Vol.2013-SLDM-163 No.34
2013/11/28

Android における GPU を用いたリアルタイム視覚効果処理アプリケーションの実装

程明宇† 北嶋 暁†

本研究では，Android における GPU を用いたリアルタイムビデオ処理を提案する．Android におけるリアルタイムビデオ処理を実行するための枠組みとして，GPU の使用を想定した OpenGL ES 2.0 を活用する．具体的な高速化法として，shading language のデータ精度によって，低精度データに対して，一般的に用いられる頂点シェーダではなく，フラグメントシェーダを用いる．実装においては，Android の HAL 層のグラフィックスドライバを置き換えて，上記の処理を目的プロセッサに特有の機能を活用して実現した．評価として，提案手法を用いて，様々な視覚効果処理を行うアプリケーションを実装した．評価実験の結果，提案手法では GPU を用いない場合より処理速度が 50%近く向上することが示された．

Real-Time Video Processing Programming with GPU on Android Platform

MINGYU CHENG† AKIRA KITAJIMA†

In this paper, we propose a real-time video processing using the GPU on Android platform. As a framework for executing a real-time video processing programming on Android platform, OpenGL ES2.0 is used to accelerate video processing in the proposed method. For the data accuracy of the shading language, the fragment shader instead of the vertex shader that commonly used for low-precision data is used for faster execution. A new driver into HAL layer instead of official driver is implemented for evaluation purpose. As an experimental result, it is shown that video processing execution time using the proposed method is around 50% faster than the case of video processing without GPU.

1. はじめに

近年の携帯端末の性能向上に伴い，様々な視覚効果を加えることができるビデオ撮影処理アプリケーションの注目が高まっている．しかしながら，現在市販されているビデオ撮影アプリケーションは，撮影後にビデオの視覚効果を追加するという方法しかとられていない．これに対し，撮影中の映像に対してリアルタイムに視覚効果を加えることができれば，アートやエンターテインメントなどの分野を中心に，用途が広がると期待される．

リアルタイムビデオ処理を高速に実行するための枠組みとして，GPU の使用を想定した OpenGL ES（OpenGL for Embedded Systems）2.0 が挙げられる．OpenGL ES は，3D コンピューターグラフィックス用 API である OpenGL のサブセットであり[1]，2007 年に公開され，1.1 との後方互換性を持つ．OpenGL ES はハードウェアアクセラレーションによるグラフィックレンダリングのための業界標準となりつつある．OpenGL ES の 1.0 と 1.1 は，すべてのバージョンの Android において Java のライブラリとして公開されている．OpenGL ES 2.0 は Android2.2 以降からサポートされている．つまり，まだかなりの市場シェアを占めているエミュレータや古いデバイスのほとんどは，OpenGL ES 2.0 をサポートしていない．それゆえ，Android2.2 より前のバージョンでは，OpenGL ES 2.0 で導入された，いくつかの重要な機能を用いることができない． また，別の問題として，Android 標準のグラフィックスドライバは汎用的に作られているため，プロセッサ固有の機能を生かしきれていないことが挙げられる．

本研究では GPU を用いたリアルタイムビデオ処理を提案する．具体的には，OpenGL ES 2.0 で用いる shading language のデータ精度によって，低精度データに対して，一般的に用いられる頂点シェーダではなく，フラグメントシェーダを用いる．頂点シェーダよりもフラグメントシェーダの方が計算処理が単純であるため，フラグメントシェーダを適宜用いることで高速化が見込める．

提案手法の有効性を確かめるため，Android の HAL 層のグラフィックスドライバを，提案手法を実現できるように実装して置き換えた．これにより目的 CPU に固有の機能を活用することで，十分性能を引き出した結果を得ることができる．その上で，様々な視覚効果処理が行えるアプリケーションを作成した．

評価として，提案手法を用いて，様々な視覚効果処理を行うアプリケーションを実装した．評価の結果より，提案手法では GPU を用いない場合より処理速度が 50%近く向上することが示された．

本論文の構成は以下の通りである．第 2 章では提案手法について述べる．第 3 章では本研究で提案手法の実装につ

† 大阪電気通信大学
Osaka Electro-Communication University

いて述べる．第 4 章では本研究では提案したアプリケーションの評価手法について述べる．第 5 章では評価結果の考察について述べる．第 6 章で本稿をまとめる．

2. 提案手法

本研究では GPU を活用したビデオ処理の高速化手法を提案する．具体的には OpenGL ES 2.0 の利用を前程として高速化を行う．

高速化の基本的な原理は次のとおりである．スループットを向上させる為に，OpenGL ES 2.0 では 3 種類の変数型がサポートされている. 即ち lowp, medium, highp である[2]. highp は 32 ビットの浮動小数点変数である．medium は 16 ビットの浮動小数点変数であり，値の範囲は (-65520, 65520) に限られている．Lowp は固定小数点変数であり，値の範囲は (-2, 2) に限られている．OpenGL ES 2.0 の機能の一つに画像の区域に対する演算を行う頂点シェーダとフラグメントシェーダがある．フラグメントシェーダでは，用いるデータの精度を変えることができる．同じ計算内容が同じでも低い精度の方が高速に処理することができる．

提案手法では，次のとおり頂点シェーダとフラグメントシェーダを使い分ける．図 1 に示すように 3x3 の区域を処理する場合，中心点はキーポイントとして高い精度が必要なので，highp 変数型を用いた頂点シェーダで処理する．キーポイントである中心点に隣接する区域は medium 変数型の精度で十分なので，medium 変数型を用いたフラグメントシェーダで処理する．

medium	medium	medium
medium	highp	medium
medium	medium	medium

図 1　区域内の変数型

典型的なデータの使用法として，ピクセルのデータがある．多くのカメラは YUV 色空間を出力している.一方，画像処理には RGB の色空間に変換した方がよい[3]．この際に用いる変換の方法を図 2 に示す．階調の映像処理では輝度（Y）は 0～255 のスケールにマッピングされている．RGB 色空間処理として，すべてのチャンネルで YUV 色空間を変換しなければならない．この際，変換後のデータの精度を高く保った方が良い画像処理効果が得られる．このような場合にフラグメントシェーダが活用できる．

$$R = 1.164(Y - 16) + 1.596(V - 128)$$
$$G = 1.164(Y - 16) – 0.813(V - 128) – 0.391(U - 128)$$
$$B = 1.164(Y - 16) + 2.018(U - 128)$$

図 2 色空間の変換方法

フラグメントシェーダの活動例として，画像処理の際の畳み込み操作がある．この処理に用いるフィルタリングには，フラグメント出力を計算するために近傍のピクセルにアクセスすることが必要である．フィルタリング出力値に影響する近傍画素の値を取得するには，通常，フラグメントシェーダのアドレスを計算する必要がある．一般的な画像処理操作のために必要な頂点の数は，フラグメントの総数よりもはるかに少ない．従来は頂点シェーダのみで処理されていたが，同じ処理をフラグメントシェーダで実現した場合，フラグメントシェーダの方が，処理速度が速く，GPU の利用率も高い．フィルタリングの場合には，簡単化のためにあらかじめ頂点シェーダで近隣テクスチャアドレスを計算する方法を用い，それをもとにフラグメントシェーダを活用することができ，パフォーマンスの増加につながる．

3. 提案手法の実装

提案手法の有効性を確かめるため，3 種類の端末に提案手法を実装した．以下，3.1 節で実装に用いた既存技術について触れる．3.2 節で端末に用いられる目的 CPU について述べる．3.3 節で Java から OpenGL ES を利用する方法を述べる．3.4 節でグラフィックスドライバの実装法について述べる．

3.1 実装に用いた既存技術

(1) **Android プラットフォーム**

Android は，Google 社が中心になって組織された Open Handset Alliance[4]によって開発されたプラットフォームであり，スマートフォンやタブレット PC などの情報端末を主なターゲットとしている．2007 年の発表以来注目を集めており，スマートフォン用の OS としては，日本とアメリカでトップシェアである．非常に汎用性が高く，様々なデバイスに移植可能であることが魅力の 1 つだが，最も注目すべきは，誰でも無償で利用することができるオープンソースな OS であるということであり，これが Android OS の普及における大きな要因の 1 つであると考えられる．

Android のアーキテクチャ図を 3 に示す．

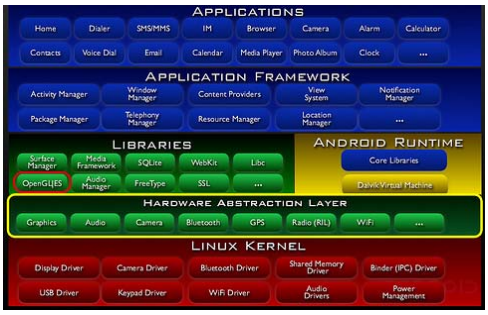

図 3　Android のアーキテクチャ

(2) **OpenGL ES2.0**

OpenGL ES（OpenGL for Embedded Systems）は主に携帯電話などの組み込みシステムで広く使用される．iOS や Android，Symbian OS などの携帯端末向けオペレーティングシステムで採用されているほか，プレイステーション 3 やニンテンドー3DS にも採用されており，近年ではゲーム開発での使用が注目される．OpenGL と同様にクロノス・グループ[5]によって管理されている．

(3) **JNI（Java Native Interface）**

JNI は, Java で記述されたプログラムと他の言語(C, C++)でかかれたコードを連携するためのインターフェースである．メリットとして既存のライブラリの活用ができることや，パフォーマンスの向上が挙げられる．これは，C 言語で各プロセッサに最適化したライブラリを Java から利用できるからである．高速化のためには，理論上的には JNI を用いるより JIT（Just-In-Time Compiler）を利用したほうがよい．それは，１つの中間コードで複数のプロセッサ（ARM, MIPS, Intel など）に対応できるためである．JNI の利用法として，計算量の多いプログラムを部分的ネイディブコードに置き換えて高速化する場合の他に，ハードウェアアクセス場合がある．特に，Java で作成したユーザーアプリケーションからでは直接ハードウェアにアクセスすることができないため，ハードウェアを利用するには JNI に利用は必要である．図 4 に JNI を用いたハードウェアアクセスの流れを示す．

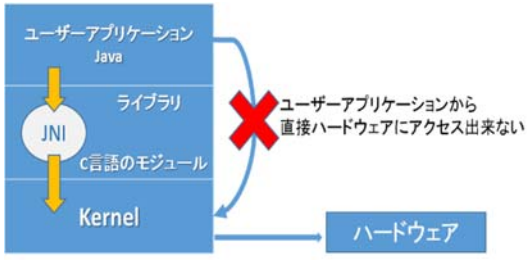

図 4　JNI を用いたハードウェアアクセスの流れ

3.2 目的 CPU

本手法では，目的の携帯端末上のプロセッサとして，Tegra 系のプロセッサを用いた．実装した機器の仕様を表 1 に示す．

機器名	CPU	CPU の周波数	CPU のコア数	GPU のコア数
AT703	Tegra 4	1.9 GHz	4 + 1	72
TF700T	Tegra 3	1.7 GHz	4 + 1	12
TF101	Tegra 2	1.2 GHz	2	8

表 1　実装した機器の仕様一覧

Android では全て機器に向けた汎用的なドライバが搭載されている．頂点シェーダの数は増えて来たが，HAL 層のプロファイルには対応する内容は変わらなため，そのままでは Tegra 系プロセッサの機能を充分に活用していない．

3.3 Java からの OpenGL ES の利用

Android アプリケーションは通常 Java で記述されるが，Java では直接 OpenGL ES を用いることができない. 本手法では，JNI を用いて OpenGL ES を用いている．この場合の OpenGL ES レンダリング処理の構造を図 5 に示す．

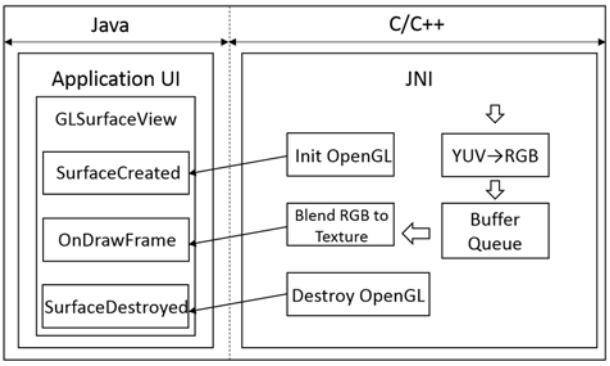

図 5　OpenGL ES レンダリング処理の構造

高速化のために，提案アプリケーションの UI 層は JNI 層を用いる[2][6]．ネイティブウィンドウなどの Java 表面オブジェクトで関連付けを行っている．ネイティブコードを介してそれと通信する．アプリケーション層では，UI を表示するサーフェスオブジェクトを作成する．そして JNI 層に，このサーフェスを渡す．これにより JNI 層からネイティブウィンドウを取得することができる．

JNI 層の RGB 色空間からアプリケーションの UI 層に変換する為に，本手法では GLSurfaceView クラスを用いた．GLSurfaceView クラスには 3 つイベントがあり，即ち SurfaceCreated と SurfaceChanged と SurfaceDestroyed である．OpenGL ES の初期化と破棄の際に SurfaceCreated と

SurfaceDestroyed が呼び出される．

3.4 グラフィックスドライバの実装

GPU を充分に利用する為には，HAL 層のプロファイルの内容を変更し置き換えることが必要である．例えば，CPU と GPU の優先度を変える為に/system/lib/egl 中に置かれた egl.cfg ファイルの中身「0 0 android」を削除する[7]．同じフォルダの libGLES_android.so ファイルを削除したら，処理性能が向上するが，システムの安定性は弱くなるので，残している．Surface composition 処理をハードウェアで高速化する為に，Copybit HAL ファイルの内容を変更した．Copybit HAL ファイルを変更する場合は，gralloc と libagl と surfacefinger ファイルの内容も変更する．また，Tegra 系 CPU を対象として[8][9]，Tegra 系 CPU 仕様に基づく最適化コードを組み込んでいる．GPU で画像を処理する前に Gralloc モジュールをロードすることにより，フレームバッファが使えるようになる．

4. 評価

4.1 評価内容

FPS（1 秒に処理されたフレーム数）を用いて画像処理性能を評価する．評価項目は次のとおりである．

- 用いる視覚効果
- 画像の解像度
- Tegra 系 CPU 別

比較の対象は以下のとおりである．

- 提案手法 (GPU で処理した場合)
- 汎用ドライバ (Android に搭載されている公式 GPU ドライバを利用した場合)
- GPU 無 (GPU を利用しない場合)

視覚効果の例として，以下の処理 A, B,C の三つを用いる．処理 A の内容は，図 6 に示すように元画像に 3 つの視覚効果を加えることである．具体的には，鉛筆の視覚効果と 4 つの画像を 1 つの画像に並べる処理とネガフィルムの枠である．処理 B の内容は，陰画とラフガラスの視覚効果とコルクボードである．図 7 に示す．処理 C の内容は，熱画像の視覚効果と画面回転と枠組である．図 8 に示す．

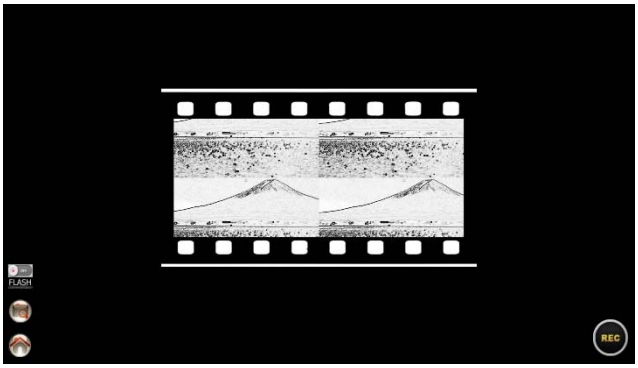

図 6　処理 A (フィルタ：鉛筆　効果：4 つの画像を 1 つに入れ込む　マスク：ネガフィルム)

図 7　処理 B (フィルタ：陰画　効果：コルクボード　マスク：ラフガラス)

図 8　処理 C (フィルタ：熱画像　効果：回転　マスク：枠組)

4.2 評価結果

Tegra4 の場合の，解像度 1280x720 における実験結果を表 2 に示す．提案手法の画像処理性能は GPU を用いない場合より 47.8%向上し，Android の公式 GPU ドライバを用いた場合より 25.58%向上した．

類別	最大値	最小値	平均値
提案手法	22.6	20.6	21.52
汎用 GPU ドライバ	19.24	16.13	17.12
GPU を用いない場合	15.07	13.34	14.56

表 2　解像度 1280x720 での処理 A の結果 (fps)

Android に様々タスクがあり，優先度によって，システムは自動的に優先度高いタスクを執行する．その原因で，Android の公式 GPU ドライバを用いた場合には，処理の安定性は良くない．ある時に強い変動がある．図 9 に示す．

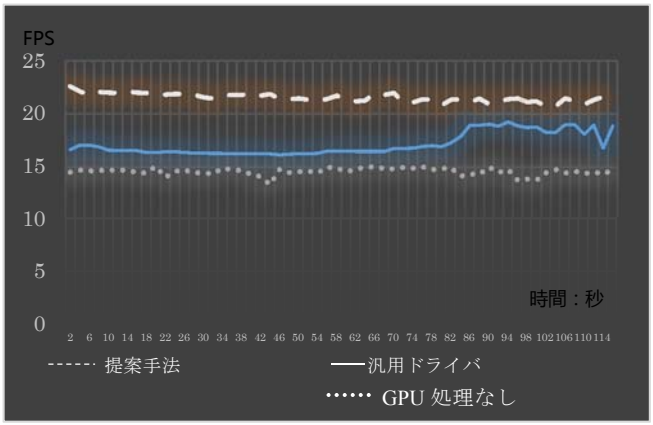

図 9　FPS の変動範囲

1280x720 の解像度でのプロセッサ別での処理の結果を表 3 に示す.

プロセッサ	最大値	最小値	平均値
Tegra4	22.6	20.6	21.52
Tegra3	14.84	14.14	14.57
Tegra2	11.87	11.36	11.66

表 3 プロセッサ別の処理結果 (fps)

解像度の違いについての評価は，処理 A の他に、処理 B、処理 C の 3 通りの結果を示す．視覚効果によって，アルゴリズムは異なるので，頂点シェーダとフラグメントシェーダを用いた数は異なり、結果に差が出る．処理 B の実験結果を表 4 に示す．処理 C の実験結果を表 5 に示す.

類別	最大値	最小値	平均値
提案手法	24.99	23.39	24.50
汎用ドライバ	24.56	20.82	22.99
GPU 処理なし	15.77	14.95	15.77

表 4 処理 B の結果 (fps)

類別	最大値	最小値	平均値
提案手法	26.13	24.93	25.20
汎用ドライバ	23.3	20.36	22.77
GPU 処理なし	17.98	15.35	16.44

表 5　処理 C の結果 (fps)

解像度の違いについての評価は，1280x720 および 768x432，176x144 の 3 通りで行った．768x432 の解像度での処理結果を表 6 に示す．176x144 の解像度での処理結果を表 7 に示す．解像度を下げると，処理性能は良くなっていく．しかしながら，解像度を下げると，CPU と GPU 性能による影響は弱くなっていく．

類別	最大値	最小値	平均値
提案手法	26.79	24.98	25.27
汎用ドライバ	25.65	19.66	21.89
GPU 処理なし	20.6	18.31	19.35

表 6　解像度 768x432 での処理結果 (fps)

類別	最大値	最小値	平均値
提案手法	30.94	29	29.99
汎用ドライバ	30.6	29.12	29.92
GPU 処理なし	30.92	29.04	29.92

表 7　解像度 176x144 での処理結果 (fps)

視覚効果がない場合は，CPU や GPU の性能の違いによる影響は少ない．視覚効果を入れないときの処理結果を表 8 に示す.

類別	最大値	最小値	平均値
提案手法	29.94	26.59	27.98
汎用ドライバ	27.99	24.48	26.27
GPU 処理なし	24.31	23.08	23.69

表 8　視覚効果を入れない場合の処理結果 (fps)

5. 考察

フレームレートについて，アニメ作品や映画は伝統的に 24fps(フレーム/秒)で作られており，これを基準値とすると，提案手法でのフレームレートの最小値は 20.6fps となり，やや下まわる．しかし，目視で確認した範囲では，特に不自然さは感じられず，この程度であれば許容範囲であると考えられる.

GPU のコア数が増えると，画像処理性能は良くなって行く．しかし，増えたコア数と性能の向上範囲の関係は正比例ではない.

6. おわりに

本稿では，Android におけるリアルタイムビデオ処理を実行するための枠組みとして，GPU を使用した OpenGL ES 2.0 を活用する方法を提案した．また，提案手法を用いて，視覚効果処理アプリケーションを実装した．実験結果により，提案手法では GPU を用いない場合より 50%近く向上することが示された． 今後の課題は GPU のコアの利用率の向上や低消費電力化である.

参考文献

1) A.Munshi and J.Leech, OpenGL ES Common/Common-Lite Profile Specification, version 1.1.12 (Full Specification), Khronos Group, 2008
2) Nitin Singhai, In Kyu Park, Sungdae Cho: Implementation And Optimization Of Image Processing Algorithms On Handle GPU,

Procedings of 2010 IEEE 17th International Conference on Image Processing, September 26-29, 2010, Hong Kong
3) Khairul Muzzammil bin Saipullah and Ammar ANuar and Nurul Atiqah binti Ismail and Yewguan Soo, “Real-Time Video Processing Use Native Programming on Andriod Platform,” IEEE 8th International Colloquium on Singel Processing and its Applications, 2012
4) OpenHansetAlliance, http://www.openhandsetalliance.com
5) http://www.khronos.org/opengles/
6) Zhiguang Zhang, Changqing Yin:Research on Video Rending on Android, Wireless Communications, Networking and Mobile computing (WiCOM), 2012 8th International Conference on
7) http://www.xda-developers.com/
8) NVIDIA Corporation: Whitepaper Variable SMP(4-PLUS-1™)-A Multi-Core CPU Architecture for Low Power and High Performance, 2011
9) NVIDIA Corporation: Whitepaper Bringing High-End Graphics to Handheld Devices, 2011