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 1: OpenCL Optimization

Monday 10/10/16 01:54pm
|
Posted By Jay Yun
Up 0 Down 0

The matrix multiply (MM) operation has become very popular on GPUs thanks to recent interest in deep learning, which depends on convolutions. We've been hearing from developers who want to accelerate deep learning (DL) applications on Qualcomm® Snapdragon™ processors with Adreno™ GPUs.

This is a two-part guest post by Vladislav Shimanskiy, one of our Adreno engineers. The concepts in this post and the OpenCL code listings you'll see in my next post represent an optimized implementation of device-side matrix multiply kernels and host-side reference code for Adreno 4xx and 5xx GPU families. We hope this series will help and encourage you to write your own OpenCL code using the ideas and code samples.

Vlad Shimanskiy is a senior staff engineer in the GPU Compute Solutions team at Qualcomm. He has been working on development and prototyping the new OpenCL 2.x standard features on Snapdragon, improvement of Adreno GPU architecture for compute and acceleration of important linear algebra algorithms, including the matrix multiplication on the GPU.

Parallel computing processors like the Adreno GPU are ideal for accelerating linear algebra operations. However, the MM algorithm is unique among intensively parallel problems in that it requires a great deal of data sharing between individual computing work-items. In the matrices to be multiplied – say, A and B – each element contributes many times to different components of resulting matrix C. So optimizing an MM algorithm for Adreno requires that we take advantage of the GPU memory subsystem.

What's Difficult About Matrix Multiplication on the GPU?

When we attempt to accelerate MM on the GPU, the data sharing problem noted above breaks down into several related problems:

- MM operates on the same values repeatedly, but the larger the matrix, the more likely that we must go out to (slow) memory for values that we've had to replace in the cache, which is inefficient.
- In a naïve implementation of MM, it would be natural to map scalar matrix elements to separate work-items. However, reading and writing scalars is inefficient because the memory subsystem and Arithmetic Logic Units (ALUs) on the GPU are optimized for vector operations.
- Loading elements of large matrices A and B at the same time contributes to the risk of conflicts in caches and contention in memory buses.
- Memory copying is slow, so we need a better way to make data visible to both CPU and GPU.

These problems complicate the main tasks of MM: reading the same values many times and sharing data.

OpenCL Optimization Techniques for Matrix Multiplication

We've specified an OpenCL implementation that includes techniques to address each of the problems.

1. Tiling

The first well-known problem is to minimize repetitive reading of the same matrix elements from slow memories, such as higher-level caches and DDR. We have to try to group memory accesses (reads and writes) so that they are close to one another in the address space.

Our technique for improving data re-use is to split input and output matrices into sub-matrices called tiles. We then enforce the order of memory operations so that the dot products resulting from matrix multiplication are partially completed in the entire tile before we move reading pointers outside of the tile boundaries.

Our algorithm recognizes two levels of tiling: micro-tiles and macro-tiles. The following figure represents how we map the matrices to multiply components in matrix A by components in matrix B and arrive at the single dot product in matrix C:

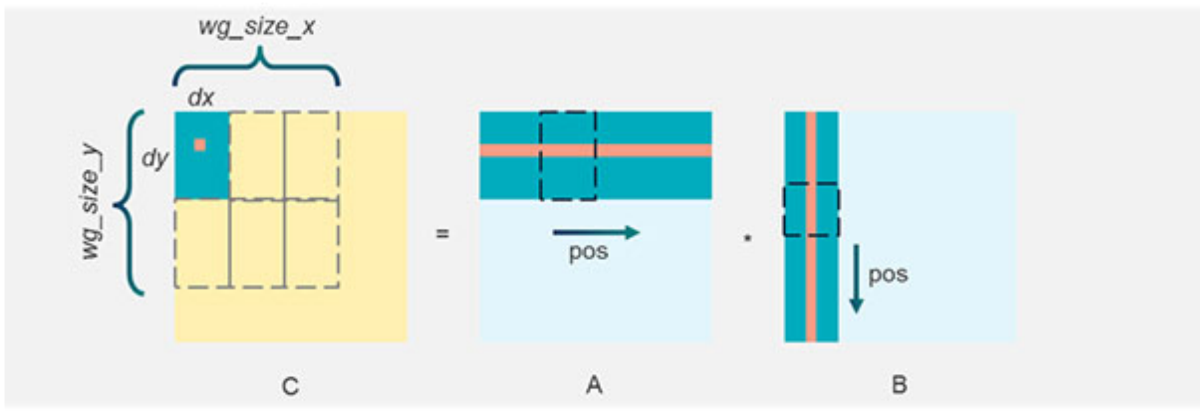

Figure 1: Tiling

The micro-tile – here, {dx, dy} – is a rectangular area inside a matrix that is processed by a single work-item in the kernel. Each work-item is a single thread within a SIMD sub-group that in turn forms an OpenCL work-group. A typical micro-tile has 4 x 8 = 32 components called pixels.

The macro-tile – here, {wg_size_x, wg_size_y} – is generally a bigger, rectangular area consisting of one or more micro-tiles and corresponding to a work-group. Within the work-group, we operate entirely within the macro-tile.

To calculate the 4x8 micro-tile in matrix C, we focus on areas in matrices A and B that have sizes 4x8 and 4x4 respectively. We start at pos = 0, calculate a partial result, or dot product, and store it in a temporal buffer for that micro-tile. Meanwhile, the other work-items in the same macro-tile calculate their partial results in parallel, using the same data loaded from matrix A or matrix B. All data in the row coming from matrix A gets shared. Likewise, all data in the column coming from matrix B gets shared between the work-items in the same column.

We calculate partial results for all the micro-tiles in the macro-tile, then we increment pos horizontally in A and vertically in B at the same time. By doing tile-focused calculations and keeping the pos incrementing gradually, we can maximize reuse of data already in the caches. The micro-tile continues to accumulate, or convolve, partial results that add up to the dot product.

So we limit ourselves to all the positions inside the macro-tile and finish all the partial calculations before we move position. We could complete the entire micro-tile, swiping through pos from left to right and top to bottom, and then advance, but that's inefficient because we need the same data, which the cache has since evicted. The point is that we're working in an area limited by the work-group with a number of work-items progressing concurrently. This approach guarantees that all memory requests from the parallel work-items are issued within the bounded address area.

Tiling optimizes the operation by focusing on a particular area in memory – a work-group – so that we can work in a cache-friendly manner. That is vastly more efficient than jumping across large swaths of memory and having to go out to DDR for values that are no longer in the cache.

2. Vectorization

Since the memory subsystem is optimized in hardware for vector operations, it is better to operate with vectors of data than with scalars, and to have each work-item deal with a micro-tile and a full vector. As a result, we use all values obtained in each vector-read operation.

For example, in the case of 32-bit floating point matrices, our kernels are written to use vectors of float4 type instead of just float. That way, if we want to read something from the matrix, we read not just a single floating point component of the matrix, but an entire block of data. That's important because it matches how buses are designed, so we read components from the matrix in chunks of 4 elements and saturate the bandwidth to memory. Correspondingly, micro-tile dimensions are all multiples of 4.

If we were working on the CPU, we would likely read a 2-D array one scalar element at a time, but OpenCL on the GPU offers a better way. To make the reads and writes efficient, we operate by variables of data type float4, or multiples of float4.

3. Texture Pipe

Using independent caching (L2 direct and Texture Pipe/L1) for two matrices, as shown in the figure below, allows us to avoid most of the contention and parallelize read operations so that the data from matrix A and matrix B gets loaded at the same time. Involving L1 helps to substantially reduce read traffic to L2.

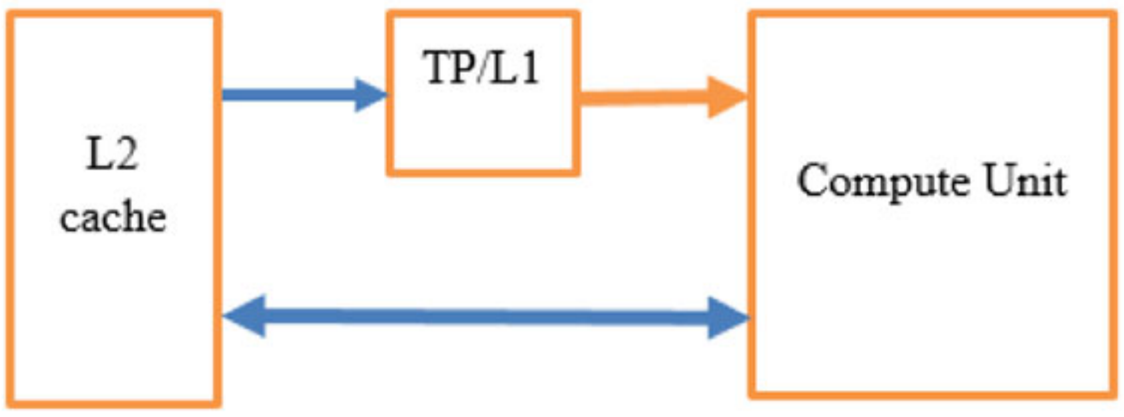

Figure 2: Texture Pipe

In Adreno as in many other GPUs, each compute unit has independent connections to a texture pipe (TP) unit. The TP has its own L1 cache and an independent connection to the L2 cache.

The trick we use to increase the bandwidth is to load one matrix through TP and the other through the direct load/store pipe. Because we're reusing components so much in matrix multiplication, we get the advantage of the L1 cache as well. We end up having much higher traffic from TP/L1 to the compute unit than from L2 to L1. That block reduces the traffic significantly. If, instead of the TP, we had just another connection to L2, it wouldn't help much because we'd have lots of contention and arbitration between the two buses.

The result is a great deal of traffic on the direct connection and very little from TP/L1 to L2. That helps us to increase total memory bandwidth, balance the ALU operations and achieve much higher performance. We spend nearly as much time on the ALU operation as we spend waiting for data to come back from caches, and we can pipeline them so that neither becomes a bottleneck.

4. Memory Copy Prevention

Our OpenCL implementation has two components: the kernel running on the GPU, and the host code running on the CPU and controlling execution of the kernel. If we implement a GPU-accelerated library such as BLAS to multiply matrices, then the input matrices will be in CPU virtual memory space and the result of the multiplication has to be available in CPU memory as well. To accelerate the matrix multiply on GPU, the matrices must first be transferred to GPU memory.

The traditional way of doing this is to copy the matrices into the GPU address space, let the GPU perform its calculations, then copy the results back to the CPU. However, the time it takes to copy big matrices can be a significant fraction of the total time to compute on the GPU, so we want to avoid the inefficiency of memory copying by the CPU. Here, the Adreno GPU has the advantage of shared memory hardware on the Snapdragon processor that we can use instead of copying memory explicitly.

Why not simply allocate memory that is automatically shared between CPU and GPU? Unfortunately, it doesn't work that way, because we need to address constraints like alignment. Only if the allocation is done properly with OpenCL driver routines can shared memory be used.

Results

The graph below depicts the performance increase across Adreno versions for Single-Precision General Matrix Multiply (SGEMM):

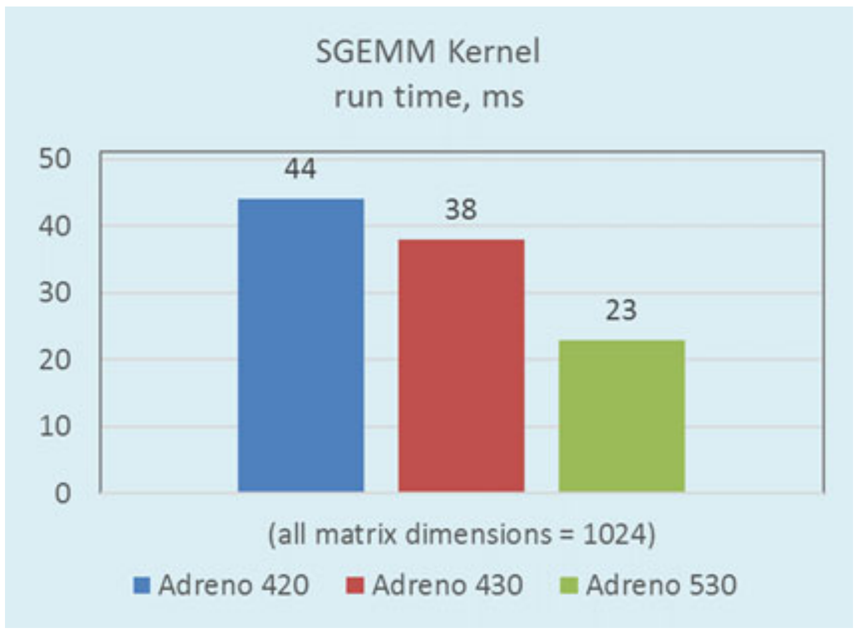

Figure 3: Performance figures for Adreno GPU 4xx and 530

The graph is based on data from common floating point operations. Other MM kernels using different data types (8-bit, 16-bit, fixed point, etc.) can be efficiently implemented following the same principles we apply for SGEMM.

In general, our implementation of MM optimized for the Adreno GPU is at least two orders of magnitude faster than the naïve implementation.

What’s Next?

In my next post I’ll give you the OpenCL code listings behind these concepts.

Matrix multiplication is an important basic linear algebra operation in convolutional neural networks. In particular, the performance of DL algorithms is tied to MM because all variations of convolution in DL can be reduced to multiplying matrices.

The concepts described above and the code you’ll see in the next post are not the only way to do convolutions. But the fact is that many popular DL frameworks like Caffe, Theano and Google’s TensorFlow often reduce convolution operations to MM, so it’s a good idea to follow the momentum in that direction. Stay tuned for code samples in part 2.

Related Blogs:

Start Cooking with Heterogeneous Computing Tools on QDN
Matrix Multiply on Adreno GPUs – Part 2: Host Code and Kernel
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

Login or Register
to post a comment.

Comments

Re: Matrix Multiply on Adreno GPUs – Part 1: OpenCL Optimization

Submitted by degepengcuo on Fri, 2016-11-11 03:11

It costs too much time to lanch kernel. What can I do?

Login or Register
to post a comment.

Share

- G+1
- Tweet
- Share 2

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