Concurrent Systems Assignment 2

Declan Quinn 20334565
Ruairí Donaghey 20332113
Arshad Rehman Mohommad 20334119

Introduction

This assignment has been designed in order to give us real experience in optimising technology used and developed in the market of today. The concepts behind it, namely multichannel multikernel convolution, is on the cutting edge of image processing. This process, which gathers information through a “moving” window, must compute multiple facets of data for every pixel in the image. This means that while being very effective in its design, it is also very costly to execute.

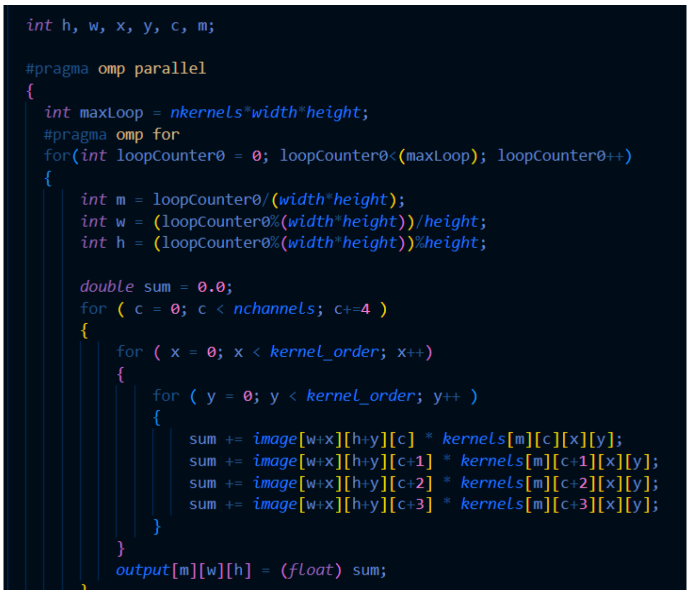
The process of doing convolution essentially boils down to an issue of matrix multiplication. We must take some target data, in this case we will work through each individual section of the image. We then do a number of multiplication and additions, based on the lens of our kernel, which filters just the information we need in the form we need. We could use this data to create and return a new image based on our input, or alternatively we could use this data in order to classify or recognise our image for use in a separate system.

Our job is to take a working program that David has already produced and make it drastically faster by using techniques taught in the module Concurrent Systems to speed up the project and to make it more efficient.

Implementation

The final implementation uses pragma omp parallel and pragma omp for, loop fusion and loop unrolling.

Pragma omp parallel and omp for are both used to split our algorithm into multiple threads in order to make the most of all of our processors at once. This is further aided by loop fusion, which works well with pragma omp. Although loop fusion was in the notes, we found a similar example on stack overflow(“OpenMP Drastically Slows Down for Loop”). This splits the first three of our for loops(nkernels, image width and image height) into one loop, and separates the counts through modulo and division operations. This cuts the number of loops we have to run drastically, meaning that after we divide our remaining loops among multiple processors, our compute time also goes down drastically.

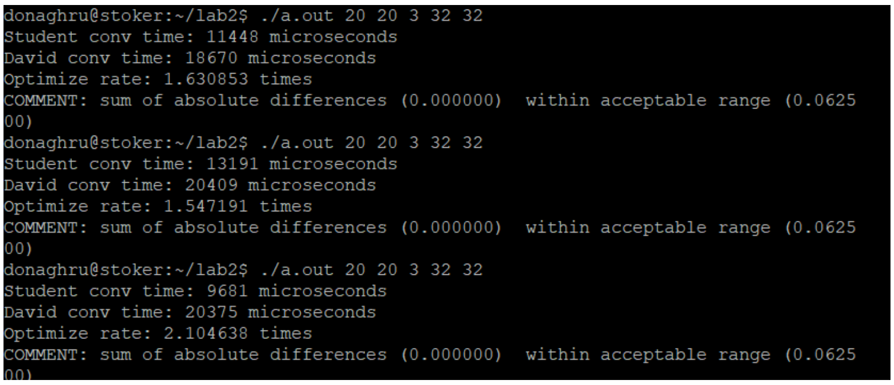


```
int h, w, x, y, c, m;

#pragma omp parallel
{
  int maxLoop = nkernels*width*height;
  #pragma omp for
  for(int loopCounter0 = 0; loopCounter0<(maxLoop); loopCounter0++)
  {
      int m = loopCounter0/(width*height);
      int w = (loopCounter0%(width*height))/height;
      int h = (loopCounter0%(width*height))%height;

      double sum = 0.0;
      for ( c = 0; c < nchannels; c+=4 )
      {
          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];
                  sum += image[w+x][h+y][c+1] * kernels[m][c+1][x][y];
                  sum += image[w+x][h+y][c+2] * kernels[m][c+2][x][y];
                  sum += image[w+x][h+y][c+3] * kernels[m][c+3][x][y];
              }
          }
          output[m][w][h] = (float) sum;
```

Within this larger loop, we then use loop unrolling to speed up our program by computing four channels at a time, which makes better use of our memory accesses. Doing 4 channels at a time strikes a nice balance between the memory used to store the channels, and the x and y of the kernels. This has meant a number of small speedups, leading to one large speedup in comparison to the original multichannels algorithm.

Test Results

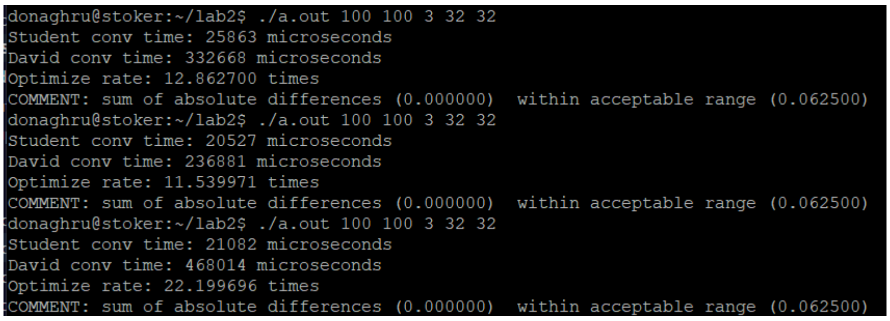
Results for input ./a.out 20 20 3 32 32



```
donaghru@stoker:~/lab2$ ./a.out 20 20 3 32 32
Student conv time: 11448 microseconds
David conv time: 18670 microseconds
Optimize rate: 1.630853 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.0625
00)
donaghru@stoker:~/lab2$ ./a.out 20 20 3 32 32
Student conv time: 13191 microseconds
David conv time: 20409 microseconds
Optimize rate: 1.547191 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.0625
00)
donaghru@stoker:~/lab2$ ./a.out 20 20 3 32 32
Student conv time: 9681 microseconds
David conv time: 20375 microseconds
Optimize rate: 2.104638 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.0625
00)
```

Average optimisation: 1.76 times

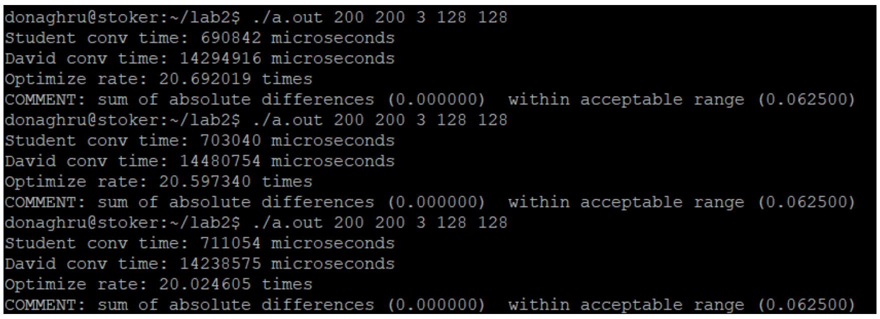
Results for input ./a.out 100 100 3 32 32



```
donaghru@stoker:~/lab2$ ./a.out 100 100 3 32 32
Student conv time: 25863 microseconds
David conv time: 332668 microseconds
Optimize rate: 12.862700 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
donaghru@stoker:~/lab2$ ./a.out 100 100 3 32 32
Student conv time: 20527 microseconds
David conv time: 236881 microseconds
Optimize rate: 11.539971 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
donaghru@stoker:~/lab2$ ./a.out 100 100 3 32 32
Student conv time: 21082 microseconds
David conv time: 468014 microseconds
Optimize rate: 22.199696 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
```

Average optimisation: 15.53 times

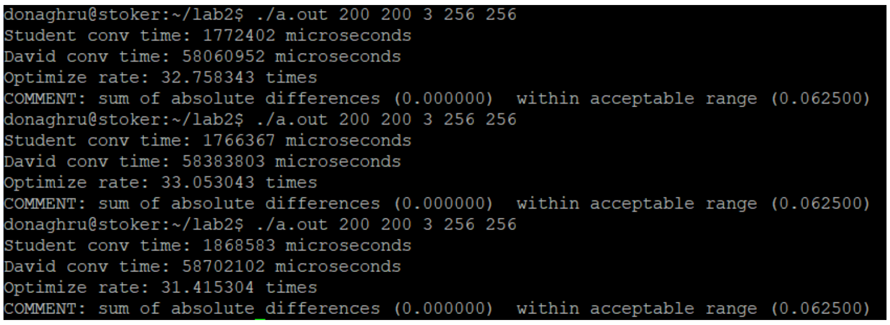
Results for input ./a.out 200 200 3 128 128



```
donaghru@stoker:~/lab2$ ./a.out 200 200 3 128 128
Student conv time: 690842 microseconds
David conv time: 14294916 microseconds
Optimize rate: 20.692019 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
donaghru@stoker:~/lab2$ ./a.out 200 200 3 128 128
Student conv time: 703040 microseconds
David conv time: 14480754 microseconds
Optimize rate: 20.597340 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
donaghru@stoker:~/lab2$ ./a.out 200 200 3 128 128
Student conv time: 711054 microseconds
David conv time: 14238575 microseconds
Optimize rate: 20.024605 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
```

Average optimisation: 20.43 times

Results for input ./a.out 200 200 3 256 256



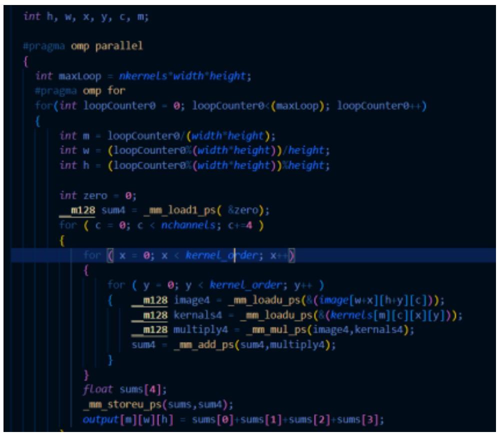
```
donaghru@stoker:~/lab2$ ./a.out 200 200 3 256 256
Student conv time: 1772402 microseconds
David conv time: 58060952 microseconds
Optimize rate: 32.758343 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
donaghru@stoker:~/lab2$ ./a.out 200 200 3 256 256
Student conv time: 1766367 microseconds
David conv time: 58383803 microseconds
Optimize rate: 33.053043 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
donaghru@stoker:~/lab2$ ./a.out 200 200 3 256 256
Student conv time: 1868583 microseconds
David conv time: 58702102 microseconds
Optimize rate: 31.415304 times
COMMENT: sum of absolute differences (0.000000)  within acceptable range (0.062500)
```

Average optimisation: 32.4 times

Attempted implementations

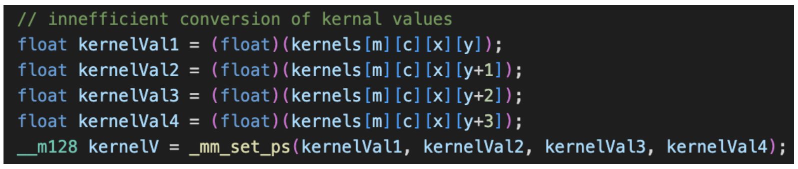
We had spent a lot of time during this project trying to implement SSE vectorization into our code. Unfortunately this did not end up being added to the final implementation as it proved to actually slow the algorithm down. In using SSE, there were multiple steps to be taken. One would have to load image and kernel values into vectors, use simple _mm_mul_ps and _mm_hadd_ps to multiply and sum the two vectors, and then store back our new values before using them to calculate the result we need. As you would imagine this approach took far too many instructions to perform efficiently due to the amount of memory loads and conversions and ran at about 0.8x the speed of david sort.

Below are screenshots of two separate attempts at using vectorisation to speed up the algorithm:



```
int h, w, x, y, c, m;

#pragma omp parallel
{
  int maxLoop = nkernels*width*height;
  #pragma omp for
  for(int loopCounter0 = 0; loopCounter0<(maxLoop); loopCounter0++)
  {
      int m = loopCounter0/(width*height);
      int w = (loopCounter0%(width*height))/height;
      int h = (loopCounter0%(width*height))%height;

      int zero = 0;
      __m128 sum4 = _mm_load1_ps( &zero);
      for ( c = 0; c < nchannels; c+=4 )
      {
          for ( x = 0; x < kernel_order; x++)
          {
              for ( y = 0; y < kernel_order; y++ )
              {   __m128 image4 = _mm_loadu_ps(&(image[w+x][h+y][c]));
                  __m128 kernals4 = _mm_loadu_ps(&(kernels[m][c][x][y]));
                  __m128 multiply4 = _mm_mul_ps(image4,kernals4);
                  sum4 = _mm_add_ps(sum4,multiply4);
              }
          }
          float sums[4];
          _mm_storeu_ps(sums,sum4);
          output[m][w][h] = sums[0]+sums[1]+sums[2]+sums[3];
```

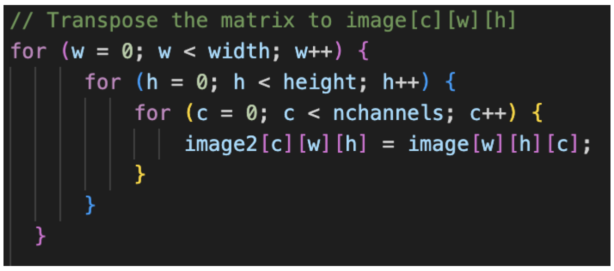
From the initial approach we learned that we would need to do something about reducing our instructions particularly in two key areas, first off loading the address values from memory of the image width and height needed to change to a format where we could just use a much faster mm_loadu instruction to save on memory accesses. The second issue we faced was that we needed to save time on converting the int16t values to floats as our first draft would store the address values in temporary variables and cast them to floats. We devised a solution that would transpose the matrix for image so that we could have our height value at the end of the matrix. This saved us some time from accessing different memory locations as we could us _mm_loadu_ps() on the innermost loops for our y values. This allowed us to vectorize our innermost loop more efficiently.



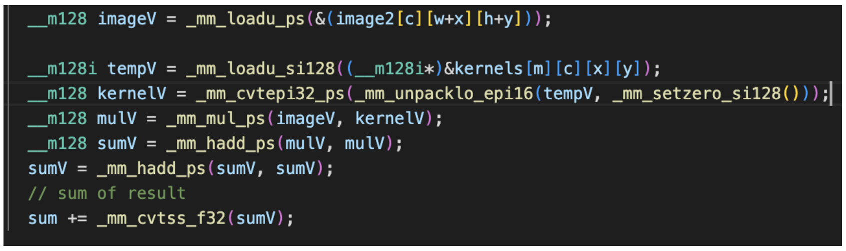
```
// innefficient conversion of kernal values
float kernelVal1 = (float)(kernels[m][c][x][y]);
float kernelVal2 = (float)(kernels[m][c][x][y+1]);
float kernelVal3 = (float)(kernels[m][c][x][y+2]);
float kernelVal4 = (float)(kernels[m][c][x][y+3]);
__m128 kernelV = _mm_set_ps(kernelVal1, kernelVal2, kernelVal3, kernelVal4);
```

We also figured out a way to convert the int16_t values to an m128 floating point vector. We load the values at the address provide as an __m128i integer vector. We then passed this vector into a function called _mm_unpacklo_epi16(). What this essentially would do was store 8 int16 values into a vector then it would take the lower half of those values, ie the first two integers or four int16 values and return them in a separate vector. This would be passed into a _mm_cvtepi32_ps() function that would essentially convert an integer vector into a float vector.

Although this approach took a number of vector instructions it turned out to be faster than manually loading and cleaning the address values. _mm_hadd_ps and _mm_mul_ps were used to calculate the sum of the multiplied vector values. Here we used _mm_cvtss_f32 to load the result from our final sum vector which is slightly faster than accessing the vector value from memory. When we ran this code in particular scenarios we would get a 1.2X to 1.05X speedup from running the code sequentially from David sort. With the speedup generally



```
// Transpose the matrix to image[c][w][h]
for (w = 0; w < width; w++) {
    for (h = 0; h < height; h++) {
        for (c = 0; c < nchannels; c++) {
            image2[c][w][h] = image[w][h][c];
        }
    }
}
```



```
__m128 imageV = _mm_loadu_ps(&(image2[c][w+x][h+y]));

__m128i tempV = _mm_loadu_si128((__m128i*)&kernels[m][c][x][y]);
__m128 kernelV = _mm_cvtepi32_ps(_mm_unpacklo_epi16(tempV, _mm_setzero_si128()));
__m128 mulV = _mm_mul_ps(imageV, kernelV);
__m128 sumV = _mm_hadd_ps(mulV, mulV);
sumV = _mm_hadd_ps(sumV, sumV);
// sum of result
sum += _mm_cvtss_f32(sumV);
```

reducing itself on larger sized inputs. Sounds great right? Unfortunately we ran into an unknown with our implementation of this code. On larger input values we ran into segmentation fault issues that we were unable to resolve in time for the submission. This was difficult as we had spent a lot of time trying to implement this feature and ended up having to scrap it in the end.

Conclusion

Although we were unable to implement our manual vectorized approach we still managed to improve the speed of David sort drastically using our program. We noticed from running our program that it tends to work better for larger datasets opposed to smaller datasets. This is to be expected as parallelisation should operate better by dividing up larger workloads by dividing more work between cores. What we found most interesting during the course of this project was that small optimisations to parallel loops lead to large increases in the optimisation rate of our final code.

Resources:

"OpenMP Drastically Slows Down for Loop." Stack Overflow,
stackoverflow.com/questions/18749493/openmp-drastically-slows-down-for-loop/18763554#18763554
IBM documentation. Available at: https://www.ibm.com/docs/en/zos/2.4.0 (Accessed: April 10, 2023).
Intel Intrinsics:
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ssetechs=SSE,SSE2,SSE3,SSSE3,SSE4_1&text=_mm_unpacklo_epi16&ig_expand=7675