CSU33014 Assignment 2 Report

Group Submission by:
Anthony Oisin Gavril - 20332783
Frank Guo 21363431

Problem Statement:

Using advanced programming techniques such as vectorization and parallelization, the goal of this project is to develop and improve a "student_conv" multichannel multikernel convolution function. Convolutional neural networks (CNNs) are essential components of modern AI systems; their layered neural data processing capabilities allow them to excel at tasks like picture classification.

In convolutional neural networks (CNNs), different layers are responsible for identifying features and performing classifications on image data in their own unique ways. Due to their use of various convolution kernels and handling of images with multiple channels, the convolutional layers are notoriously computationally intensive. These kernels greatly improve the network's prediction or classification accuracy by recognising picture patterns, edges, and textures.

The computational demands of the convolution layers, which require processing large amounts of data across multiple channels and kernels, pose the greatest challenge to this project. Processing images of increasing size and resolution adds another layer of complexity, necessitating increasingly effective computational strategies.

Process of Approach

After an in person meeting to discuss the assignment, we decided to utilise the material that we had learned in class, to prepare ourselves for the examination and to reinforce the material studied in the course; with a few additional online resources to help optimise our strategy. After some research, we came across Nvidia’s CUDA (Compute Unified Device Architecture) and OpenMP’s pragma. Initially, CUDA seemed to be a great option for us, with promises of multi core GPU speedup, but after further research, we discovered that CUDA was a little restrictive and specialised for us to use. CUDA's dependence on NVIDIA GPUs restricts its application to only those systems equipped with NVIDIA hardware, and it was clearly designed with usage for professional graphics related content in mind.
While researching, we discovered a great alternative; of pragma, a directive within OpenMP. OpenMP allows for simpler integration into existing codebases through its pragma directives, which are straightforward to implement.

Our Approach

We started by carefully examining the code provided to us, thinking of ways the SIMD instructions and other parallelisation methods provided to us in the course could be used to improve the code originally given to us.

```C/C++
/* 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;
        }
      }
    }
  }
}
```

We instantly noticed the six nested for loops. This would be an obvious culprit for slowing down code.

We first tried to implement SIMD instructions, but we unexpectedly found out that they caused the program to run slower when our method was only run with SIMD instructions as the method of parallelization. After accepting defeat and further consideration we did more research and discovered an optimisation method called Loop unrolling. Loop unrolling, also known as loop unwinding, is a well-liked optimization technique in software development. It speeds up programme execution by reducing or eliminating the overhead of looping constructs. This approach can only be implemented by modifying the loop to reduce the number of iterations and, consequently, the number of conditional branch instructions. Whether the loop should continue running or not is determined by these checks.

Based on the problem description the number of channels and number of kernels were always going to be powers of 2, that were >= 32.
We figured out that we could use SIMD (Singular Instruction, Multiple Data) instructions in parallel (no pun intended) with loop unrolling, if we set the channel increment to be 2 and the kernel increment to be 16.

This way we were able to perform operations across two channels, or as mentioned above across sixteen kernels. This would surely lead to another speedup in the code.

Upon examining the original, unoptimized code, it becomes clear that it consists of two main sections. The first section includes the outer loops, which are independent of each other since they don't share any data dependencies. The second section consists of the inner loops, where there is a data dependency involving the variable sum. Because there are no dependencies between the iterations of the outer three loops, they can be executed in any sequence. This allows for their parallelization, as each iteration operates independently without relying on the results of others.

```
C/C++
 #pragma omp parallel for if(nkernels>60) collapse(3) schedule(dynamic, 1)
shared(h, w, x, y, c, m)
  for ( m = 0; m < nkernels; m+=16 ){
    for ( w = 0; w < width; w++ ){
      for ( h = 0; h < height; h++ ){
              ...
```

The #pragma omp parallel for tells the compiler to parallelise the for loops by running each iteration on a separate execution thread. The collapse(3) part of the instruction tells the compiler that the outer 3 loops should be parallelised instead of just the very outer loop.
The shared(h,w,x,y,c,m) part tells the compiler that these variables are shared amongst the execution threads.
The if(nkernels > 60) makes the parallel execution condition. It specifies that the loop should only be executed in parallel if the number of nkernels is greater than sixty. If this condition is false then the program will be executed by a single thread.

Running 64 64 7 64 64 using only this line of code led an approximate 10x speed increase.

```
Average student run time over 10 runs: 152396 microseconds
Average David run time over 10 runs: 1655309 microseconds
Average speedup over 10 runs: 10.8805
Standard deviation of speedup over 10 runs: 1.03401
```

There was a significant improvement compared to the original unoptimised code. If we wanted to achieve a larger speedup, we would have to implement vectorisation or loop unrolling (in our case, both.)

Our original idea of working over two channels every time we went into the loop led to us working with __m128d instructions as seen in Intel's "Intel Intrinsics Guide". Using this data type, we were able to convert the image matrix and the kernel matrix into the same type and from there, we were able to operate on these values..

C/C++

```
// image is calculated for both values of c
// because we are incrememementing the channels by 2 each time.
// If we were to increment by 4 each time, we would change the number of
// image calculations

 __m128d imageCalculation =_mm_set1_pd((double)image[w+x][h+y][c]);
__m128d secondImageCalculation = _mm_set1_pd((double)image[w+x][h+y][c+1]);

// sum += image[w+x][h+y][c] * kernels[m][c][x][y];
// assigning values to the kernel

__m128d kernel0 = _mm_setr_pd((double)kernels[m][c][x][y],(double)
kernels[m+1][c][x][y]);
 ...
__m128d kernel7 = _mm_setr_pd((double)kernels[m+14][c][x][y],(double)
kernels[m+15][c][x][y]);

// The vectors hold 2 doubles each.
// here the image and the kernel values are multiplied
sum0 = _mm_add_pd(sum0, _mm_mul_pd(imageCalculation,kernel0));
...
sum7 = _mm_add_pd(sum7, _mm_mul_pd(imageCalculation,kernel7));


//The previous code can only handle 8 values at a time so we need to do it
//again to get the other other values. (because we are incrementing by 16
//each  time)

__m128d kernel01 = _mm_setr_pd((double)kernels[m][c+1][x][y],(double)
kernels[m+1][c+1][x][y]);
...
__m128d kernel71 = _mm_setr_pd((double)kernels[m+14][c+1][x][y],(double)
kernels[m+15][c+1][x][y]);

// second round of sums
sum0 = _mm_add_pd(sum0, _mm_mul_pd(secondImageCalculation,kernel01));
...
sum7 = _mm_add_pd(sum7, _mm_mul_pd(secondImageCalculation,kernel71));
```

C/C++

```
// sumX[0] accesses the first element of sum vector and sumX[1] the second
// element (__m128d, stores 2 values)
        output[m][w][h] =  sum0[0];
        output[m+1][w][h] = sum0[1];
```

```
        ....

    output[m+14][w][h] = sum7[0];
    output[m+15][w][h] = sum7[1];
```

The code snippet above demonstrates how we stored the results of our operation in a multidimensional array.

Standard input sizes

Set	Image Width	image Height	Kernel Order	Number of Channels	Number of Kernels
1	64	64	7	64	64
2	128	128	5	64	64
3	128	128	7	256	256

For this document, we used three testing size sets which are shown above. These sets will be used later on to list the execution times we achieved along with the corresponding speedups.

Time Improvement

A bash script was written to automatically run each set 100, 10 and 5 times. The run times are as follows:

Set 1:

MacNeill:

```
Average student run time over 100 runs: 80880 microseconds
Average David run time over 100 runs: 2530733 microseconds
Average speedup over 100 runs: 31.7945
Standard deviation of speedup over 100 runs: 5.4252
```

Stoker:

```
Average student run time over 100 runs: 67793 microseconds
Average David run time over 100 runs: 1608861 microseconds
Average speedup over 100 runs: 24.3011
Standard deviation of speedup over 100 runs: 4.42263
```

Set 2:

MacNeill:

```
Average student run time over 10 runs: 166990 microseconds
Average David run time over 10 runs: 5685045 microseconds
Average speedup over 10 runs: 34.1107
Standard deviation of speedup over 10 runs: 3.92842
```

Stoker:

```
Average student run time over 10 runs: 153661 microseconds
Average David run time over 10 runs: 3820825 microseconds
Average speedup over 10 runs: 24.902
Standard deviation of speedup over 10 runs: 1.35913
```

Set 3:

MacNeill:

```
Average student run time over 5 runs: 3458859 microseconds
Average David run time over 5 runs: 196458393 microseconds
Average speedup over 5 runs: 59.258
Standard deviation of speedup over 5 runs: 11.0739
```

Stoker:

```
Average student run time over 5 runs: 3722702 microseconds
Average David run time over 5 runs: 135294986 microseconds
Average speedup over 5 runs: 36.356
Standard deviation of speedup over 5 runs: 0.643981
```

Conclusion

We are extremely satisfied with the final product of the project. Our initial progress was hindered by difficulties, especially when we spent too much time on vectorization alone. We were able to overcome this obstacle, by utilising OpenMP and loop unrolling techniques. We learned a lot about creating an effective multichannel multikernel convolution routine from this assignment, which was really educational. It effectively reinforced our understanding of parallel programming and vectorization as taught in this course module.

References:

Collapse clause · OpenMP Little Book.
https://nanxiao.gitbooks.io/openmp-little-book/content/posts/collapse-clause.html

Intel Intrinsics Guide
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=SSE_ALL&ig_expand=5850,5841,5811

Wikiwand - Loop unrolling

https://www.wikiwand.com/en/Loop_unrolling