week-18

JX-Ma

2024/1/20

1 本周工作

把前几周的学到的优化结合 roofline model 优化了 12 个 benchmark;

2 实验准备

2.1 使用 simd 存储数据

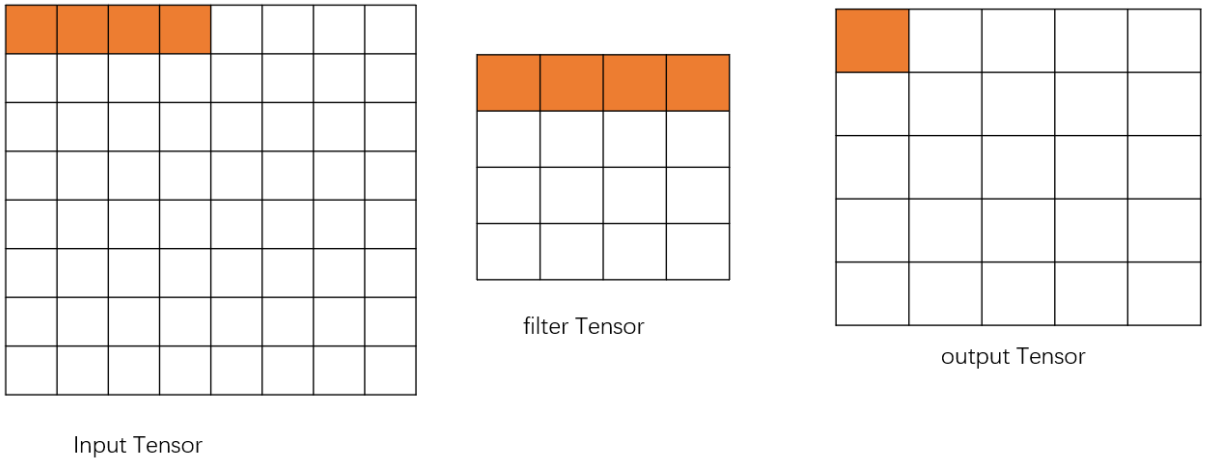

图 1: simd1

第一种 simd 存储数据的方式，输入张量存入的是每个窗口内连续的元素，卷积核存储的是连续几个数据。使用一个额外的 simd 寄存器去存储中间的结果，把中间结果内所有数据相加才是对应输出张量元素的值。
示例代码如下

```
inline void ElementMul1x8(float *a, float *b, __m256 &c)
```

```
{
        __m256 input = _mm256_loadu_ps(a);
        __m256 fiter = _mm256_loadu_ps(b);
        c = _mm256_fmadd_ps(input,fiter,c);

}
```

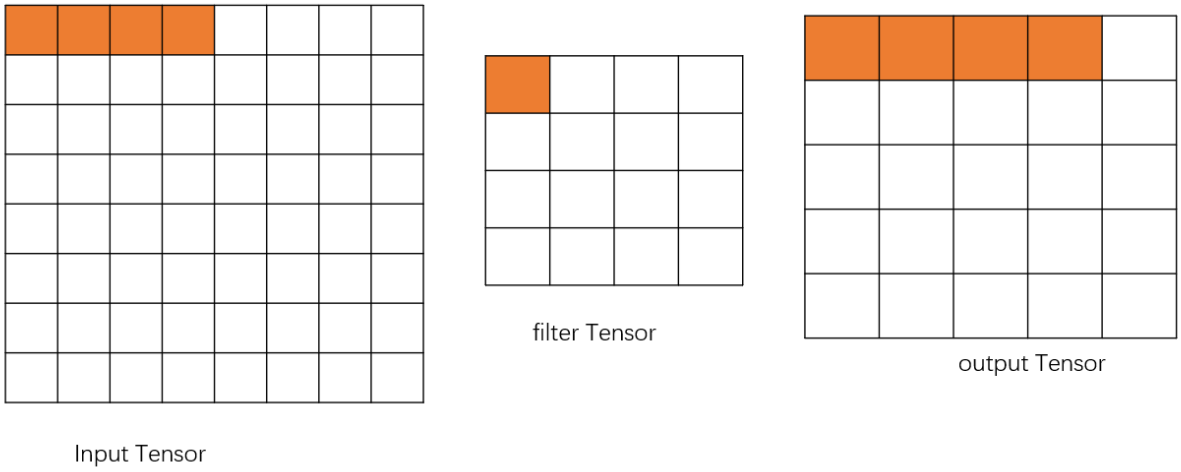

图 2: simd2

第二种只适用于步长为 1 的情况下，这个时候输入张量存储的是输入张量连续窗口内的第 i 个元素，卷积核存储的是 n 个相同的元素值，输出张量存储的是靠近的几个连续的元素。
示例代码如下:

```
inline void RBMul1x8(float *a,float *b,__m256 &c)
{
    // a0_7p代表存储的是输入张量连续8个窗口的第一个元素
    __m256 a0_7p,b0;
    // 加载输入张量各个窗口的元素
    a0_7p = _mm256_loadu_ps(a);
    // 加载卷积核的元素  将所有元素都等于b
    b0 = _mm256_set1_ps(*b);

    //calculate
    c = _mm256_fmadd_ps(a0_7p,b0,c);

}
```

2.2 算术强度计算

不进行分块的计算
假设卷积核 4x4，采用 float 存储，步长为 1
flops=4x4x2
bytes: input:4x4x4 fiter: 4x4x4 output 1x4
AI = flops/bytes;
如果使用了 128 位 simd 存储数据 flops 不变, 采用第一种 simd 存储数据的方法
bytes: input: 1x4x4 fiter: 1x4x4 output 1x4

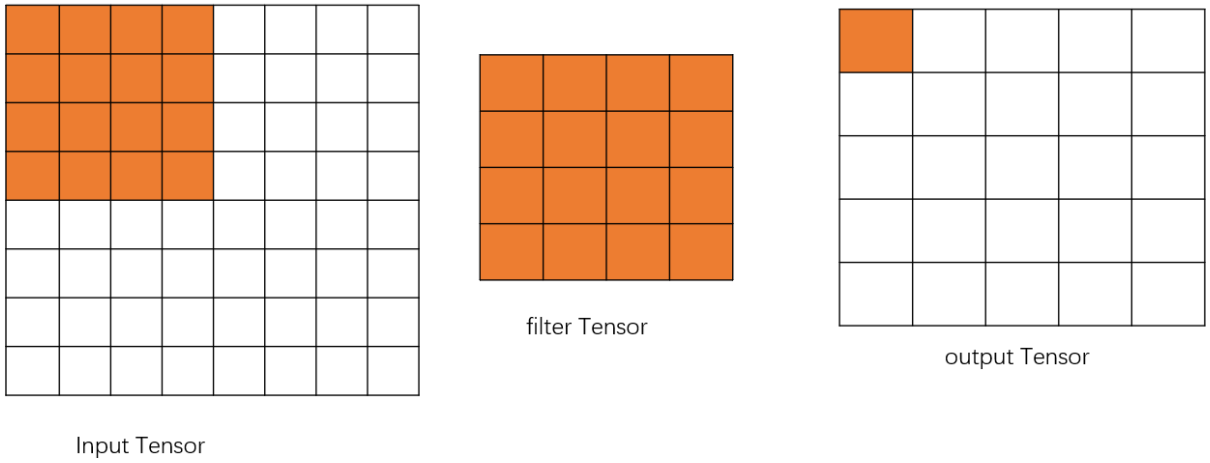

图 3: AI

1x4 分块计算
flops: 4x4x2x4
bytes: input: 7x4x4 fiter 4x4x4 output 4x4
使用第一种存储数据的方式去存 bytes –> input 4x4x4 fiter:1x4x4 output 4x4
第二种存储: bytes –> input 4x4x4 fiter 4x4 output 1x4

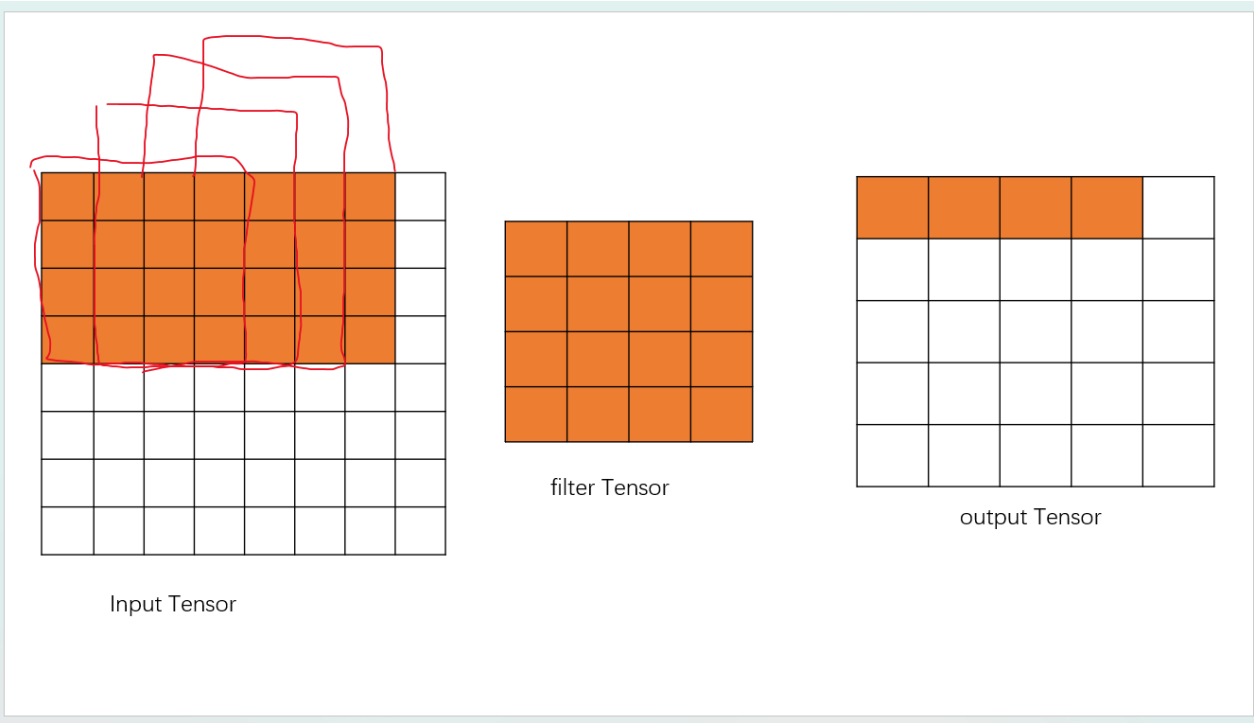

图 4: AI

3 实验部分

3.1 实验环境

- 系统: Ubantu 22.01
- gcc version : 9.5.0
- 优化选项: -O3 -fsse -favx2 -fmadd
- cpu:AMD Ryzen 7 6800H 3.20GHz

3.2 CONV1

- input(10,3,227,227)
- fiter(96,3,11,11)
- outpuy(10,96,55,55)
- stride:4

因为 index-hoist 可以直接加上去，这次实验所做的优化都是建立在索引优化之上的
首先实现的是 hoist-output-element 数据如下:

```
/**
 * AI flop/bytes = 0.249
 * flop: 11x11x2
 * bytes: input 11x11x4 output 1x4 fiter 11x11x4
 * gflops:3.1003
*/
```

此时展示的算术强度较低，gflops 也较低。
第二个优化是使用了 simd，因为步长 >1，所有我采用第一种存储数据的方式，使用 128 位寄存器去存储数据，因为卷积核的宽度不能被 4 整除，剩余数据处理就单个数据处理。

```
/**
 * AI flop/bytes = 0.54
 * flop: 11x11x2
 * bytes: input 5x11x4 fiter 5x11x4 output 1x4
 * gflops: 9.05184
*/
```

第三个优化是使用 padding 填充卷积核确保卷积核的宽度能被 4 整除，然后为了保证结果正确需要对输入张量的宽度也进行填充，运行后结果

```
/**
 * AI flop/bytes = 0.900
 * flop: 11x11x2
 * bytes: input 3x11x4 fiter 3x11x4 output 1x4
 * gflops:13.0265
*/
```

第四个优化建立 padding 填充数据的基础上，对于卷积核宽度为 12，我使用了 1 个 256 位寄存器和一个 128 位寄存器存储这 12 个数据，实验结果如下

```
/**
 * AI flop/bytes = 1.13
 * flop: 11x11x2
 * bytes: input 2x11x4 fiter 2x11x4 output 1x4
 * gflops:25.5514
*/
```

第五个优化同样是使用了 padding 填充，在第四个优化的基础上，对输出张量的宽度进行 1x4 分块，结果如下

```
/**
 * AI flop/bytes = 2.63
 * flop: 11x11x2x4
 * bytes: input 6x11x4 fiter 2x11x4 output 4x4
```

```
     * gflops:37.2744 —> best gflops
    */
```

后面 2 种优化分别在第四个优化的基础上，对输出张量进行 1x7，和 1x8 分块

```
    /**
     * AI flop/bytes = 3.30
     * flop: 11x11x2x7
     * bytes: input 9x11x4 fiter 2x11x4 output 7x4
     * gflops:37.2058
    */


/**
 * AI flop/bytes =       3.46
 * flop: 11x11x2x8
 * bytes: input 10x11x4 fiter 2x11x4 output 8x4
 * gflops:33.0729
*/
```

实验结果和 roofline 如下:
下标解释: none: 什么优化都没有的直接卷积
opt1: hoist-output-element
opt2: unroll1x4, 使用向量寄存器存储连续 4 个元素
opt3: 在 opt2 基础上，使用 padding 填充
opt4: unroll1x12, 使用一个 256，一个 128 存储连续的 12 个元素
opt5: 在 opt4 基础上，对输出张量进行 1x4 分块
opt6: 在 opt4 基础上，对输出张量进行 1x7 分块
opt7: 在 opt4 基础上，对输出张量进行 1x8 分块

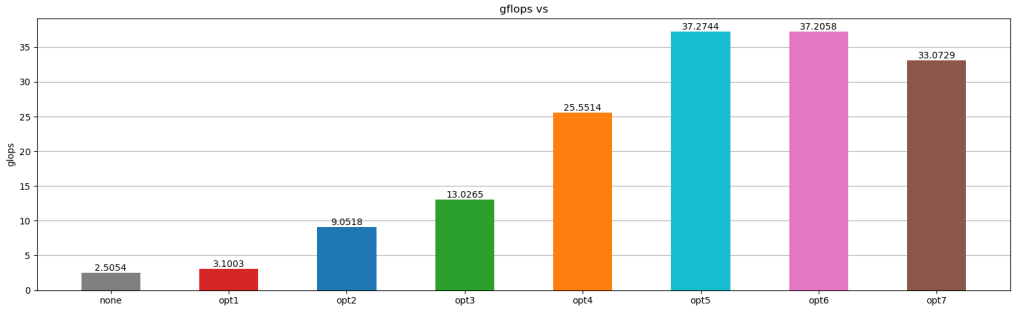

图 5: CONV1-gflops

roofline model 点的颜色对应的是 gflops bar 的颜色
后面写了 4 个算法用来验证优化的方向:
1. 之后在 unroll1x12 的基础上尝试着调整下循环顺序，

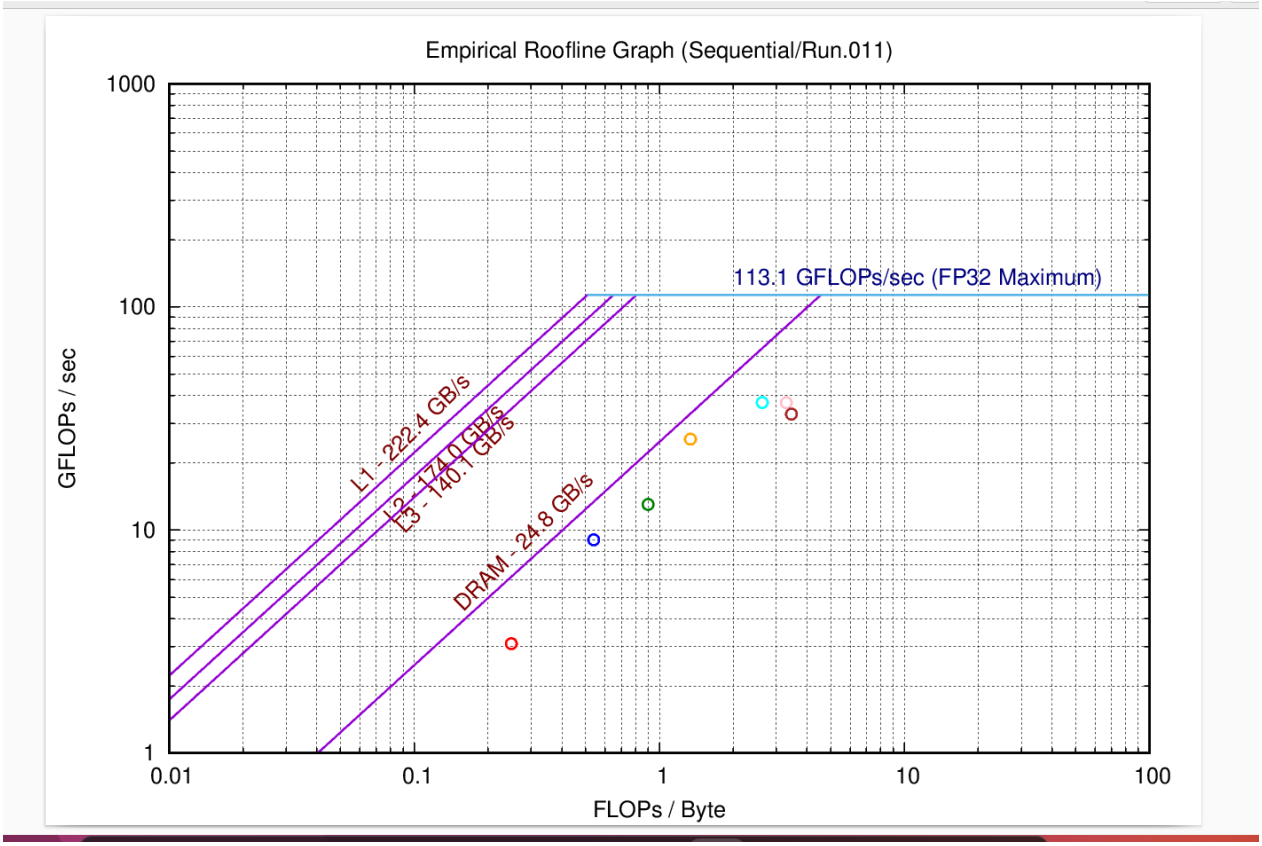

图 6: CONV1-roofline

2. 在 unroll1x12 基础上，对卷积核的按照 4 进行分块
3. 在 unroll1x12,RB1x4 基础上，对卷积核高度按照 2 进行分块，
4. 在 unroll1x12,RB1x4 基础上，对卷积核高度按照 4 进行分块

图中 opt1 对应的是 Unroll1x12

opt2: 调换顺换顺序

opt3: 在 unroll1x12 基础上，对卷积核的按照 4 进行分块

opt4 unroll1x12-RB1x4

opt5: 在 unroll1x12,RB1x4 基础上，对卷积核高度按照 2 进行分块，

opt6: 在 unroll1x12,RB1x4 基础上，对卷积核高度按照 4 进行分块

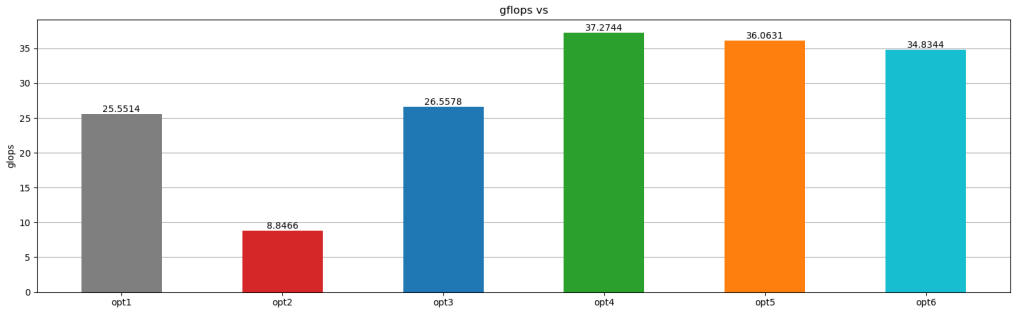

图 7: CONV1-2-gflops

对于调整循环顺序来说，因为我们 hoist 的时候需要使用临时变量去存储，而如果把输出张量的宽放在最里面一层，我们就需要很多临时变量去存储，或者我们可以在最里面循环计算完后直接写入输出张量中，但是这样会增加很多写操作，我们对输出张量的宽度进行分块，也可以类似于将输出张量的宽度放在最里面一层，因为按照输出张量宽度分块，可以在最里面一层循环尽可能的优先访问输出张量宽度上连续的元素。

对于第二种对卷积核高度分块来说，首先算术强度不会改变，这样做的好处是读取卷积核数据 cache 命中率更高，访问卷积核不同的高度时，对应我们也需要访问输出张量不同高度的元素时，当输出张量宽度很大，cache 命中率会降低

第三四种，原因与上面第二种一样，而且 simd 寄存器数量有限，故我们分块顺序是 unroll-fiter-width, 然后对寄存器尽可能大的分块。

3.3 CONV2-12

unroll 指的是使用 simd 存储的数据大小，RB1xi 代表按照输出张量宽度 i 分块

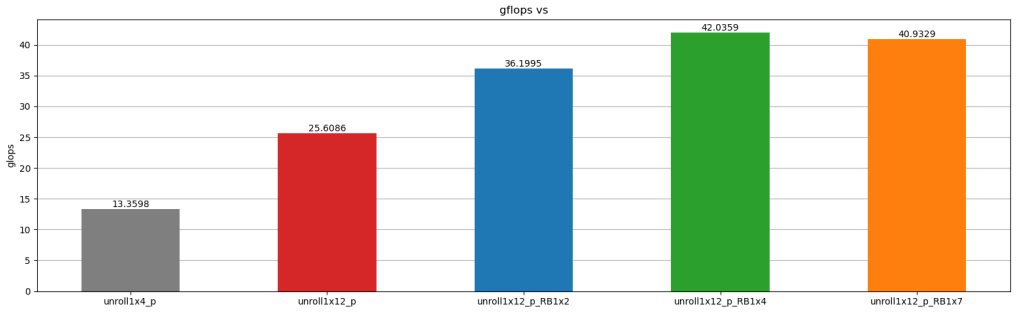

图 8: CONV2-gflops

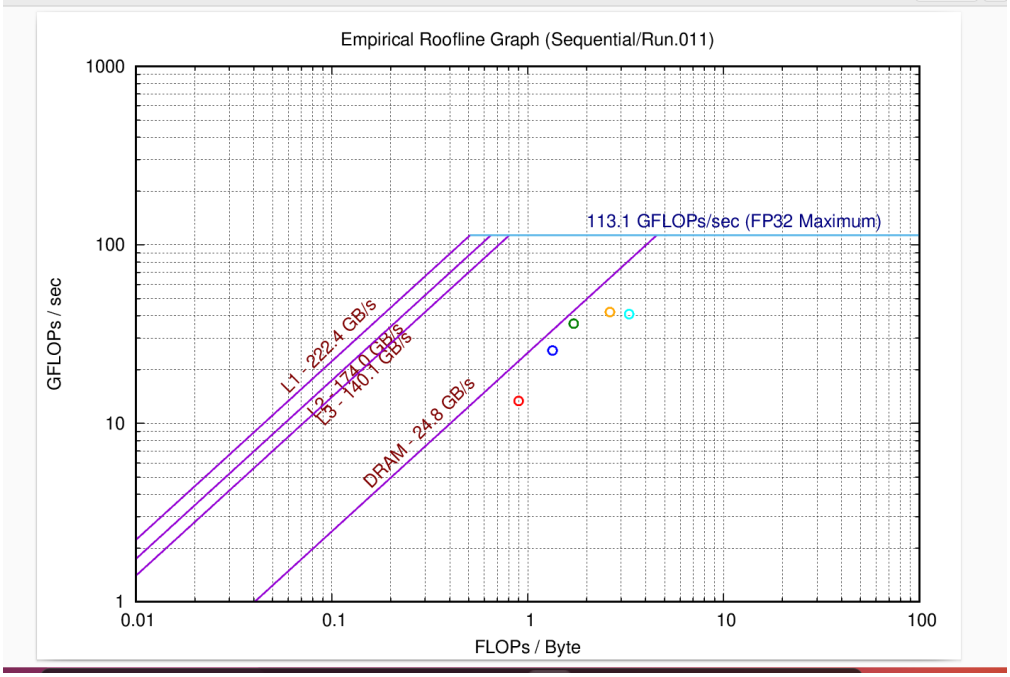

图 9: CONV2-roofline

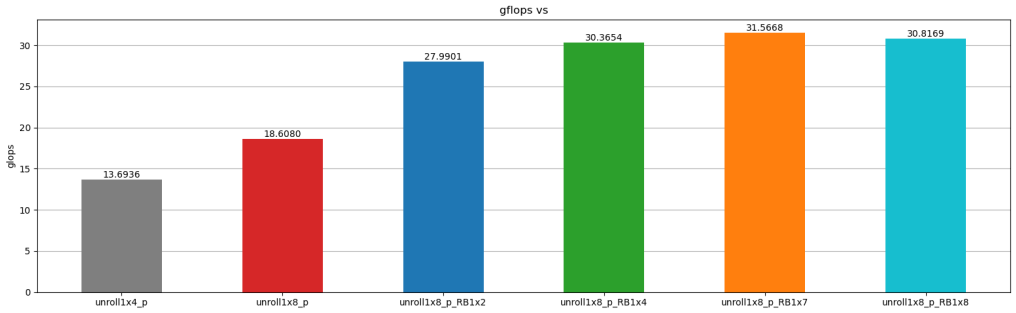

图 10: CONV3-gflops

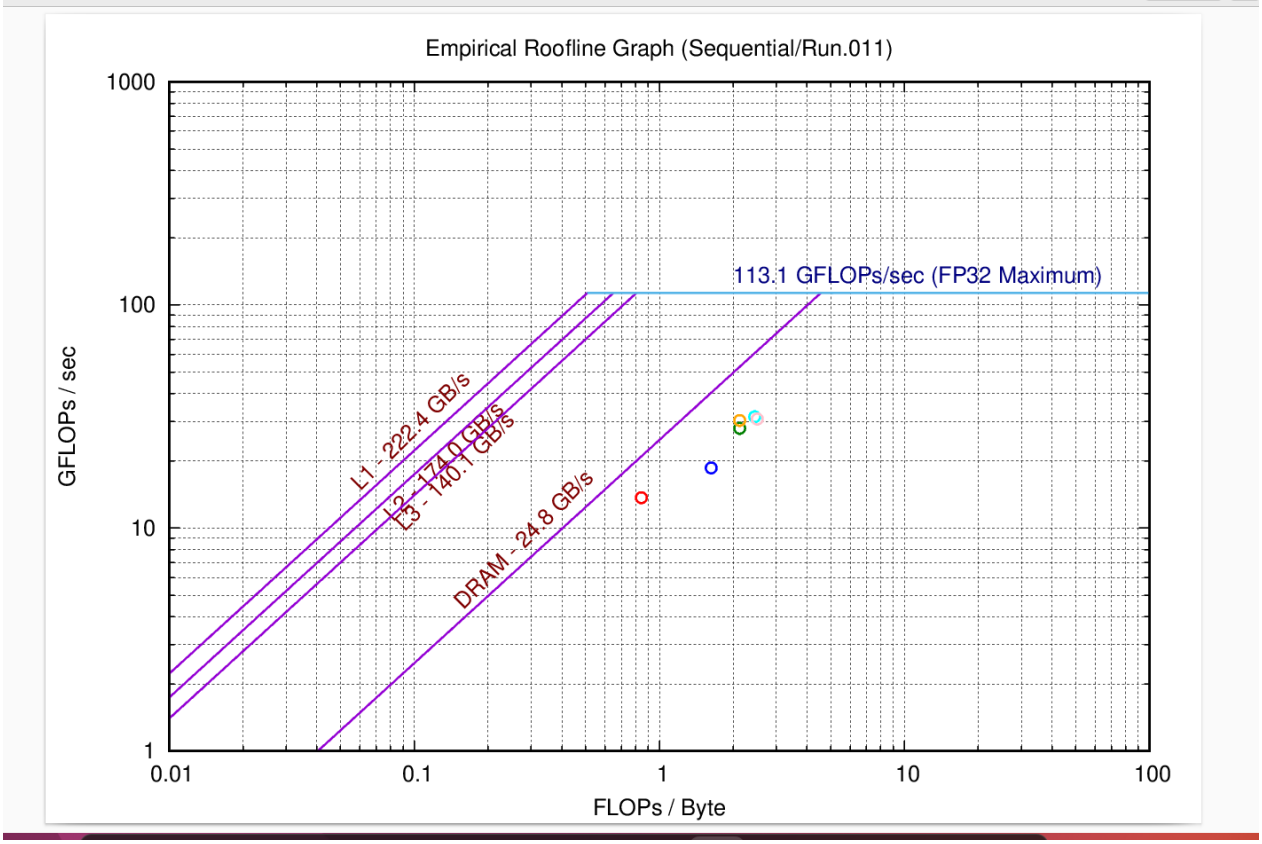

图 11: CONV3-roofline

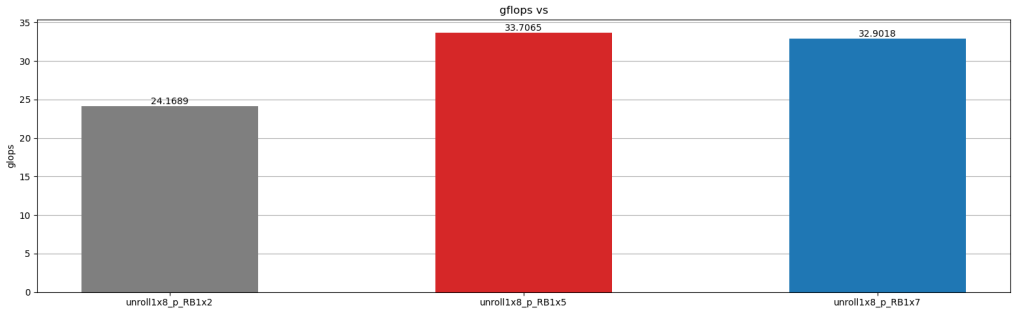

图 12: CONV4-gflops

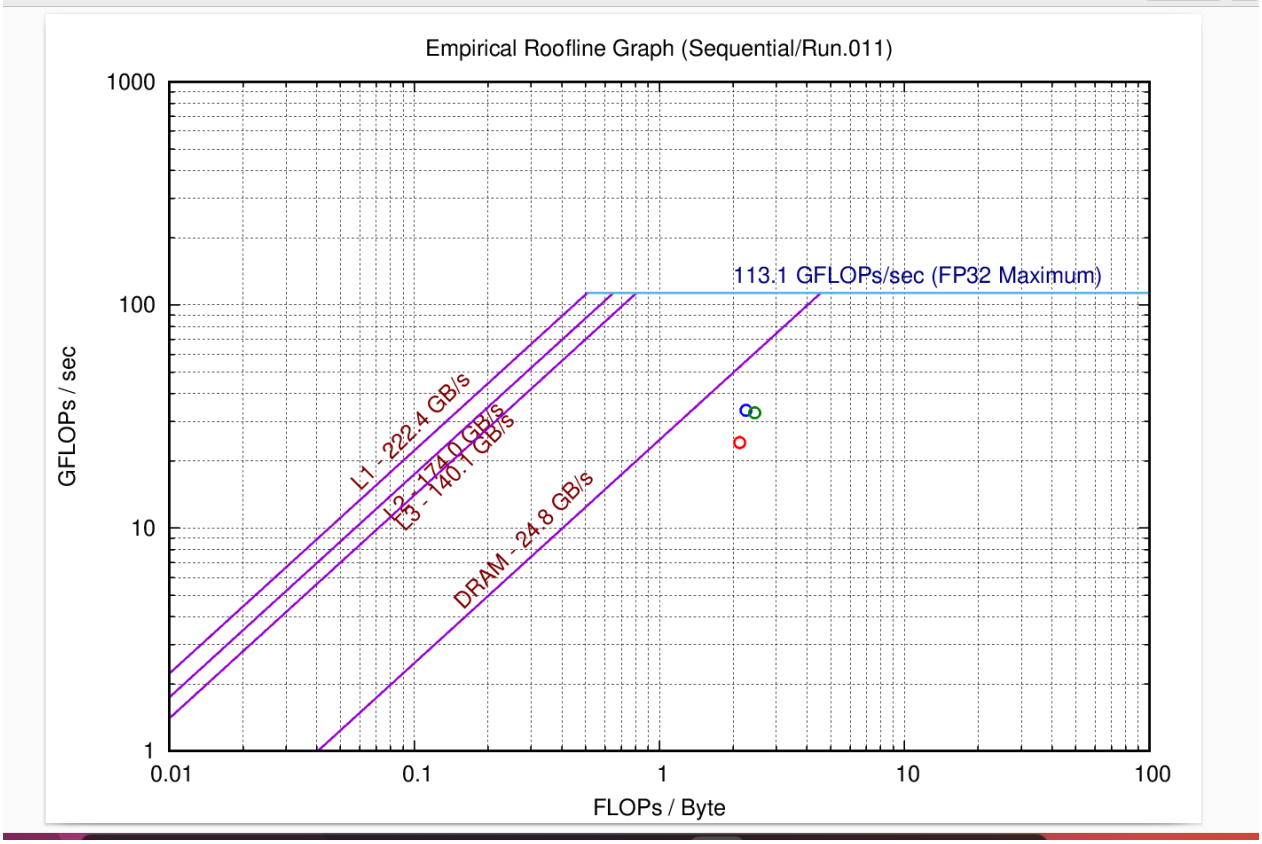

图 13: CONV4-roofline

因为步长为 1，所以这里我们采取两种存储数据方式优化,simd1 代表第一种存储数据的方式

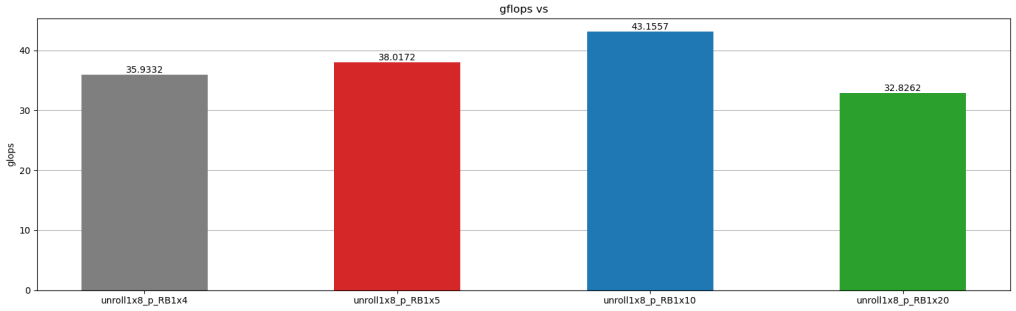

图 14: CONV5-simd1-gflops

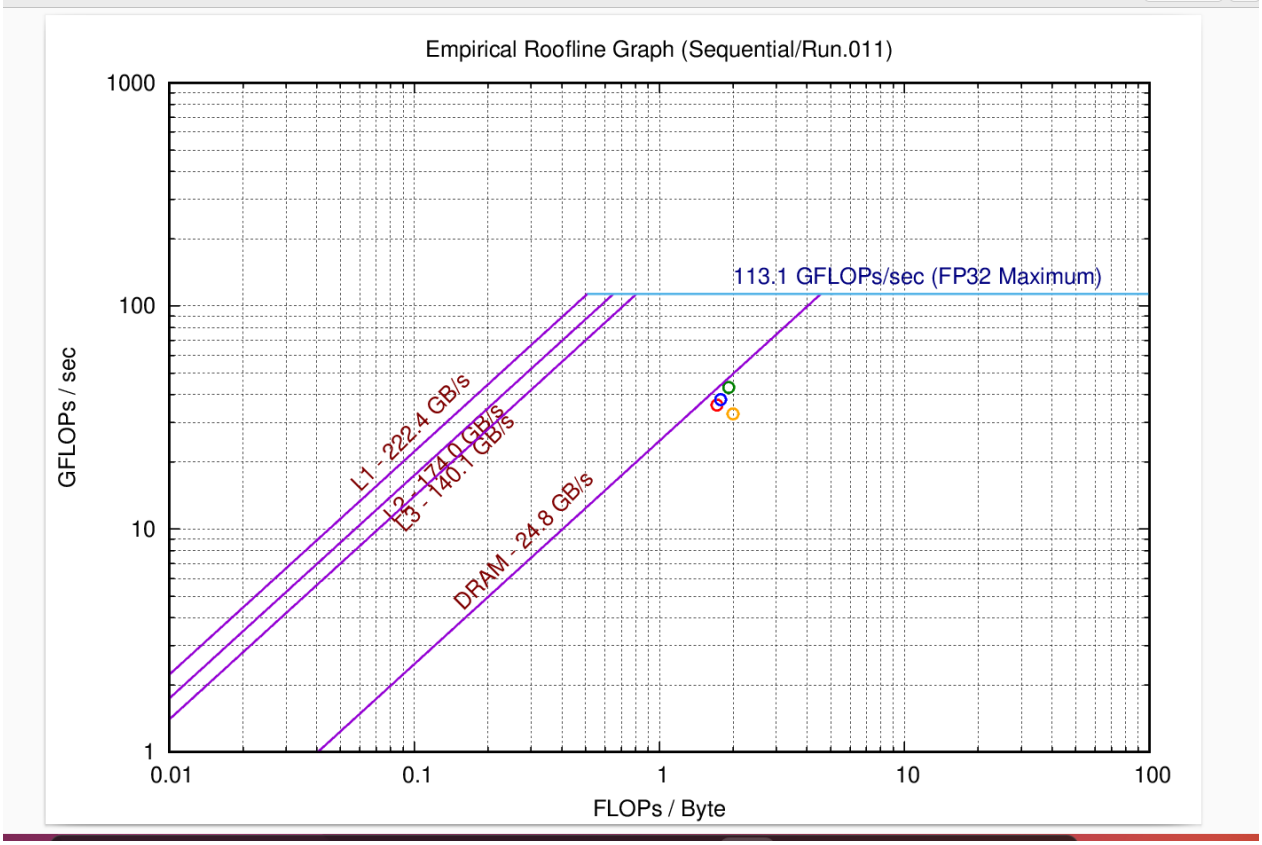

图 15: CONV5-simd1-roofline

simd2 代表第 2 种存储数据的方式

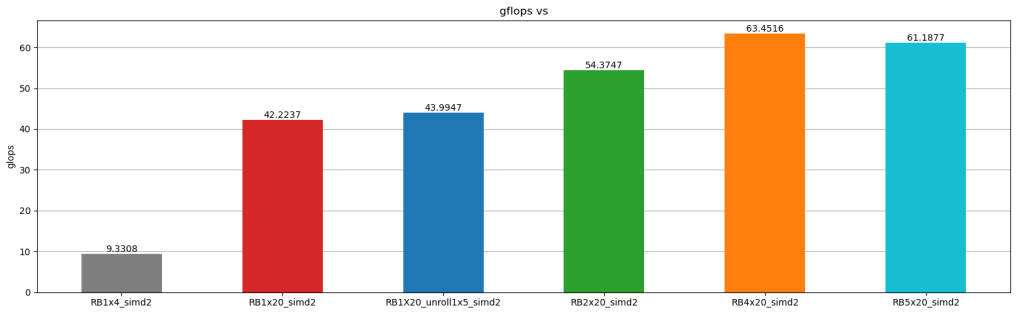

图 16: CONV5-simd2-gflops

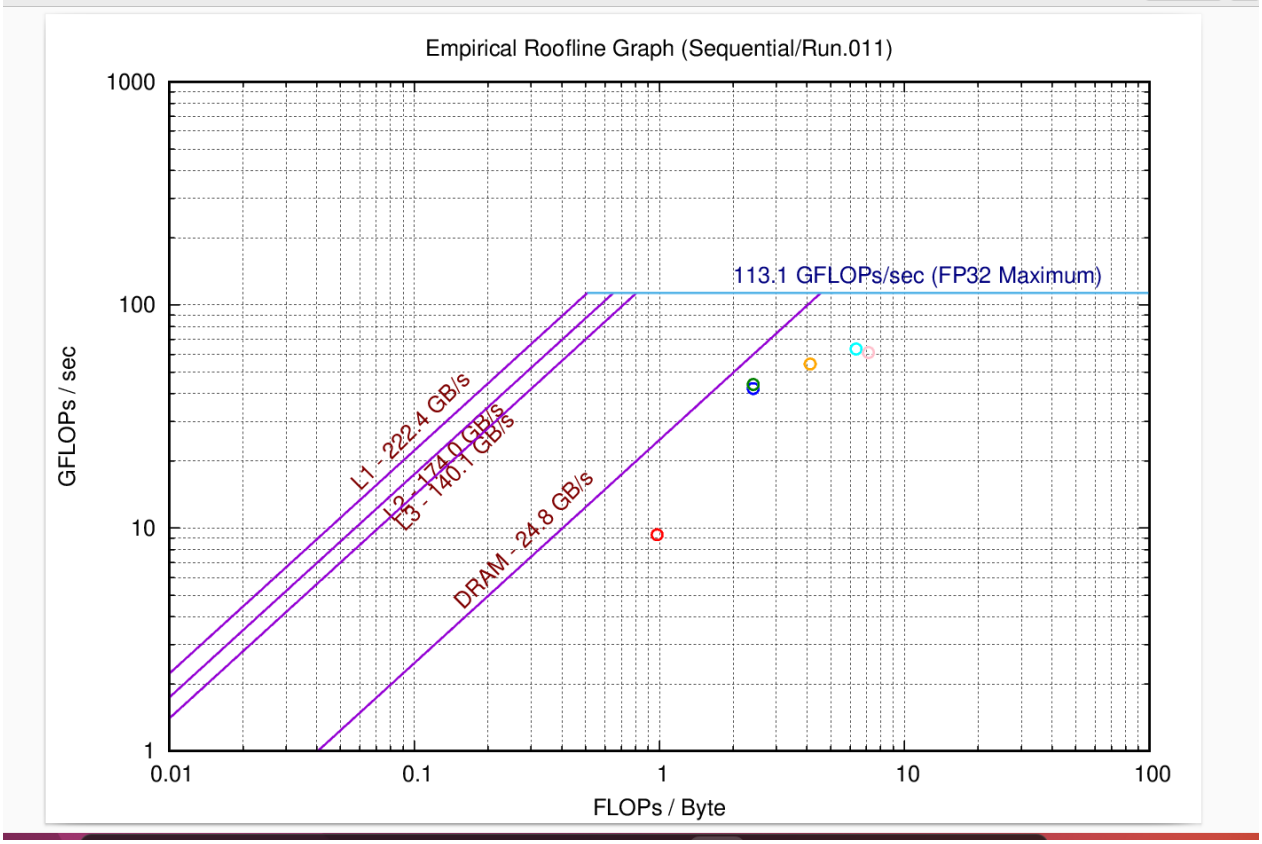

图 17: CONV5-simd2-roofline

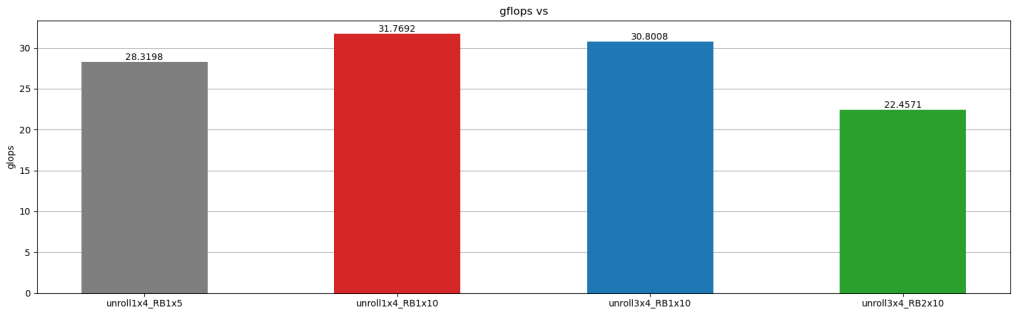

图 18: CONV6-simd1-gflops

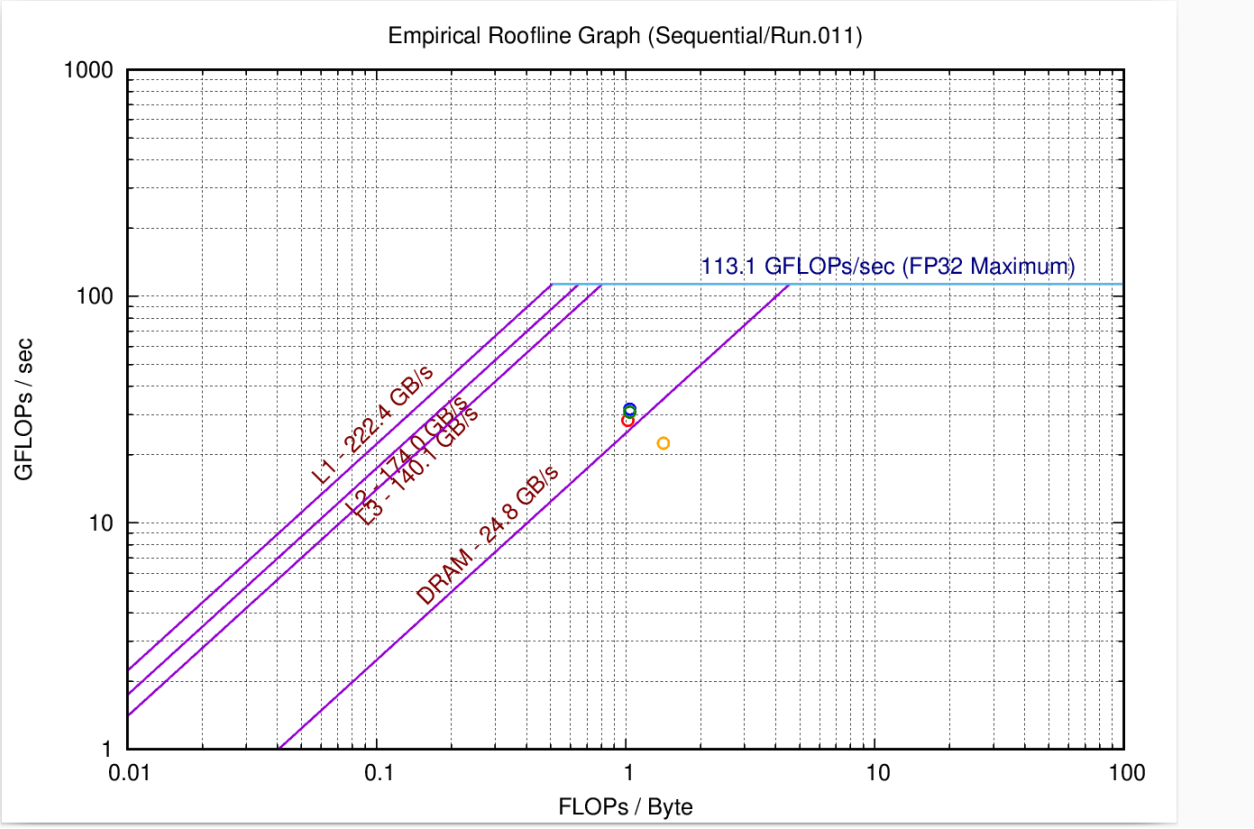

图 19: CONV6-simd1-roofline

simd2 代表第 2 种存储数据的方式

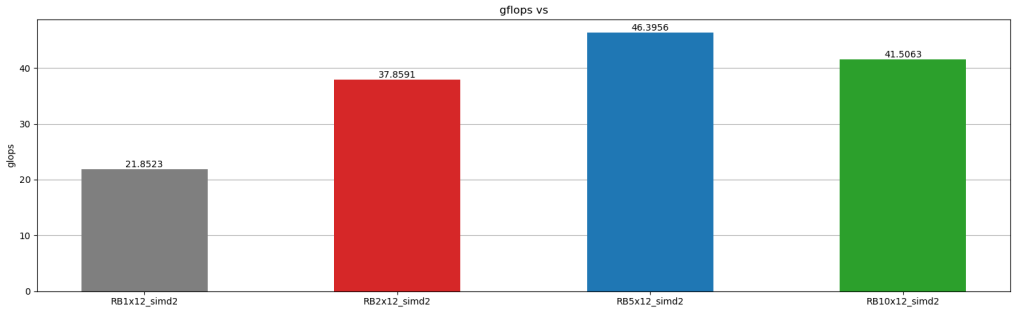

图 20: CONV6-simd2-gflops

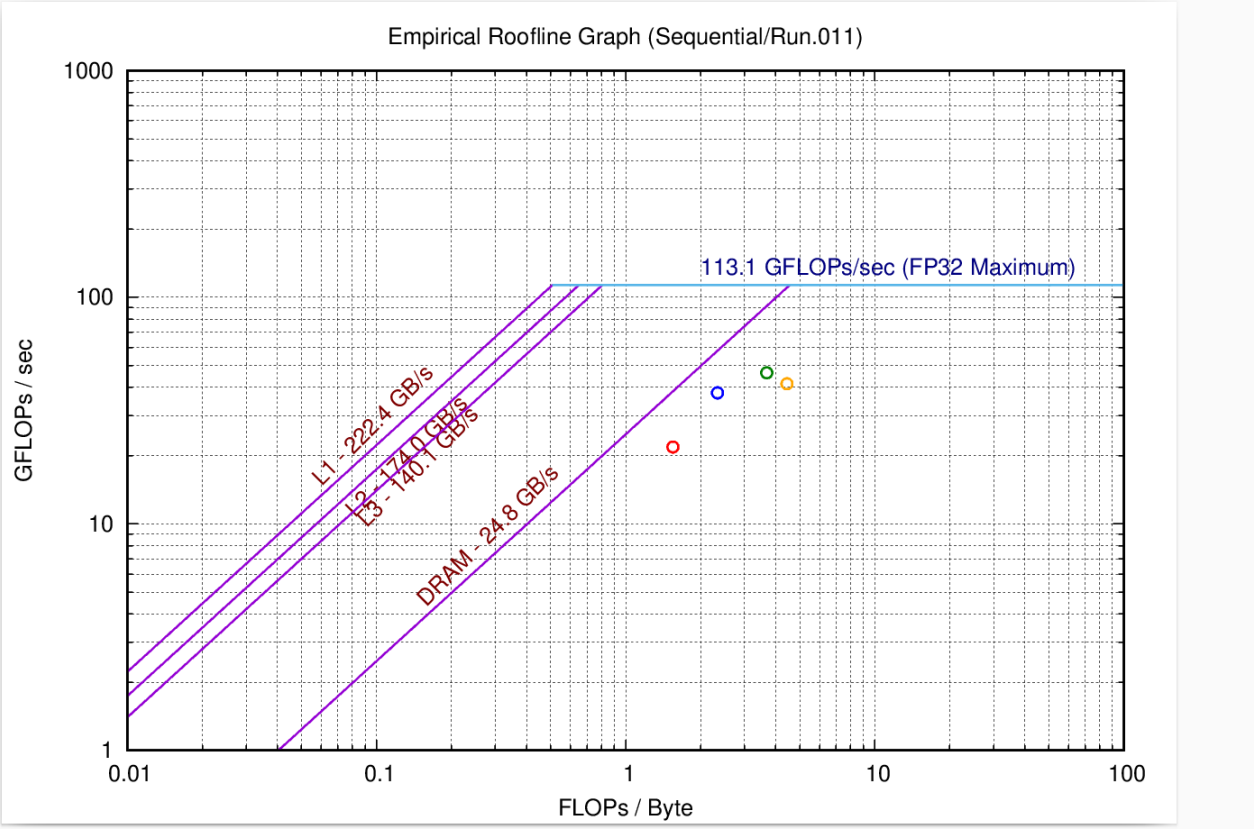

图 21: CONV6-simd2-roofline

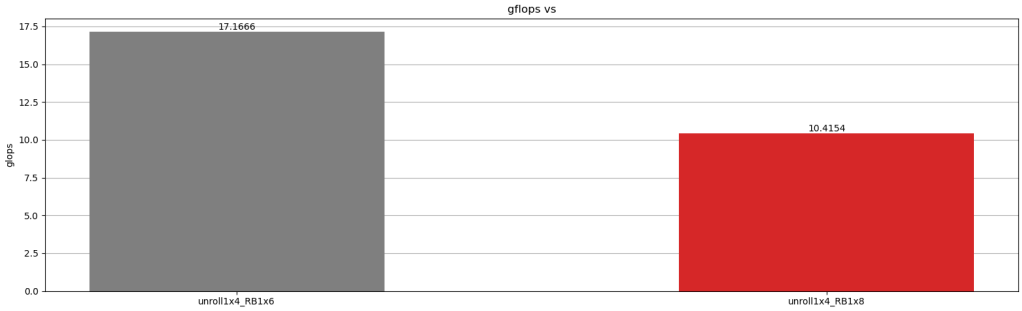

图 22: CONV7-simd1-gflops

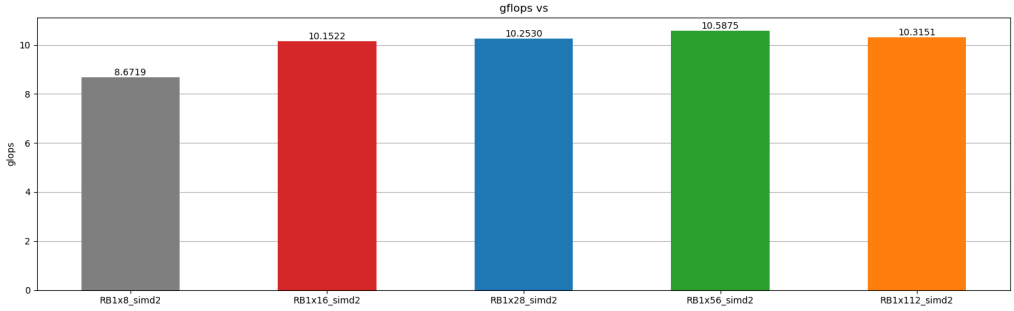

图 23: CONV7-simd2-gflops

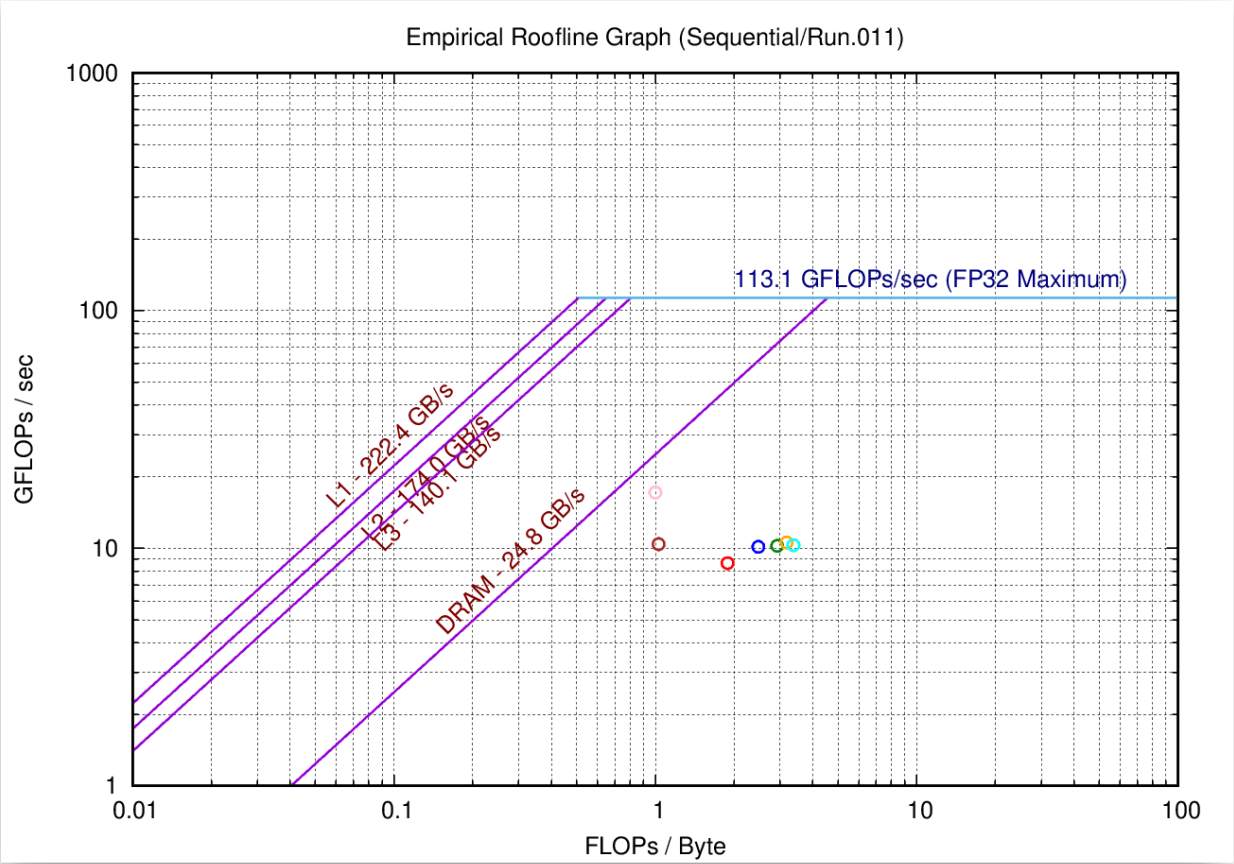

图 24: CONV7-simd1-simd2-roofline

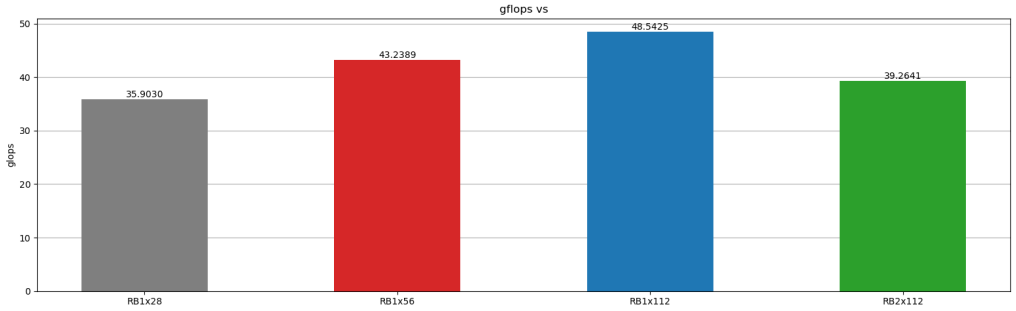

图 25: CONV8-simd2-gflops

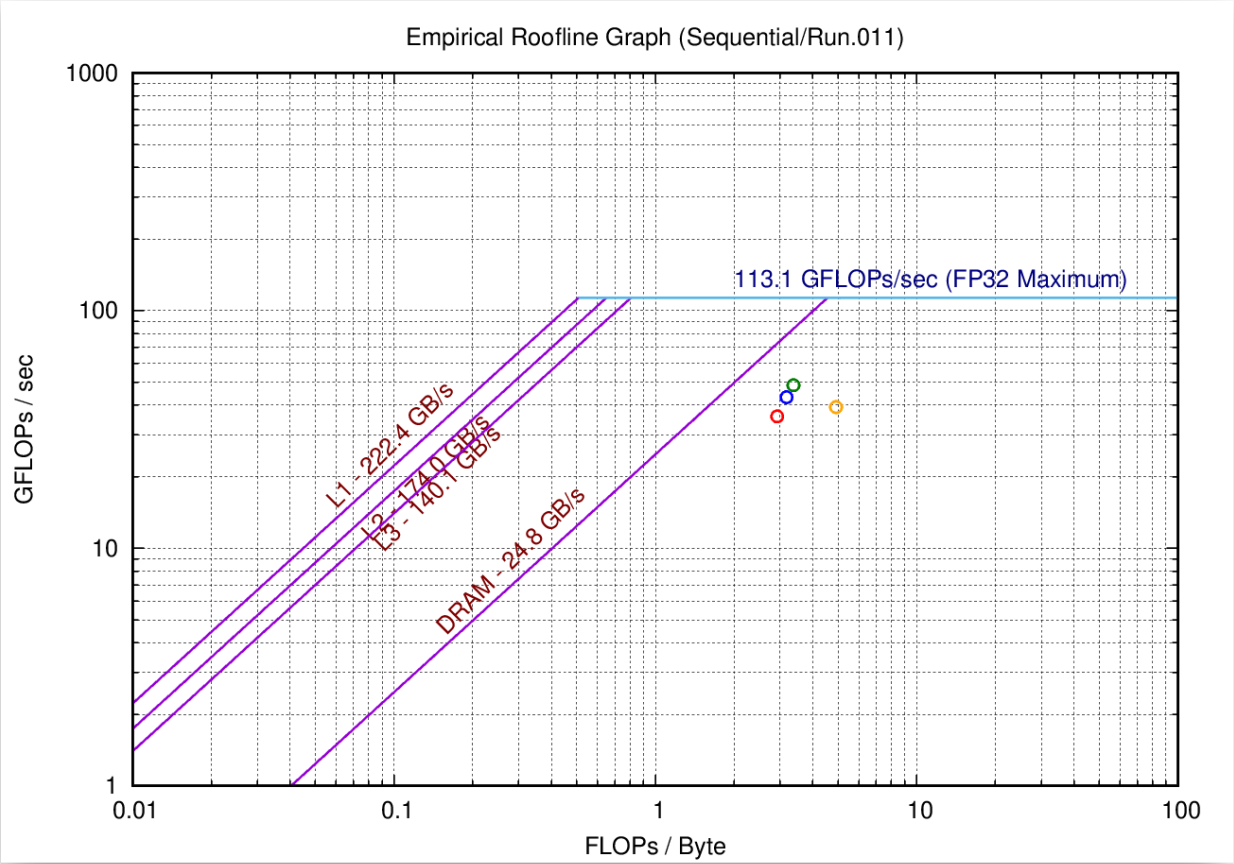

图 26: CONV8-simd2-roofline

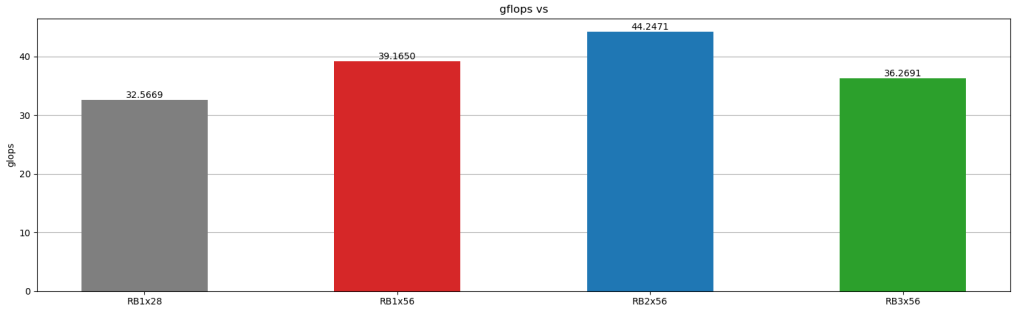

图 27: CONV9-simd2-gflops

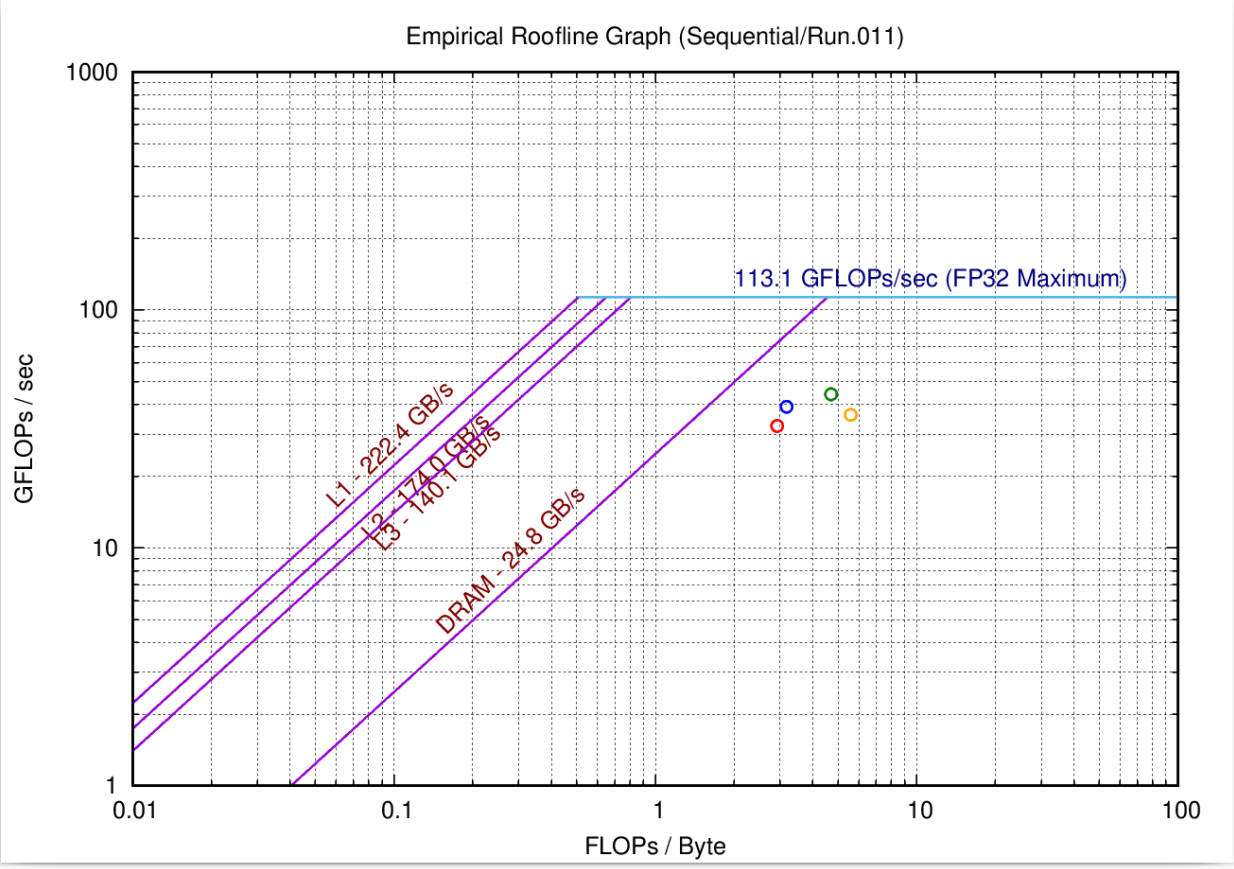

图 28: CONV9-simd2-roofline

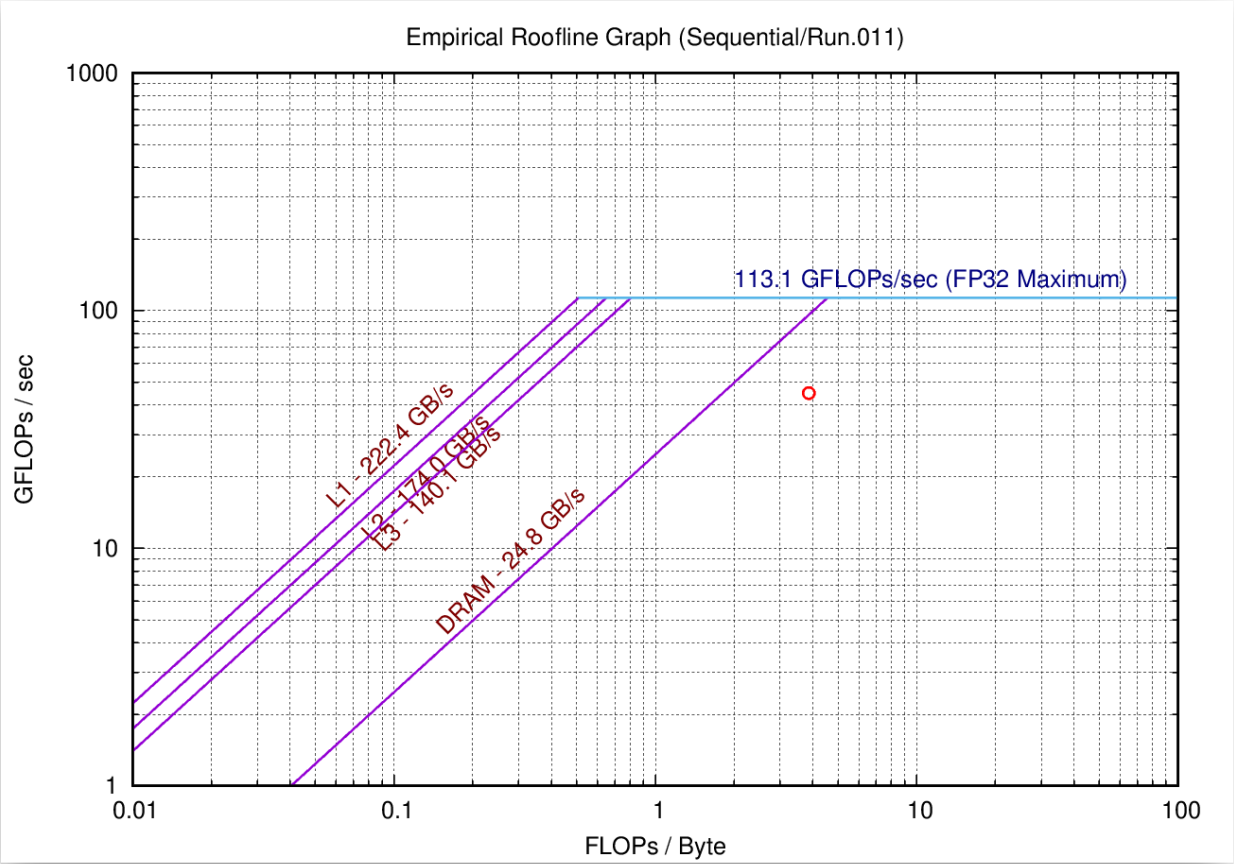

图 29: CONV10-simd2-roofline

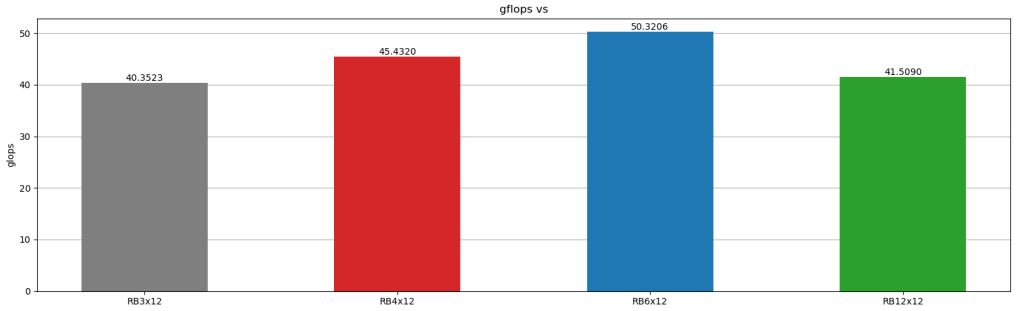

图 30: CONV11-simd2-gflops

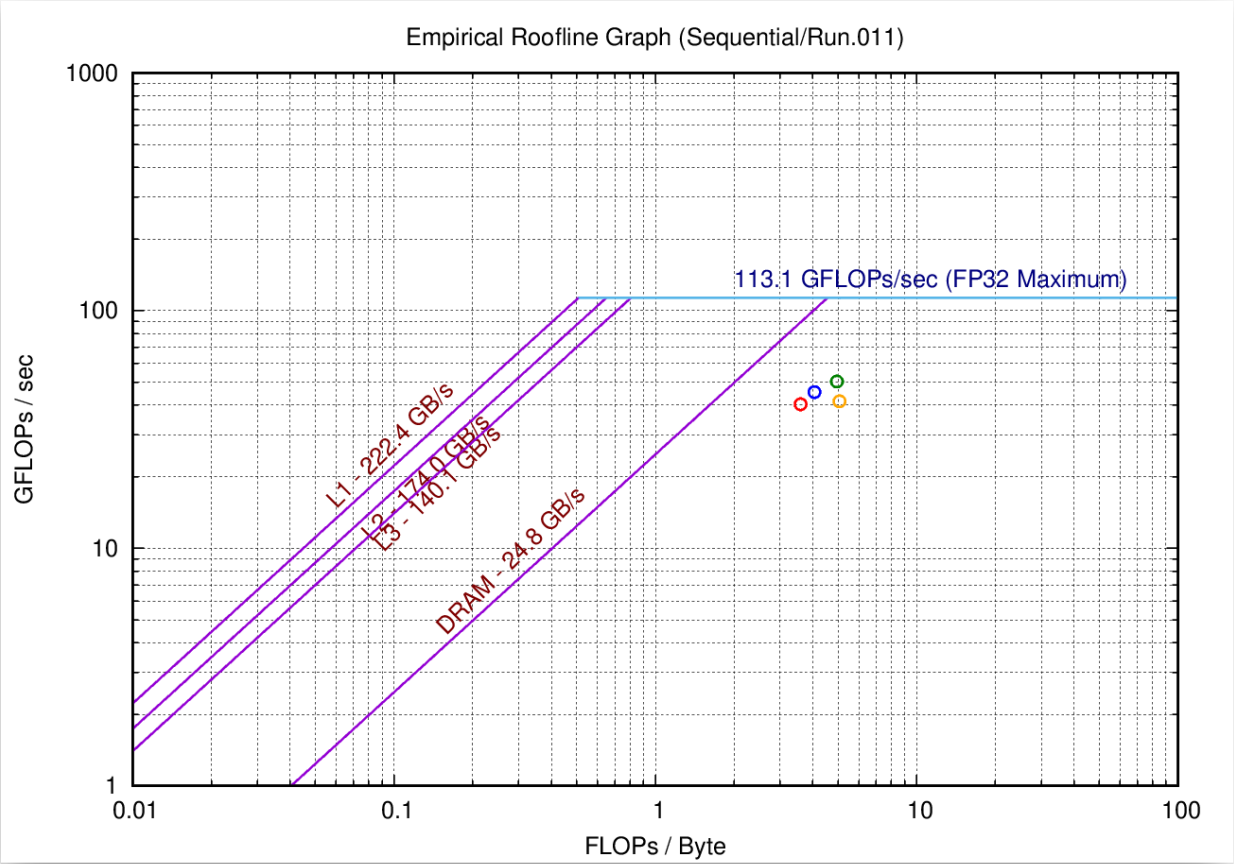

图 31: CONV7-simd2-roofline

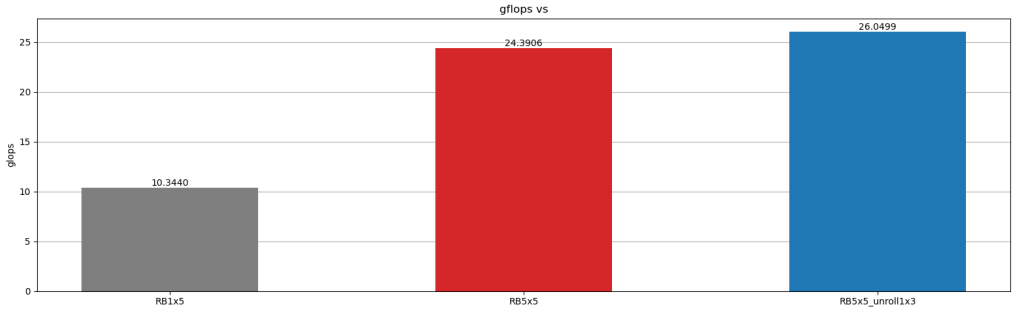

图 32: CONV12-simd2-gflops

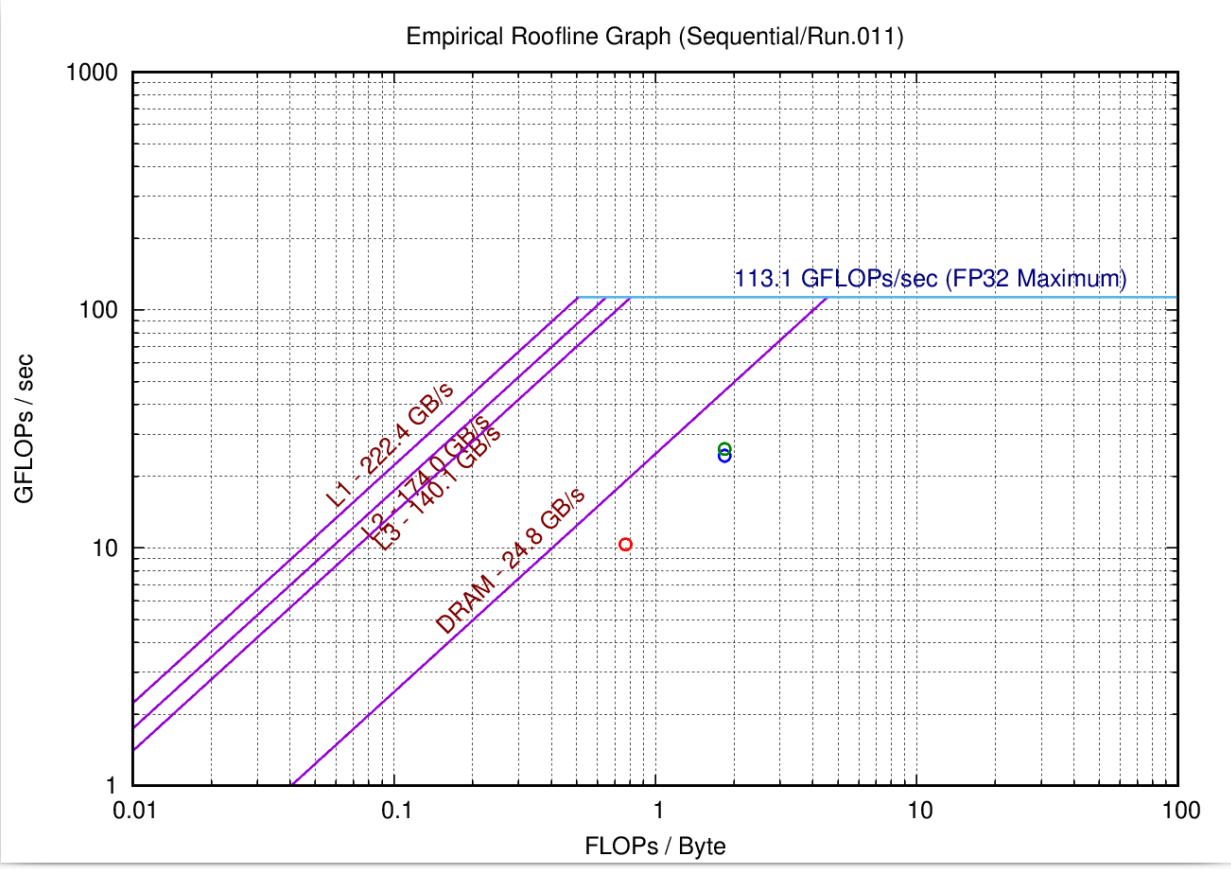

图 33: CONV12-simd2-roofline

4 实验总结

对于步长不为 1，我们使用第一种存储策略，分块顺序为，fiter-width–> output-width –> fiter->height -> output -height
对于步长为 1, 使用第二种存储策略，分块顺序, output-width->output-height -> fiter-width->fiter-height
除了 CONV7 除外，CONV7 步长为 1 使用第二种存储策略 gflop 值较小，使用第一种的速度也快不了多少，第二种存储策略对于输出张量的写入也有一定的优化，我们可以使用 simd 指令集一次性写入多个数据写入内存中。
在这些优化中最快的是 CONV5，因为 CONV5 的优化不需要使用 padding 填充，这样就不需要有填充 padding 的开销，并且所有的 simd 运算都是有效运算，其他的多少都需要使用 padding 来填充。
对于 roofline 使用可以看出算术强度在一定的范围内是越大，gflops 越高，但是超过一定的值就会降低，因为寄存器存储的数据是在栈中，有可能是因为栈空间满了的原因，多于的临时变量只能存储到堆里面，如果我们使用同一个存储策略去存储两个卷积核宽度和高度，步长都相同的卷积运算时，这个时候他们的算术强度的计算是相同的。