RO-007

プロセッサ抽象化 API 利用アプリケーションの並列処理タスクへのプロセッシングエレメントの動的な配分機構の Android 実装

Android Implementation of Dynamic Processing Element Allocation Mechanism on Application Using Processor Abstraction API

榊原 宏章† 白石 善明† 岩田 彰†
Hiroaki Sakakibara Yoshiaki Shiraishi Akira Iwata

1．まえがき

アプリケーションは，異種のプロセッサを使用できるプロセッサ抽象化 API を利用して，端末に搭載された複数のプロセッサのリソースを使用できる．多くのプロセッサ抽象化 API はタスクに割り当てるリソースの指定が可能であり，状況に応じて適切にリソースを使用できれば，消費電力あたりのパフォーマンスの向上が期待できる．

プロセッサのリソースを適切に使用するための研究の 1 つとして，リソースの動的な配分に関する研究がある．リソースの適切な配分は端末構成や処理状況によって異なるため，リソースを動的に配分することで高いパフォーマンスを得ることを目指している．スマートフォンやタブレット PC 等の携帯端末では，消費電力や占有面積への制限という観点からリソースの有効活用に対する需要が高い．つまり，携帯端末ではアプリケーションが端末の差異を意識することなく，利用状況に応じてプロセッサリソースを適切に使用できることが望ましい．

著者らは携帯端末に多く用いられている Linux 系 OS において，バイナリに変更を加えることなく，プロセッサ抽象化 API 利用アプリケーションが利用するプロセッサリソースを動的に配分する機構の実現方式を提案している[1]．Linux 上に実装し，汎用的に利用可能であることと，省電力化効果があることを確認した．

多様な実行環境・利用状況が想定される携帯端末においては，動的にリソースの配分を行うことで，各々の端末の性能を活かした処理を行うことが期待できる．また，ランタイム環境の追加導入やアプリケーションの改変を必要としないといった既存アプリケーションに対する互換性を持つことが望ましい．Windows が搭載された PC を対象として，動的なリソース配分と互換性確保の課題に取り組んだ研究に PACUE[2]がある．PACUE ではアプリケーションへの介入に API フックを用いることで，バイナリの改変を不要としながら動的なプロセッサ変更を可能としている．提案方式は，PACUE の構成を基本的に継承することで，既存アプリケーションへの互換性を実現しつつ Linux 系 OS でのリソースの動的なリソースの配分を行う．

文献[1]では Linux 上での提案方式の実装と評価を行っているが，本論文では，Android への提案方式の適用を図る．提案方式によって配分するプロセッサのリソースは，プロセッサの中の計算ユニットの単位であるプロセッシングエレメント（PE）とした．提案方式を Android 上に OpenCL[3]を用いて実装し，提案方式によるタスクへの PE の配分の効果を確認する．

†名古屋工業大学，Nagoya Institute of Technology

本論文は，以下，2.で文献[1]において提案した方式について述べ，3.で Android における提案方式の実装について述べる．4.で評価について述べ，5.で本論文をまとめる．

2．提案方式

2.1 概要

文献[1]での提案方式によって配分するプロセッサのリソースとして PE とローカルメモリが挙げられるが，本論文では主な構成要素である PE とした．PE の適切な配分は，PE の個数および利用状況によって異なる．前者は並列に動作可能なスレッド数に合わせ，後者は負荷に合わせて調節を行うことが望ましいことからである．多くのプロセッサ抽象化 API では，PE へのタスクの割当を指定することができる．本論文ではこの機能を利用して提案方式による PE の動的な配分を行う．

提案方式の概要が図 1 である．提案方式は 2 つの機構から構成される．配分するリソースを PE と特定したことに合わせて文献[1]から機構名を変更しており，“プロセッサリソース割り当て機構”を“PE 割当機構”，“配分機構”を“PE 選択機構”とした．

提案方式は，PE へのタスクの割当をタスクへの PE の配分と捉える．PE 割当機構は，API フックによりアプリケーションからのプロセッサ抽象化 API 呼出へ介入し，タスクが割り当てられる PE を変更した上でアプリケーションが指定した同じ API を呼び出すことにより，PE への割当の変更を行う．PE 選択機構は，取得した端末情報や実行情報などの利用状況データを元に，タスクへ割り当てられる PE の選択を行う．PE の動的な配分は，PE 選択機構の結果に従って，タスクが割り当てられる PE を PE 割当機構が変更することによって実現する．

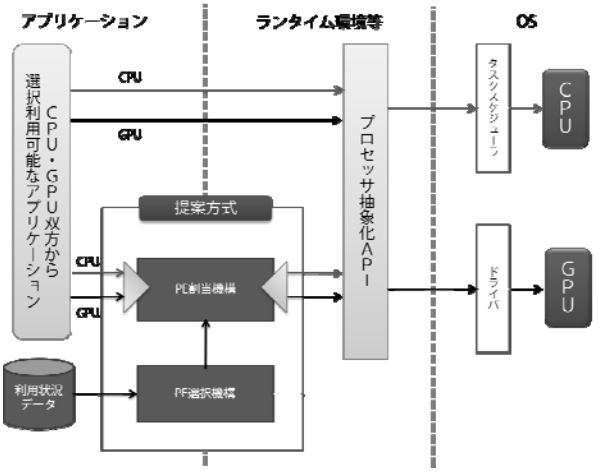

図 1 提案方式の概要

なお，提案方式は PACUE の構成を継承しているため，特別な場合としてプロセッサの動的な変更も可能である．

2.2 構成と機能

提案方式の全体構成を図 2 に示す．提案方式の 2 つの機構は，各々が複数個のモジュールを持つ．配分するリソースを PE としたことに合わせて，文献[1]の全体構成の“リソース配分モジュール”を“PE 選択モジュール”，“リソース変更モジュール”を“PE 変更モジュール”と変更した．また，使用プロセッサの変更に関わる“プロセッサ配分モジュール”と“プロセッサ変更モジュール”を省いた．

提案方式は，PE へのタスクの動的な割り当てに使われる“PE への割当の変更”，PE の配分を決定するために使用する情報の取得・保存に使われる“実行情報の取得・保存”，“端末情報の取得・保存”の 3 つの主な機能がある．それぞれの機構における各機能の動作は次の通りである．

2.2.1 PE 割当機構の機能

“PE への割当の変更”では，アプリケーションが呼出した，PE へのタスクの割り当てを指定する API に介入し，PE 変更モジュールを呼び出す．PE 変更モジュールは通信機能モジュールを介して，PE 選択機構に問い合わせて，タスクが割り当てられる PE を取得する．そして，その PE へタスクの割り当てを変更して，介入した API を呼び出す．

“実行情報の取得・保存”および“端末情報の取得・保存”では，取得する情報に関連した API の呼出に介入し，それぞれ，実行情報取得モジュールにおいて実行情報の取得，プロセッサ情報取得モジュールにおいて端末情報の取得を行い，通信機能モジュールを介して PE 選択機構へ受け渡す．

2.2.2 PE 選択機構の機能

“PE への割当の変更”では，通信機能モジュールで受信した PE 割当機構の問い合わせに対し，PE 選択モジュールを呼び出す．PE 選択モジュールは，端末情報監視モジュール，実行情報管理モジュール，プロセッサ情報管理モジュールから保存された端末情報や実行情報を取得し，それらの情報をもとにタスクへ割り当てられる PE を選択し，PE 割当機構に返す．

“実行情報の取得・保存”では，通信機能モジュールで PE 割当機構において取得した実行情報を受信し，実行情報管理モジュールにおいて保存する．

“端末情報の取得・保存”では，取得する情報によって動作が異なる．プロセッサ抽象化 API から取得する情報の場合，通信機能モジュールで PE 割当機構において取得した端末情報を受信し，端末情報管理モジュールにおいて保存する．OS から取得する情報の場合，PE 選択機構は定期的に端末情報監視モジュールを呼出し，OS の端末情報を取得して保持する．

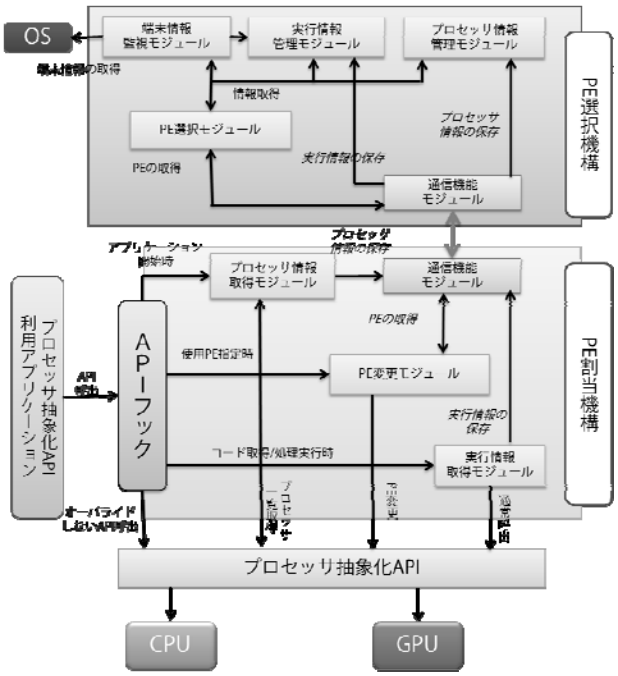

図 2　提案方式の構成

3．提案方式の Android 実装

3.1 実装環境

文献[1]の Linux での実装と同様に，プロセッサ抽象化 API には OpenCL を使用した．OpenCL では演算を行うプロセッサのことをデバイスと呼称するので，以降ではそのように呼ぶこととする．

Android での実装環境を表 1 に示す．OS のバージョンは 2.3.7 を用い，OpenCL の実行環境およびコンパイラには PGCL 12.8[4]を用いた．

3.2 実装の方針

文献[1]の Linux での実装と同様に，PE 割当機構はラッパーDLL，PE 選択機構は常駐アプリケーションとして実装を行った．

(1) PE 割当機構の実装

PE 割当機構は， C 言語で実装したラッパーDLL を用いる．PGCL は C 言語で実装されたライブラリであり，Android アプリケーションから JNI(Java Native Interface)を利用して OpenCL の API を呼ぶ．

この際に，ラッパーDLL のオーバライドされた API を呼出し，ラッパーDLL からオリジナルの OpenCL DLL の API を呼び出すことで API フックを行う．

(2) PE 選択機構の実装

PE 選択機構は，PE 割当機構からの問い合わせの待受や，端末情報の監視を行うため，Android フレームワークの Service クラスを利用して，常駐アプリケーションとして実装した．Service クラスの実装から利用する各モジュールは，文献[1]の実装と同一である．

(3) 機構間通信の実装

実装が容易で，スレッド単位での通信ができる UNIX ドメインソケット通信を用いた．

表 1　実装環境

OS	Android 2.3.7
API	OpenCL
実行環境およびコンパイラ	PGCL 12.8
本体型番	Sony Erricsson Xperia P (LT22i)
CPU	1 GHz Dual-core ARM Contex-A9
メモリ	RAM:1GB
チップセット	ST-Ericsson NovaThor u8500

(4) 並列プログラミングモデル

並列プログラミングモデルにはデータ並列とタスク並列があり，OpenCL には両方の API が用意されている．本研究は CPU と GPU で共に実行可能であるような処理を想定としている．GPU において，同時に異なる処理をするタスク並列を実行することは標準では困難であるため，データ並列について考える．

3.3 動作の流れ

提案方式のモジュール構成を図 3 に示す．配分するリソースを PE としたことに合わせて，文献[1]のモジュール構成の“リソース量配分モジュール”を“PE 選択モジュール”，“リソース量変更モジュール”を“ワークグループサイズ変更モジュール”に変更している他，モジュール名の“プロセッサ”を“デバイス”に変更している．

タスクが割り当てられる PE の動的な変更は，OpenCL 利用アプリケーションが行った API 呼出を Wrapped OpenCL API によってフックし，値を変更したのちにオリジナルの OpenCL DLL の API を呼び出すことによって行う．この一連の処理は，OpenCL によるタスクの実行までに行われる．

OpenCL によるタスクの実行までの処理の流れは次のようになる．OpenCL では，最初に仮想的な実行環境として 1 つ以上のデバイスに関連付けられたコンテキスト（cl_context）を生成し，これを使用してコマンドキュー（cl_command_queue）を生成する．このコマンドキューが，タスクが割り当てられる PE の変更に関係する．そして，タスクは生成した OpenCL のコマンドキューを介して実行される．

タスクが割り当てられる PE の変更の方法について説明する．OpenCL のデータ並列プログラミングモデルにおいては，各 PE にデータを並列に処理させるためのインデックス空間を持つことで，データ並列処理を実現している．

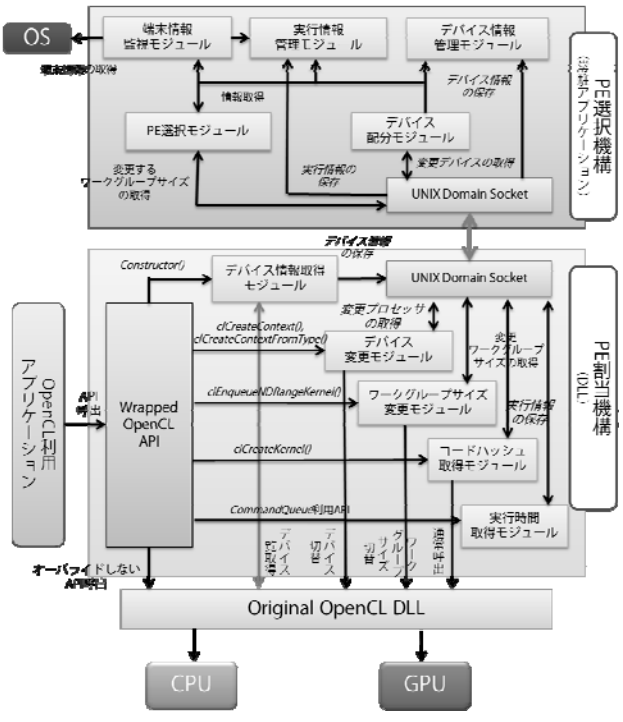

図 3　提案方式のモジュール構成

インデックス空間の 1 要素が，CPU の場合，データとそれを処理する軽量スレッドに対応する．実行インスタンスの単位をワークアイテムと呼び，この集合をワークグループと呼ぶ．OpenCL では，clEnqueueNDRangeKernel 関数の引数として，インデックス空間の次元数，ワークアイテムの総数，ワークグループサイズを指定することにより，インデックス空間を定義する．タスクへの PE の割当はインデックス空間に従って行われるため，これらのパラメータを動的に変更することによって，タスクが割り当てられる PE の動的な変更を行うことができる．パラメータのうち，インデックス空間の次元数とワークアイテムの総数は，実行するアルゴリズムとデータによって決定されるため，ワークグループサイズの動的な変更を行う．

3.4 具体的な動作

2.2 で述べた機能の本実装における具体的な動作を説明する．Linux での実装に対し，動作に変更はない．

3.4.1 PE への割当の変更

データ並列処理実行時に OpenCL の API で指定するワークグループサイズを変更することにより，PE へのタスクの割当の変更を行う．データ並列処理の実行 API は clEnqueueNDRangeKernel 関数である．この変更はカーネル関数毎に一度行う．

ただし，並列化するアルゴリズムによってインデックス空間の定義を変更すると計算が誤る場合がある（図 4）．ループのネスト数がデータの次元数であるアルゴリズムを並列化したコードの場合（a）には，インデックス空間の変更により PE 毎の処理量を変更が可能である．一方，ループのネスト数がデータの次元数×2 であるアルゴリズムを並列化したコードの場合（b）には，集合内の要素のデータ参照が必要となるが，インデックス空間を変更すると，データを示すインデックスが変更されてしまうため，計算に使用するデータを正しく参照できなくなってしまう．

本実装では，そのような場合にタスクが割り当てられる PE を変更しない．

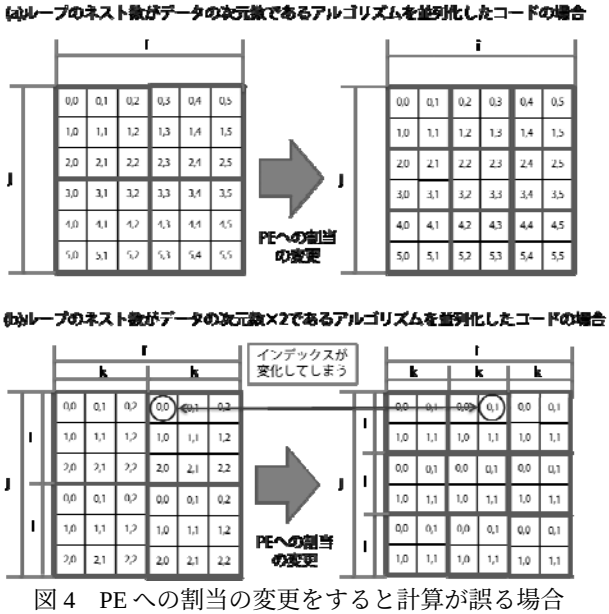

図 4　PE への割当の変更をすると計算が誤る場合

3.4.2 実行情報の取得・保存

実行情報には，処理にかかった実行時間，実行時のタスクへの PE の割当，ソースコード等が挙げられる．本実装では動作検証のため，実行時間とデバイスで実行するカーネルコードのハッシュの取得を行う．

3.4.3 端末情報の取得・保存

端末情報にはプロセッサの使用率，電源の接続状態や消費電力等の情報が挙げられる．本実装では動作検証にとどめ，CPU 使用率と電源の接続状態の取得を行う．

4．評価

表 1 の環境を用いて動作検証と性能評価を行う．

4.1 動作検証

既存アプリケーションとの互換性の確認のため，複数のアプリケーションを用いて動作検証を行う．検証は，タスクが割り当てられる PE の変更の可否と，実行情報および端末情報の取得の可否について行う．文献[5]や OpenCL フレームワーク実装ベンダである PGI，AMD，NVIDIA が提供しているサンプルコード 6 個を用いた．

いずれのサンプルコードも正常終了し，タスクが割り当てられる PE の変更，実行情報および端末情報の取得のすべてが可能であった．各 OpenCL フレームワーク実装のベンダが用意したライブラリを利用したサンプルコードで動作可能であったことから，これらのライブラリを用いて実装された多くのアプリケーションにおいて，本実装が利用可能であると考えられる．

4.2 性能評価

4.2.1 消費電力

(1) 方法

提案方式を導入した際の消費電力の変化を確認する．提案方式を用いずプロセッサ抽象化 API を利用して処理を行った場合と，提案方式を用いてタスクが割り当てられる PE の変更を行った場合について，電力が継続して消費される連続した処理を行うアプリケーションを想定した評価用コードにより消費電力の比較を行う．

評価用コードは，文献[5]のサンプルの FFT と，文献[7]のサンプルの DCT を用いる．以降，これらをそれぞれコード A，コード B と呼ぶ．コード A は，二次元画像を入力とする FFT のプログラムであり，ループのネスト数がデータの次元数であるバタフライ演算を並列化したコードである．コード B は，二次元画像を入力として 8×8 [pixel]のブロック毎の DCT のプログラムである．このプログラムは，5.4.1 で述べたループのネスト数がデータの次元数×2 であるアルゴリズムを並列化したコードであり，提案方式による PE の動的な配分をできないプログラムである．コード A では 512×512 [pixel]の画像，コード B では 64×64 [pixel]の画像を入力として評価用コードを実行する．

評価用コードの処理量の目安としてデータ入力速度を考え，一定の FPS（Frame Per Second）で画像データを評価用コードに入力し，消費電力を調べる．

計測は，バッテリを 100%まで充電した状態から 5 分間スリープさせたのちに評価用コードを 10 分間実行し，3 秒ごとに消費電力を計測する．FPS に対して処理が間に合わない場合には計測を中止する．Linux カーネルが提供している power supply class を利用して，端末全体の消費電力を計測する．

タスクが割り当てられる PE の初期値は使用するサンプルコードの初期値を用いた．また，タスクが変更される PE の選択は，予め評価環境におけるサンプルコードの値より高速な値を調査して用いた．

(2) 結果

結果を図 5 に示す．横軸が FPS，縦軸が実行中に計測した消費電力の平均値である．図中の値は，評価用コードを実行したときに計測した値から，評価用コードを実行していないアイドル状態のときの消費電力を引いた値であり，評価用コードを実行したことによる増加分を表す．以降，提案方式を利用しない場合をデフォルト状態と呼ぶ．

コード A の結果は次の通りである．全体としてデフォルトに比べ提案方式の消費電力が低かった．最大で約 49.16%（4FPS のとき）が削減された．また，デフォルトの場合では提案方式に比べて低い FPS の値までしか計測できていない．つまり，提案方式は，デフォルトに比べて多くの処理が行えることを示しており，4FPS から 7FPS で約 1.75 倍の処理性能の向上が見られた．これらの結果は，提案方式を用いることで PE に割り当てる処理量が調節され，無駄なコンテキストスイッチが削減されたことによる効果である．

コード B の結果は次の通りである．コード B は，提案方式を用いても，PE の配分による効果を得ることができないものであり，提案方式による介入や常駐アプリケーションの動作が発生するため，全体としてデフォルトに比べて高い消費電力となった．最大約 17.65%（34FPS のとき）の消費電力が増加している．しかし，コード A での消費電力の削減量が最大約 0.098VA（7FPS のとき）であるのに対し，コード B での消費電力の増加量は最大 0.016VA（34FPS のとき）と比較的小さいことがわかる．実行可能な FPS の値については変化が見られなかった．

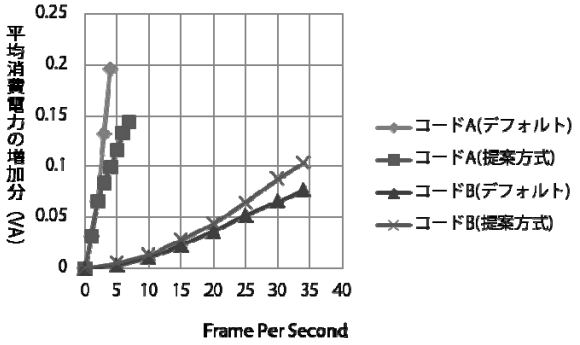

図 5　データ入力速度に対する平均消費電力

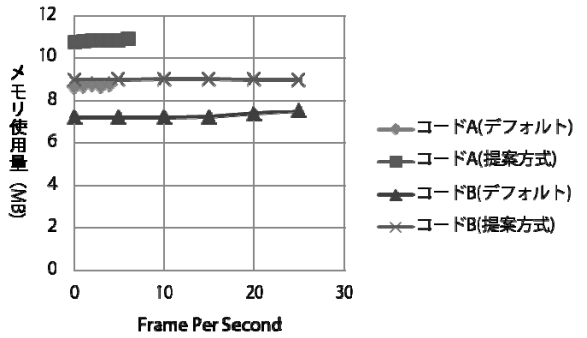

図 6　データ入力速度に対するメモリ使用量

同じ評価用コードを実行したときのメモリ使用量を図 6 に示す．提案方式を用いた場合にメモリ使用量が多くなっている．増加分のほとんどは PE 選択機構にあたる常駐アプリケーションのメモリ使用量であり，アプリケーションのメモリ使用量の増加は数十 KB 程度である．これはコード A とコード B のどちらの処理においても同じ傾向であった．

4.2.2　消費電力あたりの演算回数

(1)　方法

提案方式の PE の配分による消費電力あたりのパフォーマンスの変化を確認する．パフォーマンスの指標として，演算回数を用いる．

電力が継続して消費される連続した処理を行うアプリケーションを想定した評価用コードを用いて，提案方式により変更するワークグループサイズ毎に消費電力を計測する．そして，演算回数÷消費電力から消費電力あたりの演算回数を算出する．

評価用コードには，4.2.1 において省電力効果が得られたコード A を用いる．1 回の演算は 512×512[pixel]の画像を入力とした FFT 演算である．FPS は一定とし，評価用コードの初期ワークアイテムサイズにおいて継続して実行できる最大の値（7FPS）を用いた．消費電力の計測方法は 4.2.1 と同様であり，評価用コードを 10 分間実行した．

(2)　結果

結果を図 7 に示す．横軸がワークグループサイズ，縦軸が消費電力あたりの演算回数である．

ワークグループサイズが大きいほど消費電力あたりの演算回数は多くなり，ワークグループサイズが 1 と 16 の間で最大約 1.08 倍の向上が見られた．これは，提案方式によりワークグループサイズを適切に変更することで，消費電力あたりの演算回数を向上できることを示している．

4.2.3　実行時間

(1)　方法

評価用コードの実行時間は，OpenCL のコンテキスト生成からデバイスでの処理を開始するまでの起動時間とデバイスでの処理時間からなる．

PE 選択機構の常駐アプリケーションが起動しているときに，提案方式はデバイスでの処理を行う前に PE 選択機構との通信を行い，タスクが割り当てられた PE を変更する．提案方式はデフォルト状態に比べて，その処理に関わる時間が増加する．提案方式とデフォルト状態の起動時間

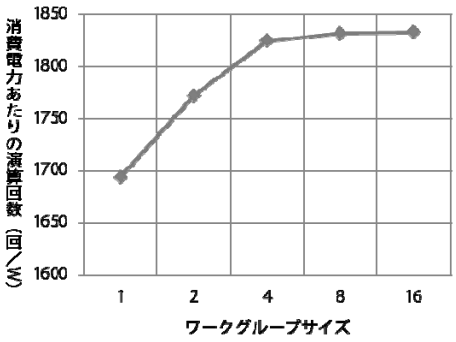

図 7　ワークグループサイズに対する消費電力あたりの演算回数

と処理時間の違いを調べ，提案方式のオーバヘッドを確認する．

評価用コードにはコード A を用い，512×512 [pixel]の画像を 1 枚入力した．

(2)　結果

結果を表 2 に示す．処理時間の削減量に比べて，起動時間の増加量が大きく，約 2 秒増加している．配分アルゴリズムの実装や DB の利用によって起動時間が更に長くなることを考慮すれば，提案方式は低頻度な処理では効果が低く，連続的に処理を行うアプリケーションに適している．

4.3　考察

4.2.1，4.2.2 より，ループのネスト数がデータの次元数であるアルゴリズムを並列化したコードにおいては省電力効果が得られ，消費電力あたりの演算回数を向上できることわかった．また，4.2.3 より，低頻度な処理の場合，提案方式を用いたタスクが割り当てられる PE の変更による処理時間の削減量に比べて，起動時間の増加量が大きいことがわかった．

これらの結果から，提案方式の適用先として，電力が継続して消費される連続した処理を行い，PE の配分に適したアルゴリズムによる動画像通信のアプリケーションを例として挙げることができる．それに対し，写真加工のように人が対話的に処理を行うようなアプリケーションでは，提案方式を用いた場合，消費電力や実行時間が増加するといえる．

常駐アプリケーションで API 呼び出しの監視と一部のコードの解析をすることで，その処理が連続的か対話的か，PE の配分が適しているかどうかは識別可能と考えられる．その識別結果に基づいて PE へのタスクの割り当てをすれば，一般的な利用状況においても省電力化を期待できるがその実現方法については今後の課題とする．

5．むすび

本論文では，著者らが提案している Linux 系 OS においてバイナリに変更を加えることなくプロセッサ抽象化 API 利用アプリケーションが使用する PE を動的に変更できる

表 2　提案方式を用いたことによるプログラムの実行時間

起動時間の差 [sec]	処理時間の差 [sec]	実行時間の差 [sec]
+2.0183	-0.0654	+1.9529

リソース配分機構の Android への適用を行った．提案方式を Android 上で OpenCL を用いて実装し，評価実験を行った．

複数のサンプルコードにおいて，提案方式は新たなランタイム環境の導入，プロセッサ抽象化 API のフレームワークの拡張，アプリケーションソースコードの改変をすることなく利用可能であることを確認した．提案方式はループのネスト数がデータの次元数であるアルゴリズムを並列化したコードにおいて省電力効果が得られ，消費電力あたりの演算回数を向上できることがわかった．

また，低頻度な処理の場合，提案方式を用いたタスクが割り当てられる PE の変更による処理時間の削減量に比べて，起動時間の増加量が大きいことを確認した．これらの結果から，電力が継続して消費される連続した処理を行い，PE の配分に適したアルゴリズムが含まれるアプリケーションにおいて，提案方式によるタスクへの PE の配分が有効であるといえる．

今後の課題としては，より一般的な利用状況において省電力化の効果が表れるように，PE の配分に適したアルゴリズムが含まれているかを常駐アプリケーションの識別と，PE の配分を決定するアルゴリズムの検討があげられる．

文　　献

[1] 榊原宏章，白石善明，岩田彰，“ 低消費電力化のための実行タスクの動的なプロセッサリソース割り当て機構 ”，情報処理学会研究報告，CDS，Vol.2013-CDS-6，No.38，pp.1-8，Jan，2013.

[2] Tetsuro Horikawa, Michio Honda, Jin Nakazawa, Kazunori Takashio and Hideyuki Tokuda, ”PACUE: Processor Allocator Considering User Experience”, Workshop on UnConventional High Performance Computing 2011,Euro-Par 2011,pp.335-344,Aug-Sep,2011.

[3] The Khronos Group, “OpenCL – The open standard for parallel programming of heterogeneous systems”, http://www.khronos.org/opencl/

[4] The Portland Group, “PGI OpenCL Compiler For ARM”, http://www.pgroup.com/products/pgcl.htm

[5]（株）フィックスターズ，土山了士，中村孝史，飯塚拓郎，浅原明広，三木聡，OpenCL 入門 -マルチコア CPU・GPU のための並列プログラミング-，インプレスジャパン，東京，2010.