并行计算第一次作业报告

1900012901 王泽州

介绍

本文主要报告完成并行计算第一次作业的内容.

算法优化

本节主要描述算法优化的过程, 并给出我实现的多个`Conv2d`算法伪代码.
算法的起点是助教在教学网上给出的样例.

Naive

即样例程序.

```
for b = 0 ... BATCH_SIZE-1 do
 for c2 = 0 ... COUT-1 do
  for c1 = 0 ... CIN-1 do
   for i = 0 ... H-1 do
    for j = 0 ... W-1 do
     for ki = -(KH / 2) ... KH / 2 do
      for kj = -(KW / 2) ... KW / 2 do
        d ← (i + ki, j + kj) in boundary ?
            I[b][c1][i + ki][j + kj] : 0.0
        O[b][c2][i][j] += d * W[c2][c1][KH/2+ki][KW/2+kj]
```

通过观察程序我们不难发现两个可以改进的点:

1. 一些变量可以提前备计算, 避免重复计算. 例如`KH/2`, `KW/2`等.
2. 避免内层循环中判断属于boundary的条件语句, `zero-padding`.

No-Branch

通过改进`Naive`程序中的两个点得到的`No-Branch`算法.

1. 定义变量`h_KH`, `h_KW`避免重复计算.
2. 引入数组`offset`, 保证跨越`boundary`的变量一定为`0`.

```
h_KH ← KH / 2
h_KW ← KW / 2
for b = 0 ... BATCH_SIZE-1 do
 for c2 = 0 ... COUT-1 do
  for c1 = 0 ... CIN-1 do
   for i = 0 ... H-1 do
    for j = 0 ... W-1 do
     for ki = -h_KH ... h_KH do
      for kj = -h_KW ... h_KW / 2 do
        d ← I[b][c1][i + ki + offset][j + kj + offset]
        O[b][c2][i][j] += d * W[c2][c1][KH/2+ki][KW/2+kj]
```

通过对程序的观察, 我们发现在`(c2, c1)`确定之后, `W`数据后面的`KH*KW`的kernel就确定. 此处的优化思路是直接保存该kernel并做循环展开优化.

Loop-Unrolling

利用kernel只有`3*3`的性质, 直接循环展开.

```
h_KH ← KH / 2
h_KW ← KW / 2
for b = 0 ... BATCH_SIZE-1 do
 for c2 = 0 ... COUT-1 do
  for c1 = 0 ... CIN-1 do
   ker ← W[c2][c1]
   for i = 0 ... H-1 do
    for j = 0 ... W-1 do
     result ← 0
     result += ker[0][0] * I[b][c1][ii - 1][jj - 1]
     result += ker[0][1] * I[b][c1][ii - 1][jj]
     result += ker[0][2] * I[b][c1][ii - 1][jj + 1]
     result += ker[1][0] * I[b][c1][ii][jj - 1]
     result += ker[1][1] * I[b][c1][ii][jj]
     result += ker[1][2] * I[b][c1][ii][jj + 1]
     result += ker[2][0] * I[b][c1][ii + 1][jj - 1]
```

```
        result += ker[2][1] * I[b][c1][ii + 1][jj]
        result += ker[2][2] * I[b][c1][ii + 1][jj + 1]
        O[b][c2][i][j] += result
```

在此算法基础上还尝试了指针优化例如访问`I[b][c1][x][y]`是连续的, 因此可以单独开辟指针递增, 避免计算地址表达式. 但是这种优化并不本质故在报告中舍去.

通过比较`pytorch`的baseline之后, 我发现即使去掉内层计算的指令, 速度仍然不如`pytorch`. 因此我认为主要是没有使用`SIMD`限制了计算的性能.

Naive-SIMD

在`Loop-Unrolling`里我们发现卷积核`3*3`的性质, 在本算法中仍然利用了该性质进行`SIMD`的运算, 主要用到的指令为`_mm256_fmadd_pd`, 并行计算4个乘法. 由于`load`必须连续内存的原因实际上只做了3个有效的乘法, 最后一个系数为0.

```
for b = 0 ... BATCH_SIZE-1 do
 for c2 = 0 ... COUT-1 do
  for c1 = 0 ... CIN-1 do
   ker ← W[c2][c1]
   __m256d k1 ← _mm256_load_pd(ker, 0) // last is zero.
   __m256d k2 ← _mm256_load_pd(ker+3, 0) // last is zero.
   __m256d k3 ← _mm256_load_pd(ker+6, 0) // last is zero.
   for i = 0 ... H-1 do
    to ← O[b][c2][i]
    pt0 ← I[b][c1][i + offset - 1] + offset - 1
    pt1 ← I[b][c1][i + offset] + offset - 1
    pt2 ← I[b][c1][i + offset + 1] + offset - 1

    for j = 0 ... W-1 do
      __m256d res ← _mm256_set_pd(0, 0, 0, 0)
      __m256d a ← _mm256_load_pd(pt0)
      __m256d b ← _mm256_load_pd(pt1)
      __m256d c ← _mm256_load_pd(pt2)
      res ← _mm256_fmadd_pd(a, k1, res)
      res ← _mm256_fmadd_pd(b, k2, res)
      res ← _mm256_fmadd_pd(c, k3, res)
      res ← _mm256_hadd_pd(res, res)

      *to += ((double*)&res)[0] + ((double*)&res)[2]
```

Blocked-SIMD

本算法思路主要从https://arxiv.org/abs/1808.05567 借鉴, 通过分块的思想优化cache的访问, 并且让更大规模使用`SIMD`优化成为可能. 但是由于本人技术原因实现出来的`Conv2d`仍然达不到`pytorch`的水平.

注意这里分块的长度`VLEN`选择了32, 因此对于输入的要求为`CIN`, `COUT`参数均为32的倍数.

```
Kb ← COUT/VLEN
Cb ← CIN/VLEN

for n = 0 ... BATCH_SIZE-1 do
 for kb = 0 ... Kb-1 do
  for cb = 0 ... Cb-1 do
   for i = 0 ... H-1 do
    for j = 0 ... W-1 do
      __m256d val[VLEN]
      for k = 0 ... VLEN-1 do
        val[k] ← _mm256_set_pd(0, 0, 0, 0)
      for ki = -h_KH ... h_KH do
        for kj = -h_KW ... half_KW do
          double *pt1 = I[n][cb][ii+ki][jj+kj]
          double *pt0 = W[kb][cb][h_KH+ki][h_KW+kj][0]
          for k = 0 ... VLEN do
            for c = 0 ... VLEN do
              __m256d a ← _mm256_load_pd(pt0)
              __m256d b ← _mm256_load_pd(pt1)
              val[k] = _mm256_fmadd_pd(a, b, val[k])
              pt0 += 4
              pt1 += 4
            pt1 -= VLEN
      for k = 0 ... VLEN do
        val[k] = _mm256_hadd_pd(val[k], val[k])
        O[n][kb][i][j][k] += val[k][0] + val[k][2]
```

实验

实验主要分为两部分, 在`Setting_1`下开启`O1`优化, 这一部分主要由于观察到开启`O3`之后算法的优化效果被减弱, 因此为了更明显的观察性能的提升而设置. 在`Setting_2`中开启`O3`优化, 于`pytorch`的`baseline`进行对比.

经过多次测试后发现每次测试的方差可以忽略, 因此下表中结果均为一次测试的结果, 可以认为其有代表性.

Setting_1: With O1 optimization.

	H=W=256	H=W=128	H=W=64
Naive	18.531 s	18.539 s	19.069 s
No-Branch	14.233 s	18.125 s	14.330 s
Loop-Unrolling	4.834 s	5.067 s	4.932 s
Naive-SIMD	3.719 s	3.533 s	3.676 s
Blocked-SIMD	3.130 s	2.474 s	2.482 s

不难发现没有反常点, 经过优化之后的算法均优于之前的算法.

Setting_2: With O3 optimization.

	H=W=256	H=W=128	H=W=64
Naive	8.081 s	8.093 s	8.015 s
No-Branch	3.664 s	3.621 s	3.649 s
Loop-Unrolling	2.768 s	2.855 s	2.750 s
Naive-SIMD	3.638 s	3.491 s	3.565 s
Blocked-SIMD	1.774 s	1.631 s	1.603 s
pytorch	2.615 s	0.713 s	0.590 s

这里`Naive-SIMD`成为一个反常点, 我个人推测由于其对`O3`更加难以优化导致.

总结

通过本次作业, 我清楚的认识到了`SIMD`的强大威力, 尤其是在当我空循环都没有`pytorch`跑的快时, 这带给我了前所未有的思想. 通过阅读intel的官方文档, 我锻炼了查找文档以及代码资料能力.

最后感谢https://github.com/Triple-Z/AVX-AVX2-Example-Code让我学会了怎么使用`SIMD`指令.