Exercise #1 (HPC . ICS632 Sep. 2015) ---------- Ehsan Kourkchi

Sequential Warmup

1)

The main program: `exercise1.c`

To compile on `uhhpc` cluster:

```
$ icc  exercise1.c -o exercise1.x  -Ofast -D TILE=1 -D N=18000 -D
b=10 -mcmodel medium -shared-intel
```

To compile on my own desktop:

```
$ gcc exercise1.c -o exercise1.$iteration.x  -Ofast -D TILE=1 -D
N=16000 -D b=$i -mlong-double-64
```

N: is the array size
b: is the tile size
TILE: runs in the naive mode if set to 0, and in the tiling mode if set to 1

To use Perf command

```
$ perf stat -e L1-dcache-load-misses -e LLC-load-misses ./exercise1.x
```

I have tested the naive version versus the tiled version, to make sure that they both produce the same results.

2)

I ran 50 trials, for b= 1..300

To do that automatically, I wrote a script "**`task.bash`**", that compiles the program for each iteration and then stores the outputs in a file.

Exercise #1 (HPC . ICS632 Sep. 2015) ---------- Ehsan Kourkchi

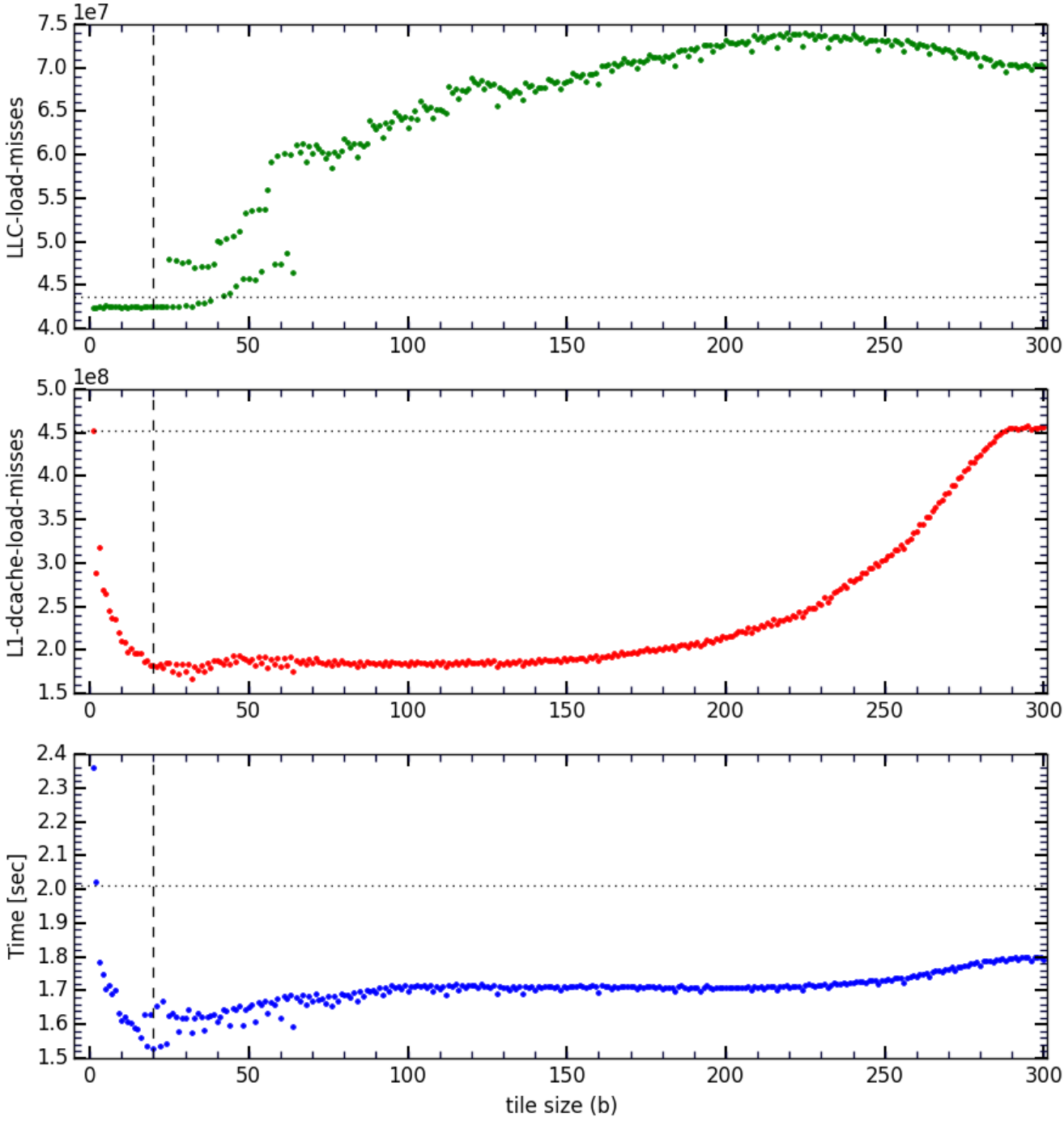

Fig 1: Efficiency analysis of tiling algorithm using UH-HPC cluster. Upper pane: LLC cache load-miss vs. tile size, b. Middle panel: L1-cache load miss vs b. Lowe panel: running-time vs b. The vertical dashed-line in all panels shows the position of b=20, for which the running-time is minimum. The horizontal dotted line indicate the corresponding value when the program is run in naive mode.

Exercise #1 (HPC . ICS632 Sep. 2015) ---------- Ehsan Kourkchi

For each iteration, there would be an output file "`output.xxx.txt`" which has 4 columns, i.e. b, time, L1-dcache-load-misses, LLC-load-misses. `xxx` is the iteration number.

To plot the averaged results, there is a python program "`plot_time.py`" that reads all output files, averages the numbers and plot them together.

As seen in the above figure (the bottom most panel), as the tile size increases, the run-time decreases rapidly until b=20, for which we have the minimum running time. For larger tiles (b>20), the run-time increases gradually, which means the usage of memory is getting less efficient, in terms of the size of cache, and they way out tiling method access the memory. Even around b=20, there are instabilities in the run-time, it means that incrementing or decrementing the tile-size just by one, results in a rapid change in the running-time.

As the tile-size increases, we deal with larger sub-arrays, that have elements stored apart in the memory line. Therefore, after the certain tile-size, the number of cache-load-miss increases again, as we in the other panels of the last page figure.

For b=20, the minimum running time is Min_time: **1.52994387035 (s)**

Therefore the speed-up w.r.t. the naive version is
speed_up = 2.01/1.53 = 1.31

This is 10 iteration of running the program in the naive mode on UH-HPC cluster

Table 1

iteration	Time (s)	L1-dcache-load-misses	LLC-load-misses
1	2.012115802	453086192	43676811
2	2.01058882	453069728	43681614
3	2.010919608	453066979	43677658
4	2.009834078	453083180	43687323
5	2.009536412	453072544	43682121
6	2.008302763	453083740	43686472
7	2.010593376	453081733	43685268
8	2.010904983	453089501	43683943
9	2.012674928	453089792	43691890
10	2.00886361	453085556	43683539
Average	**2.010433438**	**453080894.5**	**43683663.9**

Exercise #1 (HPC . ICS632 Sep. 2015) ---------- Ehsan Kourkchi

3)

For b=20,

- T=1.53 (s),
- L1-dcache-load-misses = 181971987
- LLC-load-misses = 42458716

Table 2

	Run-Time (s)	L1-dcache-load-misses	LLC-load-misses
naive	2.01	4.5E8	4.3E7
b = 20	1.53	1.8E8	4.2E7
Improvement Factor	1.31	2.5	1.02

Some elaboration about L1 cache load miss:

The middle panel in Fig 1, shows the L1 cache load miss. For smaller tile size, it make sense to have lower load miss. For all tile size, b, smaller than ~290, L1 cache load miss is smaller when using the til-version. The dotted line shows l1 cache load miss for the naive version.

We note that, the major source of the cache load is to access the array elements in different rows, or in another word when we span the array along its columns.

Let's say, each load into the L1 cache takes 100 rows of the array in our example. Therefore if the tile size, b, is smaller than 100, cache load miss would be theoretically zero (if we neglect the other sources of cache load miss). If the tile size becomes larger than 100, then even inside each tile, we need to reload the cache and impose a large cache load miss on the system. That's why the value of the L1 cache load miss stays the same for large number of b-values. In middle panel of Fig 1, cache load miss stays the same for 20<b<150.

Increasing the tile-size, would eventually create the situation that L1 cache load miss becomes the same as that of the naive version (for b>~290). In these cases, the tile size is that big that even for spanning one tile, L1 cache need to get updated several times.

Some elaboration about Last Level Cache (LLC)

LLC is technically the biggest cache which is shared by multi-core processors. The size of the cache is huge compared to L1. Therefore it gets less frequently updated when running the program. So it is reasonable to have one order of magnitude smaller cache load miss compared to the L1 cache.

Exercise #1 (HPC . ICS632 Sep. 2015) ---------- Ehsan Kourkchi

I didn't expect to see a rise in the LLC load miss when increasing the tile size, b. As seen in upper panel of Fig 1, the LLC load miss is even larger than its value when running the program in its naive version. I could not find any reasonable explanation for this behavior.

In Table 2, we compare between the best optimum tile-program and the naive one.

Which tile size to choose?

Based on the plots in Fig 1, we see that around b=20, the running-time is the smallest, however there are some instabilities in the curve, meaning that increasing or decreasing the tile size can increase running-time non-linearly.
Looking at all tree diagrams together, 10<b<30 sounds reasonable for optimizing the cache load miss. In terms of running time, this interval results in the running-times of <~1.6 (s) which is kind of optimum.

The lesson to learn is that, to find the most optimum value, both time and cache load miss diagrams should be considered.

Some analysis in my own desktop computer

I ran the same analysis for my desktop computer for smaller size of array (i.e. N=16,000)
This is 10 iteration of running in the naive mode on my desktop computer

Table 3

Iteration	time (s)	L1-dcache-load-misses
1	0.559564768	332955290
2	0.58413899	332479926
3	0.5566014	335629932
4	0.551123027	332879650
5	0.551696179	334073000
6	0.584902027	333675738
7	0.568789928	332629877
8	0.550343495	332589473
9	0.567114583	332868770
10	0.551251515	332975591
Average	**0.5625525912**	**333275724.7**

Exercise #1 (HPC . ICS632 Sep. 2015) ---------- Ehsan Kourkchi

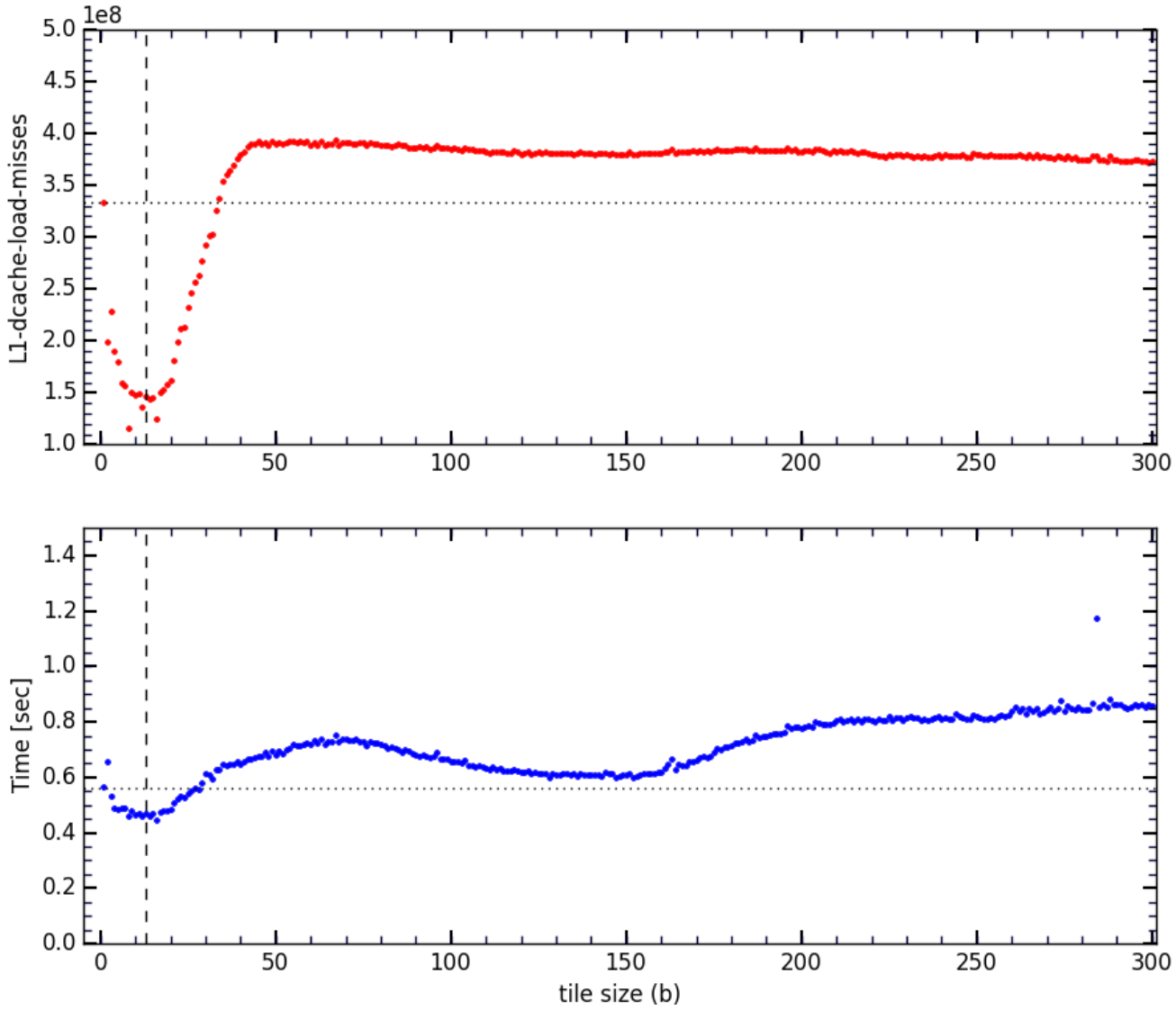

Fig 2: The same as Fig 1, for my desktop computer

For b = 13, we have a minimum in the running-time.
Time = 0.45 (s)
L1-dcache-load-misses = 143584864

speed-up = 0.56/0.45 = 1.24
L1 cache load-miss improvement factor = 333275725/143584864 = 2.32

As seen in Fig. 2, due to the different architecture of my computer, these curves look different, and even for b>30, tiling does not necessarily results in a better performance. T

here is no large plateau in the plot of cache load miss. In fact the plateau is squeezed into the global minimum of the curve (upper pane in Fig. 2). After the local minimum, we reach the maximum usage of the memory locality in each tile at b=~50. For larger tile szie, i.e. b>50, tiles are big that even for each tile, the L1 cache needs to be loaded several times.

Exercise #1 (HPC . ICS632 Sep. 2015) ---------- Ehsan Kourkchi

Comparing Fig 2, with Fig1, I conclude that, the L1 cache size of my desktop computer is way smaller than that of UH-HPC (which is saturated in b=~290). One has to note that the chosen array size for UH-HPC was 18,000x18,000 and for my desktop PC was 16,000x16,000. Even using a smaller array in my PC, I get smaller tile size for the L1 cache saturation, which mean a smaller L1 cache size.