ConcurrentML

HEDGEHOG Michael McGuinness(16322635), Liam Sherlock(17308853)

Clearing the output.

There are two modifications in this section. It started out as 3 forloops with a $O(m * h^2)$ runtime. The h^2 is derived from the fact that given the bounds of the assignment the hight and width will always be equal.

```
for (m = 0; m < nkernels; m++)
    {
        for (h = 0; h < height; h++)
        {
            for (w = 0; w < width; w++)
            {
                output[m][h][w] = 0.0;
            }
        }
    }
```

The optimizations we applied were two fold.

1. We used Open MP to paralelize the whole process.
2. Instead of using 3 forloops to set each slot as 0 we modified it to use 2 for loops and setting the resulting 1d matrix to zero using memset.

Intialy we tried using memset to set the entire array but unfortunatly due to the way the 3d array is created you can't just set the memory in one go without causing errors.

```
#pragma omp parallel for collapse(2)
    for (m = 0; m < nkernels; m++)
        for (h = 0; h < height; h++)
            memset(output[m][h], 0, width * sizeof(float))\
```

The resulting code has a final efficiency of just $O(m * h)$

Running the calculations.

In this sections there are also two main modifications to the code. But looking at the base code it is quite ineficinet as not only is it 6 nested forloops it contains zero parallelization.

```
for (w = 0; w < width; w++)
{
    for (h = 0; h < height; h++)
    {
        double sum = 0.0;
        for (x = 0; x < kernel_order; x++)
        {
            for (y = 0; y < kernel_order; y++)
            {
                struct sparse_matrix *kernel = kernels[x][y];
                for (m = 0; m < nkernels; m++)
                {
                    for (index = kernel->kernel_starts[m]; index < kernel->kernel_starts[m + 1];
                    {
                        int this_c = kernel->channel_numbers[index];
                        assert((this_c >= 0) && (this_c < nchannels));
                        value = kernel->values[index];
                        output[m][h][w] += image[w + x][h + y][this_c] * value;
                    }
                } // m
            }     // y
        }         // x
    }             // h
}                 // w
```

As mentioned above there are two key changes.

1. We used Open MP to paralelize the whole process.
2. To optimize for open mp we changed the order of the for loops to maximize the use of threads and allowing us to use the no wait clause.

Intialy we tried to reduce the amount of forloops by compining the loops however we found this to be slowing as even though you are using one loop you have to decript the different xy hw values using divide and modulo. In the end we found this to be slower over all.

```
#pragma omp parallel
{
    for (x = 0; x < kernel_order; x++)
        for (y = 0; y < kernel_order; y++)
            for (m = 0; m < nkernels; m++)
                for (index = kernels[x][y]->kernel_starts[m]; index < kernels[x][y]->kernel_star
                {
#pragma omp for collapse(2) nowait
                    for (h = 0; h < height; h++)
                        for (w = 0; w < width; w++)
                            output[m][h][w] += image[w + x][h + y][kernels[x][y]->channel_number
                }
}
```

Open MP.

```
omp_set_num_threads(32);
```

For the assignment we found that when using stoker 32 threads was the optimal amount of threads any more and we found that it slowed it down.

For the zeroing of the output we used.

```
#pragma omp parallel for collapse(2)
```

It collapse the nested for loops so that the instantiations may run in parallel using the available threads

For the computations we used

```
#pragma omp parallel
```

to compute the multichannel, multikernel convolution. We set a parallel region to allow the later nowait to work

Finaly we used

```
#pragma omp for collapse(2) nowait
```

Which collapsed the for loops that are non dependent, allowing the operation to run in parallel. We used nowait, because later iterations in the parallel region are not dependent on the results of this, therefore allowing the threads to be fully used at all times.