Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU33014 Concurrent Systems
CPU Image Convolution Optimisation

Ted Johnson, TCD 19335618
13th of April, 2022

Contents

1 Description of Problem

To write an optimised multi-channel multi-kernel convolution routine in C/C++ utilising the various strategies discussed in class. Provided is an unoptimised but correct function:

```
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;
        }
      }
    }
  }
}
```

The routine will be required to operate on the following input sizes:

- `width`: 16 .. 512
- `height`: 16 .. 512
- `kernel_order`: 1, 3, 5, or 7
- `nchannels`: 32 .. 2048 powers of 2
- `nkernels`: 32 .. 2048 powers of 2

2 Solution & Implementation

2.1 Multiprocessing

The most obvious improvement can be made by utilising our target machines four processors with eight cores each capable of two-way simultaneous multithreading. This problem is embarrassingly parallelizable as every input kernel is only used to write to its own output image. It is possible for two threads to both read from the input image, operate with their own kernels and then write to their own output buffers without ever needing to synchronise.

I have chosen to use OpenMP to implement this with minimal effort. The outermost loop is split up by OpenMP between at least 32 threads, each which operate with a different input kernel on the same input image. As this input image is never written to and each thread always writes to their own output buffer, there is no need for thread synchronisation and false sharing is massively limited.

2.2 Caching

To efficiently utilise caching, we must first understand how the input data and output buffers are laid out in memory. On first inspection, the function parameters seem like arbitrary multi-dimensional arrays. These are not particularly efficient to access sequentially as members of the lower dimensions are not guaranteed to be placed in contiguous memory. However, we can observe that `new_empty_4d_matrix_int16` is carefully constructed to produce a single, contiguous array of data (`mat3`) with higher dimensional arrays only serving as abstractions to the fundamentally one-dimensional array.

This is excellent news for our optimised function, as we can take advantage of all input and output data being completely contiguous in our implementation. We can redesign our algorithm to access memory to maximise cache locality with sequential reads. Unfortunately, if we treat our input data as one-dimensional arrays, the input image becomes a 1D array of channels while the input kernels become 1D arrays of pixels. To fix this issue, each thread must first convert its kernel from pixel-first to channel-first. I suppose this could be thought of as a transposition of the input.

With the input image and kernel data now aligned contigiously channel-first in memory, we can reorder our innermost for loops such that we maximise indexing both arrays sequentially while computing the convolution of one of the output pixels. This turns out to be channel-first, then kernel height and finally kernel width. By doing so, the kernel and output buffer are only sequentially accessed for extreme cache locality while the input image has at least every channel within every pixel read sequentially with an increased likelihood of consecutive pixels being read sequentially as well.

While this seems like a lot of work for arguably not much of a performance increase (without multithreading, I was seeing maybe a two times execution time decrease), by structuring our routine in such a cache effective way allows us to take full advantage of Intel's SSE instruction set in the next section.

2.3 Vectorization

We can take advantage of the assumption that the number of channels in every pixel is at least 32 and always a power of 2. This allows us to operate on 32 channels at once without having to worry about handling remaining channels. Intel's SSE instructions allow us to operate on four of our single-precision floating point input values at once.

Unfortunately, we cannot simply operate and write out our input data. To ensure equivalence with the unoptimised function, we must use 16 bit integer values as kernel input and must aggregate the results of operations with a double-precision floating point value.

The most effective way I found to handle multiplication of four floats and 16 bit integers was to simply convert all the 16 bit integers to floats before-hand. Luckily for us, these 16 bit integer values were already being transposed from their input array anyway so it wasn't much more effort to cast these inputs into their floating point equivalents. Additionally, these converted values were memory aligned to 16 bits so that it was easier for SSE instructions to quickly load these values into vector registers.

To sum these four values at one, we have to use four double-precision floating point values. We can use two SSE vectors to hold all four of these values. Now we only have to convert from single-precision to double-precision when adding these values within the innermost loop.

Finally, after we've completed the summation of multiplications for a single output pixel, we simply have to sum these four double-precision values together and write the result as a single-precision value to the output buffer.

2.4 Possible Improvements

The algorithm performs poorly when operating on small input sizes. The unoptimised algorithm is able to execute around four times faster on the most trivial input (16x16 image, 32 channels, 32 kernels of order 1). This is very likely due to the initial overhead of creating threads with OpenMP outweighing the benefit of .

As such, an obvious improvement would be to combine both algorithms into a hybrid solution where smaller inputs are handled by the simple single-threaded algorithm while larger inputs are handled by the optimised multi-threaded algorithm. Ideally, even the simple algorithm could be optimised with vectorisation and many more measurements could be taken to determine exactly when it is optimal to include multithreading in the algorithm.

3 Execution Times

3.1 List of Execution Times

Below is a list of standard input sizes and the resulting execution times of the optimised function. I have colour-coded input sizes to try to visually present the effect different inputs have on the execution time. In the next section, we will be comparing the execution times of the unoptimised and optimised functions.

Width	Height	Order	# Channels	# Kernels	Execution Time	
16	16	1	32	32	10720 µs	*~ 0.01 sec*
128	128	3	128	32	43079 µs	*~ 0.04 sec*
128	128	3	128	128	79384 µs	*~ 0.08 sec*
128	128	3	128	512	361766 µs	*~ 0.36 sec*
128	128	3	128	2048	1067824 µs	*~ 1.07 sec*
128	128	5	32	128	81056 µs	*~ 0.08 sec*
128	128	5	128	128	240088 µs	*~ 0.24 sec*
128	128	5	512	128	816402 µs	*~ 0.82 sec*
128	128	5	2048	128	2363524 µs	*~ 2.36 sec*
128	128	7	32	32	63874 µs	*~ 0.06 sec*
128	128	7	128	128	496325 µs	*~ 0.50 sec*
128	128	7	512	512	3701848 µs	*~ 3.70 sec*
128	128	7	2048	2048	84396910 µs	***~ 84.4 sec***
512	512	1	512	512	4998231 µs	*~ 5.00 sec*
512	512	3	512	512	13494469 µs	*~ 13.5 sec*
512	512	5	512	512	40412638 µs	*~ 40.4 sec*
512	512	7	512	512	60057552 µs	***~ 60.1 sec***

3.2 Comparison of Execution Times

On larger inputs, it's clear the optimised function performs far better than the unoptimised function. It was impractical for me to measure the execution time of the unoptimised function for the most extreme inputs, so I have plotted trends to try to extrapolate the results. As mentioned before, the unoptimised function was generally able to perform better on the extremely small input sizes.

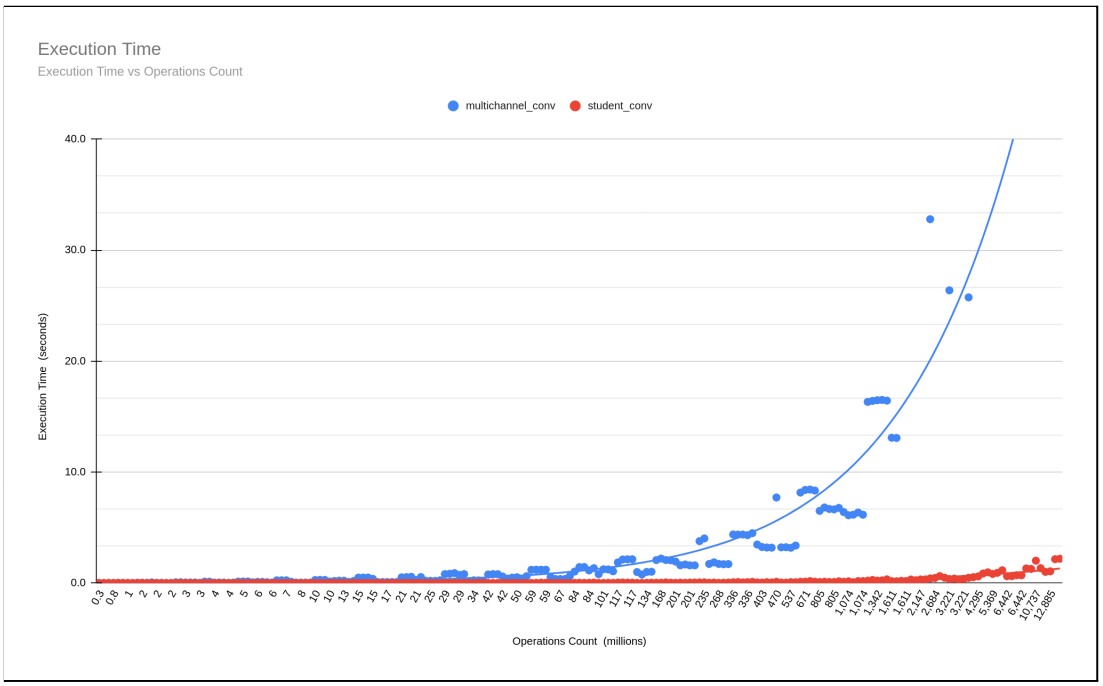

Fig 3.2.1 Comparison of function execution times on matching inputs

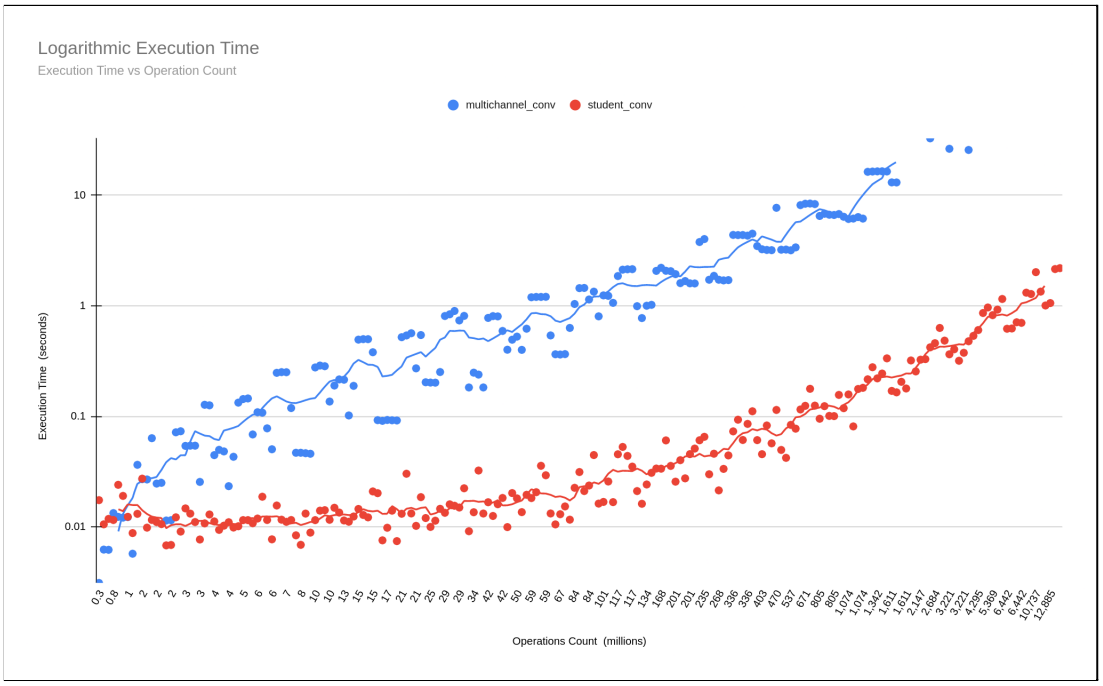

Fig 3.2.2 Logarithmic comparison of function execution times on matching inputs

3.3 Increase in Performance

The percentage change in performance can be calculated using $P = T / T_o - 1$, where T is the unoptimised execution time and T_o is the optimised execution time. For example, $P = 0.0$ signifies no change in performance while $P = 2.0$ signifies a two times performance improvement. As we can see below, for the largest input sizes, the optimised function is capable of achieving $P > 80.0$ which is upwards of eighty times increase in performance.

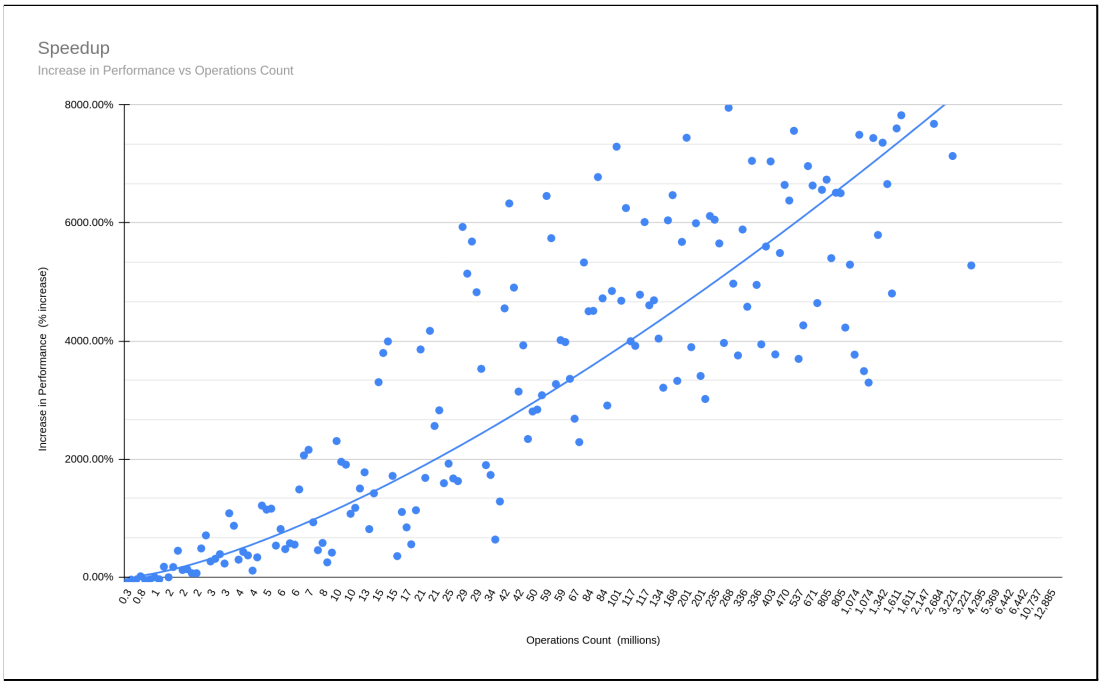

Fig 3.3.1 Achieved execution time speed-ups

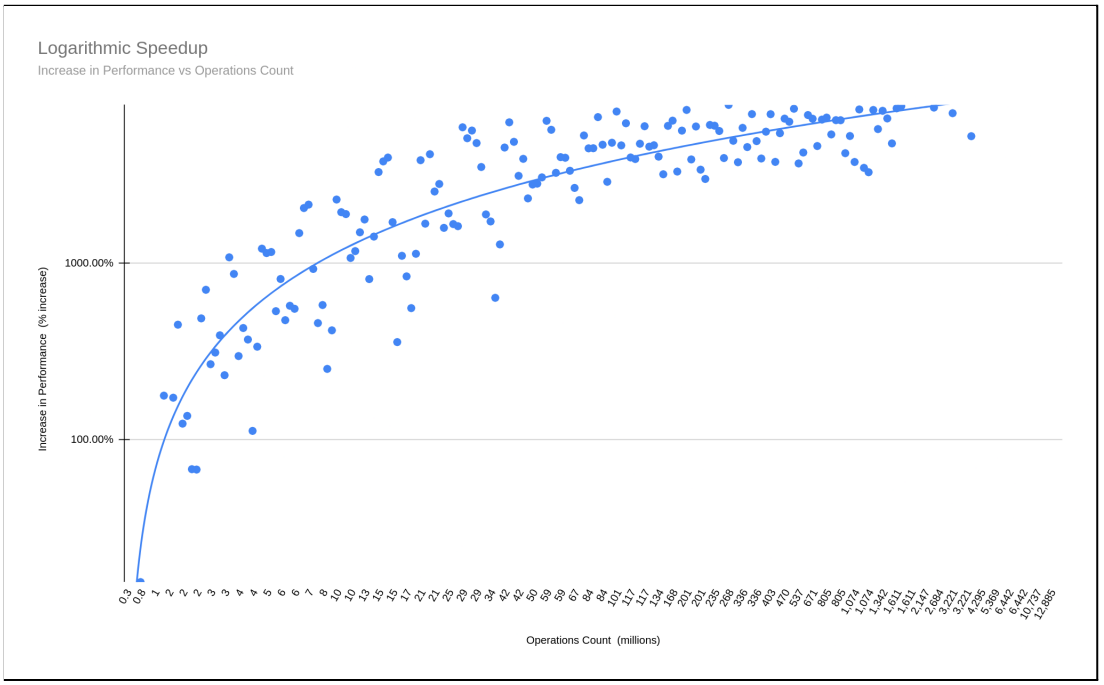

Fig 3.3.2 Achieved execution time speed-ups with logarithmic y-axis

5 Code Listing

A listing of the submitted GCC C99 code. Comments have been omitted for brevity.

```
void student_conv(float ***image, int16_t ****kernels, float ***output,
                  int width, int height, int nchannels,
                  int nkernels, int kernel_order) {

  #pragma omp parallel for
  for (int m = 0; m < nkernels; m++) {
    __attribute__ ((aligned(16)))
    float kernel[kernel_order][kernel_order][nchannels];
    for (int x = 0; x < kernel_order; x++) {
      for (int y = 0; y < kernel_order; y++) {
        #pragma GCC unroll 32
        for (int c = 0; c < nchannels; c++) {
          kernel[x][y][c] = kernels[m][c][x][y];
        }
      }
    }

    for (int w = 0; w < width; w++) {
      for (int h = 0; h < height; h++) {
        __m128d sums_lo = _mm_setzero_pd();
        __m128d sums_hi = _mm_setzero_pd();
        for (int x = 0; x < kernel_order; x++) {
          for (int y = 0; y < kernel_order; y++) {
            for (int c = 0; c < nchannels; c += 32) {
              #pragma GCC unroll 8
              for (int s = 0; s < 8; s++) {
                __m128 img4 = _mm_loadu_ps(&image[w+x][h+y][c+s*4]);
                __m128 krn4 = _mm_load_ps(&kernel[x][y][c+s*4]);
                __m128 mul4 = _mm_mul_ps(img4, krn4);
                sums_lo = _mm_add_pd(sums_lo, _mm_cvtps_pd(mul4));
                sums_hi = _mm_add_pd(
                  sums_hi, _mm_cvtps_pd(_mm_movehl_ps(mul4, mul4))
                );
              }
            }
          }
        }
        __m128d sums = _mm_add_pd(sums_lo, sums_hi);
        __m128d sum = _mm_hadd_pd(sums, sums);
        _mm_store_ss(&output[m][w][h], _mm_cvtpd_ps(sum));
      }
    }
  }
}
```