Cuda算子编程与优化

郑凯文

2021 年 5 月 27 日

1 实验概述

实验要求完成深度学习框架Jittor中的MaxPool 3D算子。由于Jittor支持内嵌Cuda代码，实验的核心是Cuda编程及优化。

对于自带的`get_score.py`程序，我进行了适当修改以更好地显示结果。修改后的程序将输出25个测例的**实际吞吐率与最高档吞吐率的比值**，即加速比。在实验中，加速比超过0.95便可得到对应测例的全部分数。

2 优化手段

在以下叙述中，规定输入大小为(N,C,T,H,W)，输出大小为(N,C,t,h,w)。首先显然可以注意到，由于操作只在最后3维进行，完全可以将前两维合在一起看待，即当作(N*C,T,H,W)大小。且由于多维数组的存储特性，这样索引在存储上也是完全等价的。

2.1 version1：循环展开与局部变量缓存

首先我对于循环变量进行修改以减小运算量。如对于如下的二重循环：

```
for(int i = i0; i < i1; i++)
  for(int j = j0; j < j1; j++)
    a = max(a, in[i * stride0 + j * stride1]);
```

我将其修改为：

```
for(int i = i0 * stride0; i < i1 * stride0; i += stride0)
  for(int j = j0 * stride1; j < j1 * stride1; j += stride1)
    a = max(a, in[i + j]);
```

经过测试，这样确实提高了加速比。这是由于尽量减少了乘法次数，而转化为等价的加法。

对于循环展开，我并没有盲目地使用`pragma unroll N`，而是对于不同的kernel size，采取不同的策略并使用局部变量进行缓存。对于kernel size=2，每个Thread只负责2x2x2大小的区域，可以全部展开。对于kernel size=3，展开内部的两层，变成3x(9)。对于kernel size=8，展开内部的一层，变成8x8x(8)。（括号表示展开的语句）。

对于kernel size=3，代码为：

```
in0_type cache[9];
int h0 = hStart, h1 = hStart + 1, h2 = hStart + 2, w0 = wStart, w1 = wStart + 1, w2 = wStart + 2
int h0_s = h0 * in0_stride2, h1_s = h1 * in0_stride2, h2_s = h2 * in0_stride2;
int w0_s = w0 * in0_stride3, w1_s = w1 * in0_stride3, w2_s = w2 * in0_stride3;
for (int t = tStart * in0_stride1; t < tEnd * in0_stride1; t += in0_stride1)
{
  cache[0] = (h0 < hEnd && w0 < wEnd) ? in0_p[t + h0_s + w0_s] : init_maximum(out0_type);
  cache[1] = (h0 < hEnd && w1 < wEnd) ? in0_p[t + h0_s + w1_s] : init_maximum(out0_type);
  cache[2] = (h0 < hEnd && w2 < wEnd) ? in0_p[t + h0_s + w2_s] : init_maximum(out0_type);
  cache[3] = (h1 < hEnd && w0 < wEnd) ? in0_p[t + h1_s + w0_s] : init_maximum(out0_type);
  cache[4] = (h1 < hEnd && w1 < wEnd) ? in0_p[t + h1_s + w1_s] : init_maximum(out0_type);
  cache[5] = (h1 < hEnd && w2 < wEnd) ? in0_p[t + h1_s + w2_s] : init_maximum(out0_type);
  cache[6] = (h2 < hEnd && w0 < wEnd) ? in0_p[t + h2_s + w0_s] : init_maximum(out0_type);
  cache[7] = (h2 < hEnd && w1 < wEnd) ? in0_p[t + h2_s + w1_s] : init_maximum(out0_type);
  cache[8] = (h2 < hEnd && w2 < wEnd) ? in0_p[t + h2_s + w2_s] : init_maximum(out0_type);
  loop9(max_v,cache,out0_type);
}}
```

其中`loop9`是一个宏，可以将`max_v`与`cache`数组逐个取较大值。上述代码其实还隐藏着一些优化：

- 预先进行乘法运算，如计算`h0_s`等。这样，对数组进行索引时，只需进行加法。
- 使用局部变量作缓存。局部变量在较少时，应该可以尽量被保存在寄存器中（我并没有严格地分析寄存器的个数等，只是保持`cache`数组大小为10左右）。并且，这种缓存可以提高并行性，如对`cache`数组的赋值、访存都是在不同的位置，减少了数据冲突。虽然访存的延迟仍然存在，但对于多发射等动态调度的处理器，仍然能提高并行性（我这里只是拿CPU作例子，GPU可能有类似的机制）。
- 使用三目运算符`:?`。相对于`if`，这样应该能促进编译器生成条件转移指令，减少分支冲突的影响。

这个版本的代码位于`pool_old.py`中第2块被注释掉的代码。

2.2 version2: 分支去除

再仔细地观察测例可以发现，padding全部为0。这样，针对测例，边界的判断都可以去除了。如padding不为0时，需要如下的代码

```
int tEnd = min(tStart + {self.kernel_size}, in0_shape1);
int hEnd = min(hStart + {self.kernel_size}, in0_shape2);
int wEnd = min(wStart + {self.kernel_size}, in0_shape3);
tStart = max(tStart, 0);
hStart = max(hStart, 0);
wStart = max(wStart, 0);
```

padding为0时，这些min、max操作都不需要了，每个Thread处理的一定是kernel size^{3}大小的数据块，且不会越界。同样地，version1中哪些?:也都不需要了。

这个版本代码位于pool_old.py中未被注释的代码。

此时的加速比为：

```
1: 3.3674736969160515
2: 1.4004439290212363
3: 4.208999629179427
4: 3.374519581996633
5: 3.3556515464290526
6: 4.792279707945015
7: 1.4088780834677055
8: 3.6933609760747306
9: 4.640600613154961
10: 4.509306236562192
11: 2.993231180658451
12: 1.404464926113545
13: 3.0982426896122646
14: 3.4063106064059987
15: 3.232415399069271
16: 0.994799657236751
17: 1.3283239770929094
18: 1.292477769766883
19: 1.2825535802657124
20: 1.4502720978602566
21: 2.53728519959295
22: 6.299548250672818
23: 4.84836081727198
24: 2.02970726884154
25: 2.054612405654507
```

2.3 version3: 失败的尝试：使用Shared Memory作缓存

我还尝试了使用Shared Memory作缓存，但效果很差，吞吐率会降低5~10倍。（代码见pool_old.py中

第1块被注释掉的代码）

使用Shared Memory的思路是，将同一个Block中所有Thread访存的区域首先复制到Shared Memory中，再访问Shared Memory进行max运算。由于Shared Memory有大小限制，一些kernel size、stride参数下是不够的，因此只对一些测例使用。

经过反思，Shared Memory效果不佳的原因可能有：

- Shared Memory适用的类型是多次访存同一区域。而对于一些测例，如kernel size=2、stride=2，完全不会进行重叠的访存，使用Shared Memory只会带来额外的开销。
- 对于一些如kernel size=8、stride=1的访存，确实会出现重叠，但复制到Shared Memory时会面临线程任务分配不均的问题：由于复制后需要使用`__syncthreads();`语句进行同步，复制的速度具有短板效应。且复制任务的分配、同步需要额外开销，可能是得不偿失的。

总之，对于这种内存重用率不高的应用场景，Shared Memory不太合适。

2.4 version4/5: 两种版本的Thread/Block/Grid分配及集成

首先相对于version3，我对kernel size=2时的循环展开方式作了改变。

原来：

```
in0_type cache[8];
int t0_s = tStart * in0_stride1, t1_s = t0_s + in0_stride1;
int h0_s = hStart * in0_stride2, h1_s = h0_s + in0_stride2;
int w0_s = wStart * in0_stride3, w1_s = w0_s + in0_stride3;
cache[0] = in0_p[t0_s + h0_s + w0_s];
cache[1] = in0_p[t0_s + h0_s + w1_s];
cache[2] = in0_p[t0_s + h1_s + w0_s];
cache[3] = in0_p[t0_s + h1_s + w1_s];
cache[4] = in0_p[t1_s + h0_s + w0_s];
cache[5] = in0_p[t1_s + h0_s + w1_s];
cache[6] = in0_p[t1_s + h1_s + w0_s];
cache[7] = in0_p[t1_s + h1_s + w1_s];
loop8(max_v,cache,out0_type);
```

现在：

```
#pragma unroll 2
for(int i = tStart; i < tStart + 2; i++){
  #pragma unroll 2
  for(int j = hStart; j < hStart + 2; j++){
    #pragma unroll 2
    for(int k = wStart; k < wStart + 2; k++){
      max_v = maximum(out0_type, max_v, in0_p[i * in0_stride1 + j * in0_stride2 + k]);
```

```
    }
  }
}
```

也就是回归了最朴素的unroll。这样做的原因是，下面比上面的效率有微小的提升。原因应该是每层循环都只有2次，过多局部变量反而成为了主要的开销。

此外，相对于version3，我还先进行了Thread分配的改进：增加了Thread的z维度，以尽量使得一个Block中有较多的线程。在version3中，Thread的x、y对应着h、w，这样，在h=w=1时，一个Block中只有1个线程。而Block中的线程是32个一组调度的，与其选择较多的Block、较少的Thread，不如将Block的个数分担到Thread上。

这样就有了version4：

```
int tx = min(1024, out0_shape4);
int ty = min(1024 / tx, out0_shape3);
int tz = min(64, 1024 / tx / ty);
dim3 block(tx, ty, tz);
int bx = CeilDiv(out0_shape4, tx);
int by = CeilDiv(out0_shape3, ty);
int bz = CeilDiv(in0_shape0 * out0_shape1 * out0_shape2, tz);
dim3 grid(bx, by, bz);
```

也就是tx、ty负责w、h，bx、by负责w、h中剩余的部分，tz、bz联合负责N*C*t。

我还试了另一种分配方式（version5）：

```
int tx = min(1024, out0_shape4);
int ty = min(1024 / tx, out0_shape3);
int tz = min(1024 / tx / ty, out0_shape2);
dim3 block(tx, ty, tz);
int bx = CeilDiv(out0_shape3, ty);
int by = CeilDiv(out0_shape2, tz);
int bz = in0_shape0 * out0_shape1;
dim3 grid(bx, by, bz);
```

也就是tx、ty、tz负责w、h、t，bx、by负责h、t中剩余的部分，bz负责N*C。

将两种方法的加速比进行比较，当下列几种情况：

- kernel size=8, stride=1
- kernel size=8, T=H=W=8
- kernel size=8, C=64, T=H=W=32

时，version4较好，否则version5较好。当kernel size较大、输出的维度t、h、w较小时，version5无法保证一个Block里有尽量多的Thread，此时version4更好。其它情况下，version5中一个Block中

不同Thread负责的是一块连续的、大的三维区域（指后三维），局部性较好，可能是表现较好的原因。

3 最终效果

将version4/5进行集成，最终的加速比如下：

```
1: 3.548753746806413
2: 2.6704076415647253
3: 4.6252922538588965
4: 3.5366507321411556
5: 3.4782040639114293
6: 5.07342927164336
7: 2.6932456658687793
8: 3.614607759031551
9: 4.895819397993311
10: 4.793439156442697
11: 3.298724296194572
12: 2.706196065886775
13: 3.384769194047605
14: 3.244456212584038
15: 2.891503898753057
16: 1.0094524916591332
17: 1.2888911913194183
18: 1.2711191983001644
19: 1.3019871331226922
20: 1.4523425108981605
21: 2.808803773446107
22: 9.064590542099193
23: 4.934951141349002
24: 2.218129668895319
25: 2.240353150462302
```