Report for intel writing test

Mail: wulsh26@mail2.sysu.edu.cn

Author: Longshi Wu

In this test, I use c++ to implement convolution , relu and pooling functions.

To do

- [x] Conv2D, Pooling2D and RELU operator
- [x] Forward
- [ ] Backward
- [x] Optimization skill
    - [x] SIMD
    - [x] OpenMP
    - [x] Cache locality
- [x] Advantage feature
    - [ ] Fuse Conv + relu + pooling into one function
    - [x] Parallel three functions with OpenMP

Laptop CPU Specs

Using command line `sysctl -a | grep cpu` to check those information below.

- Hardware：Intel® Core™ i5-8257U Processor @1.4GHz 4 core， support AVX2, 2 FMA, SSE2
- Operating System： macOS version 10.14.6 (18G103)

So the **peak GFLOPs** = 4 x 1.4 x 32 = 179.2GFlOPs in this machine.

Similarly, for a single core, this number is 44.8 GFLOPs.

Input Data

Because this work now only inlcude forward pass part, we can just input some test data into those functions (conv2d, relu, pooling) to compute performance (flops). Thus, The test data don't need labels, which be generated randomly.

Assume the input data shape is $[N, C_{in}, H_{in}, W_{in}]$. The multi-dimentional array is a little hard to construct using c++, as arrays are physically stored in a linear, one-dimentional computer memory, I just using one-dimentional array to store data.

Data Structure

```
class Data{
public:
    int batch,depth,width,height;
    float *data;
    // function
    Data();
    Data(int n,int c, int h, int w);
    ~Data();
    Data& operator=(const Data & chunks);
    float getValue(int i, int j, int m, int n);
    void SetValue(int i, int j, int m, int n, float  value);
    void AddValue(int i, int j, int m, int n, float value);
    void RandomInit();
    void Init(float fillValue = 0);
    void print();
};
```

Convolution

Simple for-loop convolution

The direct convolution is simple for-loop convolution, call it as `naiveConv`.

```
for(int idx=0; idx < input.batch; idx++) {
    for(int channel=0;channel < output.depth; channel++){
        for(int out_h=0; out_h < output.height; out_h++){
            for(int out_w=0; out_w < output.width; out_w++){
                for(int ichannel=0; ichannel< input.depth; ichannel++){
                    for(int k_h=0; k_h < kernel.height;k_h ++){
                        for(int k_w=0;k_w <kernel.width; k_w++){
```

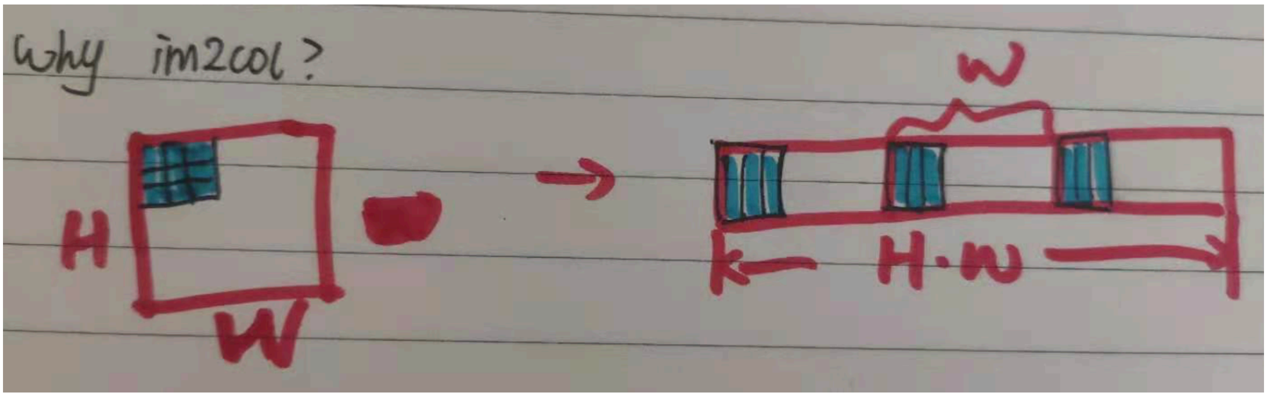
Clearly, it has 7 nested for loops. Using naiveConv as basline to compare with some optimized methods.

Optimized convolution

Implment the optimized convolution function follow the commonly used optimized method: im2col + GEMM .

im2col (image to column)

To convert for-loop convolution to maxtrix multiplication, we need to transfrom input image to matrix. In for-loop convolution, the input patch data where the conv filter is applied actually not stored linearly, which slow the speed of calculation.



im2col is proposed to rerange input data to make data stored linearly, then cpu can access data faster. The key idea is to find all the patches traversed by the convolution kernel in sequence at one time, and reorder those patches.

GEMM (generalized maxtrix multiplication)

As we know that matrix product could get the conv output directly (as below formula shows) , then the key point now is how to accelerate GEMM.

$$C_{M\times N} += A_{M\times K} * B_{K\times N}$$

Here I use three skills.

- **Loop reordering**. Reorder the loops to access data efficiently.
- Using **openMP** to parallelize outer loop.
- Using **SIMD** to do vectorization. It processes multiple data streams using a single instruction stream to accelerate computation.

```
/*GEMM function */
void gemm_nn_ijk(int M, int N, int K, float ALPHA,
             float *A, int lda,
             float *B, int ldb,
             float *C, int ldc);
void gemm_nn_ikj(int M, int N, int K, float ALPHA,
             float *A, int lda,
             float *B, int ldb,
             float *C, int ldc);
void gemm_nn_ikj_simd(int M, int N, int K, float ALPHA,
                 float *A, int lda,
                 float *B, int ldb,
                 float *C, int ldc)
```

- **Loop Reordering**

```
/* loop order is i,j,k */
void gemm_nn_ijk(...){
  ...
  for(i = 0; i < M; ++i){
          for(j = 0; j < N; ++j){
              for(k = 0; k < K; ++k){
                  C[i*ldc+j] += A[i*lda+k]*B[k*ldb+j];
              }
          }
  }
}
```

The last inner loop is traverse `k` from `0` to `K` , so when find `B[k,j]`, the next element `B[k+1,j]` may not cached, it need time to access data from RAM. So if we

reorder the loops from `i,j,k` to `i,k,j`, it may reduce cache misses.

```
/* loop order is i,k,j */
for(i = 0; i < M; ++i){
        for(k = 0; k < K; ++k){
            register float A_PART = ALPHA*A[i*lda+k];
            for(j = 0; j < N; ++j){
                C[i*ldc+j] += A_PART*B[k*ldb+j];
            }
        }
    }
```

- **Threading**

Using openMP to parallelize loops.

```
#include <omp.h>
// add this directive before outer loop
#pragma omp parallel for private(k,j) schedule(dynamic)
for(i = 0; i < M; ++i){
  ...
```

- **SIMD**

Single Instruction Multiple Data, or SIMD, which can be used to do the same operation(add, multiply, etc .) on multiple values simultaneously. A single-precision floating-point number occupies 4 bytes, the laptop I used support AVX2, so it could operate four floating-point numbers at the same time (128-bit vector containing **4** `floats`).

```
#include <x86intrin.h>
void gemm_nn_ikj_simd(...){
  ...
for(i = 0; i < M; i++){
```

```
        for(k = 0; k < K; k++){
            __m128 a4 = _mm_set1_ps(A[i*lda+k]);
            for(j = 0; j < N; j+=4){ // here vectorized 4 float
                __m128 b4 = _mm_load_ps(&B[k*ldb+j]);
                __m128 c4 = _mm_load_ps(&C[i*ldc]+j);
                c4 = _mm_add_ps(_mm_mul_ps(a4,b4),c4);
                _mm_store_ps(&C[i*ldc+j],c4);
            }
        }
    }
}
```

Testing

> Input shape: [10 x 3 x 100 x 100]
>
> Kernel shape: [5 x 3 x 7 x 7], padding = [0,0], stride=[1,1]
>
> Output shape: [10 x 5 x 94 x 94]

	GLOPS
`naiveConv` (baseline)	0.1078
`gemm_nn_ijk`	1.3126
`gemm_nn_ikj`	2.3871
`gemm_nn_ikj_simd`	3.672

As we mentioned before, `naiveConv` has simple 7 nested loops, `gemm_nn_xxx` all used `im2col` and OpenMP to parallelize loops. The last one, `gemm_nn_ikj_simd`, used all three optimized skill in the context , which increased by almost 36 times compared with baseline (3.6 vs 0.1).

Pooling

- Implemented functions: `maxpool2d` and `avgpool2d`
- no learnable parameters
- use openMP to parallelize loops.

```
Data PoolingLayer::avgpool2d(Data& input,int kernel_size, int *stride, int
*padding) {
    int N = input.batch;
    int C = input.depth;
    int outputH = computeShape(input.height,padding[0],stride[0],kernel_size);
    int outputW = computeShape(input.width,padding[1],stride[1],kernel_size);
    Data output = Data(N,C,outputH,outputW);
#pragma omp parallel for collapse(4)
    for(int i = 0; i < N; i++){
```

```
        for(int j = 0; j < C; j++){
            for(int outh = 0; outh < outputH; outh++){
                for( int outw = 0; outw < outputW; outw++){
                    float result = 0;
                    for( int m = 0; m < kernel_size; m ++){
                        for( int n = 0; n < kernel_size; n++){
                            result +=
input.getValue(i,j,stride[0]*outh+m,stride[1]*outw+n);
                        }
                    }
                    result = result / (float)(kernel_size * kernel_size);
                    output.SetValue(i,j,outh,outw,result);
                }
            }
        }
    }
    return output;
}
```

- Testing
  - Input data and kernel size have same shape as the convolution test case.
  - `avgpool2d` **0.24 Gflops /0.60 Gflops**. (no parallel / with parallel)
  - `avgpool2d` **0.15 Gflops /0.57 Gflops**. (no parallel / with parallel)

ReLu

$$ReLU(x) = max(0, x)$$

`relu` function is easy to implement, as it only need to compared itself with zero. The element-wise operate can use `OpenMP` to parallize. Because it has no paremeters, we can fuse convolution and relu to reduce runtime.

```
// void gemm_relu()
for(i = 0; i < M; i++){
  for(k = 0; k < K; k++){
      for(j = 0; j < N; j+=4){...}
  }
  // add this line to implement relu in-palce
  for (j = 0; j < N; ++j) {
    C[i*ldc+j] = ( C[i*ldc+j] > 0) ? C[i*ldc+j]:0;
```

Summary

This test was very interesting and challenging for me. In the process of writing code while searching information, I learned that the implementation of the neural network framework is complecated. Although I have not implement backward propagation, I learned somthing about how to accelerate GEMM, which i I think it is the core of DNN.