Vol.2013-OS-124 No.2
2013/2/28

マルチコア商用スマートディバイスの評価と並列化の試み

山本 英雄[1]　後藤 隆志[1]　平野 智大[1]　武藤 康平[1]　見神 広紀[1]　Dominic Hillenbrand[1]　林 明宏[1]
木村 啓二[1]　笠原 博徳[1]

概要：半導体プロセスの微細化に伴いスマートフォン, タブレットに代表される民生機器にも 4 コア程度のマルチコア SoC の採用が進んでいる. 一方, ソフトウェアはマルチコアを活用するための並列化が十分に進んでおらず, 対応が望まれている. 本稿では Android を搭載した商用スマートデバイスにおいて, 一般的な利用範囲におけるマルチコアの活用状況を評価し, 並列化されたベンチマークプログラムを用いて実行環境の課題と改善方式を述べた上で, 標準 API の仕様を変更すること無く, アプリケーションがオフスクリーンバッファを描画バッファに書く BitBLT 処理の並列化を試みた結果を報告する. この処理並列化の結果, アプリケーションから 2D 描画 API を呼び出すベンチマークテストで約 3%のフレームレートの改善を確認した.

Yamamoto Hideo[1]　Goto Takashi[1]　Hirano Tomohiro[1]　Muto Kohei[1]　Mikami Hiroki[1]
Dominic Hillenbrand[1]　Hayashi Akihiro[1]　Kimura Keiji[1]　Kasahara Hironori[1]

1. はじめに

半導体プロセスの微細化は, 携帯機器の高性能化と一層の低価格化を促進し, 民生機器出荷台数でパーソナルコンピュータを凌駕する事となった. これに伴い, 各 SoC ベンダまたは携帯機器ベンダ間の競争は激しさを増しておりGHz クラスの周波数で動作するコアを 4 台備えたマルチコアプロセッサが実用になっている.

一方、スマートディバイスは小型の LCD とタッチスクリーンで UI(ユーザインターフェース) を構成し, 利用者は原則的に 1 つの処理を実行する. この様な利用シーンでは複数の処理を同時に行う必要性は少ないと想定し, マルチコアの有効性の検証が必要と考えた.

そこで本研究では,4 コアのプロセッサを搭載した商用スマートディバイスを評価環境とし, そこでのマルチコアの使われ方と制御方式を評価し, 今後の商用化に向けた制御方式について 2D 描画ライブラリを対象として考察と評価を行った.

本稿では,2 章で評価環境について, 3 章でマルチコアの利用状況にいて, 4 章で並列化プログラムを用いたプラットフォームの評価と特徴を, 5 章で並列化プログラムのための実行環境とその実装を, 6 章で評価対象の Android の構造とその並列化を, 7 章で描画ライブラリ並列化の実装を, 8 章で関連研究を, 9 章でまとめと今後をそれぞれ述べる.

[1] 早稲田大学 理工学術院 情報理工学科
Dept. of Computer Sci. & Eng., Waseda Univ.

2. 評価環境の構成

本研究では最新の 4 コア SoC を搭載した Google Nexus-7 を評価対象とした. 主な仕様を以下に示す.

- NVIDIA 社 Tegra-3: ARM Cortex-A9 Quad-core
- Clock 最大 1.3GHz

評価環境で採用している Tegra-3 は高性能コアを 4 個, 低消費電力コアを 1 個備えており, 必要とされる処理能力に応じて高性能コアと低消費電力コアを切り替えて使う点に特徴があるとされている [7].

評価環境の構築にはカスタム ROM の開発環境を選定し, 日本で開発とメンテナンスが行われている JCROM project [9] の配布物を利用した. JCROM を利用した理由は Android Open Source Project として公開されているソースコードに加えて, 開発者向けのスクリプトと手順書が充実しており再現性が高いためである.

3. 評価環境におけるマルチコアの利用状況

この評価環境で, 一般的な利用を想定して各コアの利用状況を調査した. 調査は次の処理を順に操作した場合の各コアの利用状況を Android SDK に添付されている, システム性能解析ツール systrace で表示して解析した.

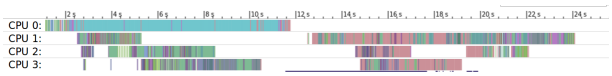

図 1　twitter と Web 閲覧時のマルチコア利用状況

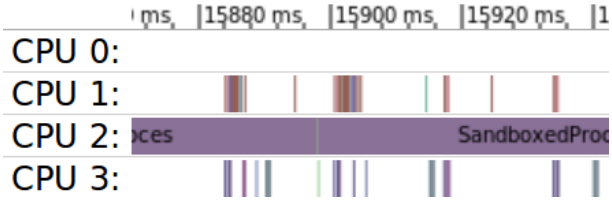

図 2　twitter と Web 閲覧時のマルチコア利用状況 拡大

- twitter の閲覧 (タイムライン更新).
- Web の閲覧 (www.cnn.com の表示完了まで).

以上の操作中に採取したマルチコア利用状況を**図 1**, **図 2** に示す.

図 1 の結果から常時 3 コア程度が動作しているように見えるが, 拡大した図 2 を見ると, ほぼ 1 コアのみが動作している事がわかる. この結果から, 今回想定した利用範囲では複数のコアを利用する機会が少ない事が解る. この理由として, Android の一般的なアプリケーションは Java で記述されており, 複雑な演算処理はマルチスレッド化しても実行性能の向上が望めない事, および Java でマルチスレッド化する場合もメインスレッドは UI を司るスレッドとして制約事項が多い事の 2 点が考えられる.

この対処法として Android では C++等の言語で作成したネイティブバイナリを Java から呼び出す手法が Android NDK として推奨されている.

Android NDK ではネイテイブバイナリを JNI [*1] を用いて Java からクラスライブラリとして利用できる. そこで JNI を用いて並列化をすすめる事を想定し, 並列化したネイティブバイナリの実行環境の評価を試みる事にした.

4. 並列化プログラムの動作

評価環境での並列化プログラムの動作を確認する目的で, 十分に並列化されたプログラムを評価環境で実行させて, その挙動を観察した.

評価はＡＡＣエンコーダ逐次プログラムを OSCAR コンパイラ [1] で並列化したベンチマークプログラムで行った. このベンチマークプログラムは wave 形式のファイルを入力としこれを AAC 形式にエンコードする. この過程で指定したコア数のスレッドを生成し, 各スレッドは同期をとりながら並列処理を行う.

このプログラムを評価環境で動作させた結果を**表 1** に示す.

一見してわかるように, 実行時間には 2 倍程度のばらつきがあり想定とは異なる結果となった.

この原因を解析するために systrace を用いて調査をおこなった. 表 1 で 5 秒程度の実行時間を要した場合を systrace で表示した結果を**図 3** に示す.

図 3 の上段はプログラムの実行から完了の時間範囲で, スレッドの実行状態を表示している. 図中 CPU0,CPU1,CPU2,CPU3 は 4 個のコアを示しており、CPU0 で起動した並列プログラムがスレッドを起動し, 特定の CPU に割り付けられる様子を示している.

同図の下段はプログラムの起動から, プログラムから指定した CPU にスレッドの割り付けが完了するまでを示している. この図で Start から Migrated の間は 440.6ms を要している事が解った. これはスレッドが CPU の割り付けを要求した時点で対象 CPU がオフライン状態にある場合に発生している. この場合, 一旦既にオンライン状態にある CPU で動作を開始し, 負荷に応じて他の CPU が順次オンライン状態に遷移する. その後に CPU 間でスレッドの移動が発生する.

以上. systrace の解析から CPU の負荷とオンライン・オフラインの自動制御について, 次の事がわかった.

- CPU 負荷が少ない場合は 1 個のコアのみがオンライン状態, 他 3 個はオフライン状態にある
- CPU 負荷の上昇により, さらにコアが 1 個オンライン状態になる
- CPU 負荷の下降により, コアが 1 個オフライン状態になる
- オンライン・オフラインの状態遷移には数 ms 以上の遅延時間が存在する

表 1 で 5 秒台の実行結果となる場合はすべてこのような遷移を経ていた.

この結果を裏付ける目的で, 評価環境でのコア数調整機能と周波数変更機能についてソースコードを調査した結果次の事がわかった.

評価環境では, 一般的な Linux に備わる CPUfreq, CPUidle に加えて,Tegra-3 用 Linux の独自のコードとして CPU の Auto hotplug 機能が追加されている.

一般的な Linux と同様に, Tegra-3 においても周波数の管理は CPUfreq が行い, CPU の負荷に応じて設定を行なっている. CPUfreq は CPU の負荷を観察し, 負荷の増加に対応して動作周波数を上げていく. CPU の動作上限まで周波数が上がっても処理能力が不足した場合には Auto hotplug が動作しオフライン状態のコアを 1 個オンライン状態に遷移させる.

同様に CPUfreq により周波数を低下させても処理能力に余裕がある場合には Auto hotplug がオンライン状態にあるコアから 1 個を選択しオフライン状態に遷移させる.

[*1] Java Native Interface

表 1　Nexus-7 現状における実行性能

サンプル	1	2	3	4	5	6	7	8	9	10	平均	最遅	最速
実行時間 (s)	5.12	5.08	3.65	5.05	2.78	2.73	5.06	2.74	5.05	2.74	4.00	5.12	2.73

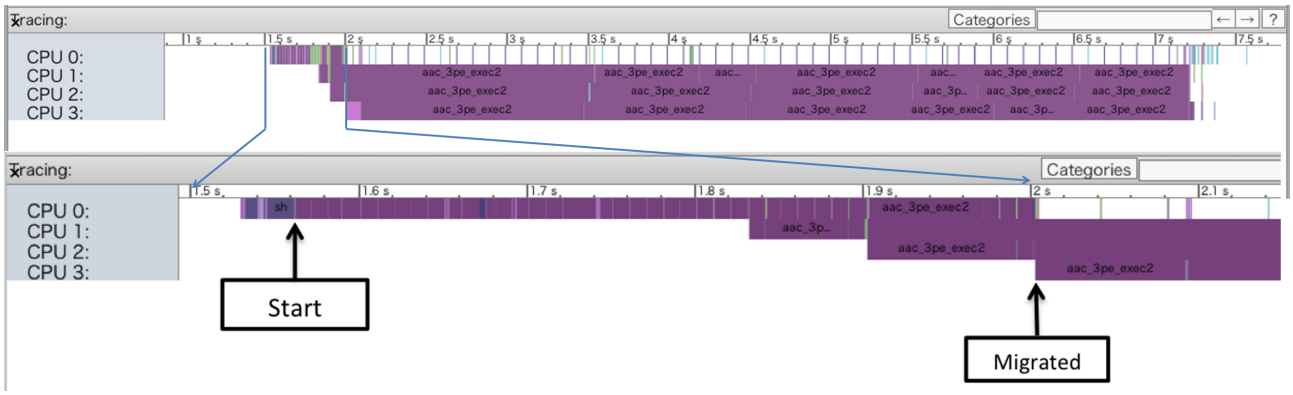

図 3　Nexus-7 現状における並列実行状況

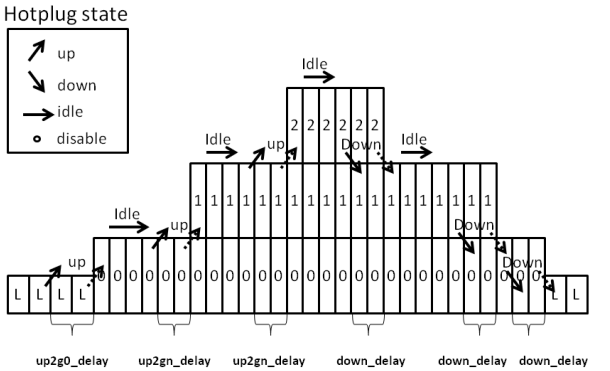

図 4　Auto hotplug 機能の動作例

Auto hotplug 機能は内部に up,down,idle,disable の 4 種類の状態を持ちそれぞれ次の意味を持つ,

up
　動作周波数が上限値に達しているのでコアを追加する必要がある

down
　動作周波数が下限値に達しているのでコアを削除する必要がある

idle
　動作周波数が上限値よりも低く、下減値よりも高いのでコア個数の変更は行わない

disable
　Auto hotplug 機能を停止する

この 4 種類の状態の変更は CPUfreq による動作周波数の調整の都度行われる．コア数の実際の変更は図 4 の up2g0_delay, up2gn_delay, down_delay の遅延時間を経て行われる．

論理的に 4 コアを持つ CPU は, 物理的には 4 個の高性能コアと 1 個の低消費電力コアを持ち, メールや SNS の同期等の負荷の低いバックグランド処理を高性能コアから切り替えて行う．

Tegra-3 用 Linux kernel は各コア毎に動作周波数を管理する構造を持つが, Tegra-3 ではすべてのコアが同じ周波数で動作する．したがって, 各コアに異なる動作周波数を設定しても有効になるのは最も高い動作周波数である．

以上をまとめるとつぎのようになる．

- 各コアは同じ周波数で動作する
- 1 コアのみ動作している場合に限り, 低消費電力コアに切り替えて動作する事ができる
- 周波数変更では対応できない範囲の処理能力調整をコアの動的な追加削除で行っている
- 急激な負荷の変動に対してコアの追加削除を意図的に遅延させ動作を安定化している (ローパスフィルタに相当).

以上の結果から, 評価環境のマルチコア制御機能 Auto hotplug は, CPUfreq による動的自動周波数変更機能の拡張として実現されており, CPU 負荷が低い場合にはコア数最少になるように調整される．このため細粒度に並列化され Spin lock を用いるような並列化プログラムの動作環境としては適切ではないと判断される．

5.　並列化プログラムの実行環境

これまでに述べたように, 評価環境でのマルチコアは低消費電力化に注力しており, 並列化プログラムの動作環境としては不適当である事が解った．そこで, 今回は低消費電力化については今後の課題として, 並列化プログラムの理想的な実行環境の作成を試みた．

並列化プログラムの実行環境としての要件は次の 3 点とした．

- アプリケーションから必要なコアをオンライン・オフライン状態にできる事
- 並列化されたスレッドが特定のコアに割り付け可能な事
- 評価対象としない他のプログラムが, 評価対象とした並列化プログラムの動作を妨げない事

5.1 並列化ランタイムライブラリの仕様

並列化ランタイムライブラリの仕様は以下とした.

- アプリケーションが CPU Auto Hot Plug 機能を無効化できる事
- アプリケーションが CPU のオンライン・オフラインの変更を行える事
- 複数の並列化アプリケーションの要求を管理できる事
- アプリケーションが異常終了した場合に獲得した資源を解放できる事
- 評価対象以外のプログラムはコア#0 で動作する事.

5.2 実行環境で利用するカーネルの機能

Auto hotplug の制御用システムテムコールは Tegra-3 用 Linux kernel では提供されていないが, ソースコードを調査したところ sysfs の操作で同等の制御が行えることが解ったため, これを用いる事とした.

評価対象以外のプログラムをコア#0 で動作させるためには, Android の init コマンドを修正し, システムコール sched_setaffinity を用いて init が起動したプログラムはすべてコア#0 で動作するよう修正を加えた.

5.3 実行結果

並列化実行環境を利用して ACC encoder を実行した結果を**表 2** に示す.

表 2 で示した並列化実行環境での結果は, 表 1 で示した結果と比較して, 実行時間のばらつきはなく, かつ実行時間も短くなっている.

この並列化実行環境での systrace の出力結果を**図 5** に示す.

図 5 では, 図 3 で見られた以下の問題が解消されている事がわかる.

- Auto hotplug によるコアの追加に伴う遅延時間
- 一旦, 任意のコアで実行を開始した後に, 追加されたコアにスレッドが移行する処理

この 2 点を含めた並列動作安定までに要した時間 (図中 Start から Migrated の間) は改善前 440.6ms から 7.2ms へ大幅に改善された.

以上で, 並列化プログラムの実行環境が整った事が確認できた.

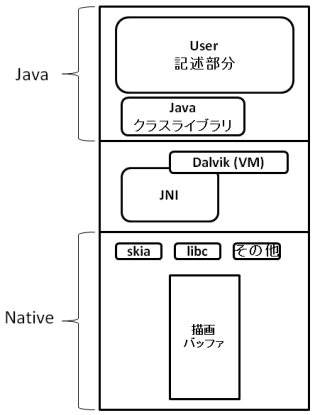

図 6 Android アプリケーション空間の構成

6. Android の構造とシステムの並列化

6.1 システムレベルの並列化

前章までにネイティブバイナリを例題として並列化アプリケーションの効率的な実行について, それを阻害する要因の解析を行い, その改善策としての並列化ランタイムライブラリを開発し, その効果について述べた. ここでは、並列化ランタイムライブラリを用いてシステムレベルの並列化について考える.

本研究で開発した並列化ランタイムライブラリを NDK と共に用いて, 並列化クラスライブラリを作成すれば, Java からそれを呼び出すことでこれまで実現できなかった並列化アプリの効率的な動作が可能となるだろう. 一方,NDK の利用には Java と C++の 2 種類のフレームワークの習熟が必要である. この事はこれまで Java のみで用いて開発を行ってきた一般的なデベロッパには大きな負担を強いる事が想定できる.

そこで, NDK や並列プログラミングの理解, および既存プログラムの変更無しに並列化による高速化の恩恵を得る手段があれば, 商用システムでのマルチコアの価値はより高いものになる. 以下に, Android の構造を振り返りシステム側での並列化について検討を行った.

6.2 Android の構造

Android アプリケーション空間の構成を**図 6** に示す. Java で書かれたアプリケーションは Dalvik と呼ばれる独自実装の Java VM で処理を行う.Android の Java API の多くは JNI として実装されており,JNI 下位層にあるネイティブライブラリで処理が行われる. 一例をあげると Java のクラスライブラリ canvas.drawRect を呼び出すと JNI で実装されたネイティブ描画ライブラリ Skia の SkCanvas が実行され矩形をメモリ上に展開する.

表 2　Nexus-7 本研究の並列化実行環境での実行性能

サンプル	1	2	3	4	5	6	7	8	9	10	平均	最遅	最速
処理時間 [s]	2.70	2.71	2.72	2.72	2.73	2.73	2.76	2.72	2.72	2.73	2.72	2.76	2.70

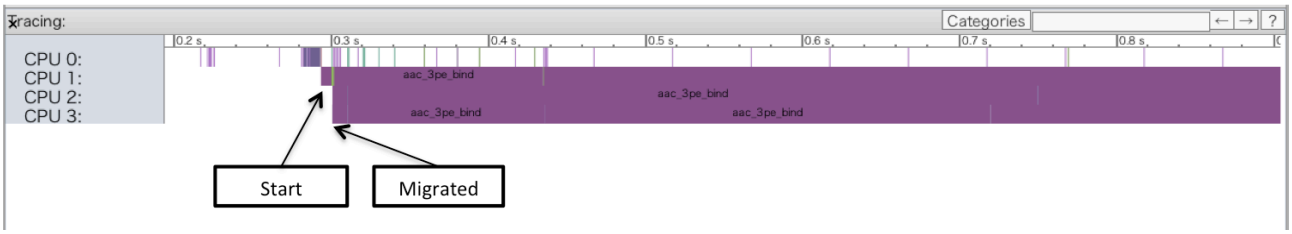

図 5　Nexus-7 本研究の実行環境　並列実行状況

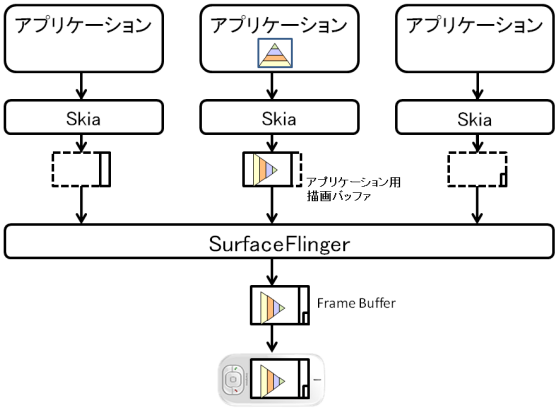

図 7　Skia と Surfaceflinger

図 7 に示すように, 描画の対象となるメモリはアプリケーションとは別プロセスとして実装された重ね合わせ処理 surfaceflinger との間で共有されている. surfaceflinger は定期的に各アプリケーション領域を重ね合わせて framebuffer に書き込む.

Skia ライブラリはインターフェース層, 演算層, BitBLT 層で構成され, 演算層は描画処理に応じた演算中心の層,BitBLT 層はアプリケーション画面へのイメージ転送を行うメモリコピー中心の層である.

BitBLT 層の処理については, 当初は単純なループ処理, 次いで軽度の unrolling で最適化され, 現在は ARM の SIMD 命令セット NEON による最適化が行われている. この事は BitBLT 層が描画性能におけるボトルネックとなっている事を示しており,Skia のドキュメントにも明示されている.

以上から Android のシステム性能向上は,JNI 経由で呼び出されるネイティブコードの改善が有効である事が解る. そこで, 前に述べた並列化ランタイムを利用して, 既に SIMD 命令セットで最適化された逐次プログラムを並列化する事により一層の改善を試みることにした.

7. 描画ライブラリ Skia の並列化

描画ライブラリ Skia は円・矩形等の基本図形の描画の他, ビットマップの展開など 2D の描画処理のすべてを処理しており各処理毎の演算層のソースが大半を占めている. 一方,BitBLT 層はスケーリング, α ブレンディングを行いながらのメモリ転送処理を中心としており. 種類・規模は限定的である. 本研究においては Skia の並列化の有効性を検証することを目的としているため, 比較的簡単に並列化が可能を思われる BitBLT 処理の 1 つを選択して並列化の対象とした.

対象とした処理は bitmap 描画の BitBLT 転送で利用される関数 decal_nofilter_scale() であり C++で記述されている. プロトタイプ宣言を以下に示す.

```
void decal_nofilter_scale(uint32_t dst[], SkFixed fx,
                          SkFixed dx, int count)
```

この関数は, fx で示される画素データを,dst で与えられたバッファに,dx で与えられた画素数分オフセットしてcount 分転送する. 1 画素ずつ count 分転送する場合の処理を以下に示す.

```
uint16_t* xx = (uint16_t*)dst;
for (i = count; i > 0; --i) {
    *xx++ = SkToU16(fx >> 16); fx += dx;
}
```

これは, アイコンのような矩形イメージをより大きなバッファ, 典型的にはアプリケーション用の描画バッファに書き込む場合に使われる代表的な BitBLT 処理と言える. この処理では, 図 8 に示すように矩形イメージの転送先の座標を 90°回転させている . この処理を画素の列方向に繰り返すことによって, ポートレートモードのアプリケーションバッファに矩形イメージを転送している.

転送の過程で 32 ビットのピクセルデータを 16 ビットに変換する. この変換と転送の Skia 処理を現在の Android 商用機では ARM の SIMD 命令セット NEON を利用して

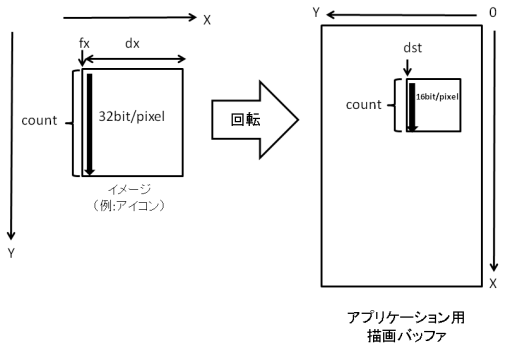

図 8　座標軸の変換とスケール処理

8 ピクセル単位で繰り返し処理を行い, 8 ピクセル以下の場合は 1 ピクセル単位で処理を行っている.

7.1　データ分割とコアの割り当て

データ分割とコアの割り当ては次の方針で行った

- Android のシステムプロセスなど、評価対象以外のプログラムは,#0 で動作
- 評価対象プログラムのメインスレッドはコア#1 で動作
- データを 2 分割し, それぞれコア#2, コア#3 に動作

並列化の対象とした処理は途中の結果を参照する必要はないため, スレッド間で同期を取るのは Skia 関数から JNI に復帰する時点まで遅延させる事ができる. この事は従来の処理の転送に要した時間を完全に隠ぺいできる可能性がある事を暗示している.

スレッドの生成と完了の待ち合わせは次の方針で行った

- 初回の呼び出し時にスレッドを生成
- システムコール sched_setaffinity でコアに割り付け
- 演算の開始、完了の待ち合わせはスレッド間で共有する volatile 変数の参照・更新で行う
- 待ち合わせは Spin lock 方式で行う
- 待ち合わせ処理は Skia のインターフェース層で行う

対象とした関数の処理量は引数 count で指定されたピクセル数に依存する. 方針決定のために count の指定を調査した結果最大でも 256 程度の値であり, 処理量は少なく、処理時間もわずかである事がわかった.

したがって, 毎回スレッドの生成・回収を行う事はピクセルの処理量を上回り並列化の効果は望めない. このためスレッドを一旦生成した後, 評価プログラムが終了するまで動作を続ける事とした.

以上の方針に沿ってプログラムを作成した. 変更前の逐次 BitBLT 処理を**図 9** に, 変更後の並列 BitBLT 処理を**図 10** に示す.

図 9 に示す変更前の処理では,8 画素単位で NEON 命令で繰り返し処理をしていた. 一方, 図 10 の変更後の処理で

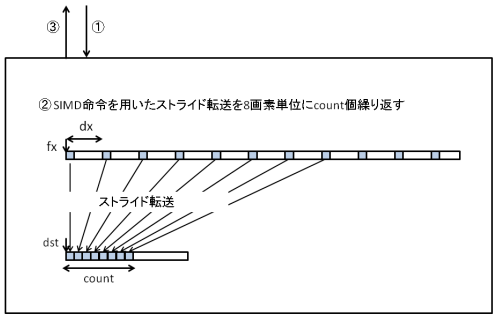

図 9　変更前の BitBLT 処理 (逐次)

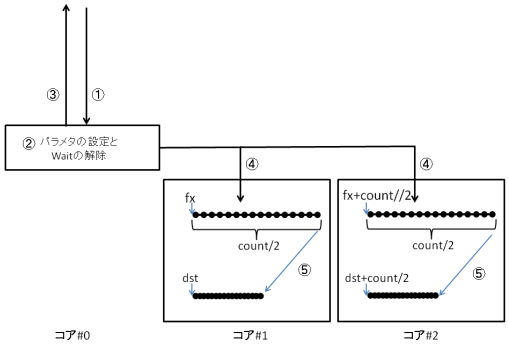

図 10　変更後の BitBLT 処理 (並列)

表 3　変更前 逐次処理の描画フレームレート (FPS)

サンプル	1	2	3	4	5
Round 0	44.56	44.88	43.95	44.56	44.82
Round 1	44.91	44.67	45.18	44.91	44.42
Average	44.74	44.77	44.56	44.74	44.62

表 4　変更後 並列処理の描画フレームレート (FPS)

サンプル	1	2	3	4	5
Round 0	45.89	45.84	45.89	45.87	45.80
Round 1	45.82	45.90	46.16	46.08	45.86
Average	45.85	45.87	46.02	45.97	45.83

は処理開始を待つスレッドにパラメタを設定した後に待ち合わせを解除した後, 復帰している.

7.2　測定結果

性能測定には 0xbench を用いた. 0xbench は Android の C ライブラリ,3D,2D, Java VM, Java script の性能を測定できる. 今回の測定はこの中の 2D - Draw Image で行った. 測定項目 Draw Image は標準 API canvas.drawImage を用いてアプリケーション描画バッファを 500 回更新し, その経過時間からフレームレートを出力する.

skia の変更前の逐次処理の結果を**表 3** に, 変更後の並列化処理の結果を**表 4** に示す. 値の単位は FPS. それぞれの最速値の値を比較して 3%の向上が確認できた.

8. 関連研究

CPU 負荷の動的な要求に適合するために, タイムインターバルを設定し、その間, 負荷に適合可能な動作周波数や動作電圧を決定し, 動作に不必要な機能ユニットへは電力の供給を遮断する制御方式が提案されている。[5]

半導体集積度の進展に伴い, マルチコアやメニーコア構成のプロセッサがハイエンドサーバからスマートデバイスにまで採用されるに至っている. マルチコアやメニーコア構成のプロセッサに対して, 低消費電力化と高性能化の実現技術として、DVFS(Dynamic Voltage Frequency Scaling) やコア個数の動的変更が, 幅広く採用されている. 制御方式は千差万別である。

プログラムを予備的に実行させその動的特性情報から最適電力供給を達成する周波数、電圧値、コア数を決定する方式 [3]. また, チップの発熱量制限値を基準に動的にコア数と周波数を選択する方式 [6]. また, コアの稼働率を監視し、高（低）負荷に遷移すれば周波数の高（低）周波数化および/またはコア数の増加（減少）を行う方式 [7] また、リアルタイム処理のアプリケーションにおいて余裕時間の部分に DVFS を適用する方式 [4] などが既存の制御方式である。

一方、本研究においては、逐次プログラムを OSCAR 自動並列化コンパイラーで並列化し、ここで生成される静的な並列プログラム情報に基づいて最適なコア数、動作周波数、電源遮断・投入を指示し [2]、Android OS 等のスケジューラと並列化ランタイムが連携動作する事による高効率な並列処理を行う事を特徴とする.

9. おわりに

9.1 本研究の成果

本研究では、消費電力の制約が強い商用スマートディバイスの DVFS とコア単位の Power Gating の 2 つの項目について実装と特性を明らかにし, OSCAR コンパイラで生成した並列化アプリケーションの効率的な実行環境の実装と評価を行った上で, Java で記述された Android アプリケーションの 2D 描画処理 Skia の並列化を行い, 有意な性能向上を Java で書かれたアプリケーションから確認できた.

4 コアのマルチコアプロセッサを搭載した最新の商用スマートディバイスは, 一定時間継続した負荷が発生し, なおかつ発生した負荷が並列処理可能な場合に限って複数のコアが一定の遅延時間を経て順次オンライン状態に遷移する事が解った.

この特性により, 短時間の負荷は並列実行の効率的対象とならず, したがって速い対話性能を要するアプリケーションでは仮に並列化した場合でも効率的に実行されない事を明らかにした.

この問題に対して, 本研究では急峻な負荷変動特性を持つ並列化プログラムを効率良く実行する並列化ランタイムを開発してその効果を単独の並列アプリケーションで確認した.

さらに,Java から JNI 経由で呼び出し可能な 2D 描画処理を標準 API を変更する事なく並列動作させ, Java からの描画性能が向上する事を示した. 今回並列化したのは skia BitBLT 層の 1 関数のみであり, より演算負荷の高い Skia 演算層の並列化を行えば一層の性能向上が期待できると考えている.

9.2 今後の課題

今回は商用プラットフォーム上で並列化アプリケーションを効率的に動作させる事を目標とし, 一定の成果をえた. 一方で今回の並列化の試みでは省電力化については考慮をしていない. 商用機をターゲットとした研究としては, 並列化による高速処理に合わせて省電力化についての検討が必要となる. 今後は OSCAR 並列コンパイラによる並列プログラム情報を利用する事で並列化による高速化と並行して消費電力の評価を進めていく.

参考文献

[1] Hironori Kasahara, Motoki Obata, Kazuhisa Ishizaka, ”Automatic Coarse Grain Task Parallel Processing on SMP using OpenMP”, Proc. of 13th International Workshop on Languages and Compilers for Parallel Computing (LCPC’00), Aug., 2000.

[2] Jun Shirako, Naoto Oshiyama, Yasutaka Wada, Hiroaki Shikano, Keiji Kimura, Hironori Kasahara, ”Compiler Control Power Saving Scheme for Multi Core Processors”, Proc. of The 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC2005), Oct. 2005.

[3] 今村智史, 佐々木広, 福本尚人, 井上弘士, 村上 和彰, ”コア数と動作周波数の動的変更によるメニーコア・プロセッサ性能向上手法の提案”、IPSJ SIG Technical Report, Vol.2012-ARC-200 No.17,

[4] 嶋田裕巳, 小林弘明, 高橋昭宏, 坂本龍一, 佐藤未来子, 近藤正章, 天野 英晴, 中村 宏, 並木 美太郎, ”リアルタイム OS における細粒度パワーゲーティング制御の設計と実装”、IPSJ SIG Technical Report, Vol.2012-ARC-200 No.16,

[5] Jonathan B. White et al: ”Adaptive Power Consumption Techniques”, NVIDIA Corporation, USP 2010/0131787 A1,May 27,2010

[6] Intel Whitepaper, ”Turbo Boost Technology in Intel Core Microarchitecture Based Processor”, 2008

[7] NVIDIA Corporation, ”Whitepaper Variable SMP (4-PLUS-1) — A Multi-Core CPU Architecture for Low Power and High Performance” (2011)
入手先 ⟨http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-Architecture-for-Low-Power-and-High-Performance.pdf⟩

[8] Google, ”skia 2D Graphics Library”
入手先 ⟨http://code.google.com/p/skia/⟩

[9] androidsola, ”JCROM Project”
入手先 ⟨https://sites.google.com/site/jcromproject⟩