Analyzing Cache Behaviour and System Performance

Objectives.

- Study and run a program that experimentally calculates the L1 data cache size
- Case Study #1. Analyze cache behaviour and performance using Timers and Valgrind tool
- Case Study #2. Analyze cache behaviour and performance using Timers and Valgrind tool
- To introduce the roofline model

Aim

The **aim** of this lab session is to understand the behaviour of cache memories and how memory hierarchy affects system performance. Developing software that utilizes memory hierarchy is of critical importance in High Performance Computing, whether we are using CPUs or GPUs.

Section 1 – Experimental procedure that calculates the L1 data cache size

In cache_benchmark.c, I have developed a program that stores an array into memory many times. In cache_benchmark() routine, the X[N] array is stored 'TIMES' times. I have included a 'weird' code in the beginning, which maps/allocates the running thread to core number zero (first CPU core only); modern multi-core processors have more than one CPU cores. You do not have to understand how this code works. So, the running process which consists of a single thread will run on one CPU core only. If we omit this piece of code, then, the Operating System (OS) will toggle our process amongst different CPU cores. The OS does this to reduce the overall heat dissipation of the CPU chip.

Task1: Compile and run the program cache_benchmark.c for N=1000,2000,4000,8000,16000 and 32000. Measure the execution time for each case. Draw a graph, where execution time is the y-axis and N is the x-axis. To compile this program you will need extra gcc options, which they are given in the first line of cache_benchmark.c. **Make sure the execution time lasts a few seconds, otherwise the execution time measured is not accurate**. This is because other processes run too; if the execution time of this process is much larger than the others', then the value measured is more accurate. To this end, you must appropriately specify the 'TIMES' value.

You will realize that the execution time is not linear to the input size. Why?

Answer: As long as X[ ] is smaller than L1 data cache size, then it is stored 'TIMES' times to L1 data cache memory, which is very fast. If X[ ] no longer fits in L1 data cache, then it is written 'TIMES' times to L2 cache, which is slower.

Section 2 – Analysing Cache Behaviour and Performance using Valgrind

Case study #1. Array Initialization

Cache Analysis using Cachegrind tool

Task1: Download '*init_array.c*' file. This program initializes a 2-d array in a column wise manner (N=1000). We have discussed in the lecture that accessing an array in column-wise order is not efficient. The following example will verify the theory. Compile it and use Valgrind Cachegrind to extract the number of L1 data cache accesses and misses. Then, apply loop interchange and apply the above process again. Compare the results. I have included specific instructions about how to do that hereafter.

Compile the program using:

gcc init_array.c -o p -O2 -g

Use valgrind to simulate the cache by typing:

valgrind --tool=cachegrind ./p

Use the annotate command to get a more detailed picture about L1 data cache accesses and misses by typing:

cg_annotate cachegrind.out.24486

or cg_annotate cachegrind.out.24486 > column.txt

Then, apply loop interchange and repeat the above steps.

```
Activities  Text Editor ▾                         Thu 09:02
Open ▾                                            column.txt
                                      ~/Desktop/comp3001/my/Labs/cache                Save
*TXT_File.txt ×        init_arrays.c ×        row.txt ×        column.txt ×
30
31 --------------------------------------------------------------------------------
32 -- Auto-annotated source: /home/user01/Desktop/comp3001/my/Labs/cache/init_arrays.c
33 --------------------------------------------------------------------------------
34 Ir                   I1mr      ILmr      Dr        D1mr      DLmr      Dw                  D1mw                DLmw
35
36 -- line 4 ----------------------------------------
37          .         .         .         .         .         .                  .                  .                  .        #define N 1000  //arrays input size
38          .         .         .         .         .         .                  .                  .                  .
39          .         .         .         .         .         .                  .                  .                  .        //In C, all the routines must be
 declared
40          .         .         .         .         .         .                  .                  .                  .        void initialize();
41          .         .         .         .         .         .                  .                  .                  .
42          .         .         .         .         .         .                  .                  .                  .        int A[N][N];
43          .         .         .         .         .         .                  .                  .                  .
44          .         .         .         .         .         .                  .                  .                  .
45      2,001 ( 0.04%) 0         0         0         0         0                  0                  0                  0        int main( ) {
46          .         .         .         .         .         .                  .                  .                  .
47          .         .         .         .         .         .                  .                  .                  .        initialize();
48          .         .         .         .         .         .                  .                  .                  .
49          .         .         .         .         .         .                  .                  .                  .
50          .         .         .         .         .         .                  .                  .                  .        return 0; //normally, by returning
 zero, we mean that the program ended successfully.
51          2 ( 0.00%) 0         0         1 ( 0.00%) 1 ( 0.07%) 0                  0                  0                  0        }
52          .         .         .         .         .         .                  .                  .                  .
53          .         .         .         .         .         .                  .                  .                  .
54          .         .         .         .         .         .                  .                  .                  .
55          .         .         .         .         .         .                  .                  .                  .        void initialize(){
56          .         .         .         .         .         .                  .                  .                  .
57          .         .         .         .         .         .                  .                  .                  .        int i,j;
58          .         .         .         .         .         .                  .                  .                  .
59      6,002 ( 0.12%) 1 ( 0.12%) 1 ( 0.13%) 0         0         0                  0                  0                  0        for (i=0;i<N;i++)
60 2,000,000 (39.00%) 0         0         0         0         0                  0                  0                  0         for (j=0;j<N;j++){
61 3,000,000 (58.51%) 0         0         0         0         0          1,000,000 (99.05%) 1,000,000 (99.95%) 62,500 (99.18%)   A[j][i]=i+j;
62          .         .         .         .         .         .                  .                  .                  .
63          .         .         .         .         .         .                  .                  .                  .        }
64          .         .         .         .         .         .                  .                  .                  .
65          .         .         .         .         .         .                  .                  .                  .        }
66          .         .         .         .         .         .                  .                  .                  .
67          .         .         .         .         .         .                  .                  .                  .
68          .         .         .         .         .         .                  .                  .                  .
69          .         .         .         .         .         .                  .                  .                  .
70 -- line 36 ----------------------------------------
71
72 --------------------------------------------------------------------------------
73 The following files chosen for auto-annotation could not be found:
74 --------------------------------------------------------------------------------
                                                     Plain Text ▾  Tab Width: 8 ▾   Ln 1, Col 1  ▾  INS
```

Fig.1 Valgrind output for init_arrays.c (column-wise)

```
Activities  Text Editor ▾                         Thu 09:02
Open ▾                                            row.txt
                                      ~/Desktop/comp3001/my/Labs/cache                Save
*TXT_File.txt ×        init_arrays.c ×        row.txt ×        column.txt ×
32 -- Auto-annotated source: /home/user01/Desktop/comp3001/my/Labs/cache/init_arrays.c
33 --------------------------------------------------------------------------------
34 Ir                   I1mr      ILmr      Dr        D1mr      DLmr      Dw                  D1mw           DLmw
35
36 -- line 4 ----------------------------------------
37          .         .         .         .         .         .                  .              .              .        #define N 1000  //arrays input size
38          .         .         .         .         .         .                  .              .              .
39          .         .         .         .         .         .                  .              .              .        //In C, all the routines must be
 declared
40          .         .         .         .         .         .                  .              .              .        void initialize();
41          .         .         .         .         .         .                  .              .              .
42          .         .         .         .         .         .                  .              .              .        int A[N][N];
43          .         .         .         .         .         .                  .              .              .
44          .         .         .         .         .         .                  .              .              .
45      2,001 ( 0.04%) 0         0         0         0         0                  0              0              0        int main( ) {
46          .         .         .         .         .         .                  .              .              .
47          .         .         .         .         .         .                  .              .              .        initialize();
48          .         .         .         .         .         .                  .              .              .
49          .         .         .         .         .         .                  .              .              .
50          .         .         .         .         .         .                  .              .              .        return 0; //normally, by returning
 zero, we mean that the program ended successfully.
51          2 ( 0.00%) 0         0         1 ( 0.00%) 1 ( 0.07%) 0                  0              0              0        }
52          .         .         .         .         .         .                  .              .              .
53          .         .         .         .         .         .                  .              .              .
54          .         .         .         .         .         .                  .              .              .
55          .         .         .         .         .         .                  .              .              .        void initialize(){
56          .         .         .         .         .         .                  .              .              .
57          .         .         .         .         .         .                  .              .              .        int i,j;
58          .         .         .         .         .         .                  .              .              .
59      4,002 ( 0.08%) 0         0         0         0         0                  0              0              0        for (i=0;i<N;i++)
60 2,000,000 (39.02%) 0         0         0         0         0                  0              0              0         for (j=0;j<N;j++){
61 3,000,000 (58.53%) 0         0         0         0         0          1,000,000 (99.05%) 62,500 (99.13%) 62,500 (99.18%)   A[i][j]=i+j;
62          .         .         .         .         .         .                  .              .              .
63          .         .         .         .         .         .                  .              .              .        }
64          .         .         .         .         .         .                  .              .              .
65          .         .         .         .         .         .                  .              .              .        }
66          .         .         .         .         .         .                  .              .              .
67          .         .         .         .         .         .                  .              .              .
68          .         .         .         .         .         .                  .              .              .
69          .         .         .         .         .         .                  .              .              .
70 -- line 36 ----------------------------------------
71
72 --------------------------------------------------------------------------------
73 The following files chosen for auto-annotation could not be found:
74 --------------------------------------------------------------------------------
75   /build/glibc-OTsEL5/glibc-2.27/elf/../sysdeps/x86_64/dl-machine.h
76   /build/glibc-OTsEL5/glibc-2.27/elf/dl-lookup.c
                                                     Plain Text ▾  Tab Width: 8 ▾   Ln 1, Col 1  ▾  INS
```

Fig.2 Valgrind output for init_arrays.c (row-wise)

The Cachegrind output is shown in Fig.1 and Fig2. The memory statistics look very similar, however, if you look closer, there is a big difference in the 'D1mw' column (L1 data cache write misses). Although, both programs write 1,000,000 elements into L1 data cache (Dw column), in the first case they are written in column-wise order, while in the second case they are written row-wise.

In the column-wise case, to write just A[0][0] into memory, an entire L1 data cache line is loaded to dL1 memory (it will contain 16 elements of A, i.e., A[0][0:15]) and then just A[0][0] is stored into it. Then, to write A[1][0], another cache line is loaded (16 elements) and just A[1][0] is stored etc. Thus, there is one dL1 miss for every store, as A[][] is stored in DDR, not dL1. On the contrary, in the row-wise case, a dL1 miss is followed by 15 subsequent dL1 hits. This is evidenced by the 'D1mw' column (L1 data miss writes). In the column-wise case, there are 1,000,000 dL1 write misses, while in the row-wise case there are just 62,500 dL1 write misses. There are 16 times less misses as each dL1 cache line contains 16 elements. Keep in mind that 62,500x16=1,000,000.

Comparison using timers

Cache misses introduce significant performance penalties, and this is reflected on the program's execution time.

Task3: Run the two different versions of the '*init_arrays.c*' program discussed above and compare their execution times. You will find out that initializing an array of integers in a row-wise manner is about 16 times faster than the column-wise. Recall that to get an accurate execution time value, **the execution time must be at least a few seconds, otherwise is not accurate.**

Generating the assembly

You can generate the assembly of the above two program versions and compare them. You can generate the assembly code by using the following command

gcc -S init_arrays.c -o assembly.s -O2

Case study #2. Matrix-Matrix Multiplication

Task1. Download 2mmm.c file. The routine we are interested in is *void mmm()*. Study the program and make sure you understand what it does. Use Cachegrind and pg_annotate to simulate the behaviour of dL1. Measure the dL1 accesses and misses. You should see something like Fig.3.

```
Ir                I1mr       ILmr       Dr                 D1mr               DLmr    Dw                 D1mw               DLmw
                                                                                                                                    struct timespec start, end; //timers
                                                                                                                                    uint64_t diff;

                                                                                                                                    initialize();

                                                                                                                                    /* measure monotonic time */
          3 ( 0.00%) 1 ( 0.10%) 1 ( 0.10%)          0                  0      0          1 ( 0.00%)        1 ( 0.00%)      0       clock_gettime(CLOCK_MONOTONIC, &start);       /* mark star
t time */
          2 ( 0.00%) 0          0                   0                  0      0          1 ( 0.00%)        0               0       mmm();
          3 ( 0.00%) 0          0                   0                  0      0          1 ( 0.00%)        0               0       clock_gettime(CLOCK_MONOTONIC, &end); /* mark the end time
 */
          5 ( 0.00%) 1 ( 0.10%) 1 ( 0.10%)          4 ( 0.00%)         0      0          0                 0               0       diff = BILLION * (end.tv_sec - start.tv_sec) + end.tv_nsec
 - start.tv_nsec;
                                                                                                                                    printf("elapsed time = %llu nanoseconds\n", (long long uns
igned int) diff);
          4 ( 0.00%) 0          0                   0                  0      0          0                 0               0       printf("elapsed time = %llu mseconds\n", (long long unsign
ed int) diff/1000000);
                                                                                                                                    return 0; //normally, by returning zero, we mean that the
program ended successfully.
          7 ( 0.00%) 0          0                   4 ( 0.00%)         1 ( 0.00%) 0      0                 0               0       }


                                                                                                                                    void initialize(){

                                                                                                                                    int i,j;

      3,501 ( 0.00%) 0          0                   0                  0      0          0                 0               0       for (i=0;i<N;i++)
    500,000 ( 0.03%) 0          0                   0                  0      0          0                 0               0        for (j=0;j<N;j++){
    500,000 ( 0.03%) 0          0                   0                  0      0    250,000 (16.55%)  15,625 ( 2.91%) 15,625 (24.77%)   A[i][j]=(i+j)%1000;
  1,000,000 ( 0.06%) 0          0                   0                  0      0    250,000 (16.55%)  15,625 ( 2.91%) 15,625 (24.77%)   B[i][j]=(i-j)%1000;
    250,000 ( 0.01%) 0          0                   0                  0      0    250,000 (16.55%)  15,625 ( 2.91%) 15,625 (24.77%)   C[i][j]=0;
    250,000 ( 0.01%) 1 ( 0.10%) 1 ( 0.10%)          0                  0      0    250,000 (16.55%)  15,625 ( 2.91%) 15,625 (24.77%)   E[i][j]=0;

                                                                                                                                    }

                                                                                                                                    }
  2,002,509 ( 0.11%) 2 ( 0.19%) 2 ( 0.19%)    500,000 ( 0.10%)    31,250 ( 0.03%) 0      3 ( 0.00%)        0               0       void mmm(){

                                                                                                                                    int i,j,k;

      1,502 ( 0.00%) 1 ( 0.10%) 1 ( 0.10%)          0                  0      0          0                 0               0       for (i=0;i<N;i++)
    501,000 ( 0.03%) 0          0                   0                  0      0          0                 0               0        for (j=0;j<N;j++)
250,750,000 (14.27%) 0          0                   0                  0      0    250,000 (16.55%) 236,752 (44.12%)     0          for (k=0;k<N;k++)
625,000,000 (35.57%) 1 ( 0.10%) 1 ( 0.10%) 250,000,000 (49.95%) 58,263,318 (49.99%) 0      0                 0               0        C[i][j]+=A[i][k]*B[k][j];


      1,000 ( 0.00%) 0          0                   0                  0      0          0                 0               0       for (i=0;i<N;i++)
    501,000 ( 0.03%) 0          0                   0                  0      0          0                 0               0        for (j=0;j<N;j++)
250,750,000 (14.27%) 1 ( 0.10%) 1 ( 0.10%)          0                  0      0    250,000 (16.55%) 236,752 (44.12%)     0          for (k=0;k<N;k++)
625,000,000 (35.57%) 0          0          250,000,000 (49.95%) 58,263,407 (49.99%) 0      0                 0               0        E[i][j]+=C[i][k]*B[k][j];

          4 ( 0.00%) 0          0                   4 ( 0.00%)         2 ( 0.00%) 0      0                 0               0       }
```

Fig.3. Valgrind Output for 2mmm.c file

The ‘D1mw’ column (dL1 write misses) values are very close to the Dw ones. This means that almost every write is a dL1 write miss; in simple words, the dL1 gets full and therefore the cache line holding the C values cannot remain in dL1. Keep in mind that the elements of C[][] are written into memory in row-wise order. Thus, to write C[0][0] to dL1, 16 elements of C are loaded into dL1 C[0][0:15] (each cache line consists of 64 bytes, i.e., 16 integer elements). However, according to the cache statistics, when C[0][1] is written to memory, it is a miss not a hit, leading us to the conclusion that the C[0][0:15] line cannot remain in the cache. This is true as to compute a single C[][] value, an entire row of A and an entire column of B are loaded. The size of a row of A is 2000 bytes, but the size of one column of B is much more. The column elements are not written in consecutive memory locations and thus for each column element, 16 elements are loaded (an entire cache line). So, 500 cache lines will be loaded for B[][], but these cache lines are not located in consecutive memory locations, leading to a large number of cache misses.

As far as the number of reads are concerned (Dr column), three values are shown (500.000 , 250.000.000 , 250.000.000). The second value refers to the number of loads because of A[][] and B[][] (first loop kernel in mmm routine), while the third values refers to the number of loads because of C[][] and B[][] (second loop kernel in mmm routine). Each array is under three loops and thus there are 500*500*500 load operations for each array (125,000,000). However, C[][] and E[][] arrays are loaded too (do not forget that C[i][j]+= means C[i][j]=C[i][j]+). Where are these loads shown? For a reason they are shown separately a little before in the code (500.000). But why there are 250.000 loads for each array and not 125.000.000 as for the other arrays? Apparently, the compiler has applied scalar replacement transformation (code is shown below) and thus C[][] and E[][] arrays are under two loops now, not three (the code is compiled with -O2 option and thus the compiler applied this optimization). To test this, we can manually apply scalar replacement, run valgrind again and test the results. The new code will be

for (i=0;i<N;i++)

```
for (j=0;j<N;j++){
c=C[i][j];
 for (k=0;k<N;k++) {
c+=A[i][k]*B[k][j];
}
C[i][j]=c;
}

for (i=0;i<N;i++)
 for (j=0;j<N;j++){
e=E[i][j];
 for (k=0;k<N;k++){
 e+=C[i][k]*B[k][j];
}
E[i][j]=e;
}
```

Thus, the binary is closer to the code above rather than the code we developed. As you can observe from Fig.3, in mmm() routine there are 500.000 writes in total (see Dw column). This value was expected as two 2d arrays of size 500x500 are stored into memory.

The Roofline Model

The roofline model [3] provides an easy way to get performance bounds for compute-bound and memory-bound loop kernels. It allows us to know how far the achieved performance is from the optimum. It is based on the concept of computational intensity, sometimes also called arithmetic or operational Intensity. The arithmetic intensity is given by the following formula: '*FP.arithmetical.instructions / number.of.bytes.loaded.stored*'. This model has several limitations [3], e.g., does not consider all features of modern processors and ignores integer computations.

Roofline Model for MMM algorithm: MMM has N^3 iterations and each iteration contains 4 Floating Point (FP) L/S operations and 2 FP arithmetical operations. So, the arithmetical intensity of MMM (the ratio between FP arithmetical operations and number of bytes loaded/stored), when the arrays are of type 'float', is 2/(4*4bytes)=1/8.

MMM loop kernel contains integer arithmetical instructions too (we could generate the assembly and check them out by typing *gcc source.c -S assembly.s -O2*). The integer arithmetical instructions are responsible for a) computing the L/S memory addresses (e.g., ...=A[i][j] will be broken down to multiple assembly instructions), b) controlling the iterators (increment i, compare i to N, branch back); however, these integer operations take 1 CPU cycle each and they are performed in parallel with the FP ones. So, for this loop kernel we could assume that performance is not affected by the integer operations. Furthermore, the roofline model does not consider integer operations, and this is a serious limitation of this model.

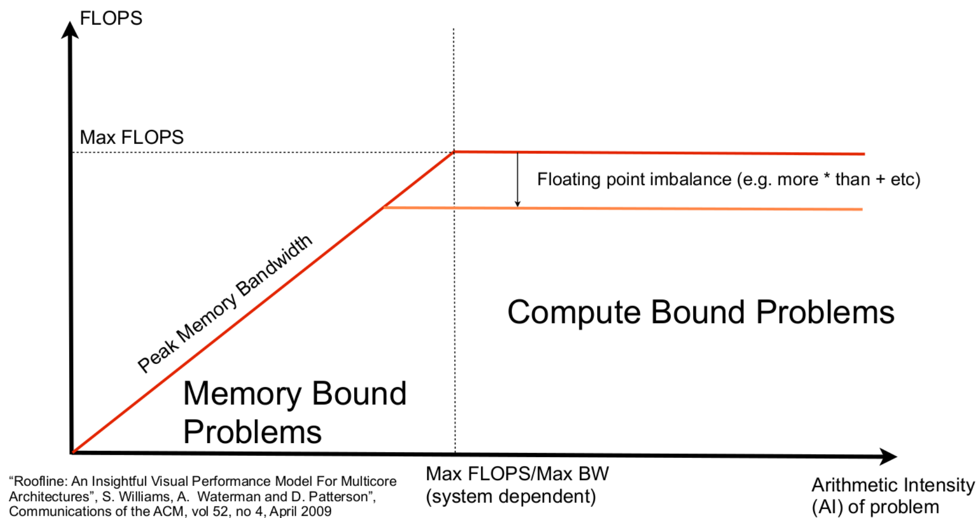

Fig.4. The Roofline Model [3]

Algorithms that have a low arithmetical intensity are **memory-bound,** while algorithms that have a high arithmetical intensity are **compute-bound**. Memory-bound means that their performance is bounded on the memory latency and bandwidth values; in simple words, no matter how high the CPU frequency is, or no matter how many cores the CPU supports, performance depends on how fast the data are loaded/stored from/to memory.

Attainable FLOPS = min(Peak Floating Point Performance, Peak Memory Bandwidth x Arithmetical Intensity)

The peak FP performance is the maximum we can get, and it refers to compute-bound loop kernels with a perfect balance between simple and complex FP operations. The latter is an advanced topic and it is not further explained here; for those who want to go deeper and learn more about it, ask the module leader. The peak FP performance is CPU dependent.

Tip. each instruction has a latency and a throughput value, where the latter one is always larger or equal to the first; thus, to achieve the optimum performance of a bunch of instructions, e.g., multiply instructions, multiple multiply instructions must be feed the instruction pipeline one after another.

The peak memory bandwidth depends on the DDR memory and memory controller hardware characteristics. Furthermore, if the data fit and remain in the cache, the peak memory bandwidth is the cache bandwidth. As a reminder, a DDR memory access takes about 100-200 CPU cycles, an L3 memory accesses about 40-70 CPU cycles, an L2 memory access 6-14 CPU cycles, while an L1 memory access takes about 1-4 CPU cycles.

So, if the peak memory bandwidth is 21GBytes/sec, then the maximum MMM performance will be 21GB/sec x (1/8)=2.65gigaflops. If the arrays fit in the precious cache memories, then the memory bandwidth is higher and thus performance is increased.

How can we improve the performance of memory-bound algorithms? The main strategies are as follows. Reducing the number of memory accesses through the whole memory hierarchy; this relates to

reducing the number of cache-misses too. Another strategy is to use software prefetching. The above can be achieved by using code optimizations such as loop tiling, register blocking, array copying, loop merge/distribution etc. We will study those next week.

Further Reading:

1. The Valgrind Quick Start Guide, available at https://www.valgrind.org/docs/manual/quick-start.html#quick-start.intro
2. Cachegrind: a cache and branch-prediction, available at https://valgrind.org/docs/manual/cg-manual.html
3. Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 4 (April 2009), 65-76. DOI=10.1145/1498765.1498785, available at https://people.eecs.berkeley.edu/~kubitron/cs252/handouts/papers/RooflineVyNoYellow.pdf