week-14

JX-Ma

2023/12/22

1 本周工作

1. 阅读了关于 gemm 在 cpu 上优化的论文的前向传播循环结构。
2. 将优化分为索引优化，hoisting 优化和 unrolling 优化，将其中几个优化的 gflop 展示在 roofline 中。

2 阅读心得

2.1 摘要

这篇文章介绍了神经网络中的卷积操作和直接卷积在 X86 架构上的实现和一种基于动态编译的直接卷积内核，针对 X86 架构的 Xeon 和 Xeon Phi 系统进行了优化。通过基于 JIT-optimized 的实现，可以实现接近理论峰值性能的表现。

2.2 基本术语

- 神经网络由多个神经元层通过权重连接而成。分配给神经元的值通常被称为激活值。
- N、C、H 和 W 表示输入张量的维度
- 输出张量 N K P Q
- 卷积核维度 K C R S

2.3 前向传播循环结构

前向传播层由七个嵌套循环组成，这些循环对输入张量 I 和权重张量 W 进行卷积，产生输出张量 O 如下

```
    for (int64_t i = 0; i < C.num_batch(); ++i) {
for (int64_t j = 0; j < C.num_channel(); ++j) {
for (int64_t r = 0; r < B.num_channel(); ++r) {
for (int64_t m = 0; m < C.num_height(); ++m) {
for (int64_t n = 0; n < C.num_width(); ++n) {
    ms = m*s;
```

```
    ns = n*s;
for (int64_t u = 0; u < B.num_height(); ++u) {
for (int64_t v = 0; v < B.num_width(); ++v) {
    // index.count(m*s and n * s) = 2 * N^7 —> 2 * N^5;
        C(i, j, m, n) += A(i, r, ms + u, ns + v) * B(j, r, u, v);
}
}
}
}
}
}
}
```

2.4 Vectorization and Register Blocking(向量化和寄存器分块)

我们选择将通道按照 VLEN 的因子进行分块，并将向量化块作为张量的最内层、运行最快的维度
我们在输出张量的空间域中应用寄存器分块，因为空间迭代空间中的点可以独立计算。通过这种方式，我们在寄存器中形成了独立的累积链，足以隐藏 FMA 的延迟。

2.5 Cache Blocking and Loop Ordering（缓存分块，循环排序)

循环顺序决定了张量的访问方式，并影响相应数据的重用。合理的分配循环顺序能更有利于增加卷积的速度。

2.6 Code Generation for Convolution Microkernel(卷积微内核的代码生成) 将初始的卷积张量分为一小块的卷积，在小块的卷积中使用指针进行相应的读取数据在进行运算。

2.7 prefetch 预取技术

现代 CPU 架构中的一个重要优化是软件预取（software prefetching)，旨在减少缓存未命中的延迟开销。意思大概就是 cpu 在内存中读取数据会以块的方式读取，会连续的读取地址靠近的多个数据，将数据放在 cache 中，在读取一个数据后，它相邻的数据也会被放入 cache 中，这个优化主要是减少 cache 不命中的次数。

2.8 并行策略

将张量切片，同时运行不同的分块后的数据，这样可以同时进行多次卷积运算来提高卷积的速度。

2.9 Layer Fusion（层融合）

现代的深度神经网络架构不仅包含卷积层，还包含诸如 **ReLU**、池化、局部响应归一化（**LRN**）、归一化和偏置等层。其中一些层可以通过将函数 **L()** 应用于张量来实现，这些非卷积层通常具有较低的操作强度，因此它们受到带宽限制。在我们的框架中，我们识别并利用层融合的机会，即我们将这些不同的操作层分解为对子张量进行操作的形式，并在涉及的数据在缓存中热时应用它们，例如由于卷积操作而热化的数据。通过利用这种时间局部性，我们节省了这些层本应消耗的内存带宽.

2.10 Kernel Streams

这一部分大概就是在分块的时候，如按照 4 分块，当遇到 4 无法被整除的时候则需要采取不同的策略去进行卷积，比如，按照输出张量的高度分块，当高度为 17 时，最后一个微内核，输出张量的高度为 1，这个时候就不能按照高度为 4 的来处理，就需要换种方式处理。

3 实验部分

本周实验就是把上周优化给分开，分别测试了 gflop，因为分块的时候没有做不能整除的处理，所以设置的张量的维度都是可以直接分块。

3.1 实验环境

- 系统: Ubantu 22.01
- gcc version : 9.5.0
- 优化选项: -O2
- cpu:AMD Ryzen 7 6800H 3.20GHz

3.2 数据

- 存储类：Tensor1D
- inputTensor: 1,4,227,227
- filterTensor: 96,4,12,12
- outputTensor: 1,96,216,216
- stride : 1

3.3 index-opt

因为代码比较长，我用截图的方式展示代码。
无优化索引计算次数 $38N^7$

```cpp
template<typename T>
void directConvolution_tensor(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {
    for (int64_t i = 0; i < C.num_batch(); ++i) {
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            for (int64_t m = 0; m < C.num_height(); ++m) {
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            // index_count 38N^7
                            // 每个数据读取需要9次运算 每个循环读写4次数据 加上 m * s & n * s 总共38N^7
                            for (int64_t v = 0; v < B.num_width(); ++v) {
                                // 下限 2xN^7 / 4x8xN^7 = 1/16
                                C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

图 1: opt-none

张新鹏实现的索引优化索引计算次数 $2xN + 2xN^2 + 2xN^3 + 4xN^4 + 2xN^5 + 3 * N^6 + 7 * N^7$

```
template<typename T>
void directConvolution_tensor_indexOpt(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {
    cout<<A(0,0,0,0)<<endl;
    double *A1 = &A(0,0,0,0);
    double *B1 = &B(0,0,0,0);
    double *C1 = &C(0,0,0,0);
    //cout<A1<<endl;
    int64_t C_cwh = C.num_channel()*C.num_height()*C.num_width();
    int64_t C_wh = C.num_width()*C.num_height();
    int64_t C_w = C.num_width();

    int64_t A_cwh = A.num_channel()*A.num_width()*A.num_height();
    int64_t A_wh = A.num_width()*A.num_height();
    int64_t A_w = A.num_width();
    int64_t B_cwh = B.num_channel()*B.num_width()*B.num_height();
    int64_t B_wh = B.num_width()*B.num_height();
    int64_t B_w = B.num_width();
    int64_t ms,ns,a_index,a_index_i,a_index_r,a_index_ms,b_index,b_index_r,b_index_u,b_index_j,c_index,c_index_i,c_index_j,c_index_m;
    // 2xN + 2xN^2 + 2x N^3 + 4x N^4 + 2xN^5 + 3*N^6 + 10*N^7
    for (int64_t i = 0; i < C.num_batch(); ++i) {
        a_index_i = i * A_cwh; c_index_i = i * C_cwh;                   // 2*N
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            c_index_j = j * C_wh; b_index_j = j * B_cwh;                // 2 * N^2
            for (int64_t m = 0; m < C.num_height(); ++m) {
                c_index_m = m * C_w;                                    // 2 * N^3
                 ms = m*s;
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    c_index = c_index_i+c_index_j+c_index_m + n;                                    // 4 * N^4
                    ns = n * s;
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        a_index_r = r * A_wh; b_index_r = r * B_wh; // 2 * N^5
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            a_index_ms = (ms+u) * A_w; b_index_u = u * B_w;  // 3 *N^6
                            for (int64_t v = 0; v < B.num_width(); ++v) {
                                a_index = a_index_i + a_index_r + a_index_ms + ns + v;
                                b_index = b_index_j + b_index_r + b_index_u + v;
                                // 下限 2xN^7 / 4x8xN^7 = 1/16
                                //C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);
                                *(C1+c_index) += *(A1+a_index) * *(B1+b_index);  // 10 * N^7
                            }
                        }
                    }
                }
            }
        }
    }
}
```

图 2: index-opt-1

上周实现的索引优化索引计算次数 $29N^6 + 3 * N^7$

```cpp
template<typename T>
void directConvolution_tensor_indexOpt_2(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {

    for (int64_t i = 0; i < C.num_batch(); ++i) {
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            for (int64_t m = 0; m < C.num_height(); ++m) {
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            // index_count 38N^7
                            // 每个数据读取需要9次运算 每个循环读写4次数据 加上 m * s & n * s 总共38N^7
                                // 下限 2xN^7 / 4x8xN^7 = 1/16
                                // 27N^6 + 2N^6 = 29N^6 + 3*N^7
                                AddDot(B.num_width(),&A(i, r, m * s + u, n * s),&B(j, r, u, 0),&C(i,j,m,n));
                                //C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);

                        }
                    }
                }
            }
        }
    }
}
```

图 3: index-opt-2

上周实现的索引优化基础上再次优化索引计算次数 $29N^6$

```cpp
template<typename T>
void directConvolution_tensor_indexOpt_3(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {
    int64_t ms,ns;
    for (int64_t i = 0; i < C.num_batch(); ++i) {
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            for (int64_t m = 0; m < C.num_height(); ++m) {
                ms = m * s;
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    ns = n * s;
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            // index_count 38N^7
                            // 每个数据读取需要9次运算 每个循环读写4次数据 加上 m * s & n * s 总共38N^7
                                // 下限 2xN^7 / 4x8xN^7 = 1/16
                                // 27N^6
                                AddDot(B.num_width(),&A(i, r, ms + u, ns),&B(j, r, u, 0),&C(i,j,m,n));
                                //C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);

                        }
                    }
                }
            }
        }
    }
}
```

图 4: index-opt-3

上周实现的索引和张新鹏的结合索引计算次数 $11N^6$

```cpp
template<typename T>
void directConvolution_tensor_indexOpt_4(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {
    double *A1 = &A(0,0,0,0);
    double *B1 = &B(0,0,0,0);
    double *C1 = &C(0,0,0,0);
    //cout<A1<<endl;
    int64_t C_cwh = C.num_channel()*C.num_height()*C.num_width();
    int64_t C_wh = C.num_width()*C.num_height();
    int64_t C_w = C.num_width();

    int64_t A_cwh = A.num_channel()*A.num_width()*A.num_height();
    int64_t A_wh = A.num_width()*A.num_height();
    int64_t A_w = A.num_width();
    int64_t B_cwh = B.num_channel()*B.num_width()*B.num_height();
    int64_t B_wh = B.num_width()*B.num_height();
    int64_t B_w = B.num_width();
    int64_t ms,ns,a_index,a_index_i,a_index_r,b_index,b_index_r,b_index_j,c_index,c_index_i,c_index_j,c_index_m;
    // 2xN + 2xN^2 + 2x N^3 + 4x N^4 + 2xN^5 + 3*N^6 + 10*N^7
    for (int64_t i = 0; i < C.num_batch(); ++i) {
        a_index_i = i * A_cwh; c_index_i = i * C_cwh;                   // 2*N
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            c_index_j = j * C_wh; b_index_j = j * B_cwh;                // 2 * N^2
            for (int64_t m = 0; m < C.num_height(); ++m) {
                c_index_m = m * C_w;                                    // 2 * N^3
                 ms = m*s;
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    c_index = c_index_i+c_index_j+c_index_m + n;                                        // 4 * N^4
                    ns = n * s;
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        a_index_r = r * A_wh; b_index_r = r * B_wh; // 2 * N^5
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            a_index = a_index_i + a_index_r + (ms+u) * A_w + ns;
                            b_index = b_index_j + b_index_r + u * B_w;  //11 * N^6
                                // 下限 2xN^7 / 4x8xN^7 = 1/16
                                //C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);
                                AddDot(B.num_width(),A1+a_index,B1+b_index,C1+c_index);
                        }
                    }
                }
            }
        }
    }
}
```

图 5: index-opt-4

glops 比较 分析：索引优化也可以在很大程度上提高效率，从 opt-1 和 opt-4 可以看出，opt-1 比较快，

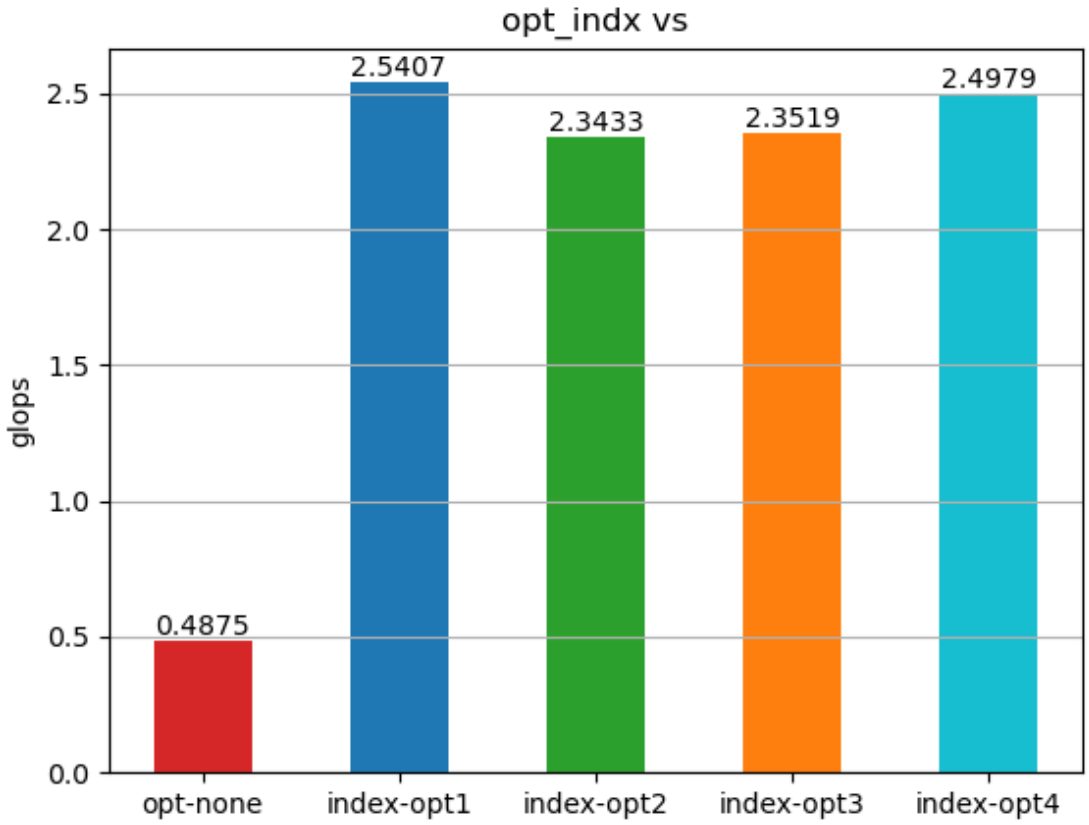

图 6: index-gflops

实质上两者差不多，只是对于指针读取数据的时候，一个是 *(a+1) 和 *a++，这两者的区别类似于 ++i 和 i++，后者都需要使用额外的寄存器去存储 a+1 后的值。

3.4 unrolling

这一部分的我的理解可能有点错误，因为我把寄存器分块和向量化都规划为 Unrolling，而从得出的实验结果看，寄存器分块和向量化只是单纯的分块并不会对算法有优化，这些应该结合其他的优化才可以对卷积有优化的作用，寄存器分块可以利用 simd 指令集来优化，而向量化则是可以结合并行化来优化。寄存器分块，按照输出张量的宽和高进行分块

```
template<typename T>
void directConvolution_tensor1x4(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {
    for (int64_t i = 0; i < C.num_batch(); ++i) {
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            for (int64_t m = 0; m < C.num_height(); ++m) {
                for (int64_t n = 0; n < C.num_width(); n+=4) {
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            // 40xB.nun_width()
                            for (int64_t v = 0; v < B.num_width(); ++v) {
                                // 下限 2xN^7 / 4x8xN^7 = 8/13*8 = 1/13
                                C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);
                                C(i, j, m, n + 1) += A(i, r, m * s + u, (n + 1) * s + v) * B(j, r, u, v);
                                C(i, j, m, n + 2) += A(i, r, m * s + u, (n + 2) * s + v) * B(j, r, u, v);
                                C(i, j, m, n + 3) += A(i, r, m * s + u, (n + 3) * s + v) * B(j, r, u, v);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

图 7: c-width

```cpp
template<typename T>
void directConvolution_tensor4x1(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {
    for (int64_t i = 0; i < C.num_batch(); ++i) {
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            for (int64_t m = 0; m < C.num_height(); m+=4) {
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            // 40xB.nun_width()
                            for (int64_t v = 0; v < B.num_width(); ++v) {
                                // 下限 2xN^7 / 4x8xN^7 = 1/16
                                C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);
                                C(i, j, m + 1, n) += A(i, r, (m + 1) * s + u, n * s + v) * B(j, r, u, v);
                                C(i, j, m + 2, n) += A(i, r, (m + 2) * s + u, n * s + v) * B(j, r, u, v);
                                C(i, j, m + 3, n) += A(i, r, (m + 3) * s + u, n * s + v) * B(j, r, u, v);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

图 8: c-height

```cpp
template<typename T>
void directConvolution_tensor_vectorization(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {
    for (int64_t i = 0; i < C.num_batch(); ++i) {
        for (int64_t j = 0; j < C.num_channel(); j+=4) {
            for (int64_t m = 0; m < C.num_height(); ++m) {
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            for (int64_t v = 0; v < B.num_width(); ++v) {
                                // 下限 2xN^7 / 4x8xN^7 = 1/16
                                C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);
                                C(i, j + 1, m, n) += A(i, r, m * s + u, n * s + v) * B(j + 1, r, u, v);
                                C(i, j + 2, m, n) += A(i, r, m * s + u, n * s + v) * B(j + 2, r, u, v);
                                C(i, j + 3, m, n) += A(i, r, m * s + u, n * s + v) * B(j + 3, r, u, v);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

图 9: c-channel

```cpp
template<typename T>
void directConvolution_tensor1x1_4(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {
    for (int64_t i = 0; i < C.num_batch(); ++i) {
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            for (int64_t m = 0; m < C.num_height(); ++m) {
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            // 40xB.nun_width()
                            for (int64_t v = 0; v < B.num_width(); v+=4) {
                                // 下限 2xN^7 / 4x8xN^7 = 8*8/10 = 1/10
                                C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v)
                                               + A(i, r, m * s + u, n * s + v + 1) * B(j, r, u, v + 1)
                                               + A(i, r, m * s + u, n * s + v + 2) * B(j, r, u, v + 2)
                                               + A(i, r, m * s + u, n * s + v + 3) * B(j, r, u, v + 3);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

图 10: b-width

```cpp
template<typename T>
void directConvolution_tensor_vectorization_2(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C,int64_t s) {
    for (int64_t i = 0; i < C.num_batch(); ++i) {
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            for (int64_t m = 0; m < C.num_height(); ++m) {
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    for (int64_t r = 0; r < B.num_channel(); r+=4) {
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            for (int64_t v = 0; v < B.num_width(); ++v) {
                                // 下限 2xN^7 / 4x8xN^7 = 1/16
                                C(i, j, m, n) += A(i, r, m * s + u, n * s + v) * B(j, r, u, v)
                                               + A(i, r + 1, m * s + u, n * s + v) * B(j, r + 1, u, v)
                                               + A(i, r + 2, m * s + u, n * s + v) * B(j, r + 2, u, v)
                                               + A(i, r + 3, m * s + u, n * s + v) * B(j, r + 3, u, v);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

图 11: b-channel

gflops 比较

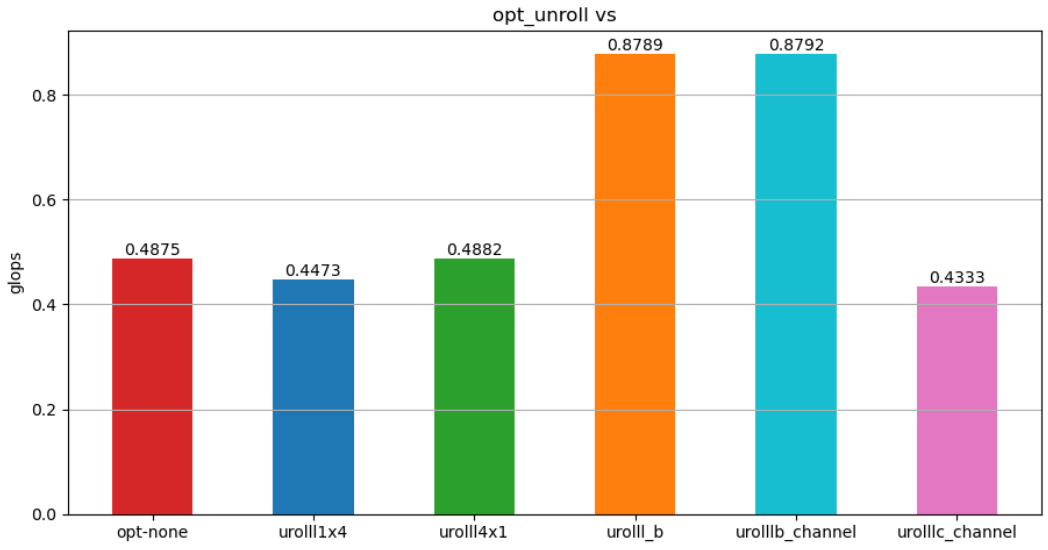

图 12: gflops

分析: 从图中可以得，向量化和寄存器分块都没有优化的作用，甚至 glops 没有没开优化的时候高，而 unrolling 的优化速度大概提高了 2 倍。

3.5 hoisting 优化

因为只有一种，所以只给了代码，后续会把 Unrolling 和 hoisting 结合对比。

```
template<typename T>
void directConvolution_tensor_hoisting(Tensor<T>& A, Tensor<T>& B, Tensor<T>& C, int64_t s) {
    T t;

    for (int64_t i = 0; i < C.num_batch(); ++i) {
        for (int64_t j = 0; j < C.num_channel(); ++j) {
            for (int64_t m = 0; m < C.num_height(); ++m) {
                for (int64_t n = 0; n < C.num_width(); ++n) {
                    t = 0.0;
                    for (int64_t r = 0; r < B.num_channel(); ++r) {
                        for (int64_t u = 0; u < B.num_height(); ++u) {
                            // 40xB.nun_width()
                            for (int64_t v = 0; v < B.num_width(); ++v) {
                                // 下限 2xN^7 / 4x8xN^7 = 1/8
                                t += A(i, r, m * s + u, n * s + v) * B(j, r, u, v);
                            }
                        }
                    }
                    C(i,j,m,n) = t;
                }
            }
        }
    }
}
```

图 13: hoisting

3.6 unrolling+hoisting

把两种优化结合起来 分析: 可以看出两者结合后的效果优化效果更好，而对于 output 高度宽度寄存器分块提升非常明显。读取 hoisting 优化的这些数据的次数变少了，由于循环结构可能读取这些所花的代价比较高。

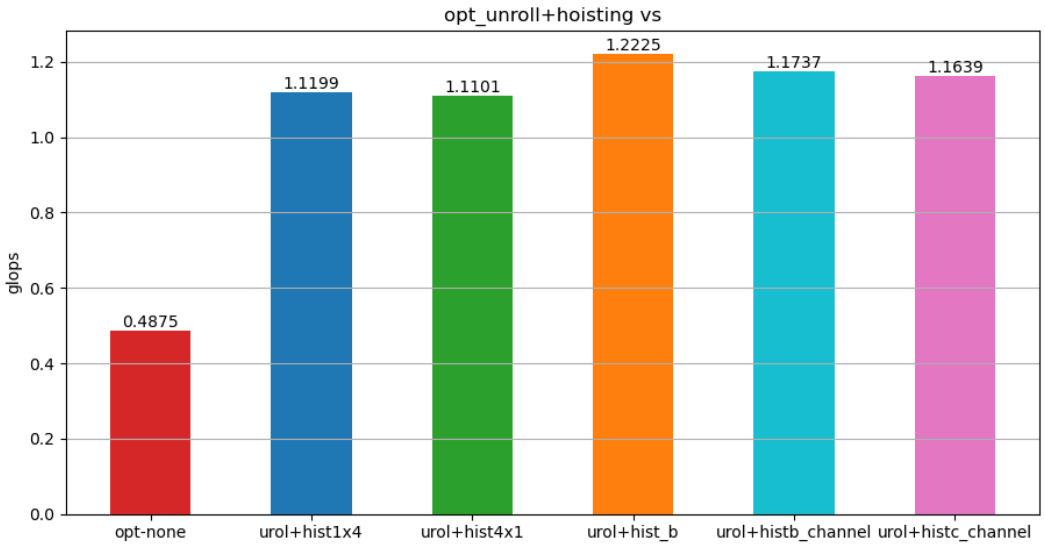

图 14: unrolling+hoisting

3.7 roofline

给出了优化等级在 -O2 情况下的 roofline，以及不同优化在 roofline 上表现，刚开始选取的优化图中下标所示，各柱状图的颜色对于 roofline 中点的颜色。

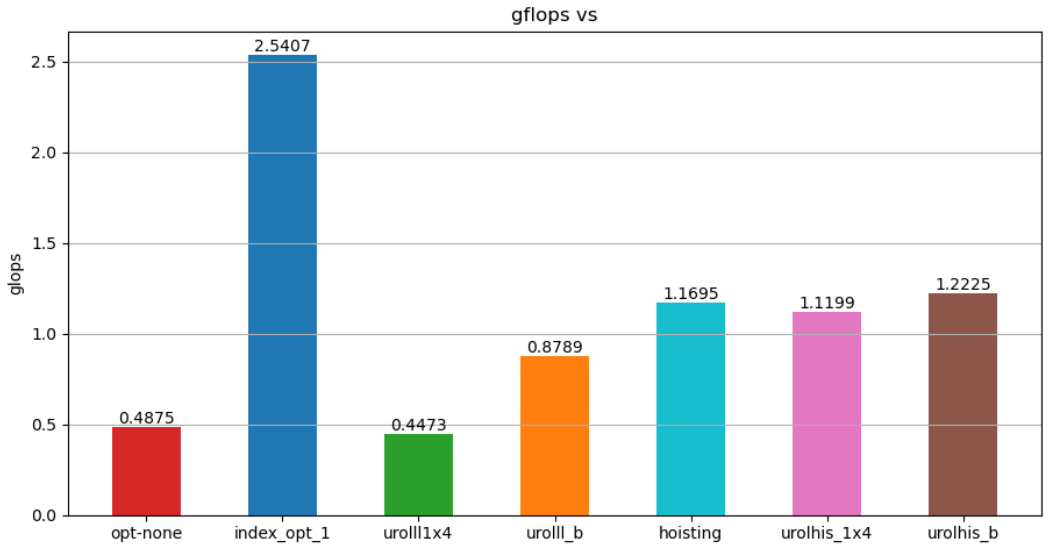

图 15: gflop-bar

4 总结

从这几周的工作结合论文来看，gemm 在 cpu 上的优化目前我只做到了向量化和寄存器分块，还有应用了 simd 指令集优化 gemm 算法，本周实现的索引优化和 unrolling 还有 hoisting 可以结合使用这样的效果更好，而 simd 应该是需要结合 hoisting 和向量化以及寄存器分块一起使用，上周张新鹏所做的事是更改循环顺序，从这周论文可以知道循环顺序决定了张量的访问方式，并影响相应数据的重用。合理的分配循环顺序能更有利于增加卷积的速度。对于分块后除不尽的处理方式还没实现，下周的任务打算使用前几周学到的优化算法去尽可能的让算法的速度更快，并结合 roofline 去确定优化的方向，因为前几周使用的都是理想的维度，下周结合 conv1 去实现算法优化。

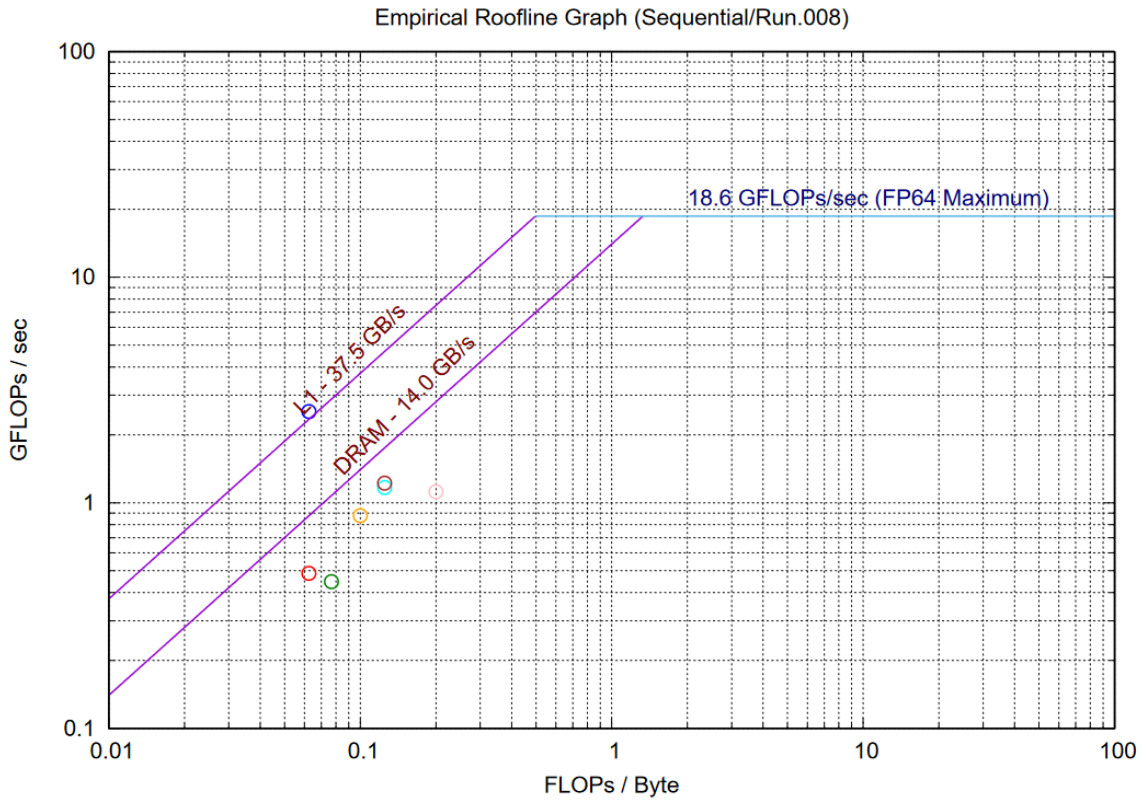

图 16: roofline-1

为了便于分析，我选择了无优化，unrollb,hoisting, unrollb+hoisting

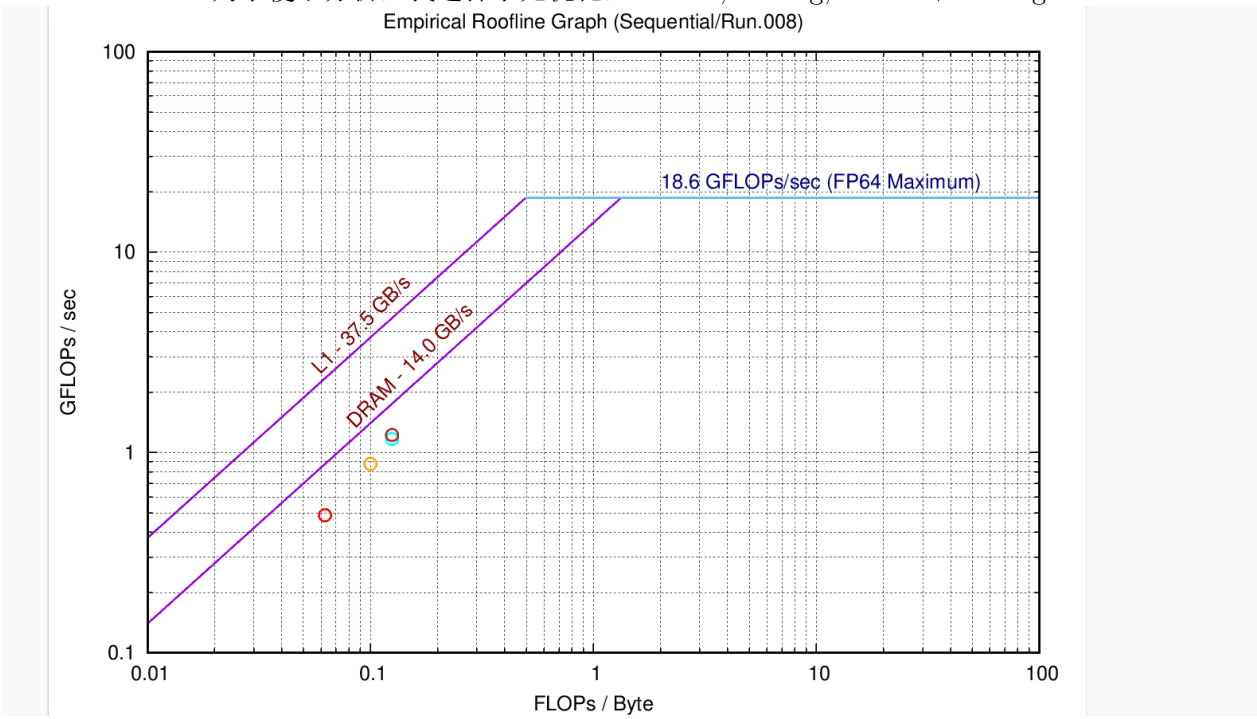

图 17: roofline2

从图中红点是没有做优化的点，图中 4 个点都处于 bandwidth-bound 区域，这个时候可以选择继续增大算术强度来提高 gflops.