**Parallel Multichannel Multikernal Convolution**

**CSU33014**

**Conor Doherty, Aaron Bruce, John Cosgrove**

*General Strategy*

For our approach, as stated in the assignment brief, we paid close attention to writing an efficient algorithm but also to other issues such as locality of data access and multiple available processor cores. Our group came together several times in order to work on our approach, with the end result being an amalgamation of optimisations suggested by each member. We made incremental changes which eventually brought our range of input sizes within the desired epsilon, while also negotiating segmentation faults and other errors along the way.

*Vectorisations*

An important detail in our optimisation strategy was utilising \_\_m128d instructions as opposed to \_m128. It quickly became apparent that \_\_m128 operations resulted in values which were not precise enough and led to the sum of absolute differences being very large, even with the smallest input values. Therefore it was imperative from the offset to use operations such as \_mm\_mul\_pd instead of \_mm\_mul\_ps so ensure accuracy in all calculations. This meant that we had to use vectors of size two instead of size four, which was an inconvenience to say the least but an unavoidable one. In order to work with \_\_m128d we had to ensure that all of the arrays that we worked with were of type double as well. To achieve this, we had to change the type of all of the given functions in the original code from type float and int16 to type double as well as the type of the all of the arrays such as ‘kernels’, ‘image’, ‘output’ etc.

To vectorise the code, we load two consecutive values from the current channel of the current kernel and load in the corresponding values in the image. We then multiply these vectors together and add their total to the current sum. The final sum will be the accumulation of these products across all channels at a particular point in the image for a single kernel. A single element of the output is equalled to one of these sums.Text

Description automatically generated

As each kernel order is negative and our vector size being positive, we have to ensure that we don’t load in the wrong values for the kernel and the image when at the last load of each row of a kernel. To combat this, we load in too many elements from the kernel (past the final index) but set the corresponding value in the image to zero. This nullifies the above problem by ensuring that the product with the wrong value is always zero.

Text

Description automatically generated

*Parallelisation*

*Conclusions*

*Timings*

|  |  |
| --- | --- |
| Input sizes | Execution times |
|  |  |
|  |  |
|  |  |
|  |  |

\_m128d to vectorise the code as \_m128(4 floats) isn’t precide enough and sum of absolute differences ends up being very large even with the smallest input values