Design and Optimization of Image Processing Algorithms on Mobile GPU

Nitin Singhal*
Samsung Electronics Co. Ltd., Suwon 443-742, Korea

Jin Woo Yoo Ho Yeol Choi In Kyu Park†
Inha University, Incheon 402-751, Korea

1 Introduction

The advent of GPUs with programmable shaders on mobile phones has motivated developers to utilize GPU to offload computationally intensive tasks and relive the burden of embedded CPU. In this paper, we present a set of metrics to measure characteristics of a mobile phone GPU with the focus on image processing algorithms. These measures assist users in design and implementation stage and in classifying bottlenecks. We propose techniques to achieve increased performance with optimized shader design. To show the effectiveness of the proposed techniques, we employ cartoon-style non-photorealistic rendering (NPR), belief propagation (BP) stereo matching [Yang et al. 2006], and speeded up robust features (SURF) detection [Bay et al. 2008] as our example algorithms.

2 Metrics to Characterize Mobile GPUs

Memory Transfer Bandwidth: On a mobile phone application processor, the memory is shared between CPU and GPU. However, textures have to be properly wrapped for the graphics core and cannot be accessed directly as the CPU image array. The overhead incurred in texture wrapping serves as a significant bottleneck when operating on large memory buffers. For CPU (ARM CORTEX A8 @1GHz) and GPU (POWERVR SGX 540 @200MHz), the memory bandwidth (processor to memory) stands at 220.54 MB/s (CPU to GPU) and 30.92 MB/s (GPU to CPU);

Floating Point vs. Fixed Point: Mobile phone GPU such as POWERVR SGX 540, has fast vectored floats compared to VFPLite hardware accelerator in ARM CORTEX A8 CPU. As a result, an image processing algorithm with high floating point arithmetic is likely to have high acceleration when implemented on a mobile phone GPU as compared to the CPU. For example, a 5×5 Gaussian filter, implemented on a mobile phone GPU shows 30x and 2x speedup in comparison to CPUs floating and fixed point implementation, respectively;

Shader Instruction vs. Rendering Cycles: On a mobile phone GPU, the number of instruction slots for a vertex and fragment shader is limited. There is a trade-off between combining multiple rendering cycles in a single pass and splitting a single pass into multiple rendering cycles. Packing multiple rendering cycles into a single fragment shader increases the instruction count. On the other hand, increasing the number of rendering cycles reduces the parallel fraction. The best solution depends on the needs of the application.

3 Performance Optimization Techniques

We present optimization techniques which are customized for image processing and address the need for compact shader code that matches the smaller hardware limits of a mobile phone GPU.

Precision Control : Shaders use precision such as *highp*, *mediump*, and *lowp*, to provide hints to the compiler on how the variable is used. The *lowp* is useful for representing colors in the 0.0 to 1.0 range. Choosing a lower precision increases the shader performance.

Loop Unrolling : To process a loop (`for` or `while`), a shader needs more instructions in increment and comparison. Eliminating loop by either an optimized unrolling or using vectors to perform the operations results in lower shader instruction count.

*email: n.singhal@samsung.com
†pik@inha.ac.kr

SIGGRAPH 2011, Vancouver, British Columbia, Canada, August 7 – 11, 2011.
ISBN 978-1-4503-0921-9/11/0008

Table 1: *Fragment shader optimization example.*

Optimization	Instruction Count	Execution Time (ms)
Basic	532	537.63
Loop Unroll	181	105.93
Load Sharing	150	90.25
Precision Control	69	48.90

Table 2: *Execution time comparison (in milliseconds).*

Algorithm	Parameters	CPU	GPU	Speedup
NPR	800×480, 5×5 mask	1545.7	242.7	6.37x
BP Stereo Matching	384×288, 4 iterations	6976	1086	6.42x
	384×288, 8 iterations	12161	2083	5.84x
	384×288, 12 iterations	17413	3125	5.57x
SURF	800×480	1703	943	1.81x

Load Sharing : Accessing neighborhood color values in a fragment shader commonly results in dependent texture read, which results in a stall until the texture information is retrieved. To avoid dependent texture read, a straightforward way is to pre-compute neighboring texture coordinates in a vertex shader.

Texture Compression : Texture compression helps in reducing the memory transfer overhead. If the texture cannot be compressed, a lower precision pixel format such as RGB565, RGBA5551, or RGBA444 should be used.

Table 1 shows the acceleration achieved after different optimization steps on a 5×5 Gaussian filter fragment shader.

4 Experimental Results

Table 2 shows the acceleration results for the test algorithmss. All CPU implementation uses fixed point arithmetic. In case of cartoon-style NPR, the large number of dependent texture lookups force the GPU to execute instructions sequentially, which limit the speedup achieved. High memory access intensity coupled with intensive logical operations significantly increases the execution time for BP stereo matching GPU implementation. SURF implementation achieves the lowest speedup. The overhead incurred in grouping and splitting the integral image values into RGBA texture components in one of the major bottleneck. Additionally, the large number of rendering cycles, inherently limit the maximum speedup.

Acknowledgement

This research was supported by The Ministry of Knowledge Economy, Korea and Samsung Electronics CO., LTD., under IT/SW Creative research program supervised by the NIPA (NIPA-2010-(C1810-1004-0002)).

References

BAY, H., ESS, A., TUYTELAARS, T., AND GOOL, L. V. 2008. SURF: Speeded up robust features. *Computer Vision and Image Understanding 110*, 3, 346–359.

YANG, Q., WANG, L., YANG, R., WANG, S., LIAO, M., AND NISTÉR, D. 2006. Real-time global stereo matching using hierarchical belief propagation. In *Proc. British Machine Vision Conference*, 989–998.