C Okafor, 20332400

CSU33014 Lab 2

This document will explain the steps I took to improve upon the existing algorithm for creating a parallel multichannel multikernel convolutional neural network (**CNN**).

Initial Algorithm

The algorithm we were initially given consisted of a sextuple-nested for loop with the intention of summing a value in the innermost loop. The loops iterate between 0 and one of five values which are listed below in order:

- [`nkernels`], the number of kernels;
- [`width`], the width of the image;
- [`height`], the height of the image;
- [`nchannels`], the number of channels;
- [`kernal_order`], the order of kernels.

Between the second and third loops, a new value [`sum`] is instantiated as the double value 0.0, and after the fifth for loop, a value in a 3D array is set to the value of the sum after being cast to a float. The sum is the multiplication of two array accesses; a 3D array and a 4D array.

The for loops are all mutually independent; they only interact with each other in the array accesses. The loop iterators are declared before the loops and are assigned upon entering them.

Initial Code

```
/* the slow but correct version of matmul written by David */
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;
        }
      }
    }
  }
}
```

Improved Algorithm

To improve upon the algorithm, I started by doing what the for loops did and instantiated [`sum`] outside the loops, and only assigned it its value inside them. I also removed the cast to float. This alone didn't appear to have much impact on the overall speed of the program.

The second thing I did was manually collapse the width and height (2nd and 3rd) loops by multiplying the width and height limits in a single for loop. To later determine the width or height for any given iteration, I could simply divide the result by the height to find the width, and modulo (%) the result to find the height.

Finally, I used OpenMP multiprocessing to speed up the code. The single line of code I added parallelised the for loops, and reduced the summation to individual threads so they shared the value. I then collapsed outer two for loops – now the [`for m`] and [`for wh`] loops – using the [`collapse`] attribute, which served to speed up the processing time in addition to the previous methods.

Improved Code

```
void stu_mulchan_conv(float ***image, int16_t ****kernels,
             float ***output, int width, int height,
             int nchannels, int nkernels, int kernel_order) {
  int x, y, c, m, wh, mc;

  double sum;
  #pragma omp parallel for reduction(+:sum) collapse(2)
  for (m = 0; m < nkernels; m++) {
    for (wh = 0; wh < width*height; wh++) {
        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[wh/height + x][wh%height + y][c] *
                         kernels[m][c][x][y];
          }
        }
        output[m][wh/height][wh%height] = (float)sum;
      }
    }
  }
}
```

Times

All times recorded below are the average of 5 consecutive runs with the same values (width, height, kernel order, etc.). Each row in the table will take the form:

[width] | [height] | [kernel order | [no. channels] | [no. kernels]	[time in microseconds]	[time in microseconds]

Parameters	**Original (μs)**	**Improved (μs)**
16 | 16 | 1 | 32 | 32	1,541	5,170
16 | 16 | 3 | 32 | 32	5,417	6,183
16 | 16 | 5 | 32 | 32	15,185	6,299
16 | 16 | 7 | 32 | 32	48,289	7,513
64 | 64 | 1 | 32 | 32	52,196	7,274
64 | 64 | 3 | 32 | 32	186,083	10,901
64 | 64 | 5 | 32 | 32	303,172	17,159
64 | 64 | 7 | 32 | 32	387,436	26,617
128 | 128 | 1 | 128 | 128	1,591,088	86,700
128 | 128 | 3 | 128 | 128	5,644,236	315,546
128 | 128 | 5 | 128 | 128	13,888,209	585,476
128 | 128 | 7 | 128 | 128	24,229,362	1,151,520
256 | 256 | 1 | 32 | 32	472,483	26,712
256 | 256 | 3 | 32 | 32	1,402,520	100,666
256 | 256 | 5 | 32 | 32	3,519,758	271,433
256 | 256 | 7 | 32 | 32	5,924,828	347,547
16 | 16 | 1 | 1024 | 1024	1,650,229	79,453
16 | 16 | 3 | 1024 | 1024	6,084,584	479,797
16 | 16 | 5 | 1024 | 1024	18,484,264	1,021,479
16 | 16 | 7 | 1024 | 1024	38,921,471	17,781,520

16 | 32 | 1 | 64 | 128	24,413	7,033
16 | 32 | 3 | 64 | 128	87,590	11,371
16 | 32 | 5 | 64 | 128	217,285	18,230
16 | 32 | 7 | 64 | 128	419,212	28,102
64 | 64 | 1 | 256 | 256	1,584,680	87,328
64 | 64 | 3 | 256 | 256	5,645,861	308,103
64 | 64 | 5 | 256 | 256	14,197,188	852,708
64 | 64 | 7 | 256 | 256	33,526,310	1,480,247
512 | 512 | 1 | 512 | 512	416,258,557	10,480,946

Conclusion

From the listed times, it is clear that the new algorithm has significant boosts over the original one. When the parameters are small, the difference in performance is lessened; however, this difference becomes much larger when the parameters are also larger.

In the improved algorithm, when the number of kernels and channels are large, the growth of processing time between iterations of kernel order is nigh-exponential. This is the most evident in the fifth example (using parameters 16 | 16 | X | 1024 | 1024), where it is assumed that the improved algorithm would take longer than the original algorithm were the kernel order any higher.

Inversely, where the width and height of the image are large, the growth in processing time is much slower, appearing almost linear as in the fourth example (using parameters 256 | 256 | X | 32 | 32). In contrast, the original algorithm has massive growth between tests.

The point of using so few changes to the original code was to show how easy it is to improve the performance of an algorithm using OpenMP, as well as manual changes to flow control. This is dependent on the target system, as performing these same tests on my personal laptop (Intel i7-1260P CPU) using Linux resulted in times being roughly 1.5x faster/slower than on the Stoker lab machines. Regardless, multithreading can provide a clear boost to performance when given attention, as has been proven above.