Improve Image Processing Using GPU Computing on Mali™

By ArcSoft
Senior R&D manager Liu Jie
GPU Computing Based on Mali™

- **GPU Computing Introduction**
- **Image Filter Optimization by OpenGL ES**
- **JPEG Decoder Optimization by OpenCL**
- **Conclusion**
Part 1. GPU Computing Introduction

Background:

- Increasing resolution of devices’ display size, images and video contents calls for higher image processing capability. Also, CPU is no longer capable for some complex algorithm.

- Mobile devices’ and embedded systems’ GPUs are becoming unprecedented powerful. And GPU’s own architecture characteristics make it especially powerful for some algorithm.

- Better load balancing across system resources and higher energy-efficiency.
Part 1. GPU Computing Introduction

Challenges:

- Need to well-balance CPU and GPU’s workload. CPU is strong at task parallelism and GPU is strong at data parallelism.

- High efficient data pipelining between CPU and GPU is also one of the key points for designing good architecture.
Part 1. GPU Computing Introduction

Advantages of Using Mali™:

• **128 bit vectorization**, adapt to video and image processing algorithms.

• **Unified memory architecture**: shared memory system between CPU and GPU, no data copy needed.

• **Support full profile OpenCL**: same feature set as all OpenCL in desktops etc are supported in Mali™, good for portability and migration of existing s/w.

• **Architecture designed from the ground up for GPU Compute**: nearly all mathematical functions are accelerated in hardware.

• **Proven in silicon**, shipping in devices since Nexus 10 tablet in 2012, mature and reliable architecture and drivers.
Part 1. GPU Computing Introduction

Use Cases:

- Mobile: for digital camera side, HDR, night shot, anti-shaking, post-processing and so on, higher performance and lower power consumption.

- IPTV Set-top Box/Smart TV/HD Video Player: gesture recognition based on camera, video pre/post-processing and so on. Replacing specialized Post-processing IC with GPU computing based software algorithm in high resolution video playback (1080p, 4k) is cost saving and at mean time, more swift and flexible.
Part 2. Image Filter Optimization by GL/SL

OpenGL Introduction:

The Importance of OpenGL:

Regarding performance, power consumption and user experience consideration, lots of image processing effects are not suitable to be processed using CPU. Hence using GPU to bring out highly efficient parallel render effects becomes important.

Use Cases:

- Using convolution to achieve sharpening effect
- Rendering YUV420sp data to achieve high efficiency
- Image filter effects by OpenGL
Using Convolution to achieve sharpening effect:

Many filters used in digital image processing are developed based on Convolution. E.g., sharpening effect, and its process is as follows:

1. Aligning center element of Convolution kernel to current pixel to be processed, so other elements in Convolution kernel are also related to pixels respectively.
2. Multiplying each element in Convolution kernel by the color vector of corresponding pixel.
3. Finally, make all products to be added with weight. So, the sharpened color vector of the current pixel is obtained.
Part 2. Image Filter Optimization by GL/SL

Using Convolution to achieve sharpening effect:

**Advantages:**

1. Reducing CPU and memory occupancy for loading and binding textures.
2. By using GPU instead of CPU, we can effectively improve rendering performance, provide better user experience.
Part 2. Image Filter Optimization by GL/SL

Image Filter Effects by GL/SL:

Use case

- Fully understand original CPU effect algorithm, reconstruct and interpret into using shading language.

Procedure

2. Using multiply calculation instead of division as much as possible.
3. Reducing massive calculations inside shader, try doing them outside for performance consideration.
4. Avoid define unused variables and const values.
5. Avoid using if/else as much as possible.
Part 2. Image Filter Optimization by GL/SL

Classical-Nostalgia:
Part 2. Image Filter Optimization by GL/SL

Classical-Nostalgia:
Part 2. Image Filter Optimization by GL/SL

Image Filter Effects by GL/SL

Performance Comparison:

1. After doing optimization by GL/SL, rendering time cost is about 20ms, for 1920x1080 camera preview image adding classical-Nostalgia effect.

2. Using pure CPU to do same effect, rendering time cost is about 80ms.
Rendering YUV420sp data to achieve high efficiency:

**Use case**

- Receiving camera preview data and using CPU to do face beautify and then utilizing OpenGLES to do rendering.

**Procedure**

1. Using Y-luminance data, and UV-chrominance data to generate texture respectively, and then, binding them.
2. Sample Y texture, UV texture respectively in fragment shader.
3. Transform sampled YUV data to RGB data by corresponding conversion formula.
Part 2. Image Filter Optimization by GL/SL

Rendering YUV420sp data to achieve high efficiency:

Performance Comparison:

Using CPU to do rendering, time cost on each frame (1920x1080): **30ms**;
Using OpenGLES, time cost: **10ms**;
For our demo procedure (face beautify), FPS has an uplift from **16 to 26**!
Part 3. JPEG Decoder Optimization by OpenCL

Main Jobs:
Utilizing Open CL reconstructing inverse quantization and IDCT modules to Optimize decoding performance.

Background:
Platform: Samsung Exynos 5 Dual Arndale Board, Ubuntu 12.04
GPU: ARM Mali™-T604
Software: libArcJpeg, Open CL 1.1 (driver only support to this version at that time)
Part 3. JPEG Decoder Optimization by OpenCL

Optimization Keys!

1. Well designed threads model:

- Original algorithm is to do inverse quantization and IDCT right after each MCU being decoded.

- OpenCL based algorithm is to do Inverse quantization and IDCT for all after all MCUs in one frame being decoded. The optimized architecture utilize OpenCL. Let each kernel process a single MCB in MCU to avoid data reliance between kernels. Also, because GPU and CPU memory space of Mali™ can do map and unmap through OpenCL interfaces, this provide us great convenience in using buffer.
Part 3. JPEG Decoder Optimization by OpenCL

**Optimization Keys!:**

1. **Well designed threads model (YUV422):**
   
   **Optimized Kernel Model:**

   ![Kernel Model Diagram]

   - MCU0
     - Y0
     - Y1
     - U0
     - V0
   - kernel0
   - kernel1
   - kernel2
   - kernel3

   - All MCUs
   - Kernel 0 ~ 3
   - All Kernels
Part 3. JPEG Decoder Optimization by OpenCL

**Optimization Keys!:**

2. **Vectorization:**

\[
\begin{align*}
A_0 &+ B_0 &= C_0 \\
A_1 &+ B_1 &= C_1 \\
A_2 &+ B_2 &= C_2 \\
A_3 &+ B_3 &= C_3 \\
A_4 &+ B_4 &= C_4 \\
A_5 &+ B_5 &= C_5 \\
A_6 &+ B_6 &= C_6 \\
A_7 &+ B_7 &= C_7 \\
\end{align*}
\]
Part 3. JPEG Decoder Optimization by OpenCL

Optimization Keys!

3. **Optimizing Data Input:** Some adjusting in Zig-Zag index mapping table in utilizing reordering.
Optimization Keys!

4. Reducing Accessing Memory:
Better balanced loads in mapping tasks and direct calculation can fully utilize the potential of GPU’s strong calculation capability.
Part 3. JPEG Decoder Optimization by OpenCL

**Before Optimization:**
A “Range_Limit” map was constructed, to be referenced while data was range limited.

![Data before range limited](image1)

![Data after range limited](image2)
Part 3. JPEG Decoder Optimization by OpenCL

After Optimization:
Get rid of "Range Limit" map. Take Bit 7 to Bit 9 out of that data, to decide its limited value.
In 8x8 block, it reduced **64 times** memory read.

Key (Bit 7 ~ Bit 9)

- Key < 0x2: Take the low 8 bits value
- 0x2 ≤ Key < 0x5: Set to 0xff
- Key ≥ 0x5: Set to 0
Part 3. JPEG Decoder Optimization by OpenCL

Optimization Keys:

5. Reducing Using Switch:

Avoiding frequently using "if" and "switch" in our kernels, and in some simple case replacing them with build-in functions such as "select", which can be benefit in optimizing performance.
Part 3. JPEG Decoder Optimization by OpenCL

Performance Comparison:

Compared with ArcSoft Neon based Jpeg decoder, performance in decoding 4000x3000 resolution images uplifted as 25%.

Compared with OpenCL based open-source project Jpeg-OpenCL, the efficiency of IDCT uplifted as 15 times.
Thoughts or Challenges:

**Calling interfaces cost time:**
Most interfaces related to memory such as clCreateBuffer(), clEnqueueMapBuffer() cost too much time, which greatly counteract the benefits brought by using OpenCL.

**Reasonably arranging buffer allocation:**
Better creating warm up kernel to allocate all necessary buffers ahead each of them is really used.

**OpenCL version:**
Only support to version 1.1, which brought us troubles in both debugging and further optimizing performance.
Part 4. Conclusion

**GPU Computing Benefits:**

- GPU Computing is a must, an unstoppable trend, and already have brought benefits in bringing better software.
- Higher performance in imaging, computer vision, computational photography and multimedia codecs.

**ArcSoft Efforts:**

- H.265 decoder, Face beautifier, NightHawk (Enhancement shot under low light condition), Gesture.

**ArcSoft Plan to do:**

- ArcSoft has lots of CPU based algorithm and already deliver them to our customers and made big success, we need port more CPU algorithms to GPU.