Matrix Multiply(A:Buffer, B:Image, C:Buffer) on Adreno GPUs #55

Open ysh329 opened this issue on Mar 1, 2021 · 6 comments

ysh329 commented on Mar 1, 2021 • edited Owner

- Matrix Multiply on Adreno GPUs – Part 1: OpenCL Optimization - Qualcomm Developer Network
  https://developer.qualcomm.com/blog/matrix-multiply-adreno-gpus-part-1-opencl-optimization
- Matrix Multiply on Adreno GPUs – Part 2: Host Code and Kernel - Qualcomm Developer Network
  https://developer.qualcomm.com/blog/matrix-multiply-adreno-gpus-part-2-host-code-and-kernel

ysh329 commented on Mar 1, 2021 • edited Owner Author

本文作者是Adreno工程师Vladislav Shimanskiy，包括两部分：OpenCL的优化方法，host端和kernel端的代码实现。本文的优化思想体现了端侧（device-side）矩阵乘法的优化思路，针对的平台是Adreno 4xx和5xx系列。

并行计算处理器如Adreno GPU是加速线性代数运算的理想硬件，但是矩阵乘法在计算密集型并行问题中却也是独特的——独立的计算线程需要有较好的数据共享：需要被乘的矩阵即A，矩阵B——给结果矩阵C，每次贡献自己的每个不同的部分与A相乘。换句话说，在Adreno GPU上优化矩阵乘法需要利用好GPU内存子系统（GPU memory subsystem），我觉得理解为"利用好GPU不同层级的内存"更合适。

GPU上矩阵乘法的难点

对于内存的复杂性以及乘法在数据共享的难题，可以将其分解为下面几个问题：

- 由于矩阵乘法反复在相似的数据上操作，随着矩阵越大，对于内存的利用越低，也就是说不得不一直替换cache里的数据；
- 原始实现的矩阵乘法使用标量计算将操作映射到每个单独的work-item，但是读写标量是相当低效的，因为GPU上的内存子系统（memory subsystem）和算术逻辑单元（ALUs）是专为向量操作而优化设计的；
- 同时加载大型矩阵A和B的元素会导致缓存冲突和内存总线争用的风险；
- 内存拷贝缓慢，因而需要一种更好的方式使得CPU和GPU的数据可以同时被访问，即避免复制。

这些问题使矩阵乘法中的操作变得更复杂：如何高效地多次读取一样的数据值并共享。

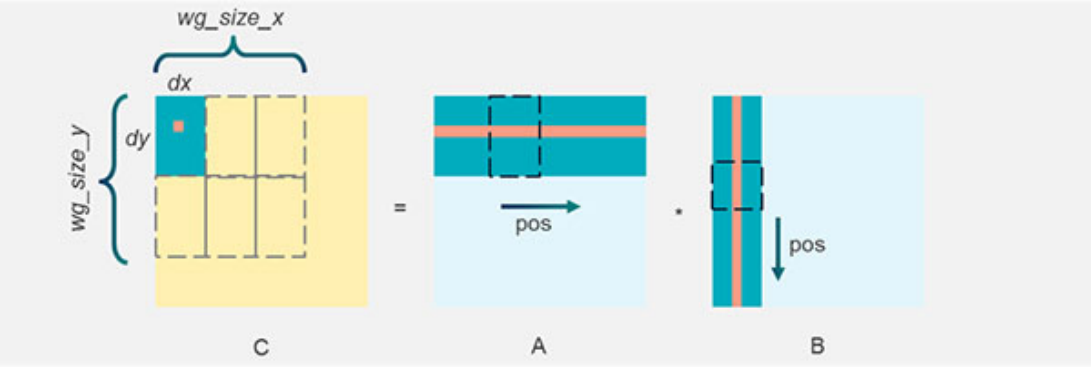
矩阵乘法的OpenCL优化方法

下面将逐一解决上述问题：

1. 分块（tiling）

第一个问题是减少对数据的重复读取，这里的数据是指高层级cache、全局内存。为了能对地址连续读取，需要将独立的读写操作组合起来。

该算法过程有两个层级的分开：小分块（micro-tiles）和大分块（macro-tiles）。下图展示了如何将部分的矩阵A与部分的矩阵B相乘，并得到矩阵C中的单个结果元素：



图：tiling示意图

- 小分块（micro-tile），即长宽为 `{dx,dy}` 的矩形区域，该区域的计算由kernel内的一个work-item完成，每个工作单元（work-item）是一个线程，每个线程位于一个SIMD子组（sub-group）中，这个子组（sub-group）也就是OpenCL的工作组（work-group）一个小部分。一个常见的小分块（micro-tile）有32个像素，即 `4x8=32`，`{dx=4,dy=8}` 。注：这里的小分块表示一个work-item待完成的工作量；
- 大分块（macro-tile），即 `{wg_size_x, wg_size_y}` 由一个或者多个小分块组成起来的整体，其工作量对应一个工作组的所有线程。即在一个工作组中，我们所有操作都是在大分块（macro-tile）内进行。

为了每次能计算C矩阵 `4x8` 维度的小分块（micro-tile），需要相应的处理矩阵A和B大小为 `4x8` 和 `4x4` 的小分块。需要从 `pos=0` 起始到k，每次计算小分块的部分结果，并将接错存入小分块中。同时在更大维度即大分块（macro-tile）上，其它的work-item会并行的计算他们相应的结果。这个过程，矩阵A的行被共享，同样，在矩阵B的一列上的数据也被共享。

在计算了大分块上的小分块的部分结果后，便在矩阵A的水平方向的 `pos` 值增加，矩阵B的竖直方向增加。通过对 `pos` 即K方向的分块累加，可以最大化cache中的数据复用，以此完成小分块的累加、卷积、点乘等操作。

这个过程会计算大分块的所有位置的计算：A从左到右，B从上到下地完成小分块计算，小分块组成了大分块。通过tile方式，缓存中存在的数据达到复用，而且在work-group即大分块的区域内的计算是同时进行的，分块平铺的方式相比——在DDR上跨越大片内存的跳转且该数据不能复用，前者内存复用带来的性能提升非常明显。

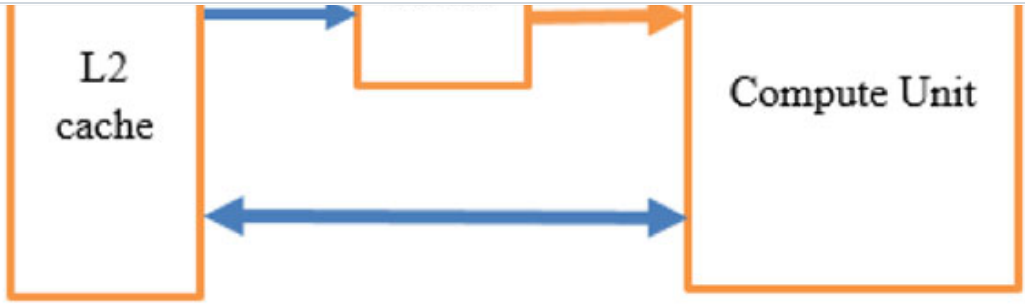
2. 向量化（Vectorization）

硬件层面的内存子系统（memory subsystem）转为向量操作进行了优化，所以在单线程的小分块（micro-tile）的处理上，向量操作效率自然比标量操作高。

例如对FP32精度为元素的矩阵上，kernel的实现以 `float4` 的矢量类型作为默认而非标量 `float` ，通过这种方式可以读取一块数据如128位的 `float4` 。这个大小也符合且填满了内存带宽上的设计，因此在实现kernel时，小分块的维度也最好是4的倍数。

3. 纹理管道（Texture Pipe）

对两个矩阵使用独立的缓存如L2 cache或纹理管道的L1 cache，可以尽量避免数据竞争（大小有限），并可以对A和B矩阵并行化进行读操作，如下图所示。而通过使用L1 cache里数据的复用，可大幅减少寄存器到L2的读操作：



图：纹理管道

在Adreno和其他许多GPU中一样，每个计算单元都与纹理管道（TP，Texture pipe）单元有独立的连接。TP有自己对一级缓存和到二级缓存的独立连接。

增加带宽的技巧是通过纹理管道（Texture pipe）加载一个矩阵，因为该过程会复用矩阵元素，因而得到了一级缓存的优势，即从TP/L1到计算单元的流量要比从L2到L1的流量高得多，达到的数据复用的目的。相反，若没有使用TP/L1，而是计算单元到L2，将会在两条总线之间产生争用。

对TP/L1的使用，从L2搬运回TP/L1的流量很小，增加了内存带宽，也平衡ALU操作并获得更高的性能：ALU计算的时间与等待数据从缓存中返回所花费的时间一样多，对二者流水线处理，两者都不会成为瓶颈。

4. 避免内存复制

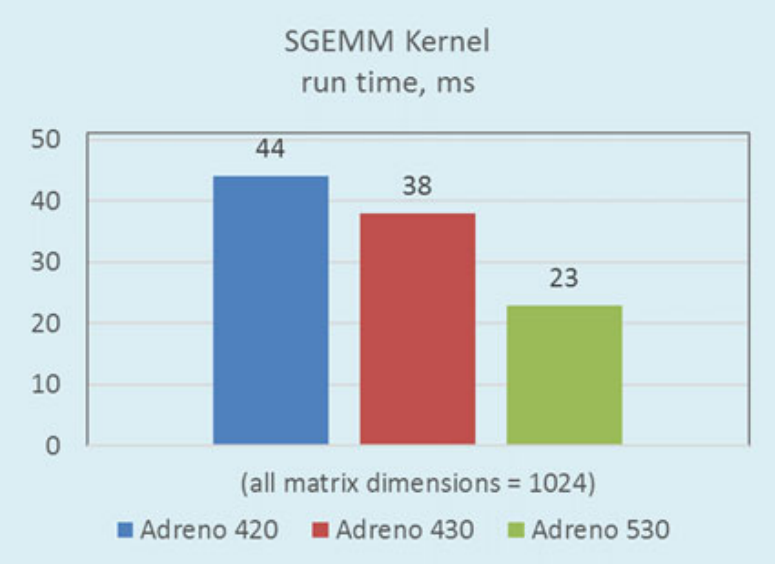
OpenCL实现由两部分代码组成：

1. GPU上计算的内核代码；
2. 运行在主机CPU上并控制内核执行的代码。

在计算GPU内核前，通常需要将输入矩阵放入CPU虚拟内存中，乘法的最终计算结果也必须能在主机内存上能获取到。即这个过程，前后需要将：输入数据从CPU拷贝到GPU上，结果数据从GPU下载回CPU上，存在来回拷贝的时间。

骁龙系列的Adreno GPU具有共享内存的优势，可以避免来回拷贝，即CPU和GPU的内存共享，但其使用并非简单的直接分配，需要通过使用OpenCL驱动提供的方法，且有如内存对齐等等限制条件。

经过上述的优化，下图展示了最终的单精度矩阵乘法的性能：



图：Adreno 4xx系列与530的性能对比

ysh329 commented on Mar 1, 2021 • edited Owner Author

主机端与Kernel代码

主机端

主机端的特点是要尽可能避免内存的拷贝，此外还有前文提到的，两个输入矩阵：

1. 矩阵B通过纹理管道（TP/L1）加载，纹理管道在OpenCL的内存对象上表现为Image格式，访问Image格式需要使用Image专有的读操作API如 `read_image` ；
2. 矩阵A则通过常规的全局内存（global memory）来访问。

结果矩阵C则是从全局内存中直接访问，对C的访问其实占比带宽不大，因为C的访问不存在像是A和B的数据复用，而是对C的单一一个元素算完后后续就不会再访问了，即写操作仅一次。

对矩阵A和C的内存分配

代码：Memory allocation for matrices loaded via L2 cache (A and C)

```
cl::Buffer * buf_ptr = new cl::Buffer(*ctx_ptr,
                                      CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR,
                                      na * ma * sizeof(T));
T * host_ptr = static_cast<T *>(queue_ptr->enqueueMapBuffer( *buf_ptr,
                                                CL_TRUE,
                                                CL_MAP_WRITE,
                                                0, na * ma * sizeof(T)));
lda = na;
```

下面的代码简单说明：

1. 代码第一行：由OpenCL驱动创建的 `buf_ptr` 可用于CPU的读写操作，即在创建 `cl::Buffer` 时声明 `CL_MEM_READ_WRITE` 和 `CL_MEM_ALLOC_HOST_PTR` 。此外， `na` 和 `ma` 分别是矩阵A的列数和行数。GPU内存不能用malloc函数创建，在CPU代码操作GPU buffer前，需要使用CL API的映射函数（mapping）映射到CPU地址空间；
2. 代码第二行：
3. 代码第三行：定义了矩阵A的每行所占元素个数，例如一个 `100x100` 维度的矩阵其 `lda` 则为100个元素。注：lda并非等于矩阵的水平方向的长度，不同场景下是不同的，比方对矩阵分块，那么只是代表这一小块上的长度。此外， `ldb` 、 `ldc` 与 `lda` 类似。

矩阵B的内存分配（images）

对矩阵B的image内存分配代码见下面，相比Buffer形式略复杂。Images相比buffer有诸多限制约束：除了读取image需要用专门的API且不能按照下标随机访问以外。

多通道的使用如RGBA、RGB，还需要在内存上进行合理的行对齐。对B使用image，每个颜色成分实际用浮点表示，从矩阵角度看，实际是把当前行的每4个元素压到RGBA的4元素对应的一个像素中，那么image的行为原本矩阵行的 `1/4` ，当然前提是需要对行做4倍数的对齐，这样后续在计算中，每次从image读取一个像素即一个float4，相当于将原本的数据以4个打包为一起到了image的像素中。

下面是创建Image2D GPU内存的一段代码实例：

代码：Memory allocation for float32 matrices loaded via texture pipe (B)

```
cl::size_t<3> region;
origin[0] = 0; origin[1] = 0; origin[2] = 0;
region[0] = na/4; region[1] = ma; region[2] = 1;
size_t row_pitch;
size_t slice_pitch;
T * host_ptr = static_cast<T *> (queue_ptr->enqueueMapImage( *img_ptr, CL_TRUE, CL_MAP_WRITE, origin, region
ldb = row_pitch / sizeof(T);
```

第1行，即第一个函数是使用 `CL_MEM_ALLOC_HOST_PTR` 标志从主机端分配内存；
第8行，对 `enqueueMapImage` 的调用，类似对矩阵A和C使用了 `enqueueMapBuffer` ，这里 `enqueueMapImage` 为提供了另一种主机指针，即在GPU内存中分配的Image对CPU可见。该操作确保CPU和GPU对Image的缓存数据一致。

在CPU端调用Kernel

该操作包含下面3步：

1. 为GPU的后续执行，取消CPU端对矩阵A和B的映射关系；
2. 执行GPU kernel；
3. 将C矩阵结果映射回CPU，确保CPU可见。

不难发现来回有对CPU和GPU的映射操作，因为这个映射对CPU和GPU不能同时存在，OpenCL官方禁止是同一时间对同一个数据在CPU和GPU做操作，可能造成问题。下面的代码，将会利用骁龙处理器在共享虚拟内存（Shared Virtual Memory）上的优势，实现刚所说的功能（部分与上面Image内存分配的代码略有重复）：

代码：Kernel run cycle and memory synchronization procedures

```
// update GPU mapped memory with changes made by CPU
queue_ptr->enqueueUnmapMemObject(*Abuf_ptr, (void *)Ahost_ptr);
queue_ptr->enqueueUnmapMemObject(*Bimg_ptr, (void *)Bhost_ptr);
queue_ptr->enqueueUnmapMemObject(*Cbuf_ptr, (void *)Chost_ptr);
// run kernel
err = queue_ptr->enqueueNDRangeKernel(*sgemm_kernel_ptr, cl::NullRange, global, local, NULL, &mem_event);
mem_event.wait();

// update buffer for CPU reads and following writes
queue_ptr->enqueueMapBuffer( *Cbuf_ptr, CL_TRUE, CL_MAP_READ | CL_MAP_WRITE, 0, m_aligned * n_aligned * size
// prepare mapped buffers for updates on CPU
queue_ptr->enqueueMapBuffer( *Abuf_ptr, CL_TRUE, CL_MAP_WRITE, 0, k_aligned * m_aligned * sizeof(float));
// prepare B image for updates on CPU
cl::size_t<3> origin;
cl::size_t<3> region;
origin[0] = 0; origin[1] = 0; origin[2] = 0;
region[0] = n_aligned/4; region[1] = k_aligned; region[2] = 1;
size_t row_pitch;
size_t slice_pitch;
queue_ptr->enqueueMapImage( *Bimg_ptr, CL_TRUE, CL_MAP_WRITE, origin, region, &row_pitch, &slice_pitch);
```

上述代码讲解如下：

- 第一部分：使用 `enqueueUnmapMemObject=` 取消映射，为了GPU后续的矩阵乘法能将CPU做的操作可见，这步骤是必须的。换句话说，这也是缓存一致性的考虑：分配了矩阵A和B，起初在CPU端填充它们，后续使它们对GPU可见，这个过程不需复制内存；
- 第二部分：现在GPU能看到映射（Map）来的矩阵数据，通过 `enqueueNDRangeKernel` 将执行菊哲内乘法操作。这里的示例代码为了简略，省去了对kernel设置参数的过程；

> 注：其中可以看到A和C的Buffer内存在创建时是考虑对齐的，即 `k_aligned * m_aligned` 以及 `m_aligned * n_aligned` ，一般是4的倍数对齐。

ysh329 commented on Mar 4, 2021 • edited Owner Author

GPU上的Kernel代码

下面列出FP32精度的sgemm代码的核心部分，这是对BLAS库中sgemm操作简化版的实现： $C = \alpha AB + \beta C$ ，为简略 $\alpha = 1$ ， $\beta = 0$ 。

代码：Example of a kernel implementing a C = A * B matrix operation

```
__kernel void sgemm_mult_only(
                            __global const float *A,
                            const int lda,
                            __global float *C,
                            const int ldc,
                            const int m,
                            const int n,
                            const int k,
                            __read_only image2d_t Bi)
{
    int gx = get_global_id(0); // [0, N / 4)
    int gy = get_global_id(1);  // [0, M / 8)

    if (((gx << 2) < n) && ((gy << 3) < m))
    {
        float4 a[8];
        float4 b[4];
        float4 c[8];

        for (int i = 0; i < 8; i++)
        {
            c[i] = 0.0f;
        }
        int A_y_off = (gy << 3) * lda; // a_y_idx

        for (int pos = 0; pos < k; pos += 4)
        {
            // 准备数据b: 1行4列
            #pragma unroll
            for (int i = 0; i < 4; i++)
            {
                b[i] = read_imagef(Bi, (int2)(gx, pos + i));
            }

            // 准备数据a: 8行4列（4列为float4）
            int A_off = A_y_off + pos;
            #pragma unroll
            for (int i = 0; i < 8; i++)
            {
                a[i] = vload4(0, A + A_off);
                A_off += lda;
            }

            #pragma unroll
            for (int i = 0; i < 8; i++)
            {
```

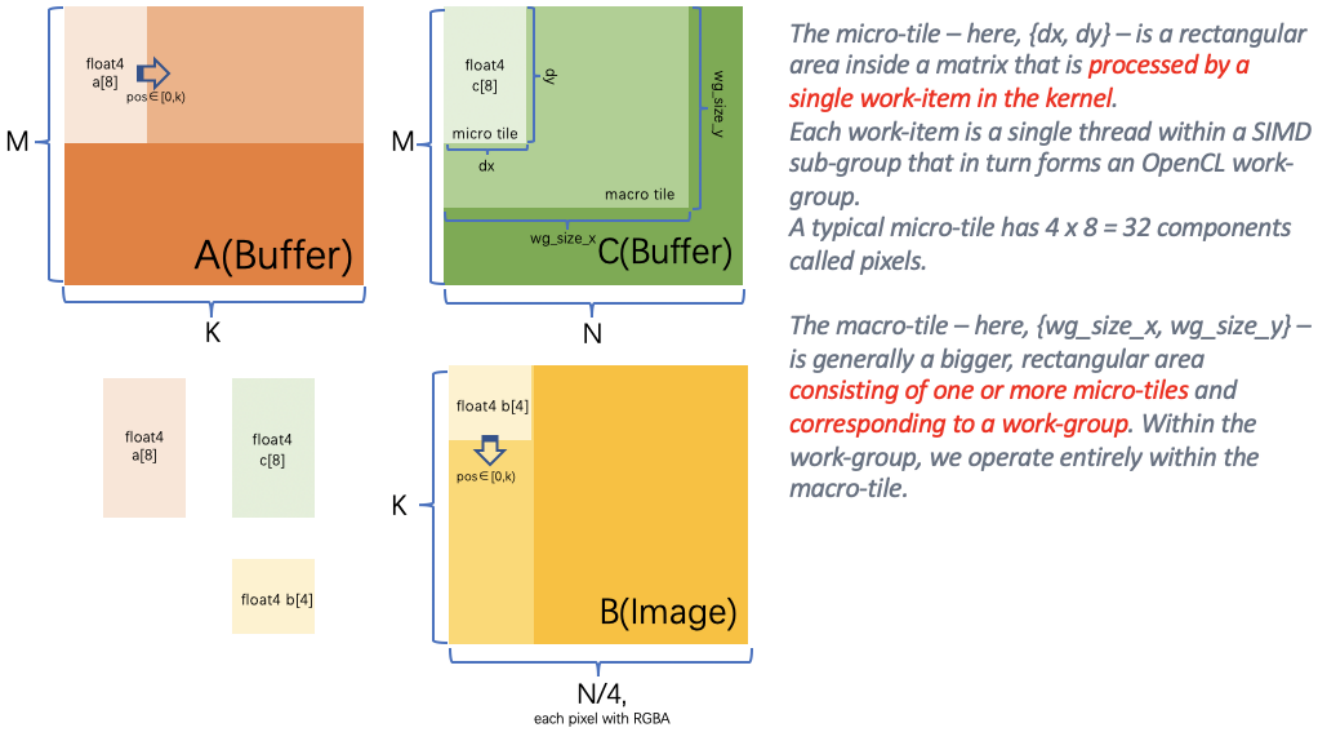
```
        }

        #pragma unroll
        for (int i = 0; i < 8; i++)
        {
            int C_offs = ((gy << 3) + i) * ldc + (gx << 2);
            vstore4(c[i], 0, C + C_offs);
        }
    }

}
```

ysh329 changed the title ~~Matrix Multiply on Adreno GPUs~~ **Matrix Multiply(A:Buffer, B:Image, C:Buffer) on Adreno GPUs** on Mar 4, 2021

ysh329 commented on Mar 4, 2021 • edited Owner Author

该kernel实现的基本思想：对buffer形式的矩阵A和image形式的矩阵B，在K方向上分别做固定大小的循环展开，`float4 a[8]` 即8行4列的A， `float4 b[4]` 即4行4列的B， `float4 c[8]` 即8行4列的C。

计算乘加的for循环每次计算1个float4的A与4个float4的B，计算乘加的for循环循环8次，即完成前面所说的对A在K方向上的8行（齐头并进向右），B在K方向的4列（齐头并进向下），得到C的4行8列的结果（即每个线程的工作量）。



图：我对该Kernel实现的理解（类似前文的tiling示意图）

下面说一下该kernel实现的要点：

- 矩阵维度必须是8行4列的倍数。这部分关于micro-tile和macro-tile的说明，涉及到workgroup的占用率等等，我一开始没看明白，但联系前文讲概念的部分，看明白了。kernel内的实现限定了单个线程操作相对矩阵C/B/A的维度： `if (((gx << 2) < n) && ((gy << 3) < m))` ，这个大小即C的 `8x4` ， A的8行，B的4列。前文说：work-group是大分块（macro tile）而单个线程在C上的计算结果大小为小分块（micro tile），为了高效性，大分块（macro tile）即work group一般为1个或多个小分块（micro tile）组成，但有个问题有时候占不满，这就会导致GPU的利用效率不高，需要尽量避免这种情况。所以在设置work group时，需要是小分块即单个线程的工作量的倍数，且

- 外层pos变量对k的循环，包含3个子循环：
- 第一个子循环：用 `read_imagef` 函数通过纹理管道（Texture pipe）TP/L1读取矩阵B的元素；
- 第二个子循环：直接从L2 cache中读取矩阵A的元素（原文： `The second sub-loop contains reads of elements of matrix A from L2 directly.` ，但能看出A是 `global mem` ，**感觉应是 `to L2 directly` ？** ）；
- 第三个子循环：用来做点乘，计算部分的C结果；
- 为提高效率，所有的load/store等ALU操作都使用float4向量；

这段代码看着很简单，但实际上是较为高度优化的——平衡了计算和数据大小。可以带上 `-cl-fast-relaxed-math` 编译选项进行编译。

ysh329 commented on Mar 4, 2021 • edited Owner Author

Workgroup大小

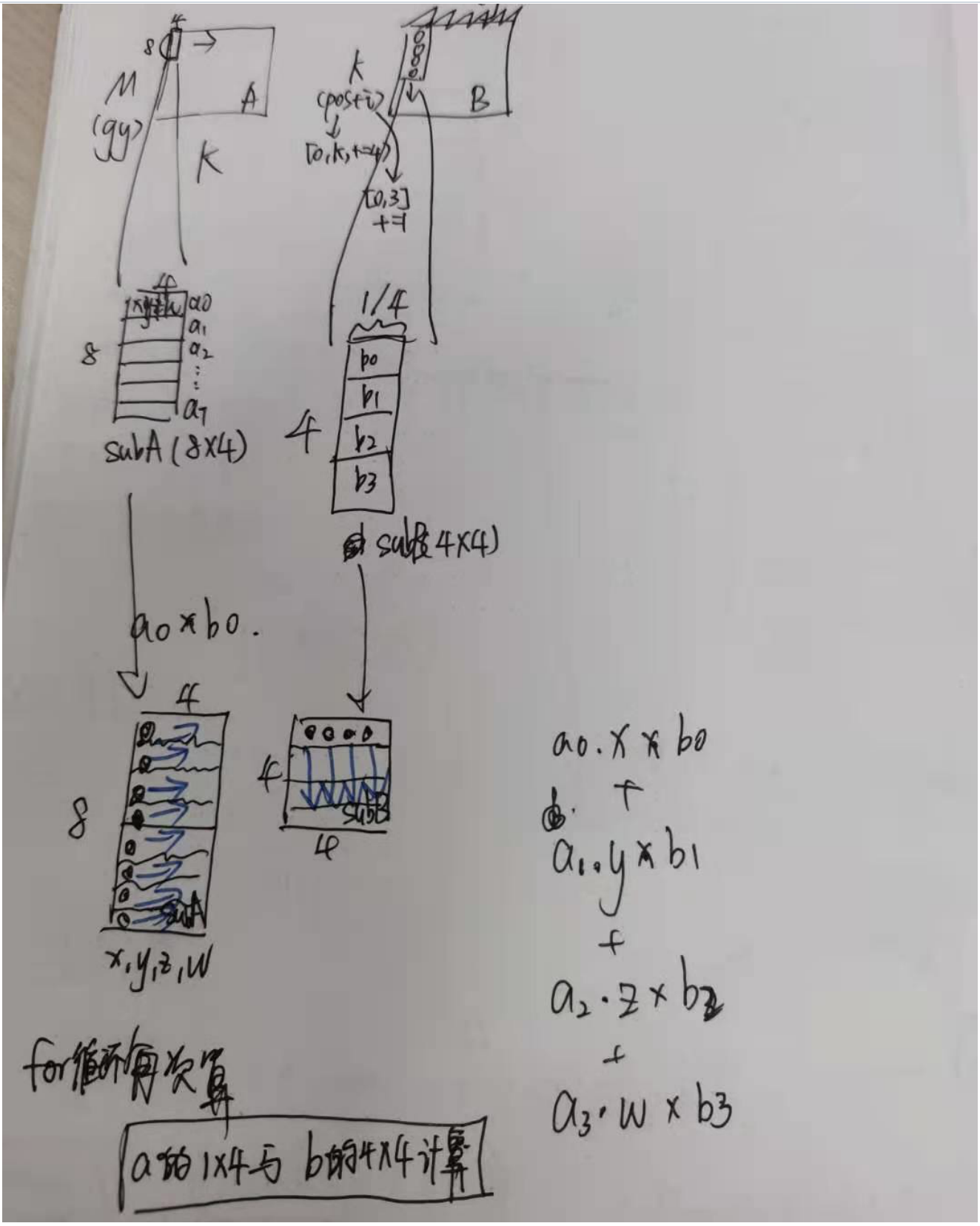
如上所述，macro tile由一组 `4x8` 大小的micro tile组成，准确的micro-tile的个数是由2-D的workgroup在水平和竖直方向的大小所决定。一般来说，使用较大的work group尺寸可避免GPU计算单元（Compute Unit）的低利用率问题。这需要设置work group尺寸，但其尺寸通过OpenCL API `getWorkGroupInfo` 可以获取到最大上限，上界是work-group里的work-item总数。所以，在上限以内都是可以调整的范围，下面是找到合适work group的参考建议：

1. 尽量减少占不满work groups的情况，以提高利用率；
2. 在运行时，对一些不同尺寸规模下的矩阵的情况，使用不同的work group进行试探；
3. 针对特殊情况实现对应的kernel：如很小的维度比方需实现gemv；
4. 当从GPU上下载数据或者相关管道流程成为性能瓶颈时，不妨使用CPU计算，比方较小的矩阵乘法操作。（我补充：或者是将多个小矩阵的操作合并）

结语

矩阵乘法是计算密集型任务，利用好上述提到的技巧，通过优化内存系统（memory subsystem），可以高效地加速如深度学习应用。

ysh329 commented on Mar 5, 2021 Owner Author



Assignees

No one assigned

Labels

Projects

None yet

Milestone

No milestone

Development

No branches or pull requests

1 participant