

|  |
| --- |
| CONCURRENT SYSTEMS ASSIGNMENT REPORT  CS3014 – 2018/19  Code Viewable at: <https://github.com/pkjennings999/ConcurrentAssignment> |

|  |  |  |
| --- | --- | --- |
| Team members: |  |  |
| Prapti Setty  Patrick Jennings |  |  |

TABLE OF CONTENTS

Getting Started

Starting the assignment

Setup

Trying Parallelization

#pragma omp parallel for collapse(3)

#pragma omp parallel for collapse(6)

Trying Vectorization

\_m128

\_m128d

SIMD vectorization

SIMD vectorization on reordering

Unrolling the loop

Other things we tried

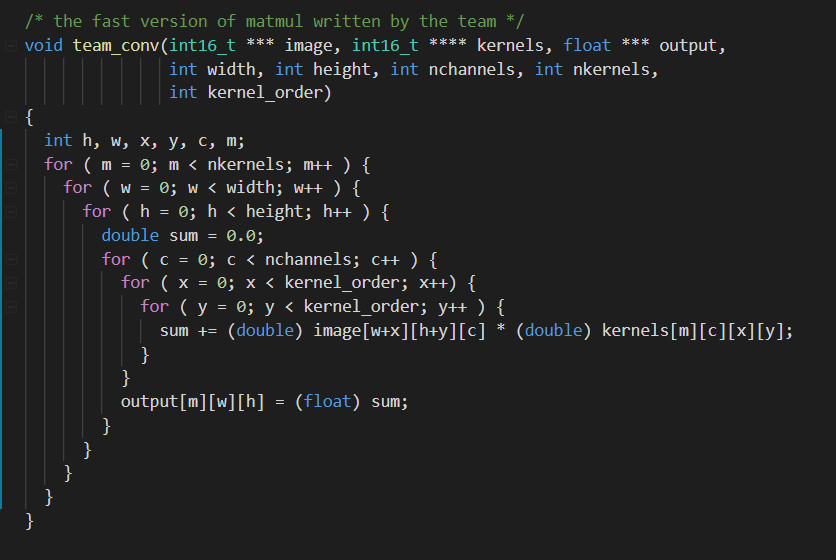
Pre vs Post Increment

Restrict vs No Restrict

Casting (doubles, ints)

Hashmaps

# Getting started



## Starting the assignment

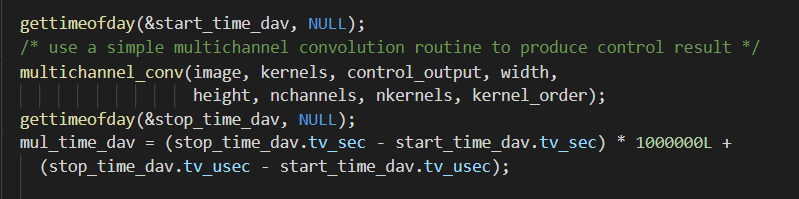
We started the assignment in reading week and our first step was to understand what was happening in the original code given to us. We ran the code a few times with some random values to see what was happening and tried to step through it using the debugger.

We then re-read the spec more carefully to determine exactly what was being asked before beginning out implementation

## Set up

Since we were working in a group, it was quite important to use some kind of version control so we would minimize our conflicts as much as possible. We set up a Github repository shared between the two team members, so we could easily share code. This also proved very useful when we wanted to work on code on our own machines and then clone and pull it to the machine we were going to run the code on (Stoker).

We then decided to add in some lines of code to allow us to track how long the original code was taking and track the speedup we gained with what we implemented.



We wanted to be able to run the sample inputs 50-70 times initially (we reduced this down to 20 as we realized we were hogging Stoker), so having something like an output file that recorded something along the lines of the output below:

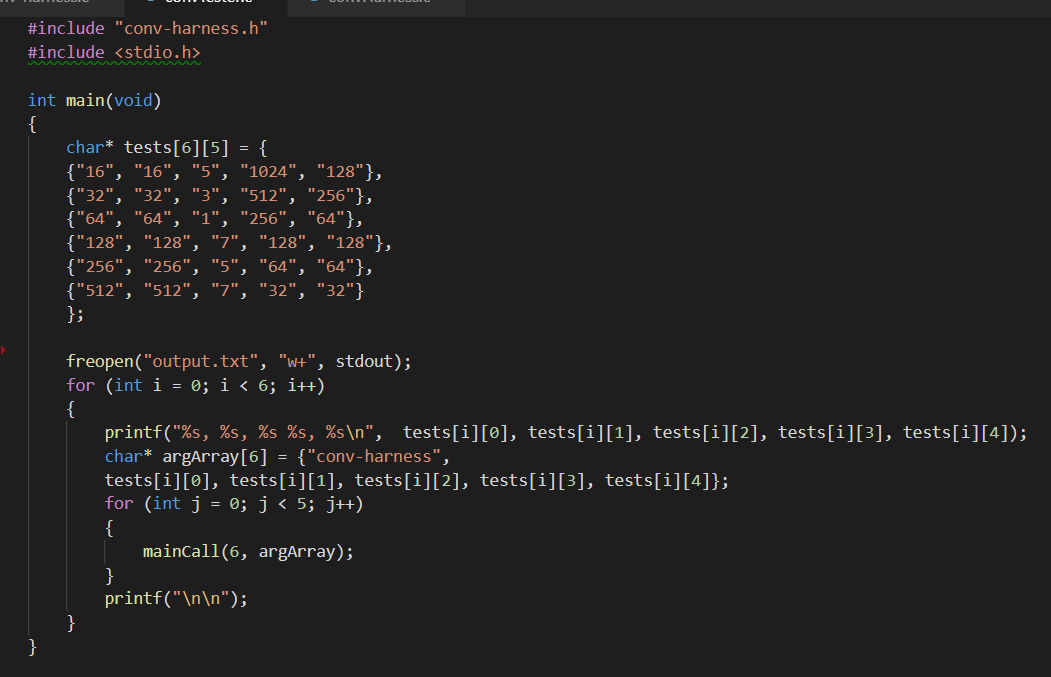
“Team conv time: 292653 microseconds, a 9 speedup! 2766819

COMMENT: sum of absolute differences (0.000000) within acceptable range (0.062500)”

We updated the print statement to the following:



We could then write in a new .c file that would take in the inputs and specify the number of times we wanted to run them. We named this concTester.c and it looked as follows:



This program sent our results to an output file that we could specify the name for. This output file we had could then be imported into a C# program and calculate averages very quickly by parsing the values obtained. This saved us a lot of time as we avoided having to do the calculations manually with a calculator, but still gave us the advantage of getting a more accurate answer for our timings, allowing us to eliminate outliers in our outputs by having an average instead of running it just once.

**Trying parallelization**

**#pragma omp parallel for collapse(3)**

We started looking through the notes to look for what we could do to improve the speed of our program and decided to start by adding in

#pragma omp parallel for

Also written as

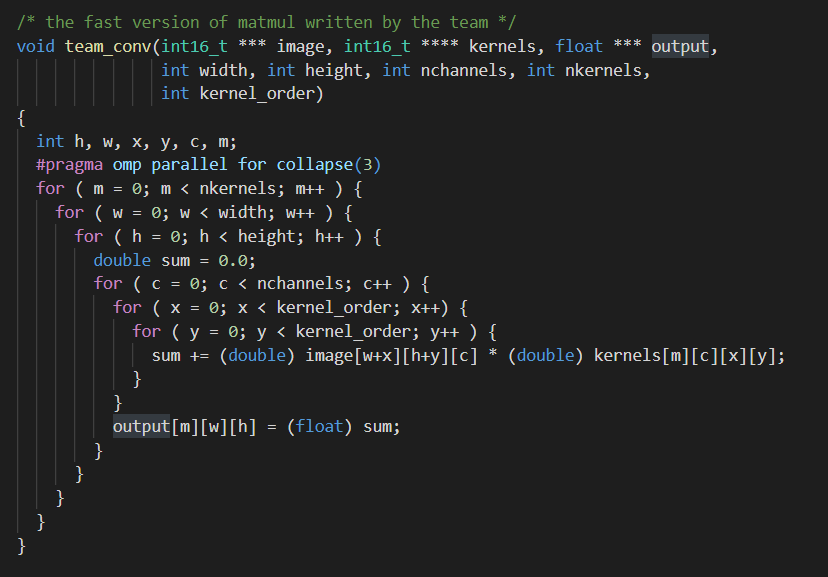
#pragma omp parallel for collapse(1)

and then tested our code. We ran this 50 times and got the averages. Here are the results for it:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Width** | **Height** | **Kernel\_**  **Order** | **Number\_Of**  **\_Channels** | **Kernel\_**  **Number** |  | **No Improvement** | **Parallelized**  **collapse(1)** | |
| 16 | 16 | 5 | 1024 | 128 |  | 2864872 | 164775 | 17 |
| 32 | 32 | 3 | 512 | 256 |  | 4137490 | 242200 | 17 |
| 64 | 64 | 1 | 256 | 64 |  | 442745 | 28149 | 15 |
| 128 | 128 | 7 | 128 | 128 |  | 32516436 | 1774827 | 18 |
| 256 | 256 | 5 | 64 | 64 |  | 19271368 | 1003890 | 19 |
| 512 | 512 | 7 | 32 | 32 |  | 32403120 | 2352950 | 13 |

We then decided to parallelize it further ( #pragma omp parallel for collapse(2) and #pragma omp parallel for collapse(3))and run it again, but this time we only ran it 20 times as we realized running it 50 times was hogging the machine.

This is what our code looked like:



These were the results we got for the new code:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Width** | **Height** | **Kernel\_**  **Order** | **Number\_Of\_**  **Channels** | **Kernel\_**  **Number** | **No Improvement** | **Parallelized collapse(1)** | | **Parallelized collapse(2)** | | **Parallelized collapse(3)** | |
| 16 | 16 | 5 | 1024 | 128 | 2864872 | 164775 | 17 | 162546 | 16 | 163035 | 15 |
| 32 | 32 | 3 | 512 | 256 | 4137490 | 242200 | 17 | 254254 | 14 | 249677 | 14 |
| 64 | 64 | 1 | 256 | 64 | 442745 | 28149 | 15 | 29390 | 12 | 28495 | 12 |
| 128 | 128 | 7 | 128 | 128 | 32516436 | 1774827 | 18 | 1574541 | 20 | 1582756 | 20 |
| 256 | 256 | 5 | 64 | 64 | 19271368 | 1003890 | 19 | 1019410 | 18 | 971688 | 19 |
| 512 | 512 | 7 | 32 | 32 | 32403120 | 2352950 | 13 | 1585599 | 20 | 1618731 | 20 |

At this point we noticed that by parallelizing it down to collapse(3), for smaller inputs, was actually slowing our improvements down very slightly on average. Initially confused about this, we decided to do some research into it and came to a conclusion that it was possible hat for the smaller values, there was a lot of time involved to get set up with the threads, thus leading to it overall being a little slower.

**#pragma omp parallel for collapse(6)**

We wondered if trying to parallelize it all the way down to the inner most for loop would have any improvements or would be worse. We couldn’t parallelize it all the way down by simply updating the line to make it from ‘#pragma omp parallel for collapse(3)’ to ‘#pragma omp parallel for collapse(6)’. We needed to do some altering of the code to be able to parallelize it down to the inner most for loop and here is the snippet of code we came up with that gave us a sum within the acceptable differences for all our tests:

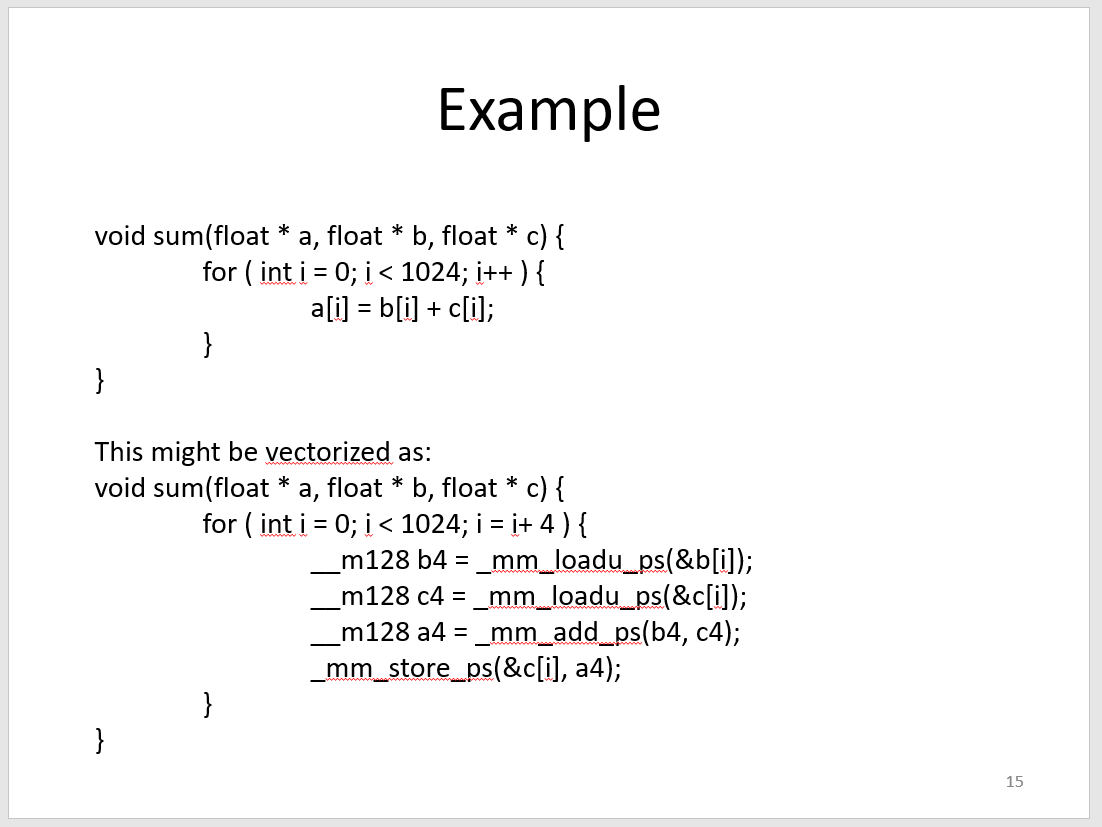


Unfortunately, after running these only a few times, we noticed that the times we got were worse than the original slower program, so we abandoned this attempt and moved on.

**Trying Vectorization**

**\_m128**

We began our first attempt at Vectorization when we found a slide in the notes Vector programming with the following example:



We attempted to vectorize our code. To do this, we needed to reorder the dimensions so that the dimension for number of channels was at the end. It proved a little tricky, but we came up with this piece of code that gave us the results within the accepted difference:

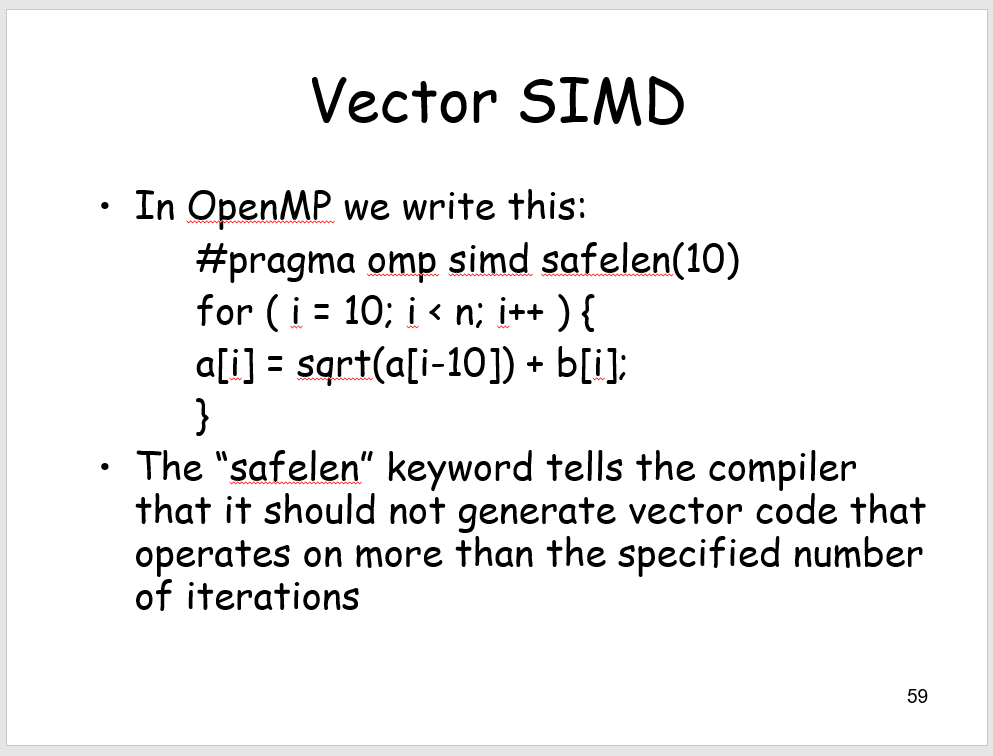
We realized that we not improving our times much at all, maybe 2x occasionally, but the average was no improvement.

**\_m128d**

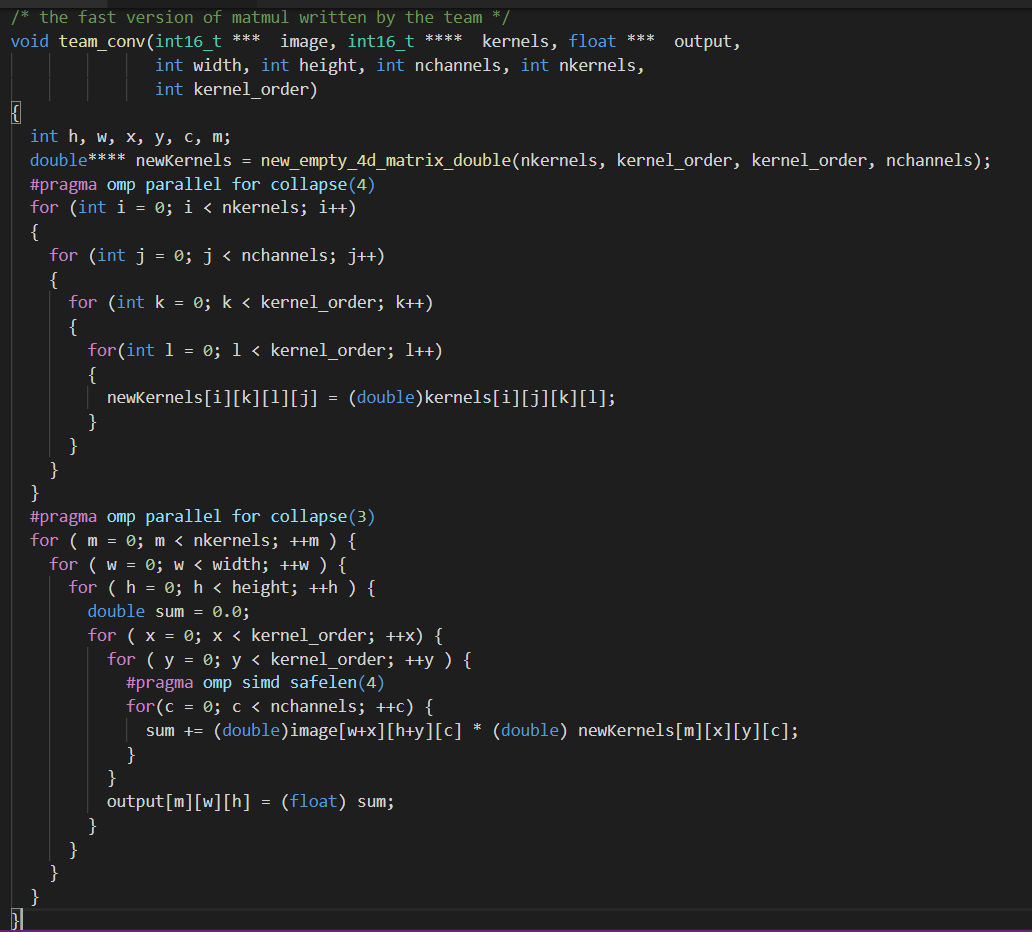
We switched over to using \_m128d to see if this made any difference and if it improved our time in anyway. This required a lot of casting to doubles and adjustments to the code and again, we found that we were not improving our times much at all, maybe 2x occasionally, the same as with \_m128 and on average there was no improvement.

**SIMD Vectorization**

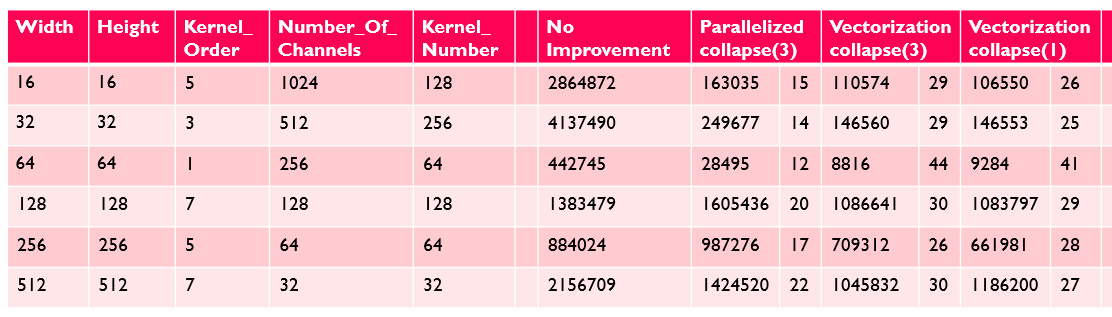
Our attempt at Vectorization to this point had not been very successful. We weren’t happy with it “possibly” improving and therefore made an effort to look through the notes for even more possible ways to fasten our code and we found the following slide:



Based on this slide, we were able to fasten up our code much more. We came up with this code snippet to speed up our code:



We were also curious if collapse(1) would be faster than collapse(3) so we recorded results and came up with the following times for the same outputs as before:

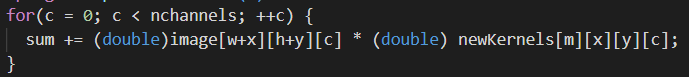


**SIMD Vectorization on reordering**

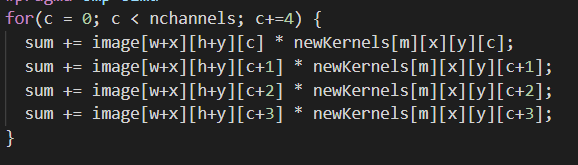
Then we decided that based on our results obtained to this point, that when we were reordering the dimensions so that the dimension for number of channels was at the end, vector SIMD was a better and faster way rather than what we previously had: #pragma omp parallel for collapse(4)

**Unrolling the loop**

We also realized that we could speed up even further by unrolling the loop, i.e. making this:



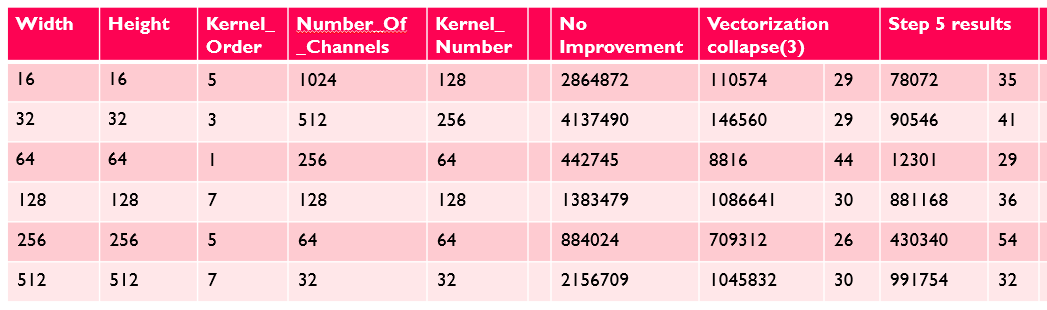
Into this:



And after this point, with the SIMD vectorization applied to the reordering dimensions and the unrolling of the loop, our function looked as follows:



And the results for our standard inputs at this point were (where Step 5 refers to unrolling and SIMD vectorization on reordering):



# Other things we tried

Along the way we tried a few other things too that helped us come to the conclusions we eventually came to and get our final piece of code working as we wanted. We have outlined these briefly below:

**Pre vs Post Increment**

Theoretically, pre-increment should be faster, as with pre-increment, there is no need for a second temp value to be created, whereas with post-increment, there is a value and a temp increment value. This temp increment value gets loaded over the original value. (“Pre-increment is faster than post-increment because post increment keeps a copy of previous (existing) value and adds 1 in the existing value while pre-increment is simply adds 1 without keeping the existing value” - <https://www.includehelp.com/tips/c/pre-increment-is-faster-than-post-increment.aspx>). We noticed no obvious speed-ups when we tried tests on these.

**Restrict vs No Restrict**

“The restrict keyword is a declaration of intent given by the programmer to the compiler. It says that for the lifetime of the pointer, only the pointer itself or a value directly derived from it (such as pointer + 1) will be used to access the object to which it points. This limits the effects of pointer aliasing, aiding optimizations.” (<https://en.wikipedia.org/wiki/Restrict>) Theoretically, restricting the function parameters should result in a speed-up as the compiler will know that the values coming into the function will be restricted to the type labelled, but as with pre-increment, we noticed no obvious speed-ups when we tried tests on these.

**Casting (doubles, ints)**

We wanted to know whether casting would speed up or slow down the time it took to execute our function. We thought that int16\_t should be the quickest as it would remove the need for previous casting, but when we tested this, to our surprise, we actually found that this took a little more time than casting to doubles so we decided to stick with the casting to doubles.

**Hashmaps**

We tried to use hashmaps as from our previous experience with hashmaps, they had been very fast when trying to add and locate/fetch values that were input onto the hashmap with the keys associated, but we found that the setup of the hashmap was taking far too long and completely outweighed the possible, theoretical speedups. We decided that hashmaps were not the way to go to help us with this particular problem.

# Finally, Our demo Code!

Once we had run tests for all the possible improvements we could think of, we remembered that we need to be able to show our work in the demo, and this needs to be optimized for small inputs too. Hoping to get the best we could, we did the following to our code: