Assignment 1: Sequential warmup

Xun Zhu

Question 1

The source code is included in the folder under the name `main.c`. Parameters are defined as C-preprocessor variables and algorithm (henceforth referred to as `i-j`, `j-i`, or `tiled`) and block size are chosen at the compile time.

Note that in the tiled version, I used four separate for-loops for the edge cases when N is not divisible by the block size, as opposed to having one for-loop and branching using if-statements, which would be slower.

With $N = 18,000$ and using the GNU Compiler Collection (`gcc`), the running time for i-j was around 2 seconds. However, with the array being of type double (8 bytes = 64 bits), this means the the memory allocated for either arrays would be

$$64 \times 18,000^2 \approx 4.83 \times 2^{32}\text{bits},$$

which necessitates the `-mcmodel=medium` parameter when compiled.

An interesting observation is that `icc` (14.0.2.144 Build 20140120) did not generate faster code than `gcc` (7.1.0). The binary created by the former finished on average 2.118 seconds. The following results are therefore based on compilation using `gcc`, unless specified otherwise.

I wrote a Python script for invoking the compiler with various parameter combinations, collecting the results, and aggregate the results into a CSV file.

I noticed that the first few runs take much longer than following runs, usually over 8 minutes or so. I suppose this is when the OS is recovering from the last task which potentially tapped into swap, but I'm not so sure. For example, here are the logs of the first few runs in a particular experiement, notice how the wall clock time dropped from hundreds of seconds to about two seconds:

algo	bs	wall_time	rep	l1_load_misses	llc_load_misses
ij	1	194.70775351	1	4719950260	168155701
ji	1	589.410447918	1	11118078659	317424545
tiled	512	212.40171788	1	5067632874	169008293

algo	bs	wall_time	rep	l1_load_misses	llc_load_misses
tiled	1	11.366876506	1	645914378	47143981
tiled	2	1.969204341	1	451714702	42381861
tiled	3	1.965555544	1	451693187	42362063
tiled	4	1.992775057	1	451775277	42408239
tiled	5	1.99024135	1	456029172	45395570

To compensate for that, I decided to do a couple of *warm-up* compilation and execution rounds before the real runs.

Question 2

Using 10 trials for each tile size, execute your tiled version for all tile sizes between 1 and 300. Plot the average elapsed time vs. the tile size.

See Figure 1 (last panel.)

Explain the trend that you observe. What is the best tile size?

The best wall-clock time (1.614 sec) was achieved at $b = 4$. However, as is evident on Figure 1, among the lowest block sizes (less than 20) the wall-clock time exhibited highly irregular growth trend. Although within any fixed block size the time recorded from different repeated runs remained relatively consistent (maximum s.d. is no higher than 0.026, except when $b = 1$). After the initial chaotic turbulence, the wall-clock time growed steadly, until reaching around $b = 100$, from which point onward it stayed around 1.89 with minimal growth.

What speedup is achieved w.r.t. the basic version?

As shown on Figure 1 (pink lines) algorithm `i-j` took on average 1.963 seconds, where as algorithm `j-i` took around 2.211. This makes the algorithm `tiled` (at $b = 4$) having a speed-up factor of 1.216 comparing to `i-j`.

Question 3

How do the number of cache misses compare between the fastest tiled version and the naive version?

At $b = 4$ `tiled` has 230,809,152 L1 load misses, whereas `i-j` has 451,674,532.

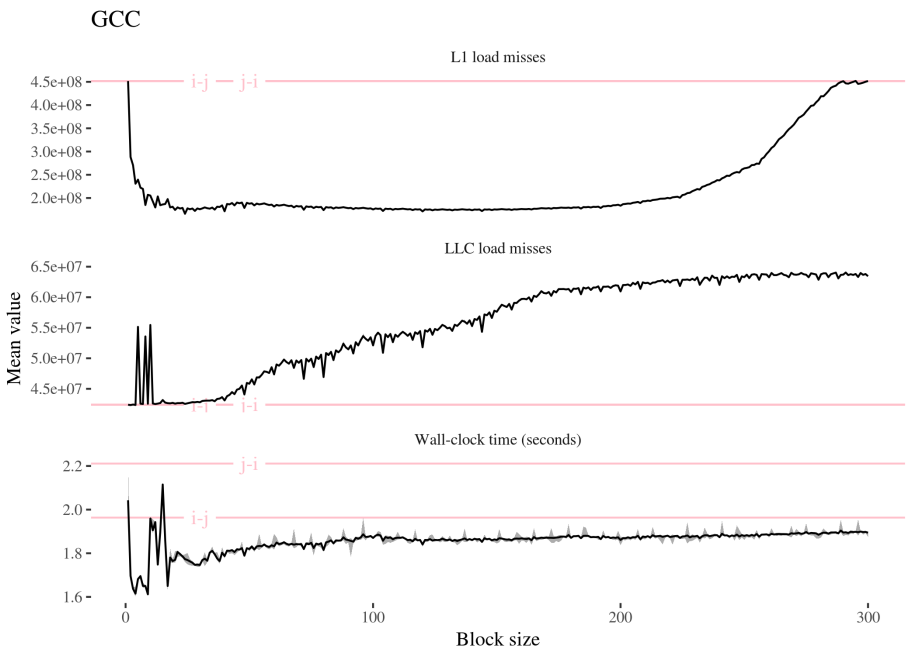

Figure 1: L1 load misses, LCC load misses, and Wall-clock time, plotted against the block size in the tiled algorithm. The thick black line indicates the mean value among 10 repeated runs, whereas the gray shade around the black line indicates the minimum and maximum value reached within these 10 runs.

Add the corresponding curves to the plot from the previous question. (You may want to normalize curves in some fashion.)

See Figure 1 (the first two panels.)

Discuss the trends you observe in the plot. (Do they make sense? if so why? if not why not?)

The dropping of L1 load misses started rapidly from $b = 1$ to $b = 20$, and remained stable until around $b = 200$, from which point onward it steadly grows until finally reaching back up to the same level as that of `i-j` and `j-i`, at around $b = 280$. The lowest L1 load misses was reached at $b = 24$, where it is 166,038,348.

The inital turbulence is likely related to the number of registers in CPU, which might be sensitive to the change of divisibility of the block sizes by certain numbers.

The `prod` machines on HPC have a `PAGESIZE` of 4096 bytes, which is about 512 doubles, or two 16-by-16 double 2d-arrays. They have 32K of L1d cache and 30720K L3 cache (LLC). 32K is about two 44-by-44 double 2D-arrays. And 30720K is about two 1385-by-1385 double 2D-arrays. The incrase of LLC load misses from 20 onward can be explained by the fact that as the block size gets larger,

However, rather perplexingly, the dramatic increase of L1 misses from $b = 200$ onward is not accompanied by any significant increase in wall-clock time. Indeed, at $b = 280$, `tile` has as many L1 load misses as `i-j` or `j-i` does, but it remained marginally faster than `i-j`. One reason could be that the cause (and occurrence pattern) of L1 load misses is completely different in `i-j` and `tiled` at larger b's. And the miss pattern in `tiled` is in a way less costly.

The other thing hard to explain is that `j-i` had significantly worse performance than `i-j` does, despite the fact that they have exactly the same L1 load misses and LLC load misses. I observe that this oddity is not present when compiling using `icc` – with `icc` the performance of `i-j` and `j-i` was exactly the same. Therefore I suspect that there's an optimization procedure that is incorrectly implemented in GCC, so that it over-looks it at certain cases.

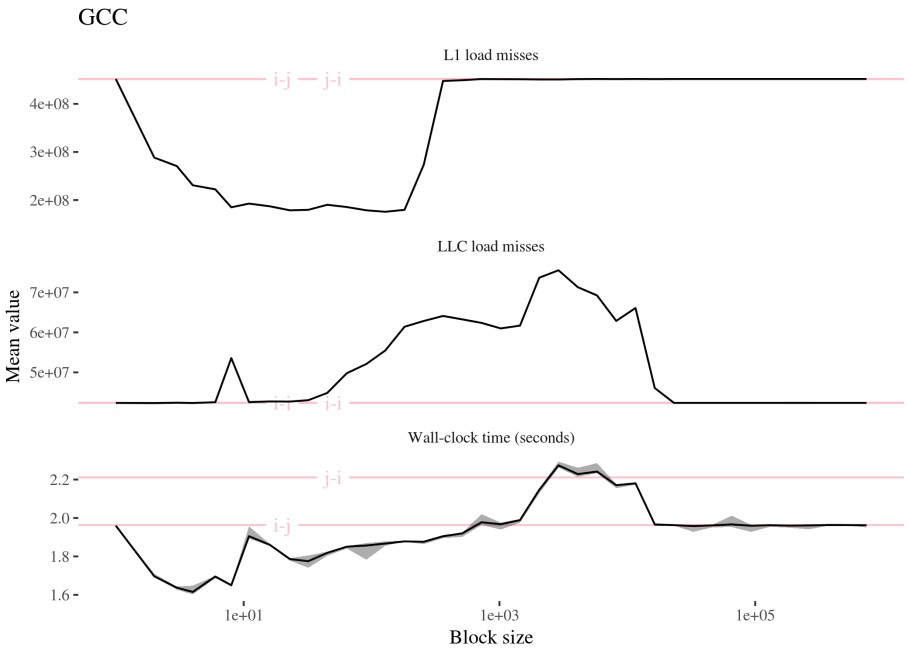

Figure 2: The same plot as Figure 1 but using exponentially increasing b.