实例：手写 CUDA 算子，让 Pytorch 提速 20 倍

CV开发者都爱看的 极市平台 2023-04-29 22:01:46 发表于湖北 手机阅读

作者丨PENG Bo@知乎（已授权）

来源丨https://zhuanlan.zhihu.com/p/476297195

编辑丨极市平台

极市导读

本文通过举例说明如何给pytorch 加入有趣的新 CUDA 算子（包括前向和反向）。 >>加入极市CV技术交流群，走在计算机视觉的最前沿

本文的代码，在 win10 和 linux 均可直接编译运行：

https://github.com/BlinkDL/RWKV-CUDAgithub.com/BlinkDL/RWKV-CUDA

先看需提速的操作，在我的 RWKV 语言模型【 GitHub - BlinkDL/AI-Writer AI 写小说：https://github.com/BlinkDL/AI-Writer 】，类似 depthwise 一维卷积，伪代码：

```
w.shape = (C, T)
k.shape = (B, C, T)
out.shape = (B, C, T)
out[b][c][t] = eps + sum_u{ w[c][(T-1)-(t-u)] * k[b][c][u] } 这里 u 从 0 到 t
```

它的意义，是让 $k[u \leq t]$ 对 $k[t]$ 产生影响，具体的影响程度由 $w[t-u]$ 决定，且影响在每个通道 c 都不同。

用代码写（四重循环）：

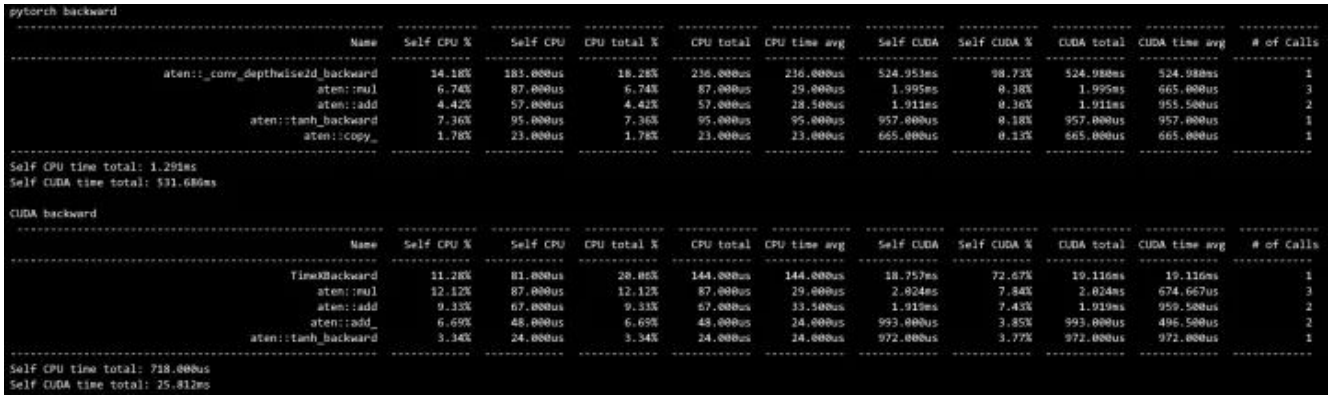
```
    out = torch.empty((B, C, T), device='cuda')
    for b in range(B):
        for c in range(C):
            for t in range(T):
                s = eps
                for u in range(0, t+1):
                    s += w[c][(T-1)-(t-u)] * k[b][c][u]
                out[b][c][t] = s
    return out
```

这个操作，用 pytorch 只需一行，但实际速度不佳，尤其是，反向梯度很慢：

```
out = eps + F.conv1d(nn.ZeroPad2d((T-1, 0, 0, 0))(k), w.unsqueeze(1), groups=C)
```

因此，我们可以用 CUDA 手写算子。实际测试，正向和反向速度可以 20x。

> **而且，这里的代码还有很多优化的空间。还望各位 CUDA 高手指导如何进一步优化，多谢多谢。**



如果你从未尝试给 pytorch 添加 CUDA 算子，可以先阅读下面这个教程：

godweiyang：熬了几个通宵，我写了份CUDA新手入门代码

下面我们看看，如何逐步优化 CUDA kernel 的写法。

1. 最简单的 CUDA Kernel 写法

最简单的写法，是直接在每个 thread 求和。这会有大量内存存取，因此效率很低。**速度为 45 毫秒。**但也比 pytorch 的 94 毫秒更快了。

Grid 和 Block：

```
dim3 gridDim(1, B * C);
dim3 blockDim(T); // 注意，我们先只在 T 分 thread，因为这样的代码简单，而且效率也够高
kernel_forward<<<gridDim, blockDim>>>(w, k, x, eps, B, C, T);
```

Kernel：

```
template <typename F>
__global__ void kernel_forward(const F *__restrict__ const w, const F *__restrict__ const
                               const F eps, const int B, const int C, const int T)
{
    const int i = blockIdx.y;
    const int t = threadIdx.x;

    F s = eps;
    const F *__restrict__ const www = w + (i % C) * T + (T - 1) - t;
    const F *__restrict__ const kk = k + i * T;
    for (int u = 0; u <= t; u++)
    {
        s += www[u] * kk[u];
    }
    x[i * T + t] = s;
}
```

2. 运用 shared memory 改善存取效率

优化 CUDA kernel 的第一步，是用 shared memory（就像矩阵乘法做 tiling）。**速度提升到 17 毫秒。**

```
template <typename F>
__global__ void kernel_forward(const F *__restrict__ const w, const F *__restrict__ const
                               const F eps, const int B, const int C, const int T)
{
    const int i = blockIdx.y;
    const int t = threadIdx.x;

    __shared__ F ww[1024]; // 这里限制了 T <= 1024 因为我实际只会用到这么多
    __shared__ F kk[1024];
    ww[t] = w[(i % C) * T + t];
    kk[t] = k[i * T + t];

    __syncthreads();
```

```
    F s = eps;
    const F *__restrict__ const www = ww + (T - 1) - t;
    for (int u = 0; u <= t; u++)
    {
        s += www[u] * kk[u];
    }
    x[i * T + t] = s;
}
```

我们在每个 CUDA thread，预先读取 w 和 k 进入 shared memory 中的 ww 和 kk，然后 __syncthreads() 等待全部读取完毕，然后可使用速度快得多的 ww 和 kk。

3. 将 thread 四合一，并运用 float4 告诉 nvcc 产生 SIMD 代码

优化 CUDA kernel 的第二步，可能是解决 bank conflict，不过，这个话题比较复杂。

我们看另一个简单易懂的步骤：将 thread 四合一，这通常是个好主意。**速度提升到 14 毫秒。**

Grid 和 Block：

```
dim3 gridDim(1, B * C);
dim3 blockDim(T >> 2); // 四合一，这里需要保证 T%4 == 0，因为我没有处理除不尽的情况
kernel_forward<<<gridDim, blockDim>>>(w, k, x, eps, B, C, T);
```

然后 CUDA 有个 float4 结构，是 4 个 float 合起来。如果用它，更容易让 nvcc 产生 SIMD 代码。

Kernel：

```
template <typename F>
__global__ void kernel_forward(const F *__restrict__ const w, const F *__restrict__ const
                               const F eps, const int B, const int C, const int T) {
    const int i = blockIdx.y;
    const int tt = threadIdx.x;
    const int t = tt << 2;

    __shared__ F wk[2048]; // 这里我们将 w 和 k 也合并了，以后会有好处
    ((float4 *)wk)[tt] = ((float4 *)w)[(i % C) * (T >> 2) + tt];
    ((float4 *)wk)[256 + tt] = ((float4 *)k)[i * (T >> 2) + tt];
    __syncthreads();

    float4 s = {eps, eps, eps, eps};
```

```
    const F *__restrict__ const ww = wk + T - t - 4;
    const F *__restrict__ const kk = wk + 1024;
    for (int u = 0; u <= t; u++) {
        F x = kk[u];
        s.x += ww[u + 3] * x;
        s.y += ww[u + 2] * x;
        s.z += ww[u + 1] * x;
        s.w += ww[u + 0] * x;
    }
    s.y += ww[t + 3] * kk[t + 1];
    s.z += ww[t + 2] * kk[t + 1];
    s.z += ww[t + 3] * kk[t + 2];
    s.w += ww[t + 1] * kk[t + 1];
    s.w += ww[t + 2] * kk[t + 2];
    s.w += ww[t + 3] * kk[t + 3];

    ((float4 *)x)[i * (T >> 2) + tt] = s;
}
```

可见，四合一还有额外的好处：循环可以重用 k[u]，进一步减少了内存读取。

4. 继续将 B 分组整合

@有了琦琦的棍子（//www.zhihu.com/people/581a2fcdf24763fbb9ec2900065986b4）指出，之前我们每个 thread 都只处理一行 T，但是，注意到 w 在 B 向是共享的，所以应该每个 thread 处理多个 w 重复的行。

我实验了代码，的确可以将正向速度提速几倍，**速度提升到 3.4 毫秒**。而对于反向，只有 grad_K 可利用重复的 w，所以效应弱一些。

```
dim3 gridDim(1, B * C / BF);
dim3 blockDim(T >> 2);
kernel_forward<<<gridDim, blockDim>>>(w, k, x, eps, B, C, T);
```

正向可以用 BF = 8，即，每个 thread 处理 8 个 B。反向似乎只适合 thread 处理 2 个 B。

```
// require T <= Tmax, T % 4 == 0, B % BF == 0, B % BB === 0 (Tmax and BF and BB are passed

#define F4(A, B) ((float4 *)(A))[(B) >> 2]
```

```
template <typename F>
__global__ void kernel_forward(const F *__restrict__ const __w, const F *__restrict__ cons
                               const F eps, const int B, const int C, const int T) {
    const int i = blockIdx.y;
    const int ij = (B * C) / BF;
    const int t = threadIdx.x << 2;

    __shared__ F ww[Tmax];
    __shared__ F kk[Tmax * BF];
    F4(ww, t) = F4(__w, t + T * (i % C));

    #pragma unroll
    for (int j = 0; j < BF; j++) {
        F4(kk, t + Tmax * j) = F4(__k, t + T * (i + ij * j));
    }
    __syncthreads();

    float4 s[BF];
    #pragma unroll
    for (int j = 0; j < BF; j++) {
        s[j] = {eps, eps, eps, eps};
    }
    const F *__restrict__ const w = ww + T - t - 4;
    for (int u = 0; u <= t; u++) {
        #pragma unroll
        for (int j = 0; j < BF; j++) {
            const F x = kk[u + Tmax * j];
            s[j].x += w[u + 3] * x;
            s[j].y += w[u + 2] * x;
            s[j].z += w[u + 1] * x;
            s[j].w += w[u + 0] * x;
        }
    }
    #pragma unroll
    for (int j = 0; j < BF; j++) {
        const F *__restrict__ const k = kk + Tmax * j;
        s[j].y += w[t + 3] * k[t + 1];
        s[j].z += w[t + 2] * k[t + 1];
        s[j].z += w[t + 3] * k[t + 2];
        s[j].w += w[t + 1] * k[t + 1];
        s[j].w += w[t + 2] * k[t + 2];
        s[j].w += w[t + 3] * k[t + 3];
        F4(x, t + T * (i + ij * j)) = s[j];
```

```
    }
}
```

5. 对齐每个 thread 的任务长度

@有了琦琦的棍子同时指出，目前每个 thread 的任务长度不同（因为 t 不同），因此会降低效率（快的 thread 会等慢的 thread）。我预计这个改动可以让速度再提升一倍，稍后加入。

6. 进一步优化

下面怎么进一步优化？还请各位 CUDA 高手指导。可以先看看 B=32，C=768，T=768 的情况，多谢多谢。

本文的代码，在 win10 和 linux 均可直接编译运行：

https://github.com/BlinkDL/RWKV-CUDAgithub.com/BlinkDL/RWKV-CUDA

极市干货

极视角动态：推进智能矿山建设，极视角「皮带传输系列算法」保障皮带安全稳定运行！

CVPR2023：CVPR 2023 | 21 篇数据集工作汇总（附打包下载链接）

数据集： 垃圾分类、水下垃圾/口罩垃圾/烟头垃圾检测等相关开源数据集汇总 | 异常检测开源数据集汇总 | 语义分割方向开源数据集资源汇总

极市原创作者激励计划

极市平台深耕CV开发者领域近6年，拥有一大批优质CV开发者受众，覆盖微信、知乎、B站、微博等多个渠道。通过极市平台，您的文章的观点和看法能分享至更多CV开发者，既能体现文章的价值，又能让文章在视觉圈内得到更大程度上的推广，并且极市还将给予优质的作者可观的稿酬！

我们欢迎领域内的各位来进行投稿或者是宣传自己/团队的工作，让知识成为最为流通的干货！

投稿须知：

1.作者保证投稿作品为自己的原创作品。

2.极市平台尊重原作者署名权，并支付相应稿费。文章发布后，版权仍属于原作者。

3.原作者可以将文章发在其他平台的个人账号，但需要在文章顶部标明首发于极市平台

投稿方式：

点击阅读原文进入CV社区

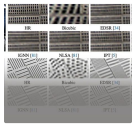
收获更多技术干货

阅读原文

喜欢此内容的人还喜欢

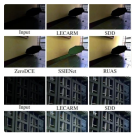
ICCV 2023｜南开程明明团队提出适用于SR任务的新颖注意力机制（已开源）

极市平台



ICCV23｜将隐式神经表征用于低光增强，北大张健团队提出NeRCo

极市平台



实践教程｜从零开始用pytorch搭建Transformer模型

极市平台