

# **ECE 565: Performance Optimization & Parallelism**

## **Homework 2**

**Prathikshaa Rangarajan (pr109)**

October 2, 2019

## Problem 1

L1 DCache	Parameters
Cache Size	512 B
Line Size	64 B
# of Cache Lines = cache size/line size	$512/64 = 8$
# of ways	2
# of sets = (#cache lines)/(#ways)	$8/2 = 4$

Table 1: L1 DCache Parameters Provided and Derived

Address Part	# of Bits
Total Address size (assuming 64-bit machine)	64
Offset = reference to byte # within cache line (64 B)	$\log(64, 2) = 6$
Index = reference set number (4 sets)	$\log(4, 2) = 2$
Tag = remaining # of bits	56

Table 2: Addressing bits

Tag	Index	Offset
56	2	6

Address Format

Set # = Index	Tag	Data
00	0 ① 1a3 ④ 8f2 ⑦ 52c	14327 ⑥ 1432f <del>8f22a</del> <del>52c22</del>
01	1 ③ df1 2 ⑤ cde 3 ⑩ f12	<del>df148</del> cde4a f125c
10	4 ⑨ 92d 5	92da3
11	6 ⑦ abc 7	⑧ abcf2 abcde

Figure 1: Cache hit/miss full

Set #	V	Tag	Data
00	1	143	1432f
	1	52c	52c22
01	1	cde	cde4a
	1	f12	f125c
10	1	92d	92da3
	0	x	x
11	1	abc	abcf2
	0	x	x

Table 3: Cache Final

## Problem 2

Level	Hit Time
L1 Cache	1 cycle
L2 Cache	20 cycles
Memory	250 cycles

(a)

L1 Miss Rate	L2 Miss Rate	Memory Access Time (cycles)
2%	40%	$1 + 0.02 \cdot (20 + 0.4 \cdot 250) = 3.4$

(b)

L1 Miss Rate	L2 Miss Rate	Memory Access Time (cycles)
10%	5%	$1 + 0.1 \cdot (20 + 0.05 \cdot 250) = 4.25$

## Problem 3

```
pr109@vcm-6252:~/ece565-perf/hw2/problem3$ lscpu | grep cache
L1d cache:          32K
L1i cache:          32K
L2 cache:           256K
L3 cache:           35840K
pr109@vcm-6252:~/ece565-perf/hw2/problem3$
```

Figure 2: lscpu for cache details

```
pr109@vcm-6252:~/ece565-perf/hw2/problem3$ for file in /sys/devices/system/cpu/cpu0/cache
/index0/*; do echo $file; cat $file; done;
/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
/sys/devices/system/cpu/cpu0/cache/index0/id
0
/sys/devices/system/cpu/cpu0/cache/index0/level
1
/sys/devices/system/cpu/cpu0/cache/index0/number_of_sets
64
/sys/devices/system/cpu/cpu0/cache/index0/physical_line_partition
1
/sys/devices/system/cpu/cpu0/cache/index0/power
cat: /sys/devices/system/cpu/cpu0/cache/index0/power: Is a directory
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list
0
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_map
1
/sys/devices/system/cpu/cpu0/cache/index0/size
32K
/sys/devices/system/cpu/cpu0/cache/index0/type
Data
/sys/devices/system/cpu/cpu0/cache/index0/uevent
/sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity
8
```

Figure 3: Cache Index Files for more details such as line size, # sets and associativity

## (c) Program Description

### I Directions to run the program

The script run.sh has been configured to execute the C program binary with the required inputs to measure bandwidth of cache and main memory.

The make file allows compilation of various optimization levels using the commands 'make o1', 'make o2' and 'make o3'. After generating the binary, the script may be executed as './run.sh'.

The program takes inputs as described below and this message can be seen when running the executable per without any arguments as './perf':

```
1 Usage: ./cache_test <cache_size(KB)> <mode(1|2|3)> <num_iterations>
2 Modes:
3 1: write only workload
4 2: read to write ratio = 1:1
5 3: read to write ratio = 2:1
```

Here, num of iters is an independent averaging metric to specify the num of times the testing is performed. The program takes the memory size (a number) measured in KB as an input and generates the appropriate number of array elements and loop iterations.

### II Program Structure

The program has three functions for the different types of workloads which can be invoked by the three modes described above.

The bandwidth measurement is done using a 1-D array. In order to measure bandwidth of the L1 cache, array size is same as the L1 cache size (32K). All elements are initialized to ensure subsequent L1 cache hits.

### III Write Only Workload

Write a constant into every array element in sequential order.

### IV One Read, One Write

Read an array element and write a constant into the next array element.

### V Two Reads, One Write

Read two sequential array elements and write a constant into the array element after.

### VI Stressing Bandwidth

Different compilation flags were tested with objdump. The compiled code shows the expected loads and stored within the loop. The following results are using the O2 optimization. Compile as 'make o2'.

## VII Results

Based on the program I ran, the following are the performance results:

```
1 -----
2 Cache Bandwidth Measurement
3 -----
4 Time(s) = 0.006101
5 Num of iters: 1000
6 Data = 62MB
7 Bandwidth = 10.74GBps
8 ./perf 32 1 1000
9 -----
10 Time(s) = 0.001751
11 Num of iters: 1000
12 Data = 62MB
13 Bandwidth = 37.43GBps
14 ./perf 32 2 1000
15 -----
16 Time(s) = 0.001736
17 Num of iters: 1000
18 Data = 93MB
19 Bandwidth = 56.61GBps
20 ./perf 32 3 1000
21 -----
22 Main Memory Bandwidth Measurement
23 -----
24 ./perf 4194304 1 1000
25 Time(s) = 4.604504
26 Num of iters: 1000
27 Data = 70000MB
28 Bandwidth = 15.94GBps
29 -----
30 ./perf 4194304 2 1000
31 Time(s) = 3.566904
32 Num of iters: 1000
33 Data = 70000MB
34 Bandwidth = 20.58GBps
35 -----
36 ./perf 4194304 3 1000
37 Time(s) = 3.541490
38 Num of iters: 1000
39 Data = 105000MB
40 Bandwidth = 31.09GBps
41 -----
```

## VIII Analysis

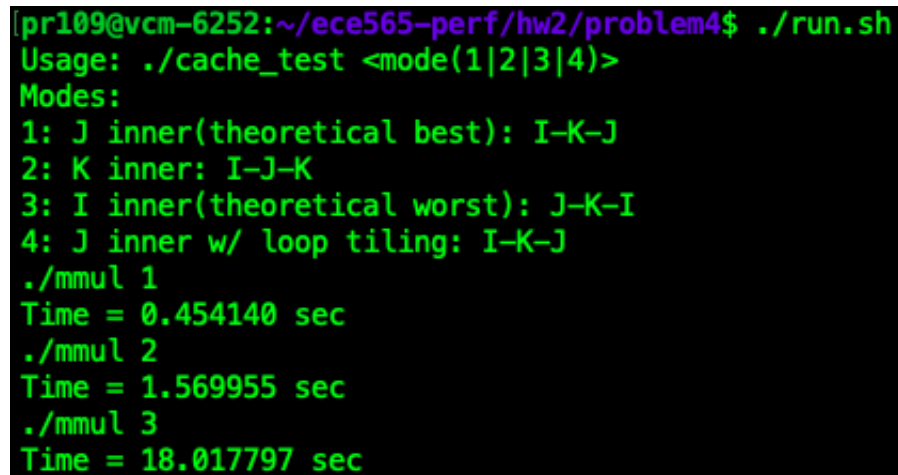
The read-write bandwidth is higher than write only bandwidth which may correspond to the existence of parallel read-write ports. Since the 2:1 read:write ratio is the highest bandwidth, it may be inferred that there are at least two read ports to every write port on the cache.

Main memory results also show similar relative results, while the absolute bandwidth drops. This maybe due to high latency and strain on the Main Memory interconnect resulting in a lower bandwidth.

However, cache bandwidth expectation is at least an order of magnitude higher than what was observed. This may be due to insufficient parallelism in code or inefficient code with too much loop overhead.

## Problem 4

### (a) Implementing the three loop orderings over entire matrices



```
pr109@vcm-6252:~/ece565-perf/hw2/problem4$ ./run.sh
Usage: ./cache_test <mode(1|2|3|4)>
Modes:
1: J inner(theoretical best): I-K-J
2: K inner: I-J-K
3: I inner(theoretical worst): J-K-I
4: J inner w/ loop tiling: I-K-J
./mmul 1
Time = 0.454140 sec
./mmul 2
Time = 1.569955 sec
./mmul 3
Time = 18.017797 sec
```

Figure 4: Performance of the three loop orderings

### (b) Results

The theoretical and practical expectations are met as seen in the results in fig. 4.

When  $j$  is the innermost loop, an element of  $A$  is taken and computed against row major access of  $B$  and  $C$ . The row major access results in a series of cache hits with the periodic miss when moving onto the next cache line.

Using  $i$  as the innermost loop results in worst case performance because  $A$  and  $C$  are accessed in column major pattern forcing every access to be a miss.

### (c) Loop Tiling

In it's existing form the algorithm is good enough, as the innermost loop iteration loads  $B$  and  $C$  along row length  $1024 * 8 * 2 = 16384B$  which comfortably fits in the 256KB L2 cache.

A tiling experiment on this structure was to deal with loop tiles of 512 elements along  $j$  and  $k$ .

#### (d) Results

```
[pr109@vcm-6252:~/ece565-perf/hw2/problem4$ ./run.sh
Usage: ./cache_test <mode(1|2|3|4)>
Modes:
1: J inner(theoretical best): I-K-J
2: K inner: I-J-K
3: I inner(theoretical worst): J-K-I
4: J inner w/ loop tiling: I-K-J
./mmul 1
Time = 0.455182 sec
./mmul 2
Time = 1.613903 sec
./mmul 3
Time = 17.942037 sec
./mmul 4
Time = 3.250157 sec
```

Figure 5: Tiling performance

As seen in fig. 5, the performance after loop tiling is about 3secs and is worse than the performance without tiling, although better than the worst case loop ordering.

A previous experiment with loop tiling at all three levels yielded performance worst than the worst case loop ordering.