卒業論文 2010年度(平成22年度)

PSEUDO:
バイナリ互換の動的使用プロセッサ変更機構

PSEUDO: Binary Compatible Dynamic Processor Switching System

指導教員

慶應義塾大学総合政策学部

徳田 英幸

村井 純

楠本 博之

中村 修

高汐 一紀

重近 範行

Rodney D. Van Meter III

植原 啓介

三次 仁

中澤 仁

武田 圭史

慶應義塾大学 総合政策学部

堀川 哲郎

techi@ht.sfc.keio.ac.jp

卒業論文要旨 2010年度(平成22年度)

PSEUDO: バイナリ互換の動的使用プロセッサ変更機構

近年のPCでは，マルチコアCPUに加えて汎用処理が可能なGPUを搭載する構成が増えている．GPUは並列性の高い処理を高速に実行でき，3Dゲームや動画再生を除いてあまり使われないため，新たな計算資源として注目されている．GPUを活用するため，現在ではGPUを汎用処理に用いるGPGPU(General-purpose computing on graphics processing units)のAPIが多数提供されており，GPUに加えてマルチコアCPUやCell等のプロセッサを抽象化し，同様に扱うことが出来るAPIも登場している．しかし，これらのAPIを通じて開発されたアプリケーションであっても，アプリケーションが使用するプロセッサを選択するため，PC全体では特定のプロセッサへ負荷が集中し処理速度を低下させる可能性がある．また，複数のGPUが搭載される環境においては適切なGPUが利用されない可能性があり，処理速度や消費電力等様々な面でユーザーエクスペリンスを低下させる可能性がある．現在に至るまで，HPC(high performance computing)分野ではこのような異種プロセッサ間において効率的な使用プロセッサの選択を行う研究が多数行われてきた．しかし，PC上のアプリケーションは容易に変更が出来ないという制約があり，PCにおいてユーザーエクスペリンスを高めるためには利用状況に応じた使用プロセッサ選択の必要があるため，HPC分野における研究をPCに直接適用することは出来ない．そこで，本研究ではプロセッサを抽象化するAPIを用いるアプリケーションに対し，バイナリを改変せずに動的なプロセッサの変更を実現し，PCの利用状況等の情報から使用プロセッサの選択を行うことを可能にする機構であるPSEUDOを提案する．動作の検証を行った結果，PSEUDOは多くのアプリケーションにおいて動的な使用プロセッサの切り替えを実現出来た．

キーワード：
プロセッサ切り替え，バイナリ互換，CPU, GPU, GPGPU, OpenCL, API-Hook

慶應義塾大学 総合政策学部
堀川 哲郎

Abstract of Bachelor's Thesis

PSEUDO: Binary Compatible Dynamic Processor Switching System

Due to spreading of high performance and general purpose GPUs, using GPGPU(General-purpose computing on graphics processing units) is becoming more and more important in PC applications. In these days, there are many APIs for GPGPU, also some processor abstraction APIs exist which enables utilizing heterogeneous processors similarly. Programming with these abstraction APIs make applications possible to choose a processor on execution, but there is a risk that processors' load biases if applications choose a processor individually. Many studies are performed in HPC(high performance computing) field concerning dynamic processor selection on heterogeneous processors environment, though applying these studies to PC is complicated, due to some restrictions in PC like impossibility of modifying applications. In addition, maximizing performance or minimizing power consumption does not always achieve good user experience, therefore it is almost impossible to simply apply those technique designed for HPC, to PC. To solve this problem, in this paper I propose the processor switching environment PSEUDO, which enables applications built with processor abstraction APIs to switch processor to utilize without any modification of application binary. As a result, PSEUDO satisfied processor switching on many applications.

Keywords:
Processor switching, Binary compatible, CPU, GPU, GPGPU, OpenCL, API-Hook

Tetsuro Horikawa

Faculty of Policy Management
Keio University

目次

図目次

表目次

ソースコード目次

第 1 章

序論

本章では，まず本研究の背景を述べ，次に本研究の概要を述べ，最後に本論文の構成について述べる．

1.1 本研究の背景

近年のPCには，マルチコアCPU（Central Processing Units）に加えて高い演算性能を持つGPU（Graphics Processing Units）が搭載されている．PCに搭載されるCPU（AMD64及びIntel 64アーキテクチャ）はマルチコア化が進んでいる．シングルコアCPUの処理能力は，「ポラックの法則」によりダイサイズのほぼ平方根に比例する．したがって，「ムーアの法則」に準じて「18ヶ月で集積回路上のトランジスタ数が倍になる」というペースを維持しても，ダイサイズが同じならば処理速度は18ヶ月で約1.4倍にしか増加しない．この数年間において，プロセスの微細化に伴うリーク電流増加が顕著になり，消費電力と比例関係にあるダイサイズを大きくすることは困難になった．そのため，シングルコアCPUのパフォーマンス向上は難しくなり，CPUベンダーはプロセスの微細化により可能となったトランジスタ増加分を「コア数を増やす」ことに用いて，処理能力の継続的向上を図っている．

CPUの性能向上が難しくなるなか，近年注目を集めているのがGPUである．GPUは元来ディスプレイ表示のためのプロセッサで，3Dゲームのフレームレートの向上，及び画質面におけるリアリティ追求という観点から性能を伸ばしてきた．GPUは特定のグラフィックス処理を高速に実行することが使命であり，固定回路によって構成されてきた．しかし，3Dグラフィックスにおいてリアリティが追求された結果，グラフィックス処理をプログラマブルにすることが求められ，シェーダープログラムを用いた柔軟なグラフィックス表現が可能になった．当初グラフィックス処理の一部のみをプログラム可能だったシェーダー機能は徐々に強力なものとなり，グラフィックス処理の大部分をプログラム可能になっている．このように，GPUのプログラマブル化が進んだ結果，現在のGPUは汎用処理を行うことが出来るようになり，汎用処理のための機能追加も積極的に行われるようになってきた．GPUは複数の画素を同時に扱う並列処理向きのプロセッサであり，数十個以上のコアを持つ究極のマルチコアプロセッサである．したがって，GPUはCPUとは異なりコア数を増やすことで容易にパフォーマンスを向上でき，現在その処理能力は2.7TFLOPs以上に到達している．

このようにマルチコアCPU・高性能GPUが普及したことで，PCのプロセッサ利用においては以下の 2 つを考える必要が出てきた．第一に，GPUを汎用処理に用いるGPGPU(General-Purpose computing on Graphics Processing Units)の活用が挙げられる．[29] [23] [36] [35] [33] 前述のようにGPUは並列処理を得意とするため，並列化が可能でかつ高い処理能力が求められる分野からGPGPUの普及が進んでいる．GPGPUは科学技術演算から普及が進み，近年スパコンへGPUが搭載される事例が増えているが，PCでは3Dゲームにおける物理演算，動画のエンコード処理等を中心に普及が始まっている．第二に，マル

チコアCPUとGPUの併用が挙げられる．[25] [34] [18] [20] [13] マルチコアCPUを用いる並列化とGPGPUを同時に利用することで，パフォーマンスを向上出来る．また，併用はPCの多様なプロセッサ構成に対応する目的のよっても必要となる．併用により，コア数が多く高性能なCPUと低性能なGPUを搭載するPCではCPUを利用し，一方低性能なCPUと高性能なGPUを搭載するPCではGPUを利用して処理を行う，といったことが可能になる．

現在GPGPU用途等の並列コンピューティング用APIは乱立しており，特にPCにおいてはまだ普及の初期段階である．そのため，それぞれのアプリケーションが個別に使用プロセッサとしてGPUを指定した場合でも，GPUに負荷が集中する可能性は低く，アプリケーションが個別に使用プロセッサを選択する弊害が現れる機会は少ない．しかし，今後より多くのアプリケーションがGPUを利用するようになった場合，GPUへの負荷集中が顕著となる可能性がある．また，使用プロセッサの選択によって近年のCPUやGPUが搭載する消費電力管理機能の動作を阻害する可能性があり，消費電力の増大や消費電力あたりのパフォーマンス低下を招く可能性がある．近年増加しているCPU若しくはチップセット内蔵GPUに加えて外付けGPUを搭載するPCにおいては，使用プロセッサの選択肢が増えるため，使用プロセッサの動的で適切な選択の重要性が高まる．

HPC(High-Performance Computing)分野においては，GPGPUとマルチコアCPUを搭載するコンピュータを対象とし，特定処理の高速化を図る研究や，CPU・GPU間で処理を分散させて処理の高速化や処理に必要な消費電力の低下を図る研究は既に多数行われている．HPCでは，利用するコンピュータの構成に合わせてプログラムをカスタマイズすることが容易であり，実行する処理はPCと比べると限定的である．したがって，アプリケーションの改変が困難なPCにおいては，HPC分野における研究を直接適用することは難しい．また，PC上で実行されるアプリケーションは多様で，ユーザーが実行するアプリケーションを制限出来ないという特徴がある．そのため，特定のアプリケーションを対象とした既存の手法をPCで利用することは困難である．これらの制約があるため，PC上で一般的に利用されるOSでは，アプリケーションが使用するプロセッサをCPU・GPU間で使用状況に応じて動的に分配することは，現在行われていない．

1.2 本研究の概要

以上のような背景を踏まえ，本論文ではPCにおいてアプリケーションが使用するプロセッサをPCの利用状況に応じて異種プロセッサ間で切り替えることを可能にし，より効果的なプロセッサ利用を実現する手法であるPSEUDOを提案する．

PSEUDOでは，CPU・GPUの双方を利用出来るアプリケーションを主な対象とし，アプリケーションのバイナリを改変せずに動的にプロセッサに処理を分配することを可能にする．PSEUDOはアプリケーションが指定するプロセッサを動的に変更する機能を提供することで，アプリケーションが使用するプロセッサを動的に変更する．

PSEUDOは既存のアプリケーションやランタイム環境，OSに至るまで，バイナリの改変を全く必要とせずに動作する．そのため，導入コストが極めて低く，コストパフォーマンスに優れている．

1.3 本論文の構成

本論文は以下の構成を取る．２章では本研究における問題意識を述べ，本論文の手法と既存の異種プロセッサ間の動的なプロセッサ選択手法と比較を行う．３章では，本研究の目標と研究の対象範囲について述べ，本論文におけるアプローチについて説明する．４章では，本論文で提案するPSEUDOの設計と動作概要について述べ，５章では，PSEUDOの実装について述べる．６章では評価方針と評価実験結果について述べ，考察を行う．７章では，本論文をまとめ，今後の展望について述べる．

第 2 章

PC の異種プロセッサ利用の課題

本章では，最初に本研究における４つの問題意識について述べる．次に，関連研究を列挙し本論文で提案する手法とを比較し，最後に本研究の問題意識についてまとめる．

2.1 問題意識

本節では本研究の問題意識について述べる．本研究で掲げる問題意識は４つあり，現状のプロセッサ利用法によって生じる弊害，PCのプロセッサ構成の多様性へ対応出来ていない問題，利用状況に応じたプロセッサ利用が行われていない問題，プロセッサ切り替え手法としてバイナリ互換が実現されていない問題がある．本節ではこれらの問題意識を順を追って説明する．

2.1.1 PC のプロセッサ利用方法

現在のPCでは，CPUはOSによって管理され，基本的には時分割によってスケジューリングされる．マルチコアCPUでは，CPUのコア単位で異なるアプリケーションを実行することが可能であり，OSはその管理を行っている．一方GPUはドライバにより管理されるため単独では利用出来ず，CPUをホストプロセッサとするコプロセッサとして動作する．

アプリケーションが使用するプロセッサをCPU・GPU間で選択する場合，その選択はアプリケーションによって行われる．例えば，GPUを利用するためのAPIを用いる場合，その処理は一部のエミュレーションモード等を除いてGPUで実行される．また，CPU・GPUの双方を利用出来るAPIを用いた場合にも，実際に使用するプロセッサを指定するのは各アプリケーションである．加えて，CPUのみで実行される従来のアプリケーションでは，その処理をGPU等の他のプロセッサで行うことは出来ない．以上のような理由により，OSは各アプリケーションが使用するプロセッサを指定・変更することが出来ない．そのため，現在PCを対象とする主なOS上では各アプリケーションが使用プロセッサの選択を各個行っており，特定のプロセッサに負荷が集中する等のリスクがある．したがって，OSが複数のCPU間でアプリケーションが利用するCPUを切り替えるように，CPU・GPU間においても利用状況に応じてプロセッサを切り替えられるようにすることが望ましい．

2.1.2 GPGPU(General-purpose computing on graphics processing units) とプロセッサ構成

GPUはCPUと比べ特に単精度浮動小数点演算において高速で，3Dゲーム中や動画再生等以外では利用される機会が少ないため，PCにおける新たな演算資源として注目されている．GPGPUは当初グラフィックス用APIを用いて行われたため，入出力を画像に見立てて行う必要があり，そのハードルは高かった．しかし，GPUベンダー等によりGPGPUを含む並列コンピューティングのためのAPI等が公開されたことで，GPGPUのハードルは低下している．

開発環境の整備が進んだことで，PC上でGPGPUを用いるアプリケーションが多数登場している．コンシューマー向けアプリケーションにおいては，動画変換・動画編集ソフトに加え，ゲーム内の物理演算部分等でGPGPUが用いられている．

現在では，GPGPUを利用可能なアプリケーションはGPUの利用が可能な場合はGPUを利用するように作成されているか，若しくはユーザーに使用プロセッサとしてGPUの選択を薦める場合が多い．しかし，PCによっては高性能なCPUと低性能なGPUを搭載する構成や，内蔵GPUと外付けGPUを切り替えて使用出来る構成等があるため，高性能なCPUと低性能なGPUを搭載する環境でGPUを利用した場合，実行速度が低下するといった問題が発生し得る．また，内蔵GPUと外付けGPU等複数のGPUを同時，または切り替えて利用出来る環境では，高速で消費電力の大きいGPUと低速で消費電力の小さいGPUのどちらを利用するかが問題となる．そのため，使用プロセッサ決定にあたっては，要求される実行速度や消費電力の制約を考慮すべきであり，アプリケーション側が高速なGPUを無条件で指定するような実装，あるいはユーザーの設定によって静的に使用GPUを設定するような実装では，利用状況に応じた適切なプロセッサの利用を行うことが出来ない．したがって，様々なPC上のプロセッサ構成や利用状況に対応するためには，各アプリケーションが使用するプロセッサを選択するのではなく，PCの利用状況に応じて使用プロセッサを割り当てる新たな主体が必要となる．

さらに考えなければならないのは，PC上ではCPU・GPUの双方を利用可能なアプリケーションのみではなく，従来のCPUのみを利用するアプリケーションや，3D表示のためにGPUを常に利用するアプリケーションが多数実行される点である．したがって，使用プロセッサを切り替えることが可能なアプリケーションと使用可能なプロセッサが固定的なアプリケーションの双方が多数実行されている状況において，使用プロセッサの切り替え方法を考えなければならないという問題がある．

2.1.3 利用状況に応じたプロセッサ利用

処理を行うプロセッサを決定する場合，処理速度を最大化する，消費電力を最小化する，消費電力あたりの処理能力を最大化する，といった方針が考えられる．しかし，PCにおいてはそれらの要素の最適化は必ずしもユーザーの希望とは限らない．例えば，GPUの機能を活用して動画を再生することで，CPUのみの場合よりもわずかに画質が向上するとする．ここで，CPUで処理する場合に比べてGPUで処理する場合のは方が高速に処理出来る処理が追加されるような状況を考える．もし，ユーザーが画質を向上させたいために追加された処理をCPUで行わせる場合，それはPC全体において処理速度を最大化したり，消費電力を最小化したり，あるいは消費電力あたりの処理能力を向上させたりはしていない．逆に，

ラップトップPCをバッテリー駆動している場合には，画質を犠牲にしてもバッテリー駆動時間を重視することが好まれる可能性がある．但し，ユーザーによっては鑑賞したい動画の総時間バッテリー駆動出来るの場合には，画質を優先してGPUを利用することを好む可能性もある．

このように，どのようなプロセッサを選択することが適切であるかはPCの利用状況とユーザーの嗜好によって大きく異なる．そのうえ，前述の動画再生支援機能のように，PCではGPU利用のためのAPIが多数存在し，同時に利用されている．したがって，様々なGPU利用形態を監視し，PCの利用状況やユーザーの嗜好に合わせてプロセッサの選択が行われる必要がある．

2.1.4 既存のアプリケーションとの互換性

PCで利用される主なOSでは，多数のソフトウェアベンダーがソフトウェアを提供しており，既存のソフトウェアが利用可能であるという互換性が強く求められる．PCのOS上で利用されるソフトウェアは，ユーザーが対価を払うことで購入している場合が多数あり，その場合ユーザー側の負担により更新が難しくなる．また，ソフトウェアベンダーにとっても新しいバージョンのOSとの間に起きる互換性問題のためにソフトウェアを更新するコストは高い．加えて，これらのOS上で利用されるアプリケーションは，クローズドソースで開発されているものが多数を占め，開発元以外が互換性問題を解決するのは困難である．

このようなプラットフォームの特徴から，OSやドライバーのベンダーはその更新によって既存のアプリケーションに互換性の問題が発生しないことを重視している．例えば，AppleのMacintosh OSにおいてPowerPC向けのバイナリをIntel x86アーキテクチャのCPUで実行するためのRosseta [19]や，MicrosoftのWindows 7において実装されたMinWin・Virtual DLLによるカーネル実装は，PCで要求される互換性の要求を端的に示している．

以上から分かるように，PCを対象とする場合には既存のアプリケーションの変更を必要としないことが重要である．使用するプロセッサを利用状況に応じて切り替える際も，新たにコンパイラやランタイム環境等を作成し，アプリケーションの改変を必要とする実装は，PCにおいて普及させることは難しい．したがって，今まで研究されてきたプログラムの改変や再設計を必要とするプロセッサの動的選択手法はPCに適用することは出来ない．

2.2 関連研究

本節では本研究の関連研究について述べ，本論文で提案する手法との比較を行う．まず，異種プロセッサ利用のためのプログラミング環境に関する研究を列挙し，次に動的な使用プロセッサ切り替えのための研究を列挙する．続けて，GPUの分割利用に関する研究につ

いて列挙し，最後に本研究の位置づけについて述べる．

2.2.1 異種プロセッサ利用のためのプログラミング環境

アプリケーション実装時に利用することで，異種プロセッサから使用プロセッサを選択して利用可能とするための研究としては，[24] [20] [18] [13] [31]等が挙げられる．以下にその一部について概要を説明する。

- OpenCL [13]
  OpenCLは，Khronos Groupによって策定された並列コンピューティングのためのフレームワークである．OpenCLでは，既存のGPGPUのためのフレームワーク等と異なり，GPUのみならずマルチコアCPUやCellプロセッサ等も計算資源として利用出来るため，単一のOpenCL C言語によるソースコードを元に様々なプロセッサで処理を行うことが出来るようになった．OpenCLでは，使用プロセッサの決定はソースコードの段階やコンパイルの段階では無く，アプリケーション実行時に行われるため，実行環境に存在するプロセッサの中から適宜使用するプロセッサを選択して利用することが出来る．OpenCLは標準化された規格であり，プロセッサベンダーによりサポートが表明され，OpenCLに対応するフレームワークが既に登場している．OpenCLにより，異種プロセッサを同様に扱うプログラムを作成することは容易になったが，個々のアプリケーションが実行時に使用プロセッサを決定しているため，PCの利用状況に応じてプロセッサの選択をすることは出来ない．さらに，OpenCLはプロセッサをCPUやGPU等のタイプに分類しているため，例えばGPUタイプのプロセッサを利用するように静的にコーディングされている場合，実質的に使用プロセッサが固定となる可能性がある．しかし，OpenCLは多数のハードウェアベンダーからサポートが積極的に行われており，今後の利用増加が見込まれる．そのため，今後の標準的なプロセッサの抽象化機構としてOpenCLは極めて参考になる．
- A balanced programming model for emerging heterogeneous multicore systems [31]
  A balanced programming model for emerging heterogeneous multicore systemsでは，OpenCL等に代わる新たなプログラミング手法を提案している．この研究では，今後普及が見込まれるGPUが統合されたCPU等のヘテロジニアスマルチコアプロセッサを視野に入れ，shared virtual memoryを用いることで，使用可能なプロセッサが変化する環境においても瞬時に使用プロセッサを切り替えて実行を可能にするフレームワークが提案されている．この研究ではIntelのLarrabeeが用いられているが，Larrabeeの一般リリースは見送られており，現在ではPCにおいて提案されている手法を用いることは困難である．しかしながら，AMDのFusionを含め，今後は使用可能なプロ

セッサが動的に増減するシステムの増加が見込まれるため，そのような環境における使用プロセッサの動的な切り替え手法として，提案されている手法は参考になる．

2.2.2 動的な使用プロセッサの切り替え

動的な使用プロセッサの切り替えを実現する手法としては，アプリケーション実装者の負担軽減を主目的としたもの，及びパフォーマンス最大化等の最適化を主目的としたものの 2 種類がある．

まず，アプリケーション実装者の負担軽減を主眼として，ライブラリを用いてプロセッサの動的な使用プロセッサを切り替え，自動的に使用プロセッサを決定する機構に関する研究としては，[25] [34] [32]等が挙げられる．以下にその一部について概要を説明する．

- SPRAT: Runtime Processor Selection for Energy-aware Computing [34]
  SPRATでは，CPUとGPUによるヘテロジニアスプロセッサ構成のPCを対象として，CPU・GPU間で消費電力あたりのパフォーマンスが最大になるプロセッサを自動的に選択して利用するランタイム環境を提案している．SPRATでは，CPU・GPU間におけるプロセッサの自動的な選択を実現しており，GPUを利用することで消費電力あたりのパフォーマンスが改善する場合に限りGPUを利用するため，極めて効率的である．しかし，SPRATでは著者の作成したランタイムを利用するようプログラムを改変する必要があるため，アプリケーションのバイナリ変更を必要としない本論文の手法と比べ導入コストが高い．また，消費電力あたりのパフォーマンスのみに焦点を当てているため，ユーザーの要求を考慮する必要があるPCには直接適用することは出来ない．しかし，使用プロセッサをCPU・GPU間で選択する際のアルゴリズムとして，SPRATの提案する手法は参考になる．

- Maestro: data orchestration and tuning for OpenCL devices [32]
  Maestroでは，CPUやGPU，FPGA等，3 つ以上のプロセッサを搭載するコンピュータを対象として，動的な使用プロセッサの変更を行い，OpenCL Cのソースコード等を変更せずに動的な最適化を行う機構を提案している．Maestroでは，主にメモリサイズの指定やホストプロセッサとコプロセッサ間のメモリ転送におけるダブルバッファリングを通じて実行速度の変化を計測し，適切な指定を行うことで実行速度の高速化を実現している．Maestroを用いる場合，デバイス毎にOpenCL Cのコードを最適化する必要がなくなるため，アプリケーション実装者は負担が軽減され，既存のOpenCLを利用したアプリケーションのソースコードの改変は少量で済む．しかし，プログラムの改変を必要とするため導入コストが高く，パフォーマンスの最大化のみ

を目的としている点からもPCへの適用は難しい．一方で，OpenCLのコードは実際にはハードウェア毎に最適化が必要であるため，Maestroのように動的な最適化を行う手法は今後重要性を増してくると考えられる．

次に，パフォーマンス最大化等の最適化を主目的とした使用プロセッサの切り替えアルゴリズムに関する研究としては[26] [30] [27]等が挙げられる．以下にその一部について概要を説明する．

- Predictive Runtime Code Scheduling for Heterogeneous Architectures [30]
  この研究では，コンピュータ上で実行されるほぼ全てのタスクがCPU・GPUの双方で実行出来る環境を想定し，CPU・GPU間におけるタスクスケジューリングアルゴリズムを提案している．結果として，GPUのみを利用する場合に比べ，30〜40%のパフォーマンス向上を実現しており，適切なアルゴリズムを用いてCPU・GPU間でタスクの分配を行うことが効果的であることを示している．しかし，コンピュータ上で実行されるほぼ全てのタスクがCPU・GPUの双方で実行される環境を想定しているため，近い将来のPCにおいてこの手法を適用することは困難である．しかし，使用プロセッサをCPU・GPU間で選択する際のアルゴリズムとして，Predictive Runtime Code Scheduling for Heterogeneous Architecturesの提案する手法は参考になる．

- Power Management of Multicore Multiple Voltage Embedded Systems by Task Scheduling [27]
  Power Management of Multicore Multiple Voltage Embedded Systems by Task Schedulingでは，ヘテロジニアスなマルチコアプロセッサを搭載し，プロセッサのDVFSが可能な組み込み機器を対象として，消費電力を最小化するためのアルゴリズムを提案している．ここで提案されているアルゴリズムは，組み込み機器であるためデッドラインを利用しており，PCには適用出来ない．しかし，現在のPCにおけるプロセッサ構成はヘテロジニアスなプロセッサ構成で各々のプロセッサがDVFS可能であるという点で良く似ている．このような構成において最適なプロセッサ利用を考える場合NP困難となるため，どのようにNP困難となることを解決するかというアプローチは参考になる．

2.2.3 GPU の抽象化と分割利用

GPUは，元来グラフィックス処理を専門に行うプロセッサであったため，CPUと異なりリソースの分割の柔軟性が乏しい．しかし，GPUをVMを通じて仮想化し用いる場合等は，GPUのリソースの管理が重要となる．このようなGPUリソースの仮想化に関する研究とし

ては，[28]等が挙げられる．

- GViM: GPU-accelerated virtual machines [28]
  GViMでは，通常VM上のアプリケーションではGPUに直接アクセス出来ないため利用出来ないGPGPUのAPIに対し，VM上で動く仮想のドライバを用意し，VMから利用出来るようにする手法を提案している．GViMでは，メモリ確保等をホスト上で動く管理アプリケーションが実際のGPUのドライバを通じて行い，さらに各VM上のGPU使用要求に対し，時分割による実GPUのリソース割り当てを行うことで，複数のVM上においてGPGPUの利用を可能としている．GPUのリソースはCPUのようにOSが細かく管理を行うものでないため，GViMのようリソース割り当てを行う管理アプリケーションを実装するという手法は極めて有効であると考えられる．

2.2.4 本研究の位置付け

関連研究は，以下の4つの内の1つまたは複数を実装し，各研究における目的の実現を図っている．

- プロセッサの抽象化を行うプログラミングモデル・実行環境
- 使用プロセッサの動的な切り替えを実現する機構
- 使用プロセッサの決定等リソース割り当てのアルゴリズム
- 使用プロセッサ毎にプログラムを最適化する手法もしくは機構

本研究では，上記の4項目の中で使用プロセッサの動的な切り替えを実現する機構及び使用プロセッサを決定するアルゴリズムを実装してPCにおけるバイナリ互換の使用プロセッサの動的な切り替えを実現する．この理由としては，プロセッサの抽象化を行うプログラミングモデル・実行環境は既に多数提案されていることが挙げられる．また，使用プロセッサ毎にプログラムを最適化する手法もしくは機構に関しては，効果を高めるためにはプログラム毎，ハードウェア毎の最適化が必要となるため，多様なPCのハードウェア構成，多数のプログラムにおいて動的な使用プロセッサ切り替えを目的とする本研究では，プログラムの最適化は研究の対象とはしない．

2.3 本章のまとめ

本章では，まず本研究の問題意識について述べた．問題意識として，まず現状のプロセッサ利用方法において異種プロセッサ間で使用するプロセッサを動的に切り替えることが出来ない問題について述べ，次にGPUを複数搭載するPCにおいて動的なプロセッサ選択を行

う重要性について述べた．そして，PCにおいては利用状況に応じたプロセッサ選択が必要であることを述べ，PCではアプリケーションにおける互換性が重視されることを説明した．次に，プロセッサの動的な切り替え機構に関する関連研究とプロセッサの切り替えアルゴリズムに関する関連研究を列挙し，最後に本論文で提案する手法と他の研究について位置づけの違いについて述べた．

第 3 章

目的とアプローチ

本章では，まず本研究の目的について述べ，次に本論文において対象とする範囲について述べる．その後本論文において提案する PSEUDO のアプローチについて述べ，最後に本章のまとめを行う．

3.1 研究の目的

本研究ではCPU及び汎用処理に対応するGPUを搭載するPCを対象とし，CPU・GPUの双方で処理を行うことが可能なアプリケーションに対し，使用するプロセッサを動的に変更可能にすることを目的とする．動的な使用プロセッサの変更にあたり，アプリケーションや既存のランタイム環境，OSの全ての改変を不要にし，導入コストを最小化する．また，PCの利用状況に基づいた使用プロセッサの選択を可能とするため，PCの利用状況を取得し，ユーザーの嗜好に基づいて各アプリケーションに使用プロセッサの指示を行うための主体を作成する．

目的となる要件を整理すると以下のようになる．

- 既存アプリケーションとの互換性
  前章で述べた通り，既存の研究ではCPU・GPU間で使用するプロセッサーを動的に選択する場合，動的な選択を行うアプリケーションの改変が必要であるか，若しくは実行するアプリケーションを専用に設計する必要があるという問題があった．本論文において提案する手法では，各アプリケーションの改変を必要としないだけでなく，既存のランタイムやOSの改変も不要とするため，アプリケーション等の開発者へ全く負担が掛からない．また，既存のアプリケーションに適用出来るため導入によって直ぐに効果が発揮され，アプリケーションの更新を必要とせずに段階的に適用出来るというメリットがある．

- CPUのみ利用可能なアプリケーションの考慮
  PCにおいては，CPUのみを利用可能な従来のアプリケーションが大多数を占める．これには主に３つの理由があり，第１に各アプリケーション内でGPGPU等の並列コンピューティングによって処理速度を改善出来る部分が限られており，実装が煩雑になるためそれらのAPIが用いられる可能性が低いということが挙げられる．第２に現在普及しているインストール済みのアプリケーションの大部分はCPUのみを利用可能な従来のアプリケーションであることが挙げられる．第３に，GPGPUのためのAPIは複数存在するが，エミュレーションモードを除いてCPUも利用可能なAPIが少ないことが挙げられる．

- 利用状況や嗜好に基づいたプロセッサの選択
  前章までに述べた通り，使用すべきプロセッサはPCの利用状況やユーザーの嗜好によって変化する．そのため，各アプリケーションが使用するプロセッサを決定する状況からPC全体の状況に基づいて使用するプロセッサが決定される状況に変える必要

がある．その際，多数のAPIによって利用されるGPUの情報等，使用プロセッサの決定にあたって必要なPCの使用状況を取得し，さらにユーザーの嗜好をデータベース化することで，利用状況に応じた使用プロセッサの選択を行う．これは，ユーザーの満足度をより高い水準で実現することを目指すという意味であり，QoS(Quality of Service)の向上が目標と捉えることも出来る．しかし，前述のようにPCでは特定のプロセッサの使用が必須となるアプリケーションが多く，使用プロセッサを自由に選択出来るアプリケーションの割合が大きく無ければQoSの向上は不可能である．本研究では，現実的なユーザーエクスペリンスの向上に繋がると予測されるパフォーマンスの最大化等の要求のみならず，ユーザーの興味のために特定のGPUを偏重して用いるといった要求も対象としており，リソース利用の公正性等はあまり考慮の対象とならない．

本研究では，CPUの利用率が高い場合にCPU・GPUの双方を利用可能なアプリケーションの使用プロセッサをGPUに切り替えて特定のプロセッサへの負荷集中を防止する等，PCを対象としてバイナリ互換性を維持したまま動的な使用プロセッサの切り替えを実現する機構であるPSEUDOを提案する．PSEUDOでは，複数のGPUを搭載する環境ではCPU・GPU間のみでなく個々のGPU間で切り替えも可能なため，PSEUDOを用いることでPSEUDOに対応するアプリケーションの割合が少ない状況から割合が多い状況まで，様々な状況で効果を発揮する．

3.2 研究の範囲

本研究では，対象とするコンピューターをPCに限定し，実装を行う．これは，HPC分野に用いられるコンピュータとPCとではプロセッサの構成が異なり，アプリケーションを改変したり，あるいは専用に設計することが認められるかといった制約も異なるためである．

また，プロセッサの変更を行う対象はCPU・GPUの双方を利用出来るアプリケーションとする．CPUのみで実行出来る既存のアプリケーションに対し，バイナリ・OS等を変更せずに他のプロセッサを用いることは非常に困難かつ高コストである．したがって本研究ではCPU・GPUの双方を利用出来るよう設計されたアプリケーションを対象として使用プロセッサの変更を行う．

本研究の最終的なゴールは，既存のアプリケーションが異種プロセッサ間で使用プロセッサを各々決定している状態から，PCの利用状況やユーザーの嗜好を元に各アプリケーションが使用するプロセッサを動的に切り替えて，ユーザーエクスペリンスを向上させることである．これを実現するためには，既存のアプリケーションを改変せずにプロセッサ切り替えを可能にする機構を作成すること，及びPCの利用状況やユーザーの嗜好を取得・

管理し，使用プロセッサを決定するアルゴリズムを作成することの二段階が必要となるが，本論文では切り替えを可能にする機構に的を絞って研究を行った．この切り替えを可能にする機構について，次節で実現のためのアプローチを述べる．

3.3 PSEUDO のアプローチ

本論文において提案するシステム，PSEUDOでは，前述の目的を実現するため，以下のようなアプローチを取る．

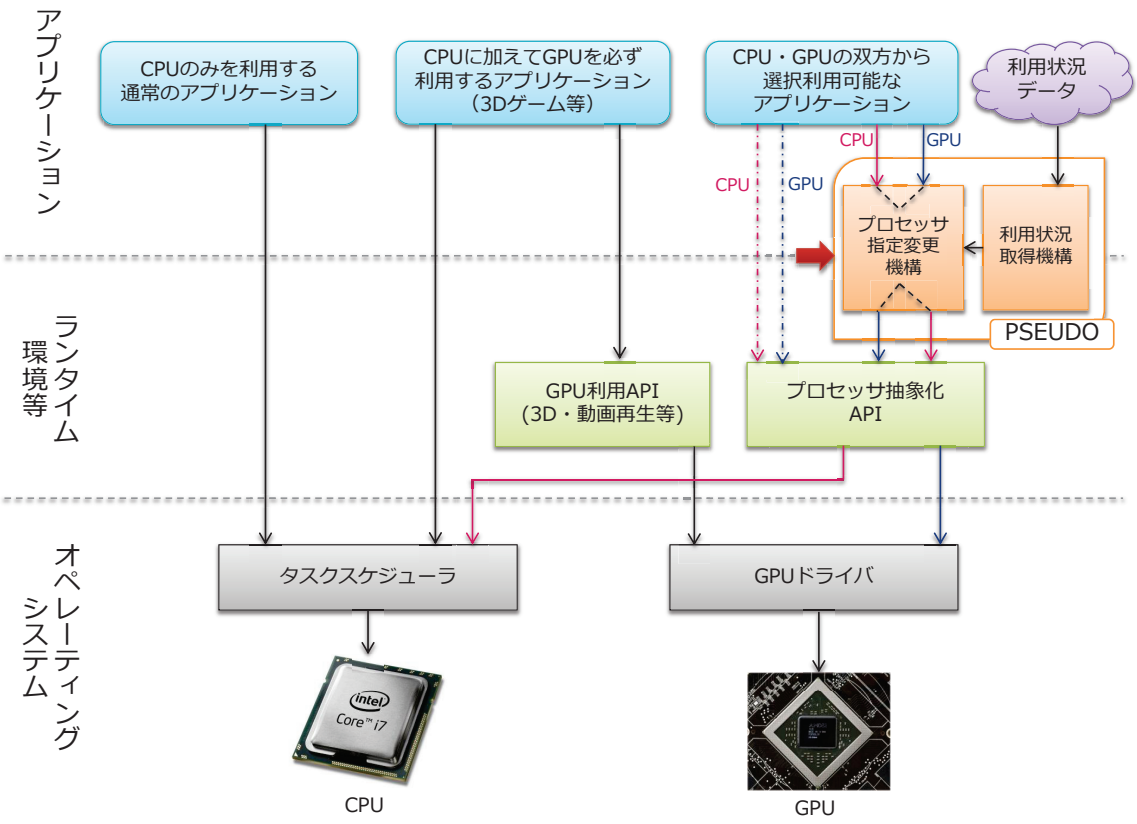


図 3.1 PSEUDO の動作概要

- アプリケーション・プロセッサ抽象化機構間への介入
  PSEUDOでは，既存のアプリケーションとの互換性確保のため，図3.1のように，CPUやGPUを抽象化して利用する機構を利用する機構とその機構を利用するアプリ

ケーションの間に介入し，使用プロセッサの指定を動的に変更する．プロセッサの抽象化機構に対する使用プロセッサの指示を通じて，実際に使用するプロセッサの変更が行われる．この介入にあたり，アプリケーションが利用しているAPIをオーバーライドすることで，アプリケーションや，プロセッサ抽象化機構，OSへの改変を必要とせずに動作する．また，プロセッサ抽象化機構を利用しないアプリケーションに対してはPSEUDOは何も干渉を行わないため，他のアプリケーションへの悪影響を最小化出来る．

- プロセッサ利用の監視・管理
  PSEUDOでは，使用プロセッサの選択の助けとするため，指定の変更を通じてどのプロセッサにどの程度のタスクを配分したのかを把握する．また，CPU利用率を取得することで，CPU使用率の高い場合にはGPUに優先的に処理を割り振る等，CPUのみ利用可能なアプリケーションを考慮したアルゴリズムの実装を可能にする．

- プロセッサ指定機構と利用状況を取得する機構の分離
  PSEUDOでは，使用プロセッサの指定を行う機構と利用状況等を取得する機構とを別個に用意する．そのため，利用状況等を取得する機構によってPC全体の情報や現在の利用状況等を考慮して使用プロセッサの指示を行い，指示に基づいて実際の切り替えが行われるという２段階の動作となる．このように，実際の指定を行う機構と利用状況の取得等を行う機構とを分離することで，ユーザーの嗜好をデータベース化する等の手法を用いて利用状況やユーザーの嗜好に基づいたプロセッサの選択アルゴリズムを組むことが可能になる．

以上の通り，PSEUDOは利用状況等を取得する機構と実際に使用プロセッサの切り替えを行う機構との２つによって構成される．機構の分離により，既存アプリケーションへの高い互換性とプロセッサ切り替え決定方法に対する高い柔軟性が実現される．

3.4 本章のまとめ

本章では，まず本研究の目的として，既存アプリケーションとの互換性を確保する，CPUのみを利用するアプリケーションを考慮したプロセッサ選択を行う，PCの利用状況やユーザーの嗜好に基づいたプロセッサ選択を可能とするという３つの項目を挙げた．続けて，はPCのみを対象とし，プロセッサの選択を行う際のアルゴリズムについては研究の範囲に含めないことを説明した．次に本論文において提案するPSEUDOのアプローチについ

て述べた．PSEUDOでは，アプリケーション・プロセッサ抽象化機構の間に介入するプロセッサ指定機構により既存アプリケーションのバイナリを変更とせずにプロセッサの切り替えを実現し，プロセッサ指定機構とPCの利用状況等を取得するモジュールを分離することで，CPUのみを利用する従来のアプリケーションやPCの利用状況等を考慮したプロセッサ選択アルゴリズムを作成可能な環境を提供する．

第 4 章

想定環境と設計

本章では，始めに本論文において提案する PSEUDO の概要について述べ，次に本研究の想定環境について述べ，その後に PSEUDO の設計について述べる．最後に本章のまとめを行う．

4.1 PSEUDO の概要

前述の通り，現在のPCにはマルチコアCPUとGPUが搭載され，CPU・GPUの双方を同等に利用可能なAPIが登場している．しかし，現状ではこれらのAPIを通じてアプリケーションが各々使用プロセッサを決定しているため，特定のプロセッサへの負荷集中等の問題がある．

PCにおいて利用状況に応じて使用プロセッサを動的に割り振る機構がPC上で利用されない理由として，既存の研究の大部分がHPCを対象としていること，PCにおいて重要な互換性を確保出来ていないことが挙げられる．

そこで，本論文ではプロセッサを抽象化しCPU・GPUの双方を利用出来るAPIを用いて実装されたアプリケーションを対象とし，アプリケーションのバイナリやAPIを提供する実行環境，及びOSの改変を必要とせずに使用プロセッサの切り替えを実現するPSEUDOを提案する．PSEUDOは既存のアプリケーションと互換性が高く，PSEUDOによるプロセッサ切り替えの対象とならないアプリケーションに対しては悪影響を及ぼさない．また，PSEUDOはGPUを複数搭載するような構成のPCにも対応でき，スケーラビリティが高い．さらに，PCの利用状況等を取得してプロセッサの切り替えに反映出来るため，PSEUDOを通じて高度な使用プロセッサ切り替えアルゴリズムを実装することが可能である．

4.2 想定環境

本節では，本論文における想定環境について述べる．最初に想定するPC上のプロセッサ構成，次にプロセッサ利用環境について述べ，最後に実行されるアプリケーションについて述べる．

4.2.1 プロセッサ構成

本研究では，近年のPCおけるプロセッサ構成を対象とし，以下のような条件が満たされていると想定する．

- マルチコアCPU
  本研究で対象とするコンピュータはPCであるため，搭載されるCPUは基本的に１ソケットでマルチコア構成であると仮定する．このようなPCでは，GPUのみならずマルチコアCPUを用いた並列処理を行い，高速に演算を行うことが出来る．

- 1つ以上のGPGPUに対応するGPU

現在販売されるPC向けの外付けGPUのほぼ全てが何らかのGPGPU用のAPIに対応し，GPGPUを利用することが出来る．チップセットやCPUに内蔵されるGPUはこの限りではないが，今後は内蔵GPUにおいてもGPGPU APIへの対応が進むと考えられるため，本研究ではGPGPUに対応するGPUが１つ以上搭載されるPCを想定する．また，高性能なGPUと低性能なGPU等，複数のGPUが搭載されるPCも想定に含める．

4.2.2 切り替え対象のアプリケーション

本研究では，動的に使用プロセッサをCPU・GPU間で切り替えることを目標とするため，CPUやGPUを抽象化し同様に扱うことが出来る実行環境がPC上にインストールされ，この実行環境により提供されるAPIを通じてアプリケーションがプロセッサを指定しているような環境において，使用プロセッサの動的な切り替えを実現する．

4.3 PSEUDO の設計

本節では，PSEUDOの設計について述べる．まず，使用するプロセッサの指定を変更する機構について説明し，次に利用状況等の取得を行い使用プロセッサの決定を行う機構について説明する．

4.3.1 プロセッサ指定・変更機構

PSEUDOでは，CPU・GPUの双方が利用可能な既存のアプリケーションについて，バイナリの変更を必要とせずにプロセッサの切り替えを実現する．また，OSや既存のCPU・GPUを抽象化して同様に利用出来る実行環境についても，改変を不要とする．この目的を達成するため，PSEUDOではアプリケーションがプロセッサを抽象化する実行環境のAPIを呼び出す際に，APIのフックを行うことでプロセッサの指定を変更する．（図4.1）プロセッサの指定は以下の３つのタイミングで変更を行う．

1. アプリケーションが使用可能なプロセッサ一覧の取得を試みた場合
   プロセッサが抽象化されている環境では，アプリケーションは使用するプロセッサを決定する前にPC上で使用可能なプロセッサの一覧を取得する必要がある．アプリケーションが使用するプロセッサはこの一覧の中から選択されることになるため，この時点で使用可能なプロセッサを絞り込んでおくことで，アプリケーションに対し本来使用予定であったプロセッサと異なるプロセッサを使用させることが出来る．ま

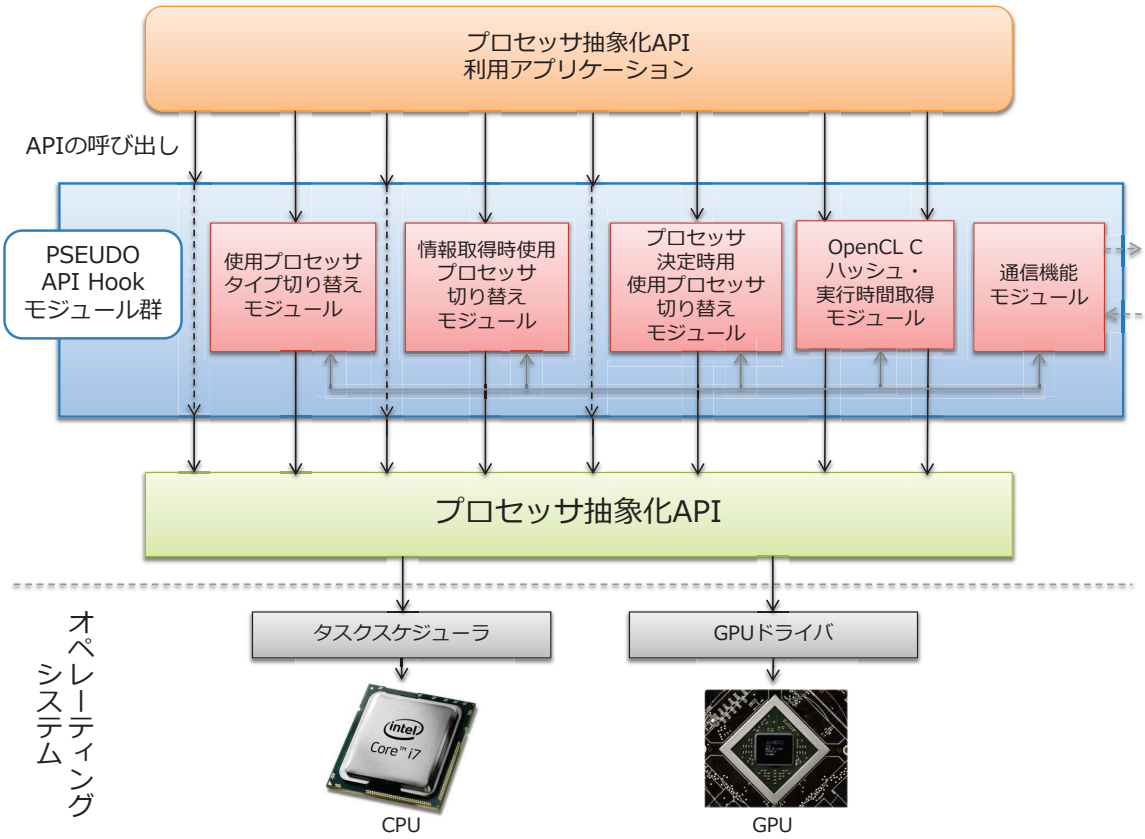


図 4.1　PSEUDO API Hook モジュール群の設計

た，アプリケーションがCPUやGPU等プロセッサの種類を既に絞り込んで一覧を取得しようとした場合に，全ての種類のプロセッサを対象としたり，あるいは指定と異なるプロセッサの種類を指定することによって，アプリケーションが使用するプロセッサを変更することが出来る．

2. アプリケーションが特定のプロセッサの詳細情報の取得を試みた場合
アプリケーションは，取得したプロセッサの一覧から実際に使用するプロセッサを決定するため，特定のプロセッサの詳細情報の取得を試みる場合がある．この時点でアプリケーションが問い合わせを行ったプロセッサ以外のプロセッサの使用が適切であると予測される場合，PSEUDOでは指定されたプロセッサのIDを変更し，アプリケーションの指定と異なるプロセッサの情報をアプリケーションに渡すことが出来る．アプリケーションはプロセッサの持つ特性を元に処理に用いるデータの並列性を最適化を行う等の使用プロセッサに合わせた動作の変更を行う可能性がある．そ

のため，実際に使用予定のデバイスの情報をアプリケーションに与えることは，アプリケーションの実行速度面及び実行安定性の面で重要である．

3. アプリケーションが使用するプロセッサを確定しようとした場合
   アプリケーションがプロセッサの一覧から使用するプロセッサを確定しようとした場合，それが使用するプロセッサの指定を変更出来る最後のタイミングとなる．ここで，アプリケーションが指定した以外のプロセッサ使用が適切であると判断された場合，PSEUDOは使用プロセッサの指定を変更する．この時点で使用するプロセッサは確定するため，その後の連続した処理は同一のプロセッサ上で実行される．但し，アプリケーションが新たな処理を行うために再度プロセッサを指定しようとする場合には，PSEUDOは再度プロセッサの指定を変更することが出来る．

前述の通り，プロセッサの指定・変更を行う機構と利用状況等の取得や実際の使用プロセッサの指示を行う機構は分離される．したがって，使用すべきプロセッサの問い合わせ等のために利用状況等の取得や実際の使用プロセッサの指示を行う機構との間で通信が必要となる．プロセッサの指定変更機構においては，主に上記のプロセッサの指定を行うタイミングで使用すべきプロセッサの問い合わせを行う通信機能を設ける．

4.3.2 利用状況等の取得機構

PSEUDOでは，APIのフックにより前述のタイミングでプロセッサの指定変更が可能である．しかし，複数のプロセッサからどのプロセッサを使用すべきかの判断はPC全体の状況を元に行う必要がある．各プロセッサの使用率を取得する等の方法を用いれば，アプリケーションが各々使用すべきプロセッサを判断することは可能であるが，その場合PCの細かな利用状況やユーザーの嗜好などを踏まえた高度なプロセッサ選択を行うことは出来ない．そこで，PSEUDOでは，プロセッサの指定を変更する機構とは別に利用状況等を取得し，使用すべきプロセッサの指示を行うアプリケーションを用意する．

本論文では，実際の使用プロセッサ選択アルゴリズムについては研究の対象としないため，詳細なPCの利用状況の取得等を行う機能については実装を行わない．そのため，PSEUDOにおいて使用すべきプロセッサの指示等を行うアプリケーション（図4.2）は以下の機能によって構成される．

- 使用すべきプロセッサの指示機能
  この機構においては，プロセッサの指定を行う機構からの問い合わせに応じて，使用

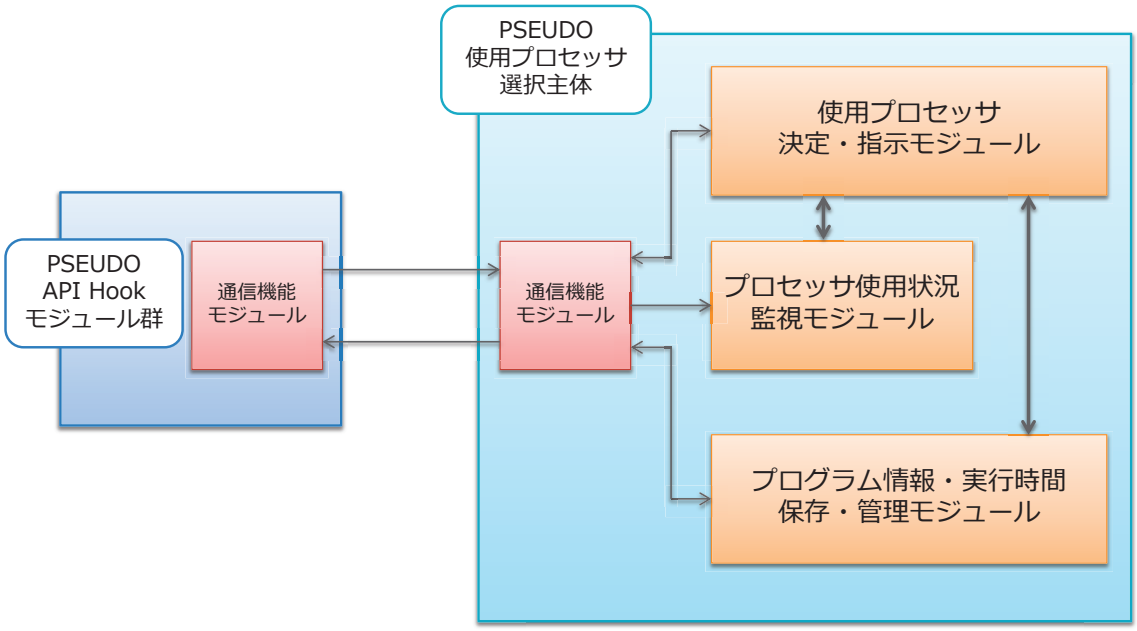


図 4.2　PSEUDO 使用プロセッサ選択主体の設計

すべきプロセッサの指示を行う必要がある．アプリケーションが使用可能なプロセッサ一覧の取得を試みた場合には，使用すべきプロセッサの種類について指示を行う．一方，特定のプロセッサの情報取得を試みた場合や使用するプロセッサを確定させようとした場合には，具体的に使用するプロセッサを指示する．

- プロセッサの指定によるプロセッサの使用状況監視機能
  使用するプロセッサを決定するためには，プロセッサの使用状況を取得する必要がある．そこで，この機構では自らが行った使用すべきプロセッサの指示を蓄積し，実際にどのプロセッサがどのアプリケーションによって使われているのかを把握する機能を設ける．これによって，その後の使用プロセッサ決定の助けとする．

- プロセッサの指定・変更機構との通信機能
  前述の通り，PSEUDOはプロセッサの指定・変更を行う機構とこの機構間で通信を行う必要がある．この機構では主に使用すべきプロセッサの返答のために通信機能を

用意する．

- プログラムの情報及び実行時間取得機能
  使用するプロセッサを決定するアルゴリズムを作成にするにあたり，各プログラムの処理を各プロセッサで行った場合の実行時間が必要になる．そのため，プログラム全体やプロセッサの切り替えが可能な部分の情報，実際の実行時間の取得を行い，前述の通信機能を通じてそれらの情報を送信する．（図4.1），プロセッサの指示主体が保存を行う．（図4.2）

4.4 本章のまとめ

本章では，まず本論文において提案するPSEUDOの概要について述べた．PSEUDOは，CPU・GPUの双方を利用出来るアプリケーションを対象とし，アプリケーションのバイナリ変更を必要とせずに動的な使用プロセッサを切り替える手法である．次に，本研究ではマルチコアCPUとGPUを搭載するPCにおいて，CPU・GPU等を同様に扱える実行環境が利用され，CPU・GPUの双方を利用可能なアプリケーションと従来のアプリケーションが並行して動作している環境を想定することを述べた．最後にPSEUDOの設計について述べ，PSEUDOがプロセッサの指定・変更を行う機構とPCの利用状況を取得しプロセッサを指定する機構によって構成されることを述べた．PSEUDOはこれらの機構の通信によって動作し，アプリケーションが使用可能なプロセッサの一覧を取得する際や，使用するプロセッサを確定しようとした際等に，使用すべきプロセッサを問い合わせ，プロセッサの変更を行う．また，プロセッサ選択のアルゴリズムを実装出来るように，プログラムの情報と実行時間の記録を行う．

第 5 章

PSEUDO 実装

本章では，まず PSEUDO の実装環境について述べ，次に実際の実装について説明を行う．最後に本章で述べた PSEUDO の実装についてまとめを行う．

5.1 PSEUDO の実装環境

本節では，PSEUDOの実装環境について述べる．まず実装に用いるOSについて述べ，次に実装に用いるAPI等について述べる．

5.1.1 OS

本研究では，Windows 7 x64 [22]を実装環境として用いる．実装環境としてWindowsを選択した理由は以下である．

- 高い普及率
  Windowsは，PCで使われるOSとして最も普及しているOSである．NetApplications社のTop Operating System Share Trend [15]では，2009年1月から2010年12月に至るまで，Windowsのシェアは90%を超えている．したがって，Windowsを実装環境とすることはPCを対象とする本研究の趣旨に合致している．

- 多数のGPU利用API
  Windowsでは，他のPC向けOSと比べて多数のGPU利用APIが利用出来る．例えば，Windowsのみが搭載するDirectX [7]は3Dゲーム等で利用されるAPI群であり，3Dや動画再生のアクセラレーション等で利用出来る．市販のPC向けグラフィックボードは，利用頻度の高いAPIであるDirectXに最適化されている．また，DirectXはGPGPUのためのAPIとしてDirectComputeを内包しており，GPUのベンダーを問わずに利用出来る．Windowsでは，これらに加えグラフィックス処理のためのGPU利用APIとしてOpenGL [14]等を，GPGPUのAPIとしてCUDA [4]やATI Stream [2]，OpenCL等を利用出来る．したがって，WindowsはGPUが様々なアプリケーションから様々なAPIを通じて利用される状況が発生しやすく，本研究の想定する環境と合致している．

- プロセッサ利用手法の多様性
  CPU若しくはチップセットに内蔵されるGPUと外付けのGPUを搭載するPCでは，PCの利用状況に合わせて用いるGPUを切り替えることが出来る．これらに用いられる技術として，GPUベンダーが提供するAMDのHybrid CrossFireX Technology [1]やnVidiaのOptimus Technology [16]，及び各PCメーカーが独自に実装する技術が存在する．このような内蔵・外付けGPU間の切り替え・協調動作のサポートはWindowsが

主であり，通常他のOSはサポート対象外若しくは限定的なサポートである．したがって，Windowsは様々なプロセッサ構成への対応が可能な点で本研究の実装環境として適している．

5.1.2 API 及びコンパイラ

本研究では，プロセッサの動的な選択を行う手法として，OpenCLのAPIを用いる．表5.1.2のように，CPU・GPUを含む複数のプロセッサを同様に扱うことが出来る並列コンピューティングフレームワークは少数で，そのうち標準化されているのはOpenCLのみである．OpenCLはプロセッサのベンダーを中心として，実行環境の整備が進んでいる．

OpenCLの実行環境であるフレームワークには，ATI Stream Software Development Kit (SDK) 2.2 With OpenCL 1.1 Support [3] (32bitバイナリ）を用いる．このATI Stream SDKでは，x86 CPUとAMD製のGPUを利用することが出来るため，CPU・GPU間の切り替えという本研究の目的に適している．

コンパイラを含む開発環境には，Microsoft Visual Studio 2010 Ultimate [12]を用いる．

フレームワーク名	CUDA	Sh [20]	OpenCL
CPU	△（エミュレーション）	△（エミュレーション）	○
GPU	△（nVidia 製 GPU のみ）	◎（多数の GPU に対応）	○
他のプロセッサ	×	×	○
提供者	nVidia	Intel	Khronos Group
標準化	×	×	○

表 5.1　並列コンピューティングフレームワークの比較

5.2 PSEUDO の実装

本節ではPSEUDOの実装について述べる．まず，アプリケーション等の改変を必要とせずに動作させるためのAPIフックについて述べ，次にプロセッサの指定・変更機構とプロセッサの指示を行う機構間の組み合わせについての説明を行う．その後，使用プロセッサの指定方法の実装について述べ，プロセッサ指定時のプロセッサ同定方法について説明する．続いて，使用プロセッサの指示を行いサーバーとして動作するアプリケーションについて述べ，サーバーとプロセッサ指定・変更機構の間における通信方法について述べる．その後，PSEUDOにおいて使用するプロセッサの変更を行うタイミングについて説明し，最後にプロセッサ切り替えのアルゴリズム実装のために必要なOpenCLコードのハッシュ取

得機能及び実行時間計測機能について述べる．

5.2.1 API フック機構

各アプリケーションやOpenCLフレームワークのバイナリを変更せずに使用プロセッサを動的に変更するため，PSEUDOではOpenCLのAPIをフックし，アプリケーションがOpenCLのAPIを呼び出す際にその動作をオーバーライドする．

開発環境であるWindows 7において利用可能なAPIフック手法としては主に以下の４種類が挙げられる．（表5.2.1）

	他プロセス上へのスレッド作成	グローバルフック	ラッパー DLL	API フックライブラリ
任意の API をフック可能	○	○	○	○
フックの確実性	△	○	◎	○
アプリケーション単位のフック	○	×	○	○
実装の容易さ	×	△	○	◎
マルウェア誤検知	×	○	○	△

表 5.2　API Hook 手法の比較

- 他プロセス上へのスレッド作成

  Windowsには，CreateRemoteThreadというAPIが存在し，このAPIでは別のプロセスのアドレス空間上で稼働するスレッドを作成出来る．このAPIを用いることで，任意のDLLを任意のプロセスにマップすることが可能になる．この方式では，DLL上の任意のAPIを任意のプロセス上で走らせることが出来るが，特定のアプリケーションのみを対象としてDLLのマップを行うため，複数のアプリケーションへの対応は煩雑である．この方式では，OpenCLを利用するアプリケーションが起動したタイミングでリモートスレッドを作成してAPIのフックを行わなければ，OpenCLのAPIのフックに失敗する可能性がある．

  またこの方式は，他のプロセスのメモリ空間上で任意のAPIを実行出来るため，キーロガー等マルウェアが自らを隠蔽する手法としてしばしば用いられる．そのため，この方式を用いるとウイルス対策ソフト等によりマルウェアと判定されるリスクが高い．

- グローバルフック

  グローバルフックはWindowsアプリケーションにおいてしばしば用いられるAPIフックの手法である．この方式では，SetWindowsHookExというAPIを利用し，実行中の全

てのプロセスにDLLをマップすることでAPIのフックを行う．グローバルフックはクリップボードの履歴を取得するアプリケーションやスクリーンショットを取得するアプリケーション等に広く利用されており，マルウェアと判定される可能性は低い．グローバルフックは全てのプロセスにDLLをマップする用途では便利であるが，一部のアプリケーションのみAPIをフックする必要がある本研究には適さない．また，対象となるプロセスを確実にフックするためには，グローバルフックを行うアプリケーションを常時起動させておく必要がある．

- ラッパーDLL
  ラッパーDLLを用いる方式は，DLL上に存在するAPIをフックする場合に適した方法である．この方式では，オリジナルと同名のDLLをラッパーとして作成し，オリジナルDLLをリネームしたものをラッパーDLLから読み込む．DLL上のAPIのうち，フックしたいAPIはラッパーDLL上に実装し，フックを必要としないAPIについては，オリジナルDLLのAPIを呼び出すことで動作する．
  この方式では，アプリケーションの起動時にラッパーDLLが必ず読み込まれるため，APIのフックが保証される．また，APIをフックしたいアプリケーションの実行ファイルと同じディレクトリにのみラッパーDLLを配置することで，APIのフックを行いたいアプリケーションにのみフックを行うことが可能となる．この手法はサウンドカードベンダーが3Dオーディオ機能の実現のために利用することもあるため，マルウェアとして検知されるリスクは低い．
  この方式では，オリジナルのDLLの持つAPIを全て持つDLLを作成する必要があるため実装が煩雑である．また，フックを行うアプリケーションの実行ファイル毎にラッパーDLLやオリジナルDLLを配置することは，同一のAPIをまとめて多数のアプリケーションから利用可能にし，ディスク使用量を削減するという本来のDLLの趣旨から外れてしまう．しかし，ディスク装置の容量増加は続いているうえ，シンボリックリンクを用いることでディスク使用量を抑えることも可能であり，あまり問題とはならないと考えられる．

- APIフックライブラリ
  Microsoft ResearchのDetours [5]に代表されるライブラリを用いてAPIをフックする方法がこの手法である．APIフックに特化したライブラリを用いるため実装が容易であるが，多数のAPIをフックする必要がある場合にはソースコードの記述量が増え煩雑になる．
  これらのライブラリは複数のAPIフック手法を提供するなど，APIのフック機能は充

実している．しかし，ラッパーDLLと異なり対象アプリケーションに対してAPIがフックされることは保証されない．実際のAPIフックの方法は，APIフックライブラリやAPIフックライブラリのAPIによって異なるため，マルウェアとして検知されるリスクは未知数である．

本研究では，アプリケーション単位で任意のAPIを確実にフック出来るラッパーDLL方式を用いて実装を行う．ラッパーDLLはOpenCLを用いるアプリケーションとオリジナルのOpenCL.dllの間で仲介として動作し，使用するプロセッサの動的な変更等を行う．

5.2.2 ラッパー DLL ＋サーバーアプリケーション

ラッパーDLLは，ラッパーDLLを呼び出したアプリケーションプロセスのメモリ空間上で動作するのみであり，単独では使用すべきデバイスを決定出来ない．CPU・GPU間における実行速度比やプロセッサの使用率を元に使用するプロセッサを決定することは可能であるが，その場合PC全体の利用状況に基づいて使用プロセッサを選択するという本研究の目的は達成出来ない．

そこで，PSEUDOでは使用プロセッサを決定する際，PCの利用状況を判別し使用すべきプロセッサの指示を行うサーバーアプリケーションに問い合わせを行うという形を取る．つまり，ラッパーDLLは使用プロセッサ指定のオーバーライド機能を提供し，使用プロセッサの決定はサーバーアプリケーション上で行われるということである．

ラッパーDLL＋サーバーアプリケーションという構成を取る場合，前節で挙げた他のAPIフックの方式と比べて使用プロセッサを問い合わせて切り替えを行うことが出来る確実性が高い．これは，DLLの読み込みによってPSEUDOのDLL内のAPIの実行は保証されるため，仮にサーバーアプリケーションが起動していない場合でもPSEUDOのDLLからサーバーアプリケーションを起動し，サーバーアプリケーションと通信を開始することが出来るからである．したがって，PSEUDOではこのラッパーDLL＋サーバーアプリケーションという形式を用いて実装を行う．

なお，本論文では以降ラッパーDLLについては“ PSEUDO DLL ”，サーバーアプリケーションについては“ PSEUDO サーバー ”（実行画面：図5.1）と呼称する．このPSEUDO DLL及びPSEUDOサーバーの実装は図5.2のように表される。

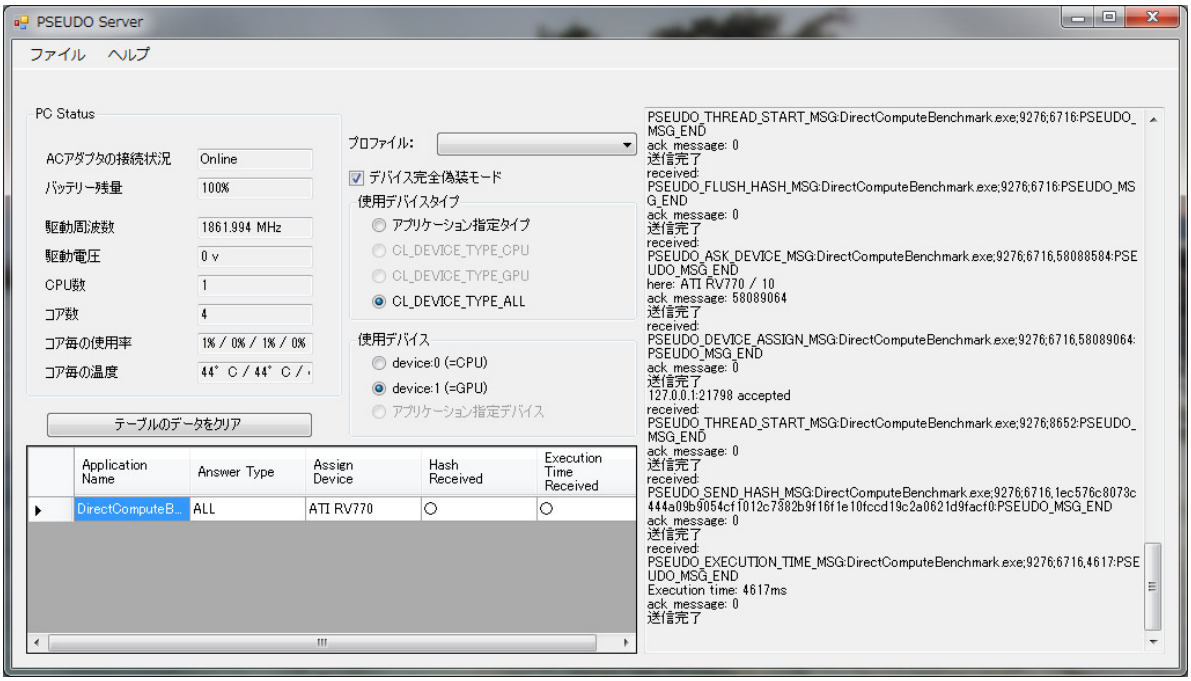


図 5.1　PSEUDO サーバーの動作の様子

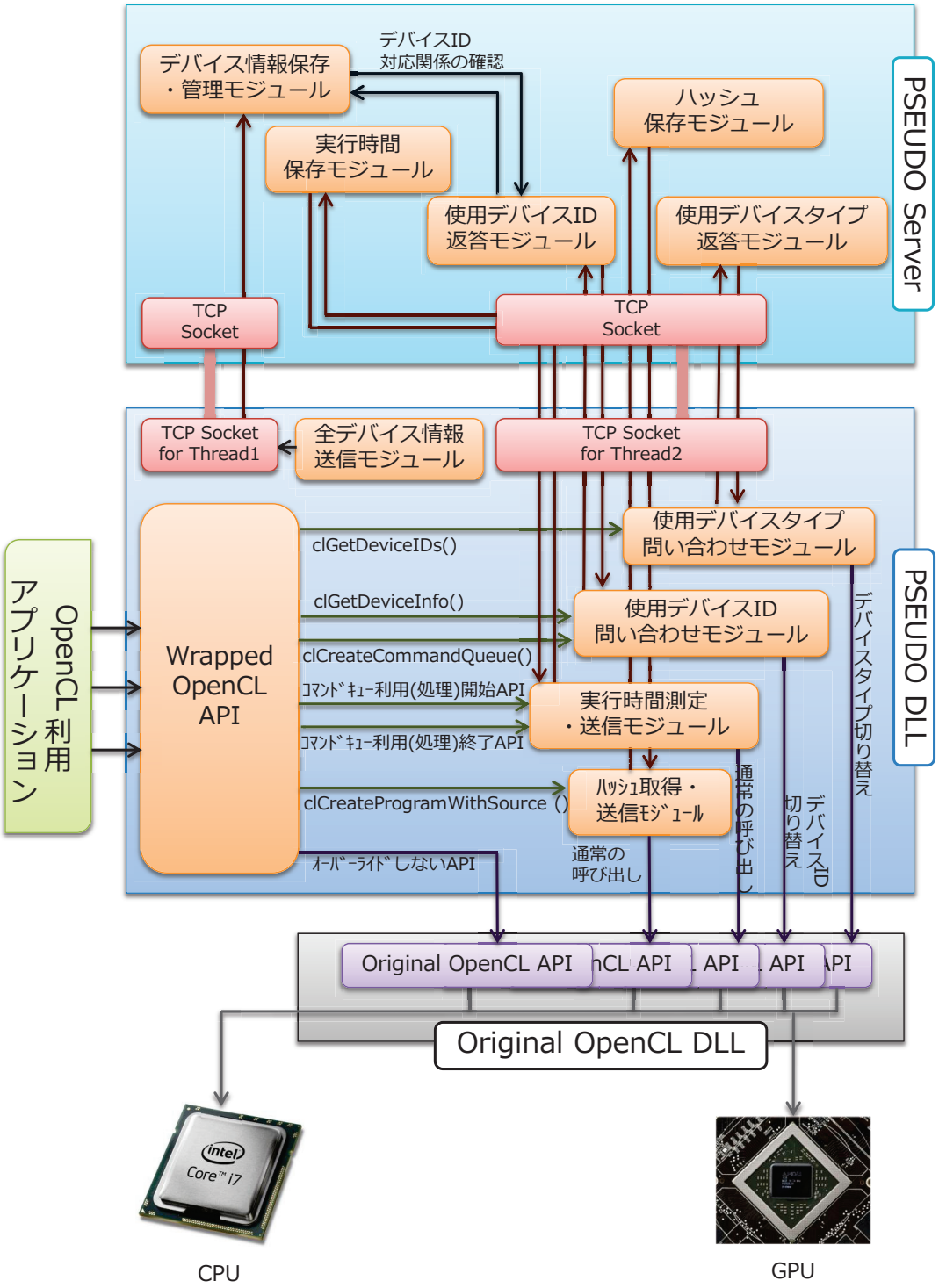


図 5.2　PSEUDO の実装

5.2.3 使用デバイスの指定

OpenCLでは，実際に処理を行うプロセッサをデバイスと呼称し，以下の粒度で指定を行うことが出来る．

1. PlatformID

   OpenCLの実行環境はフレームワークと呼ばれ，デバイスベンダー毎等の単位で複数のフレームワークが提供されている．例えば，AMD製GPUのためのOpenCLフレームワークとnVidia製GPUのためのOpenCLフレームワークは異なり，それらフレームワークの持つプラットフォーム情報は異なる．各フレームワークがICD(Installable Client Driver)に対応している場合は，複数のプラットフォームを切り替えて利用することが可能であるが，実際にはドライバのバージョン等により複数のプラットフォームを併用出来ない場合もあるため，今回はOpenCLにおいて複数のPlatformIDが存在する環境は想定せず，PlatformIDを用いてOpenCLフレームワークを切り替えることは行わない．

   OpenCLのプラットフォームはclGetPlatformIDs()（ソースコード5.1）というAPIにより取得出来る．

   ソースコード 5.1　プラットフォーム一覧の取得

```
1 cl_int clGetPlatformIDs(cl_uint num_entries,
2   cl_platform_id *platforms,
3   cl_uint *num_platforms)
```

   このAPIでは，num_entriesが取得するプラットフォームの最大数，platformsが取得したプラットフォームの出力先，num_platoformsが実際に取得出来たプラットフォーム数の出力先になっている．仮に，num_entriesに1を指定してAPIを呼び出し，PC上におけるOpenCLのフレームワークが複数ある場合，そのアプリケーションでは1番目に取得出来たプラットフォームしか使用することが出来ない．そのため，複数のプラットフォームが存在する環境を考えるならば，このAPIについてもPSEUDOによるオーバーライドが必要になる．

2. cl_device_type

   OpenCLにおいてデバイスを指定する手法として, cl_device_typeがある．cl_device_typeはOpenCLで利用するプロセッサのタイプを指定するもので，CL_DEVICE_TYPE_DEFAULT（デフォルトのデバイス）, CL_DEVICE_TYPE_CPU（CPUデバイス）, CL_DEVICE_TYPE_GPU

（GPUデバイス）, CL_DEVICE_TYPE_ACCELERATOR（アクセラレーターデバイス）, CL_DEVICE_TYPE_ALL（全てのタイプのデバイス）が定義されている．
デバイスの一覧は，clGetDeviceIDs()（ソースコード5.2）というAPIによって取得出来る．

ソースコード 5.2　デバイス ID 一覧の取得

```
1 cl_int clGetDeviceIDs(cl_platform_id platform,
2   cl_device_type device_type,
3   cl_uint num_entries,
4   cl_device_id *devices,
5   cl_uint *num_devices)
```

このAPIでは，platformにより対象とするプラットフォームを指定し，device_typeによって一覧に含めるデバイスの種類を指定する．num_entriesにて取得するデバイスの最大数を指定し，devicesに取得出来たデバイスID一覧，num_devicesに取得出来たデバイス数が出力される．
本研究では，OpenCLデバイスとしてCPU及びGPUを利用するため，アプリケーションが指定しているcl_device_typeをオーバーライドし，device_typeにCL_DEVICE_TYPE_CPUやCL_DEVICE_TYPE_GPU，あるいはCL_DEVICE_TYPE_ALLを指定することで，使用するプロセッサタイプの変更を行う．本研究ではPCを対象としているため，PCに搭載される頻度の低いアクセラレーターデバイスは主な対象とはしないが，PSEUDOの手法はアクセラレーターにおいても適用出来る．

3. cl_device_id

OpenCLにおいてデバイスを指定する最も具体的な指定方法がcl_device_idを用いる方法である．cl_device_idを用いることで，デバイスを一意に確定し指定することが出来る．但し，cl_device_idは仕様上常に同一の値であるとは限らないため，アプリケーションの実行毎に異なる値が設定されることがある．
本研究では，cl_device_idを用いて使用プロセッサの指定を行うが，clGetDeviceInfo()（ソースコード5.3）を用いてcl_device_infoを取得し，デバイスの情報から同一のデバイスを検索することでcl_device_idの値が異なる場合であっても同一のデバイスであると同定する．

ソースコード 5.3　デバイス情報の取得

```
1 cl_int clGetDeviceInfo(cl_device_id device,
2   cl_device_info param_name,
```

```
3    size_t param_value_size,
4    void *param_value,
5    size_t *param_value_size_ret)
```

このAPIでは，deviceに情報を取得したいデバイスIDを指定し，param_nameに取得したい情報名（デバイス名等）を指定する．取得出来た情報はparam_valueに出力される．

5.2.4 デバイスの比較

PSEUDOでは，起動毎に変わるIDと実際のデバイスとの対応を確認するため，clGetDeviceInfo()（ソースコード5.3）のparam_nameにCL_DEVICE_NAME（デバイス名）及びCL_DEVICE_MAX_COMPUTE_UNITS（デバイスが同時に処理を行うことが出来るユニット数）を指定・取得してデバイスの判別を行う．これによってデバイスのIDが異なる場合にもデバイスの同定が可能になる．
PSEUDOでは，同一のCL_DEVICE_NAME・CL_DEVICE_MAX_COMPUTE_UNITSを持つ複数のグラフィックボードが搭載される環境では，それらのGPU間においてGPUの同定が出来ないため切り替えを行うことが出来ない．各グラフィックボードの装着されているスロット情報等を用いることでグラフィックボードの区別を行うことは可能であるが，それらの情報により取得したGPUとOpenCLから取得出来るデバイスに対し１対１対応を行うことは困難なため，OpenCLから取得出来るデバイス情報が同じGPU間ではGPUの同定を行わない．

5.2.5 サーバーアプリケーション

本研究では，各アプリケーション単位ではなくPC全体でプロセッサ間のタスク分配を実現することを目的としている．そのため，各アプリケーションに代わって使用プロセッサを指定する主体が必要となる．PSEUDOでは，各アプリケーションと通信を行い，使用すべきプロセッサを指示するサーバーアプリケーションであるPSEUDOサーバーを実装し，各アプリケーションの使用するプロセッサを指示する．
PSEUDOサーバーでは，OpenCLのデバイス一覧を保持し，各プロセスの持つデバイス情報・デバイスIDとマッチングを行う．また，各アプリケーションに対して使用すべきプロセッサを指示し，各アプリケーションが使用しているプロセッサの情報を保存する．
PSEUDOサーバーは独自に複数のアプリケーションを管理する必要があるため，プロセス間通信を行う必要がある．プロセス間通信を通じて，各アプリケーションから使用すべきプロセッサを問い合わせる要求を受け入れ，使用すべきプロセッサを返す．また，各アプ

リケーションに既に使用プロセッサの指定を行った後に使用プロセッサの変更が必要になった場合，プロセス間通信を通じて各アプリケーションに指示を行う．

5.2.6 プロセス間通信

PSEUDOサーバーとPSEUDO DLLはプロセス間通信を通じて使用すべきプロセッサの問い合わせやデバイス情報のマッチングを行う．

実装環境であるWindows 7におけるプロセス間通信の方式としては，OLE，名前付きパイプ，共有メモリ，メモリマップドファイル，WM_COPYDATA等が利用出来る．しかし，OpenCLを利用するアプリケーションは複数のスレッドを作成しデバイスを使用する可能性があるため，PSEUDO DLLはスレッド単位でPSEUDOサーバーと通信する必要がある．そのため，パフォーマンス上不利になるもののスレッド単位での通信が容易なTCP Socketを用いてプロセス間通信を行う．

5.2.7 デバイス変更のタイミング

OpenCLでは，仮想的な実行環境の単位としてOpenCLコンテキストを作成・利用する．OpenCLコンテキストは，clCreateContext()(ソースコード5.4）のAPIによって作成でき，１つ以上のデバイスを指定することが出来る．

ソースコード 5.4　OpenCL コンテキストの作成

```
1  cl_context clCreateContext(
2    const cl_context_properties *properties,
3    cl_uint num_devices,
4    const cl_device_id *devices,
5    (voidCL_CALLBACK  *pfn_notify) (
6              const char *errinfo,
7              const void *private_info, size_t cb,
8                          void *user_data
9  ),
10   void *user_data,
11   cl_int *errcode_ret)
```

このAPIでは，num_devicesにコンテキストで使用するデバイス数，devicesに使用するデバイスを指定する．そのため，プロセッサを自由に選択出来るよう，このAPIをアプリケー

ションが呼び出す際にはPSEUDOはnum_devicesを全てのデバイス数に，devicesを全てのデバイスに変更してコンテキストを作成する．

一方, 実際にOpenCL Cのコードをデバイスに投入して実行する際には, clCreateCommandQueue()（ソースコード5.5）を利用し，特定の１つのデバイスを利用するためのコマンドキューを作成してそのキューに入れる形を取る．

ソースコード 5.5　コマンドキューの作成

```
1 cl_command_queue clCreateCommandQueue(cl_context context,
2   cl_device_id device,
3   cl_command_queue_properties properties,
4   cl_int *errcode_ret)
```

このAPIでは，contextにOpenCLコンテキストを，deviceに使用する１つのデバイスを指定し，使用するcommand_queueを返り値で得る．OpenCLの仕様上コマンドキューが利用されている最中にコマンドキューの対象デバイスを変更することは出来ないため，PSEUDOではコマンドキューが作成されるタイミングでサーバーに使用プロセッサの問い合わせを行い，deviceに使用すべきデバイスのIDを指定する．

5.2.8　デバイス一覧の取得

PSEUDOでは，アプリケーションの指定と異なるデバイスを扱う場合があり，前述のclCreateContext()において全てのデバイスを指定する際やデバイス名とデバイスIDの対応を調べる場合等において全てのデバイス情報が必要となる．そのため，PSEUDO DLLはアプリケーション起動時にPC上で利用可能な全てのデバイス情報を取得し，PSEUDOサーバーに送信する．また，PSEUDO DLLによって全てのデバイスIDを保持しておくことにより，アプリケーションが利用可能なデバイスとしてIDを取得していないのデバイスの利用を可能にする．

5.2.9　スレッド単位の情報管理

OpenCLアプリケーションは，複数のデバイスを使うため，あるいはその他の理由で複数のスレッドからDLLの関数を呼び出す可能性がある．このとき，DLL内のグローバル変数等，DLL内で情報がスレッド間で共有されてしまう可能性がある．これにより異なるスレッド間で干渉が起こり，正しくプログラムが動作しない可能性がある．そこで，PSEUDOではスレッドローカルストレージ（TLS: Thread Local Storage）を利用することで，スレッド間で共有すべきでない変数を保護する．

TLSの利用にあたっては，__declspec(thread)キーワードを利用して記述を行った．（ソースコード5.6）

ソースコード 5.6　TLS の例

```
1 __declspec(thread) cl_context n_context = NULL;
```

TLSによりスレッド単位で情報が分割されることで，PSEUDOではOpenCLを利用するアプリケーションが複数のスレッドを作成してOpenCLのAPIを呼び出した場合に異なる複数のデバイスを利用して処理を行うことが出来るようになる．そのため，プロセッサ間で負荷を分散させることが容易になるほか，処理を行うデバイスを複数にすることによるパフォーマンスの向上が期待出来る．

5.2.10　OpenCL C コードのハッシュ取得

PSEUDOでは，OpenCL Cにより書かれたプログラムのソースコードについて，ハッシュを取得する．これは，OpenCL Cのコードと使用したデバイスにおける実行時間とを対にして保存することで，PSEUDOサーバーが次に同じOpenCL Cのコードを実行する際の使用デバイス決定の補助とするためである．本論文では，使用するデバイスの決定アルゴリズムに関しては研究の対象には含めていないが，アルゴリズムの作成を考えた場合，実行コードと実行時間といった情報が必要になると考えられるからである．

PSEUDO DLLは，clCreateProgramWithSource()（ソースコード5.7）のAPIをフックすることで，OpenCL Cのソースコードを全て取得することが出来る．

ソースコード 5.7　OpenCL カーネルのソースからのプログラム作成

```
1 cl_program clCreateProgramWithSource (cl_context context,
2   cl_uint count,
3   const char **strings,
4   const size_t *lengths,
5   cl_int *errcode_ret)
```

このAPIでは，contextに対象のOpenCLコンテキスト，stringsにOpenCL Cで書かれたカーネルのソースコードを指定することで，OpenCLのプログラムが作成される．このとき，stringsからソースコードを全て取得し保存するとOpenCLを用いるアプリケーションのライセンスに抵触したり，著作権を侵害する可能性が高い．そこで，PSEUDO DLLではソースコードのハッシュのみをPSEUDOサーバーに送信する．ハッシュのみを保存する場合，ソースコードの解析が出来なくなり実行速度の予測面で不利になるが，一方で

PSEUDO DLL・PSEUDOサーバー間で転送されるデータ量を低減出来るというメリットがある．以上を踏まえ，PSEUDOではOpenCL Cのソースコードからハッシュを計算し，PSEUDOサーバーに送信する．

5.2.11 実行時間の計測

前述の通り，PSEUDOではOpenCL Cコードのハッシュと実行時間を対で保存する．しかし，PSEUDO DLLはラッパーDLLであり，OpenCLのAPIが呼び出されたタイミングでのみ時間の計測が可能である．

したがって, PSEUDO DLLでは正確な実行時間の計測は不可能であるが, clEnqueueWriteBuffer()（ソースコード5.8），clEnqueueNDRangeKernel()（ソースコード5.9），clEnqueueTask()（ソースコード5.10）等のコマンドキューを利用するAPIが最初に呼ばれた時点を開始時刻とし，キューの全てのタスクを実行するclFinish()（ソースコード5.11）またはコマンドキューを解放するclReleaseCommandQueue()（ソースコード5.12）の終了時点を終了時刻として実行時間の計測を行う．PSUEDO DLLはこの実行時間をPSEUDOサーバーに送信する．

ソースコード 5.8　バッファ領域へのデータの書き込み

```
1 cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
2     cl_mem buffer,
3     cl_bool blocking_write,
4     size_t offset,
5     size_t cb,
6     const void *ptr,
7     cl_uint num_events_in_wait_list,
8     const cl_event *event_wait_list,
9     cl_event *event)
```

ソースコード 5.9　データの並列実行

```
1 cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
2     cl_kernel kernel,
3     cl_uint work_dim,
4     const size_t *global_work_offset,
5     const size_t *global_work_size,
6     const size_t *local_work_size,
7     cl_uint num_events_in_wait_list,
```

```
8       const cl_event *event_wait_list,
9       cl_event *event)
```

ソースコード 5.10　タスクの実行

```
1  cl_int clEnqueueTask (cl_command_queue command_queue,
2       cl_kernel kernel,
3       cl_uint num_events_in_wait_list,
4       const cl_event *event_wait_list,
5       cl_event *event)
```

ソースコード 5.11　投入された全てのコマンドをデバイスで実行

```
1  cl_int clFinish (cl_command_queue command_queue)
```

ソースコード 5.12　コマンドキューの参照カウントを 1 減らし，0 になった場合オブジェクトを破棄

```
1  cl_int clReleaseCommandQueue(cl_command_queue command_queue)
```

5.3　本章のまとめ

本章では，まずPSEUDOの実装環境について述べ，実装環境としてWindows及びOpenCLを選択した理由について説明を行った．その後，PSEUDOの実装について詳細な説明を行い，PSEUDOがフックするAPIについてそのソースコードを提示した．PSEUDOでは，デバイスのタイプやIDを指定するAPIをフックすることで使用デバイスの動的な変更を実現し，OpenCL Cソースコードのコンパイルを行うAPIを呼び出した際にソースコードのハッシュ取得を行う．また，PSEUDOはOpenCLコマンドキューへタスクを入れるタイミングで実行時間の計測を始め，コマンドキューのタスクが空になった段階，またはキューの利用が終わった段階で計測を終了することで，OpenCLの実行時間を計測する．

第 6 章

PSEUDO 評価

本章では，始めに本研究で提案した PSEUDO の評価方針と評価環境について述べる．次に評価結果について述べ，評価結果を踏まえた考察を行う．最後に本章のまとめを行う．

6.1 評価方針

本研究では，複数のアプリケーションについてPSEUDOの利用可否を調べ，アプリケーション毎に動作の検証を行う．

まず，本研究の最大の目標である既存アプリケーションとの互換性の確認のため，以下の観点から評価する．

- アプリケーションの正常動作
  アプリケーションがクラッシュやエラーを起こさずに動作出来たかを検証する．

- デバイスタイプの切り替え可否
  各アプリケーションに対し，デバイスのタイプ(CPU・GPU) 指定をPSEUDOが変更出来たかを検証する．

- デバイスの切り替え可否
  各アプリケーションに対し，デバイスの指定をPSEUDOが変更出来たかを検証する．

次に，互換性の確認に加えて利用状況や嗜好に基づいた使用デバイス選択のプラットフォームとしての性能を確認するため，以下の観点から評価する．

- OpenCL Cコードハッシュの取得可否
  PSEUDO DLLがアプリケーションの実行時にOpenCL Cコードのハッシュを取得し，PSEUDOサーバーに送信出来たかを検証する．

- OpenCL実行時間の取得可否
  PSEUDO DLLがアプリケーションの実行時にOpenCLを用いて処理が行われる部分の実行時間を取得し，PSEUDOサーバーに送信出来たかを検証する．

尚，本論文では研究の対象に含めていないCPUのみ利用可能なアプリケーションを考慮した動作については，評価を行わない．

実際の評価には以下のアプリケーションを用いる．

- DirectCompute & OpenCL Benchmark [6] v0.45
  DirectCompute & OpenCL BenchmarkはPat氏により作成され公開されているベンチマークアプリケーションである．本アプリケーションでは，DirectCompute及びOpenCL，CPUネイティブコードを用いたベンチマークを実行出来る．評価には，本アプリケーションのOpenCLを用いたベンチマーク機能を用いて，PSEUDOによる

デバイスの切り替えを試みた場合の動作を検証する．

- SiSoftware Sandra 2011 [21]
  SiSoftware Sandraは，PCのハードウェアやシステムについて詳細な情報を取得し，システムの安定性の評価等の目的のためにベンチマークを行うことが出来る統合的なメンテナンスアプリケーションである．評価においては，本アプリケーションの搭載するGPGPUのベンチマーク機能の１つである，OpenCLを利用するモードを用いてPSEUDO利用時の動作の検証を行う．

- OpenCL入門 マルチコアCPU・GPUのための並列プログラミング [37]に付属するサンプルコード
  OpenCL入門 マルチコアCPU・GPUのための並列プログラミングは，OpenCLについて日本語で書かれた数少ない書籍で，並列化の概念から実際のOpenCLの利用方法等について纏められている．評価においては，本書籍にて解説され，Web上で配布されているサンプルコードを利用し，PSEUDOを用いた際の動作の検証を行う．

6.2 評価環境

6.2.1 PC の仕様

評価には以下のPCを用いる．

メーカー	Hewlett-Packard
製品名・型番	HP Pavilion Desktop PC m9690jp/CT [9]
CPU	Intel Core i7-920 [10]
メモリ	PC3-8500 2GB*3 Triple-channel
チップセット	Intel X58 Express [11]
グラフィックボード	ATI Radeon HD 4850 1GB [17]
OS	Microsoft Windows 7 Professional x64

表 6.1　評価 PC の構成

図 6.1　評価 PC：m9690jp/CT

6.2.2 コンパイラ・SDK 等

サンプルコードのビルドには，Microsoft Visual Studio 2010 Ultimate及びATI Stream SDK 2.2 with OpenCL 1.1 Supportを用いてx86 デバッグビルドのバイナリを作成し，検証を行う．

6.3 評価結果

6.3.1 DirectCompute & OpenCL Benchmark

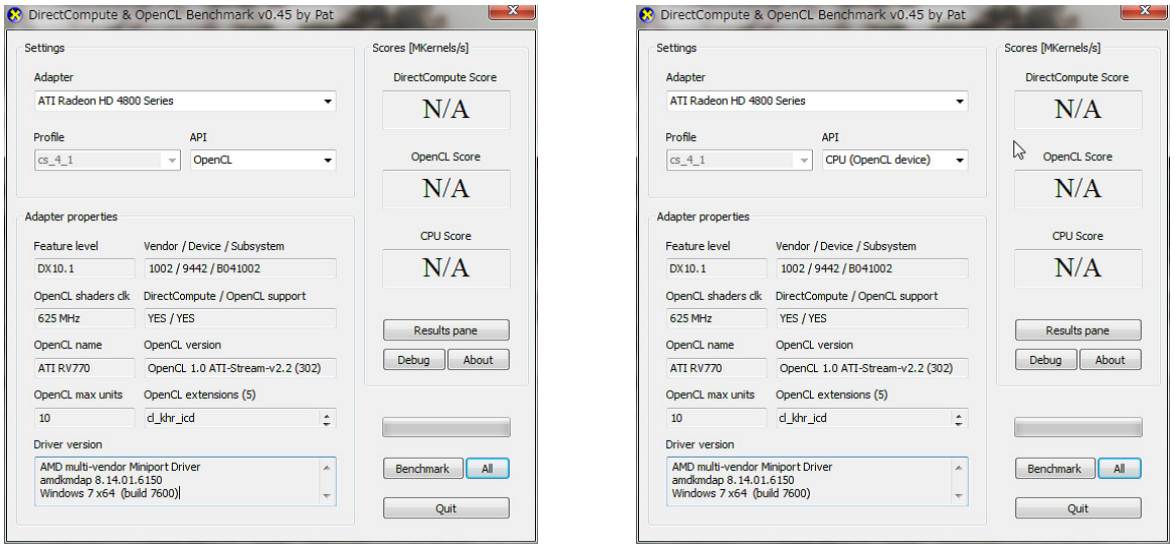
DirectCompute & OpenCL Benchmarkにおいては，OpenCLを用いるベンチマークを行う際，処理を行うデバイスの指定には以下の2種類がある．

- OpenCL（図 6.2(a)）
  本指定は，OpenCLを利用してベンチマークを行うためのものである．評価環境においては，本指定を行った場合DirectCompute & OpenCL BenchmarkではOpenCLデバイスとしてGPUが利用出来る場合にはGPUを，利用出来ない場合にはCPUを用いてベンチマークの処理が行われた．

- CPU (OpenCL device)（図 6.2(b)）
  本指定はOpenCLから利用可能なCPUデバイスを利用するモードである．評価環境においては，本指定を行った場合DirectCompute & OpenCL BenchmarkはOpenCLからCPUデバイスが利用可能な場合にはCPUを利用してベンチマークを行い，OpenCLから利用可能なCPUデバイスが発見出来なかった場合には，”Could not find CPU device!” というエラーが表示され，ベンチマークは実行されなかった．

評価結果は表 6.2のようになった．

まず，デバイスのタイプをCL_DEVICE_TYPE_ALLにオーバーライドし，デバイスのID指定をCPUに指定したところ，アプリケーションからはCPUが 2 つあるように見えた．（A-1）そのため，APIに”OpenCL” を指定した場合も，”CPU (OpenCL device)” を指定した場合もCPUでベンチマークが実行され，CPUが利用されたとアプリケーションは認識した．そのため，どちらの指定を行った場合もベンチマークスコアはCPUのスコアとなり，スコアは”CPU Score” 欄に表示された．このとき，使用可能なデバイスをCPUのみに制限出来たものの，アプリケーションからはCPUが 2 つあるように見える等の問題があるため，適切な動作とは言えない状態であった．



(a) API に”OpenCL” を指定　　(b) API に”CPU (OpenCL device)” を指定

図 6.2　DirectCompute & OpenCL Benchmark における使用デバイスの指定

同様に，デバイスのタイプをCL_DEVICE_TYPE_ALLにオーバーライドしたうえで，先程と異なりデバイスのID指定をGPUとしたのがA-2である．この場合は，APIに”OpenCL”を指定した場合はGPUで実行されたが，”CPU (OpenCL device)”を指定した場合には利用可能なCPUデバイスが存在しないと判断され，”Could not find CPU device!”というエラーが表示された．先程の場合と同様にアプリケーションからはGPUが 2 つあるように見える等，動作はあまり適切ではなかった．

次に，デバイスのタイプをCL_DEVICE_TYPE_CPUとするオーバーライドのみを行ったのが，B-1である．この場合は，アプリケーションからはCPUが 1 つ認識され，APIに”OpenCL”を指定した場合も，”CPU (OpenCL device)”を指定した場合もCPUでベンチマークが実行され，CPUが利用されたとアプリケーションは認識した．したがって，このオーバーライド方法では使用可能なデバイスをCPUのみに制限し，アプリケーションからの認識にも問題が無かったため，デバイスの切り替えは実現出来ている．しかし，この場合は常にCPUを利用することになるため，動的な使用デバイスの切り替えは出来ない．

同様に，デバイスのタイプをCL_DEVICE_TYPE_CPUとするオーバーライドのみを行ったのがB-2である．この場合，アプリケーションからはGPUが 1 つ認識され，APIに”OpenCL”を指定した場合はGPUで実行されたが，”CPU (OpenCL device)”を指定した場合には利用可能なCPUデバイスが存在しないと判断され，”Could not find CPU device!”

というエラーが表示された．これは，CL_DEVICE_TYPE_CPUを指定した場合と同様に使用デバイスをGPUに制限出来たことを表しており，静的な使用デバイスの切り替えは実現出来ているが，動的な切り替えを行うという観点からは十分でない．

続けて，コマンドキュー作成時のデバイスID指定のみCPUを指定したのがC-1である．この場合，アプリケーションからはCPUが１つのみ認識された．これは，アプリケーションがデバイスのタイプにCL_DEVICE_TYPE_DEFAULTを指定しているためと考えられる．ベンチマークを実際に実行した際には，APIに”OpenCL”を指定した場合も，”CPU (OpenCL device)”を指定した場合もCPUでベンチマークが実行され，CPUが利用されたとアプリケーションは認識した．

同様に，コマンドキュー作成時のデバイスID指定のみGPUに指定したのがC-2である．この場合，アプリケーションからはCPUが１つのみ認識された．APIに”OpenCL”を指定した場合はGPUで実行され，”CPU (OpenCL device)”を指定した場合もGPUで実行された．”CPU (OpenCL device)”を指定した場合，GPUで処理を行っているにも関わらず，アプリケーションからはCPUを利用していると認識され，”CPU Score”欄にスコアが表示された．

C-1及びC-2のようにコマンドキュー作成時にデバイスのIDのオーバーライドを行った場合，アプリケーションの指定するデバイスを動的に切り替えて使用デバイスを変更することが出来る．但し，アプリケーションにはCPUが１つのみ搭載されているように見えている点が問題である．

最後に，デバイスのタイプをCL_DEVICE_TYPE_ALLにオーバーライドしたうえで，コマンドキュー作成時にデバイスのID指定をCPUとしたのがD-1である．この場合，アプリケーションからはCPUが１つとGPUが１つ利用可能であるように見え，APIに”OpenCL”を指定した場合も，”CPU (OpenCL device)”を指定した場合もCPUでベンチマークが実行され，CPUが利用されたとアプリケーションは認識した．

同様に，D-2ではデバイスのタイプをCL_DEVICE_TYPE_ALLにオーバーライドしたうえで，コマンドキュー作成時にデバイスのID指定をGPUとした．この場合，アプリケーションからはCPUが１つとGPUが１つ利用可能であるように見え，APIに”OpenCL”を指定した場合も”CPU (OpenCL device)”を指定した場合もGPUでベンチマークが実行された．”CPU (OpenCL device)”を指定した場合はCPUで実行されていると認識され，”CPU Score”欄にスコアが表示された．

D-1及びD-2のようにオーバーライドを行った場合，アプリケーションからはCPUが１つ，GPUが１つ利用可能に見え，実際の使用デバイスを動的に切り替えることが可能である．この状態では，アプリケーションに対し異なるデバイスが利用されていることは隠蔽されるため，DirectCompute & OpenCL Benchmarkでは使用デバイスを適宜切り替えるオーバーライド手法として適切であった．

尚，以上の評価全てにおいてOpenCL Cソースコードのハッシュ取得及び，OpenCLの実行時間の取得が可能であった．但し，DirectCompute & OpenCL Benchmarkでは実行時間の終了と見なすトリガーとなるAPIが呼ばれるのはベンチマーク終了後ユーザーがOKボタンを押した直後タイミングであったため，実際の実行時間よりも長い時間が計測された．

オーバーライド内容	A-1	A-2	B-1	B-2	C-1	C-2	D-1	D-2
デバイスのタイプ指定	ALL	ALL	CPU	GPU	—	—	ALL	ALL
デバイス ID 指定	CPU	GPU	—	—	—	—	—	—
コマンドキューのデバイス ID 指定	—	—	—	—	CPU	GPU	CPU	GPU
動作の安定性	○	○	○	○	○	○	○	○
アプリケーションから見えるデバイス	CPU*2	GPU*2	CPU*1	GPU*1	CPU*1	CPU*1	CPU*1+GPU*1	CPU*1+GPU*1
動的な使用デバイス切り替えの可否	×	×	△	△	△	△	○	○
OpenCL C ハッシュの取得	○	○	○	○	○	○	○	○
OpenCL 実行時間の取得	○	○	○	○	○	○	○	○

表 6.2　DirectCompute & OpenCL Benchmark の評価結果

6.3.2　SiSoftware Sandra 2011

SiSoftware Sandraは，Windows x64環境では64ビットバイナリが利用されるため，32ビットバイナリが利用されるよう，フォルダ名のリネーム等を行って試験を行った．

結果として，Sandraではオーバーライドの指定方法に関わらず，アプリケーションを正常に動作させることが出来なかった．アプリケーションのクラッシュ等は起こらなかったものの，アプリケーション側におけるデバイス名表示に文字化けが発生したり，実在するデバイス数を上回る数のデバイスが認識される等の問題が発生した．また，PSEUDOによるオーバーライドの結果，アプリケーション上の使用デバイスの表示がCPUになってるにも関わらず，実際に使用されるデバイスがGPUとなってしまうなど，意図しない動作が多発した．オーバーライドの指定内容によってはアプリケーションの動作が不安定になることもあり，SandraにおいてはPSEUDOの利用は現実的でなかった．

6.3.3　OpenCL 入門 サンプルコード

OpenCL入門 マルチコアCPU・GPUのための並列プログラミングのサンプルコードによる評価結果は以下のようになった．

- 3-1 hello及び6-2 mt 4

  3-1 helloのサンプルは，PSEUDOを利用しない場合であってもGPUを用いて実行を試みるとエラーが発生し，コンソールに”Hello, World!”の文字を表示することが出来なかった．また，6-2 mt 4のサンプルでは，PSEUDOを利用しない場合であっても

GPUを利用して実行を試みるとPCがフリーズし，操作を受け付けなくなった．そのため，CPUを利用するようオーバーライド指定を行い，評価を行った結果が表 6.3である．デバイスのタイプをCL_DEVICE_TYPE_ALL，デバイスIDをCPUのIDにオーバーライドした場合（P1），デバイスのタイプをCL_DEVICE_TYPE_ALL，コマンドキューのデバイスIDをCPUのIDにオーバーライドした場合（P2），デバイスのタイプ指定をCL_DEVICE_TYPE_CPUにオーバーライドした場合（P3）の全てにおいて，CPUを利用して処理を行うことが出来た．但し，デバイスのタイプ指定のみを行った場合は，使用デバイスをCPUのみに制約することは出来ているものの，使用プロセッサの動的な変更は出来なくなる．なお，OpenCL Cソースコードのハッシュ取得，実行時間計測は問題無く実行出来た．

- 5-1 local

  このサンプルには，ローカルメモリ（OpenCLにおけるメモリ階層モデルの１つ）のサイズを取得するコードが含まれている．評価としては，デバイスのタイプにCL_DEVICE_TYPE_ALLを指定し，デバイスのIDにCPUを指定した場合（R1），デバイスのタイプにCL_DEVICE_TYPE_ALLを指定し，デバイスのIDにGPUを指定した場合（R2），デバイスのタイプにCL_DEVICE_TYPE_CPUを指定した場合（R3），デバイスのタイプにCL_DEVICE_TYPE_GPUを指定した場合（R4），デバイスのタイプにCL_DEVICE_TYPE_ALLを指定し，コマンドキュー作成時のデバイスIDにCPUを指定した場合（R5），デバイスのタイプにCL_DEVICE_TYPE_ALLを指定し，コマンドキュー作成時のデバイスIDにGPUを指定した場合（R6）の6通りの方法で検証を行った．検証の結果，表 6.5のように全ての場合でアプリケーションはエラー等を表示せずに動作し，指定したデバイスを用いて処理が行われた．しかし，このサンプルのようにメモリのサイズ等のデバイス情報を用いて動作するアプリケーションでは，R5・R6のようにアプリケーションがデバイス情報を取得したデバイスと実際に使用するデバイスが異なるようなオーバーライドは危険であり，処理が正常に行われない等アプリケーションの動作に支障を来す恐れがある．また，デバイスのタイプ指定のみを行った場合は，使用デバイスをCPUのみ若しくはGPUのみに制約することは出来ているものの，使用プロセッサの動的な変更は出来なくなる．なお，OpenCL Cソースコードのハッシュ取得，実行時間計測は問題無く実行出来た．

- 他のサンプル

  4-2 online，4-3 dataParallel，4-3 taskParallel，5-1 image，5-1 local，5-2-5，5-2-7，5-2-9，5-2-11，5-2-12，6-1 fft，6-1 fft_double，6-1 fft_optimize，6-2 mt 1，6-2 mt 2，6-2 mt 3，

6-2 mt 5における評価結果が表 6.4である．これらのサンプルでは，デバイスのタイプにCL_DEVICE_TYPE_ALLを指定し，デバイスのIDにCPUを指定した場合（Q1），デバイスのタイプにCL_DEVICE_TYPE_ALLを指定し，デバイスのIDにGPUを指定した場合（Q2），デバイスのタイプにCL_DEVICE_TYPE_CPUを指定した場合（Q3），デバイスのタイプにCL_DEVICE_TYPE_GPUを指定した場合（Q4）の4種の方法で動作の検証を行った．これらのサンプルは，いずれの指定方法であっても指定したデバイスを用いて処理を行うことが出来た．但し，デバイスのタイプ指定のみを行った場合は，使用デバイスをCPUのみ若しくはGPUのみに制約することは出来ているものの，使用プロセッサの動的な変更は出来なくなる．OpenCL Cソースコードのハッシュ取得，実行時間計測については問題無く実行出来た．なお，今回のOpenCLフレームワークでは利用出来ない4-2 offlineサンプル，OpenCLを用いるコードが記述されていない5-2-1，5-2-2，5-2-3，5-2-4，5-2-6，5-2-8，5-2-10のサンプルでは評価を行っていない．

オーバーライド内容	P1	P2	P3
デバイスのタイプ指定	ALL	ALL	CPU
デバイス ID 指定	CPU	—	—
コマンドキューのデバイス ID 指定	—	CPU	—
動作の安定性	—	—	—
デバイス切り替えの可否	○	○	△
OpenCL C ハッシュの取得	○	○	○
OpenCL 実行時間の取得	○	○	○

表 6.3　OpenCL 入門サンプルコードの評価結果（1）

6.4　考察

以上の評価のように，PSEUDOは複数のアプリケーションにおいて動的な使用デバイスの切り替えを実現した．

しかし，デバイスタイプの指定や，デバイスIDの指定，コマンドキュー作成時のデバイスID指定等のオーバーライド手法について，それらをどのように組み合わせるかはアプリケーションに合わせて変更する必要があることが明らかになった．サンプルコードのような単純なアプリケーションでは，タイプの指定やID指定の変更を行った場合であっても，アプリケーションの動作に支障を来すことは少ないが，一般に使用されるアプリケーショ

オーバーライド内容	Q1	Q2	Q3	Q4
デバイスのタイプ指定	ALL	ALL	CPU	GPU
デバイス ID 指定	CPU	GPU	—	—
コマンドキューのデバイス ID 指定	—	—	—	—
動作の安定性	○	○	○	○
デバイス切り替えの可否	○	○	○	○
OpenCL C ハッシュの取得	○	○	○	○
OpenCL 実行時間の取得	○	○	○	○

表 6.4　OpenCL 入門サンプルコードの評価結果（2）

オーバーライド内容	R1	R2	R3	R4	R5	R6
デバイスのタイプ指定	ALL	ALL	CPU	GPU	ALL	ALL
デバイス ID 指定	CPU	GPU	—	—	—	—
コマンドキューのデバイス ID 指定	—	—	—	—	CPU	GPU
動作の安定性	○	○	○	○	△	△
デバイス切り替えの可否	○	○	○	○	○	○
OpenCL C ハッシュの取得	○	○	○	○	○	○
OpenCL 実行時間の取得	○	○	○	○	○	○

表 6.5　OpenCL 入門サンプルコードの評価結果（3）

ンでは，どのようなオーバーライドを行うかを個別に対応する必要がある．

オーバーライドを行う場合，指定方法の選択はアプリケーションの動作時における安全性と多数のアプリケーションへの互換性とのトレードオフとなる．例えば，デバイスタイプの指定のみを行う場合は極めて安全性が高い一方で，使用するデバイスのタイプの制約が可能なだけで動的なデバイスの切り替えが実現出来ないという問題がある．デバイスタイプをALLにオーバーライドし，デバイスIDをオーバーライドする場合は利用される可能性のあるデバイスがアプリケーションから全て認識される可能性が高く，アプリケーションがデバイスの詳細情報を取得しようとした場合にもデバイスIDのオーバーライドが行われるため，アプリケーションが利用を試みているデバイスと実際に利用されるデバイスが一致し，動作における安全性は高い．しかし，デバイスの詳細情報を元に動作するアプリケーションでは，最初に指定したデバイスタイプでないデバイスであると判断され，デバイスの切り替えが正常に行われない可能性がある．デバイスタイプ指定をオーバーライド

せずに，デバイスIDのオーバーライドを行った場合，アプリケーションが利用可能なデバイスとして取得したデバイス一覧に存在するデバイスを利用してしまう可能性があり，実行時の安全性は低下する．しかし，デバイスタイプのオーバーライドが行われないことで，アプリケーションへの互換性は向上する．

一方，デバイスタイプをALLに指定してコマンドキューのデバイスIDを指定する場合，あるいはコマンドキューのデバイスIDのみをオーバーライドする場合では，アプリケーションが指定と異なるデバイスを利用していることを関知出来ない状態になり，デバイスは完全に偽装される．そのため，デバイスの詳細な情報を取得して動作するアプリケーションであっても，デバイスの切り替えを実現出来る可能性が高まる．但し，デバイスが利用出来るメモリのサイズ等の情報についても実際に利用されるデバイスの情報とは異なるものとなるため，動作の安全性は低くなる．特にコマンドキューのデバイスIDのみをオーバーライドした場合，アプリケーションが利用可能なデバイスとして認識していないデバイスを用いて処理が行われる可能性があり，動作の安全性はさらに低下することになる．

以上から分かるように，アプリケーションの仕様デバイスの切り替えを行う場合には，アプリケーション毎にどのような方法でオーバーライドするのかを切り替えて動作する必要がある．安全性を担保しつつ多くのアプリケーションでPSEUDOを動作させるという目的から考えると，まずはデバイスタイプをALLにし，デバイスIDをCPUやGPUのIDに指定する方法での動作を試み，それが不可能な場合はデバイスタイプをALLにしたうえで，コマンドキューのデバイスIDのみをオーバーライドしてデバイスの偽装を行う方法が好ましい．それでも動作出来ない場合には，デバイスタイプをCPUやGPUに指定し，静的な使用デバイスの切り替えを試みることで，より多くのアプリケーションで動作出来ると考えられる．

PSEUDOでは，使用されるアプリケーションに関わらず，OpenCL Cコードのハッシュ取得及びOpenCLの実行時間取得を実現した．

但し，DirectCompute & OpenCL BenchmarkのようにOpenCLを用いた処理を終了させるためにユーザーの入力を待つようなアプリケーションでは，入力待ち時間が実行時間に含まれる形となり，正確な実行時間計測を行うことが出来なかった．しかし，一般的にこのような仕様のアプリケーションは少ないと考えられるため，大きな問題とはならないと考えられる．

6.5 本章のまとめ

本章では，PSEUDOの評価方針及び評価環境と，実際の評価結果，評価結果に基づく考察について述べた．PSEUDOの評価では，PSEUDOを用いた場合に，アプリケーションが正常に動作出来るか，使用デバイスのタイプの切り替えが可能か，動的な使用デバイスの切り替えが可能か，OpenCL Cコードのハッシュ取得が可能か，OpenCLの実行時間が取得

可能かという観点から，複数のアプリケーションを実際に動作させて検証を行った．評価環境にはCore i7-920，Radeon HD 4850を搭載するWindows PCを利用し，アプリケーションとしては，一般に配布されるOpenCLを用いたアプリケーション及びOpenCLのサンプルコードを用いた．

評価の結果，DirectCompute & OpenCL Benchmark及び『OpenCL入門 マルチコアCPU・GPUのための並列プログラミング』のサンプルコードでは，デバイス及びデバイスタイプの切り替え，OpenCL Cコードのハッシュ取得，実行時間取得の全てを実現出来た．一方，SiSoftware Sandraにおいては，アプリケーションの動作に支障を来し，PSEUDOを利用することは出来なかった．

PSEUDOでは，デバイスタイプの指定やデバイスIDの指定，コマンドキューのデバイスID指定を組み合わせてデバイスの切り替えを実現しているが，これらの指定方法はアプリケーションの動作の安全性とアプリケーションに対する互換性のトレードオフとなる．したがって，アプリケーション毎にオーバーライドの指定方法を切り替える必要がある．

第 7 章

まとめ

本章では，始めに本研究で提案した PSEUDO についてのまとめを述べ，次に本研究の今後の展望について述べ，最後に本論文の結論を述べる．

7.1 まとめ

本研究では，マルチコアCPUと高性能GPUがPCに搭載されるようになり，今後GPGPUの利用機会が増えることによってGPUへのリソース集中が起こる可能性があるといった背景から，動的な使用プロセッサの切り替えを実現するPSEUDOの提案を行った．

PSEUDOでは，PCを対象として動的な使用プロセッサの切り替えを実現するため，既存のアプリケーションのバイナリ互換性を確保し，利用状況に応じたプロセッサ切り替えを行うことが出来る柔軟なプラットフォームとすることを目標とした．

既存のアプリケーションにおいて，バイナリの変更を必要とせずに使用プロセッサの切り替えを行うため，PSEUDOでは利用可能なプロセッサ一覧の取得時や使用プロセッサの指定時等の処理に介入を行う形とした．また，利用状況に応じたプロセッサ切り替えを行うために，プロセッサの指定等の介入を行う機構と使用すべきプロセッサを決定・指示する機構とを分離した．PSEUDOでは，この２つの機構を用いて，実際に使用プロセッサを決定する際には問い合わせの通信を行うことで動的な使用プロセッサの切り替えを行った．

実装はOpenCLを用いて行い，OpenCLのDLL上のAPIをフックすることで使用プロセッサ，OpenCLで言うところの”デバイス”の切り替えを行った．PSEUDOの動作手順としては，まずCPUやGPUといったタイプを指定して利用可能なデバイス一覧を取得する際にデバイスのタイプを必要に応じて切り替え，アプリケーションから見えるデバイス一覧の変更を行う．次に，アプリケーションがデバイスのIDを指定してデバイスの情報等を取得しようとした場合に，使用すべきデバイスのIDに切り替えることでアプリケーションに対して異なるデバイスを提示する．さらに，OpenCLのコマンドキューを作成し使用されるデバイスを確定する際に再度必要に応じてデバイスIDを変更することで使用デバイスの切り替えを行う．これらに加えて，実際に使用デバイスを切り替える際の指針となるよう，OpenCL Cコードのハッシュと OpenCLの実行時間を取得する．

評価は一般に配布されているDirectCompute & OpenCL Benchmark，SiSoftware Sandra 2011の各アプリケーション，及び『OpenCL入門 マルチコアCPU・GPUのための並列プログラミング』のサンプルコードを用いて行った．結果，DirectCompute & OpenCL Benchmark，及びOpenCL入門のサンプルコードでは使用デバイスタイプの切り替え，使用デバイスの切り替え，OpenCL Cコードのハッシュ取得，OpenCLの実行時間取得全てを実現出来た．一方Sandraについては，アプリケーションがOpenCLによるデバイス情報取得API以外を用いてデバイス情報を取得しているため，PSEUDOによるAPIのフックを行うと認識されるデバイスの整合性が取れなくなり，デバイス一覧の表示に文字化けが生じる等の問題が発生した．

PSEUDOによる使用デバイス切り替えは，デバイスのタイプ指定のオーバーライド，デ

バイスIDの指定オーバーライド，コマンドキュー作成時のデバイスID指定のオーバーライドの組み合わせにより実現される．これらの指定の組み合わせはアプリケーションの動作の安定性と様々なアプリケーション互換性とのトレードオフとなり，アプリケーション毎にどの指定をオーバーライドするかを決定する必要がある．

PSEUDOはバイナリを改変せずに動的な使用プロセッサの変更を実現し，ユーザー及びアプリケーション開発者の双方に負担をかけることなく，利用状況に応じた使用プロセッサ切り替えによるユーザーエクスペリンス向上を実現するプラットフォームの提供に寄与した．

7.2 今後の展望

PSEUDOは，前述の評価結果の通り，OpenCL以外のAPIを用いてデバイス情報を取得するアプリケーションを除いて，アプリケーションのバイナリ変更を行わずにデバイスの切り替えを行うことが出来た．しかし，今回の評価はCPUが１つ，GPUが１つ搭載される環境であり，GPUが複数搭載されるようなPCでは評価を行うことが出来ていない．GPUを複数搭載するような環境では，デバイスのタイプにGPUを指定するのみでは使用デバイスを確定出来ないため，デバイスIDによる指定が必須となると推測される．したがって，今後はそのような構成のPCにおいてもPSEUDOが正常に使用デバイスの切り替えを実現出来るか検証し，より多くの環境・多くのアプリケーションでデバイスの切り替えが出来るよう改良を加えようと考えている．

また，本研究のゴールはPCの利用状況やユーザーの嗜好に基づいて，OpenCLを用いるアプリケーションに対して使用デバイスの切り替えを実現することである．そのため，今後はPCの利用状況の取得，ユーザーの嗜好を管理するプロファイル機能等を実装し，それらを元にデバイスの切り替えを実現するアルゴリズムを実装する．

7.3 結論

本研究では，OpenCLを利用して実装されたアプリケーションに対し，APIのフックを用いてバイナリの変更を必要とせずに仕様デバイスの切り替えを行うための手法であるPSEUDOを提案した．PSEUDOでは，一般に配布されているアプリケーションやOpenCLのサンプルコードに対し，CPU・GPUといった使用デバイスのタイプや使用デバイスのID等の指定をオーバーライドでき，アプリケーションが処理に用いるデバイスを動的に切り替えることが可能となった．

謝辞

本研究を遂行するにあたり，終始適切な助言を賜り，また丁寧に指導して頂きました慶應義塾大学環境情報学部教授徳田英幸博士に深く感謝致します．また，本論文の執筆に際しご指導を賜った慶應義塾大学環境情報学部准教授高汐一紀博士や，多くの貴重なご助言を頂きました慶應義塾大学環境情報学部専任講師中澤仁博士に深く感謝致します．

また，慶應義塾大学徳田研究室 [8] のファカルティの方々や，先輩方々には折に触れ貴重なご助言を頂き，また多くの議論の時間を割いて頂きました．特に，本論文の執筆にあたって細やかなご指導を頂きました政策・メディア研究科特別研究助教米澤拓郎博士及び環境情報学部講師榊原寛氏には大変感謝しております．

本論文を執筆することが出来たのは，徳田研究会move!グループの皆様の日々の叱咤激励があってこそのものでした．特に政策・メディア研究科修士課程野沢孝弘氏には，様々な助言を頂き感謝しております．また，度々声を掛けて下さった本多倫夫氏，荒木貴好氏，米川賢治氏の先輩方，同輩である中原洋志氏，上田真央氏，西和也氏，丹羽亮太氏の皆様にも感謝しております．

最後に本論文執筆に関してお世話になった全ての方々に深い謝意を表し，謝辞と致します．

2011年2月14日

堀川 哲郎

参考文献

[1] ATI Hybrid CrossFireX Technology. http://game.amd.com/it-it/crossfirex_hybrid.aspx.

[2] ATI Stream. http://www.amd.com/stream/.

[3] ATI Stream Software Development Kit (SDK) With OpenCL 1.1 Support. http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx.

[4] CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[5] Detours. http://research.microsoft.com/en-us/projects/detours/.

[6] DirectCompute & OpenCL Benchmark. http://www.ngohq.com/graphic-cards/16920-directcompute-and-opencl-benchmark.html.

[7] DirectX. http://www.microsoft.com/japan/directx/default.mspx.

[8] Hide Tokuda Lab. http://ht.sfc.keio.ac.jp/.

[9] HP Pavilion Desktop PC m9690jp/CT. http://h50146.www5.hp.com/products/desktops/personal/m9690

[10] Intel Core i7-920 Processor. http://ark.intel.com/Product.aspx?id=37147.

[11] Intel X58 Express Chipset. http://www.intel.co.jp/jp/products/desktop/chipsets/x58/x58-overview.htm.

[12] Microsoft Visual Studio 2010 Ultimate. http://www.microsoft.com/japan/visualstudio/products/2010-editions/ultimate.

[13] OpenCL. http://www.khronos.org/opencl/.

[14] OpenGL. http://www.opengl.org/.

[15] Operating System Market Share. http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8.

[16] Optimus Technology. http://www.nvidia.com/object/optimus_technology.html.

[17] Radeon HD 4850. http://www.amd.com/jp/products/desktop/graphics/ati-radeon-hd-4000/hd-4850/Pages/ati-radeon-hd-4850-overview.aspx.

[18] Rapidmind. http://www.rapidmind.net/.

[19] Rosetta. http://www.apple.com/jp/rosetta/.

[20] Sh. http://libsh.org/.

[21] SiSoftware Sandra 2011. http://www.sisoftware.net/.

[22] Windows 7. https://www.microsoft.com/japan/windows/windows-7/default.aspx.

[23] Erik Alerstam, Tomas Svensson, and Stefan Andersson-Engels. Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration. *Journal of Biomedical Optics*, Vol. 13, No. 6, p. 060504, 2008.

[24] James C. Brodman, Basilio B. Fraguela, María J. Garzarán, and David Padua. New abstractions for data parallel programming. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pp. 16–16, Berkeley, CA, USA, 2009. USENIX Association.

[25] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pp. 197–200, New

York, NY, USA, 2008. ACM.

[26] 浜野 智明al. ヘテロ並列環境のための省電力タスクスケジューリング(並列計算,swopp佐賀2008-2008年並列/分散/協調処理に関する『佐賀』サマー・ワークショップ). 電子情報通信学会技術研究報告. CPSY, コンピュータシステム, Vol. 108, No. 180, pp. 97–102, 2008.

[27] Qu Gang. Power management of multicore multiple voltage embedded systems by task scheduling. In *Proc. ICPPW 2007*, p. 34, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pp. 17–24, New York, NY, USA, 2009. ACM.

[29] Fumihiko Ino, Jun Gomita, Yasuhiro Kawasaki, and Kenichi Hagihara. A gpgpu approach for accelerating 2-d/3-d rigid registration of medical images. In Minyi Guo, Laurence Yang, Beniamino Di Martino, Hans Zima, Jack Dongarra, and Feilong Tang, editors, *Parallel and Distributed Processing and Applications*, Vol. 4330 of *Lecture Notes in Computer Science*, pp. 939–950. Springer Berlin / Heidelberg, 2006. $10.1007/11946441_84$.

[30] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pp. 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] Wei Liu, Brian Lewis, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Sai Luo, and Bratin Saha. A balanced programming model for emerging heterogeneous multicore systems. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pp. 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[32] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: data orchestration and tuning

for opencl devices. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pp. 275–286, Berlin, Heidelberg, 2010. Springer-Verlag.

[33] Robert Szerwinski and Tim Gneysu. Exploiting the power of gpus for asymmetric cryptography. Elisabeth Oswald and Pankaj Rohatgi, editors, Cryptographic Hardware and Embedded Systems — CHES 2008, 第5154巻 of *Lecture Notes in Computer Science*, pp. 79–99. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-85053-3_6.

[34] Hiroyuki Takizawa, Katsuto Sato, and Hiroaki Kobayashi. Sprat: Runtime processor selection for energy-aware computing. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pp. 386–393, 2008.

[35] Leslie Vogt, Roberto Olivares-Amaya, Sean Kermes, Yihan Shao, Carlos Amador-Bedolla, and Alan Aspuru-Guzik. Accelerating resolution-of-the-identity second-order moller plesset quantum chemistry calculations with graphical processing units. *The Journal of Physical Chemistry A*, Vol. 112, No. 10, pp. 2049–2057, 2008. PMID: 18229900.

[36] Li Zhang and R. Nevatia. Efficient scan-window based object detection using gpgpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pp. 1 –7, 2008.

[37] 株式会社フィックスターズ, 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, 1 2010.