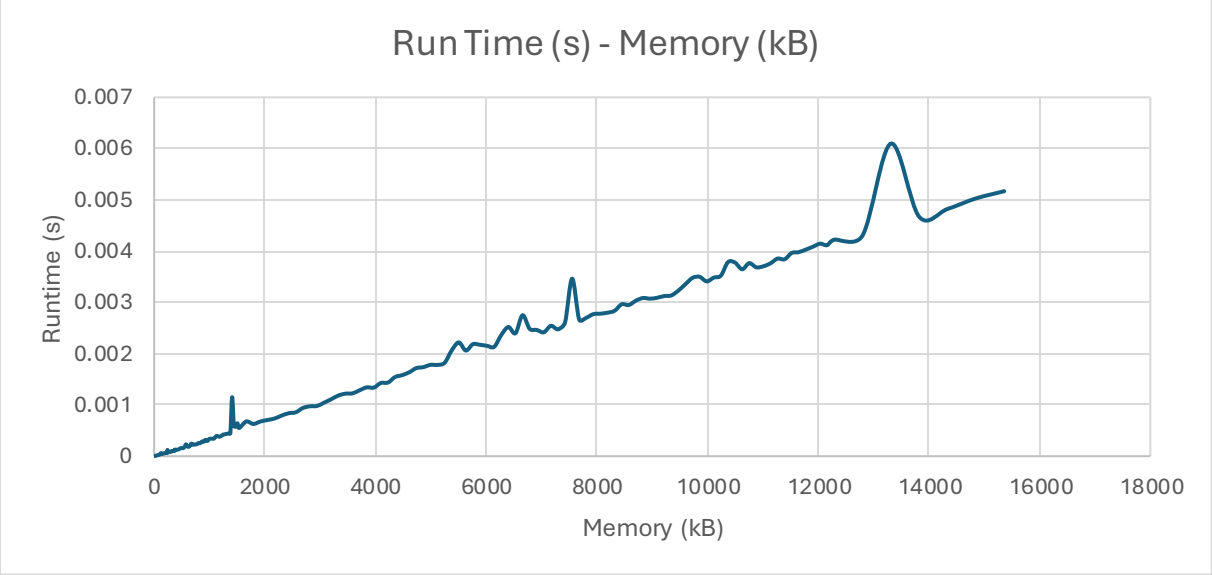
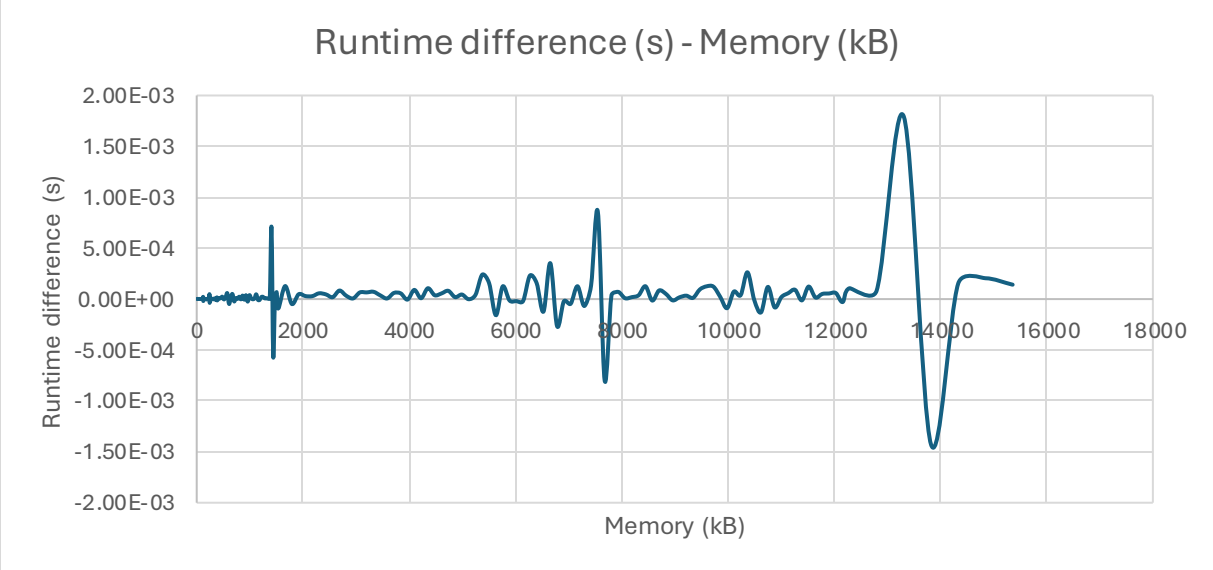
The data I will present here for part two of the coursework is a collection of my 2nd and 3rd attempts at measuring the effect of varying array sizes on program runtime. I will not be including my first dataset as I believe this was corrupted as a result of hyperthreading concealing memory access characteristics. After turning this off I completed the first half of part two [2(a) & 2(b)] but when the results I saw weren't exactly what I expected I decided to try and disable some prefetching features available on my system as a potential solution (Hardware & Adjacent cache line). Whilst not producing what I expected the results did vary again and I believe both datasets should be included in this report.

My CPU: **Intel i5 10400f** Cache: **L1 – 324Kb, L2 – 1.5MB, L3 – 12MB**

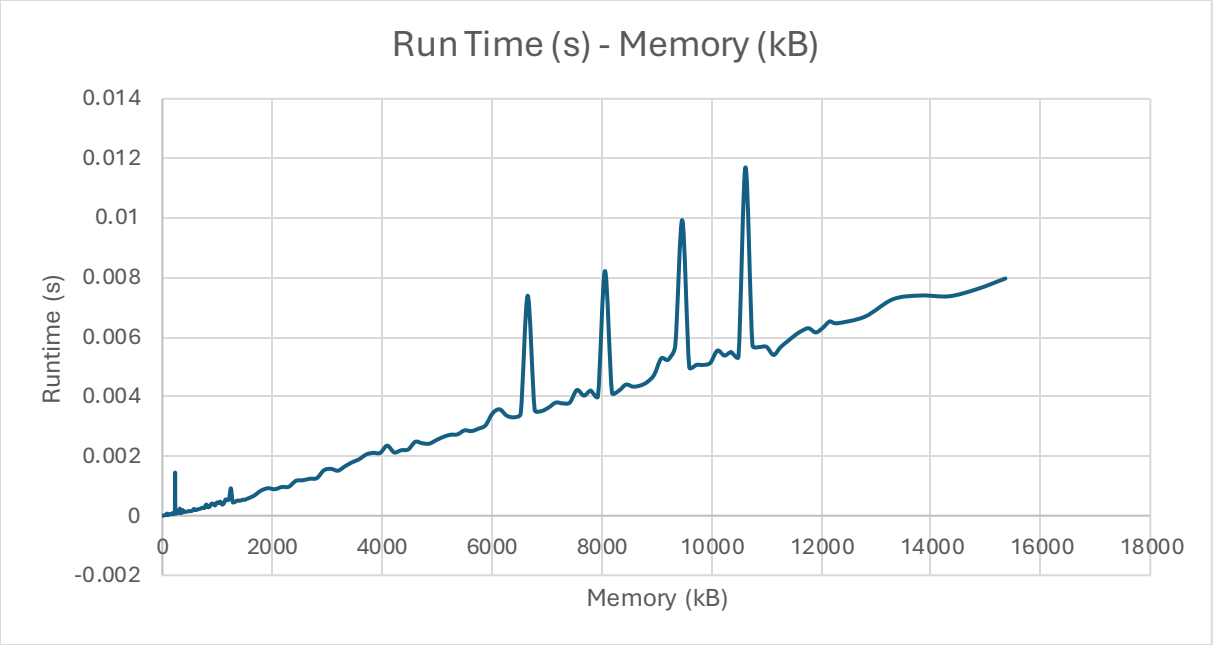
[2a] – Hyperthreading disabled, Prefetching enabled

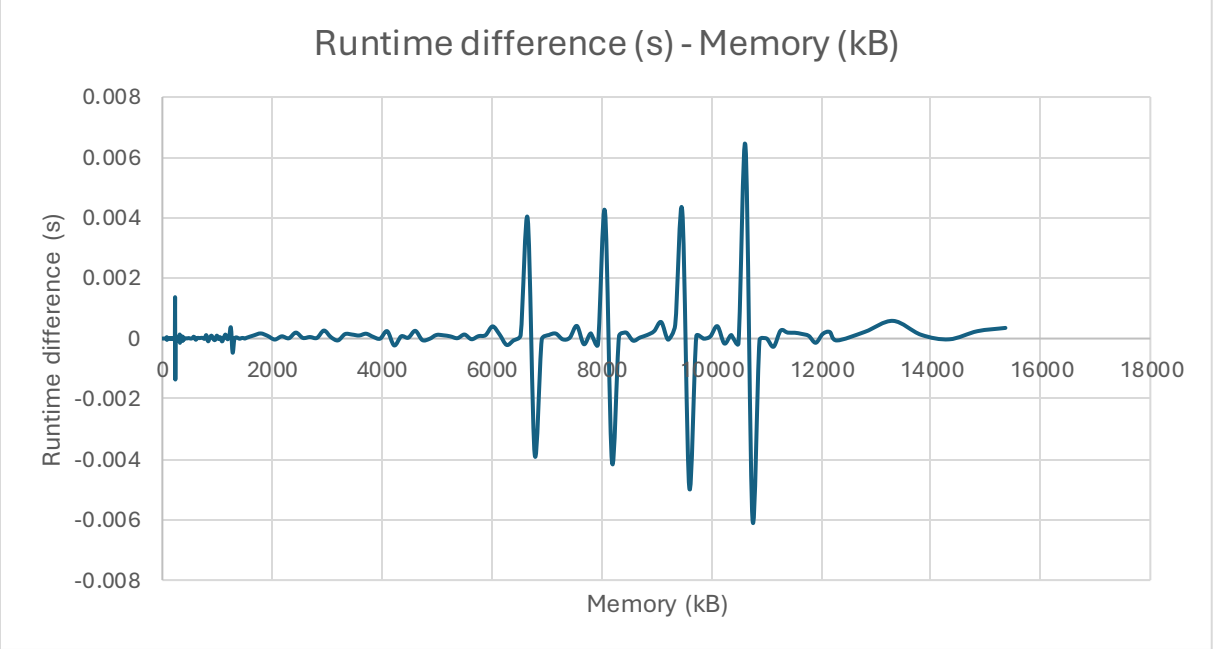




For this data I chose to explore offset values around 1408, 7552 & 13332 kilobytes which is where the runtime appears to spike in these graphs.

[2a] – Hyperthreading disabled, Prefetching disabled

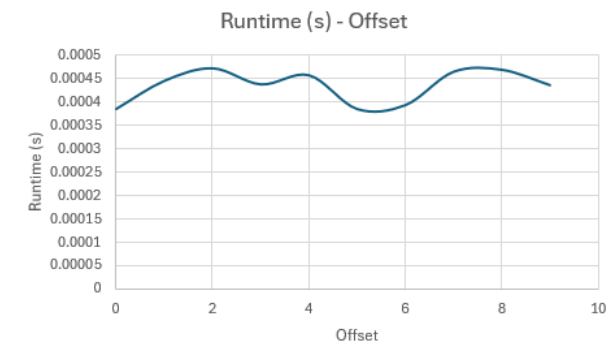


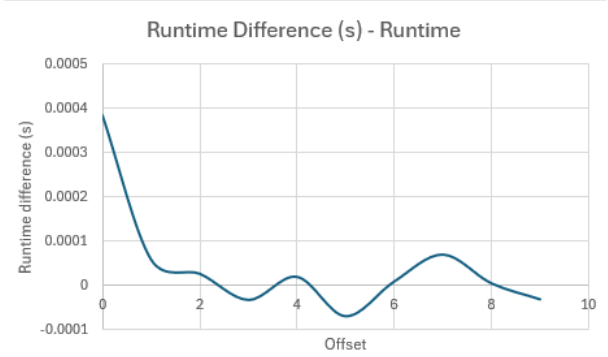


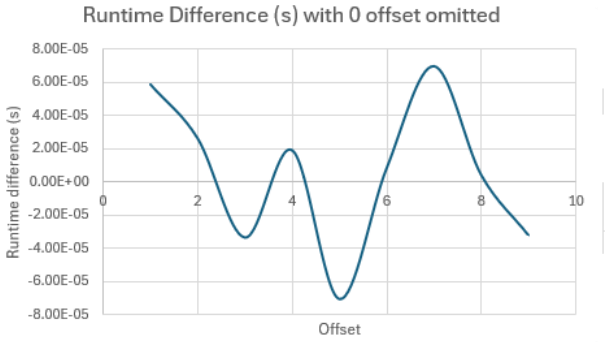
After disabling prefetching, I was able to see a clear spike where my L1 cache had reached capacity which was not at all visible before considering this setting. The noticeable spikes here that I chose to investigate for part b were 232, 6656, 8064, 9472 & 10624 kilobytes.

[2b] – Hyperthreading disabled, Prefetching enabled

[1408Kb]

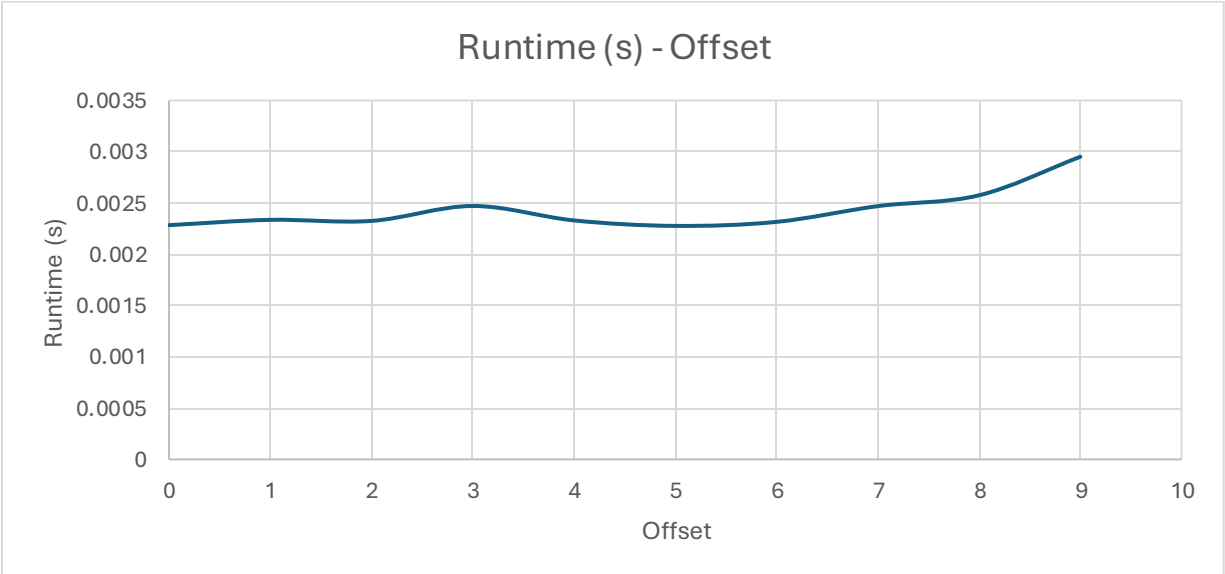




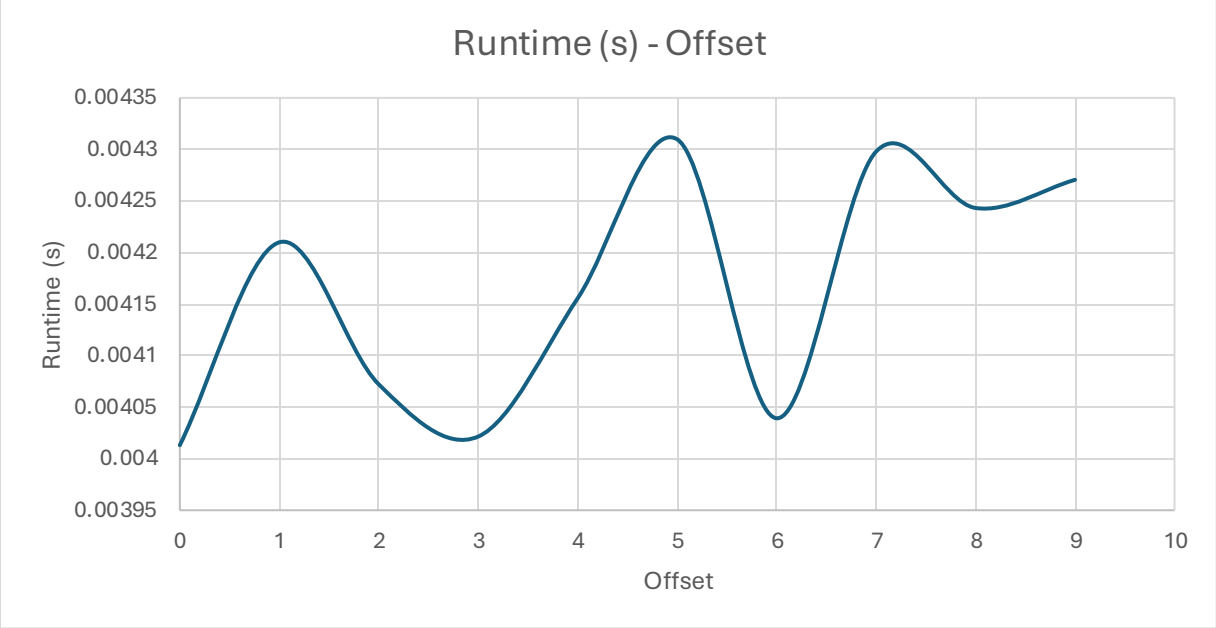


I began to plot the difference between the current iteration's runtime and the one prior to better visualize their relative performance. A pattern that emerged often was that seemingly regardless of the memory, an offset of zero would yield a runtime almost half an order of magnitude larger than the other offset values. I found that through omitting this first point we could better see how these larger offset values compare to one another.

[7552kB]



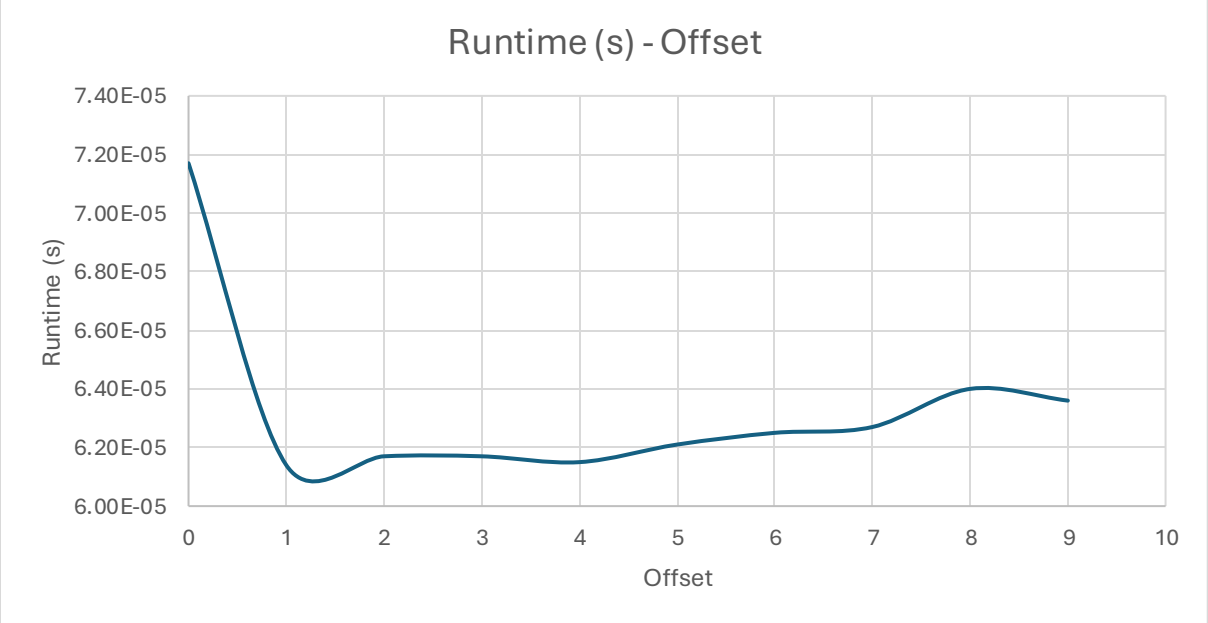
[13332kB]



The three tests conducted here indicate that after safely exceeding the L3 cache, variation in the offset of memory access has a much more tangible effect on the program run time. This semi-transient behavior could be the effect of continuously accessing main memory as it is practically much slower than cache access.

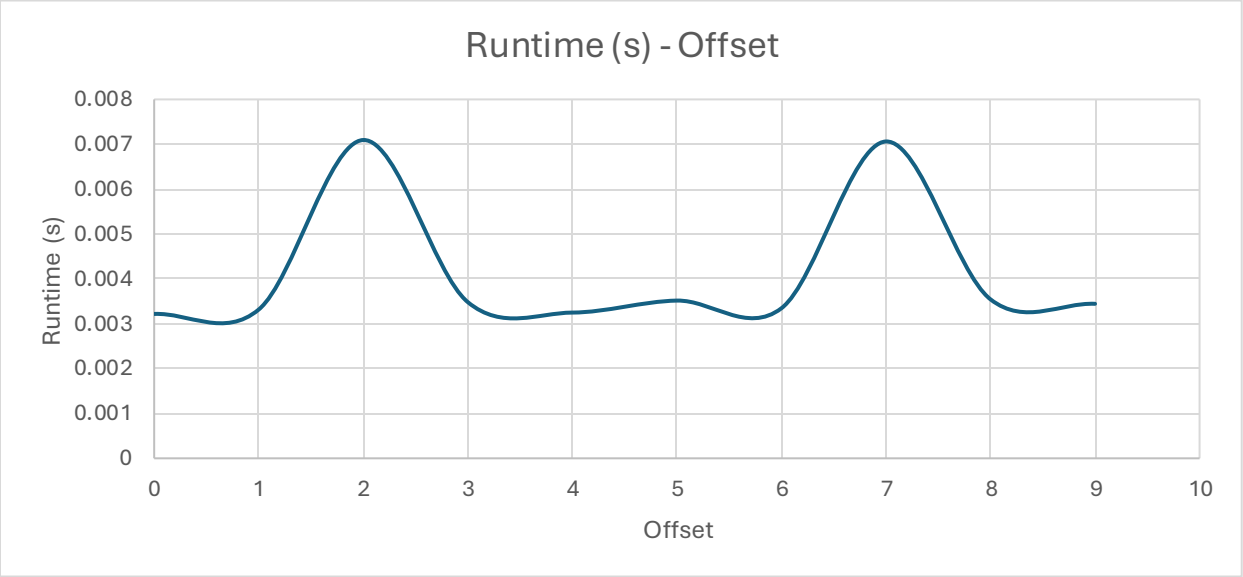
[2b] – Hyperthreading disabled, Prefetching disabled

[232kB]



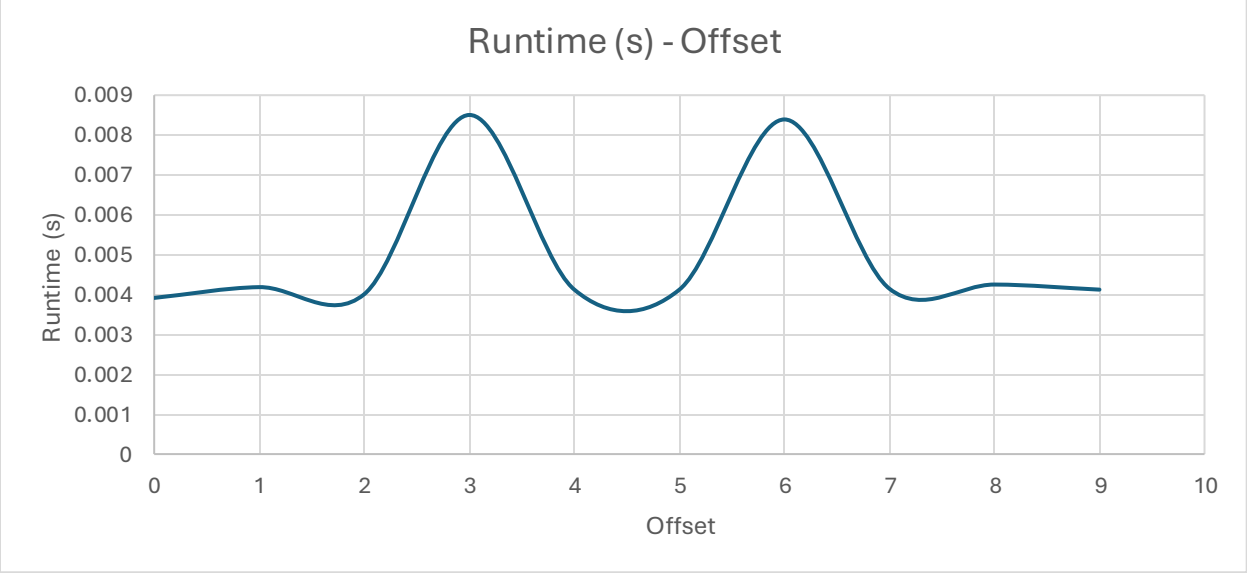
Here we find that operating close to the memory capacity of the L1 cache that an offset of one produces a noticeable reduction in runtime. This pattern could be explained by a reduction in cache thrashing. This is typically why we choose to offset our access patterns so that we are minimizing the amount of data that we are overwriting in the cache and making the most of the space we have.

[6656kB]



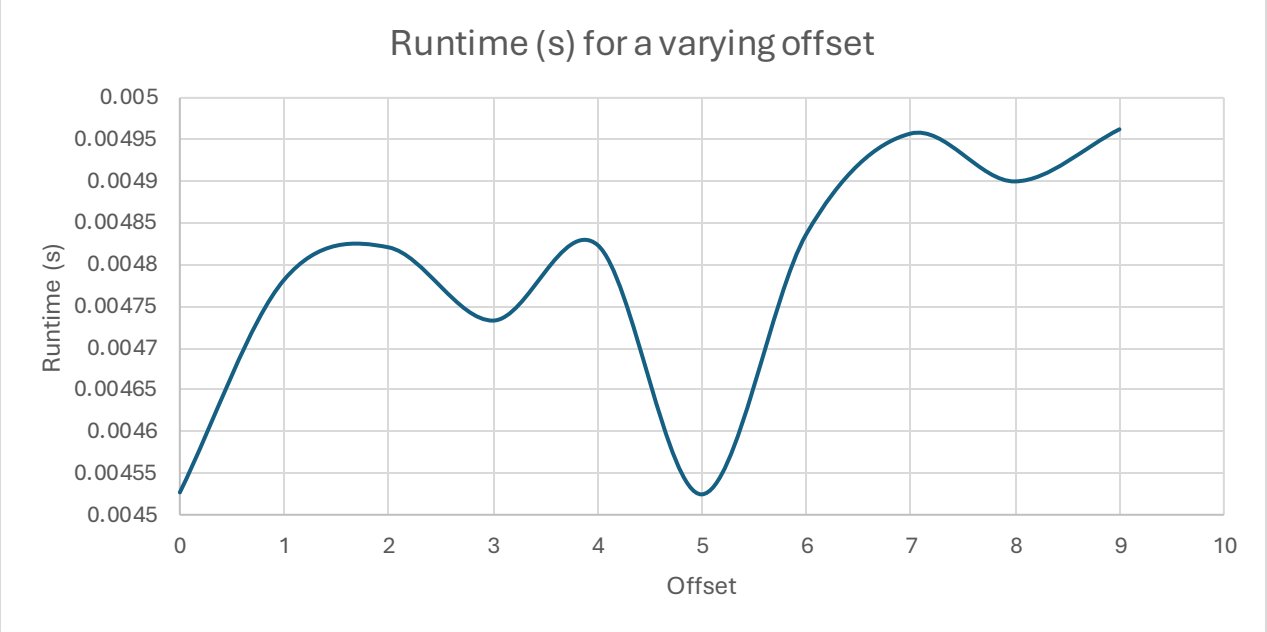
Here we see that an offset of two and seven will yield a drastic drop in performance with the runtime more than doubling for these values

[8064kB]



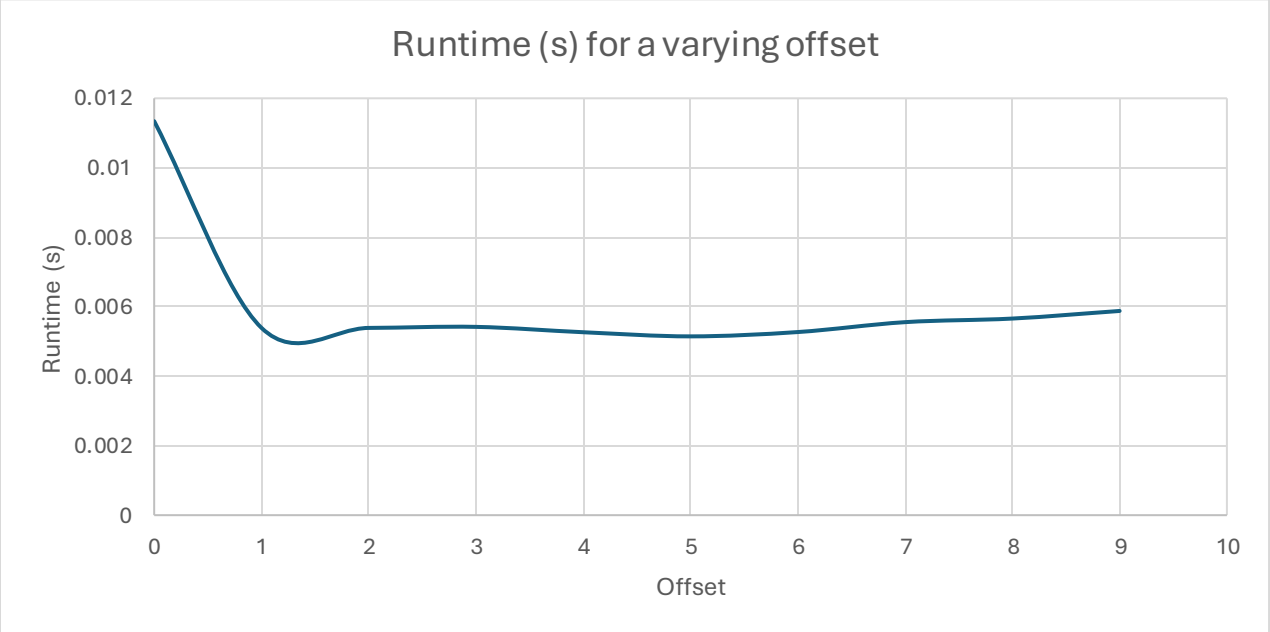
A similar sort of pattern here to the previous point in memory. This suggests that we are still operating in the L3 cache as the characteristics are almost identical.

[9472kB]



A very volatile pattern here with only a small jump in memory usage, probably worth mentioning here that the nature of these tests is very inconsistent, and this could even be a result of some sort of background processing although measures were taken to avoid this.

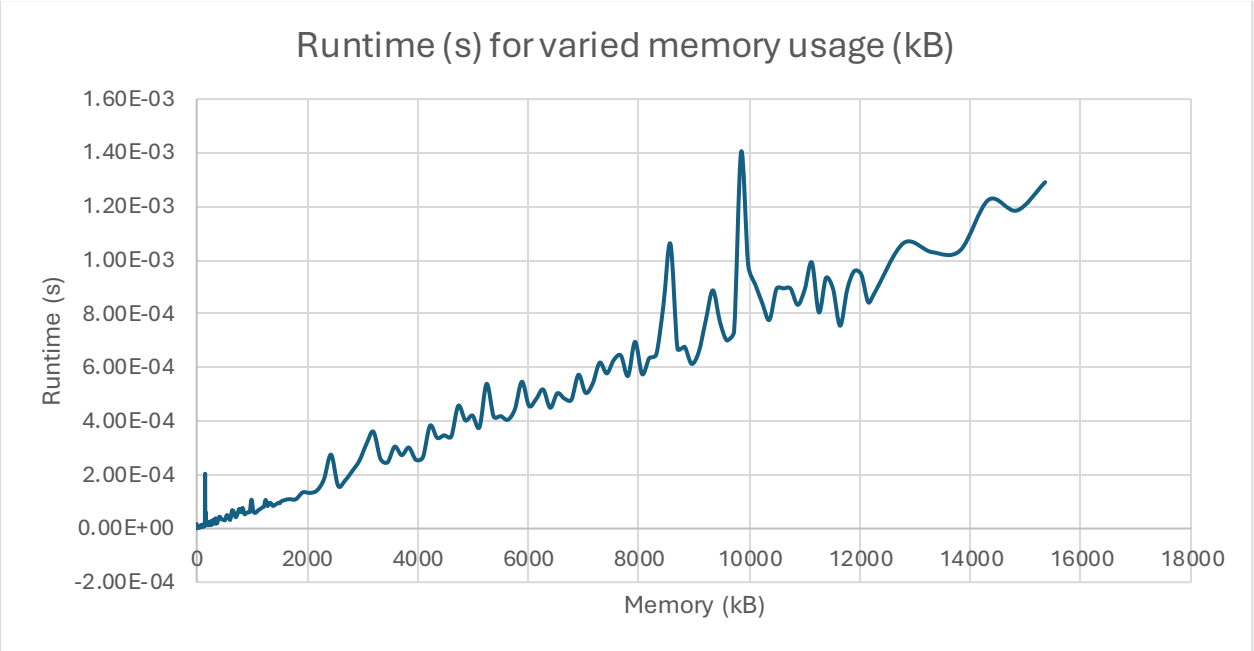
[10624kB]



We expect to be approaching the memory capacity of the L3 cache somewhere near here. Perhaps testing more offset values would be preferable or otherwise maybe reducing the testing memory slightly to capture the 'jump' in runtime observed in part A.

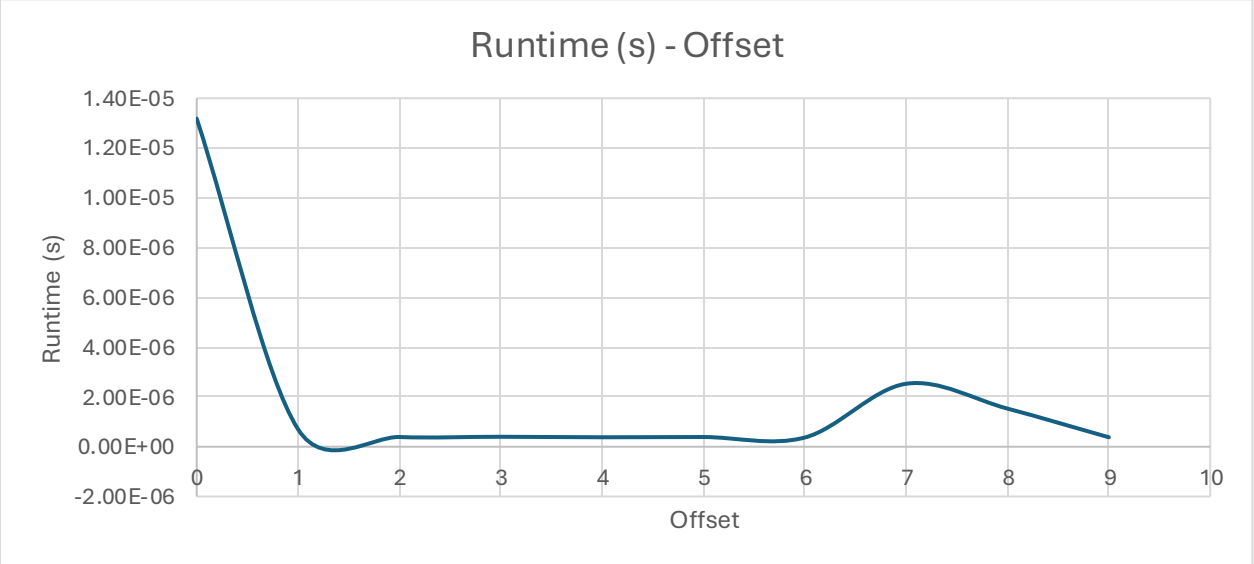
For the remaining questions I have chosen to enable cache prefetching as this was not a specified requirement in the coursework briefing document however, I will keep hyperthreading disabled

[2c]



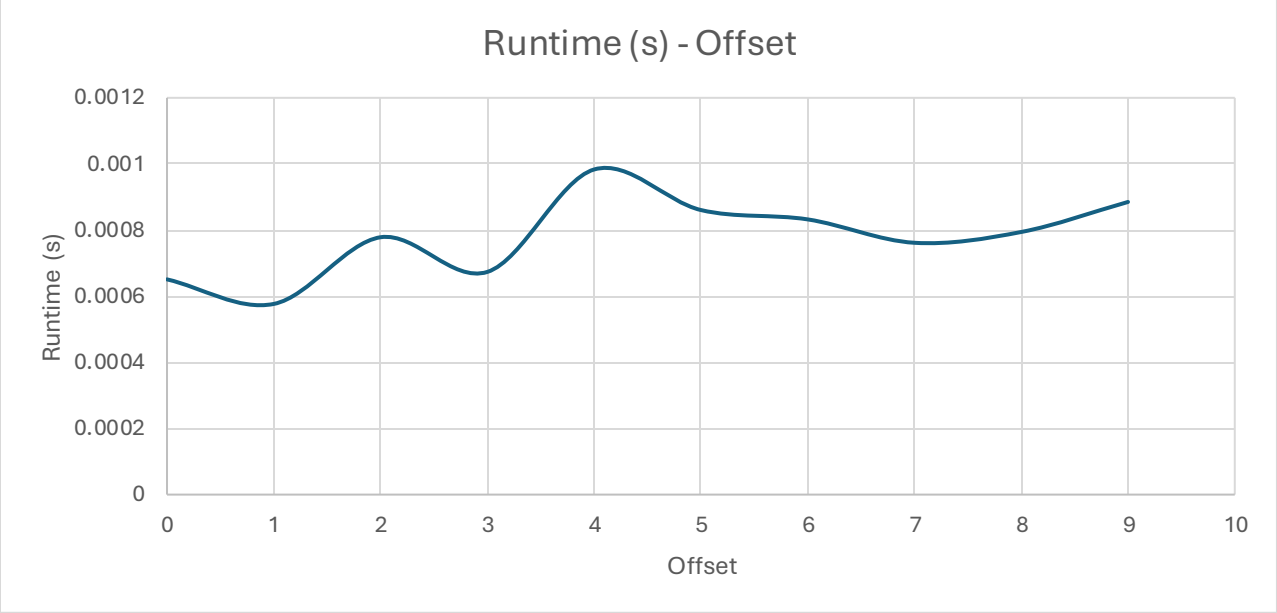
The values I have chosen to investigate are at the spikes at 150, 8500, 9800kB.

[150kB]



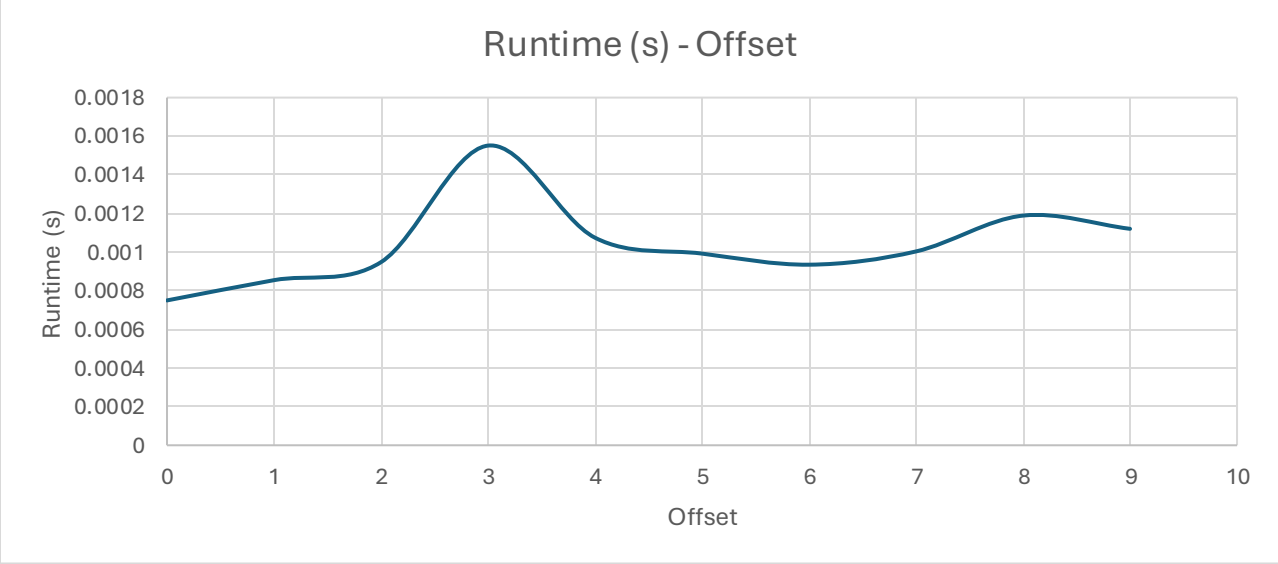
At this amount of memory usage, using an offset noticeably improves the runtime of the program.

[8000kB]



We see here that the contrary is shown as it appears as the addition of an offset slightly reduces our performance, increasing the runtime.

[9500kB]



Here it's more of the same, however the performance is worse with an offset of three. I suspect that cache thrashing is the result of these results. Some offset values are clearly making better use of cache access patterns, which explains their preferable performance.

[Q2d]

I made many small changes to my implementation during testing, and I found that the dynamic scheduling consistently offers better performance when we simulate an even computational workload which is in line with what we expect from the two protocols. I have included a test output here to show that computational complexity is even in the data static tends to perform better/on par with dynamic as the overhead of thread management outweighs its potential for optimization. We see that as our memory increases, dynamic allocation of work allows us to consistently finish these operations faster than static.

[Output Log]

Static scheduling: for n = 4000 and offset = 0, average wall time = 0.157051 seconds

Dynamic scheduling: for n = 4000 and offset = 0, average wall time = 0.158179 seconds

Static scheduling: for n = 8000 and offset = 0, average wall time = 0.320701 seconds

Dynamic scheduling: for n = 8000 and offset = 0, average wall time = 0.301468 seconds

Static scheduling: for n = 16000 and offset = 0, average wall time = 0.637693 seconds

Dynamic scheduling: for n = 16000 and offset = 0, average wall time = 0.592015 seconds

Static scheduling: for n = 32000 and offset = 0, average wall time = 1.20671 seconds

Dynamic scheduling: for n = 32000 and offset = 0, average wall time = 1.19173 seconds

Static scheduling: for n = 64000 and offset = 0, average wall time = 2.42603 seconds

Dynamic scheduling: for n = 64000 and offset = 0, average wall time = 2.41058 seconds

Static scheduling: for n = 128000 and offset = 0, average wall time = 4.77105 seconds

Dynamic scheduling: for n = 128000 and offset = 0, average wall time = 4.72666 seconds

Total time with dynamic scheduling = 9.49385 seconds

Total time with static scheduling = 9.64015 seconds

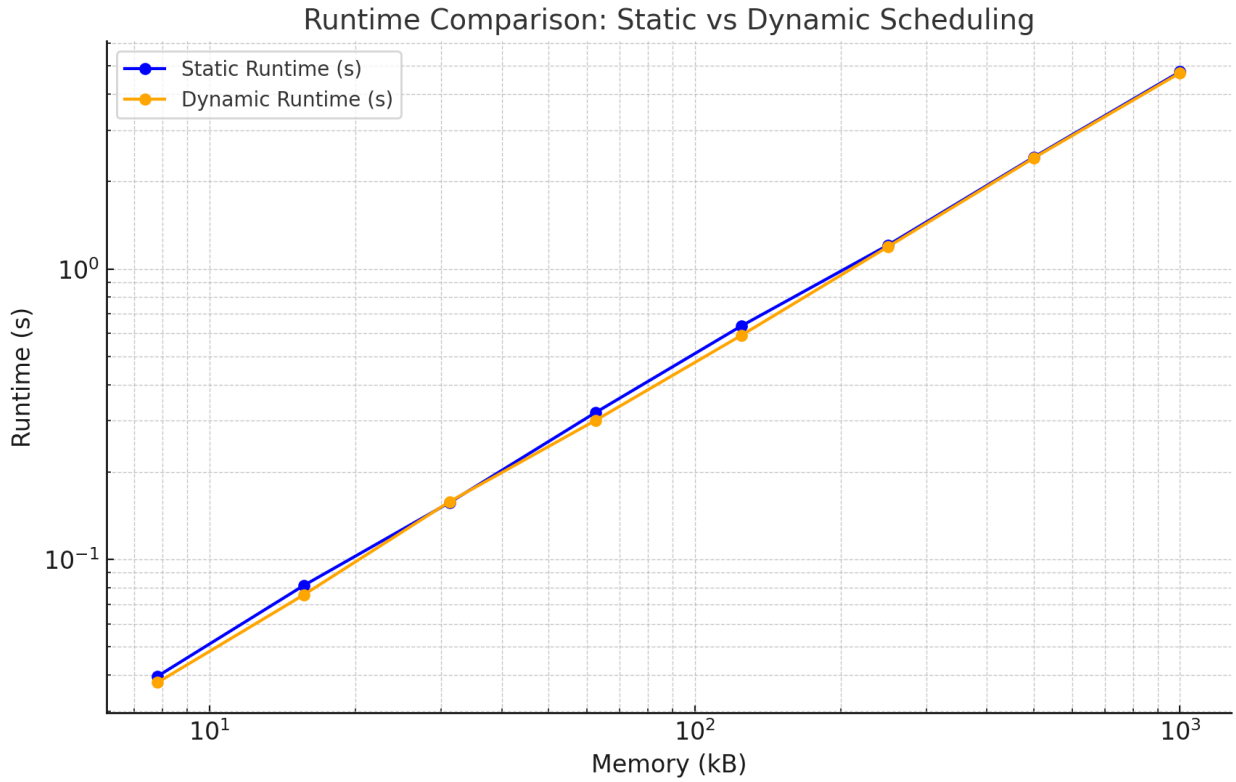
[Other results]

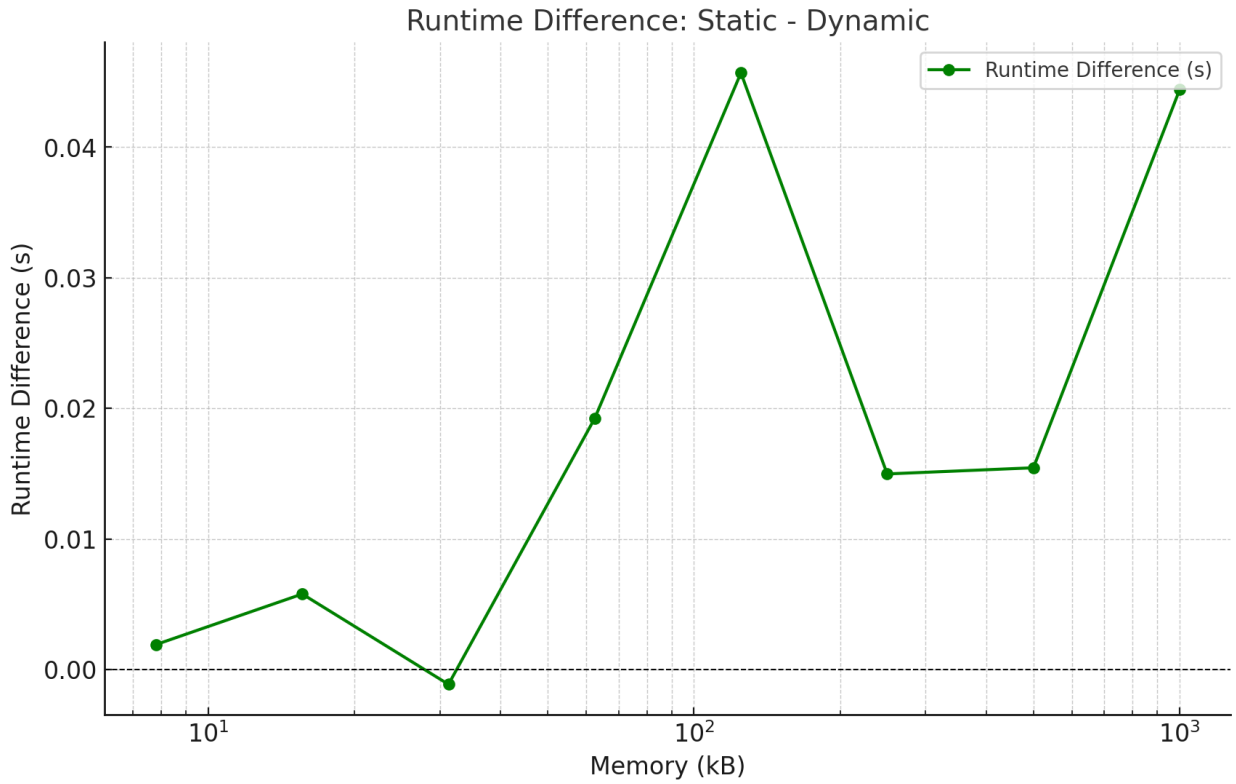
Total time with dynamic scheduling = 11.9608 seconds

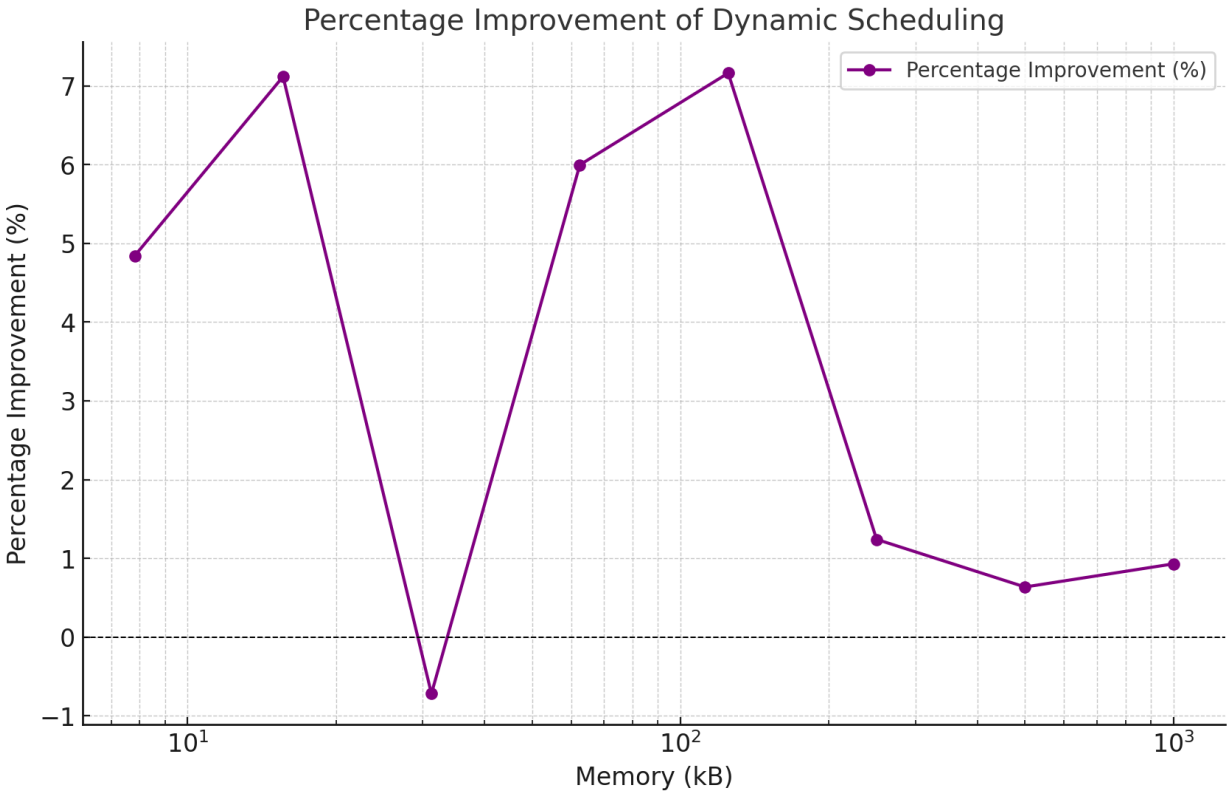
Total time with static scheduling = 12.996 seconds

Total time with dynamic scheduling = 72.1324 seconds

Total time with static scheduling = 73.6396 seconds


Runtime Comparison: Static vs Dynamic Scheduling
Static Runtime (s)
Dynamic Runtime (s)
Runtime (s)
10^{0}
10^{-1}
10^{1}
10^{2}
10^{3}
Memory (kB)


Runtime Difference: Static - Dynamic
Runtime Difference (s)
Runtime Difference (s)
0.04
0.03
0.02
0.01
0.00
10^{1}
10^{2}
10^{3}
Memory (kB)


Percentage Improvement of Dynamic Scheduling
Percentage Improvement (%)
Percentage Improvement (%)
7
6
5
4
3
2
1
0
−1
10^1
10^2
10^3
Memory (kB)