2D-Convolution performance

1. Introduction

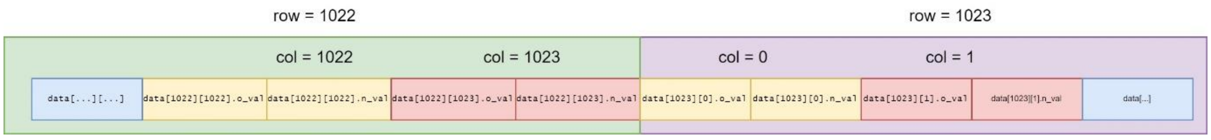
Today, achieving optimal performance in software solutions is crucial for ensuring maximal efficiency in computing systems. Many developers are now looking for solutions within computer architecture to enhance their programs significantly. One such method that developers are employing is cache-conscious programming. In this method, developers design and write programs with an awareness of the memory hierarchy and cache behaviour. This allows programs to minimise cache misses and ultimately maximise the performance of a given program. In this report, we'll investigate two different implementations of performing multiple convolutions on a 1024x1024 input array using a 5x5 filter and determine which implementation is more cache-conscious by their respective runtime.

2. 2D-Convolution

Our program aims to mimic the operation of a 2D-convolution. We will use four nested `for` loops to apply the appropriate filter for each convolution we perform. Additionally, we will use two different data structures to store the results and see which data structure more cache-conscious is based on performance. In both implementations, we will use an additional nested loop to bring the filtered values back into the input data array after each convolution once scaling and saturation are applied. Furthermore, both implementations need 1024 * 1024 * 2 addresses to store and retrieve the data from the input array.

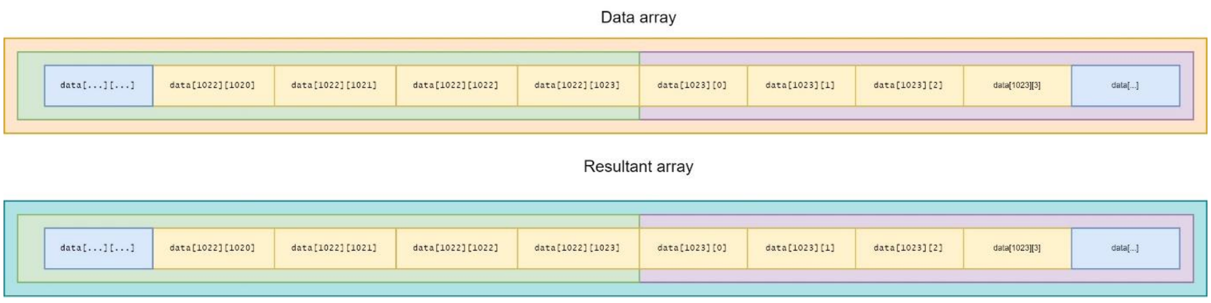
2.1 Struct

In our first implementation, we will store the filtered values with a struct data structure. Essentially, values within our data array will be of type `matrix` and contain two data items, `n_val` to store the input data and `o_val` to store the input data. This means that each value in the 2D-array must occupy two memory addresses for both data items.



2.2 Resultant Array

Our second implementation will use a 1024x1024 array to store the filtered values. Therefore, each value in the 2D array will need only one memory address to store its contents. However, we will also need an additional array to store the filtered values, and therefore we will need to access another 1024x1024 array in memory during storage.



3. Results

To test our implementations, we will use the time command in Linux and compare the user times amongst each program. Additionally, each test will be tested on the University of Auckland UNIX server to ensure consistency within our results.

Our tests show that the 2D-Convolution implementation using structs was consistently quicker than the implementation using a resultant array to store the filtered values. These results can be seen in the table below.

Number of convolutions (#)	Convolution1 time (s)	Convolution2 time (s)
1	0.308s	0.320s
10	1.644s	1.684s
100	14.856s	15.204s
1000	141.296s	150.836s

3.1 Explanation

Our results are reasonably expected as to how C manages memory. In C, the array elements are laid out by row-major order in memory. This means that each block representing a row in the input array will be between its two neighbouring rows in memory. Therefore, assuming the cache is large enough to store each block representing a row, we can expect at least one cache miss for each row in our input array during the convolution algorithm for both programs. However, in our second program, the cache needs to evict the contents of the cache to store the filtered values in the resultant array. Therefore, the program would have an additional cache miss each time it tries to convolve a data point. However, its also needs to be considered that the cache will also evict the input data when it references the kernel array allowing the program to filter each value. Hence, there will be another two cache misses (one to input kernel row, one to recover data row) each time a value gets checked before being added to the convolution summation. Therefore, the loss in performance for program 2 will not be directly caused within the convolution algorithm itself. This is because at the end of each convolution, the algorithm needs to change the input data to the newly convolved values. Hence, we will only have 1 cache miss for program 1, but two cache misses in program 2 because of evicting the resultant array to store the value back into the input array. Therefore, the results we received are expected as more convolutions we do, the more evictions convolutions 2 does and the longer it takes to execute the program.

4. Conclusion

In conclusion, cache conscious programming is crucial for creating efficient programs. By understanding cache behaviour in memory, developers can create programs which aim to mitigate the number of cache misses that occur. This can be seen in the 2D-convolution program where it was shown that the implementation using structs was overall faster than the implementation using a secondary array. This is because during the recovery phase of the algorithm, we found there to be an extra cache miss for our second implementation, therefore slowing down the overall performance of the program. However, it is also worth noting that as computer systems continue to get more advance, programs will continue to get faster due to an increase in cache memory on board. Therefore, developers can become more lenient in their cache-consciousness due to hardware continuing to become more efficient.