CS 773: Assignment-1 Task-1

Approach

- Below is the code for this task:

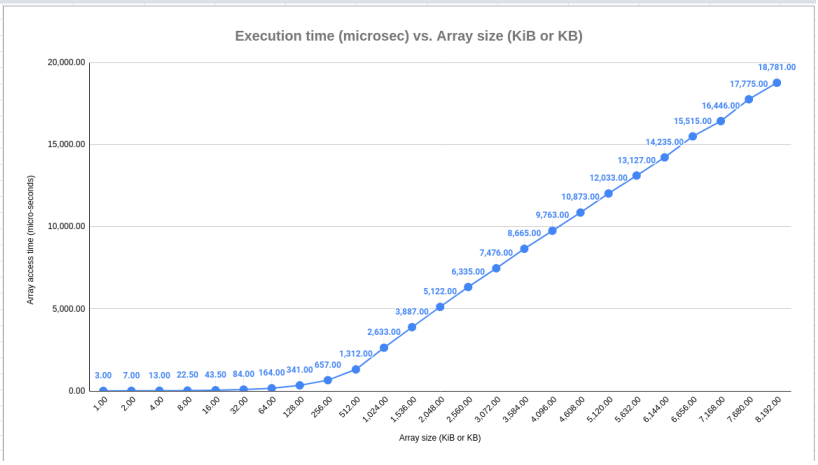
```
// CS773: Assignment-1 Task-1

#include <stdio.h>
#include <time.h>

#define KB_BYTES 1024       // 1KiB (Kibibyte) or 1KB = 1024 bytes

void mem_access()
{
    int arr_len = 8192 * KB_BYTES;
    char arr[arr_len];      // array of size in multiple of 1024 bytes (1 KiB
data)
    int k;

    for ( int i = 0; i < arr_len; i++ )
    {
        k = arr[i];
    }
}

void mem_thrash()
{
    int thrash_arr_len = 8 * KB_BYTES * KB_BYTES;
    char large_arr[thrash_arr_len]; // array of large size approx. equal to
L3 cache (8 MiB)
    int j;

    for ( int i = 0; i < thrash_arr_len; i++ )
    {
        j = large_arr[i];
    }
}

int main()
{
    // execute the function mem_thrash to clear all caches
    mem_thrash();

    // clock object for timing
    clock_t start_time, end_time;

    // note the start time before executing memory access
    start_time = clock();

    // execute the function mem_access
    mem_access();
```

```
    // note the end time after executing memory access
    end_time = clock() - start_time;

    // compute the time taken in seconds
    long double exec_time = ((long double) end_time) / CLOCKS_PER_SEC;

    printf("mem_access took %.9Lf seconds to execute\n", exec_time);

    return 0;
}
```

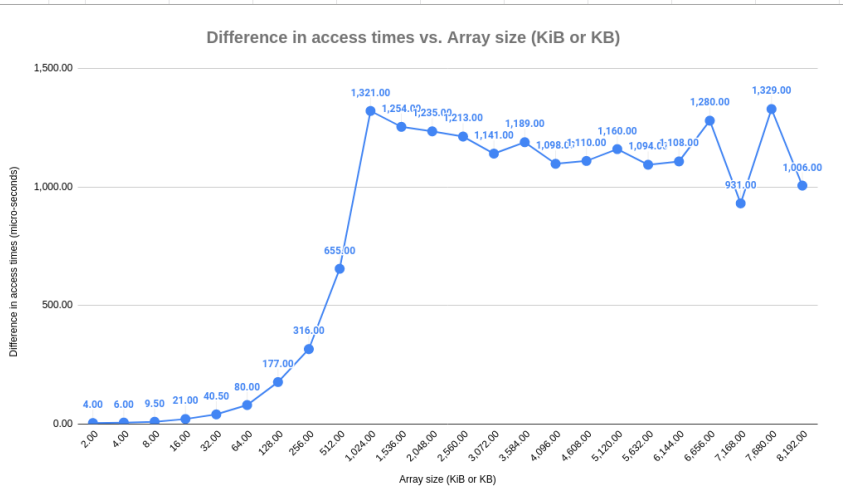
- Initially, the caches are cleared or thrashed by accessing large array that is greater than the **L3** (**8 MB** in my case) cache size in `mem_thrash` function. This will make the processor load values from memory.
- Once the caches are cleared, we access the array of variable sizes in `mem_access` function and time the execution of this function, thus it is similar to access time of array.
- The time in **micro-seconds** is reported on the output for a particular array size.
- For varying the array size, one needs to update the multiplier in `mem_access` for `arr_len` variable. For array size varying from **1 KB** to **8 MB**, the multiplier ranges from **1** to **8192**.
- It was observed that by running the executable repeatedly for the same array size, the access time was varying with some range (range was different for different array sizes).
- Hence, a short Python script was written to execute the C code ***1000 times*** with a delay of ***100 ms***. The ***average access time*** was computed and output on the Terminal. This access time is plotted against the array size in figure below.

> **Note:**
>
> - It was asked to disable the pre-fetcher using **msr** tool and find the register and its value from the ISA manual for the processor in our laptop.
> - In my case, its **Intel i5-9300H** CPU. The ISA manual for Intel processor described to disable few pre-fetcher bits in `IA32_MISC_ENABLE` register. Due to some unknown reasons, I am unable to perform write operation on the pre-fetcher bits of the register. I have posted this query at this link: https://piazza.com/class/l5kjeeswcso3v6/post/8_f3.
> - I continued with the Task-1 without disabling the pre-fetcher on the system.

	Array size (KiB or KB)	Execution time (microsec)	diff
1 KB	1.00	3.00	
2 KB	2.00	7.00	4.00
4 KB	4.00	13.00	6.00
8 KB	8.00	22.50	9.50
16 KB	16.00	43.50	21.00
32 KB	32.00	84.00	40.50
64 KB	64.00	164.00	80.00
128 KB	128.00	341.00	177.00
256 KB	256.00	657.00	316.00
512 KB	512.00	1,312.00	655.00
1 MB	1,024.00	2,633.00	1,321.00
1.5 MB	1,536.00	3,887.00	1,254.00
2 MB	2,048.00	5,122.00	1,235.00
2.5 MB	2,560.00	6,335.00	1,213.00
3 MB	3,072.00	7,476.00	1,141.00
3.5 MB	3,584.00	8,665.00	1,189.00
4 MB	4,096.00	9,763.00	1,098.00
4.5 MB	4,608.00	10,873.00	1,110.00
5 MB	5,120.00	12,033.00	1,160.00
5.5 MB	5,632.00	13,127.00	1,094.00
6 MB	6,144.00	14,235.00	1,108.00
6.5 MB	6,656.00	15,515.00	1,280.00
7 MB	7,168.00	16,446.00	931.00
7.5 MB	7,680.00	17,775.00	1,329.00
8 MB	8,192.00	18,781.00	1,006.00



	Array size (KiB or KB)	Difference in access times
2 KB	2.00	4.00
4 KB	4.00	6.00
8 KB	8.00	9.50
16 KB	16.00	21.00
32 KB	32.00	40.50
64 KB	64.00	80.00
128 KB	128.00	177.00
256 KB	256.00	316.00
512 KB	512.00	655.00
1 MB	1,024.00	1,321.00
1.5 MB	1,536.00	1,254.00
2 MB	2,048.00	1,235.00
2.5 MB	2,560.00	1,213.00
3 MB	3,072.00	1,141.00
3.5 MB	3,584.00	1,189.00
4 MB	4,096.00	1,098.00
4.5 MB	4,608.00	1,110.00
5 MB	5,120.00	1,160.00
5.5 MB	5,632.00	1,094.00
6 MB	6,144.00	1,108.00
6.5 MB	6,656.00	1,280.00
7 MB	7,168.00	931.00
7.5 MB	7,680.00	1,329.00
8 MB	8,192.00	1,006.00



Findings

- With increasing array sizes, the instructions (LOAD, in this case) will increase and hence the access time of array elements also increases.
- Initially for smaller array sizes, the access time is few 100 micro-seconds. Access time then starts shooting up at the boundary of cache size of each level.
- This behaviour is arises since the array can't fit in the L1 cache, thus there will be L1 misses and the data will be fetched from the main memory (since caches of all levels were thrashed earlier).
- Fetching data from the main memory takes more number of cycles as compared to fetching from the caches.
- This is because the pre-fetcher speculates the pattern of accessing array and fetches data from higher level caches before the data is actually accessed.
- The plot depicting the difference in access times for adjacent array sizes is also shown.
- The output of `lscpu` command on the system:

```
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        8 MiB
```

- From the difference plot, the ***significant*** rises are observed at the below array sizes:
  - 1st rise
    - *177.00* at 128 KB => **Measured L1D cache size = 128 KB**
  - 2nd rise
    - 1321.00 at 1024 KB [1 MB] => **Measured L2 cache size = 1 MB**
  - 3rd rise
    - 1329.00 at 7680 KB [7.5 MB] => **Measured LLC cache size = 7.5 MB**

Thank you!