Adreno GPU 矩阵乘法——第2部分：主机代码和内核函数

发表于 2016-11-10 17:28 | 4776次阅读 | 来源 CSDN | 0 条评论 | 作者 CSDN

摘要：这是我们Adreno™工程师Vladislav Shimanskiy 撰写的Adreno GPU 矩阵乘法系列文章的第二部分，也是最后一个部分。上一个部分Vladislav Shimanskiy解释了Adreno 4xx和5xx GPU系列设备端矩阵乘法（MM）内核函数和主机端参考代码的优化实现相关概念。本文中，他将结合代码分析，详细介绍基于OpenCL的主机代码和内核函数的实...

这是我们Adreno™工程师Vladislav Shimanskiy 撰写的Adreno GPU 矩阵乘法系列文章的第二部分，也是最后一个部分。上一个部分Vladislav Shimanskiy解释了Adreno 4xx和5xx GPU系列设备端矩阵乘法（MM）内核函数和主机端参考代码的优化实现相关概念。本文中，他将结合代码分析，详细介绍基于OpenCL的主机代码和内核函数的实现。

Vlad Shimanskiy是**Qualcomm® GPU**计算解决方案团队的高级工程师。

正如我上次在讨论问题“GPU矩阵乘法存在哪些困难？”时提到的，由于近来依赖于卷积的深度学习引起广泛关注，矩阵乘法（MM）运算也在GPU上变得流行起来。像Adreno GPU这样的并行计算处理器是加速此类运算的理想选择。然而，MM算法需要在各个计算工作项之间共享大量数据。因此，优化Adreno的MM算法需要我们利用GPU内存子系统。

在**OpenCL**中实现

前面已经给大家介绍了常用的四种优化技术，这里，我们进一步介绍在OpenCL中实现这些优化技术的主机参考代码和内核函数，这些参考代码和内核函数你将可以直接应用到你自己的代码中。

主机代码

首先，我们运行防止内存复制的主机代码。如前文所述，一个矩阵通过TP/L1加载，另一个矩阵通过常规全局内存访问路径加载。

两个输入矩阵中的一个矩阵用图像表示方法进行表示，即示例代码中的矩阵B，通过图像对矩阵进行抽象，并利用图像读取原函数访问，如第一部分中的图3所示。对于其他矩阵，都使用全局内存缓冲区进行存储和访问。这也是为什么为矩阵A和矩阵B应用不同的内存分配方式的原因。而在矩阵C的访问和表示中，因为只需要往矩阵C是写入数据，并且每个矩阵元素只需要写一次，到C的流量非常低，所以矩阵C将始终通过直接路径访问。

*矩阵**A**和**C**的内存分配*

下面例程显示了如何分配可以通过直接路径访问的矩阵A和C，这一点相对简单：

```
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;

图4通过**L2**缓存加载的矩阵的内存分配（**A**和**C**）

根据前面介绍，为矩阵A和C分配内存中，我们是想得到一个可以被CPU运算访问的主机指针（CPU指针），并且希望可以通过该指针对CPU上的缓冲区进行写入和读取操作。因此，上述代码的第1行中调用OpenCL的Buffer函数实现了内存分配，并得到了指向CL缓冲区的指针。

- 该驱动程序分配一个缓冲区。
- CL_MEM_ALLOC_HOST_PTR宏表示该内存可以被主机访问。
- 通过na和ma我们可以指定矩阵的水平和垂直维度。

注意，这里的内存不能使用malloc()函数在主机CPU上分配；必须在GPU空间中进行分配，并在CPU代码可以写入之前，将分配得到的内存显式映射到具有CL API映射函数的CPU地址空间。

在调用buffer函数完成了缓冲区内存分配之后，我们必须得到host_ptr指针，在CPU上通过该指针可以访问分配的矩阵内存。

为了得到host_ptr指针，在图4所示代码的第2行中，我们调用了OpenCL API中的enqueueMapBuffer，使用第1行代码中得到的缓冲区指针buf_ptr来获得host_ptr指针。enqueueMapBuffer函数返的host_ptr指针是一个T类型的指针（示例中T是浮点数），使用host_ptr指针可以在CPU上对分配得到的矩阵缓存区内存进行读写。如果我们已经分配了矩阵A，这就是我们用来传递该矩阵的指针。

接着我们看到图4中代码的第3行，这里通过lda 确定矩阵每行使用的内存量，以类型T为单位。因此，如果我们在程序中分配一个100×100矩阵，则lda将为100个T类型长度的内存空间。（注意，lda不一定等于矩阵的水平维度；在某些情况下，lda可能与之不同）。

这里，我们在主机端将lda、ldb和ldc提交给内核，以指定矩阵A、B和C的行距。

*矩阵**B**的内存分配（图像）*

接下来我们来了解矩阵B是如何分配的，矩阵B的分配比前面介绍的矩阵A和C的分配更复杂，因为在矩阵B的分配中我们使用了2D图像。

图像比缓冲区限制更加严格。它们通常拥有4个颜色通道（RGBA），并且在内存中为图像分配内存空间的时候必须保证适当的对齐。这里，我们先假定一个图像，并且图像的每个颜色分量是一个浮点数。如果我们从矩阵的角度来观察图像，我们希望平展颜色分量。如上所述，为提高效率，我们通过一个包括4个float类型数据的向量运算来读取矩阵，将元素按每4个float类型打包到图像像素中。因此，我们在计算过程中必须将矩阵的水平大小除以4，这样我们表示的才是图像的像素数量，具体实现代码如下图5所示：

*cl::Image * img_ptr = new cl::Image2D(*ctx_ptr, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR, cl::ImageFormat(CL_RGBA, CL_FLOAT), na/4, ma, 0);*

cl::size_t<3> origin;

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,*

相关热门文章

2017高等教育信息化创新论坛召开 搜狗校园搜…
桂林银行与华为签订战略合作协议
华为HCIE：能力越大，未来越大
又前进一步！百家号注册优化 审核最快可秒过
一个喷嚏就造成惨烈车祸，有望通过真无线蓝牙…
华为视讯打造稳定快捷指挥调度 ——华为助力平…
水滴筹到底凭什么获“年度十大慈善项目”
2017华为中国城商行峰会成功举办，共话银行数…
附近的小程序功能开放！会给商家带来什么巨大…
美团外卖推外卖行业首款端内IM工具 引领用户…

活动

01-01 英特尔正调查苹果iPhone与PC资料同步化失败问题
01-01 10个windows8应该改进的地方
01-01 Windows7时代,我们如何攒机?
01-01 英特尔高管称Win7普及快于Vista
01-01 XP升级Windows7 硬盘数据被全部清空
01-01 Windows7 RTM大战Vista SP2! Win7性能稍强

CL_MAP_WRITE, origin, region, &row_pitch, &slice_pitch));

ldb = row_pitch / sizeof(T);

图**5**：通过纹理管道（**texture pipe**）**(B)**加载的**float32**矩阵进行内存分配

上述代码中，第1行通过调用OpenCL中的Image2D函数来分配内存，与A和C的内存分配一样，使用了CL_MEM_ALLOC_HOST_PTR宏来指定分配的内存可以从主机端访问。

分配得到图像可以从主机端访问的图像内存后，接着看第8行，通过enqueueMapImage返回可以在CPU端使用的指针host_ptr（和前面矩阵A和C使用的enqueueMapBuffer类似），并确保我们在GPU内存中分配的图像区域对于CPU可见。在CPU端可以通过host_ptr访问到该图像数据。

*从**CPU**调用内核函数*

前面已经介绍了如何分配内存，接下来介绍如何从CPU调用内核函数，该操作包括三个步骤：

- 从CPU中取消映射，使矩阵A和B针对GPU更新。
- 运行内核函数。
- 重新映射，使得矩阵C中的结果对于CPU可见。

这个过程中我们还必须将A和B的内存映射回CPU，以便CPU可以更改这些矩阵；但是，这些更改不能同时被GPU和CPU获取，需要一个同步的过程。在下面的列表中，我们利用了Snapdragon处理器上的共享虚拟内存（SVM）方法来实现内核函数运行周期和内存同步：

// 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 * sizeof(float));*

// 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);*

图6：内核函数运行周期和内存同步过程

上述代码实现分为两个部分，其中第一部分是使用enqueueUnmapMemObject函数调用取消映射过程。需要传递对CPU端矩阵做出的所有改变，使其对于GPU可见，供乘法使用。这是一个缓存一致性事件：我们分配了矩阵A和B，在CPU端传播，然后使它们对GPU可见，而不是复制内存。

完成了第一部分的处理，到了第二部分，GPU现在可以看到分配的矩阵了，并且可以使用。enqueueNDRangeKernel运行将对矩阵进行运算的内核函数。（经验丰富的OpenCL程序员知道如何设置内核函数的参数，为简洁起见，在此予以省略）。

第二部分的其余部分大同小异，不过与第一部分相反。内核函数将矩阵乘以矩阵C，因此现在我们需要使矩阵C对CPU可见。MM运算经常重复，因此我们将A和B内存映射回CPU，为下一个运算周期做好准备。在下一次迭代时，CPU能够为A和B分配新值。

*运行在**GPU**上的内核函数代码*

前面已经知道了如何进行内存分配和内核函数的调用，为了进一步了解整个MM运算的性能，我们来分析运行在GPU上的MM运算内核函数代码，这部分代码说明了拥有float 32格式元素的MM运算的本质。它是BLAS库中SGEMM运算的简化版本，C = αAB + βC，（为简洁起见）其中，α= 1和β= 0。

```
__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);
    int gy = get_global_id(1);
        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;
            for (int pos = 0; pos < k; pos += 4)
        {
            #pragma unroll
            for (int i = 0; i < 4; i++)
            {
                b[i] = read_imagef(Bi, (int2)(gx, pos + i));
            }
            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++)
        {
            c[i] += a[i].x * b[0] + a[i].y * b[1] + a[i].z * b[2] + a[i].w * b[3];
        }
            }
            #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);
        }
    }
            }
```

图7：实现**C = A * B**矩阵运算的内核函数示例

一般而言，我们会展开固定大小的循环，然后将从矩阵A中读取图像和数据的操作进行分组。具体过程如下：

- 开始时，我们设置了一些限制，确保在处理矩阵时不致严重限制其维度，因此可以部分占用工作组。每个工作组水平和垂直地覆盖一定数量的micro-tile，但是视乎不同的矩阵维度，我们可能面临这样的情况，即macro-tile中的micro-tile仅部分被矩阵占用。因此，我们要跳过macro-tile未占用部分中的任何运算；这就是这个条件的作用。矩阵维度仍然必须是4x8的倍数。
- 然后，通过代码将矩阵C的元素初始化为零。
- 最外层的for循环遍历pos参数，并包含三个子循环：
- 第一个子循环中，我们通过拥有read_imagef函数的TP/L1读取矩阵B的元素。
- 第二个子循环包含直接从L2读取的矩阵A的元素值。
- 第三个子循环计算部分点积。
- 注意，为提高效率，所有加载/存储和ALU操作均使用由4个float元素构成的向量。

通过上述代码分析，整个内核函数可能看起来比较简单，但实际上它是一个经过高度优化、均衡的运算和数据大小组合。在使用的过程中南建议使用-cl-fast-relaxed-math标记编译内核函数。

工作组大小

根据上述分析，macro-tile是由多个4×8 micro-tile组成。水平和垂直维度中micro-tile确切数量由2-D工作组大小确定。通常，最好使用较大的工作组，避免GPU计算单元利用不足。我们可以使用OpenCL API函数getWorkGroupInfo查询最大工作组大小。但是，上边界为工作组中工作项的总数。因此，我们仍然可以在总的大小的限制下，自由选择实际的维度组成。以下是查找正确大小的一般方法：

- 最小化部分占用工作组的数量。
- 基于不同大小的矩阵开发启发式算法，并在运行时使用。
- 使用为特殊情况量身定制的内核函数；例如，在矩阵维度特别小的时候。
- 如果GPU卸载开销成为瓶颈，就在CPU上完成小型MM运算。

开始行动

如本文中所示，MM是一项瓶颈运算，因此，您需要在OpenCL代码中利用上述高性能技术。这是一种加速使用Adreno GPU上内存子系统的深度学习应用的有效方法。

更多**Qualcomm**开发内容请详见：**Qualcomm开发者社区**。

推荐阅读相关主题：

相关文章 最新报道

已有0条评论

还可以再输入500个字

有什么感想，你也来说说吧！

haijunz 欢迎您！ 发表评论

最新评论 最热评论

请您注意

·自觉遵守：爱国、守法、自律、真实、文明的原则

·尊重网上道德，遵守《全国人大常委会关于维护互联网安全的决定》及中华人民共和国其他各项有关法律法规

·严禁发表危害国家安全，破坏民族团结、国家宗教政策和社会稳定，含侮辱、诽谤、教唆、淫秽等内容的作品

·承担一切因您的行为而直接或间接导致的民事或刑事法律责任

·您在CSDN新闻评论发表的作品，CSDN有权在网站内保留、转载、引用或者删除

·参与本评论即表明您已经阅读并接受上述条款