An Independent Evaluation
of
Implementing Computer Vision Functions with OpenCL on the Qualcomm Adreno 420

By the staff of

Berkeley Design Technology, Inc.

July 2015

OVERVIEW

Computer vision algorithms are becoming increasingly important in mobile, embedded, and wearable devices and applications. These compute-intensive workloads are challenging to implement with good performance and power-efficiency. In many applications, implementing critical portions of computer vision workloads on a general-purpose graphics processing unit (GPU) is an attractive solution.

Qualcomm enables programming of the Adreno GPU in its Snapdragon application processors via the open standard OpenCL language and API. OpenCL support enables programmers to offload computer vision algorithm kernels to the GPU, which in turn provides speed and power-consumption advantages over a CPU implementation.

BDTI developed an Android application demonstrating computer vision functionality utilizing the Adreno 420 GPU in Qualcomm's Snapdragon 805. The BDTI application can run compute-intensive computer vision functions on the GPU or the CPU, enabling comparisons of GPU and CPU performance and power-efficiency. This paper discusses the BDTI application, implementation and optimization techniques used in its development, and the substantial benefits observed when offloading compute-intensive kernels to the GPU.

Contents

1. Introduction

Computer vision promises to bring exciting new features and user experiences to mobile devices such as smart phones, tablets and wearables. Many vision-enabled applications are already commonplace: photo-stitching techniques enable automatic generation of panoramic views from a series of images, feature detection and tracking techniques enable augmented reality experiences, and so on. And computer vision algorithms and techniques are advancing rapidly, promising to deliver many new features and applications. But computer vision algorithms tend to be very compute-intensive, threatening to bog down mobile devices' CPU cores and memory bandwidth, and to drain their batteries.

Mobile application processors increasingly include general-purpose graphics processing units: graphics processors (GPUs) capable of massively-parallel general-purpose computing. These specialized processing engines are often a great fit for computer vision algorithms, which are typically characterized by very high data parallelism (''Data parallelism" refers to the ability to distribute data among many parallel compute resources such as those available in a GPU). To implement computer vision applications, GPUs can be programmed using the open standard OpenCL language and API from the Khronos Group. For example, the Qualcomm's Snapdragon 805 application processor includes the Adreno 420 GPU. The Adreno 420 can be programmed with OpenCL to offload vision algorithms from the CPU cores, increasing performance and reducing power consumption. In this paper we explore how BDTI used OpenCL to offload vision algorithms from the CPU to the GPU in a demonstration application, and discuss the improvements in performance and power consumption obtained using the Qualcomm Adreno GPU in this way.

The BDTI Background Blur OpenCL Android application was designed to run on the Snapdragon 805 MDP tablet reference design from Intrinsyc. The application detects the largest foreground object in the camera's view—usually the user—and displays the video with the foreground object shown unaltered, and the background shown with severe blurring applied. This functionality could be useful, for example, in a video conference call where we may want to obscure confidential information on a whiteboard behind the user, while the image of the user is seen clearly.

The BDTI Background Blur OpenCL application includes two modes of operation: in the default "GPU" mode, the computationally-intensive portions of the algorithm are offloaded to the GPU using OpenCL. For comparison, a "CPU" mode is provided in which the computationally-intensive portions of the algorithm are executed on a single CPU core. This demo therefore clearly illustrates the capabilities of the Qualcomm Adreno GPU for computer vision applications.

2. Algorithm Overview

The BDTI Background Blur OpenCL application employs cutting-edge background subtraction techniques to separate foreground objects from background. The algorithm is illustrated in Figure 1.

The algorithm begins with a background subtraction kernel. This kernel employs a modified version of the background subtraction algorithm described in [1]. This kernel continuously updates a model of the background, and compares each pixel in the current frame to the background model to estimate the background/foreground classification of the pixel. The background model contains multiple samples of pixel intensity and local binary pattern descriptors for each color component at each pixel position. At each pixel position, the kernel searches the samples in the background model for samples that match the color intensities and local binary pattern descriptors for the pixel position in the current frame. If sufficient matches are found, the pixel is estimated to be in the background. Otherwise it is estimated as foreground. In addition to the background/foreground

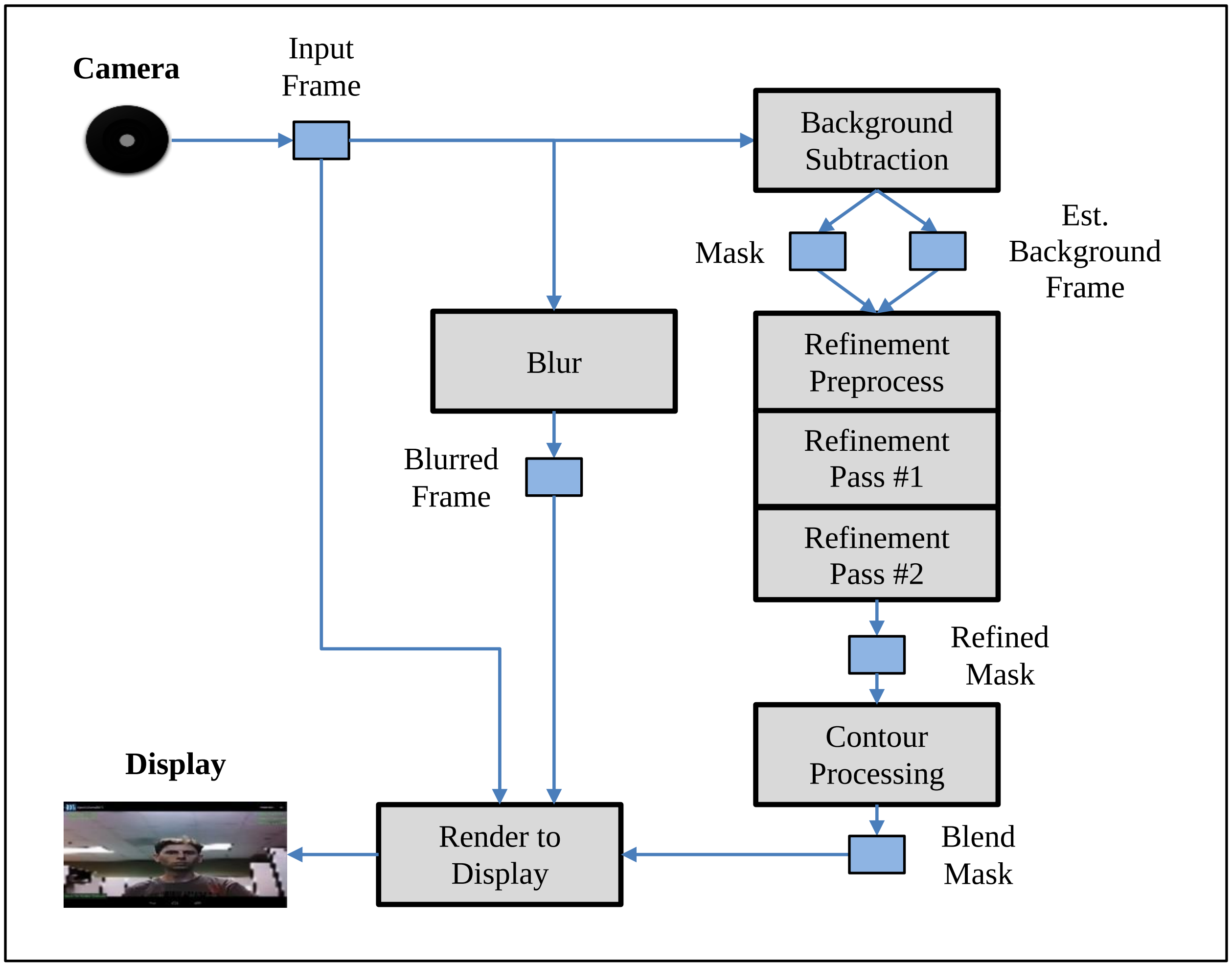

Figure 1 BDTI Background Blur Algorithm

segmentation mask, the background subtraction kernel outputs an estimated background image.

When areas of the foreground closely match the background in both color and intensity, the background subtraction kernel can sometimes misclassify an entire region of foreground pixels as background. To greatly reduce this artifact, the application refines the estimated foreground mask using a proprietary method developed by BDTI and inspired by k-means clustering. The refinement compares pixels that are marked as background in the estimated mask to nearby pixels marked as foreground in the estimated mask; it marks a background pixel as foreground if it is more similar to nearby foreground pixels than it is similar to the estimated background image. This refinement procedure fills in portions of the foreground mask where the foreground pixels are too close to the background model to be correctly marked as foreground by the background subtraction kernel. The refinement method is performed in three stages:

Preprocessing: in this stage the estimated foreground mask is eroded with a 3×3 structuring element to reduce false positives, and a dynamic threshold is computed for each pixel position. The dynamic threshold is used in the following stages.

First-pass refinement search: in this stage, for each pixel marked as background in the estimated mask, the algorithm searches for nearby foreground pixel matches, and marks the background pixel as foreground if enough matches are found.

Second-pass refinement search: this stage is identical to the previous stage, further refining the estimated foreground mask output by the previous refinement pass.

Finally, the algorithm applies morphological operations, finds the contours of foreground objects, selects the largest contour, and removes all other contours from the foreground mask. The resulting mask is used to blend the original video frame with a blurred version of the same video frame. The final refined and contour-processed mask is also passed to the background subtraction kernel along with the next video frame, where the processed mask is used to control thresholds and background model updates.

To reduce computational requirements, the background subtraction and refinement kernels downsample the input frame by a factor of two horizontally, and by a factor of four vertically. This downsampling is performed by simply accessing a subset of the input pixels—a downsampled image is never physically generated in memory. Due to this downsampling of the input, the blend mask is generated at the downsampled resolution. The blend mask is passed to OpenGL-ES as a texture, and is automatically upsampled by the GPU to the original frame size during texture mapping. Because the original input image is never physically downsampled, the sharpness of the rendered foreground pixels is not impacted. Therefore, downsampling dramatically reduces computational demand with negligible impact on output quality.

3. Implementation Overview

The BDTI Background Blur OpenCL demo application is architected to realistically portray the advantages of the Adreno GPU in vision-enabled applications. The background subtraction and refinement kernels are very computationally intensive and comprise the bulk of the processing in the application. Therefore, optimizations focus on these kernels. To ensure representative performance in both the default GPU mode and in the CPU mode, thorough and reasonable optimizations are employed in both modes. Optimizations of the application's software architecture are discussed in this section, and optimizations of the GPU and CPU implementations of the kernels are discussed in Section 4 and Section 5, respectively.

When offloading computation from a CPU to a GPU, a key consideration on most hardware platforms is the need to copy data between CPU and GPU memory spaces. Memory copies introduce latency and consume power, especially for large buffers such as video frames. The BDTI Background Blur OpenCL demo application is therefore designed to minimize the need to move data between CPU and GPU memory spaces.

The input video frame is needed by the GPU in both the CPU and GPU modes of operation, since the GPU renders the output in both modes. The input video is also needed by the CPU in CPU mode, but isn't needed by the CPU when operating in GPU mode. Video frames are fetched from the camera directly into GPU memory space, eliminating the need to copy input frames in GPU mode.

Figure 2 depicts the partitioning of the algorithm among processors and APIs on the Snapdragon application processor. The background subtraction and refinement kernels are implemented in OpenCL in the GPU mode, and in C using ARM NEON compiler intrinsics in the CPU mode.

The contour processing portion of the algorithm is implemented on the CPU using Qualcomm's FastCV library. Contour tracing kernels are difficult to parallelize efficiently, and therefore are not an attractive target for GPU optimization. Qualcomm's FastCV library includes a highly optimized CPU implementation of contour tracing. In GPU mode, the refined foreground mask must be copied from GPU memory to CPU memory for contour processing. In both modes, after contour processing the resulting blend mask must be copied to GPU memory for rendering. Because the subsampled masks are only a fraction of the size of a video frame, these copies have a relatively small impact on speed and power consumption.

In the GPU mode of the application, the CPU can perform contour processing in parallel with the background subtraction and refinement kernels running on the GPU. To enable this parallel operation of the CPU and GPU, the application is pipelined as illustrated in Figure 3. For each frame, background subtraction and refinement consume the frame directly from the camera, while the contour processing and rendering to the display consume a mask, an input frame, and a blurred frame from the previous frame period.

The application uses OpenGL-ES to render output to the display. In addition, the application uses OpenGL-ES to render the input video frame to two textures. One is a low-resolution texture, which results in blurring of the background when this texture is interpolated back to full resolution during rendering to the display. The other texture is a full-resolution texture, which implements a

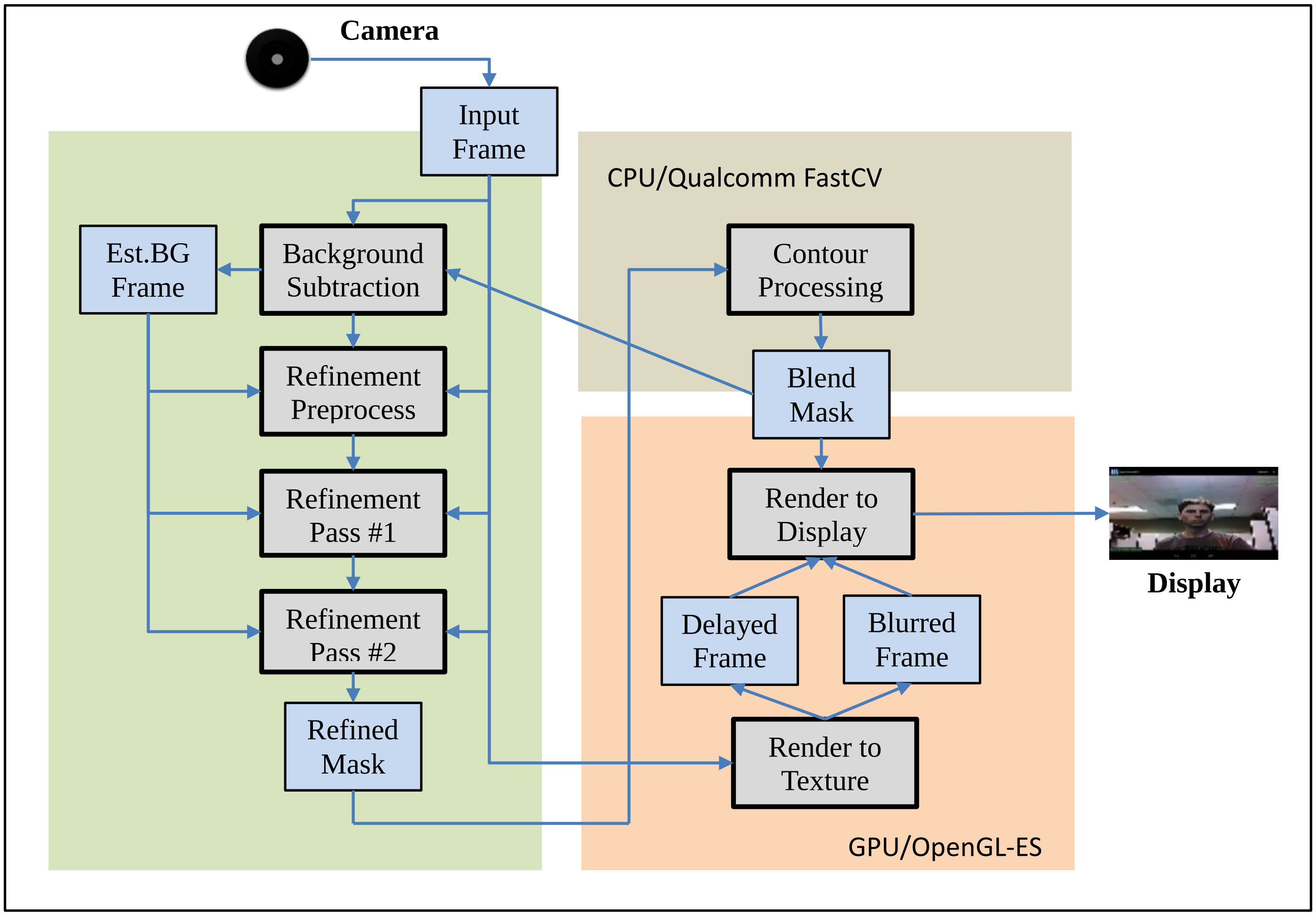

Figure 2 BDTI Background Blur implementation partitioning

one-frame delay needed due to the software pipelining of the application.

4. Adreno GPU OpenCL-Accelerated Implementation

In OpenCL, data-parallel algorithm kernels are broken down into a large number of very small "work items." A work item typically represents the set of operations for processing a single pixel or small group of pixels. The implementation is designed to minimize—and hopefully eliminate—any dependencies between work items so that work items can execute in parallel. Programmers generally think of work items as independent parallel threads, and GPGPUs typically execute many work items in parallel. Spinning off hundreds or in some cases even thousands of work-items enables the GPU to hide memory latency.

OpenCL also provides Single Instruction Multiple Data (SIMD) capabilities, with explicit support for two-, four-, eight-, and sixteen-element vectors. This enables programmers to take advantage of additional data parallelism within a work item.

In the GPU mode of the BDTI Background Blur OpenCL application, background subtraction and refinement are offloaded to the Adreno GPU via OpenCL. The OpenCL code comprises three kernels: background subtraction with local binary patterns, refinement pre-process, and refinement search. The refinement search kernel is executed twice per frame. All of the OpenCL kernels are carefully refactored[1] and SIMD-optimized to expose the inherent parallelism of the algorithms and efficiently utilize the resources of the Adreno 420 GPU architecture. All three kernels operate

[1] Code refactoring is the process of restructuring existing code without changing its external behavior.

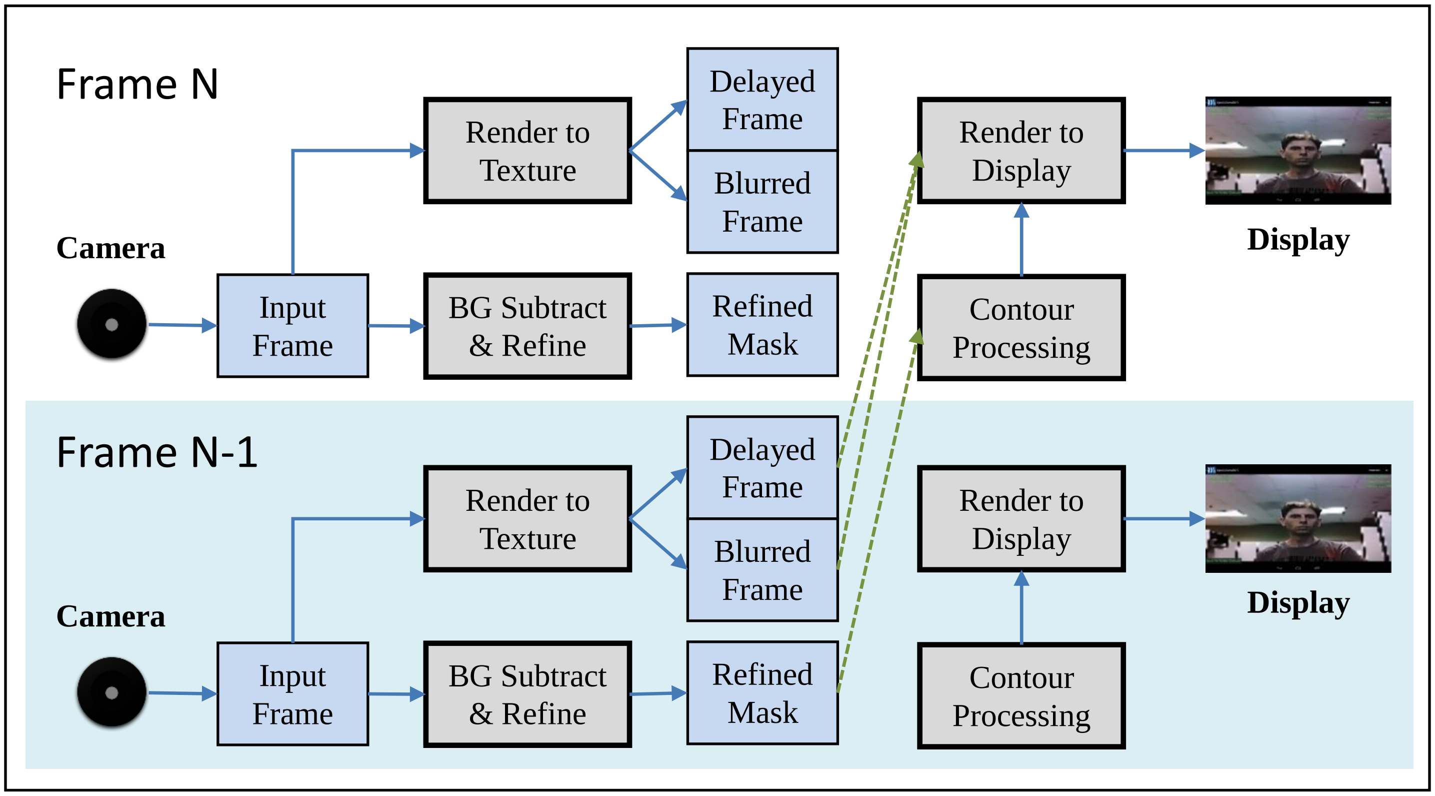

Figure 3 Pipelining of Background Blur Implementation

on one pixel position per work-item. Additional OpenCL implementation and optimization considerations are discussed below.

Memory Footprint and Local Memory

Data-intensive algorithms usually require efficient use of fast local memories for optimal performance. On a CPU, for example, algorithms can be refactored to optimize utilization of L1 caches. On a GPU, fast local memory must be explicitly managed, or else performance suffers dramatically. A collection of work-items is referred to in OpenCL parlance as a work-group, and each work-group has access to a pool of fast local memory.

To minimize the impact of long DDR access latencies, each work-item copies the state and input data it needs from DDR into variables and arrays residing in local memory. The work-item then operates on this data locally. This idiom is utilized for all three OpenCL kernels in the BDTI Background Blur OpenCL application, with some important differences among the kernels described below.

In the background subtraction kernel (unlike most computer vision kernel functions), each work-item accesses only a minimal amount of data from neighboring pixel positions. Although the background model includes many background samples per pixel position, there is little overlap in state and input data among the kernel's work-items. Therefore, each work-item can copy its state and input data into local variables and arrays without duplicating the copies performed for neighboring pixels –thus avoiding overflowing the local memory and/or causing redundant accesses to slow DDR memory.

The OpenCL code for the background subtraction kernel copies state and data from DDR into local variables, but it does not explicitly declare its local copies of input and state data as residing in local memory. For this kernel it was not necessary to manage local memory more explicitly, probably because most local variables and arrays fit in GPU registers for this kernel, and the minimal overlap with neighboring pixels means that even without more explicit techniques few DDR memory access conflicts occur. This is in contrast to the refinement pre-processing and search kernels, where more explicit memory management is required.

The refinement pre-process and refinement search kernels both process a neighborhood of pixels centered on each pixel position. Therefore, most of the input data for each pixel position overlaps with the input data of neighboring pixel positions in these kernels. Explicit management of GPU local memory is needed to avoid redundant copies as each work-item copies its input into local variables and arrays.

Work-items in these kernels are grouped into an eight-by-eight tile of pixel positions per work-group. Local arrays are used to store the input data required for an entire eight-by-eight tile, and are shared by all of the work-items in the respective work-group. Each work-item copies a small portion of the input data for the entire work group into the local array, and the portions fetched by work items do not overlap. After copying the data, work-items within a work-group synchronize using OpenCL's "barrier" mechanism to ensure that all input data has been loaded before work-items proceed to perform their computations. The work-items thus cooperate to fetch overlapping inputs from slow DDR memory. This implementation technique eliminates redundant copies of data in local memory, reduces redundant accesses to DDR, and helps the GPU hide the latency of slow DDR memory accesses.

Per-Pixel Pseudo-Random Number Generation

To achieve desirable statistical properties, updates of the background model are randomized in the background subtraction kernel. Because a pseudo-random number generator updates its state with each invocation, calling a single random number generator from each work item would create a dependency as all work items attempt to access and update the same state. This dependency would block the work items from executing in parallel. Therefore, each work item includes an independent random number generator with its own state. To ensure that the random number generators for all of the work items are uncorrelated, each random number generator must be randomly seeded at initialization. The OpenCL random number generator code is based on [2].

Conditional Operations and Branches

GPU architectures typically require that many OpenCL work items share a single instruction stream. Multiple execution paths due to data-dependent branches or conditional operations within a work item can therefore reduce performance, oftentimes drastically. OpenCL kernels (and OpenGL shaders) are often refactored to eliminate branches.

The background subtraction kernel includes many branches per pixel. However, this kernel's performance on the Adreno GPU was about twice the performance of the CPU version without requiring refactoring to eliminate the branches. This may be due to very good correlation between the execution paths for the work items at neighboring pixel positions—when a certain branch is taken for one pixel position, it is likely for the same branch to be taken for neighboring pixels. Therefore it is likely that many work items naturally execute the same instruction stream despite the presence of branches. However, this is not always the case, and it may be possible to further improve the performance of this kernel with refactoring to eliminate some of the branches. However, in order to eliminate a branch, the refactored kernel must sometimes perform the work of both branch-taken and branch-not-taken execution paths, thus increasing the computational workload. Optimizing the background subtraction kernel further would therefore require laborious statistical analysis to balance the increase in parallelism gained from eliminating each branch against the resulting increase in computation.

The refinement search kernel is explicitly refactored to eliminate branches. This kernel attempts to match each background pixel against foreground pixels in a neighborhood centered on the background pixel position. The CPU implementation of this kernel includes a branch in the kernel's inner loop: for each pixel position the search is stopped once enough matches are found. On the GPU, however, eliminating branches is more efficient than reducing the workload by terminating the search. Therefore, the OpenCL implementation does not include the stopping condition, and always iterates through the entire inner loop.

Byte-Wide Fixed-Point and Logical Operations

The Adreno 420 GPU includes native support for 16-bit fixed-point data. To support 8-bit data, 16-bit operations are performed by the hardware, with additional operations such as sign extension sometimes added by the compiler in order to guarantee correct functionality. To minimize unnecessary operations, the OpenCL code for all three kernels promotes some 8-bit data to 16-bit fixed-point or 32-bit floating-point data types.

Additionally, the background subtraction OpenCL kernel uses a 256-entry lookup table to perform a population-count operation (the population-count operation counts the number of bits in the input word that have a value of one).

5. ARM CPU NEON-Accelerated Implementation

In the CPU mode of the BDTI Background Blur OpenCL application, background subtraction and refinement are implemented on the CPU and refactored to efficiently utilize the CPU caches. The refinement pre-process step is split into independent operations: the erosion operation is implemented with a call to Qualcomm's FastCV library, and the threshold computation is interleaved with the refinement search as described below.

The threshold computation and two refinement search passes are pipelined on a scan-line basis, with a five scan-line delay between the first and second refinement search passes. Pipelining these functions greatly improves cache utilization and is paramount to achieving good performance on the CPU.

Background subtraction and all refinement steps are carefully optimized using ARM NEON instructions to perform SIMD-parallelized operations. NEON optimizations make use of NEON's native support for eight-bit data and native population-count instruction.

6. Benefits of OpenCL Acceleration on the Adreno GPU

The BDTI Background Blur OpenCL Android application illustrates the advantages of the Adreno 420 GPU over the ARM CPU, for massively parallel algorithms programmed in OpenCL. Comparing the application's behavior in the GPU and CPU modes of operation reveals the performance benefit of the GPU.

The computational workloads of the background subtraction and refinement kernels are data dependent. Furthermore, the computational workload's dependencies on input data vary somewhat between the ARM NEON-optimized code and the OpenCL code. Therefore, precise comparisons of performance of the two modes can be made only for precisely defined operating conditions.

BDTI has not attempted extensive, rigorous performance and power measurements on the application under carefully controlled conditions. Therefore, results measured by BDTI and presented below do not represent a comprehensive range of operating conditions and should be considered as a coarse estimate. However, BDTI has observed the performance of both the CPU and GPU modes under conditions that can be considered typical. The typical difference in performance between the GPU and CPU is striking, as discussed below.

GPU vs. CPU Speed Comparison

A comparison of video display frame rates achieved under typical operating conditions in the GPU and CPU modes, respectively, is shown in Table 1. Overall, the GPU mode typically achieves a frame rate nearly two times higher than that of the CPU mode.

Mode	Typical average frame rate	Observed range
GPU	30 fps	25-33 fps
CPU	16 fps	14-18 fps

Table 1 Frame rate comparison of GPU and CPU modes

Table 2 shows the approximate time in milliseconds per invocation of the compute-intensive background subtraction and refinement kernels on the GPU and CPU. As described in Section 5 above, the two refinement search passes are tightly interleaved on the CPU, along with part of the refinement pre-processing. Therefore, it is not practical to individually profile these processing steps on the CPU. The background subtraction kernel appears to be slightly more than two times faster on the GPU compared to the CPU, although significant data-dependent timing variations occur on both the CPU and GPU. The three refinement steps combined are likewise roughly twice as fast on the GPU compared to the CPU.

Contour processing requires several additional milliseconds of computation on the CPU. In GPU mode, contour processing still executes on the CPU but occurs in parallel with the OpenCL kernels running on the GPU. However, the application incurs some additional overhead in both modes for rendering, synchronization, and housekeeping, reducing the overall speedup of the application to slightly less than a factor of two.

	Background Subtraction	**Refinement Pre-process**	**Refinement Pass #1**	**Refinement Pass #2**	**Contours**
GPU	14 ms	0.9 ms	5 ms	5 ms	n/a
		Refinement total: 11 ms			
CPU	30 ms	22 ms			4-7 ms

Table 2 Kernel duration comparison of GPU and CPU implementations, under typical conditions

Note that the CPU mode of the application uses only one of the Snapdragon processor's ARM cores. Using two CPU cores instead of one, it is possible to achieve a frame rate roughly equivalent to that of the GPU, at the cost of slightly higher code complexity, and greatly increased power consumption.

7. Conclusions

As new computer-vision-enabled user experiences emerge in mobile, embedded, and wearable devices, computational demands will continue to rise, while size, cost, and power constraints will become more stringent. In many products, massively-parallel GPGPU implementations of key algorithm kernels will be critical to meeting application requirements. Qualcomm's support for OpenCL on the Adreno GPU makes this possible on Snapdragon application processors.

As illustrated by the BDTI Background Blur OpenCL demo application, offloading compute-intensive kernels to the Adreno 420 GPU can dramatically reduce CPU utilization in a computer-vision-enabled application, freeing CPU resources to tackle additional applications and features. Additionally, offloading compute-intensive tasks from the CPU can dramatically improve power consumption. Because of their specialized massively-parallel architectures and lower clock rates, GPUs tend to be more power-efficient than CPUs. Although BDTI did not independently measure the power consumption of the BDTI Background Blur OpenCL demo application, Qualcomm has reported that the GPU mode of the demo consumes half as much power as the CPU mode when throttling the frame rate of the GPU mode to match the highest frame rate achieved in the CPU mode.

However, effective GPU programming and code optimization can be tricky. Algorithm implementations must be refactored to maximize parallelism, and conform to the memory system and core architectures of the GPU, as exemplified by the considerations discussed in this paper:

- The application must be architected to minimize memory copies between GPU and CPU memory spaces.
- GPU code must carefully manage limited fast local memory.
- Programmers must be aware of GPU core architectural characteristics, even when programming in a high-level language such as OpenCL. For example, code must minimize the use of branches and take care to utilize the most appropriate SIMD data types.

When implemented with best practices, computer-vision functions run efficiently on the GPU. Qualcomm's Adreno GPU with support for OpenCL will enable vision functions in a wide range of mobile devices and applications.

8. References

[1] P.-L. St-Charles and G.-A. Bilodeau. Improving background subtraction using local binary similarity patterns. In Applications of Computer Vision (WACV), 2014 IEEE Computer Society Winter Conference on, 2014. 1

[2] David B. Thomas. The MWC64X Random Number Generator. Retrieved from http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.htm