All tests were run on an Intel 7-7820x CPU with 4.5GHz turbo boost, 32KB L1 data cache, 1MB L2 cache and 11MB shared L3 cache.

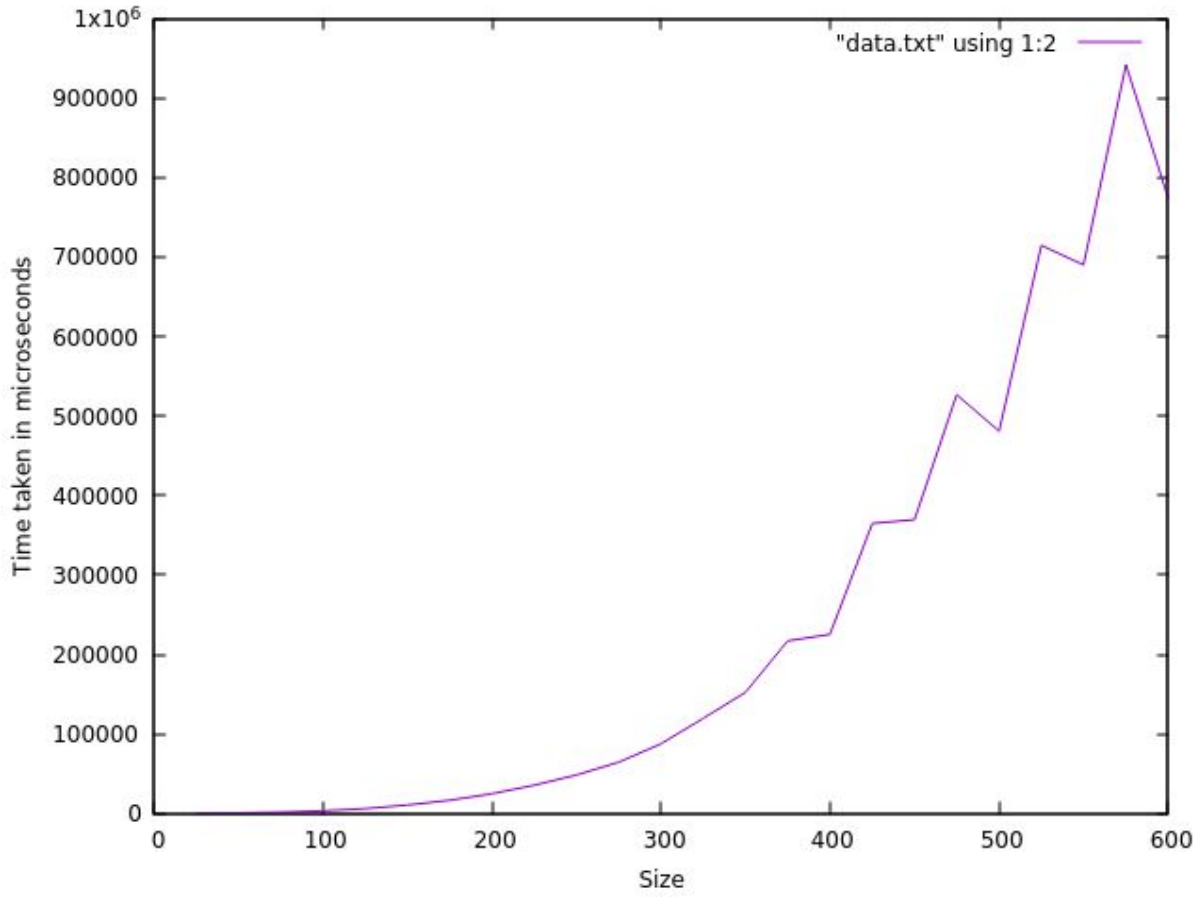
Task 1:

See file matmul.c

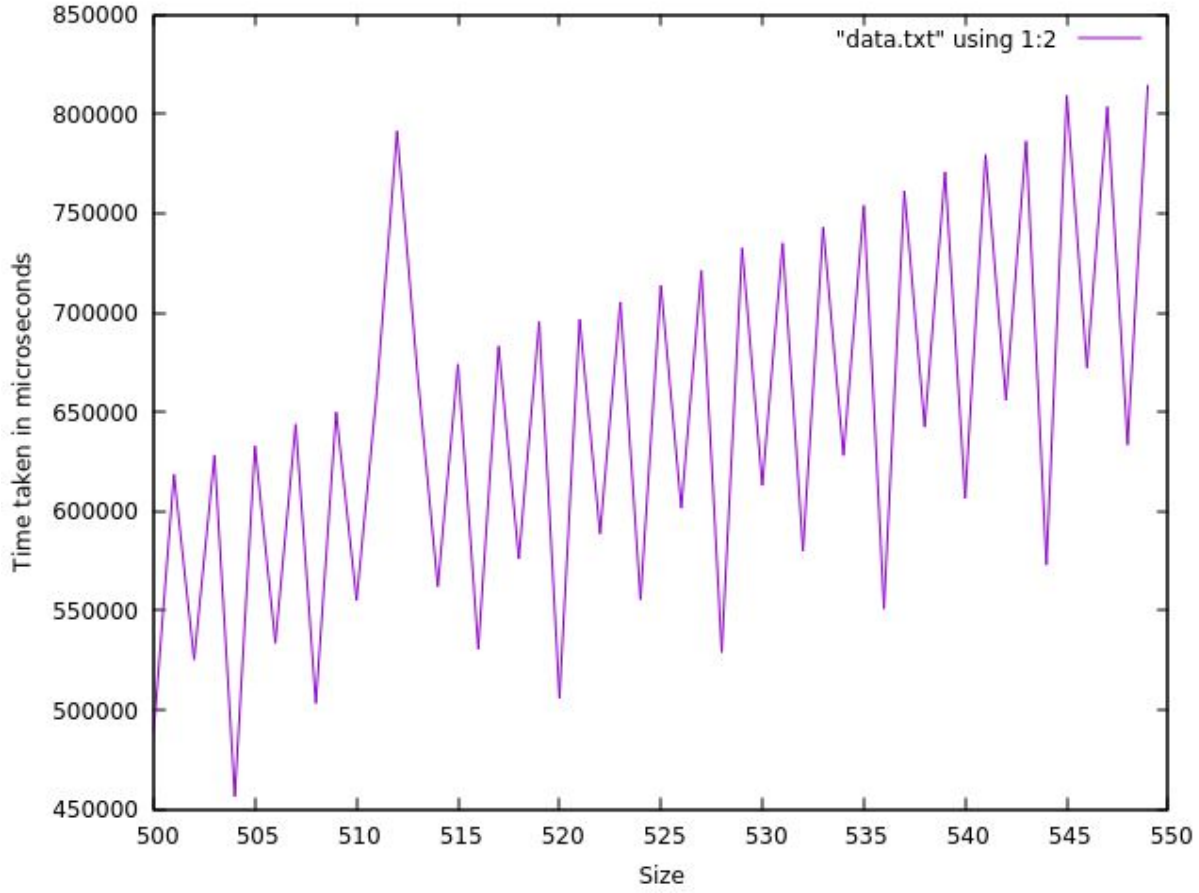
Task 2:

See file matmul_time_test.c and the postscript files.

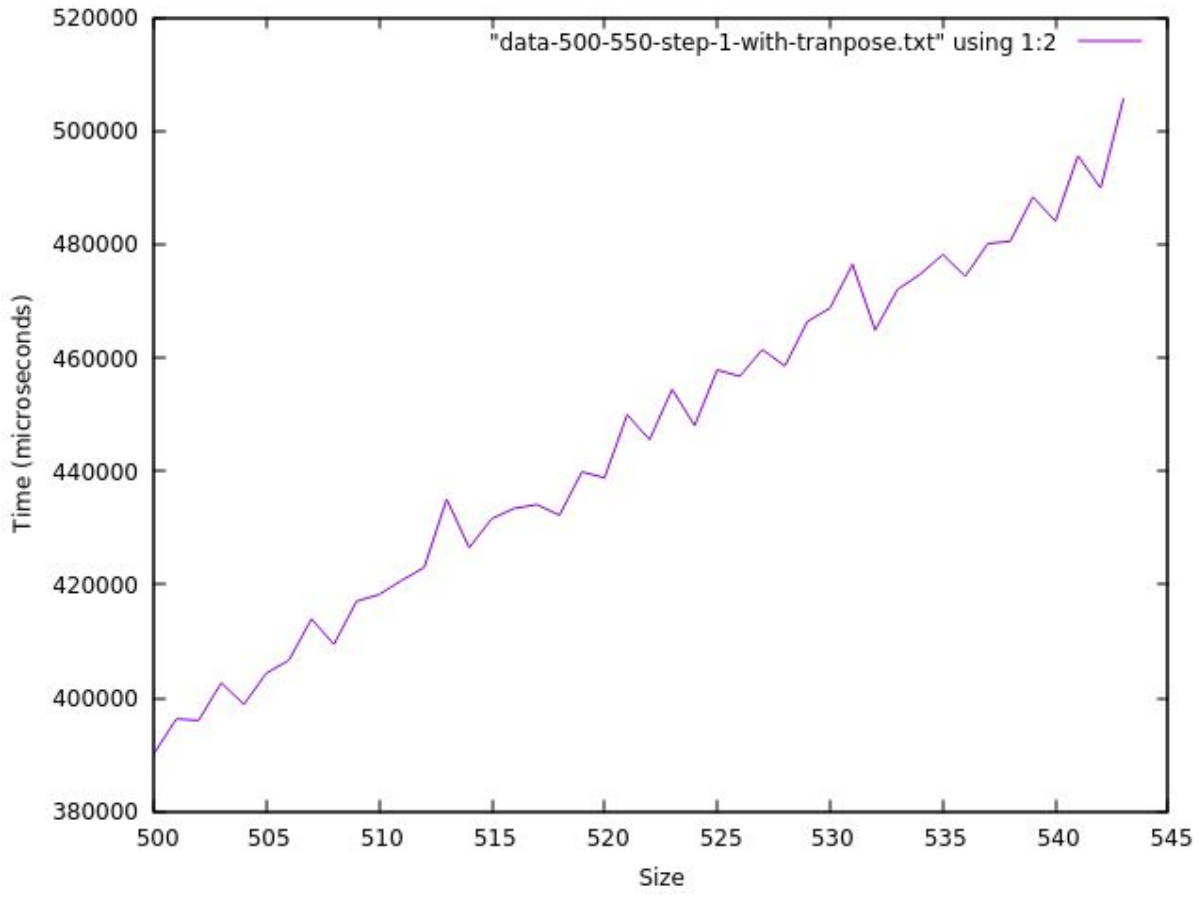
My approach was to multiply two nxn matrices 20 times, then find the average time taken. I used the range 10-600 with step size 25.
Below you can see my graph generated with gnuplot.
Up to n=375 the graph is exponential. This shows that matrix multiplication has an order of $O(x^3)$. The naive approach quickly becomes unusable.
However past this the growth of the time taken slows down or decreases in several areas.
I ran the program many times and got very similar results each time.



These results intrigued me so I re-ran the program with range 500-550 and step size 1, again averaging the time taken for 20 multiplications using each size. I did this to isolate the area and get a better data resolution.
The graph for this can be seen below. It is extremely interesting to see wild spikes in the time taken, both increases and decreases, especially since the size is only increasing by 1 each time. With the exception of n=512 we can see a repeating pattern of decreases and increases in the time taken.
I think because the access to B is strided then at certain sizes many addresses are mapping to the same to the same set in the cache. This is highly noticeable at n=512.
Consider an example: we access B[0][0] and the row is loaded into cache. We then access B[0][1] and so on. If each column access in B maps to different sets the rows could stay in the cache and be used again at B[1][0] and B[1][1].
But if many accesses to B map to the same set then the data will be evicted by the time B's row index is incremented and a column of that row is accessed again.



As a further test I changed the multiplication process slightly so that accesses to B are no longer strided (see task 4 for details). With this change the graph smooths out greatly, as seen below, though it is not completely smooth. This furthers my belief that due to strided accesses many elements of B mapped to the same set.



Task 3:

For testing compiler optimisation improvements I will used n=200 as the test case. As before multiplication was be repeated 20 times and the average time was recorded. I used gcc as a compiler for this test.

Compiler optimisations	**Time taken (microseconds)**
None	25129
-funroll-loops	24878
-fcombine-stack-adjustments	24922
-O1	11231
-O2	11185
-O3	11146

I tried many other gcc optimisation options as listed here:
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
However none of them provided any appreciable speedup.

In particular I tried optimisations that relaxed floating point math accuracy, but these did not provide any speedups either.
The speedup given by using -O1 shows that investigating compiler optimisations is definitely worthwhile.

Task 4:

Optimising cache accesses:

See file matmul_transpose.c

Matrices in C are stored in row-major format. This leads to poor cache usage when accessing the B matrix during multiplication with large matrix sizes. This is because we are accessing the columns of B and not the rows, while the cache is loading the row data. When the cache loads the row data it is not used as we are accessing the next row in the following operation. This is not an issue for accessing A since we are accessing A's rows in order.

To fix this we can simply transpose matrix B and adjust the addition stage from using b[k][j] to using b[j][k]. Using my original matmul.c code multiplying two 2000x2000 matrices took 67.4 seconds. However using this transpose optimisation the same multiplication took only 25.5 seconds (including the time taken to transpose the matrix), almost a 2.6x speedup for a simple change. Note for this test I did not enable any compiler optimisations.

Note that this optimisation can actually increase the time taken in certain situations. This is due to the overhead in transposing the matrices. This can happen if the matrices are small, or if B has a large number of columns but a small number of rows. In these situations accessing a column could incidentally load the next column in the unoptimised version since the rows are small, meaning that the optimisation is then pointless and time has been wasted transposing the matrices Before using this optimisation the matrix sizes should be checked.

Using the BLAS library:

See file matmul_cblas.c

After installing CBLAS and including the header file I was able to use the cblas_dgemm function to multiply matrices. Using matrices of size 2000x2000 the dgemm function took only 3.9 seconds, a massive improvement compared to my cache optimised version above.