**Hardware & Software Setup:**

* **CPU Specs:**
* AMD Ryzen 7 9800X3D with 8 cores, each with 1MiB L2, and a 96MiB shared L3 cache.

A black rectangle with white text

AI-generated content may be incorrect.



* **OS/Environment:**
* Linux (WSL Ubuntu 22.04) on Windows 11.
* Java OpenJDK 17 headless installed via *sudo apt install openjdk-17-jdk-headless*

**Algorithm and Code break down:**

1. Set maximum array size to test to 512MiB.
2. As mentioned in the Assignment document, a typical Intel/AMD cache line is 64 Bytes. Since an int is 4 bytes, stepping by *step = 64/4 = 16* ints will touch one element per cache line.

A computer screen shot of a black background with text

AI-generated content may be incorrect.

1. Ensure each array, regardless of its size is accessed the same number of times. This prevents bigger arrays taking more time simply because they need to be accessed more times. Using *lengthMod = arr.length – 1* makes the index calculation wrap around, cycling through every valid index in the array repeatedly.

A computer screen with text

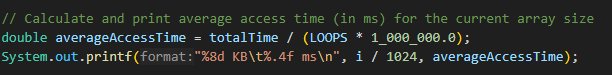
AI-generated content may be incorrect.

1. Measure the time from the uniform-access nested loops above, repeating 10 times to calculate an average and mitigate the impacts of outliers.

A computer screen with text

AI-generated content may be incorrect.

1. Printing results (average time per array)



**Results and Evaluation:**

**A screen shot of a computer

AI-generated content may be incorrect.**

* From 4KiB through 32KiB times stay around 48ms. This tells me that up to 32KiB, those arrays fit in L1 data cache. Although my CPU has 640KB of L1 cache total, there are only 80KB per core (since there are 8 cores total). In reality, 32KiB of the 80KB are used as L1 instruction cache, so a single thread can only store 48KiB of data in L1 cache.
* From 64KiB to 1024KiB times stay consistent around 65ms. This is likely because those arrays can be fully stored in L2 cache. Again, my CPU has 8MB total for L2, but only 1MiB per core.
* From 1024KiB to 65536KiB times stay consistent around 69ms. This is likely because those arrays can be fully stored in LLC. In this case, the 96MB of total cache is shared, so dividing the total by 8 is not necessary in this case.
* There is a huge jump between 65536KiB and 131072KiB from 70ms to 183ms. This sudden increase shows that the LLC has been exceeded, and RAM is now being used to fetch from main memory. It is important to note that a large fraction of this array size is still stored in cache (97MiB). This could explain there is also a significant increase from 131072KiB and 262144KiB from 183ms to 240ms. Since in 262144KiB not as large of a fraction can be stored in cache itself.
* After 262144KiB timings remain similar, 240ms and 264ms. This indicates that for both of those array sizes, RAM is frequently being utilised to fetch from main memory.
* Since these results were gathered using WSL (Windows Subsystem for Linux), OS scheduling can introduce some jitter (in the ~1ms range). Furthermore, Java can also be a bit less precise than C as it cannot access direct CPU cycles like we did in the lab. Hence, I decided to measure the time taken to loop through each array 10 times, and finding the average. This should mitigate the effects of any outliers, and all array sizes would be tested in similar conditions.

**Conclusion:**As mentioned in the above section, timings remain flat from 4KiB up to 64MiB, which indicates that the arrays fit entirely within the cache hierarchy (L1, L2 and shared L3/LLC). As array sizes exceed 64MiB, significant time jumps occur, signalling that the array does not entirely fit in LLC, and is now spilling into main memory. This suggests a usable LLC size of more than 64MiB and less than 128MiB, which is consistent with my CPU’s specifications. Therefore, I can conclude that my algorithm successfully highlights the boundary between cache hits (between 48-70ms) and cache misses (above 180ms).