`. The resulting matrix of the second one is four timers larger than the first one, and in the algorithm, we do have to save the result to the resulting matrix, which takes times. This could explain for why the performance of the second one is worse than the first one.

But since the storing process is **NOT** all we have in the algorithm, the second one is not four times slower than the first one. In fact, we observe that it is roughly twice as slow. However, this is not absolutely true, and it would have changed when we have some larger matrices that cannot fit entirely inside the cache.