OpenCL

本田 和麻，鈴木 真理子
Kazuma HONDA, Mariko SUZUKI

1　はじめに

現在，広く使われているコンピュータには CPU だけでなく，強力な演算性能を持ち一般的な用途にも対応した GPU（Graphics Processing Unit）が搭載されている．また，プレイステーション 3 で用いられている Cell/B.E.（Cell Broadband Engine）も演算性能が高いことが知られており，プログラミング環境も提供されている．しかし，これらの強力な演算性能を持ったプロセッサで動作するプログラムの開発環境は，それぞれのアーキテクチャによって異なるものが提供されており，移植性が低く，開発側は新しい言語を習得する必要があるため効率が悪い．

OpenCL では CPU，GPU および Cell/B.E. といった異種のアーキテクチャで構成されたヘテロジニアスな計算機環境での並列プログラミングフレームワークを提供している．本報告では，OpenCL の概要とハードウェアの抽象化技術について述べ，最後に実装例を示す．

2　OpenCL とは

OpenCL（Open Computing Language）はコンピュータグラフィックス関連の国際会議である SIGGRAPH ASIS 2008 において，CPU，GPU，Cell/B.E. および DSP といったあらゆるアーキテクチャのプロセッサで構成される計算機で，共通に利用できる並列プログラミングフレームワークとして Apple を中心に提唱された．仕様の標準化は OpenGL などで有名な非営利団体である Khronos Group によって進められており，Apple をはじめとして AMD，Intel，IBM，NVIDIA，ソニーなど主要なプロセッサベンダーが仕様策定に関わっている．

次節で OpenCL の特徴であるヘテロジニアス，環境に依存しない開発環境，ハードウェアに近い抽象化について述べる．

2.1　ヘテロジニアス

ヘテロジニアス（異種混在）な環境とは，一般的には異なる機器で構成されているネットワークを表す．特に並列プログラミング環境の点では，異なるアーキテクチャのプロセッサによって構成された計算機を対象としている．x86 CPU 以外にも強力な演算性能を持った GPU や Cell/B.E. が普及し始めており，これらのアーキテクチャは次のような特徴を持っているため，それぞれ長所が異なる．GPU および Cell/B.E. の特徴について以下に示す．

- GPU
  独立して動作する多数の演算コア（スカラプロセッサ）によって構成される．データレベル並列処理に適している．
- Cell/B.E.
  強力な演算性能を持つ SPE（Synergistic Processor Element）を複数個搭載し，それぞれが独立して動作する．タスクレベル並列処理に適している．

2.2　環境に依存しない開発環境

マルチコア CPU の性能を引き出すための開発環境は OpenMP や Intel Threading Building Blocks，GPU では NVIDIA から提供されている CUDA，Cell/B.E. では IBM の Cell SDK というように，それぞれのプロセッサを使うには独自の開発環境が必要であり，ましてや協調動作をさせることは困難であった．そこで OpenCL では OpenCL C 言語仕様，OpenCL ランタイム仕様の 2 つを規定し，共通化された並列処理 API を通じたプログラミング環境が提供されている．

もし利用する OpenCL の実装（OpenCL C 言語のコンパイラとランタイム API）が複数のアーキテクチャに対応していれば，各プロセッサを同じプログラミング言語を用いて協調動作させることが可能となる．また，一度デバッグ済みのソースコードがあれば，他のアーキテクチャにおいても修正することなく移植が可能である．つまり，従来の移植工程では「再コーディング→デバッグ→パラメータチューニング」の手順を踏むところが，OpenCL を用いた場合ではいきなりパラメータチューニングを行うことができる．

2.3　ハードウェアに近い抽象化

クロスプラットフォームで移植性の高いプログラミングモデルを構築するとき，通常はハードウェア（プロセッサアーキテクチャ）の差異が見えないように抽象化される．しかし，OpenCL では通常の方法とは異なり，プログラマ側にハードウェアの差異が見えるように抽象化されている．例えば，次のような低レイヤーな機能が提供されている．

- SIMD（Single Instruction Multiple Data）を想定したベクタ演算
- DMA（Direct Memory Access）を想定した非同期メモリコピー
- 制御用プロセッサと演算用プロセッサ間のメモリコピー

CUDA の場合，メモリの管理はランタイム API に委ねられているが，OpenCL では全てプログラマ側の責任となる．

このように，OpenCL の設計は移植性よりパフォーマンスが重視されている．上記のような低レイヤーな API

により，プログラマ側にはチューニングの余地が多く残されており，各プロセッサのアーキテクチャに合わせた最適化ができることが本フレームワークの大きな特徴である．

3　ハードウェアの抽象化

3.1　プラットフォーム

OpenCL では計算機環境を構成する要素について次の 2 つに分類している．これらの関係を Fig. 1 に示す．

- ホスト（制御装置）
  制御用のプログラム（OS など）が動作する計算機環境．制御用プロセッサや全デバイス間での共有メモリなどが含まれる．
- デバイス（演算装置）
  演算用のプログラムが動作する計算機環境．演算用プロセッサやそれらに付随するメモリなどが含まれる．特に，デバイスで動作するプログラムをカーネルプログラムと表現する．

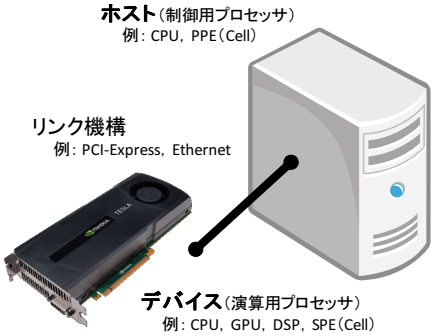

Fig.1　ホストとデバイス（参考文献 [1] より参照）

Fig. 1 のようなプラットフォーム（ホスト-デバイス，マスター-ワーカー，制御系-信号処理系）は GPU，DSP および Cell/B.E. で一般的に用いられている方式である．ホスト-デバイス間を接続する方式は，ソフトウェア，ハードウェア両面で規定されていない．したがって，PCI-Express を用いた方式や，Ethernet 経由で TCP/IP 通信を用いた方式などが実際にあり得る．OpenCL ではホスト・デバイスに如何なるアーキテクチャのプロセッサを用いた場合でも，このようなモデルに従い開発を行う．

3.2　デバイス（演算用プロセッサ）

前節ではプラットフォームの大まかな抽象化を示した．OpenCL では演算用プロセッサであるデバイスを，さらに次の 2 つの要素によって構成されていることを想定している．Fig. 2 にデバイスと NVIDIA CUDA との比較を示す．

- 演算ユニット
  複数のプロセッシングエレメントを構成する単位．OpenCL ではこれらのまとまりをワークグループと呼び，固有の ID が付与される．

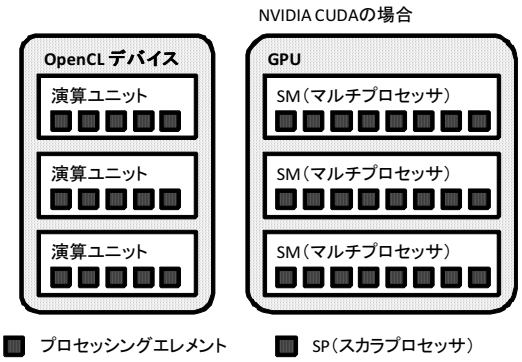

Fig.2　デバイスと NVIDIA CUDA との構成比較（参考文献 [1] より参照）

- プロセッシングエレメント（PE）
  プロセッサを構成する計算要素の最小単位．これの 1 つをワークアイテムと呼び，それぞれにワークグループ内での固有の ID が付与される．

Fig. 2 に示したように，OpenCL のデバイスの抽象化は NVIDIA GPU の構成に酷似している．演算ユニットはマルチコア CPU，Cell/B.E. の場合にはそれぞれ各コア，SPE が対応する．このように，プロセッサのアーキテクチャに合わせてグローバルアイテム数（ワークアイテムの全体数）やローカルアイテム数（ワークグループ内のワークアイテム数）を指定することで，適切なパフォーマンスを得ることが期待できる．

3.3　メモリの構成

OpenCL ではデバイスにおけるメモリの構成を次の 4 つに分類している．この構成図を Fig. 3 に示す．

- グローバルメモリ
  デバイス内のすべての演算ユニットからアクセスできるメモリ
- コンスタントメモリ
  デバイス内のすべての演算ユニットからアクセスできる読み込み専用メモリ
- ローカルメモリ
  同一の演算ユニット内のプロセッシングエレメントで共有できるメモリ
- プライベートメモリ
  各プロセッシングユニット内のみからアクセスできるメモリ

これとは別にホスト側にはホストメモリが存在する．デバイスからはデバイス内のメモリのみにアクセスでき，ホスト側のメモリにアクセスできない．反対に，ホスト側からはデバイス側のグローバルメモリとコンスタントメモリにアクセスすることができる．

ホスト・デバイスの各メモリが物理的に配置される場所は，デバイスのアーキテクチャによって異なる．例えば NVIDIA GPU ではグローバルメモリ，ローカルメモ

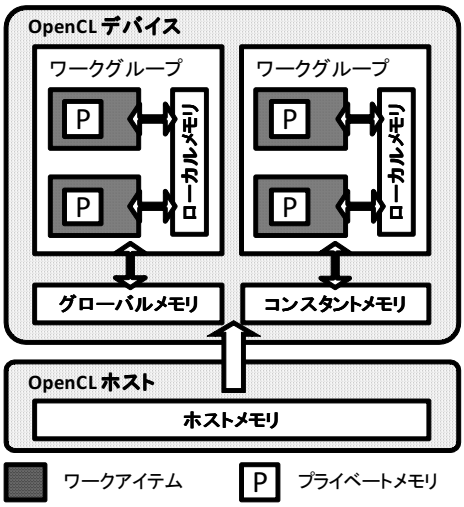

Fig.3　メモリの構成（参考文献 3）より参照）

リ，プライベートメモリは，それぞれ GPU のメインメモリ，共有メモリ，レジスタに配置される．しかし，ホスト・デバイス両方に x86 CPU を用いた場合には，全てのメモリはホスト側のメインメモリに配置される．

4　プログラムの構成

前章で示した通り，OpenCL ではホストとデバイスが異なるアーキテクチャのプロセッサーを扱うことが考慮されている．このため，OpenCL を用いた並列プログラミングでは，以下の 2 つを別々に記述する．

- カーネルプログラム（デバイス用）
- ホストプログラム（ホスト用）

次節でこれら 2 つについて述べる．

4.1　カーネルプログラム

カーネルプログラムはデバイスで動作させるプログラムであり，開発には OpenCL C 言語を用いる．この言語は標準 C 言語（C99）に加え，Fig. 3 のメモリ構成に対応したアドレス空間修飾子や，SIMD 演算を見越したベクタ型といったデータ型が追加されている．反対に，標準ヘッダ，再起呼び出し，可変長配列が使えないといった制限が課せられている．デバイスでは OS が動作していないので，カーネルプログラムの読み込み，カーネル関数の実行制御などは，すべてホストプログラム側から行う．

カーネルプログラムの例として，以下に並列化したベクトル（配列）の加算 $C = A + B$ を行う関数を示す．

```
__kernel void parallelAdd (
    __global float *A, __global float *B,
    __global float *C ){
    __private int gid = get_global_id(0);
    C[gid] = A[gid] + B[gid];
}
```

関数中の gid は各ワークアイテムに付与される固有の ID（0∼$\texttt{gid}_{max}$）であり，これは MPI では rank に相当する概念である．グローバルメモリ上の次元数 $\texttt{gid}_{max}+1$ のベクトル A, B, C に対して，各ワークアイテムは自身の gid に対応する成分の加算を実行する．

4.2　ホストプログラム

カーネルプログラムは計算に特化した並列プログラムであるのに対して，ホストプログラムではカーネルプログラムの実行制御のために様々な手続き行う．次節にホストプログラムの全体の流れを示し，特に重要なカーネルのコンパイル，キューを用いたカーネルプログラムの実行制御について述べる．

4.3　全体の流れ

ホストプログラムは標準 C/C++ 言語と OpenCL ランタイム API を用いて開発を行い，主な役目はカーネルプログラムの実行制御である．一般的なカーネルプログラムの実行までの流れを以下に示す．

1. ホスト・デバイスの設定
2. 制御用オブジェクトの作成・初期化
   - コマンドキューの作成
   - メモリオブジェクトの作成
3. カーネルプログラム用オブジェクトの作成・初期化
   - カーネルファイルの読み込み
   - カーネルのコンパイルとオブジェクトの作成
4. カーネルプログラムの実行
   - ホストからデバイスへメモリデータを転送
   - コマンドキューへの投入・カーネルの実行
   - デバイスからホストへメモリデータを転送
5. 終了処理

はじめにプログラムでホスト・デバイスの設定が行われ，デバイスの種類や数などの情報が得られる．次にコマンドキューの作成が行われる．カーネルプログラムの実行は，このオブジェクトを通して全て管理される．ホスト側では，デバイスが管理するメモリへのアクセスはグローバルメモリとコンスタントメモリに限られ，カーネルの実行前にこれらのメモリにアクセスするためのメモリオブジェクトを作成し，初期化を行う．

4.4　カーネルのコンパイル

OpenCL では以下の 2 種類のカーネルコンパイル方式を規定している．

- オフラインコンパイル方式
- オンラインコンパイル方式

これら 2 つの違いは，オンラインコンパイル方式では実行時に必要に応じてカーネルをコンパイルをするのに対して，オフラインコンパイル方式では実行前にカーネルをコンパイルすることである．

OpenCL のプログラムでは一般的にオンラインコンパイル方式が用いられており，現存の全ての実装でサポートされている．この方式の利点として，デバイスのアーキテクチャに依存することなく，環境に応じて適応的に

コンパイルできることが挙げられる．しかし，実行時にコンパイル分のオーバーヘッドが付加されるので，リアルタイム性が要求される組み込み用途には向かない．

4.5 キューによるカーネルプログラムの実行制御

プログラムの並列化手法にはデータレベル並列と，タスクレベル並列がある．OpenCL では同じカーネルを実行するか，あるいは異なるカーネルを実行するかという点で区別され，デバイスのアーキテクチャによって適した並列化手法が異なる．Fig. 4 に前者の概念図を示す．

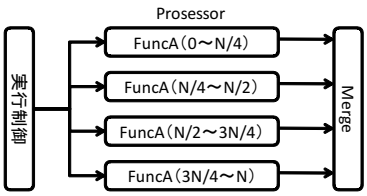

Fig.4 データレベル並列の概念図（参考文献 [1] より参照）

Fig. 4 のようなデータレベル並列は，プログラムで同じ処理を適用できるように分割することができれば非常に強力である．

GPU は多数のプロセッサを備えているが，命令フェッチやプログラムカウンタの制御は各プロセッサで共有されている．このため，複数の異なるカーネルプログラムを並列に動作させることができないので，データレベル並列に非常に適したアーキテクチャであるといえる．OpenCL ではデータレベル並列処理に適した実行制御用 API が提供されている．また，Cell/B.E. などタスクレベル並列処理に適したプロセッサ向けに，アウト・オブ・オーダー実行に対応した API も用意されている．

5 実装例（ライフゲーム）

セルオートマトンとは 2 つの内部状態を持つ格子状のセル（配列）とルールが与えられ，時間が経過する毎に内部状態が変化していく計算モデルである．その 1 つの例として Fig. 5 のようなライフゲーム（Conway's Game of Life）があり，このルールの詳細は文献 [2] に示されている．

本報告では，OpenCL を用いてライフゲームを実装し，CPU と GPU で実行時間を測定した．並列化は，格子を行方向に並列化数（ワークアイテム数）分に均等分割することで行った．実験に用いた環境は次の通りである．

- FOXC（Fixstars OpenCL Cross Compiler）
  Intel Pentium DualCore 2.4GHz，Debian
- NVIDIA CUDA 3.0
  Geforce GTS 250（プロセッサ数:128），Windows

Fig. 6 に格子サイズ 768 × 768 のライフゲームを 1000 世代繰り返した時の，各プロセッサにおける実行時間を示した．縦軸は実行時間（秒），横軸は並列数（ワークアイテム数）である．

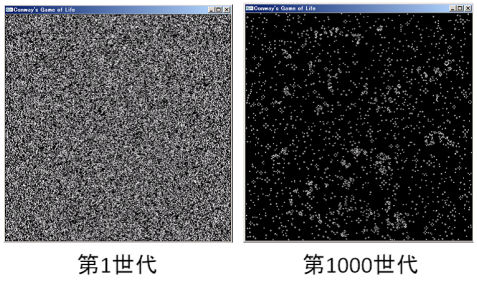

Fig.5 ライフゲーム（Conway's Game of Life）

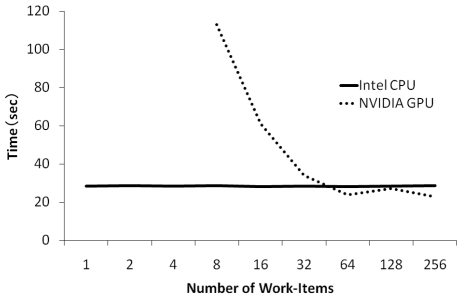

Fig.6 各プロセッサの実行時間

Fig. 6 より，GPU では並列数をプロセッサ数と同等にすることで高いパフォーマンスが得られたのに対し，CPU では並列数はパフォーマンスに影響を与えなかった．両プロセッサの実行時間を比較すると，大きな差があるとは言い難い．Fig. 3 のようなメモリの構造を意識して改良することで，GPU では実行時間を更に短縮できると考えられる．

6 まとめ

本報告では OpenCL はプロセッサのアーキテクチャに依存しないヘテロジニアスな並列プログラミング環境であり，OpenCL C 言語とランタイム API を用いて開発を行うことができることを示した．また，OpenCL は NVIDIA GPU の構造にならい，ハードウェアを低レイヤーな部分で抽象化している．このため，パラメータチューニングによるパフォーマンスの最適化を行えることができる．ライフゲームのプログラムから，GPU では並列数とワークアイテム数を同等にすることで高いパフォーマンスが得られることを確認した．

参考文献

1) 株式会社フィックスターズ 著．OpenCL 入門 - マルチコア CPU・GPU のための並列プログラミング，株式会社インプレスジャパン，2010
2) Clay Breshears 著．並行コンピューティング技法 - 実践マルチコア/マルチスレッドプログラミング，株式会社オライリー・ジャパン，2009
3) OpenCL Parallel Computing for Heterogeneous Devices: http://www.khronos.org/developers/library/overview/opencl_overview.pdf
4) PC Watch: http://pc.watch.impress.co.jp/docs/2009/0330/kaigai497.htm