CS3014 Lab 1: Sparse parallel multichannel multikernel convolution

Can ZHOU zhouc@tcd.ie 19324118

The efforts made to make the code efficient:

1. Loop unrolling

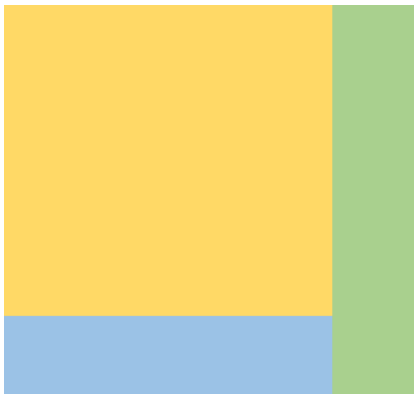
Loop unrolling is a technique for loop transformation which helps to optimise a program's execution time. It can remove or reduce iterations. I used this method with four iterations unrolling on the second *for* loop in both initialization and convolution parts. By using this, the convolution part can calculate four sum values in one w(width)'s iteration.

2. SIMD SSE -x86

The second optimization I used is SSE. __m128 (4 32-bit precision floats) in SSE could load, calculate, and store four float numbers at same time. By using this I am able to operate four calculations in one operation time.

SSE has been used in both *initialize the output matrix to zero* and *compute multichannel, multikernel convolution* parts. In order to use SSE, I have to change some code structures:

a. In both initialization and convolution parts, the matrix is divided into three blocks.



The yellow part will use SSE to speed up. As I mentioned before I have used loop unrolling (operate four operations in one iteration in the second for loop) to optimize the code. So, the size of yellow part should be (width – width%4) * (high – height%4), the size of green part is height * (width%4) and blue part is (height%4) * (width – width%4).

After calculated the yellow part, because of the small values of width and height, I just use normal *for* loop to operate green and blue parts.

b. Change nested *for* loops order.

In the initialization part, when calculate green and blue part, I put m (nkernels) as the innermost loop to avoid unnecessary for loop when the width or height is exactly divided by 4.

I got some ideas when I read *multichannel_conv_dense* function, so I put m's *for* loop as the outermost loop in convolution part. In this order, I can implement the SSE on w (width).

In conclusion, in initialization part, I used SSE with this: *_mm_storeu_ps* instruction to assign four elements in width to zero in one operation time. And in convolution part, SSE was used in h(height) variable with those instructions _mm_setzero_ps;_mm_set1_ps; _mm_setr_ps;_mm_add_ps.

3. OpenMP

The third method I used is OpenMP. OpenMP can let the program run in multiple threads automatically. At first, I added the "*#pragma omp parallel for*" for the outermost loop (nkernels) in convolution part, but I found some issues:

a. Result Mistakes:
   OpenMP thread are sharing loop variables. I will get wrong output because some image parts have been skipped and it will cause segmentation fault as well. So, I added the *private()* and *shared()* clause to make sure OpenMP was working properly.
b. The threshold to use OpenMP
   Both SSE and OpenMP can speed up the program to a certain extent and I found using SSE and OpenMP at same time can have a faster speed. But when input is small, the program actually slows down. Because it takes more time to generate and handle more threads (which don't really do anything).
   So, I used *if()* clause in OpenMP to avoid using OpenMP when the dataset is small, but the point is what is the threshold for the "small" dataset. In order to solve this issue, I wrote some shell and python files to gain the threshold by testing different inputs automatically. Each different input test is run 10 times to avoid coincidence and I use the average time in the comparation.
   I find when the input larger than 270 * 32 * 64 * 3 (width * nchannels * nkernels * kernel_order), OpenMP + SSE will faster than SSE, but it may not very accurate as this threshold is from experiences.

The final OpenMP command I used was *#pragma omp parallel for if(OpenMP_flag) private(w, h, m, x, y) shared(kernels, image, output)* for

the outermost for loop in convolution part. The OpenMP_flag will be 1 if input (width * nchannels * nkernels * kernel_order) > 270 * 32 * 64 * 3.

Timings: [All values are in Microseconds]

The machine I used:

Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz

Sockets: 1

Cores: 6

Logical processors: 12

I have attached the /proc/cpuinfo file in the email.

Input 16 16 1 32 32 20

	Average Time (10 tests)
Original code	45
My Code	28.9
Faster times	**1.56 times**

Input 32 32 3 32 64 200

	Average Time (10 tests)
Original code	881.3
My Code	286.3
Faster times	**3.08 times**

Input 128 128 3 32 32 20

	Average Time (10 tests)
Original code	18900.4
My Code	5208.1
Faster times	**3.63 times**

Input 128 128 5 64 64 20

	Average Time (10 tests)
Original code	197486.3
My Code	12345.5
Faster times	**15.997 times**

Input 256 256 1 256 256 20

	Average Time (10 tests)
Original code	864744.3
My Code	95502.9
Faster times	**9.05 times**

Input 256 256 3 256 256 200

	Average Time (10 tests)
Original code	2298194.1
My Code	100624.7
Faster times	**22.84 times**

From the results above, we can see that no matter the input is small or large, my code is faster than the original code and this feature is especially obvious when inputting large data, and sometimes my code can even be 23 times faster than the original program.

References:

1. Lecture notes: https://www.scss.tcd.ie/David.Gregg/cs3014/notes/
2. Configuring the Kernel Launch Parameters Part 1: https://www.youtube.com/watch?v=9ZhuzpitHgM&list=PLAwxTw4SYaPnFKojVQrmyOGFCqHTxfdv2&index=40
3. Intrinsics Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#
4. Vectorization part1. Intro.: https://easyperf.net/blog/2017/10/24/Vectorization_part1
5. Why increase the execution time in my openmp code? https://stackoverflow.com/questions/30286704/why-increase-the-execution-time-in-my-openmp-code