Design and Optimization of Image Processing Algorithms on Mobile GPU

Nitin Singhal, Jin Woo Yoo, Ho Yeol Choi, In Kyu Park
Digital Media & Communication R&D Center, Samsung Electronics Co. Ltd., Suwon, Korea
School of Information and Communication Engineering, Inha University, Incheon, Korea

Introduction

- We present a set of metrics to measure characteristics of a mobile phone GPU with the focus on image processing algorithms
- We propose techniques to achieve increased performance with optimized shader design
- We employ three algorithms
  - Cartoon-style non-photorealistic rendering (NPR)
  - Belief propagation (BP) stereo matching
  - Speeded-up robust features (SURF)

Performance Optimization Techniques

Precision Control
- OpenGL ES provides three precision modifiers
  - highp: single precision 32 bit floating point value
  - mediump: half precision floating point value
  - lowp: 10 bit fixed point format, allowing values in the range [-2,2)
- Shader variables use precision hints to provide hints to the compiler on how the variable is used in an application
- lowp is useful in representing normalized color value between 0.0 to 1.0 range
- Choosing lower precision increases performance, but may introduce overflow

Loop Unrolling
- To process a loop shader need extra instructions in increment and comparison operations
- Eliminating loop by either an optimized unrolling or using vectors results in lower instruction count and help achieve higher performance
- When a loop cannot be unrolled, it is preferred that the loop have a constant limit to avoid dynamic branching

Load Sharing
- Dependent texture read
  - A dependent texture read occurs when a fragment shader computes texture coordinate rather than using the unmodified texture coordinate passed into the shader
  - Causes stall of the process until the texture is read
- To avoid dependent texture read, pre-compute texture coordinate in vertex shader and pass to the fragment shader
- Total of 16 texture coordinates can be passed from vertex to fragment shader

Texture Compression
- Reducing the memory is at the cost of final rendered quality
- Computer vision algorithms such as feature detection and edge detection need not consider the final rendering quality
- Texture compression provides the best balance between memory saving and quality
- OpenGL ES supports POWERVR Texture Compression (PVRTC) format
- Two levels of PVRTC, 8:1 and 16:1 compression

<table>
<thead>
<tr>
<th>Optimization Step</th>
<th>Instruction Count</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>531</td>
<td>537.63</td>
</tr>
<tr>
<td>Loop Unroll</td>
<td>181</td>
<td>105.93</td>
</tr>
<tr>
<td>Load Sharing</td>
<td>150</td>
<td>90.25</td>
</tr>
<tr>
<td>Precision Control</td>
<td>69</td>
<td>48.90</td>
</tr>
</tbody>
</table>

Fragment shader optimization example (on POWERVR SGX540)

Characteristics of Mobile GPU

Memory Transfer Bandwidth
- Memory is shared between CPU and GPU
- Texture need to be wrapped to the graphics core and cannot be directly accessed as CPU arrays
- Algorithms with high floating point intensity can hide the memory transfer latency by scheduling parallel operations
- Bottleneck for algorithms involving large memory buffers and low floating point intensity

Floating Point vs. Fixed Point Implementation
- Mobile GPU is designed such that more transistors are allocated to data processing
- Well suited to algorithms with high floating point intensity
- Embedded CPU either lacks a Floating Point Unit (FPU) or relies on software libraries
- CPU vs. GPU (ARM CORTEX A8 CPU and SGX 540 GPU)
  - SGX 540 has fast vectored floats compared to the VPLite hardware acceleration of CORTEX A8
  - ARM Neon hardware is not fully optimized for floating point operations

<table>
<thead>
<tr>
<th>Resolution</th>
<th>CPU (ARM CORTEX A8)</th>
<th>GPU (POWERVR SGX 540)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Floating point</td>
<td>Fixed point</td>
</tr>
<tr>
<td>800 x 480</td>
<td>3074.8</td>
<td>207.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>107.8</td>
</tr>
</tbody>
</table>

Shader Instruction vs. Rendering Cycles
- On a mobile phone GPU, the number of instruction slots for a vertex and fragment shader is limited
- Packing multiple rendering cycles into a single fragment shader increases the instruction count
- Increasing the number of rendering cycles reduces the parallel fraction
- The best solution depends on the needs of the application

Experimental Result (on Samsung Galaxy S with CORTEX A8 CPU and SGX540 GPU)

Cartoon-Style NPR
- Multi-layer bilateral abstraction
- Edge detection

Stereo Matching
- Hierarchical belief propagation (BP)

SURF (Speeded-Up Robust Feature)
- Subjective result on different platforms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Parameters</th>
<th>On CPU</th>
<th>On GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPR</td>
<td>800x480, 5x5 mask</td>
<td>1545.7</td>
<td>242.7</td>
<td>6.37x</td>
</tr>
<tr>
<td>BP Stereo Matching</td>
<td>384x288, 4 iterations</td>
<td>6976</td>
<td>1086</td>
<td>6.42x</td>
</tr>
<tr>
<td></td>
<td>384x288, 8 iterations</td>
<td>12161</td>
<td>2083</td>
<td>5.84x</td>
</tr>
<tr>
<td></td>
<td>384x288, 12 iterations</td>
<td>17413</td>
<td>3125</td>
<td>5.57x</td>
</tr>
<tr>
<td>SURF</td>
<td>800x480</td>
<td>1703</td>
<td>943</td>
<td>1.81x</td>
</tr>
</tbody>
</table>

Execution time comparison (in milliseconds)

Performance Optimization Techniques

Precision Control
- OpenGL ES provides three precision modifiers
  - highp: single precision 32 bit floating point value
  - mediump: half precision floating point value
  - lowp: 10 bit fixed point format, allowing values in the range [-2,2)
- Shader variables use precision hints to provide hints to the compiler on how the variable is used in an application
- lowp is useful in representing normalized color value between 0.0 to 1.0 range
- Choosing lower precision increases performance, but may introduce overflow

Loop Unrolling
- To process a loop shader need extra instructions in increment and comparison operations
- Eliminating loop by either an optimized unrolling or using vectors results in lower instruction count and help achieve higher performance
- When a loop cannot be unrolled, it is preferred that the loop have a constant limit to avoid dynamic branching

Load Sharing
- Dependent texture read
  - A dependent texture read occurs when a fragment shader computes texture coordinate rather than using the unmodified texture coordinate passed into the shader
  - Causes stall of the process until the texture is read
- To avoid dependent texture read, pre-compute texture coordinate in vertex shader and pass to the fragment shader
- Total of 16 texture coordinates can be passed from vertex to fragment shader

Texture Compression
- Reducing the memory is at the cost of final rendered quality
- Computer vision algorithms such as feature detection and edge detection need not consider the final rendering quality
- Texture compression provides the best balance between memory saving and quality
- OpenGL ES supports POWERVR Texture Compression (PVRTC) format
- Two levels of PVRTC, 8:1 and 16:1 compression

<table>
<thead>
<tr>
<th>Optimization Step</th>
<th>Instruction Count</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>531</td>
<td>537.63</td>
</tr>
<tr>
<td>Loop Unroll</td>
<td>181</td>
<td>105.93</td>
</tr>
<tr>
<td>Load Sharing</td>
<td>150</td>
<td>90.25</td>
</tr>
<tr>
<td>Precision Control</td>
<td>69</td>
<td>48.90</td>
</tr>
</tbody>
</table>

Fragment shader optimization example (on POWERVR SGX540)