ConcurrentML

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

Clearing the Output.

There are two modifications in this section. It started out as 3 for loops 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. Instead of using 3 for loops to set each slot as 0 we modified it to use 2 for loops and setting the resulting 1d matrix to zero using memset.
2. We used Open MP to parallelize the whole process.

Intially we tried using `memset()` to set the entire array but unfortunatly due to the way the 3D array is created, often using linked lists, 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 section, there are also two main modifications to the code. Looking at the base code it is quite inefficient since it not only is it 6 nested for loops, 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]; index++)
                    {
                        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 a few key changes:

1. The set of for loops is placed into a parallelisable area using `#pragma omp parallel`.
2. The nesting of the for loops is changed. It was initially changed to use and set a variable only when it is actually needed. This makes the generally smaller `kernel_order` for loops be placed more, as well as `nkernels`. Then the `height` and `width` for loops are placed deep within the nesting.
3. We parallelised the `height` and `width` for loops using `#pragma omp for collapse(2)`. This is advantageous for a few reasons. In order to properly parallelise nested for loops, we need to make sure that the for loops that are being nested do not rely on each other, and this is the case for these two for loops. The second advantage is that these are generally the largest loops, i.e. they have the most amount of iterations out of any of the other loops being nested. This allows for the bulk of the work to be parallelised.
4. The `nowait` clause is used to parallelise these for loops. This is possible because the future operations done within the loops is not affected by other iterations of the external loops being done first. It also allows the available threads to be fully used at all times, rather than waiting for each set of loops to be completed first.

Intially, we tried to reduce the amount of for loops by combining the loops. However, we found this to be slowing the program down, as even though you are using one loop you have to decrypt the different x, y, h, w values using divide and modulo. In the end we found this to be slower over all. We also condidered using `#pragma omp for collapse(3)` for the first three nested for loops, intead of using the internal case, however, we decided against this because of how the height and width are far larger in most cases. On top of this, we look into vectorising the operation being done, and found no appropriate solution because of how 3-dimensional dimensional arrays are stored in c.

```
#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_starts[m + 1]; index++)
                {
#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_numbers[index]] * kernels[x][y]->values[index];
                }
}
```

Open MP.

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 otherwise. The amount of threads was determined using `omp_set_num_threads(32)`.

For the zeroing of the output we used `#pragma omp parallel for collapse(2)`. It collapses the nested for loops so that the internal opreations 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.

Finally, 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, allowing the threads to be fully used at all times.

Test Results.

Image Width	Image Height	Kernel Order	Number of Channels	Number of Kernels	Non-Zero Ratio	Team Execution Time (μs)
300	300	3	16	512	20	581986