C/C++ Project 3 Report

Qiu Kunyuan 11913019@mail.sustech.edu.cn

Introduction

This project implements convolution between a small kernel K and a large image X in C++. To maximize the efficiency, optimizations ranging from mathematical refactoring to SIMD operations and OpenMP multi-threading library is applied.

Optimizations

Convolution is a computation dense application, thus the naive method from the mathematical principle

$$(f*g)(i,j)=\sum_{p=0}^{k}\sum_{q=0}^{k}f(p,q)g(k-1-p,k-1-q)$$

is highly complicated in time with the complexity of $\mathcal{O}\left(K^2HW\right)$, where the K is the size of the square kernel matrix K and the dimension of a channel of the image $X[i]$ is $H\times W$.

Mathematical Level Optimization

There are two mainstreams of optimizations at the level of mathematics. The first method based on the linear operation and transformation, among which the most widely applicated one is Winograd transformation and Fourier transformation.

$$\mathcal{T}(X*K)=\mathcal{T}(X)\times\mathcal{T}(K)$$

However, transformations on computer costs additional time and space, and the implementation of matrix multiplication often suffers I/O bottleneck and buffer-memory delay since the output from the highly optimized transformation algorithms are mostly poorly aligned.

The reshaping method, in contrast, re-arranges the rather small kernel to a single rowed vector. The convolution operation can then be performed by highly optimized matrix operations (for example `sgemm` functions in modern BLAS/LAPACK libraries), getting rid of the bottlenecks encountered in transformation method.

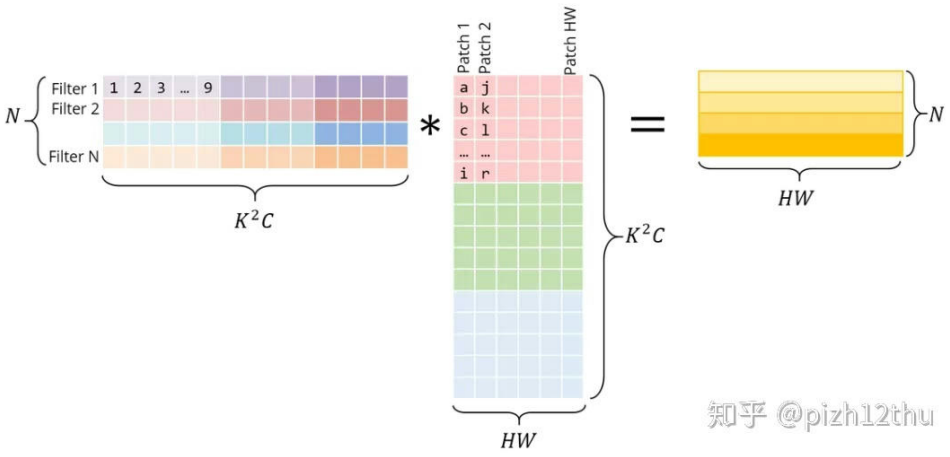

Figure 1: Rearrangement of kernel and image

For the row-major matrix storage, the relation between the 2D index `K[i][j]` and the 1D memory index `K[m]` is following:

`H[m_1]=X[m_1+(W*i+j)]*K[k^2+1-(k*i+j)]`

where `k` is the dimension of the kernel, and the `m_1` is the linear index in memory map. The operation `K[k^2+1-(k*i_j)]` is the reversion in convolution operation, in the actual implementation this part is replaced by a easy for-loop inversion.

```
void loadKernel(float *kernel, int len) {
  float temp = 0.0;
  for (int i = 0; i < len / 2; i++) {
    temp                = kernel[i];
    kernel[i]           = kernel[len - 1 - i];
    kernel[len - 1 - i] = temp;
  }
}
```

SIMD Optimization

To maximize the efficiency, the convolution operation is implemented by SIMD intrinsic commands since the operation has been paraphrased to vector dot.

`H[m]=vector_dot(X[m+(W*i+j)],reverse(K[k*i+j]))`

The vectors sliced from the data matrix is stored in the `__m256` vector data type intrinsically provided by the AVX2 ISA. To modify one single float32 elements in a vector when needed, the `__m256` type is packaged to a union so that the data can be modified as a `float32` array.

```
union M128 {
  __m128 m128;
  float  f32[4];
};
union M256 {
  __m256 m256;
  float  f32[8];
};
```

The implementation of dot products is the follwing snippet:

```
src_loaded[0] = src[i];
src_loaded[1] = src[i + 1];
src_loaded[2] = src[i + 2];
src_loaded[3] = src[i + W];
src_loaded[4] = src[i + W + 1];
src_loaded[5] = src[i + W + 2];
src_loaded[6] = src[i + 2 * W];
src_loaded[7] = src[i + 2 * W + 1];
src_loaded[8] = src[i + 2 * W + 2];
d0_7.m256     = _mm256_loadu_ps(src_loaded);
d8.m256       = _mm256_setzero_ps();
d8.f32[0]     = src_loaded[8];
res0_7.m256   = _mm256_mul_ps(k0_7.m256, d0_7.m256);
dst[i]        = res0_7.f32[0] + res0_7.f32[1] + res0_7.f32[2] + res0_7.f32[3]
+ res0_7.f32[4] + res0_7.f32[5] + res0_7.f32[6] + res0_7.f32[7] + k8.f32[0] *
d8.f32[0];
```

To avoid unsafe copy in `memcpy` function and to reduce debugging time, the transformation of the image data from the source array to the kernel-sized small matrix is naive assigning, since the kernel is rather small.

Benchmarking

The 3×3 convolution subroutine is compared with NumPy `convolution2d` and OpenCV convolution to verify its correctiness in result.

The evaluation time is rather slow comparing to the two highly optimized libraries mentioned.