Developer Network

Search Site

Search

Search

Log InRegister

Site Navigation

- Get StartedExpand
  - Start Here
  - Android Development
  - Embedded Computing
  - Gaming & Graphics
  - Internet of Things
  - Why Snapdragon Processors?
- SoftwareExpand
  - CompilersExpand
    - Snapdragon LLVM Compiler
  - Specialized SolutionsExpand
    - Adreno GPU SDK
    - AllJoyn Proximal Connectivity Platform
    - AllPlay Click Wireless Home Audio SDK
    - FastCV Computer Vision SDK
    - HEVC Encoder for Servers
    - Hexagon DSP SDK
    - LTE Broadcast SDK
    - Snapdragon Math Libraries
    - Snapdragon Neural Processing Engine
    - Snapdragon VR SDK
    - Symphony System Manager SDK
  - DebuggersExpand
    - Snapdragon Debugger for Eclipse
    - Snapdragon Debugger for Visual Studio
  - ProfilersExpand
    - Adreno GPU Profiler
    - App Tune-up Kit
    - Snapdragon Profiler
    - Trepn Power Profiler
- HardwareExpand
  - Wi-Fi Connectivity for IoTExpand
    - QCA4002/4
    - QCA4010/12
  - RoboticsExpand
    - FIRST Robotics
    - Snapdragon Flight
    - Snapdragon Micro Rover
  - Snapdragon for EmbeddedExpand
    - Which Processor is Right for You?
    - Snapdragon 410E Processor
    - Snapdragon 600E Processor
    - Additional Snapdragon Boards
  - Bluetooth Connectivity for IoTExpand
    - Which BLE Solution is Right for You?
    - CSR102x Product Family
    - CSR101x Product Family
    - BlueCore CSRB534x Product Family
  - Additional SolutionsExpand
    - 2net mHealth Platform
    - Snapdragon 835 VR Development Kit

Matrix Multiply on Adreno GPUs – Part 2: Host Code and Kernel

Monday 10/17/16 01:12pm
|
Posted By Jay Yun
Up 0 Down 0

This is the second and final part of a guest post by Vladislav Shimanskiy, one of our Adreno™ engineers. His previous post explained the concepts behind an optimized implementation of device-side matrix multiply (MM) kernels and host-side reference code for Adreno 4xx and 5xx GPU families. In this post, he walks you through OpenCL listings you can use to implement the kernels and host code.

Vlad Shimanskiy is a senior staff engineer in the GPU Compute Solutions team at Qualcomm®.

As I mentioned last time when I addressed the question, “What’s difficult about matrix multiplication on the GPU?” the MM operation has become very popular on GPUs thanks to recent interest in deep learning, which depends on convolutions. Parallel computing processors like the Adreno GPU are ideal for accelerating that operation. However, the MM algorithm requires a great deal of data sharing between individual computing work-items. So optimizing an MM algorithm for Adreno requires that we take advantage of the GPU memory subsystem.

Implementation in OpenCL

Source code for implementing the four optimization techniques I described in my previous post consists of host reference code and OpenCL kernels. The listings below include code snippets you can apply in your own programs.

Host Code
First, we run host code that prevents memory copies. As mentioned above, one matrix gets loaded through the TP/L1 and the other through the regular global memory access path.

One of two input matrices is represented by an image. It is matrix B in our sample code. We use image abstraction from one matrix and access it using image read primitives, as you’ll see in Figure 3. For the other matrix we use the global memory buffer. That’s why we apply different memory allocation routines to matrix A and matrix B. Matrix C is always accessed by direct path; traffic to C is very low because it’s write traffic and we write each matrix element only once.

Memory Allocation for Matrices A and C
The routine listed below shows how we allocate matrices A and C for direct path access, which is relatively simple:

```
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;
```

Figure 4. Memory allocation for matrices loaded via L2 cache (A and C)

We want a host pointer (CPU pointer) that can be accessed by the CPU operations, and we also want to write to and read from that buffer on the CPU. So line 1 allocates the memory and provides the pointer to the CL buffer.

- The driver allocates a Buffer.
- The special flag CL_MEM_ALLOC_HOST_PTR indicates that this memory will be accessible by the host.
- We specify na and ma, which are horizontal and vertical dimensions of the matrix, respectively.

Note that memory cannot be allocated on the host CPU by using the malloc() function; it has to be allocated in the GPU space and explicitly mapped to the CPU address space with the CL API mapping function before the CPU code can write to it.

Once we have Buffer allocated, we must get host_ptr, the pointer we use on the CPU to access the matrices.

In line 2 we use enqueueMapBuffer from the OpenCL API and the buffer buf_ptr defined in line 1. The result is a type T pointer (T is float in our example), and we can use it on the CPU to write the data into the matrix. If we've allocated matrix A, this is the pointer we use to populate it.

In line 3 lda defines how much memory each row of the matrix will use, in units of type T. So if we allocate a 100 x 100 matrix, lda will equal 100 floats. (Note that lda is not necessarily equal to the horizontal dimension of the matrix; in some cases lda could be different.)

We submit lda, ldb and ldc to the kernel to specify the row pitch of matrices A, B and C.

Memory Allocation for Matrix B (images)
The allocation for matrix B, shown in the listings below, is more complex because we use 2D images.

Images are more restrictive than buffers. They usually have 4 color channels – RGBA – and must be allocated with proper row alignment in memory. We're trying to pretend that we have an image, but each color component of the image is actually a floating point number. When we look at the image from a matrix perspective, we want to flatten the color components. As noted above, for efficiency reasons we read matrices by float4 vector operations, and it's convenient to pack the elements by 4 into image pixels. That's why we have to divide the horizontal size of the matrix by 4, which becomes the number of pixels.

```
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, CL_MAP_WRITE, origin,
region, &row_pitch, &slice_pitch));
ldb = row_pitch / sizeof(T);
```

Figure 5: Memory allocation for float32 matrices loaded via texture pipe (B)

In line 1 the first function call allocates memory using the same flag, CL_MEM_ALLOC_HOST_PTR, so that we can access that area from the host side.

In line 8 the second call is enqueueMapImage; recall that we used enqueueMapBuffer for matrices A and C. enqueueMapImage gives us the other host pointer and ensures that the image area we allocated in GPU memory becomes visible to the CPU. The operation makes sure the CPU and GPU caches are coherent with regard to the image data.

Invoking the Kernel from the CPU
This operation comprises three steps:

- Unmapping from CPU to make matrices A and B updated for the GPU.
- Running the kernel.
- Mapping back so that the results in matrix C are visible to the CPU.

We must also map A and B back to the CPU so that the CPU can make changes to those matrices; however, the changes cannot be available to both the GPU and CPU at the same time. In the following listing we take advantage of shared virtual memory (SVM) on the Snapdragon processor:

```
// 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);
```

Figure 6: Kernel run cycle and memory synchronization procedures

The first part is an unmapping procedure using enqueueUnmapMemObject. It is required to propagate all the changes made to the matrices on the CPU side to make them visible to the GPU for multiplication. That is a cache coherency event: We allocated matrices A and B, we populated them on the CPU side and now we make them visible to the GPU without copying memory.

In the second part, the GPU can now see the matrices. enqueueNDRangeKernel runs the kernel that will operate on the matrices. (Experienced OpenCL programmers will know how to set the arguments for the kernel, which I omit here in the interest of brevity.)

The remainder of the listing is similar but opposite to the first part. The kernel has multiplied the matrices to matrix C, so now we need to make matrix C visible to the CPU. MM operations often get repeated, so we map A and B memories back to the CPU to get ready for the next cycle. On the next iteration the CPU will be able to assign new values to A and B.

Kernel Code Running on the GPU
This final listing illustrates the essence of MM with elements in float32 format. It is a simplified version of the SGEMM operation from the BLAS library, C = αAB + βC, where α = 1 and β = 0 for purposes of brevity.

```
__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);
  }
 }

}
```

Figure 7: Example of a kernel implementing a C = A * B matrix operation

The general idea is that we unroll the loops of fixed size, then group the read operations of image and data from matrix A. To be more specific:

- In the beginning we set some limitations to ensure we can deal with matrices without severely restricting their dimensions, so the work-groups can be partially occupied. Each work-group covers a certain number of micro-tiles horizontally and vertically, but depending on the matrix dimension, we may face the situation that only part of those micro-tiles in the macro-tile are occupied by the matrix. So we want to skip any operations in the non-occupied part of the macro-tile; that's what this condition does. Matrix dimensions must still be multiples of 4x8.
- Then the code initializes elements of matrix C to zero.
- The outermost for-loop iterates over the pos parameter and contains three sub-loops:
- In the first sub-loop we read elements of matrix B through the TP/L1 with the read_imagef function.
- The second sub-loop contains reads of elements of matrix A from L2 directly.
- The third sub-loop calculates partial dot products.
- Note that all load/store and ALU operations use float4 vectors for efficiency.

The kernel may look simple, but in fact it is a highly optimized, balanced mix of operations and data sizes. We recommend that you compile the kernel with the -cl-fast-relaxed-math flag.

Work-Group Size
As mentioned above, a macro-tile consists of a number of 4x8 micro-tiles. The exact number of micro-tiles in horizontal and vertical dimensions is defined by the 2-D work-group size. It is generally better to use bigger work-groups to avoid underutilization of GPU compute units. We can query the maximum work-group size with the OpenCL API function getWorkGroupInfo. However, the upper boundary is provided as the total number of work-items in the work-group. So we still have the freedom to choose the actual dimension composition satisfying the total size constraint. Here are general ideas for finding the right size:

- Minimize the number of partially occupied work-groups.
- Develop heuristics based on experimentation with matrices of different sizes and use them at run-time.
- Use kernels tailored for special cases; for example, when a matrix has an exceptionally small dimension.
- Complete small MM jobs on the CPU when the overhead of offloading to the GPU becomes a bottleneck.

Get Started

As we've demonstrated in this post, MM is a bottleneck operation, so take advantage of the high-performance techniques described above in your own OpenCL code. It's an efficient way to accelerate your deep learning applications using the memory subsystem on the Adreno GPU.

Have a crack at it and send me your questions and comments below.

Related Blogs:

Start Cooking with Heterogeneous Computing Tools on QDN
Matrix Multiply on Adreno GPUs – Part 1: OpenCL Optimization
Better OpenCL Performance on Qualcomm Adreno GPU – Memory Optimization
Guest Blog: Silk Labs Build Their Product Using Development Boards & Resources From Qualcomm Developer Network
Introducing Vulkan – Explicit Control Over Graphics Acceleration

Related Tags:

- adreno gpu

Comments

Login or Register
to post a comment.

Share

About the Blogger


Jay Yun

Jay Yun is a Principal Engineer on the Graphics Compute Solutions team at Qualcomm Technologies, Inc., where he leads performance analysis and architecture enhancements for compute applications.

Subscribe to Blogs

Log in to subscribe to blog updates by email.

Blog Topics

Wearables
Snapdragon Tools for Android
Developer of the Month
Computer Vision
Development Devices
Gaming & Graphics
Mobile & Wireless Health

Internet of Things
Android

Most Read Blogs

Multi-threading Android Apps for Multi-core Processors – Part 1 of 2
Peer-to-Peer Apps on iOS, Android & Windows 8 with AllJoyn
Why Wait for Commercial Devices to Start Your Development? Your Snapdragon S4 MDP is Now Available.
Enter to Win a Snapdragon 805 Mobile Development Platform

Search Site

Search

Search

Sign Up for Developer News & Updates

E-mail *

Sign Up

Follow Us

- Youtube
- Slideshare
- RSS

- Get Started
  - Start Here
  - Android Development
  - Embedded Computing
  - Gaming & Graphics
  - Internet of Things
  - Why Snapdragon Processors?
- Software
  - Compilers
  - Specialized Solutions
  - Debuggers
  - Profilers
- Hardware
  - Wi-Fi Connectivity for IoT
  - Robotics
  - Snapdragon for Embedded
  - Bluetooth Connectivity for IoT
  - Additional Solutions
- Downloads
  - Software Development
  - Hardware Development
- Forums
  - Software Development
  - Hardware Development
- Community
  - Projects
  - Case Studies
  - Blogs
  - Get Noticed
  - Stay Informed

- Follow Us
- About Us
  - About Us
  - Events
  - Stay Informed
  - Contact Us

Footer Links

Sitemap
Privacy
Terms of Use
Cookie Policy