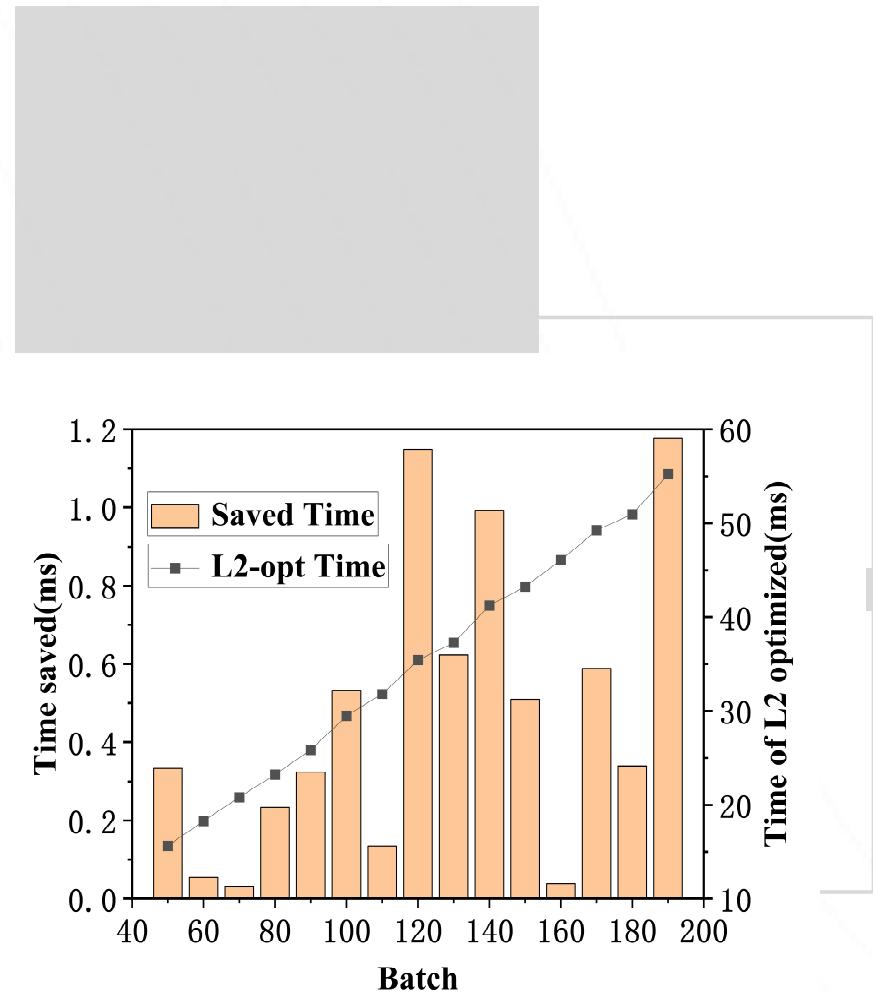


A Framework for Fine-Grained Synchronization of Dependent GPU Kernels

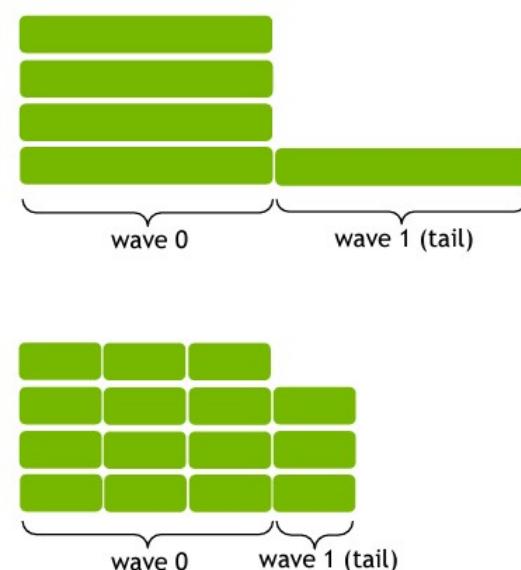
Huang Ziyu

2023/12/26



Abstract

机器学习(ML)模型包含高度并行的计算,如矩阵乘法、卷积、Dropout等。这些计算通常在图形处理单元(gpu)上执行,方法是将计算划分为独立的处理块(称为块)。由于tile的数量通常高于GPU的执行单元,因此tile以波的形式在所有执行单元上执行。然而,在最后一波中执行的贴图可能会对执行单元的利用不足,因为贴图并不总是执行单元的倍数。这种利用率不足可以通过在GPU上并发执行多个独立内核或者减小内核的尺寸来减少,但目前对于依赖内核是不可能的。在本文中,我们提出了cuSync,一个为依赖内核编写自定义细粒度同步策略以提高GPU利用率的框架。cuSync同步tile而不是内核,这允许执行多个依赖内核的tile。使用cuSync,我们在几行代码中表达了几个同步策略,并将GPT-3和ResNet-38的推理时间分别减少了 1.19倍和 1.16倍。



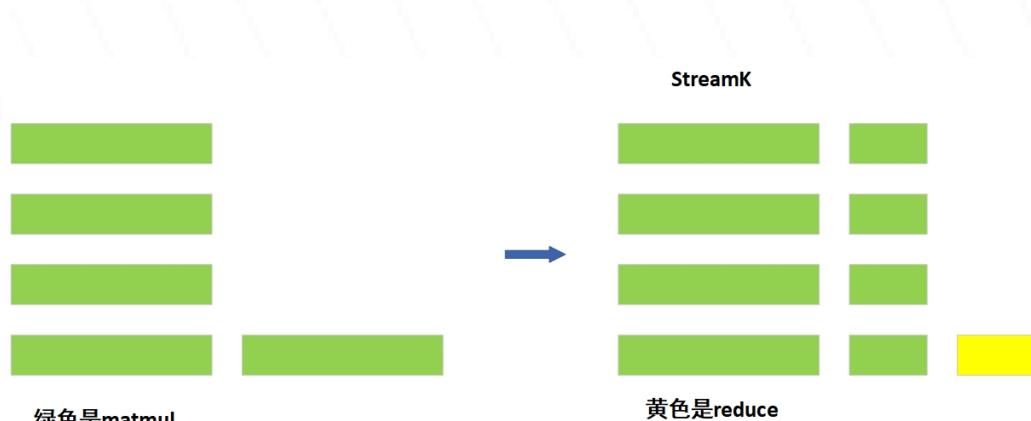
- **2 waves of threadblocks**

- Tail is running at 25% of possible
- Tail is 50% of time
 - Could be improved if the tail work could be better balanced across SMs

传统方法

- **2 waves of threadblocks**

- Tail is running at 75% of possible
- Tail is 25% of time
 - Tail work is spread across more threadblocks, better balanced across SMs
- Estimated speedup: 1.5x (time reduced by 33%)

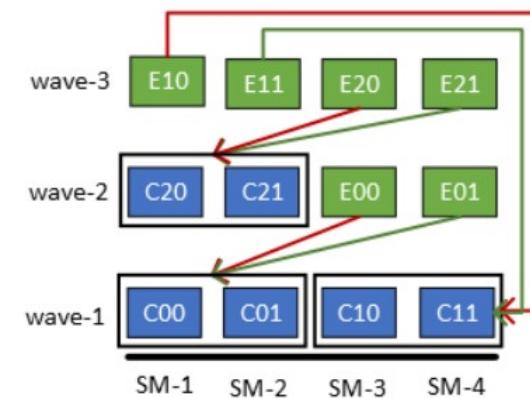
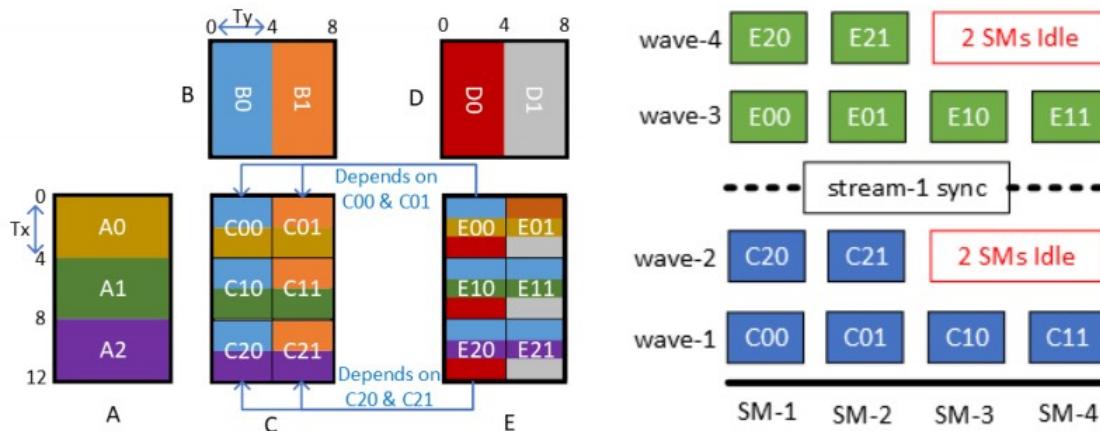


绿色是matmul

黄色是reduce

Impulse

给定以下案例。 $A \times B \rightarrow C$, $C \times D \rightarrow E$ 。假定尺寸为： $A = [12 \times 8]$ ， $B = [8 \times 8]$ ， $C = [12 \times 8]$ ， $D = [8 \times 8]$ ， $E = [12 \times 8]$ 。假设一波能够执行4个block的计算，一个block计算 4×4 的尺寸，一个SM执行一个block。如果按照中间图例的计算方式，分别进行计算第一个GEMM和第二个GEMM，那么每次都需要6个block，则都会空出2个SM，硬件利用率为75%。如果我们能够控制计算的顺序，使得优先计算 C_{00}, C_{01} ，那么与之依赖的 E_{00} 和 E_{01} 就可以在wave2中被执行，从而实现100%的硬件利用率。



Detail

A. 调用依赖内核

精细同步的第一步是消除内核间的流同步。cuSync通过在不同的CUDA流上调用内核来实现这一点。cuSync为每个内核关联一个CuStage对象。右图中的第20和21行为两个GeMM内核创建了一个生产者 (producer) prod和一个消费者 (consumer) cons阶段。这些内核在与各自阶段相关的不同流上被调用。在此之前，代码在第23行声明了两个阶段之间的依赖关系，指定消费者的输入A依赖于生产者的输出。

```
1 //CUDA Kernel to compute C = A * B
2 global void gemm(f16* A, f16* B, f16* C,
3                   int K, CuStage stage) {
4     stage.start();
5     row, col = stage.tile();
6     for (tk = 0; tk < K; tk += TileK) {
7         stage.wait(A, row, tk);
8         LoadTileToShMem(Ash, A, row, tk);
9         stage.wait(B, col, tk);
10        LoadTileToShMem(Bsh, B, col, tk);
11        MultiplyAccumulate(C, Ash, Bsh,
12                            row, col, tk);
13        stage.post(row, col);
14    }
15 void MLP(int BS, int H, f16* X, f16* W1,
16           f16* XW1, f16* W2, f16* XW12) {
17     grid1 = {B, 4*H/8}/tile1;
18     grid2 = {B, H}/tile2;
19     CuSync cs;
20     CuStage prod = cs.stage(grid1, tile1, RowMajor);
21     CuStage cons = cs.stage(grid2, tile2, RowMajor);
22     // declare prod to cons[XW1] dependency
23     cs.dependency<RowSync>(prod, cons, XW1);
24     // invoke the producer gemm
25     gemm<<<grid1, tb1, prod.stream()>>>
26     (X, W1, XW1, H, prod);
27     // invoke waitKernel and then consumer
28     cons.waitKernel();
29     gemm<<<grid2, tb2, cons.stream()>>>
30     (XW1, W2, XW12, 4*H/8, cons);}
```

(a) Synchronize two GeMM kernels using cuSync. The kernels are invoked on different streams. The wait kernel ensures the order of kernel invocation. The post and wait methods ensure tile dependency. Changes to the GeMM kernel are underlined.

Detail

B. 阶段处理顺序

在CUDA中，不同流的内核调度缺乏直接的顺序强制机制，可能导致消费者内核先于生产者内核执行，影响性能或造成死锁。为解决此问题，cuSync提供了强制内核调度顺序的机制。例如，在消费者阶段，通过调用等待内核 (wait-kernel) 来控制执行顺序。等待内核在消费者流中以单线程运行，通过全局内存中的信号量，使用繁忙等待的循环监控依赖内核的状态。（这里会反复访问全局内存，是缓慢的，是否可以移到共享内存里？）内核需调用stage.start()方法设置信号量。等待内核完成后，CUDA运行时才会调度相关联的内核，确保消费者内核不会在生产者内核至少一个线程块执行前启动。

```
1 //CUDA Kernel to compute C = A * B
2 global void gemm(f16* A, f16* B, f16* C,
3                   int K, CuStage stage) {
4     stage.start();
5     row, col = stage.tile();
6     for (tk = 0; tk < K; tk += TileK) {
7         stage.wait(A, row, tk);
8         LoadTileToShMem(Ash, A, row, tk);
9         stage.wait(B, col, tk);
10        LoadTileToShMem(Bsh, B, col, tk);
11        MultiplyAccumulate(C, Ash, Bsh,
12                           row, col, tk);
13        stage.post(row, col);
14    }
15 void MLP(int BS, int H, f16* X, f16* W1,
16           f16* XW1, f16* W2, f16* XW12) {
17     grid1 = {B, 4*H/8}/tile1;
18     grid2 = {B, H}/tile2;
19     CuSync cs;
20     CuStage prod = cs.stage(grid1,tile1,RowMajor)
21     CuStage cons = cs.stage(grid2,tile2,RowMajor)
22     // declare prod to cons[XW1] dependency
23     cs.dependency<RowSync>(prod, cons, XW1);
24     // invoke the producer gemm
25     gemm<<<grid1, tb1, prod.stream()>>>
26     (X, W1, XW1, H, prod);
27     // invoke waitKernel and then consumer
28     cons.waitKernel();
29     gemm<<<grid2, tb2, cons.stream()>>>
30     (XW1, W2, XW12, 4*H/8, cons);}
```

(a) Synchronize two GeMM kernels using cuSync. The kernels are invoked on different streams. The wait kernel ensures the order of kernel invocation. The post and wait methods ensure tile dependency. Changes to the GeMM kernel are underlined.

Detail

C. Tile处理顺序

CUDA运行时可以以任意顺序将线程块调度到SM (流式多处理器) 上, 这可能导致依赖内核中的不可预测等待时间。理想情况下, 我们希望按照生产者内核生成tile的顺序来调度消费者内核的线程块。cuSync按以下方式强制执行此顺序。每个内核调用stage.tile() (第5行) 来计算它接下来需要计算的tile。在内部, cuSync为每个阶段维护一个全局计数器, 用以决定tile计算的顺序。在示例中, 第20行和第21行的模板参数RowMajor确保了两个内核以行主序的方式生成瓦片, 这独立于CUDA运行时如何调度线程块。cuSync还支持其他tile顺序, 如列主序和跳跃行主序。

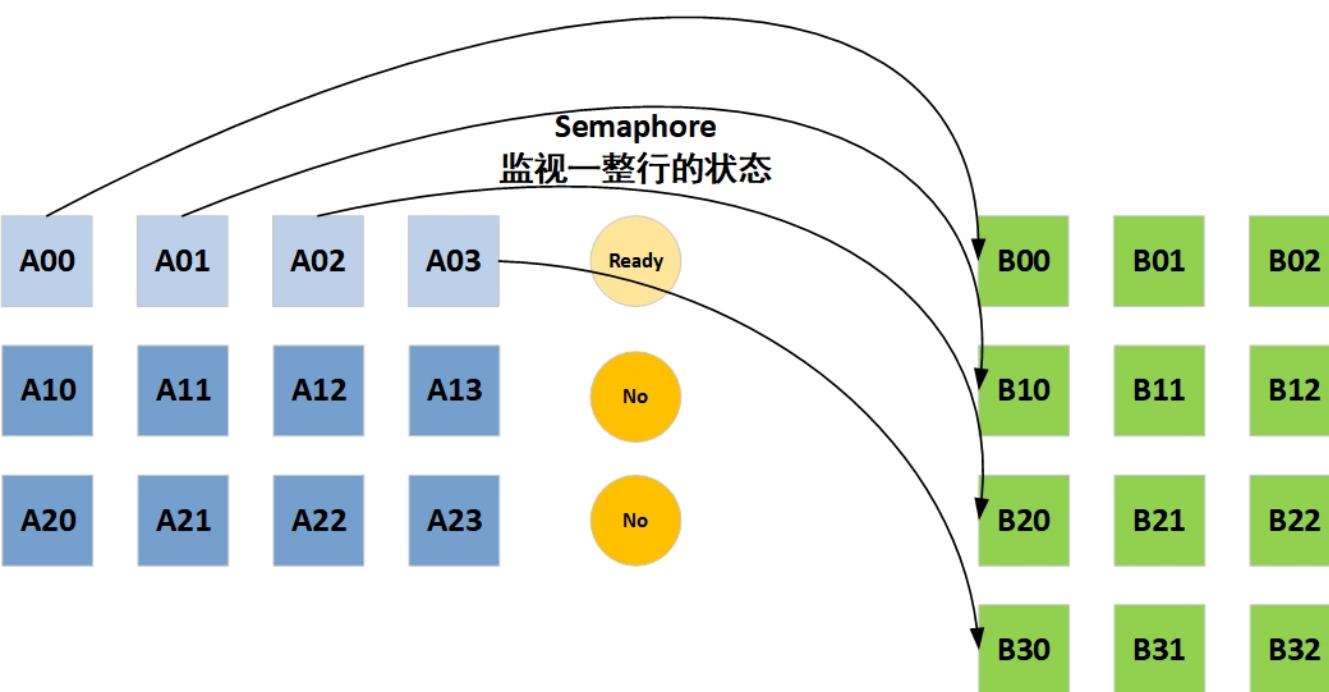
```
1 //CUDA Kernel to compute C = A * B
2 global void gemm(f16* A, f16* B, f16* C,
3                   int K, CuStage stage) {
4     stage.start();
5     row, col = stage.tile();
6     for (tk = 0; tk < K; tk += TileK) {
7         stage.wait(A, row, tk);
8         LoadTileToShMem(Ash, A, row, tk);
9         stage.wait(B, col, tk);
10        LoadTileToShMem(Bsh, B, col, tk);
11        MultiplyAccumulate(C, Ash, Bsh,
12                           row, col, tk);
13        stage.post(row, col);
14    }
15 void MLP(int BS, int H, f16* X, f16* W1,
16           f16* XW1, f16* W2, f16* XW12) {
17     grid1 = {B, 4*H/8}/tile1;
18     grid2 = {B, H}/tile2;
19     CuSync cs;
20     CuStage prod = cs.stage(grid1,tile1,RowMajor)
21     CuStage cons = cs.stage(grid2,tile2,RowMajor)
22     // declare prod to cons[XW1] dependency
23     cs.dependency<RowSync>(prod, cons, XW1);
24     // invoke the producer gemm
25     gemm<<<grid1, tb1, prod.stream()>>>
26     (X, W1, XW1, H, prod);
27     // invoke waitKernel and then consumer
28     cons.waitKernel();
29     gemm<<<grid2, tb2, cons.stream()>>>
30     (XW1, W2, XW12, 4*H/8, cons);}
```

(a) Synchronize two GeMM kernels using cuSync. The kernels are invoked on different streams. The wait kernel ensures the order of kernel invocation. The post and wait methods ensure tile dependency. Changes to the GeMM kernel are underlined.

Detail

RowSync:

很显然第二个GEMM所依赖的是C的每个整行，如果以每个block的完成为粒度设置信号量则会耽误太多的通信时间。如右图所示，对一整行设置信号量，使用post和wait函数进行计算控制。

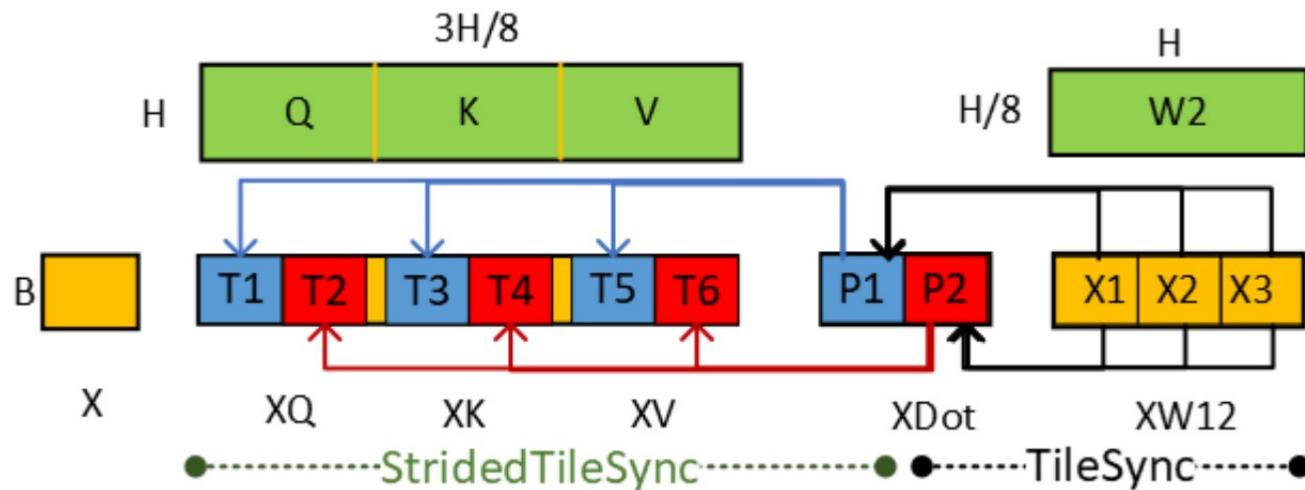


```
1 void wait_till(sem, expected) {  
2     if(threadIdx == {0,0,0})  
3         while(*sem != expected);  
4     __syncthreads();}  
5  
6 void post(sem) {  
7     __syncthreads();  
8     if(threadIdx == {0,0,0})  
9         __threadfence_system();  
10    atomicAdd(sem,1);}  
11  
12  
13 class TileSync  
14     void init(int* sem) {  
15         // Create semaphore for each tile  
16     }  
17     void wait(row, col){  
18         wait_till(  
19             &sem[row*numCols + col], 1);}  
20  
21     void post(row, col){  
22         post(&sem[row*numCols + col]);}  
23  
24 class RowSync  
25     void init(int* sem) {  
26         // Create semaphore for each row  
27     }  
28     void wait(row, col){  
29         if(col == 0)  
30             wait_till(sem[row], numCols);}  
31  
32     void post(row, col) {  
33         post(&sem[row]);}
```

Detail

StridedSync:

下图展示了GPT-3自注意力（Self-Attention）中三个内核之间Tile的依赖关系。第一个GeMM的结果 $X \cdot QKV$ ，沿着列被等分为三个矩阵： XQ 、 XK 和 XV 。点积内核执行 XQ 、 XK 和 XV 的点积，并对点积输出执行softmax，以返回矩阵 $XDot$ 。因此， $XDot$ 的第 i 列Tile依赖于 XQ 、 XK 和 XV 的第 i 列Tile。此外， $XW12$ 的每个行Tile依赖于计算相同行的 $XDot$ 的所有Tile。



Detail

Post方法：策略的post方法获取一个信号量，然后对该信号量调用post。在右图中，post方法首先执行`_syncthreads()`以确保线程块的所有线程都计算了Tile并发出了存储指令，将计算出的Tile元素写入全局内存（第7行）。然后，该方法执行一个内存屏障，以确保所有对全局内存的写入对其他内核的线程都是可见的（第9行）。最后，该方法将信号量的值增加1。

```
_device_ int X = 1, Y = 2;  
  
_device_ void writeXY()  
{  
    X = 10;  
    _threadfence();  
    Y = 20;  
}  
  
_device_ void readXY()  
{  
    int B = Y;  
    _threadfence();  
    int A = X;  
}
```

For this code, the following outcomes can be observed:

- › A equal to 1 and B equal to 2,
- › A equal to 10 and B equal to 2,
- › A equal to 10 and B equal to 20.

```
1 void wait_till(sem, expected) {  
2     if(threadIdx == {0,0,0})  
3         while(*sem != expected);  
4     __syncthreads();}  
5  
6 void post(sem) {  
7     __syncthreads();  
8     if(threadIdx == {0,0,0})  
9         __threadfence_system();  
10        atomicAdd(sem,1);}  
11  
12  
13 class TileSync  
14     void init(int* sem) {  
15         // Create semaphore for each tile  
16     }  
17     void wait(row, col){  
18         wait_till(  
19             &sem[row*numCols + col], 1);}  
20  
21     void post(row, col){  
22         post(&sem[row*numCols + col]);}  
23  
24 class RowSync  
25     void init(int* sem) {  
26         // Create semaphore for each row  
27     }  
28     void wait(row, col){  
29         if(col == 0)  
30             wait_till(sem[row], numCols);}  
31  
32     void post(row, col) {  
33         post(&sem[row]);}
```

Detail

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
                    volatile float* result)
{
    // Each block sums a subset of the input array.
    float partialSum = calculatePartialSum(array, N);

    if (threadIdx.x == 0) {
        isLastBlockDone = (value == (gridDim.x - 1));
    }

    // Synchronize to make sure that each thread
    // reads
    // the correct value of isLastBlockDone.
    __syncthreads();

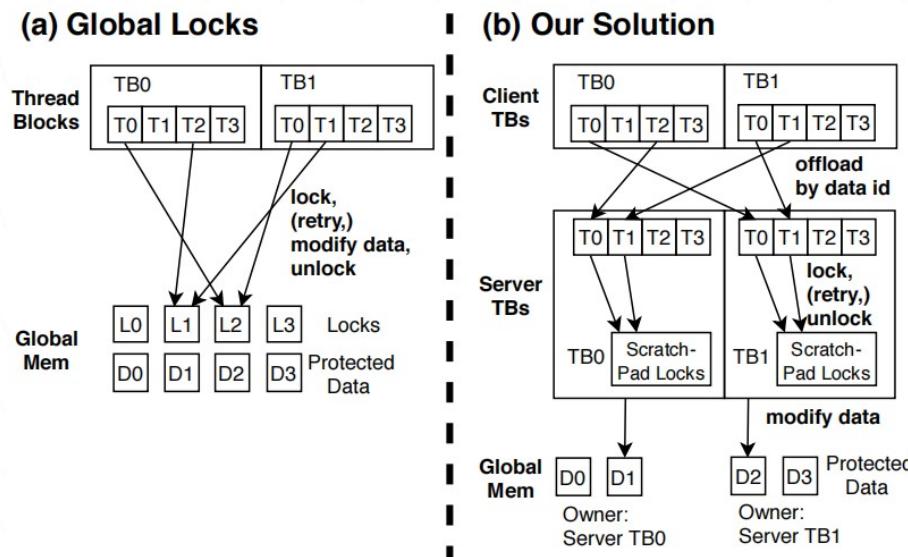
    if (isLastBlockDone) {
        // The last block sums the partial sums
        // stored in result[0 .. gridDim.x-1]
        float totalSum = calculateTotalSum(result);

        if (threadIdx.x == 0) {
            // Thread 0 of last block stores the total sum
            // to global memory and resets the count
            // variable, so that the next kernel call
            // works properly.
            result[0] = totalSum;
            count = 0;
        }
    }
}
```

Detail

Wait方法：策略的wait方法获取一个信号量，然后对该信号量调用wait_till。在右图中，wait_till函数使用线程块的第一个线程在一个while循环中读取信号量的值（第3行）。当第一个线程在信号量上等待时，线程块的所有其他线程在__syncthreads上等待（第4行）。当值变为预期值时，第一个线程到达__syncthreads，线程块的所有线程都从生产者Tile读取数据。

这里可以考虑使用一个单独的block的共享内存来处理。因为反复询问全局内存可能是低效的。



```
1 void wait_till(sem, expected) {
2     if(threadIdx == {0,0,0})
3         while(*sem != expected);
4     __syncthreads();
5
6 void post(sem) {
7     __syncthreads();
8     if(threadIdx == {0,0,0})
9         __threadfence_system();
10    atomicAdd(sem,1);
11
12
13 class TileSync
14     void init(int* sem) {
15         // Create semaphore for each tile
16     }
17     void wait(row, col){
18         wait_till(
19             &sem[row*numCols + col], 1);
20
21     void post(row, col){
22         post(&sem[row*numCols + col]);
23
24 class RowSync
25     void init(int* sem) {
26         // Create semaphore for each row
27     }
28     void wait(row, col){
29         if(col == 0)
30             wait_till(sem[row], numCols);
31
32     void post(row, col) {
33         post(&sem[row]);
34 }
```

Optimization

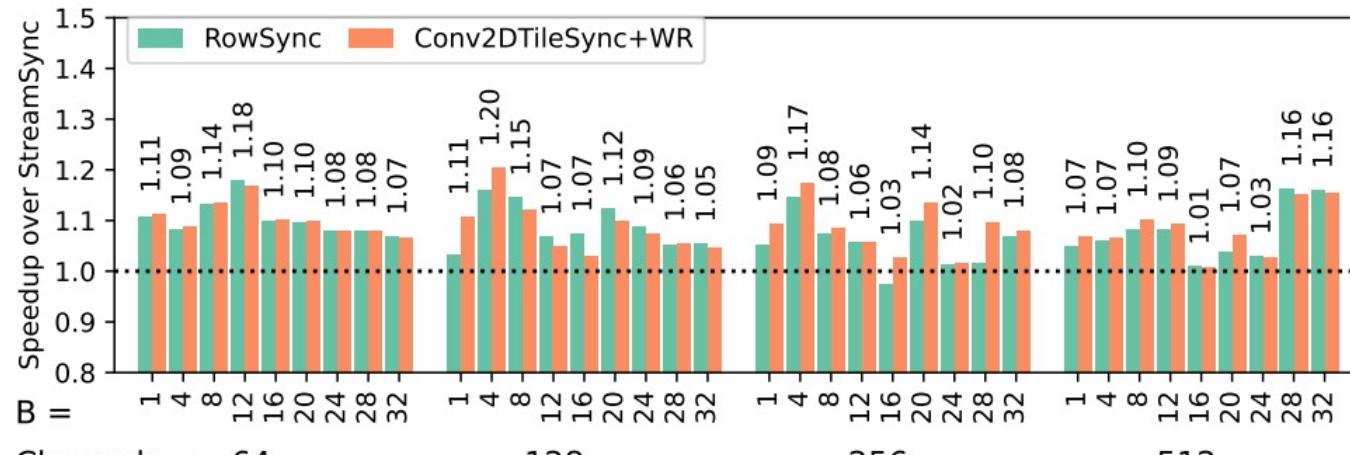
避免使用等待内核：等待内核是必需的，以确保生产者内核的所有线程块在消费者内核之前被调度到GPU上。然而，如果生产者和消费者内核的线程块可以在单个波次中被调度，我们可以避免调用等待内核。当两个内核的线程块总和小于或等于两个内核的最小占用率时，这个条件就满足了。

重新排序Tile加载和同步：基于Tile的CUDA内核的一般工作流程是加载每个输入的Tile，然后对这些Tile执行操作。我们可以重新排序一个输入的Tile等待与另一个Tile的加载，以使一个Tile的等待与另一个输入的Tile的加载重叠。

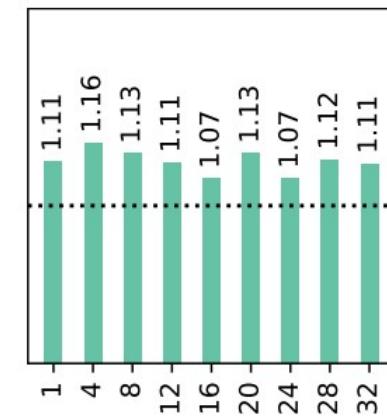
此处比如说 $A*B*C$ ，那么作为 $(A*B)*C$ ，需要等 $A*B$ 得到结果，但是这个等待的过程可以先加载C，而不是等到 $A*B$ 发出semaphore之后再加载 $A*B$ 的结果和C。

Results

||



(a) cuSync on both Conv2Ds of each layer



(b) ResNet-38 Inference

Fig. 7: Speedup of cuSync policies for both Conv2D kernels of each layer over StreamSync speedup of ResNet-38 for batch sizes from 1 to 32.

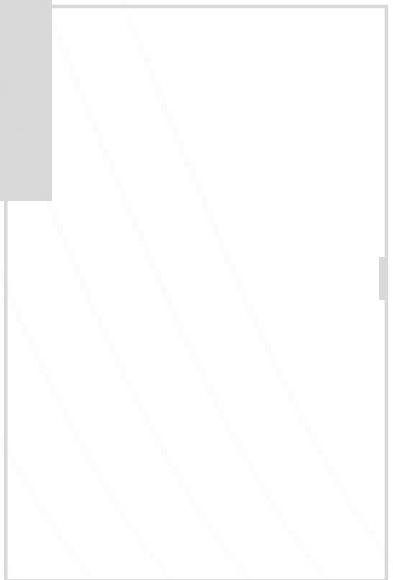
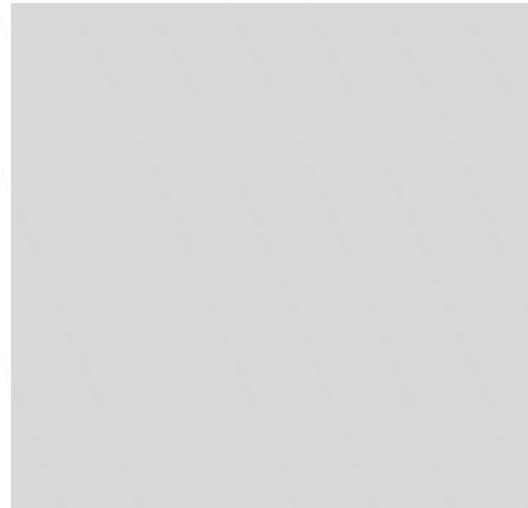
在self attention和MLP，以及conv2D和resnet38的推理场景下进行了测试，结果平均能够提升10%左右



Fig. 6: Speedup of best cuSync policy of Self-Attention and MLP

—

Thank you!



—