CSU33014 Lab 2: Parallel multichannel multikernel convolution

Alexander Sepelenco 20335014, Niall Sauvage 20334203
Trinity College Dublin

March 27, 2023

Contents

1 How to run the code

Our code was designed with Stoker in mind, in order to ensure the same compilation, Our Makefile is attachted with our submission, use it in order to ensure one to one compilation.

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 Loop Fusing

The discussion on loop fusing was inspired from searching for optimisation strategies online. This stackoverflow discussion on loop fusing was beneficial in our programme, 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.

```
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
}
```

2.2 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).

```
// 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];
    }
}

// code
// __m128d v4sum = _mm_setzero_pd();
// ... more for loops ...
// Vectorised Section
for ( c = 0; c < nchannels; c+=2) {

    __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_loadu_pd(t_kernel+kernel_total_offset+c);

    __m128d product = _mm_mul_pd(v4image_1d_pd, v4t_kernel_pd);
    v4sum = _mm_add_pd(v4sum, product);
}
// .. end of for loops
v4sum = _mm_hadd_pd(v4sum, v4sum);
output[m][w][h] = (float) _mm_cvtsd_f64(v4sum);
```

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

2.2.1 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.

```
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: 24483093 microseconds
Student conv time: 602688 microseconds
The total speed up time was 40.62x and 23880405 microseconds less
COMMENT: sum of absolute differences (0.000000) within acceptable range (0.062500)

David conv time: 24629741 microseconds
Student conv time: 629424 microseconds
The total speed up time was 39.13x and 24000317 microseconds less
```

The code from multiple tests, averages around 39 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}