Spike: Task 8
Title: Performance Measurement
Author: Sam Huffer, 101633177

Goals / deliverables:

- Demonstrate (through this spike report) the following performance and measurement concepts:
    1) Measurement of single and multiple functions' execution times.
    2) Linear and exponential ramp-up testing of function execution time, comparing to ramp-down testing.
    3) How repeatability will vary depending on test conditions.
    4) Comparison of functions providing the same functionality.
    5) Whether there's any difference in execution time between debug and release settings in Visual Studio.
    6) Whether turning down / off compiler optimisation makes a difference.
- Record and chart the data collected for each section.

Technologies, Tools, and Resources used:

- Visual Studio 2017
- Microsoft Word
- Microsoft Excel

Tasks undertaken:

1) *Single Tests: Demonstrate how to measure both single and multiple function execution time.*
    - Had a look at how the time recording was done in the ramp-up testing methods, and tweaked the main method of task08_sample1.cpp so that it would output how long the linear and exponential ramp-up methods had taken to execute individually, then how long they took together. "steady_clock::now()" assigned the start and end times to variables, the duration<double> "diff" took the result of the difference in the times, and "diff.count()" allowed for outputting the result to the terminal as part of a "cout" statement.
    - I ran each combination of methods 5 times, noting down the results. Once complete, I took the results and calculated the average time for each combination of methods.
2) *Ramp-Up Test: Execute and show (numbers/make a chart) both linear and exponential ramp-up testing of function execution time. Is there a difference to ramp-down tests?*
    - I examined the code for both of the ramp-up test methods in task08_sample1.cpp to get an understanding of how they operated, then ran the code as is.
    - I ran the code 5 times, and noted down all of the results for each iteration in Excel.
    - Below the linear and exponential data sets, I configured some cells to calculate the average time and time per item for each quantity of items tested in both the linear and exponential data sets.
    - I used the average data sets to create line charts, labelling the charts and axes, and displaying individual data points and trendlines appropriately.
    - Next, I examined the code for the ramp-up and ramp-down test methods in tasks08_sample2.cpp to check that the to methods did the same actions just in inversely scaling quantities. I added a loop so that the methods would repeat themselves 5 times to quickly give a large data set, and then ran the code, recording the data.
    - I configured cells to calculate the average duration in both seconds and nanoseconds for each number of items for both the ramp-up method and ramp-down method, and used them to construct line charts representing the data.
3) *Repeatability: Show (with numbers/chart) how repeatability will vary depending on test conditions.*

- This part confused me, so I asked Tien what this part of the spike was asking, and he explained this part more or less as “repeat a test 50 times or something, and have a look at how much variability there is in the output”.
- I took the method call for the linear ramp-up test method, and surrounded it with time recording code to determine how long it took and store the time in a vector<float>, and to loop through this process 50 times. Then I added a loop and “cout” statements after that to print all times to the terminal.
- Next, I ran the code, recording all of the data in Excel, sorting it, and adding below the data set cells to calculate the minimum, maximum, average and standard deviation values of the data set, and the normal distributions of each value.
- Last, I took the data and compiled it into a scatterplot akin to a bell curve.

4) *Function Comparison: There are two "char in string” counting functions provided (code sample 1). Clearly show the difference in performance (if any), and show if the speed difference is linear to string size. (Note, you will probably want to create random strings of the various size to test with.)*
   - I examined the code for both of the string counting methods to get an understanding of how they operated. Particularly, I had a look at how the time recording was done in the ramp-up testing methods, and replicated that around the methods being tested for this part of the spike. taking how it recorded the time it took to execute a chunk of code, and implemented that around the count_char…() methods. Ran the code several times using strings of different lengths, and recorded the results.
   - I adopted the existing test string and created 5 more test strings of varying lengths and compositions, and ran the code 5 times for each, noting down all of the results for each iteration in Excel.
   - Below the linear and exponential data sets, I configured some cells to calculate the average time each of the two methods took to calculate each string.
   - I used the average data sets to create a line chart comparing the two methods, labelling the charts and axes, including trendlines, and displaying individual data points as appropriate for the chart.
5) *IDE Settings: Show what, if any, is the difference in execution time between debug settings and release settings. (Remember to have a task that runs for long enough that it matters.)*
   - I took the linear ramp-up method and tweaked it to run through 100,000 items 5 times.
   - I ran the code 4 times using debugging settings, and 4 times with release settings, and noting down all of the results for each iteration in Excel.
   - Below the data sets for debugging and release settings, I configured cells to calculate the average time and time per item for each quantity of items tested in both data sets.
   - I used the sample of total times to create a scatterplot comparing the time taken for all data points, and used the average total times to create a bar chart directly comparing those two values.
6) *Compiler Settings: Turn down/off compiler optimisation and demonstrate a difference.*
   - I took the linear ramp-up method with the modifications for the IDE Settings testing, and tweaked it to iterate 20 times rather than 5, to increase efficiency of data entry for me.
   - To enable and disable optimisation settings for the code, I went into the solution explorer, right clicked on the project, and clicked on “Properties” at the bottom of the context menu (fig. 1).

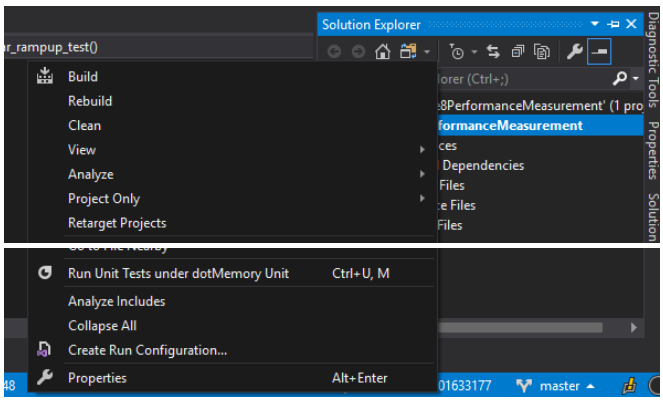


Figure 1: how to open up the project properties window.

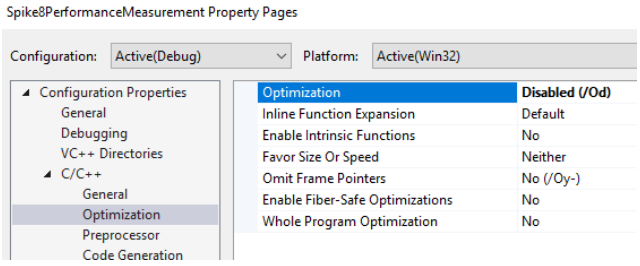


Figure 2: the default optimisation properties.

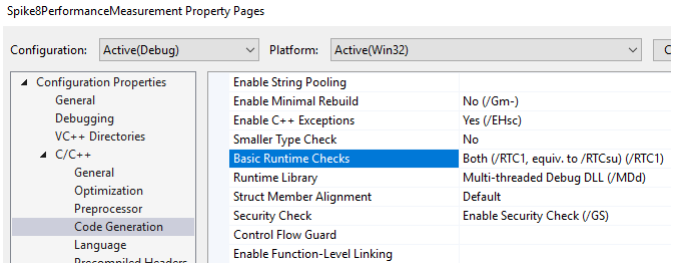


Figure 3: the default code generation properties.

This opened up the project properties window, which had the default values shown in fig. 2 and fig. 3.

- For the test with the optimisation off, I left the values as they were and executed the code, recording the results. For the test with the optimisation on, I altered the optimisation and code generation settings to what is shown in fig. 4 and fig. 5, turning on as much optimisation as I could without generating errors about conflicting settings. The “Optimisation” setting wouldn’t accept any value other than “Disabled” without throwing an error unless the “Basic Runtime Checks” field was changed to “Default”. Once the optimisation settings were configured, I ran the code again and recorded the results.
- Once all the results were recorded, I generated a scatterplot of the two data sets, comparing them directly, and a bar chart comparing the averages of the data sets.

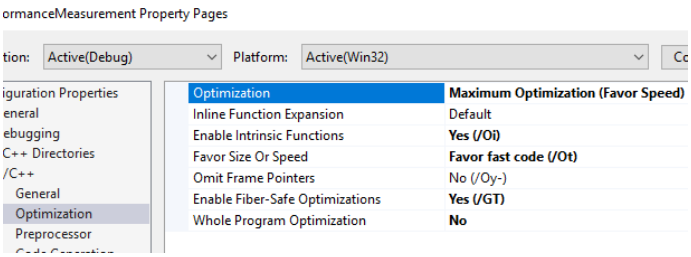


Figure 4: the optimised optimisation properties.

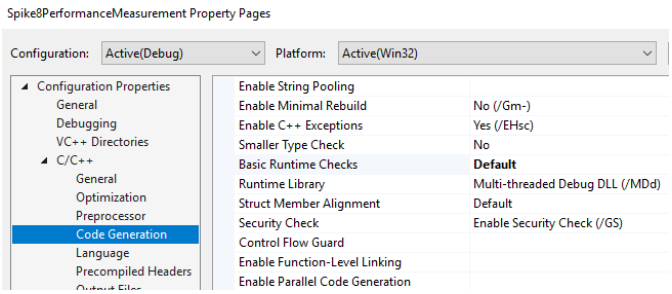


Figure 5: the optimised code generation properties.

What we found out

General

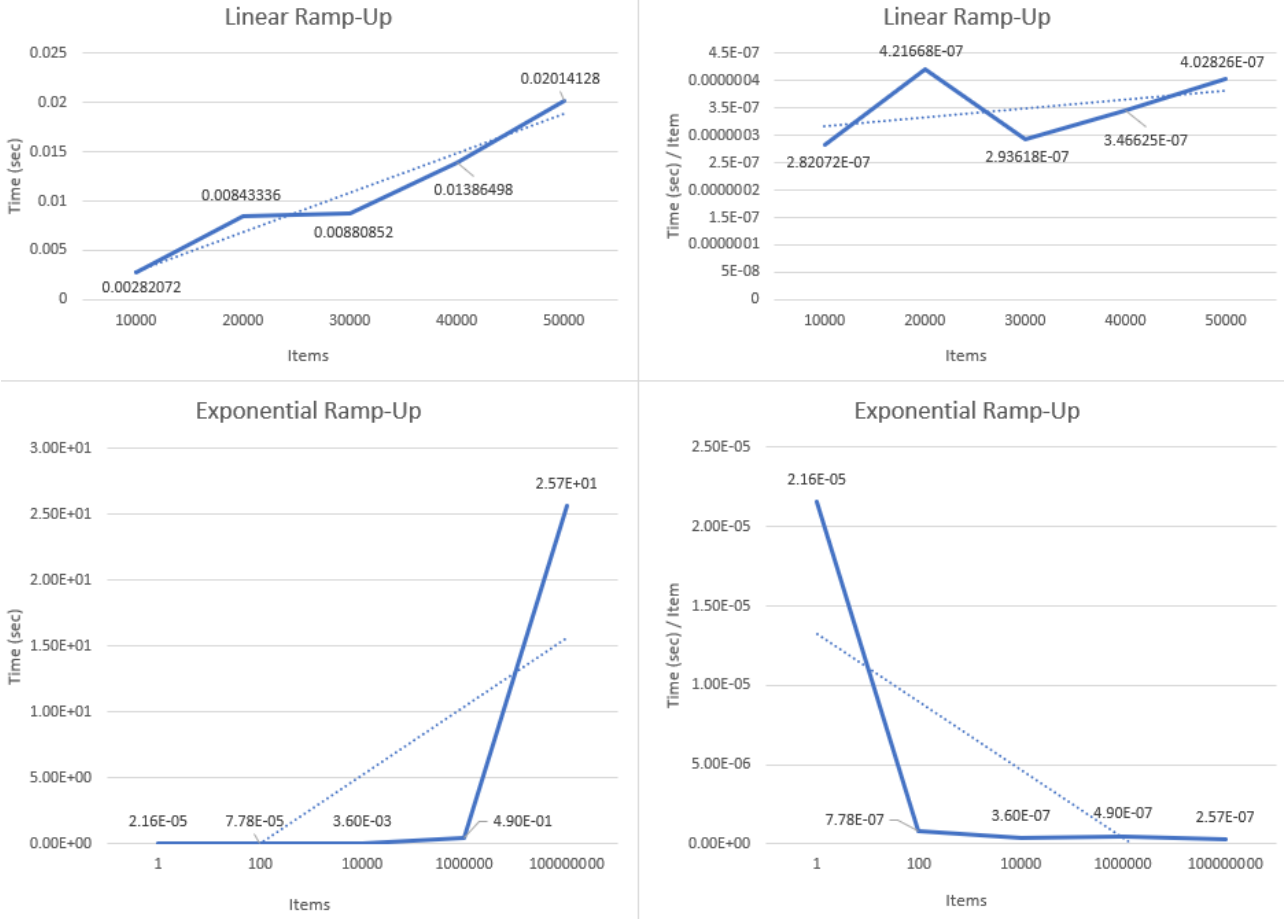
- How to use stopwatch-like functionality in C++ to examine how long something takes to be executed.
- How to do linear and exponential ramp-up testing, and repeatability testing.
- How to enable / disable various optimisation settings.

Test results

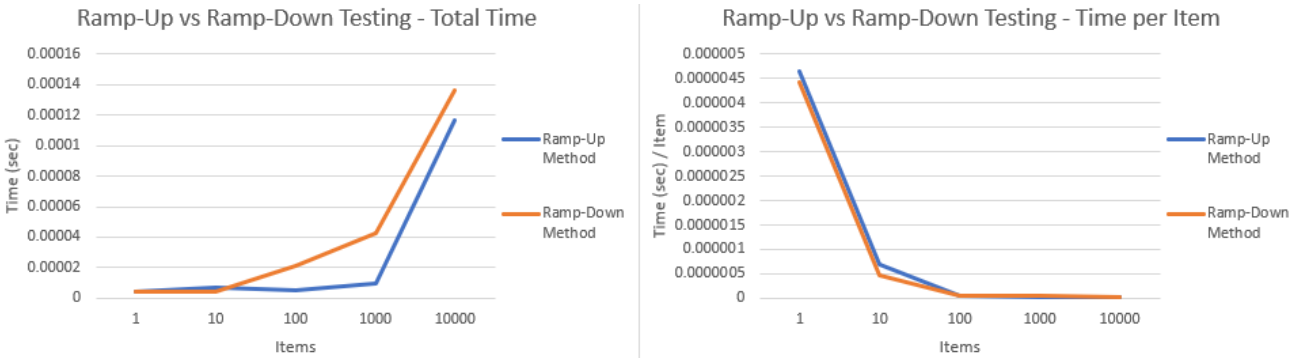
1: Single Tests

I recorded how long the linear ramp-up and exponential ramp-up methods took to execute with their default values, both individually and together, five times over. The average time for the linear ramp-up method alone was 0.02527926 seconds; for the exponential ramp-up method, 1.5372924 seconds; and for both together, 1.114022 seconds. From these times, it would seem my computer was struggling a bit when it was doing the exponential ramp-up method on its own, as its average time was greater than the combined average of it and the linear ramp-up method.

2: Ramp-Up Test

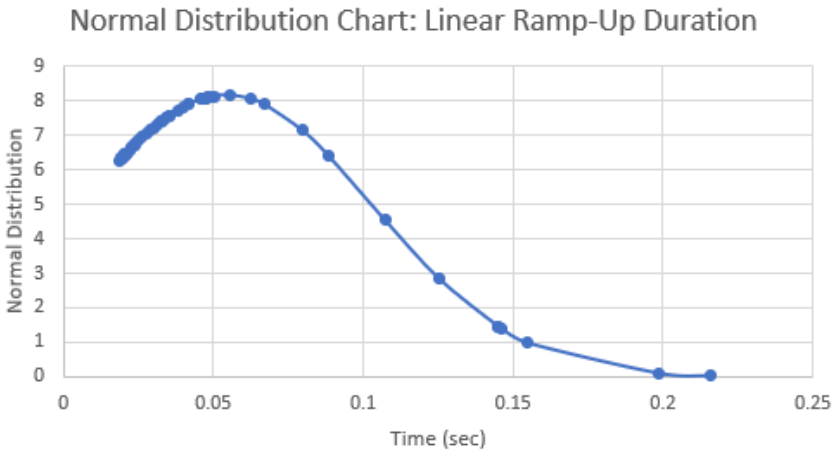


Graph 2a: the total time for the linear test method to execute was roughly linear, increasing in duration proportionate to the increasing number of items. Interestingly, its duration per item was rather variable, but the trend line did show a very slight increase in time per item as the number of items increased; I suspect that might be due to the linear disparity in average time per item for 10,000 items and 50,000 items. The exponential ramp-up method showed an exponential increase in time taken to process all items in line with the exponential increase in the number of items, but an exponential decrease in the time per item as the number of items increased exponentially. Note: while I was executing the two methods to obtain their respective data sets, I noticed that, at least several times, the exponential ramp-up method, would print out all the times and the time for the 100,000,000 item iteration would be zero point something like the rest, but it would stall before printing "done", and once "done" was printed, the number would change to twenty-something. That said, each increase in the number of items to be processed by a factor of 10 (barring the single-item iteration) corresponds to an increase in duration by roughly a factor of 10, which the aforementioned data point conforms to, so it would not be unexpected that the zero point something values originally displayed were an algorithmic error and the twenty-something values were the accurate times.

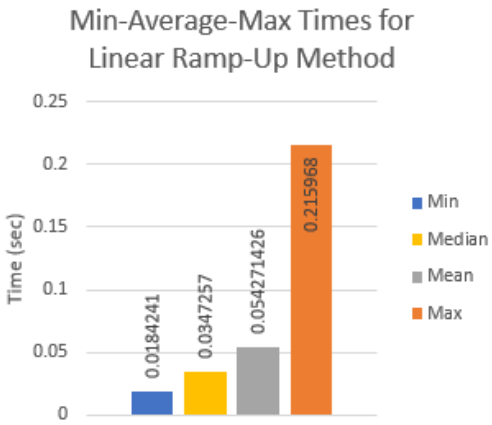


Graph 2b: I looked on Google for a comparison of ramp-up testing versus ramp-down testing, and the best answer I could find was that ramp-down is simply the inverse of ramp-up, where you reduce the number of items being tested each iteration. (http://www.geekinterview.com/question_details/42823) Then I noticed that task08_sample2.cpp had ramp-up and ramp-down methods operating exactly the same barring changing the intervals inversely to each other. The two methods performed roughly the same, the ramp-down method being slightly more time consuming at higher volumes of items.

3: Repeatability

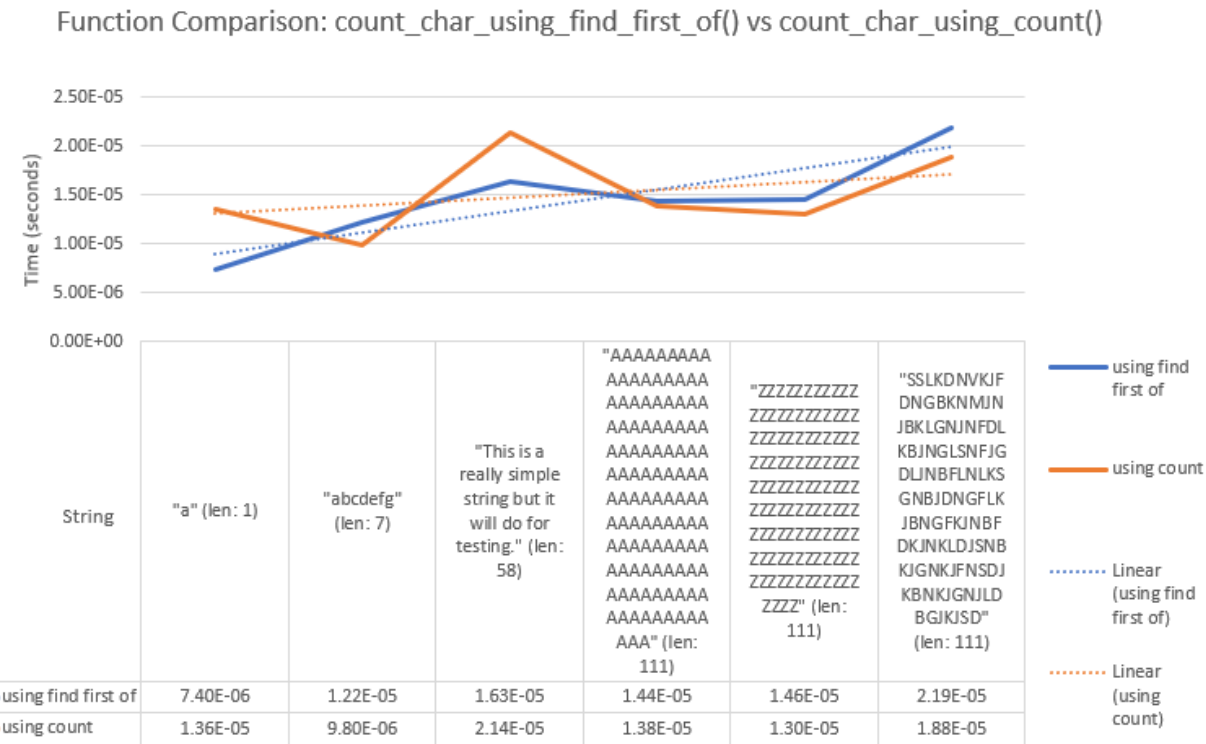


Graph 3a: a bell curve of all 50 times recorded for the duration of the linear ramp-up method. The values recorded are mostly bunched up below the 0.05 seconds mark. Occasionally linear_ramp_up() does run substantially longer, but for the most part the variability in its duration is minimal.



Graph 3b: the lowest, mean (average) and median (centre-most) values are pretty unremarkably bunched together. The highest value, however, is a significant outlier, and likely skews the mean somewhat higher than it would otherwise be.

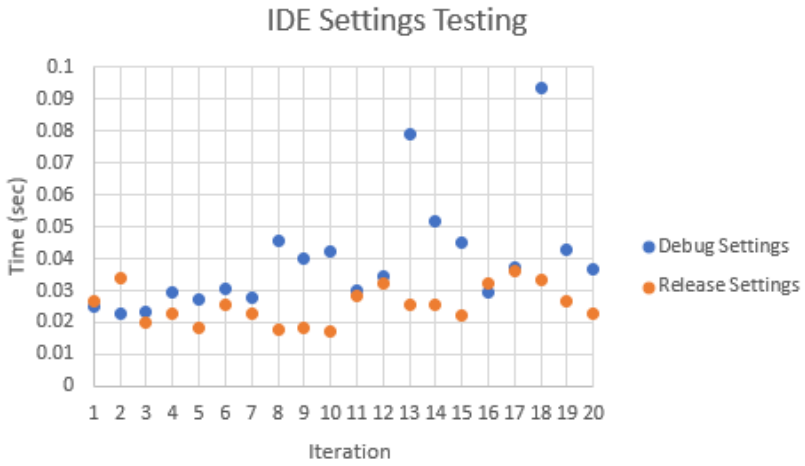
4: Function Comparison



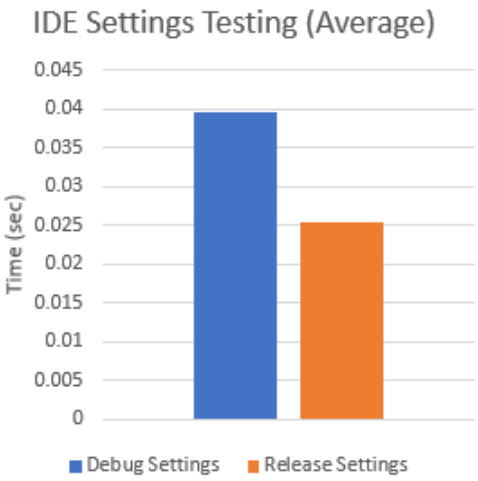
String	"a" (len: 1)	"abcdefg" (len: 7)	"This is a really simple string but it will do for testing." (len: 58)	"AAAAAAAAAA... (len: 111)	"zzzzzzzzzz... (len: 111)	"SSLKDNVKJF DNGBKNMJN JBKLGNJNFDL KBJNGLSNFJG DLJNBFLNLKS GNBJDNGFLK JBNGFKJNBF DKJNKLDJSNB KJGNKJFNSDJ KBNKJGNJLD BGJKJSD" (len: 111)
using find first of	7.40E-06	1.22E-05	1.63E-05	1.44E-05	1.46E-05	2.19E-05
using count	1.36E-05	9.80E-06	2.14E-05	1.38E-05	1.30E-05	1.88E-05

Graph 4: the time each method took to process the strings was, on average, proportionate to the length of the string. Diversity of characters seemed to affect efficiency as well (compare the three strings of length 111). The using count method, for whatever reason, showed the most variability, struggling particularly with the third string ("This is . . .") but the using find first of method exhibited the steeper trendline. Whether that's because of the former method's variability or poorer handling of longer strings on its own part is unclear from the data collected.

5: IDE Settings

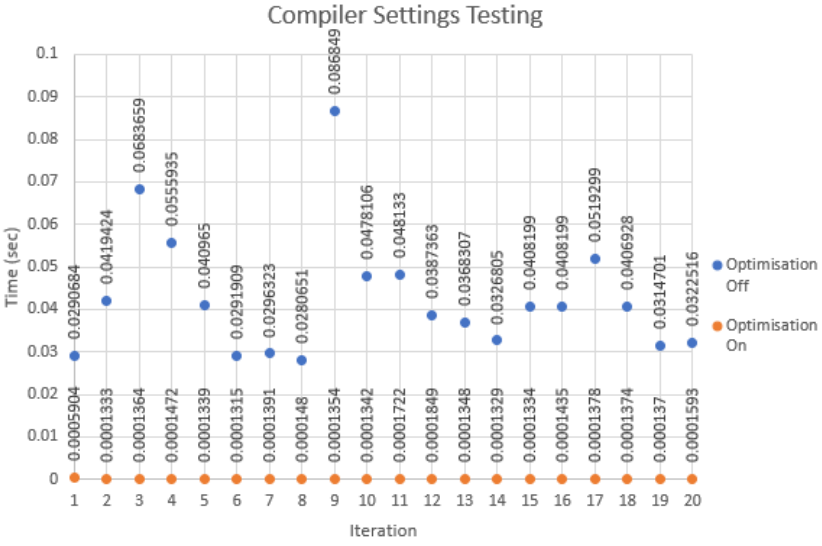


Graph 5a: the majority of the time, the linear ramp-up method, processing 100,000 items, ran faster using release settings than debug settings, sometimes substantially so. This suggests that debug settings do introduce additional overhead that impacts performance slightly.

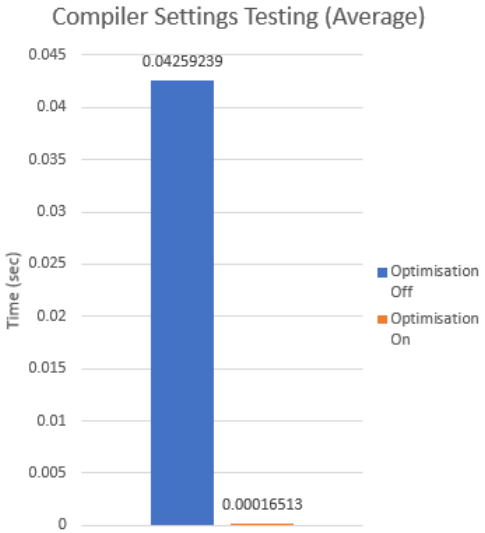


Graph 5b: the average times taken for the linear ramp-up method to iterate through all 100,000 items using debug settings and release settings. On average, the latter reduced the duration by about 0.015 seconds.

6: Compiler Settings



Graph 6a: the compiler settings testing showed a clear different in processing speed between having optimisation settings active and inactive. All times given without optimisation active were in the range of M x 10^{-2}; all times with optimisation active were in the range of N x 10^{-4}, 100 times faster, give or take.



Graph 6b: the average time without optimisation was almost 258 times greater than the average time with optimisation.