Parallel Multichannel Multikernal Convolution

CSU33014

Conor Doherty, Aaron Bruce, John Cosgrove

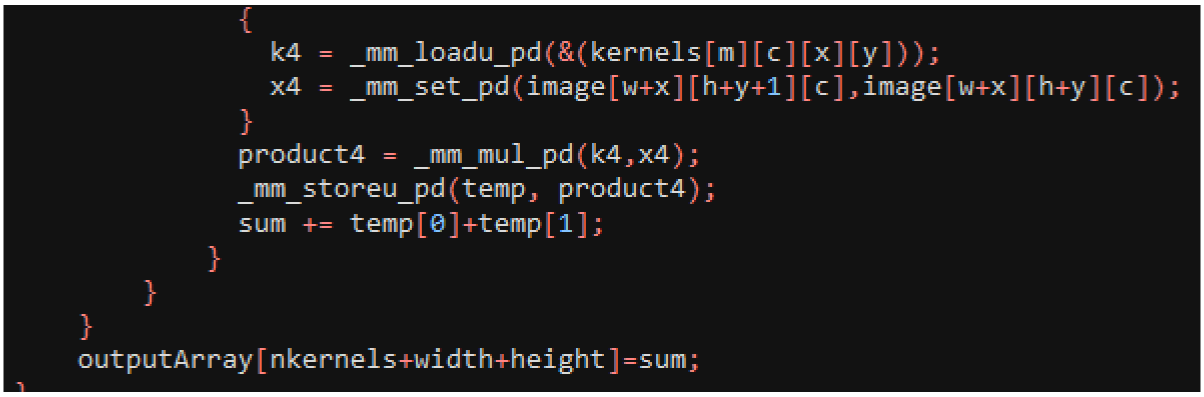
General Strategy

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

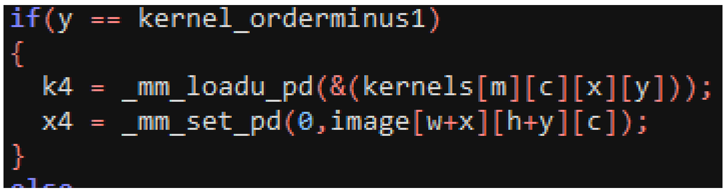
An important detail in our optimisation strategy was utilising __m128d instructions as opposed to _m128. It quickly became apparent that __m128 operations resulted in values which were not precise enough and led to the sum of absolute differences being very large, even with the smallest input values. 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.



```
              {
                k4 = _mm_loadu_pd(&(kernels[m][c][x][y]));
                x4 = _mm_set_pd(image[w+x][h+y+1][c],image[w+x][h+y][c]);
              }
              product4 = _mm_mul_pd(k4,x4);
              _mm_storeu_pd(temp, product4);
              sum += temp[0]+temp[1];
            }
        }
    }
    outputArray[nkernels+width+height]=sum;
```

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.



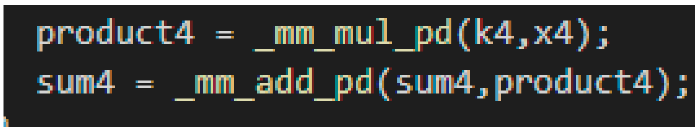
```
if(y == kernel_orderminus1)
{
  k4 = _mm_loadu_pd(&(kernels[m][c][x][y]));
  x4 = _mm_set_pd(0,image[w+x][h+y][c]);
}
```

Parallelisation

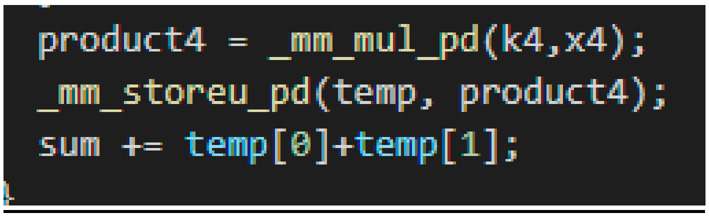
Vectorisation of course made a difference, but our code would be nowhere near fast without parallelisation. A program like this can run much faster if it is split up into threads(short for thread of execution). This means that multiple parts of the program can be run simultaneously. This isn't possible in every program project, but most that are centre around large calculations(such as this one) suit multithreading.

For this project we used OpenMP, an API which can be used for paralleisation of C/C++ code. It provides a simple, flexible interface and is easy to use.

My teammate Aaron provided me with the vectorised code, and it was my job to parallelise it. I started by making changes around the sum4 variable as the current way it was being handled (adding vectors) would not be able to be parallelised properly. I turned sum4 into a double, used a temp variable and _mm_storeu_pd to turn the add the product4 result to the double sum.



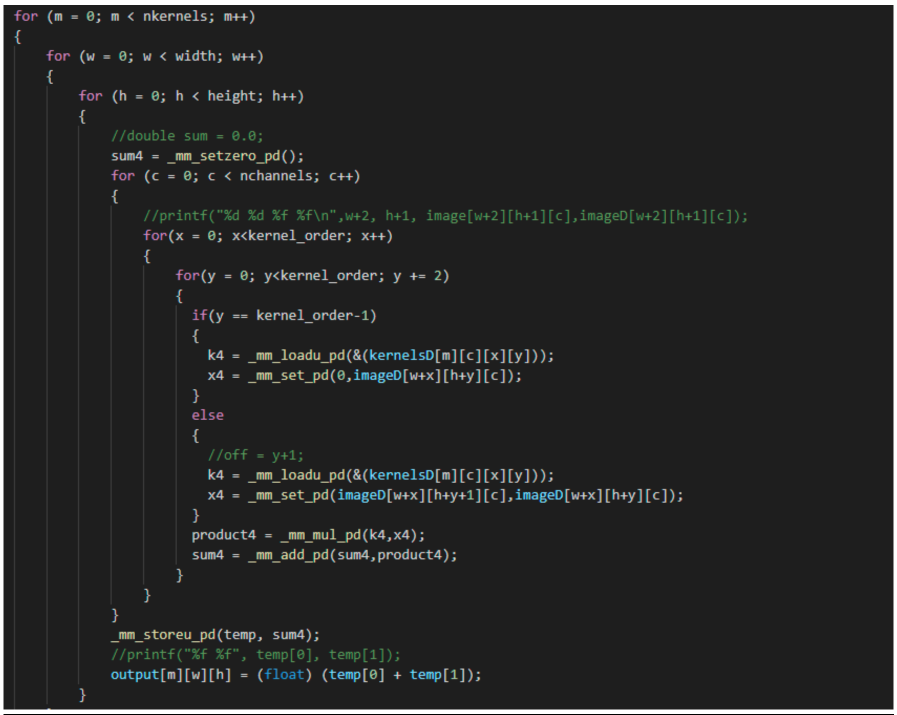
```
product4 = _mm_mul_pd(k4,x4);
sum4 = _mm_add_pd(sum4,product4);
```



```
product4 = _mm_mul_pd(k4,x4);
_mm_storeu_pd(temp, product4);
sum += temp[0]+temp[1];
```

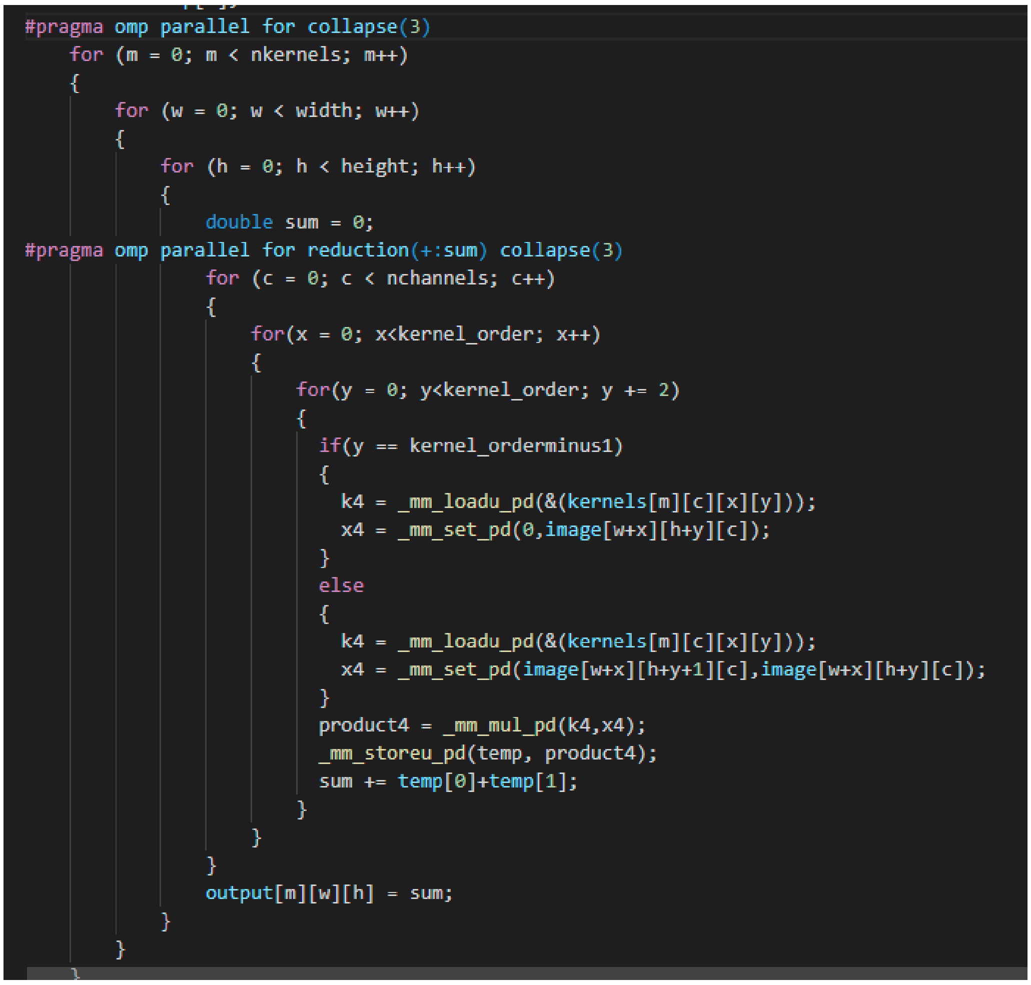
The main part of the program contains 6 loops. I knew I needed to use "omp parallel for" for this but was unsure about the accompanying arguments. I was originally going to use collapse(6) but realised the loops weren't nested in a way to make this possible. I then diced to split this into two instructions, each with collapse(3) . One of the functions had reduction(+:sum), this was extremely important. All of the threads had to be added to sum, and if we tried to do this manually we would receive inaccurate and unreliable results. OpenMP provides a nifty way to do this, reduction, this takes care of the locking etc of the variable. This ensures we get the same, correct answer every time.

Loops before



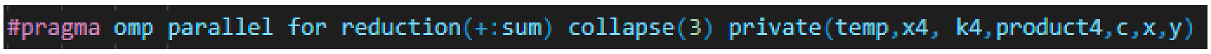
```c
for (m = 0; m < nkernels; m++)
{
    for (w = 0; w < width; w++)
    {
        for (h = 0; h < height; h++)
        {
            //double sum = 0.0;
            sum4 = _mm_setzero_pd();
            for (c = 0; c < nchannels; c++)
            {
                //printf("%d %d %f %f\n",w+2, h+1, image[w+2][h+1][c],imageD[w+2][h+1][c]);
                for(x = 0; x<kernel_order; x++)
                {
                    for(y = 0; y<kernel_order; y += 2)
                    {
                      if(y == kernel_order-1)
                      {
                        k4 = _mm_loadu_pd(&(kernelsD[m][c][x][y]));
                        x4 = _mm_set_pd(0,imageD[w+x][h+y][c]);
                      }
                      else
                      {
                        //off = y+1;
                        k4 = _mm_loadu_pd(&(kernelsD[m][c][x][y]));
                        x4 = _mm_set_pd(imageD[w+x][h+y+1][c],imageD[w+x][h+y][c]);
                      }
                      product4 = _mm_mul_pd(k4,x4);
                      sum4 = _mm_add_pd(sum4,product4);
                    }
                }
            }
            _mm_storeu_pd(temp, sum4);
            //printf("%f %f", temp[0], temp[1]);
            output[m][w][h] = (float) (temp[0] + temp[1]);
        }
```

Loops after

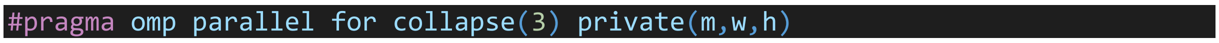


```c
#pragma omp parallel for collapse(3)
    for (m = 0; m < nkernels; m++)
    {
        for (w = 0; w < width; w++)
        {
            for (h = 0; h < height; h++)
            {
                double sum = 0;
#pragma omp parallel for reduction(+:sum) collapse(3)
                for (c = 0; c < nchannels; c++)
                {
                    for(x = 0; x<kernel_order; x++)
                    {
                        for(y = 0; y<kernel_order; y += 2)
                        {
                          if(y == kernel_orderminus1)
                          {
                            k4 = _mm_loadu_pd(&(kernels[m][c][x][y]));
                            x4 = _mm_set_pd(0,image[w+x][h+y][c]);
                          }
                          else
                          {
                            k4 = _mm_loadu_pd(&(kernels[m][c][x][y]));
                            x4 = _mm_set_pd(image[w+x][h+y+1][c],image[w+x][h+y][c]);
                          }
                          product4 = _mm_mul_pd(k4,x4);
                          _mm_storeu_pd(temp, product4);
                          sum += temp[0]+temp[1];
                        }
                    }
                }
                output[m][w][h] = sum;
            }
        }
```

Next I had to add private variables to both of my OpenMP instructions. A private variable ensures that each thread has it’s own instance of it. This stops plethora of issues which would occur if multiple threads were writing and trying to read from the same place, it also improves performance.



```
#pragma omp parallel for reduction(+:sum) collapse(3) private(temp,x4, k4,product4,c,x,y)
```



```
#pragma omp parallel for collapse(3) private(m,w,h)
```

My final touch was to turn kernel_order-1 into a constant. Many compilers could spot something like this, but I wanted to ensure that this value would not be recalculated for each thread.

Now I was done, I tested my code and was pleased to see that it worked.

Conclusions

Overall this project taught us a great deal about writing an efficient multichannel multikernal routine. We were able to apply our previous knowledge of vectorization from the first assignment and also put into practice our newly learned Openmp techniques. We faced challenges in attaining suitably precise results from our routine and handling different orders of kernel but the team came together and worked well in parallel(pun intended) to achieve a strong level of optimisation.

Timings

Input sizes	Speedup	Execution times(microseconds)
16 16 5 256 256	17.66x	105252
64 64 5 256 256	16.67x	105252
128 128 5 256 256	21.13	3692977
256 256 5 256 256	22.99x	13416523