Sparse parallel multichannel multikernel convolution

Can ZHOU
19324118 zhouc@tcd.ie

Part One:

The efforts made to make the code efficient

1. **Loop Unrolling**
2. **SIMD SSE -x86**
3. **OpenMP**

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 [Overall]

- __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.
- But I I have to change some code structures.

2. SIMD SSE -x86 [Code Structures]

The matrix is divided into three blocks:

- The yellow part will use SSE to speed up.
- The size of yellow part is (width – width%4) * (high – height%4)
- The size of green part is height * (width%4)
- Blue part is (height%4) * (width – width%4).

Change nested for loops order:

- 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 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.

3. OpenMP [Overall]

- 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:

3. OpenMP [Issues]

Issue one: Result Mistakes

- OpenMP thread are sharing loop variables.
- I added the private() and shared() clause to make sure OpenMP was working properly.

Issue two: the threshold to use OpenMP

- When input is small, OpenMP actually slows down the Program.
- Because it takes more time to generate and handle more threads (which don't really do anything).
- I found threshold to use OpenMP from multiple tests.
- If input (width * nchannels * nkernels * kernel_order) > 270 * 32 * 64 * 3, then use OpenMP

Part Two:

Timings

Machine I used:

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

Sockets: 1

Cores: 6

Logical processors: 12

Timings: [All result are average times]

- Input 16 16 1 32 32 20

  Original code 45 Microseconds

  My Code 28.9 Microseconds

  Faster times 1.56 times

- Input 32 32 3 32 64 200

  Original code 881.3 Microseconds

  My Code 286.3 Microseconds

  Faster times 3.08 times

Timings: [All result are average times]

- Input 128 128 3 32 32 20

  Original code 18900.4 Microseconds

  My Code 5208.1 Microseconds

  Faster times 3.63 times

- Input 128 128 5 64 64 20

  Original code 197486.3 Microseconds

  My Code 12345.5 Microseconds

  Faster times 15.997 times

Timings: [All result are average times]

- Input 256 256 1 256 256 20

  Original code 864744.3 Microseconds

  My Code 95502.9 Microseconds

  Faster times 9.05 times

- Input 256 256 3 256 256 200

  Original code 2298194.1 Microseconds

  My Code 100624.7 Microseconds

  Faster times 22.84 times

Part Two:

Timings

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.