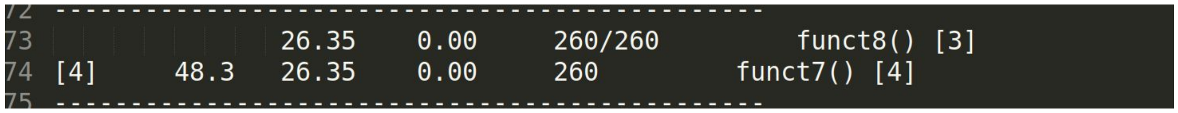
# Assignment 2

- Shaunak Badani
20171004

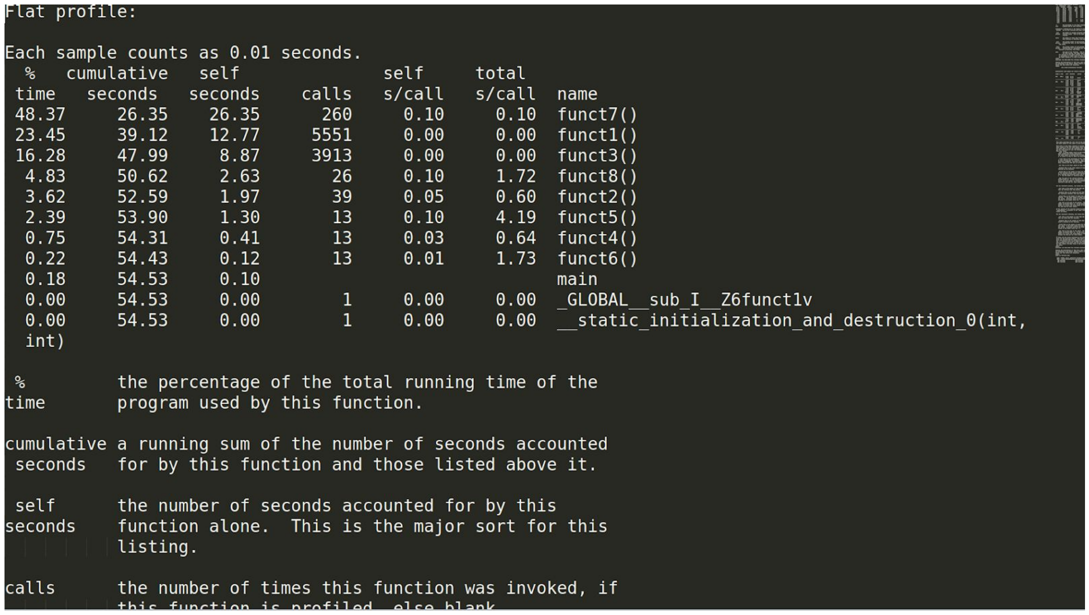
## Question 1.

1. Function **funct7** takes the most amount of time.
    a. Since this function is called 260 times, theoretically, if no of cores available were 260, then around 260x speedup could be expected, and the time required would be 0.1 seconds instead of 26 seconds.
    b. This function is called **260** times.
    c. This function does not call any other function. This is evident from the call tree of the program, in the following screenshot :



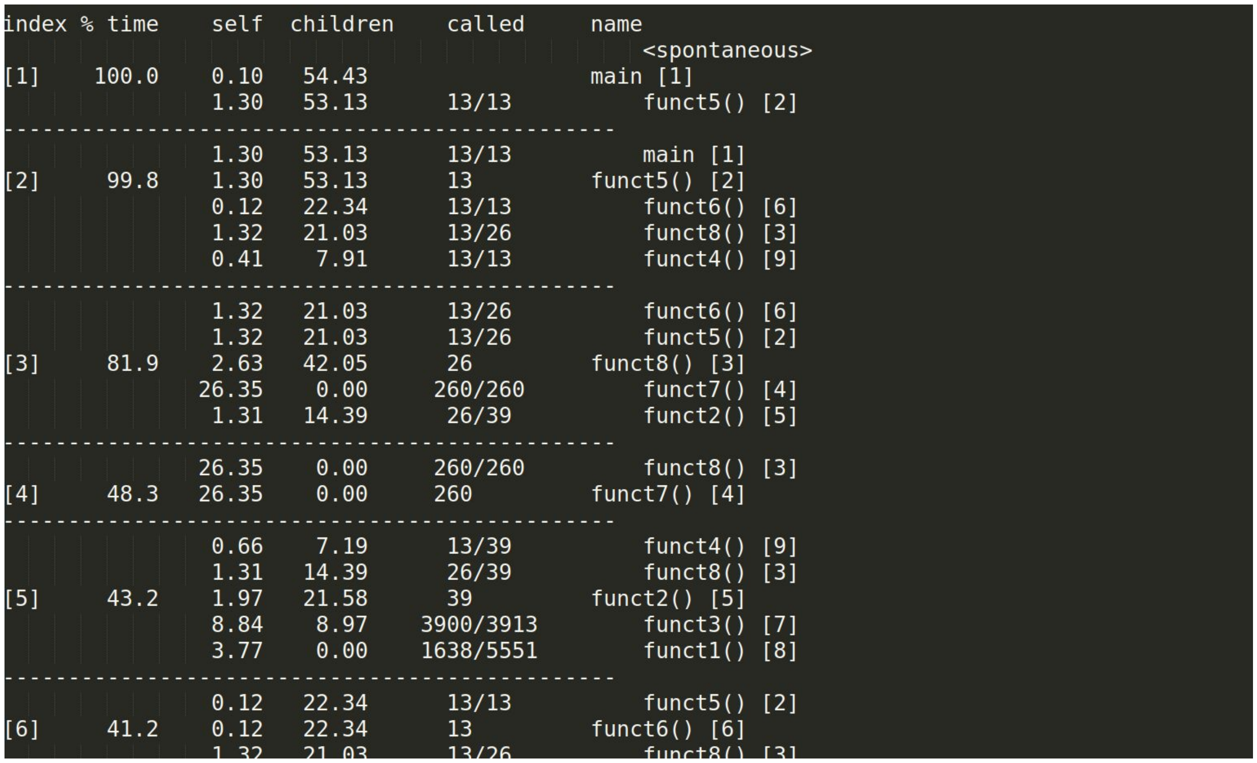
```
-----------------------------------------------
                   26.35    0.00     260/260         funct8() [3]
[4]     48.3   26.35    0.00     260         funct7() [4]
-----------------------------------------------
```

    There are no functions listed below funct7, which means that the current function did not call any other functions.
2. **Funct1** is called the most number of times : 5551 times.
    a. Funct2 invokes funct1 1638 times.
    b. Funct3 invokes funct1 3913 times.
3. Assuming that the COMPUTATION TIME asked in the question is exclusive, the computation time includes all the code that the function itself has executed, and does not include time taken for other functions that **funct5** invokes.

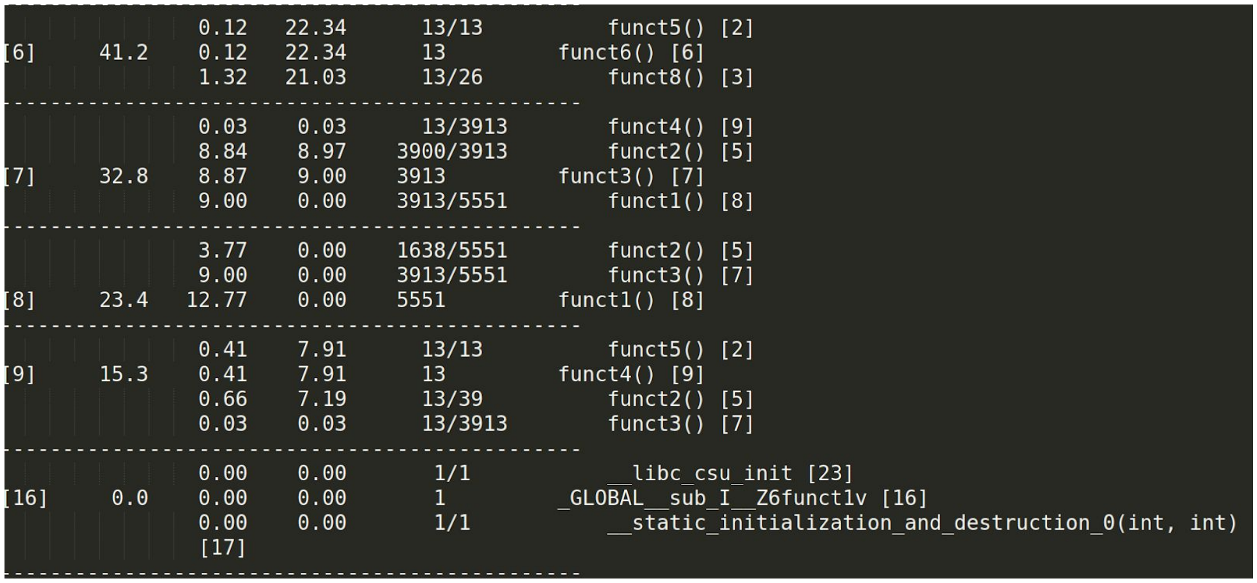


```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls   s/call   s/call  name    
 48.37     26.35    26.35      260     0.10     0.10  funct7()
 23.45     39.12    12.77     5551     0.00     0.00  funct1()
 16.28     47.99     8.87     3913     0.00     0.00  funct3()
  4.83     50.62     2.63       26     0.10     1.72  funct8()
  3.62     52.59     1.97       39     0.05     0.60  funct2()
  2.39     53.90     1.30       13     0.10     4.19  funct5()
  0.75     54.31     0.41       13     0.03     0.64  funct4()
  0.22     54.43     0.12       13     0.01     1.73  funct6()
  0.18     54.53     0.10                             main
  0.00     54.53     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z6funct1v
  0.00     54.53     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int,
  int)

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank
```

4.



```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.10   54.43                 main [1]
                1.30   53.13      13/13          funct5() [2]
-----------------------------------------------
                1.30   53.13      13/13          main [1]
[2]     99.8    1.30   53.13      13         funct5() [2]
                0.12   22.34      13/13          funct6() [6]
                1.32   21.03      13/26          funct8() [3]
                0.41    7.91      13/13          funct4() [9]
-----------------------------------------------
                1.32   21.03      13/26          funct6() [6]
                1.32   21.03      13/26          funct5() [2]
[3]     81.9    2.63   42.05      26         funct8() [3]
               26.35    0.00     260/260         funct7() [4]
                1.31   14.39      26/39          funct2() [5]
-----------------------------------------------
               26.35    0.00     260/260         funct8() [3]
[4]     48.3   26.35    0.00     260         funct7() [4]
-----------------------------------------------
                0.66    7.19      13/39          funct4() [9]
                1.31   14.39      26/39          funct8() [3]
[5]     43.2    1.97   21.58      39         funct2() [5]
                8.84    8.97    3900/3913        funct3() [7]
                3.77    0.00    1638/5551        funct1() [8]
-----------------------------------------------
                0.12   22.34      13/13          funct5() [2]
[6]     41.2    0.12   22.34      13         funct6() [6]
                1.32   21.03      13/26          funct8() [3]
```



```
-----------------------------------------------
                0.12   22.34      13/13          funct5() [2]
[6]     41.2    0.12   22.34      13         funct6() [6]
                1.32   21.03      13/26          funct8() [3]
-----------------------------------------------
                0.03    0.03      13/3913        funct4() [9]
                8.84    8.97    3900/3913        funct2() [5]
[7]     32.8    8.87    9.00    3913         funct3() [7]
                9.00    0.00    3913/5551        funct1() [8]
-----------------------------------------------
                3.77    0.00    1638/5551        funct2() [5]
                9.00    0.00    3913/5551        funct3() [7]
[8]     23.4   12.77    0.00    5551         funct1() [8]
-----------------------------------------------
                0.41    7.91      13/13          funct5() [2]
[9]     15.3    0.41    7.91      13         funct4() [9]
                0.66    7.19      13/39          funct2() [5]
                0.03    0.03      13/3913        funct3() [7]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [23]
[16]     0.0    0.00    0.00       1         _GLOBAL__sub_I__Z6funct1v [16]
                0.00    0.00       1/1           __static_initialization_and_destruction_0(int, int)
                [17]
-----------------------------------------------
```

## Question 2.

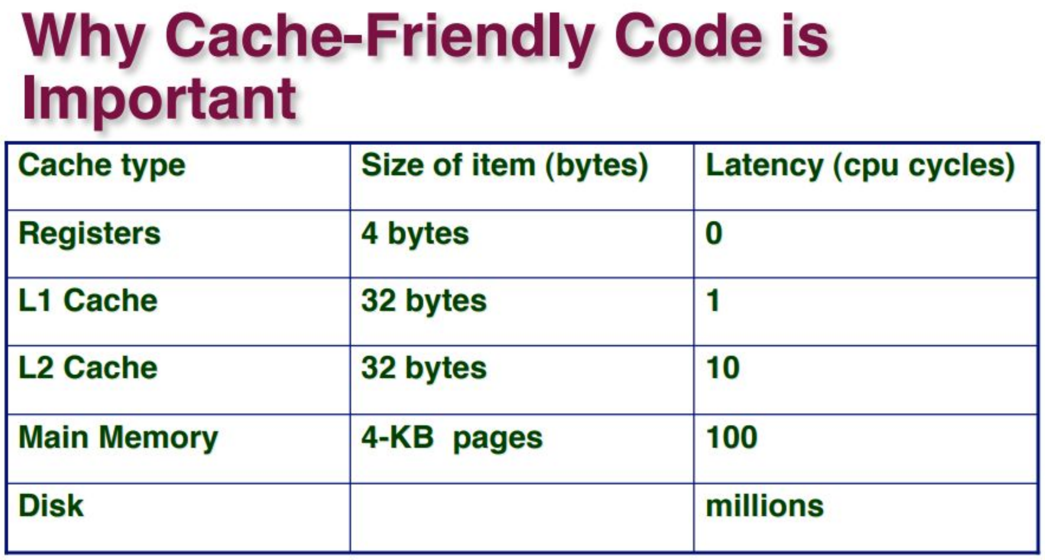
1. Execution of first program
   ./co1a  0.05s user 0.10s system 99% cpu 0.153 total
   ./co1a  0.06s user 0.09s system 99% cpu 0.145 total
   ./co1a  0.06s user 0.09s system 99% cpu 0.144 total
   ./co1a  0.06s user 0.09s system 98% cpu 0.151 total
   ./co1a  0.08s user 0.07s system 99% cpu 0.144 total

   Avg real time = 0.1474
2. Execution of second program

```
./co1b  0.05s user 0.03s system 99% cpu 0.072 total
./co1b  0.04s user 0.08s system 99% cpu 0.122 total
./co1b  0.04s user 0.07s system 99% cpu 0.105 total
./co1b  0.05s user 0.02s system 98% cpu 0.072 total
./co1b  0.07s user 0.04s system 99% cpu 0.111 total
```

Avg real time = 0.0966

According to the screenshot attached below :



**Why Cache-Friendly Code is Important**

| Cache type | Size of item (bytes) | Latency (cpu cycles) |
|---|---|---|
| Registers | 4 bytes | 0 |
| L1 Cache | 32 bytes | 1 |
| L2 Cache | 32 bytes | 10 |
| Main Memory | 4-KB pages | 100 |
| Disk | | millions |

[ Source :[http://web.cecs.pdx.edu/~jrb/cs201/lectures/cache.friendly.code.pdf](http://web.cecs.pdx.edu/~jrb/cs201/lectures/cache.friendly.code.pdf) ]
The above storage capacity of L1 cache varies from pc to pc, and is subject to change. Consider a 32 byte L1 Cache as an example.
The L1 cache has a size of 32 bytes.
The difference between the codes is this :

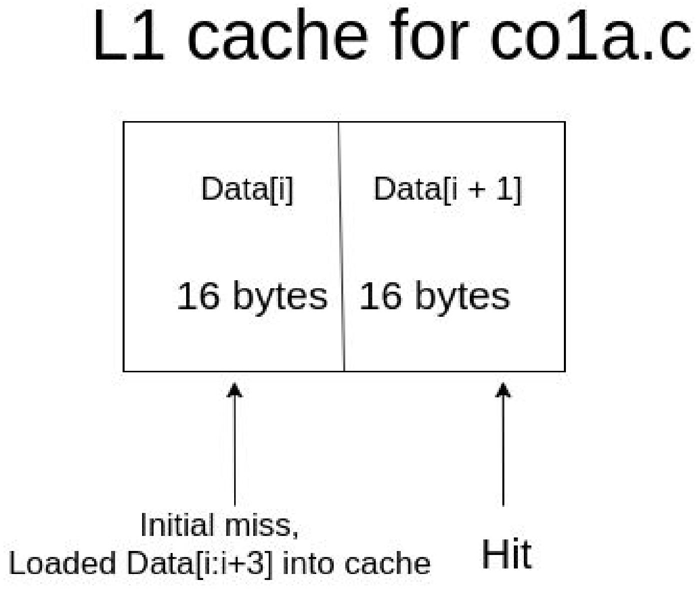
| | |
|---|---|
| struct DATA {<br>    int a;<br>    int b;<br>    int c;<br>    int d;<br>}; | struct DATA {<br>    int a;<br>    int b;<br>}; |
| Struct size: 16 bytes (4 * 4) | Struct size: 8 bytes (4*2) |

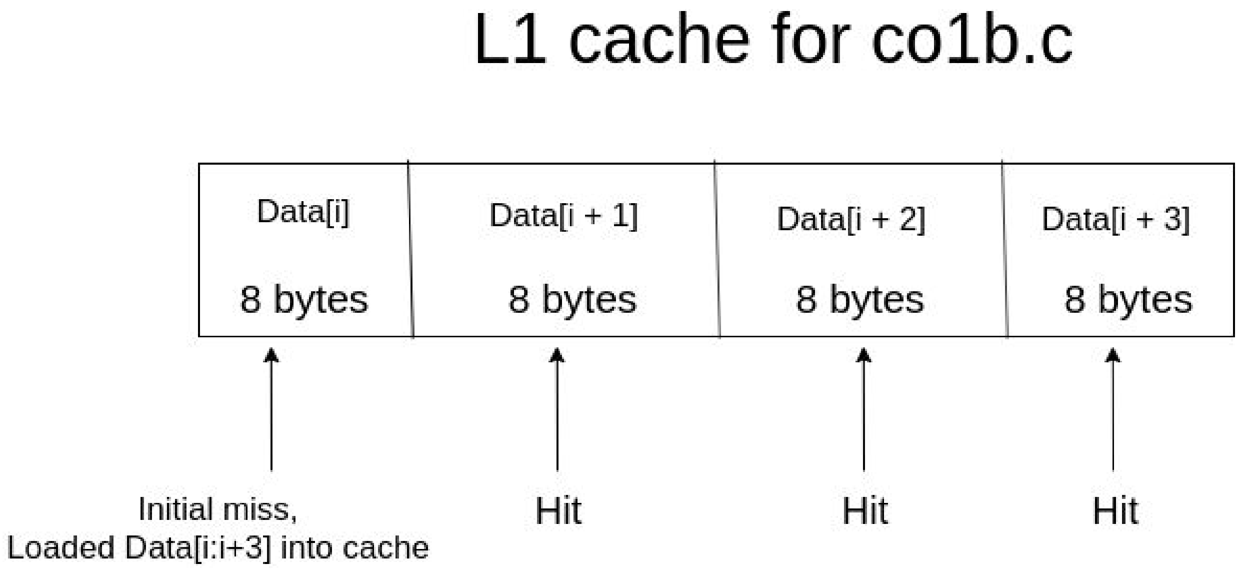
Considering only the L1 cache, for co1a.c, every time DATA[i] is accessed, only two structs are pulled into cache, the current one, and the one after ( DATA[i + 1] ). This is because the size of the struct is 16 bytes, and that of

L1 cache is 32 bytes, and 32 / 16 = 2. So there are (n/2) cache hits and (n/2) cache misses.

For co1b.c, every time DATA[i] is accessed, 4 structs are pulled into cache, DATA[i, i+1, i+2, & i+3], since the size of that struct is 8 bytes, 32 / 8 = 4. So the number of cache hits are ( 3 * n / 4 ) and the number of cache misses are (n / 4).

The above argument proves that co1b.c executes faster due to more number of cache hits.





Question 3.

The code co2a.c accesses data in a column format, which means it loads a new set of contiguous memory into cache every iteration, which causes a lot of cache misses.

Co2b.c instead accesses contiguous memory, hence making use of spatial locality.

```
./co2a  0.87s user 0.03s system 99% cpu 0.898 total
./co2b  0.21s user 0.04s system 99% cpu 0.259 total
```

## Question 4.

1. Matmul normal times :

    ```
    ./matmul  6.54s user 0.01s system 100% cpu 6.552 total
    ./matmul  6.03s user 0.02s system 100% cpu 6.039 total
    ./matmul  6.08s user 0.00s system 100% cpu 6.082 total
    ./matmul  6.69s user 0.01s system 100% cpu 6.697 total
    ./matmul  7.00s user 0.01s system 100% cpu 7.006 total
    ```

    Avg real time = 6.4752

    Matmul with transpose times :

    ```
    ./matmul_1  3.72s user 0.02s system 100% cpu 3.734 total
    ./matmul_1  3.62s user 0.02s system 100% cpu 3.638 total
    ./matmul_1  3.58s user 0.01s system 100% cpu 3.595 total
    ./matmul_1  3.58s user 0.03s system 99% cpu 3.624 total
    ./matmul_1  3.63s user 0.01s system 100% cpu 3.640 total
    ```

    Avg real time = 3.6462

2. Taking the transpose of a column matrix is an O($n^2$) operation, and hence while taking the transpose the number of cache misses is proportional to $n^2$. But matrix multiplication is an O($n^3$) operation which means that the number of cache misses is proportional to $n^3$.

    After taking the transpose of the matrix, mostly all subsequent memory accesses are contiguous, which results in very less cache misses during the matrix multiplication which is O($n^3$).

**Cache block matrix multiplication :**

1.

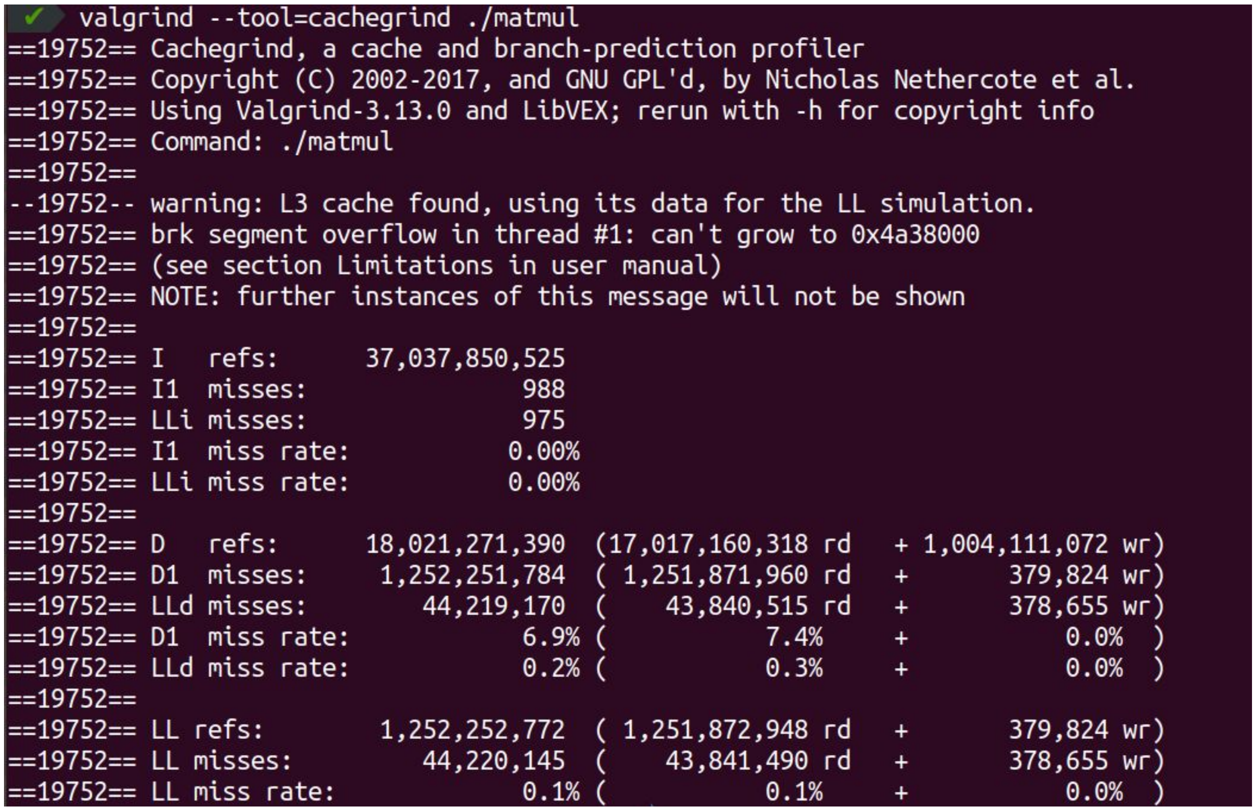
| | Normal_matrix_multiplication (time) | Transposed_matrix_multiplication | cache_block_matrix_multiplication |
|---|---|---|---|
| n = 1000 | 6.4752 | 3.6462 | 3.172 |

| | | | |
|---|---|---|---|
| n = 2000 | 1:28.49 | 29.189 | 26.125 |
| n = 4000 | 12:10.06 | 3:21.85 | 3:46.66 |

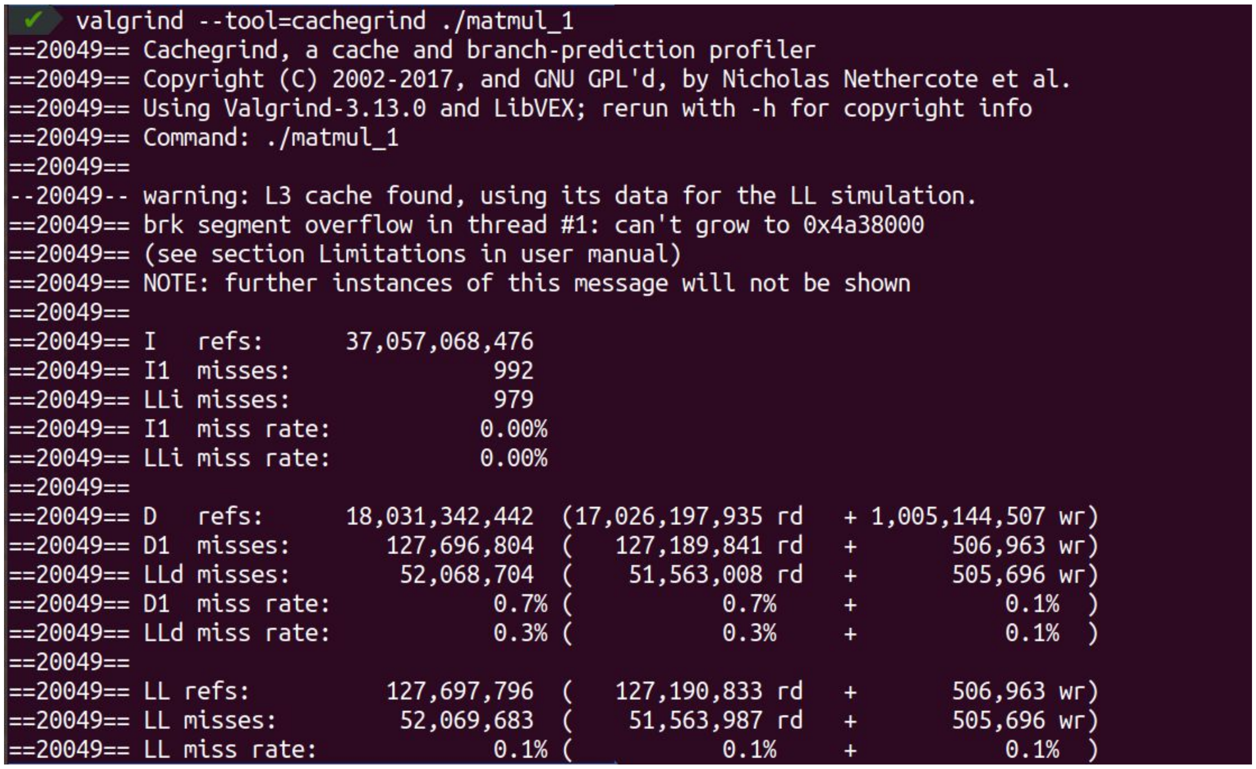
2. Yes, the second code is faster.
Define terms :
**f = number of arithmetic operations,**
**m = number of memory elements moved between fast and slow memory.**
**q = f/m = avg no of flops per slow memory access [Key to computational intensity and algorithm efficiency]**
**-------**
Naive Matrix Multiplication :
C = A * B
f = 2 * $n^3$ , m = $n^3$ + 3 * $n^2$,
[ multiply + add = 2 flops, for $n^3$ iterations ,
read each column of B n times = $n^3$,
read ith row of A once, since it is stored in cache = $n^2$,
read and write each element of C twice = 2*$n^2$,
Total = $n^3$ + 3 * $n^2$]
q = (2*$n^3$) / ($n^3$ + 3 * $n^2$) ~ 2 for large n.

---------
Blocked matrix multiplication :
If N is the size of smaller block,
f = 2 * $n^3$,
m = N * $n^2$ + $N^2$ + 2 * $n^2$,
q = f / m = n / N = block size b [for large N]
If b > 2, then blocked matrix multiplication has more avg no of flops per slow memory access, and hence performs faster.

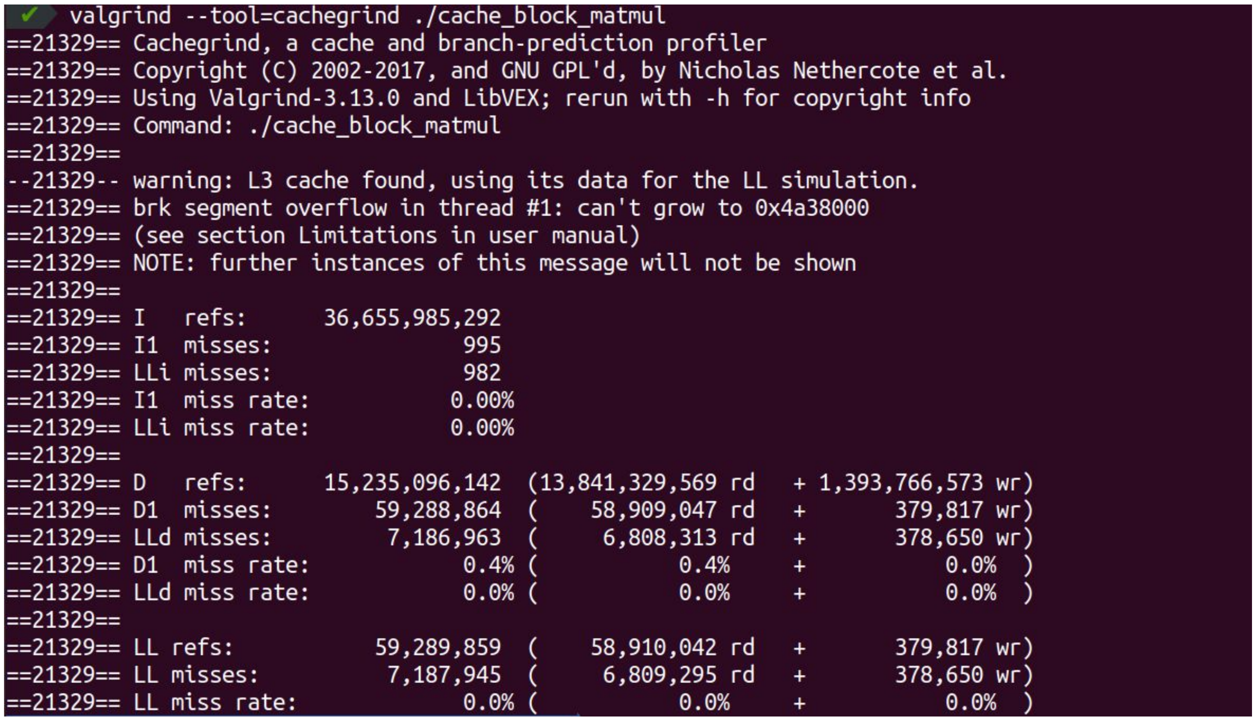
3.



```
valgrind --tool=cachegrind ./matmul
==19752== Cachegrind, a cache and branch-prediction profiler
==19752== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==19752== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==19752== Command: ./matmul
==19752==
--19752-- warning: L3 cache found, using its data for the LL simulation.
==19752== brk segment overflow in thread #1: can't grow to 0x4a38000
==19752== (see section Limitations in user manual)
==19752== NOTE: further instances of this message will not be shown
==19752==
==19752== I   refs:      37,037,850,525
==19752== I1  misses:               988
==19752== LLi misses:               975
==19752== I1  miss rate:           0.00%
==19752== LLi miss rate:           0.00%
==19752==
==19752== D   refs:      18,021,271,390  (17,017,160,318 rd   + 1,004,111,072 wr)
==19752== D1  misses:     1,252,251,784  ( 1,251,871,960 rd   +       379,824 wr)
==19752== LLd misses:        44,219,170  (    43,840,515 rd   +       378,655 wr)
==19752== D1  miss rate:            6.9% (           7.4%     +           0.0%  )
==19752== LLd miss rate:            0.2% (           0.3%     +           0.0%  )
==19752==
==19752== LL refs:        1,252,252,772  ( 1,251,872,948 rd   +       379,824 wr)
==19752== LL misses:         44,220,145  (    43,841,490 rd   +       378,655 wr)
==19752== LL miss rate:             0.1% (           0.1%     +           0.0%  )
```



```
valgrind --tool=cachegrind ./matmul_1
==20049== Cachegrind, a cache and branch-prediction profiler
==20049== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==20049== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==20049== Command: ./matmul_1
==20049==
--20049-- warning: L3 cache found, using its data for the LL simulation.
==20049== brk segment overflow in thread #1: can't grow to 0x4a38000
==20049== (see section Limitations in user manual)
==20049== NOTE: further instances of this message will not be shown
==20049==
==20049== I   refs:      37,057,068,476
==20049== I1  misses:               992
==20049== LLi misses:               979
==20049== I1  miss rate:           0.00%
==20049== LLi miss rate:           0.00%
==20049==
==20049== D   refs:      18,031,342,442  (17,026,197,935 rd   + 1,005,144,507 wr)
==20049== D1  misses:       127,696,804  (   127,189,841 rd   +       506,963 wr)
==20049== LLd misses:        52,068,704  (    51,563,008 rd   +       505,696 wr)
==20049== D1  miss rate:            0.7% (           0.7%     +           0.1%  )
==20049== LLd miss rate:            0.3% (           0.3%     +           0.1%  )
==20049==
==20049== LL refs:          127,697,796  (   127,190,833 rd   +       506,963 wr)
==20049== LL misses:         52,069,683  (    51,563,987 rd   +       505,696 wr)
==20049== LL miss rate:             0.1% (           0.1%     +           0.1%  )
```



```
✔  valgrind --tool=cachegrind ./cache_block_matmul
==21329== Cachegrind, a cache and branch-prediction profiler
==21329== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==21329== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==21329== Command: ./cache_block_matmul
==21329==
--21329-- warning: L3 cache found, using its data for the LL simulation.
==21329== brk segment overflow in thread #1: can't grow to 0x4a38000
==21329== (see section Limitations in user manual)
==21329== NOTE: further instances of this message will not be shown
==21329==
==21329== I   refs:      36,655,985,292
==21329== I1  misses:               995
==21329== LLi misses:               982
==21329== I1  miss rate:           0.00%
==21329== LLi miss rate:           0.00%
==21329==
==21329== D   refs:      15,235,096,142  (13,841,329,569 rd   + 1,393,766,573 wr)
==21329== D1  misses:        59,288,864  (    58,909,047 rd   +       379,817 wr)
==21329== LLd misses:         7,186,963  (     6,808,313 rd   +       378,650 wr)
==21329== D1  miss rate:            0.4% (           0.4%     +           0.0%  )
==21329== LLd miss rate:            0.0% (           0.0%     +           0.0%  )
==21329==
==21329== LL refs:           59,289,859  (    58,910,042 rd   +       379,817 wr)
==21329== LL misses:          7,187,945  (     6,809,295 rd   +       378,650 wr)
==21329== LL miss rate:             0.0% (           0.0%     +           0.0%  )
```

matmul -> naive matrix multiplication

matmul_1 -> transposed matrix multiplication

cache_block_matmul -> block matrix multiplication.

As is evident from the profiling info, the level 1 data miss rate (D1 miss rate) is most for the naive matrix multiplication algorithm (6.9%) , while it is the least in case of tiled (or block) matrix multiplication. (0.4%)