Vol.2013-ARC-207 No.12
Vol.2013-HPC-142 No.12
2013/12/16

プロファイル情報を用いたAndroid 2D描画ライブラリSKIAのOSCARコンパイラによる並列化

後藤 隆志[1,a)]　武藤 康平[1]　山本 英雄[1]　平野 智大[1]　見神 広紀[1]　木村 啓二[1]　笠原 博徳[1]

概要：本論文では，スマートフォンやタブレット等で広く用いられるAndroidにおいて，従来マルチコアプロセッサ上での並列化が困難で，その高速化が望まれていた2D描画ライブラリSkiaを，OSCAR自動並列化コンパイラにより，プロファイラ情報に基づいた自動並列化を行う手法を開発したのでその方法を説明する．OSCARコンパイラはParallelizable Cにより記述された逐次プログラムから様々な粒度で並列化解析を行い，自動的に並列化Cソースを出力する．しかし，SkiaはAndroid内のライブラリであり，利用する描画命令ルーチンにより制御フローが大きく変化するため，最適な並列化解析を行うことが困難である．そこで，本論文ではSkiaのような制御フローがコンパイル時に特定できないプログラムに対し，Oprofileを用いて取得したプロファイル結果をOSCARコンパイラにフィードバックすることで，並列化対象を特定の領域に絞り，高い性能向上が得られる手法を提案する．なお，並列化対象領域がParallelizable Cコードでない場合でも，解析結果により実行コストが大きい部分からParallelizable Cに変更し，チューニングを施すことで並列化が可能となる．本手法を，描画ベンチマークとして広く使われている0xbenchをNVIDIA Tegra3チップ(ARM Cortex-A9 4コア)を搭載したNexus7上で評価を行った．並列化Skiaの実行においては，並列化部分の速度向上を正確に評価するため，Androidをcore0に割り当て，残り3コアをSkiaが利用できる形とした．評価の結果として，DrawRectで従来の1.91倍である43.57[fps]，DrawArcで1.32倍の50.98[fps]，DrawCircle2では1.5倍の50.77[fps]といずれも性能向上結果が得られた．

1. はじめに

近年，マルチコアプロセッサはデスクトップコンピュータや高性能端末，組み込みシステムまでと広く用いられるようになってきた[1]．特に，スマートフォンやタブレットなどの携帯端末は急速に普及し，求められる性能も高くなっている事から，NVIDIA Tegra3[2]やQualcomm Snapdragon[3], Samsung Exynoso[4]などのマルチコアが広く用いられている．しかし，これらのマルチコアを十分に生かし，より高い性能を得るためには，ソフトウェアが並列化されていることが必要である．現在では，OpenMPやMPI[5]などのAPIを手動で入れる並列化手法が一般的であるが，ソフトウェアが複雑であるほど並列化に必要な時間は増え，開発コストが高くなってしまう．そこで，コンパイラを用いて自動的にプログラム全体から多くの並列性を抽出し，並列化を行うことで，効率的にマルチコアの性能を引き出すことが可能となる．このような自動並列化コンパイラの一例としてOSCAR compiler[6]を開発してきた．

2Dレンダリングは，携帯端末の中でも重要な要素の1つであり，高い描画処理性能によって高速に画面表示を行うことは，ユーザビリティの向上に大きく貢献する事が期待できる．携帯端末で用いられる2Dレンダリングエンジンの例としては，skia[7], Quartz[8], cairo[9]が挙げられ，これらを自動並列化によってマルチコアを利用し，高速化することは非常に有用であると言える．しかし，上記レンダリングエンジンは様々な描画命令に対応するライブラリとして作成されており，各命令毎にライブラリ内で処理する関数を含めた制御フローが全く異なるため，並列性の解析について考えると，処理フローに沿った最適な解析を行う事が困難である．

本論文では，OSCARコンパイラを用いて，プロファイル情報を元に並列化解析を行う手法について提案する．プロファイルのツールとしてはOprofileを利用し，プロファイル結果をOSCARコンパイラにフィードバックすることで，並列化による高い性能向上が期待できる領域で並列化解析を行う．この手法によって，Androidの2Dレンダリングエンジンである Skiaに対し自動並列化を行い，Google

[1] 早稲田大学
Waseda University.
[a)] tgoto@kasahara.cs.waseda.ac.jp

Nexus7 上で性能向上が得られることが確認した．本論文は，第 2 章で Skia の概要について述べ，第 3 章でプロファイル情報を用いた自動並列化について，第 4 章にて提案手法の skia への適応について，第 5 章で性能評価結果について説明する．

2. Skia 2D レンダリングエンジン

本章では，Android で 2D 描画処理を行うレンダリングエンジンである Skia について述べる．

2.1 Skia 概要

Skia はオープンソースのグラフィックライブラリであり，テキストや図形，画像などを描画する 2D レンダリングエンジンである．Skia は Google Chrome や Mozilla Firefox などブラウザに広く採用されている他，Android や Chorome OS などのオペレーティング・システムにおけるレンダリング部分としても用いられており，利用頻度は非常に高い．Android においては，2D レンダリング処理のほぼすべてをこの Skia を通じて行なっている [7]. 具体的には，Android は Java レイヤーのアプリケーションに対する，図形や画像，テキストの表示のための基本的な API(Application Programming Interface) として，android.graphics.Canvas クラスを提供している [10]．ゲームを始め，ブラウザなどの多くのアプリケーションは，この API を用いて描画を行なっている．この Canvas クラスには，drawRect や drawImage など，様々な描画に対応したメソッドが用意されているが，これらはそれぞれ JNI(Java Native interface) を通じて Skia ライブラリを呼び出す [11]．Skia は，JNI を通じて Java アプリケーションから受け取った描画命令を元に，レンダリング処理を実行し，最終的には端末のメモリにあるフレームバッファーに表示画像を転送することで画面への表示を行う．この処理は，特にブラウザやゲームなど多数の描画を行う Android アプリケーションのボトルネックとなっており，Skia の高速化は Android 全体の性能向上へと繋がり，大きな利便性の向上となることが期待できる．

2.1.1 Skia レンダリングパイプライン

Skia のレンダリング処理について説明する．Skia は，入力される描画命令に対して，図 1 で示す形のパイプラインでレンダリング処理を行う [12]．このレンダリングパイプラインは，大きく分けて Path Generation, Rasterization, Shading, (Bit-Level Block Transfer)[12] に分けられる．まず Path Generation では，描画する要素を構成する複数のパス集合へと変換する処理を行う．次の Rasterization フェーズでは，パスの情報からパスが描画する領域を決めてピクセルマスクへと変換する処理を行う．この時，色に関する処理は行わず，描画するピクセル領域とその濃さのみを表すアルファチャンネルのマスクとして出力する．色

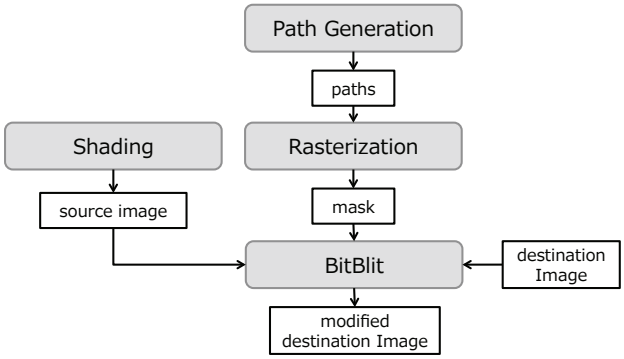

図 1 Skia レンダリングパイプライン

に関する処理を行うのが Shading フェーズである．ここでは，描画領域の各ピクセルに対する色の情報を生成を行う．最後の BitBlit では，Rastererization で生成されたマスクと，Shading で生成された色情報を元に，描画する表示画像を生成し，フレームバッファーへと転送する．

2.2 0xbench について

本節では, 本論文内での評価に用いた Android 向けベンチマークアプリである 0xbench の概要について説明する．0xbench とは, 0xlab が開発しているオープンソースの Android のベンチマークアプリケーションである [13]．計測項目としては，大きく分けて５つに分類され, C library and system call, OpenGL-ES, 2D canvas, Garbage collection in Dalvik, JavaScript engine がある．今回は，Skia を用いるレンダリングのベンチマークとして 2D Canvas 系テストを利用して計測を行った．この 2D Canvas では前節で説明した，android.graphic.Canvas クラスの各描画メソッドを数百回連続して呼び出し，全て描画し終わるまでの時間から FPS 値を算出して表示するベンチマークである． 本論文では，2D canvas 中の DrawRect, DrawArc, DrawCircle2 の 3 つを計測した．各テストの処理は以下のとおりであり，その際の描画例は図 2 で示す．

- DrawRect
  ランダムに四角形の情報 (サイズ，位置，色) を乱数により生成し，これを基に Canvas クラスの drawRect メソッドを 300 回呼び出す
- DrawArc
  画面上に配置された 17 個の扇型や円形の図形に対して，弧の大きさを変えたものをそれぞれ drawArc メソッドで描画し，これを 500 回繰り返すことで，アニメーション表示をする
- DrawCircle2
  drawRect と同様に，色やサイズが異なる 6 つの円をそれぞれ drawCircle を呼び出して描画する事を 300 回繰り返す

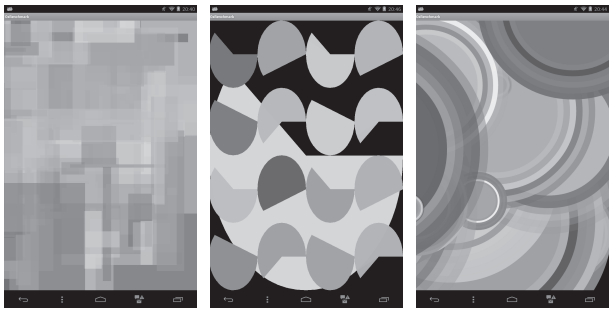

図 2 2D ベンチマークの画面表示例

3. プロファイル情報を用いた自動並列化

本章では，プロファイルツールとして Oprofile を，自動並列化コンパイラとして OSCAR コンパイラを利用し，プロファイル情報を用いた自動並列化手法について説明する．

3.1 OSCAR コンパイラと OSCAR API

OSCAR コンパイラとは，マルチグレイン並列化，キャッシュやローカルメモリ最適化，電力削減制御を可能とする自動並列化コンパイラである [14], [15], [16]．マルチグレイン並列処理とは粗粒度タスクレベル並列性，ループイテレーションレベルの中粒度並列性，ステートメントレベルの近細粒度並列性の 3 つの並列性を効果的に組み合わせた並列処理手法である [6], [17]．OSCAR コンパイラは Parallelizable C や Fortran で記述された逐次ソースプログラムを入力とする．Parallelizable C とは，ポインタの利用の制限等を設けることにより，自動並列化を可能にする記述手法である．OSCAR コンパイラは並列化された C もしくは Fortran 言語ソースファイルを出力する．並列化ソースファイルは OSCAR API を用いて記述される．

OSCAR API は様々な主記憶共有型マルチコアプロセッサ及び，マルチコアシステム上で並列化を実現するために定義された API である．OpenMP のサブセットをベースとして定義されており，スレッド生成や，データのメモリ配置，DMA によるデータ転送，電力制御，アクセラレータ及び同期処理をサポートする．OSCAR コンパイラによって並列化されたプログラムは，OpenMP に対応したコンパイラ，もしくは OSCAR API をランタイム関数に変換する API 解釈系を用いてコンパイルする．例えば，OSCAR API の 1 つである `parallel sections` ディレクティブは，API 解釈系によって `oscar_thread_create` と `oscar_thread_join` の 2 つの関数に変換される．対象のプラットフォームが pthread ライブラリを用いている場合は，`oscar_thread_create` と `oscar_thread_join` は，それぞれ `pthread_create` と `pthread_join` を用いて実装することで，並列化が可能となる．このように，OSCAR コンパイラを用いて並列化されたプログラムは，OSCAR API によって様々なマルチコアプラットフォームで容易に実行することが出来る．

3.2 OProfile

Oprofile は，指定した時間において一定周期でサンプリングを行うことで，アプリケーションからシステム全体のレベルまで，ステートメント毎に負荷が計測できるプロファイリングツールである [18][19]．Oprofile はコールグラフの出力を行えるため，プログラム内の対象関数がどこから呼ばれ実行されているか，処理パスを解析する事も可能である．

本論文では，Oprofile for Tegra (version 0.9.6) を用いてプロファイルを行った [20] ．プロファイル時は，コールグラフを 20 階層，サンプリングを 50000 サイクル周期と設定した．

3.3 プロファイル情報を用いた自動並列化

本論文で対象とする Skia における並列性の解析について考えると，描画対象やアンチエイリアスの有無等により，制御フローや関数が描画命令毎に異なることから，関数間の依存情報に基づく最適な解析を行う事が困難である．このようなプログラムに対し，それぞれの実行命令毎にプログラム全体の並列解析を手動で行うことは非効率である．

そこで，効率的に並列化を行うために，本論文では Oprofile のプロファイル情報を用いた OSCAR コンパイラによる自動並列化について提案する．この手法について，ソースファイルやプロファイル結果，OSCAR コンパイラとの連携について図案化したものを図 3 に示す．HotSpot 解析ツールに Oprofile のプロファイル結果のテキストとデバッグ情報を含んだ対象プログラムのバイナリを渡すことで，テキストから負荷の高い関数を取り出し，バイナリから対象関数のソースファイル情報を求める．ソースファイル群から対象ソースファイルを OSCAR コンパイラの入力ファイルとして渡し，解析対象の入り口として対象関数の情報を与える．これにより，OSCAR コンパイラは並列化可能であれば，並列化されたソースコードを出力する．並列化出来ない場合や，Parallelizable C に準拠していない場合は，解析結果を出力する．解析結果に基づいてプログラムを Parallelizable C へ変更あるいはコンパイラが解析できない添字パターンなどがあれば解析しやすい表現に帰る等のチューニングを行い，再度コンパイラに通すことで，並列化されたソースコードを得ることが出来る．

4. Skia の自動並列化

本論文では，Skia に第 3 章にて述べた手法を適応した．本章では，並列化にあたって取得したプロファイルの結果と，解析結果に基づいて行ったコードチューニングについて説明する．

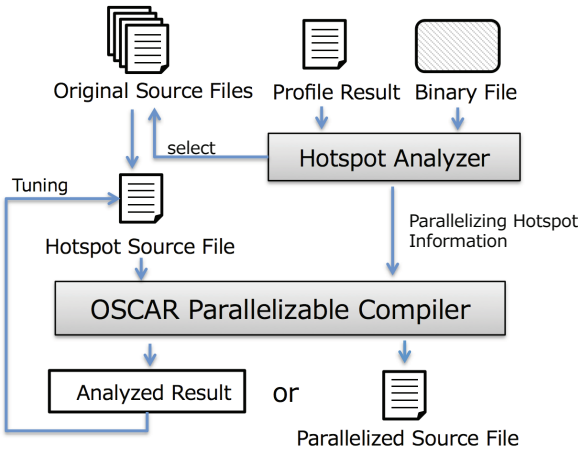

図 3 プロファイル情報に基づく自動並列化手法の処理フロー

4.1 Skia アプリケーション領域のプロファイル解析

Oprofile の Application Profiling を利用し，2.2 節で紹介したベンチマークについてプロファイリングを取得した結果について述べる．

- DrawRect
  処理割合のグラフを図 5(a) にて示す．**SkRGB16_Blitter::blitRect** 関数が処理のほぼすべてを占めていることが分かる．この処理は 2.1 節で示した BitBlit 処理にあたり，正方形に対する Blit 処理を行う関数である．xy 位置情報を起点に，縦幅と横幅でループし，各ピクセルに対して元の値 (destiniation) と値を混ぜあわせて上書きする処理となっている．
- DrawArc
  処理割合のグラフを図 5(b) にて示す．**SkRGB16_Blitter::blitH** 関数が 82%占めていることが分かる．この関数は，基本的には **SkRGB16_Blitter::blitRect** と同じであり，異なるのは横幅でのループする点のみである．
- DrawCircle2
  処理割合のグラフを図 5(c) にて示す．**SkRGB16_Blitter::blitAntiH** 関数が約 78%，続いて **SkRGB16_Blitter::blitRect** 関数が約 9%となっている．後者に関しては DrawRect で述べたとおりである．前者については，基本的には後者と同じ blit 処理であるが，前段階でアンチエイリアシング処理としてスーパーサンプリングされた情報を元に，blit 処理を行う．

4.2 Skia コードチューニング

3.3 節にて述べたツールを用いて各テストベンチマーク実行時のプロファイラ情報を用いて自動並列化を行い，得られた解析結果と，その情報を元に行ったチューニングについて説明する． 今回行った各テストにおいて基本的なチューニング手法はほとんど同じであるため，DrawRect におけるチューニングについて詳細に述べる．今回，解析結果を用いてチューニングを行った関数の Original Source Code と，After Tuning Code を図 4 にて示す．まず，DrawRect におけるプロファイラ情報を用いて OSCAR に通すと，SkRGB16_Blitter::blitRect 関数が Parallelizable C で無いという解析結果が得られる．そこで，対象関数内の C 言語コードを関数化し，while ループを for ループに書き換えて再度 OSCAR に通す．その時，for ループにおいて，device 変数に依存があるという情報が得られるため，device 変数をイテレーション値で固有の値となるように書き換えて OSCAR に通す．これにより，OSCAR は自動並列化を行い，並列化済コードを出力する．BitBlit の処理は，このような height もしくは width でループを行なっている部分がほとんどであり，同じような書き換えによって依存を解消することが可能である．

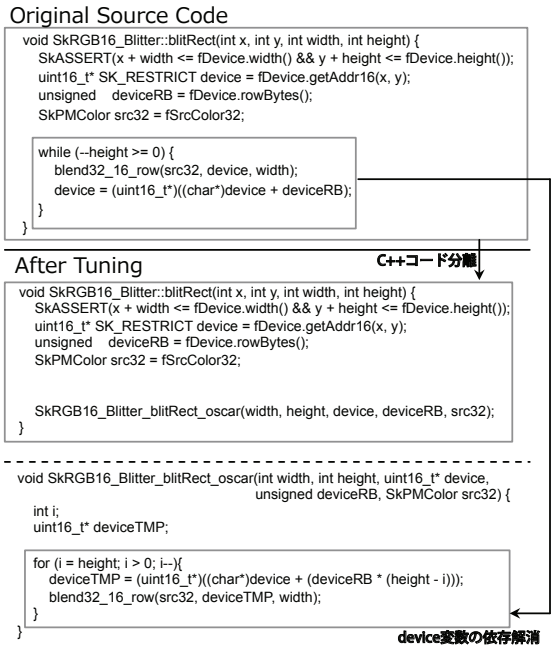

図 4 Skia のコードチューニング例

これらのプロファイル結果から，いずれのテストにおいても，呼ばれる関数自体は処理によって異なるものの，2.1 節における BitBlit のフェーズが明らかにボトルネックとなっていることが分かる．

5. 性能評価

本章では，提案手法の性能評価結果について述べる．なお，本章で述べる “逐次処理” は，従来の Skia における処理であり，“並列処理” は OSCAR コンパイラを用いて並列化を行った Skia における処理である．

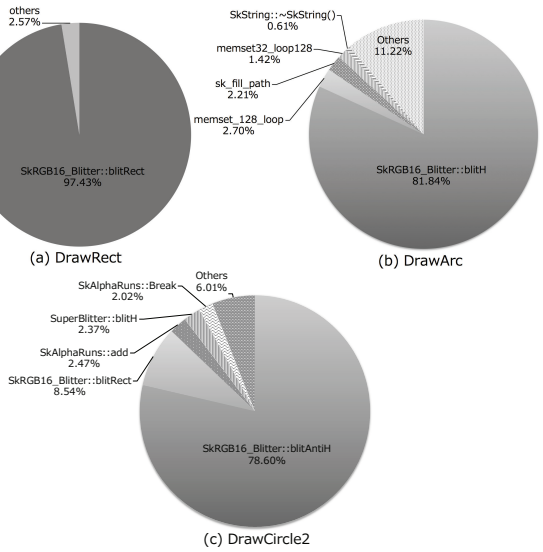

図 5 各ベンチマークテストにおけるアプリケーション領域でのプロファイル結果

表 1 Nexus7 性能一覧

CPU	ARM Cortex-A9 NVIDIA Tegra 3
CPU Frequency	1.2GHz (1.3GHz single-core mode)
CPU core	quad-core
GPU	NVIDIA GeForce ULP
GPU Frequency	416MHz
GPU core	twelve-core
RAM	1GB
Display	1280x800 WXGA pixels

5.1 評価環境

本節では，Skia の性能評価を行う際に用いた端末や設定など，評価環境について述べる．

5.1.1 Nexus7.

本論文では, 評価に用いた携帯端末として, ARM Cortex-A9 4 コアを用いた NVIDIA Tegra3 チップを搭載した 2012 年度版 Nexus7 を用いた．4 コア動作時，各コアは最大 1.2[GHz] で動作する．Nexus7 の詳細については，表 1 に示す [21].

5.1.2 プロセスのコアバインド

並列化した Skia の評価にあたっては，カーネルの init 部分に一部改変を行うことで，Android OS やその他処理を core0 に割り当て，残る 3 コアを並列化されたプログラムが動作するよう処理のスレッド割り当てを行った．これにより，バックグラウンドで処理されるプロセスが Skia の並列処理実行に影響するのを避け，安定してプログラムの効率的実行，及び評価を行う事が可能となる．

5.1.3 スレッドプール

また，今回の並列化対象となっている BitBlit 処理は，各ピクセル毎にビット演算や簡単な整数演算を行うものであり，処理の粒度が非常に小さく，高頻度で実行されるものである．そのため，並列化部分の実行時に毎回スレッド生成を行うと，オーバーヘッドが問題となる．そこで，今回はスレッドプールを用いた並列化の仕組みを導入した．OSCAR コンパイラが生成する並列化済みソースコードは，OSCAR API で記述されたものであり，この並列化済みコードを OSCAR API 標準解釈系を用いることでランタイムライブラリ関数を含んだコードに変換される．この関数において，スレッド生成を行う oscar_thread_create 関数とスレッド処理の終了待ちを行う oscar_thread_join 関数をスレッドプールを用いる形で実装した．各関数のスレッド間での処理フローを図 6 で示す．oscar_thread_create はメインスレッドで実行され，初回のみ pthread でスレッドを生成した後，生成されたスレッドは，処理関数受付と関数実行を繰り返し行うルーチンループに入る．メインスレッドからはスレッドプールに実行関数のポインタが渡される．スレッドプールでは，実行関数のポインタを確認次第，関数を実行し，終了時にその関数ポインタの値を NULL とする．oscar_thread_join では，この関数ポインタが NULL に変更されるのを待つことで join 同期を行う．

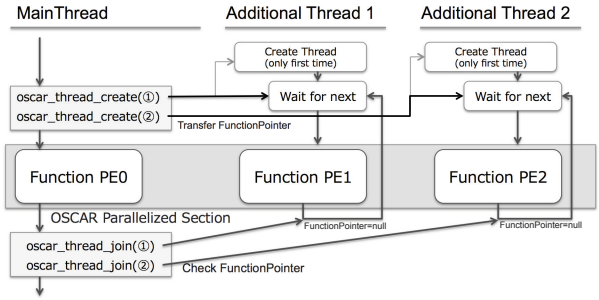

図 6 OSCAR ランタイムライブラリに適応したスレッドプール処理フロー

5.2 ARM プロセッサにおけるクロックサイクル計測手法

ARM Cortex-A9 プロセッサには，Performance Monitor Unit(PMU) が搭載されている [22]．PMU は，各コアの様々な処理イベントの調査が可能となっており，今回はその中のサイクルカウント (CCNT) レジスタを用いてクロック数の計測数を行った．ただし，CCNT レジスタへのユーザーモードでのアクセスは，ユーザイネーブル (USERNE) レジスタのビット値が 1 である必要があり，USEREN レジスタは特権モードでしかアクセス出来ない．そのため，今回は USEREN レジスタを変更するカーネルモジュールを作成し，これを計測前に実行させることで skia からクロック数の計測が可能となるようにした．クロック数の計測においては，並列化部分の前と後でクロック数の差分を取っており，サイクルカウント取得にかかるオーバーヘッド分も差し引いて算出した．

表 2 各 blitter 関数におけるクロックサイクル計測結果

	Sequential	Parallelized
DrawRect	742634	267821
DrawArc	2182	1140
DrawCircle2	8013	2764

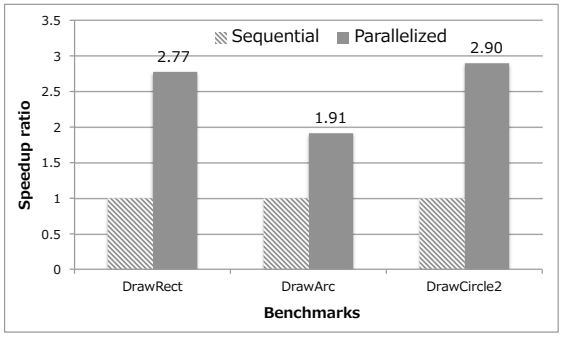

図 7 blitter 関数における速度向上結果

5.3 クロック数の計測による Nexus7 における性能評価結果

本節では，2.2 節で述べた各テストに対して，プロファイル結果に基づいて自動並列化を行った関数におけるクロックサイクル数評価結果について述べる．なお，DrawRect, DrawArc, DrawCircle2 における並列化対象となった関数はそれぞれ `SkRGB_Blitter::blitRect`, `SkRGB16_Blitter::blitH`, `SkRGB16_Blitter::blitAntiH` である．逐次処理と並列処理についての評価結果を表 2 に，性能向上率グラフ化したものを図 7 にそれぞれ示す．並列化対象関数において，DrawRect では逐次処理で 742634 サイクルであったが，3 コア並列処理によって 26821 サイクルに，同じく DrawArc では 2182 サイクルから 1140 サイクル，DrawCircle2 では 8013 クロックから 2764 サイクルとなり，DrawRect で 2.77 倍，DrawArc で 1.91 倍，DrawCircle2 で 2.90 倍の速度向上となった．

5.4 表示 FPS 値による Nexus7 における性能評価結果

本節では，ベンチマークアプリケーションの実行結果となる FPS での評価結果について述べる．FPS 値は 0xbench によるベンチマークテストにおいて，テストの実行時間と，描画命令数から，1 秒あたりの描画回数としてサイクル算出される．5.3 節での評価とは異なり，FPS は JAVA アプリケーション層から Skia の処理までを含めた描画処理全体の評価結果となる．評価結果を表 3 に，性能向上率をグラフ化したものを図 8 示す．

オリジナルの逐次処理と比べ，3 コア並列実行において，DrawRect で 22.82[fps] が 43.57[fps] に，DrawArc で 38.58[fps] が 50.98[fps] に，DrawCircle2 で 33.86[fps] が 50.77[fps] となり，DrawRect で 1.91 倍，DrawArc で 1.32 倍，DrawCircle2 で 1.50 倍の速度向上結果がそれぞれ得られた．なお，DrawCircle2 の評価では，元々のテストを並列実行した際に FPS が Android の限界値である 60 に達したため，表示する円の数を 2 倍にすることで，限界値を超えないよう設定してある．

表 3 各ベンチマークテストの FPS 計測結果

	Sequential	Parallelized
DrawRect	22.82	43.57
DrawArc	38.58	50.98
DrawCircle2	33.86	50.77

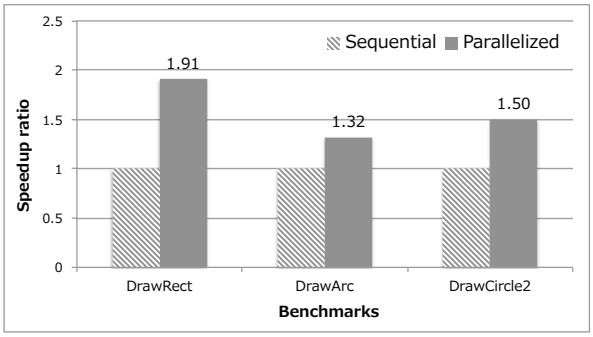

図 8 各ベンチマークテストの FPS 向上結果

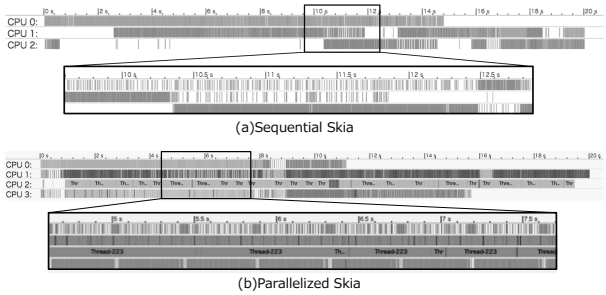

図 9 DrawRect 実行時における逐次処理と並列処理の Systrace 結果

ここで Systrace[10] を用いて，Skia の逐次処理，並列処理それぞれにおける各コアの CPU 負荷状況を詳細に解析した．図.9 は，DrawRect 実行時における，Systrace 結果を示したものである．(a) はオリジナルである Skia を用いて DrawRect を実行している時の結果であり，2 つの図は，処理全体と一部分を拡大した図である．濃く表示されている部分が CPU が処理している事を示しており，Skia が CPU1, CPU2, CPU0 とコアを変更しながら実行されていることが分かる．また，4 コア全体で空白が目立ち，マルチコアを十分に生かせていないと言える．続いて，(b) は並列化 Skia を用いて DrawRect を実行している時の結果であり，(a) と同様，2 つの図は全体と拡大図を示している．Skia の処理が CPU1,2,3 で高密度に並列実行され，その他のプロセスが CPU0 に割り当てられていることが分かる．これらの結果から，並列化 Skia の実行においては，プロセッサ全体を有効に用いることが出来ていると言える．

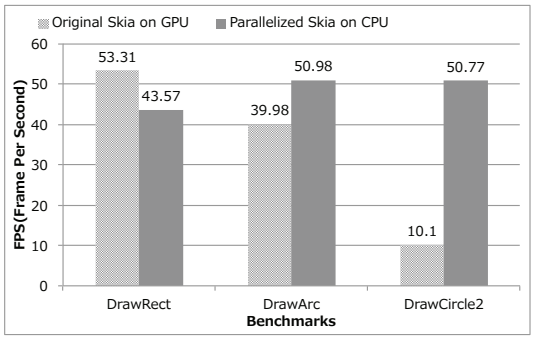

図 10 Skia の GPU 処理と並列化処理での FPS 計測結果比較

5.5 Hardware Acceralation(GPU) を用いた時との比較

Android Version 3.0 以降より，Hardware Acceralation という機能が追加された．この機能は，2.1 節で説明した，Android の Canvas API を OpenGL ES を用いて実装することで，GPU を用いて描画を高速化するために追加されたものである．アプリケーションビルド時にマニフェストファイルに以下の設定値を加える事でこの機能を用いることが可能である [10][12].

<application android:hardwareAccelerated="true">

Harware Acceralation を有効にして GPU を用いた場合と，並列化実行との結果を表したものを**図 10** で示す．DrawRect では, 3 コア並列の 43.57[fps] に比べ, GPU 処理で 53.31[fps] となったが，DrawArc で 3 コア並列の 50.98[fps] に比べ GPU 利用で 39.98[fps]，DrawCircle2 では 50.77[fps] に比べ，10.1[fps] となった．これらの結果から，DrawArc と DrawCircle2 では GPU 処理に比べても並列化による速度向上が大きく，DrawRect でのみ GPU 性能が並列化性能を上回った．速度向上率としては，GPU と比べ 3 コア並列が，DrawArc で 1,28 倍，DrawCircle2 で 5.10 倍となった．

5.6 おわりに

本論文では，Oprofile を用いたプロファイル結果を OSCAR 自動並列化コンパイラにフィードバックして，最適な並列化を行う手法の提案を行った．本手法を用いることで，コンパイル時に制御フローが定まらないライブラリ等のプログラムに対して，20 行程度の必要最低限の書き換えによって効率的に並列化による性能向上結果を得られる．本手法を Android 2D 描画ライブラリ Skia に適応し，評価を行った．評価結果としては，DrawRect テスト実行時において対象関数で 3 コアを用いて 2.77 倍，同様に DrawArc で 1.91 倍，DrawCircle2 で 2.90 倍の性能向上となった．ベンチマークテスト結果としては DrawRect で 1.91 倍，DrawArc で 1.32 倍，DrawCircle2 で 1.50 倍の表示速度向上が得られた．さらに，GPU を使用した描画処理と，3 コア並列処理との比較では，DrawArc で 1.28 倍，DrawCircle2 では 5.1 倍の速度向上が可能となることが分かった．

参考文献

[1] Blake, G., Dreslinski, R. and Mudge, T.: A survey of multicore processors, *IEEE SIGNAL PROCESSING MAGAZINE*, No. November, pp. 26–37 (2009).
[2] NVIDIA Corporation: Whitepaper NVIDIA Tegra Multi-processor Architecture, pp. 1–12.
[3] QUALCOMM Inc.: Snapdragon S4 Processors : System on Chip Solutions for a New Mobile Age (2012).
[4] Samsung Electronics Co., L.: White Paper of Exynos 5, pp. 1–8 (2011).
[5] Mallón, D., Taboada, G. and Teijeiro, C.: Performance Evaluation of MPI , UPC and OpenMP on Multicore Architectures, *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer Berlin Heidelberg, 2009.*, pp. 174–184 (2009).
[6] Kasahara, H., Obata, M. and Ishizaka, K.: Automatic coarse grain task parallel processing on smp using openmp, *Workship on Lan- guages and Compilers for Parallel Computing*, pp. 1–15 (2001).
[7] Google: skia 2D Graphics Library.
[8] Apple Inc.: Quartz 2D Programming Guide, Technical report (2012).
[9] Worth, C. and Packard, K.: Xr: Cross-device rendering for vector graphics, *Ottawa Linux Symposium* (2003).
[10] Google: Android Developers.
[11] Kim, Y.-J., Cho, S.-J., Kim, K.-J., Hwang, E.-H., Yoon, S.-H. and Jeon, J.-W.: Benchmarking Java application using JNI and native C application on Android (2012).
[12] Jim Huang: Hardware Accelerated 2D Rendering for Android, *Android Builders Summit 2013* (2013).
[13] 0xlab: 0xbench.
[14] Ishizaka, K., Obata, M. and Kasahara, H.: Coarse Grain Task Parallel Processing with Cache Optimization on Shared Memory Multiprocessor, *Proc. of 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC2001)* (2001).
[15] Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K. and Kasahara, H.: Hierarchical Parallelism Control for Multigrain Parallel Processing, *Lecture Notes in Computer Science*, Vol. 2481, pp. 31–44 (2005).
[16] Shirako, J., Oshiyama, N., Wada, Y., Shikano, H., Kimura, K. and Kasahara, H.: Compiler Control Power Saving Scheme for Multi Core Processors, *Lecture Notes in Computer Science*, Vol. 4339, pp. 362–376 (2007).
[17] Kimura, K., Wada, Y., Nakano, H., Kodaka, T., Shirako, J., Ishizaka, K. and Kasahara, H.: Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor, *Proc. of 9th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9)* (2005).
[18] Cohen, W.: Tuning Programs with OProfile, *Wide Open Magazine*, pp. 53–62 (2004).
[19] Lee, N. and Lim, S.-S.: A whole layer performance analysis method for Android platforms, *2011 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia*, pp. 1–1 (online), DOI: 10.1109/ESTIMedia.2011.6088515 (2011).
[20] NVIDIA: NVIDIA Developer Zone.
[21] ASUSTeK Computer Inc.: Nexus7 Specifications.
[22] ARM Corporation: Cortex-A9 Technical Reference Manual.