CSU33014 Lab 2

Parallel Multichannel Multikernel Convolution Report

Emmet McDonald, Emmet Morrin, & Maryann Foley

The Code

Our approach to parallelising and speeding up the convolutional layers was likely as simple as most other pairs' to start with. We knew that understanding a **pragma omp parallel for** would be one of the main sources of potential speed up, as we had been taught that OpenMP was *the* way to parallelise code, and **pragma omp parallel for** was ideal for cases where we'd be faced with lots of for loops.

However before we go too in-depth into how we got that command working, let's look at some of the smaller changes we enacted, that hopefully added up to give us an edge on other groups.

```
void multichannel_conv(float *** image, int16_t **** kernels,
                       float *** output, int width, int height,
                       int nchannels, int nkernels, int kernel_order)
{
  int h, w, x, y, c, m;

  for ( m = 0; m < nkernels; m++ ) {
    for ( w = 0; w < width; w++ ) {
      for ( h = 0; h < height; h++ ) {
        double sum = 0.0;
        for ( c = 0; c < nchannels; c++ ) {
          for ( x = 0; x < kernel_order; x++) {
            for ( y = 0; y < kernel_order; y++ ) {
              sum += image[w+x][h+y][c] * kernels[m][c][x][y];
            }
          }
          output[m][w][h] = (float) sum;
        }

      }
    }
  }
}
```

"Slow but correct" version of matmul, for reference

The first such change was very trivial, moving the **output[m][w][h] = (float) sum** outside the loop it was originally in, as in the original function, the same value would be applied to the same point in memory an extra *nchannels* times.

Other trivial changes enacted during the development process involved avoiding 'temp' variables, or any other variables that just existed for ease of understanding the code, as well as making as many variables constant as possible. These changes didn't greatly affect the overall speed of the program, but any speed up is good.

```
//new kernels rearranged so that it's easier to grab data in sequence, improves locality
  float ****kernelsRearranged = new_empty_4d_matrix_float(nkernels, kernel_order,
kernel_order, nchannels);

  //parallelise the rearranging, this doesn't improve performance significantly, most of the
work is in the next part
  #pragma omp parallel for
  for (int m = 0; m < nkernels; m++)
  {
    for (int x = 0; x < kernel_order; x++)
    {
      for (int y = 0; y < kernel_order; y++)
      {
        for (int c = 0; c < nchannels; c+=4)
        {
          //convert to float and rearrange for c to be last
          const __m128 temp = _mm_setr_ps((float) kernels[m][c][x][y], (float)
kernels[m][c+1][x][y], (float) kernels[m][c+2][x][y], (float) kernels[m][c+3][x][y]);
          _mm_storeu_ps(&kernelsRearranged[m][x][y][c], temp);
        }
      }
    }
  }
```

Creation and population of the float[][][][] **kernelsRearranged**

The next change was more significant than simply moving a variable assignment outside of a for loop. This change involved rearranging the **kernels** 4-d array so that the int **c** was the fourth dimension, and not the second, which improves locality. In order to enact this change, a new set of 4 consecutive for loops was created and parallelised using **pragma omp parallel for**. In order to save further time, the *int16_t*s that made up the original **kernels** array were cast into *float*s, which are easier, and faster, to work with.

```
  //precompute a limit on combined for loop
  const int mwhlimit = nkernels*width*height;
  __m128 vec_image, vec_kernel;
  //the three m, w, h loop is combined into one single loop, and allows it all to be
parallelised at once, meaning more threads. Collapse (3) omp option is slower, but more
consistent in its speed
  #pragma omp parallel for private(vec_image, vec_kernel)
  for (int mwh = 0; mwh < mwhlimit; mwh++ ) {
    //using 1D index, calculate the 3D coordinate of m, w, and h, const where possible
```

```
increases speed
   const int m = mwh / (width * height) % nkernels;
   const int w = (mwh/height) % width;
   const int h = mwh % height;
   //create 4 total local vector double sums
   __m128d vec_sum[2] = {_mm_setzero_pd(), _mm_setzero_pd()};
```

Pre-computation & optimization for the **nkernels**, **width**, and **height** for loops

Now we approach code resembling the original function, however there is already a significant change to be seen. Where, before, **nkernels**, **width**, and **height** had their own consecutive for loop, here we have combined them into a **mwhlimit** loop, to allow more threads to be created when we called **pragma omp parallel for**, although **m** (**nkernels**), **w** (**width**), and **h** (**height**) do have to be re-derived for usage later in the function.

```
   for (int x = 0; x < kernel_order; x++) {
     for (int y = 0; y < kernel_order; y++ ) {
       for (int c = 0; c < nchannels; c+=4 ) {
         //load image into vector
         vec_image = _mm_loadu_ps(&image[w+x][h+y][c]);
         //load the kernels into vector
         vec_kernel = _mm_loadu_ps(&kernelsRearranged[m][x][y][c]);
         //multiply 4 images by 4 kernels using vectors
         vec_image = _mm_mul_ps(vec_image, vec_kernel);

         //convert the lower 2 floats of mul to double and add to sum total
         vec_sum[0] = _mm_add_pd(vec_sum[0], _mm_cvtps_pd(vec_image));
         //rearrange mul so that the upper two floats are now the lower two floats
         vec_image = _mm_shuffle_ps(vec_image, vec_image, _MM_SHUFFLE(1, 0, 3, 2));
         //convert the lower 2 floats of mul to double and add to sum total
         vec_sum[1] = _mm_add_pd(vec_sum[1], _mm_cvtps_pd(vec_image));
       }
       //note nchannels should always be 2^n and >32, therefore it will always be %4, so no
remainder sums necessary
     }
   }
   //horizontally add all 4 doubles together to get a final sum
   vec_sum[0] = _mm_hadd_pd(vec_sum[0], vec_sum[1]);
   vec_sum[0] = _mm_hadd_pd(vec_sum[0], vec_sum[0]);

   //load final sum to output
   output[m][w][h] = (float) _mm_cvtsd_f64(vec_sum[0]);
   //note, has been moved out an array, since the dependant variables, m, w, and h do not
use c, and the output is being repeatedly overwritten unnecessarily before
  }
}
```

Vectorisation for the **kernel_order** and **nchannels** for loops & population of **output**

Finally, we get to the last few for loops, a pair of consecutive for loops for **kernel_order** and a newly vectorised for loop for **nchannels** (To note, the **nchannels** loop is now 'inside' the **kernel_order** loops, both to improve locality, and because **nchannels** is guaranteed to be divisible by 4).

Vectorisation is our final method of improving speed, and is fairly easy to pull off here, as this part of the original program was exclusively arithmetic-based. After the kernels and other data is multiplied, added and shuffled together into the _m128d array **vec_sum**, as per the original function's arithmetic, it is cast back into a *float*, and the relevant section of **output** is populated.

The Results

The commands for each of the tests were as follows:

Test 1:	./conv 16 16 1 32 32
Test 2:	./conv 64 64 3 32 32
Test 3:	./conv 128 128 3 64 64
Test 4:	./conv 256 256 5 256 256
Test 5:	./conv 512 512 7 1024 1024

Tests 1 to 4 were all run 5 times for both default and our program. The numbers in our results table are the averages of those tests. Test 5 was only run once for each. All of our tests were run on stoker.

The results of our tests:

	Test 1 - Minimum	Test 2 - Low	Test 3 - Medium	Test 4 - High	Test 5 - Maximum
Our program	0.012s	0.015s	0.035s	2.313s	425.285s
Default	0.003s	0.211s	2.015s	268.366s	DNF
Times difference	4x slower	14x faster	58x faster	116x faster	DNF

For test 5 (Maximum), the default program did not finish after running for 30 minutes, although based on our test being over 100x faster in the previous test, we estimate that it would take 11.8 hours to run.

When running the 5 trials for each test for both the default and our program, we noticed that the results of the default program were more consistent then our program, which had greater deviation.

How to Run

Inside of the folder, run:

```
make
```

Then

```
./conv <img width> <img height> <kernel order> <n channels> <n kernels>
```

The constraints of img width and height are 16-512, kernel order is 1, 3, 5, or 7, n channels and n kernels are both 32-2048, where the value is a power of 2^n.

Sources:

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

https://arxiv.org/pdf/1704.04428.pdf

https://medium.com/apache-mxnet/multi-channel-convolutions-explained-with-ms-excel-9bbf8eb77108

Lecture slides