CSU33014 Assignment 2: Parallel Multichannel Multikernel Convolution

Alexander Sepelenco 20335014, Niall Sauvage 20334203
Trinity College Dublin

April 9, 2023

1 How to run the code

Our code was designed with Stoker in mind, in order to ensure the same compilation, Our Makefile is attached with our submission, which may or may not make it easier to run depending on if you have a prexisting makefile already.

2 Parallel multichannel convolution

Parallel multichannel convolution consists of 3d images, 4d kernels, that means there are multiple 2d images, in our case "channels" worth of 2d images, a width and a hieght, and "channels" worth of 2d kernels, we also have another dimension so "nkernels" worth of 3d kernels. The goal of the assignment is to optimise convolutions in a way that utilizes parallelised code mainly. We focused on parallelisation and overall speed, utilizing a variety of techniques. These techniques are discussed in the subsections.

2.1 Flatland

One of our first key insights was how memory was allocated - instead of several non-contiguous arrays of pointers, we realised that at the lowest level, all the data was contiguous. This allowed us to minimise memory accesses by using multiplication and addition to compute the offset of any element from the first element in the array. This is much faster in terms of cycles than memory accesses, but it led to more arithmetic being performed. To remedy this, we precalculated some values in our sum. For example, if we entered the relevant section of the "m" loop, then we would calculate the offset given by the m value. This allowed us to minimise time spent waiting for memory accesses and minimise unneccessary arithmetic.

2.2 Loop Fusing

The discussion on loop fusing was inspired from searching for optimisation strategies online. The stackoverflow discussion on loop fusing below was beneficial in our programme, however it could have also been done using OpenMP and loop collapsing. This example below is how loop fusing works, given nested arrays, they can mathematically be combined into one using division and modulus. In fact this seems slower, however due to the ability to parallelise more effectively division is the least of our problems. In fact this has an openmp name, collapse(). However we decided to do it manually as it was our first solution, and it is clearer to the programmer rather than having it behind a macro of openmp.

2.3 Unrolling

In order to increase cache hits, we attempted to unroll the inner (or tight) loop. Initially this was done with GCC using #pragma GCC unroll 16 as we knew that the number of channels was a multiple of 2 of minimum 32. As we were doing two calculations per loop iteration, we could then do a loop of size 32 (2 * 16) and this would be usable with any correct kernel. However, by inspecting the size of the executable, we noticed that the size did not change with or without the directive, implying it was not working, most likely an issue with understanding its function. Moreover, we discovered we could do 4 instructions per tight loop as we could load 4 image values into a vector, cast the lowest two to doubles, multiply them and then shuffle our original vector such that we could then repeat the process with the upper two lanes. We then decided to manually unroll the loop to do 16 doubles as it was most optimal for our needs. This allowed us to use less branches and more computation. It works out most of the time, as the nchannels is the loop that was unrolled meaning that nchannels works best when a big number and it starts at 16 and goes to 2048, which is quite good for loop unrollling.

```
for ( m = 0; m < nkernels; m++ ) {
    for ( h = 0; h < height; h++ ) {
        for ( w = 0; w < width; w++ ) {
            // code
        }
    }
}
```

```
#pragma omp parallel for
for (int n = 0; n < (nkernels*width*height); n++) {
    int m = n/(width*height);
    int w = (n%(width*height))/height;
    int h = (n%(width*height))%height;
    // code
}
```

2.4 Vectorisation

Effective vectorisation requires the arrays to be loaded in contiguous memory from an array. The original section of the code however only has 1 array be in contiguous memory, since we would like to guarantee better cache hits we moved the channels for loop to the most inner for loop, as the amount of channels for each kernel image can be at a maximum 2048 which is quite high. Transposing the kernel array was a key factor. This allowed us to create an array that can be loaded in contiguous memory from channels. Another issue that we faced was that the horizontal addition of 4 very big floats results in an inaccuracy in our programme, a fix for that was that we decided to deal with double type vectors (__m128d), this ensured accuracy. Below is a section of parallelisation ustilizing our loop fusing and vectorised code section utilizing SSE1-4. SSE part requires loading as float casting to double, then loading our double tranposed kernel, and multiplying the two vectors against each other, adding them up for all the channels, and then once the kernel_order on x an y of kernels has been executed sum up the vector using a single hadd instruction. The reason is because __m128d _mm_hadd_pd(v4sum), only stores two lanes, therefore 1 instruction is needed rather than the typical four we would see with float vectors (__mm128). An attempt was made for chaining loads by splitting into many vector variables, however it ended up being slower probably because of the memory being taxed too much and the need to use many SSE vectors in which we have a limited supply off. Shuffling is used to optimise out an extra load, and _MM_SHUFFLE is optimised out by precalculating the value that the macro generates..

```
// Transpose Kernel
#pragma omp parallel for // maybe this will numb the pain
for(int n = 0; n < (nkernels*nchannels*kernel_order); n++){
    int m = n/(nchannels*kernel_order);
    int c = (n%(nchannels*kernel_order))/kernel_order;
    int x = (n%(nchannels*kernel_order))%kernel_order;
    m_times_kernel_offset = m * kernel_offset;
    c_times_ko2 = c * ko2;
    x_times_kernel_order = x * kernel_order;
    kernel_total_offset_precalc = m_times_kernel_offset +
        x_times_kernel_order + c_times_ko2;
    t_kernel_total_offset_precalc = m_times_kernel_offset +
        x_times_kernel_order * nchannels + c;
    for(int y = 0; y < kernel_order; y++){
        t_kernel[t_kernel_total_offset_precalc + y * nchannels] =
            (double) kernel[kernel_total_offset_precalc + y];
    }
}
#define SHUFFLE_MASK 0b1110 // equivalent to _MM_SHUFFLE(0, 0, 3, 2);
// Vectorised Section
        for ( c = 0; c < nchannels; c+=2 ) {
            // t_kernel is aligned hence the load instead of loadu
            // cvtps convert float vector to double vector 4 -> 2 (lower) lanes
            // shuffle by moving 2 high values to take 2 low values spot
            // mul two double vectors, add them, and sum them up.
            __m128 v4image_1d = _mm_loadu_ps(image_1d+image_offset+c);
            __m128d v4image_1d_pd = _mm_cvtps_pd(v4image_1d);

            __m128d v4t_kernel_pd = _mm_load_pd(t_kernel+kernel_total_offset+c);

            __m128d product = _mm_mul_pd(v4image_1d_pd, v4t_kernel_pd);
            v4sum = _mm_add_pd(v4sum, product);

            c+=2;
            v4image_1d = _mm_shuffle_ps(v4image_1d, v4image_1d, SHUFFLE_MASK);
            v4image_1d_pd = _mm_cvtps_pd(v4image_1d);

            v4t_kernel_pd = _mm_load_pd(t_kernel+kernel_total_offset+c);

            product = _mm_mul_pd(v4image_1d_pd, v4t_kernel_pd);
            v4sum = _mm_add_pd(v4sum, product);

            c+=2;
            //unrolled for 16 computations at a time
        }
    }
}
v4sum = _mm_hadd_pd(v4sum, v4sum); // add the two lanes together and put in lower lane
output1d[m * width_times_height + w * height + h] = (float) _mm_cvtsd_f64(v4sum); // extract
```

Since nchannels is guaranteed to be a power of 2, a postloop is not required for this for loop.

2.4.1 Alignment

Alignmnet was used on our transposed array, as it is an array we can align by 16 bytes and allow for the usage of _mm_load_pd, in order to guarantee more cache hits. The transposed array is also a double since there is an issue with accuracy with float calculations. The array could have been a float but cast as a double in the inner most loop calculation, but because generated a double array from the start does not cost speed but space, generating a double array rather than float and casting similarly how we had to cast image array. The reason image array was not cast as a double above is because the kernel is relatively small, the image array is quite big and the benefit of making a double array equivalent of image is worse on speed than casting.

```
void* _mm_malloc (size_t size, size_t align) // for aligning in intel architecutre
void _mm_free (void * mem_addr) // free aligned intel align call
__m128d _mm_load_pd(double const * mem_addr); // load 16 byte aligned memory location
```

3 Speed Tests

Tested using ./conv 128 128 7 128 128

```
David conv time: 24594400 microseconds
Student conv time: 487744 microseconds
The total speed up time was 50.42x and 24106656 microseconds less
COMMENT: sum of absolute differences (0.000000) within acceptable range (0.062500)
```

The code from multiple tests, averages around 50 times the speed increase from the original code.

4 Sources

https://stackoverflow.com/questions/18749493/
openmp-drastically-slows-down-for-loop/18763554#18763554}

https://stackoverflow.com/questions/3219393/
stdlib-and-colored-output-in-c}