Vol.2010-OS-113 No.5
2010/1/27

情報処理学会研究報告
IPSJ SIG Technical Report

異なるOpenCL実装を接続可能にするHybrid OpenCLの構築

青木 亮[†1] 追川修一[†1]
土山了士[†2] 中村孝史[†2]

本研究では，OpenCL 環境でのスケーラビリティーの向上，より多様な並列計算環境の提供，OpenCL アプリケーションの利用価値の向上などを目的として，異なる OpenCL 実装をネットワーク経由で接続するための OpenCL ランタイムと，ランタイム間を接続するブリッジプログラムを開発する．また，これらの 2 つのコンポーネントを用いたシステムを Hybrid OpenCL と呼び，本論文では，この Hybrid OpenCL の実装方法と実験結果について述べる．

Implementation of Hybrid OpenCL that enables Connection between Different OpenCL Implementations

Ryo Aoki,[†1] Shuichi Oikawa,[†1] Ryoji Tsuchiyama[†2] and Takashi Nakamura [†2]

We are currently developing am OpenCL runtime system that connects different OpenCL implementations over network and a bridge program to connect between OpenCL runtime systems. This system that consists of these two components are named Hybrid OpenCL. Hybrid OpenCL aims at the improvement of scalability in OpenCL environments, providing more various parallel computing platforms and progress of utility value of OpenCL applications. This paper describes the implementation of Hybrid OpenCL and its result of experimentation.

†1 筑波大学
University of Tsukuba
†2 株式会社フィックスターズ
Fixstars Corporation

1. はじめに

近年の計算機は長らくシングルコア CPU の性能向上によって，その速度を高めてきた．しかし，消費電力や，高クロック化の限界から頭打ちとなり，次第に CPU のマルチコア化，ベクトル化が加速している．本来 CPU は制御を目的としたプロセッサである．OS の実行や複雑な分岐処理は得意だが，大量のデータを高速に処理することにそもそも向いていないため，そのピーク性能はベクトル演算専用に設計されたプロセッサには及ばない．

ベクトル演算専用に設計されたプロセッサとしては GPU（Graphics Processing Unit）が上げられる．GPU は 3D モデルのレンダリングや，画像の加算合成や乗算合成を高速に行うため，ベクトル演算専用に設計されてきたプロセッサである．GPU のように計算目的が限定された専用のプロセッサをアクセラレータと呼ぶが，通常のアクセラレータは演算ロジックがハードウェアで決まってしまうため，様々な目的には利用できない．初期の GPU もレンダリングする画像の表現方法をハードウェアによって実装していた．しかし，半導体の集積技術や，メモリの高速化が進むにつれ，GPU の表現方法はプログラムによって変更できるプログラマブルシェーダに移行してきた．このような変遷をとげた GPU は次第にレンダリングだけでなく様々な計算を行えるようになり，汎用目的計算も実行可能になった．

こうした結果，GPU を汎用目的の計算に使用する GPGPU[1)]（General Purpose computing on GPU）という利用方法が現れた．GPGPU により GPU 上で CPU と同様の計算を CPU より並列度高めて実行できるようになり，計算によっては高いパフォーマンスを発揮できるようになった．しかし，GPGPU の開発環境は対応製品が限られている場合や，各開発環境で使用する言語が統一されていないという問題が存在した．これらの問題を解決するために，OpenCL[2)3)] というヘテロジニアスな並列コンピューティング用フレームワークが標準化団体の Khronos Group によって策定された．OpenCL によって，様々なデバイス上で動作するアプリケーションを共通の言語によって開発することが可能になったが，依然いくつかの問題が存在する．

OpenCL は主に，各デバイスの開発元が OpenCL 対応のランタイムライブラリを提供している．逆に言えば，開発するアプリケーションはリンクした OpenCL ランタイムライブラリが対応しているデバイスしか扱えないということになる．これは，同一マシンに異なる種類の OpenCL デバイスを接続しても扱うことができないことを意味する．また，OpenCL ではホストとデバイスの接続方法は特に規定されていないが，現行使用できる OpenCL デバイスは計算機の内部バスに接続されているものに限られている．これは近年の GPU など

で採用されている PCI Express x16 の場合，最大 4 個のデバイスしか接続できず，それも一部のマザーボードに限られてしまう．これはシステムのスケーラビリティーを下げることにつながる．さらに，これらの問題によりアプリケーションの利用価値が下がってしまうという問題も浮上する．

そこで本研究では，これらの問題を解決するために異なる OpenCL 実装を接続するための OpenCL ランタイムとランタイム間を接続するブリッジプログラムを開発する．ランタイム間はソケットを用いて接続することで，ネットワーク上の OpenCL デバイスも使用可能し，これらの 2 つのコンポーネントを用いて作られるシステムを Hybrid OpenCL と呼ぶ．

2. OpenCL

2.1 OpenCL の概要

OpenCL は Apple によって提唱され，その後，標準化団体 Khronos Group によって 2008 年 12 月にバージョン 1.0 が標準化された，ヘテロジニアスな環境を想定した並列コンピューティング用フレームワーク規格である．Khronos Group はこのほかにも OpenGL などの標準化を行っている．OpenCL には以下のような特徴がある．

- データ並列・タスク並列プログラミングモデルのサポート
- 並列化のための拡張を加えた ISO C99 ベースのプログラミング言語
- IEEE 754 をベースとした数値に関する必要条件の定義
- 小型・組み込みデバイス向けのプロファイルの定義
- OpenGL，OpenGL ES やその他のグラフィックス API との効率的な相互運用が可能
- マルチコア CPU，GPU，Cell/B.E.，DSP などの様々なデバイスをサポート可能

実際の OpenCL の実装はサードパーティに任されており，OpenCL の仕様通りに実装しテストに合格すれば，OpenCL 準拠を名乗ることができる．初の OpenCL 実装は Apple の Mac OS X Snow Leopard のコンポーネントであり，2009 年 8 月に発売された．これに加え現在は NVIDIA，ATI，IBM，Fixstars 製の OpenCL 実装が存在する．

2.2 OpenCL の構成

OpenCL 実装はランタイムライブラリとコンパイラから構成されている（図 1）．OpenCL ランタイムはバージョン 1.0 では 71 個の API を提供し，デバイスの制御やリソースの管理を行う．OpenCL コンパイラは OpenCL デバイス上で動作するプログラムをコンパイルする．このプログラムは C99 ベースの OpenCL C 言語で記述する．この OpenCL C 言語で記述されたプログラムは OpenCL ではカーネルと呼ばれ，カーネルは各デバイス用のコンパイラでコンパイルすることで，どの OpenCL デバイス上でも動作させることができる．OpenCL では OpenCL に関わるリソースをコンテキストと呼ばれる概念で管理する．また，バッファはメインメモリ上に確保されるものとデバイス上に確保されるものがあり，これらは OpenCL アプリケーションやカーネルが使用する．さらに，カーネルの実行や，バッファの読み書き，コピー，同期などは全て，コマンドキューと呼ばれる，デバイスに対して 1 対 1 に対応したキューにコマンドを投入することで行う．

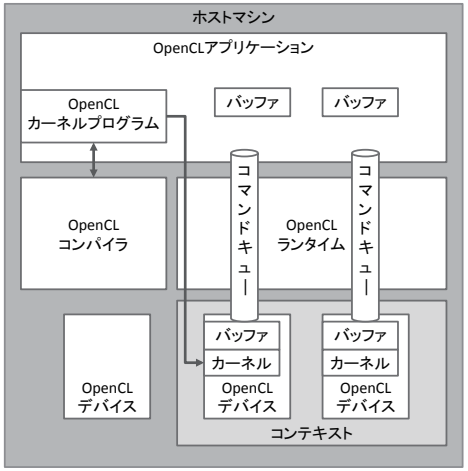


図 1　OpenCL

3. Hybrid OpenCL

OpenCL では異なる OpenCL デバイスを単一のランタイムから使用できないこと，また，使用できる OpenCL デバイス数が内部バスに接続できる数に制限されてしまうことが問題である．本研究ではこのような問題に対しスケーラビリティーの向上，より多様な並列計算環境の提供，OpenCL アプリケーションの利用価値の向上などを目的として複数の OpenCL ランタイムを接続可能な Hybrid OpenCL を開発している．この章では，Hybrid OpenCL の概要や設計，その実装について述べる．

3.1 Hybrid OpenCL 概要

Hybrid OpenCL は複数の OpenCL ランタイムを接続するために，図 2 のように，OpenCL

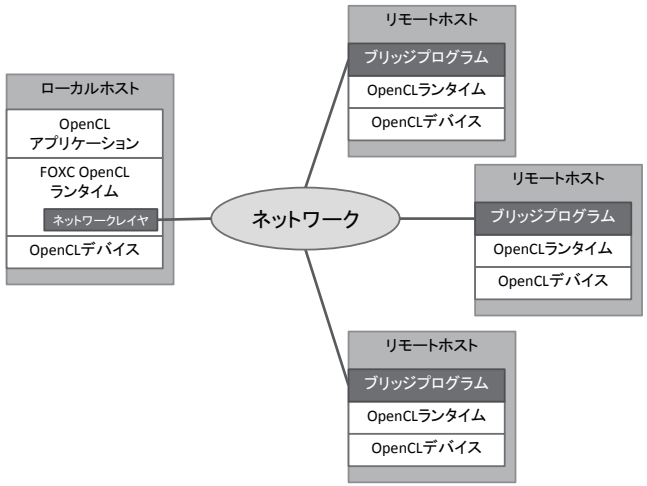


図 2　Hybrid OpenCL

アプリケーションにリンクする OpenCL ランタイムにネットワークレイヤを追加し，他の OpenCL ランタイムとブリッジする方法をとる．ネットワークレイヤには様々な環境を想定しソケットを使用する．OpenCL ランタイム間にはブリッジプログラムを配置し，これが OpenCL アプリケーションからの API 呼び出しを再現し，リモートホストにおいての実際の OpenCL リソースを管理する．本研究では OpenCL ランタイムにネットワークレイヤを追加したものを Hybrid OpenCL ランタイムと呼ぶ．

3.2 Hybrid OpenCL ランタイム

Hybrid OpenCL ランタイムの役割は，OpenCL アプリケーションからリモートホストの OpenCL デバイスを，ローカルホストの OpenCL デバイスと同じ方法で使用できるよう，抽象化することである．ここではリモートホストの OpenCL デバイスを単にリモートデバイス，ローカルホストの OpenCL デバイスをローカルデバイスと呼ぶ．

Hybrid OpenCL ランタイムはリモートデバイスを管理するために，ネットワーク上に存在するホストの中から OpenCL デバイスをもつホストを探して，リストアップしておく必要がある．現在は実装の簡単化のために，静的なテキストファイルに参加させたいホストの IP アドレスを列挙しておくことで対応している．OpenCL ランタイムの初期化時にこのテキストファイルが読み込まれ，リモートホストのリストが設定される．

その他に，OpenCL ランタイムでは，新しい OpenCL リソースを確保することで，リソースに対応するリソースハンドルが新たに割り当てられる．そのため，Hybrid OpenCL ランタイムは各ブリッジプログラムの OpenCL ランタイムで取得したリソースハンドルと，実際に OpenCL アプリケーションに渡すリソースハンドルとの関連付けも行う必要がある（図 3）．

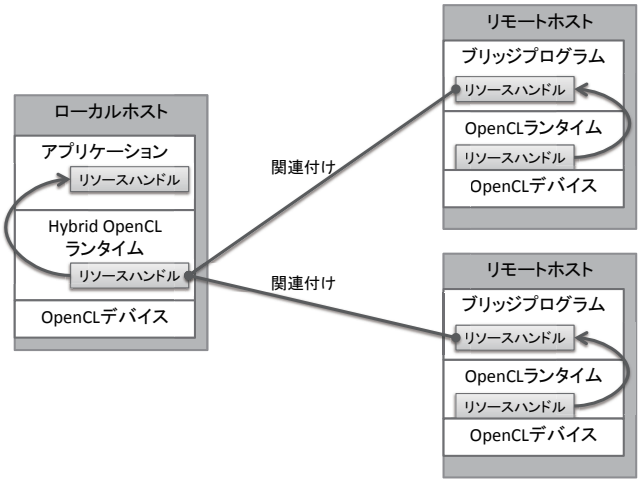


図 3　リソースハンドルの関連付け

今回は Hybrid OpenCL ランタイムを開発するに当たって，Fixstars 製の FOXC(Fixstars OpenCL Cross Compiler) OpenCL ランタイムをベースとした．これは Linux で動作し Intel のマルチコア CPU をデバイスとみなす OpenCL 実装である．

3.3 ブリッジプログラム

ブリッジプログラムの役割は，Hybrid OpenCL ランタイムからの API 呼び出しを，自身にリンクされている OpenCL ランタイムで実行し，結果を Hybrid OpenCL ランタイムに返すことである．ここではブリッジプログラムにリンクしている OpenCL ランタイムを，ブリッジドランタイムと呼ぶ．

ブリッジプログラムはブリッジドランタイムに対してリソースの確保を行うが，その際のリソースハンドルは確保を行ったブリッジドランタイム固有のものなので，これらのリソースハンドルを適切に保存，管理する必要がある．さらに，複数の OpenCL アプリケーションが利用できるようにするため，これらのリソース管理は接続してくる OpenCL アプリケーションごとに別々に管理する必要がある．今回，ブリッジプログラムは上記のような複雑なリソース管理を行う必要があるため，C++のクラスを用いて作成している．

3.4 設計と実装

この節では，上記の Hybrid OpenCL ランタイムとブリッジプログラムの要求をどのように設計し，実装したかについて具体的に述べる．

3.4.1 リソースハンドルの問題点

OpenCL のリソースには，プラットフォーム ID，デバイス ID，コンテキスト，コマンドキュー，メモリオブジェクト，プログラムオブジェクト，カーネルオブジェクト，イベントオブジェクトがあり，これらは図 4 の様な関係にある．

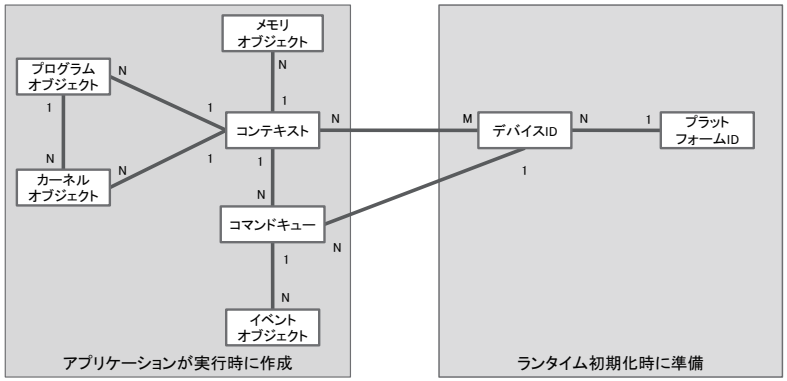


図 4 OpenCL におけるリソース間の関係

プラットフォーム ID はデバイスが属するプラットフォームの種類を識別する識別子，デバイス ID は OpenCL デバイスの識別子，コンテキストはアプリケーションが使用するデバイスを指定して作成され，そのデバイスと他のリソース（コマンドキュー，メモリオブジェクト，プログラムオブジェクト，カーネルオブジェクト，イベントオブジェクト）をグループ化する識別子，コマンドキューはデバイスに対して命令を発行する際の命令キュー，メモリオブジェクトは OpenCL デバイス上に確保するバッファや，ホストのメインメモリ上に確保するバッファに対応するオブジェクト，プログラムオブジェクトは OpenCL デバイス上で動作するプログラムのソースコードやバイナリコードに対応するオブジェクト，カーネルオブジェクトはプログラム内の関数に対応するオブジェクト，イベントオブジェクトはコマンドキューに発行した命令に 1 対 1 で対応するオブジェクトである．

注目すべきは，コマンドキューからはデバイスを一意に識別できるのに対して，メモリオブジェクト，プログラムオブジェクト，カーネルオブジェクトからは一意にデバイスを識別できない点である．これはコマンドキューの作成時には単一のデバイス ID を指定するが，その他のオブジェクト群の作成時にはコンテキストを指定するように OpenCL の仕様で定義されているためで，これらのオブジェクト群は実際にアプリケーション側がメモリに対する読み書きやプログラムのビルド，カーネルの実行を行うまで，どのホストの，どのデバイスで必要なのかが分からないということである．

このため，ホストとデバイスが特定できないリソースがアプリケーションによって作成される際は，コンテキスト作成時に指定された，リモートデバイスが存在するホストの，ブリッジプログラム全てに実際のリソース作成を指示しなければならない．結果として，アプリケーション側が受け取るリソースハンドルに対して，複数のホスト上のリソースハンドルを関連づけ，かつどのホストのリソースハンドルなのかが分かるように管理しなければならない．

3.4.2 Hybrid OpenCL におけるリソースハンドルの管理方法

そこで，本研究ではベースとなる FOXC OpenCL ランタイムにおける，リソース管理用構造体それぞれに新たにフィールドを追加する．追加するフィールドはリモートホストのリソースハンドルとそのリモートホストを識別するホスト情報構造体からなる構造体のポインタである．また，多数のリモートホストに対応するためにこの構造体を一方向リンクリストとして実装する．このように実装することで，リストの先頭ポインタである，リソース管理用構造体に追加したフィールドが NULL であればローカルホストのリソース，そうでなければリモートホストのリソースかどうかも判定できる（図 5）．

実際に OpenCL アプリケーションがランタイム API を呼び出した際は，API に指定されたリソースのリソース管理用構造体をチェックし，リモートホストのものであった場合のみ，ネットワーク越しに API の種類とその引数をブリッジプログラムに送信する．ローカルホストのものであった場合は，通常通りローカルの OpenCL ランタイムが処理を行う．ブ

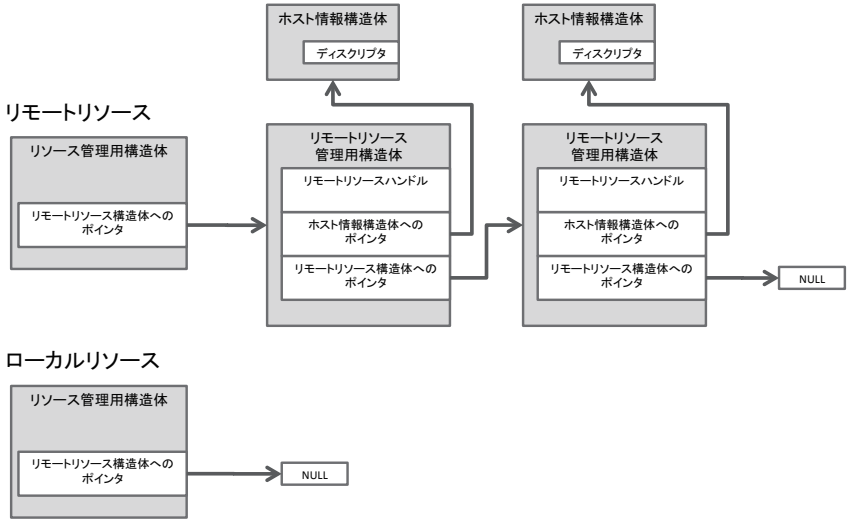


図 5　Hybrid OpenCL ランタイムにおけるリソース管理の為のデータ構造

リッジプログラムは API の種類と引数を受信し，該当 API を実行後，出力引数や，戻り値を OpenCL アプリケーションが動作するホストへ送信する．

例えばバッファの作成の場合，cl_mem = clCreateBuffer(context, flag, size, host_ptr, errcode) という API を使用する．OpenCL アプリケーションによって clCreateBuffer() が呼び出された際，context がリモートデバイスを含む場合は，clCreateBuffer() の API を表す整数値とリモートの context，引数である flag，size，host_ptr が context 内に含まれる全てのリモートデバイスを持つホストに渡される．errcode は出力引数であるのでブリッジプログラムへは送信されない．ブリッジプログラムはこららを受け取ると，ブリッジドランタイムの API にそれらを渡し，実行する．そして，結果である cl_mem と errcode を呼び出し元のホストへ送信する．データを受け取ったホストはメモリオブジェクトのハンドルである cl_mem をリソース管理用構造体の一方向リンクリストに追加していく．

また，バッファを読み込む場合は，cl_int = clEnqueueReadBuffer (command_queue, cl_mem, blocking_read, offset, cb, ptr, num_events_in_wait_list, event_wait_list, event) という API を使用する．第 2 引数の cl_mem は作成時にはどのホストの，どのデバイスで実際に使用されるか分からなかったが，ここでようやく使用するホストとデバイスを特定することができる．コマンドキューのリソース管理用構造体からデバイス ID を特定し，リモートリソースの一方向リンクリストから対象のデバイスが接続されているホストのホスト情報構造体を特定する．このホスト情報構造体を用いて対象ホストの特定のデバイスにのみ API を適用している．

実際の通信にはコンテキスト作成時に作成されたディスクリプタがホスト情報構造体に格納され，以後の各ホストへのアクセスはこのディスクリプタを使って行われる．このようにすることで，各コンテキストごとに別々のセッションが確保される．ブリッジプログラムではこのセッションごとにスレッドを生成し，各スレッドでコンテキストに対応したクラスをインスタンス化する．このクラスは，各 OpenCL リソースを管理するために，ハッシュテーブルや，マップなどのデータ構造を用いて高速にリソースを管理できるように実装している．

3.5 OpenCL の仕様に準拠するための問題

この節では，OpenCL の仕様に準拠する際に，実装上問題となる事柄について述べる．

3.5.1 イベントオブジェクトの解放

OpenCL ではイベントオブジェクトというコマンドキューへ投入された命令の同期を管理するためのオブジェクトがある．コマンドキューに命令を投入する clEnqueue から始まる API は全てこのイベントオブジェクトを返せる仕様となっている．ここで “返せる” とは，使わない場合は NULL を指定することで，イベントオブジェクトを生成しないことも可能であるという意味である．しかし，実際には NULL を指定しても，今回使用した FOXC OpenCL ランタイムでは，内部的なコマンドキューの管理にイベントオブジェクトを常に使用している．この場合はイベントに対応する命令が完了した際に解放される．また，アプリケーションがイベントオブジェクトを取得した場合は，clReleaseEvent() を用いて明示的に解放しなければならない．これらのルールはローカルデバイスの時にはうまく動作するが，リモートデバイスの時はうまく動作しない可能性がある．

アプリケーションがリモートデバイスに対応づけられたコマンドキューに命令を投入する際，イベントオブジェクトを取得したとする．このとき，Hybrid OpenCL ランタイムではリモートリソースをローカルリソースの構造体にリンクリストで連結していることは述べた．ただし，ローカルリソースの構造体は実際には使われず，単にアプリケーション側からリモートホストのイベントオブジェクトを指すだけの役割しかない．つまり，このときの命令はリモートコマンドキューで実行されてしまうため，ローカルコマンドキューの命令完了

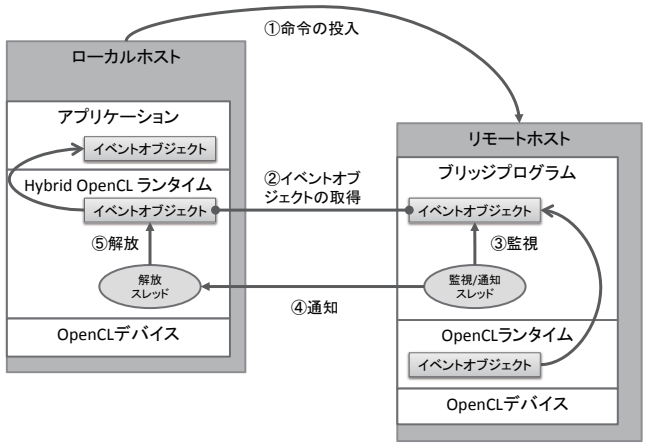


図 6　イベントオブジェクトの解放処理

時に走るイベントオブジェクトを解放するためのコードが走らない．よってこの状況下では，アプリケーションが取得したイベントオブジェクトを使い終わり，明示的に解放したとしても，内部的な解放が行われず，必要のないイベントオブジェクトが無限に増え続けてしまう．

そこで Hybrid OpenCL ランタイムでは，イベントオブジェクトを使わない場合は，直ぐにイベントオブジェクトを解放し，もし使う場合は図 6 のように，ローカルホスト側でイベントの終了を待ち，リモートホスト側でのイベント終了を監視，通知してもらうことで対応している．これらの処理は各ホストに新しいスレッドを作り，スレッド間で通信することで実装している．

3.5.2　メモリオブジェクトのマップ

OpenCL におけるメモリオブジェクトは，そのデータをホストのメインメモリ領域にマッピングすることが可能である．Hybrid OpenCL ランタイムではメモリのマッピングを行うために，リモートホストにあるデータをマップの際にローカルホストにコピーすることで対応している．また，Hybrid OpenCL ランタイムでは，マップしている間のメモリオブジェクトのデータはローカルホストでのみ有効で，たとえ書き込みが起こっても，自動的にリモートホストに書き戻されることはない．しかし，アプリケーションがメモリオブジェクトをアンマップしたときには，ローカルホストのデータをリモートホストに書き戻す．

3.5.3　ネイティブカーネル

OpenCL ではデバイス上で動作するプログラムをカーネルと呼ぶが，このカーネルは通常，実行される OpenCL デバイスに対応した OpenCL コンパイラでコンパイルされる．OpenCL にはこの通常のカーネル以外に，アプリケーションのコードをカーネルとして実行できるネイティブカーネルという概念が存在する．ネイティブカーネルはホストのコードの関数ポインタを受け取り，その関数ポインタを実行している．ネイティブカーネルは関数ポインタを使用しているので，実行するデバイスがリモートデバイス場合は，ネイティブカーネルを特定できないので実装できないことになる．しかしながら，OpenCL ではネイティブカーネルがオプション扱いであるため，Hybrid OpenCL ランタイムではリモートデバイスの場合，ネイティブカーネルをサポートしないこととして対応している．

3.5.4　ホストに割り当てたメモリオブジェクトの同期

OpenCL におけるメモリオブジェクトは，OpenCL デバイスだけでなく，ホストのメインメモリ領域に割り当てることも可能である．メモリオブジェクトのデータをリモートホストで使う場合は，必要になったときに対応するブリッジプログラムにコピーすることで対応している．実際に同期しているタイミングは，メモリオブジェクトのマップ時やネイティブカーネル実行時，カーネルへの引数の設定時など，メモリオブジェクトを引数にとれる API 実行時である．

3.5.5　通知用コールバック関数

OpenCL ランタイム API にはその完了を通知してもらうためのコールバック関数を引数にとれるものが存在する．Hybrid OpenCL ランタイムでは，現在これらの通知用コールバック関数を実装できていないが，以下のように実装することで対応可能だと考えている．

まず，ローカルホスト側でコールバック関数を管理し，リモートホスト側にはコールバック関数に対応する識別子を渡した後，リモートホスト側でも通知用コールバック関数を使う．コールバック関数にはユーザ引数を持たせることができるため，ここにコールバック関数の識別子を渡しておく．通知を受けた後，識別子からどの通知かを判断し，ローカルホストに通知する．

3.6　実　装　量

本研究で開発した Hybrid OpenCL の実装量は，FOXC OpenCL ランタイムへの追加コードが C 言語 約 5300 行，ブリッジプログラムは C++ 約 4500 行となった．実装した OpenCL ランタイム API は 71 個中 42 個でイメージオブジェクト，サンプラオブジェク

ト，OpenCL/OpenGL シェアリングの為の API 群は未実装である．また，コードの大半は API の引数や戻り値をホスト間でやりとりするために必要なデータのシリアライズ・デシリアライズのためのコードである．

4. 実　　験

4.1 動 作 検 証

以上のように実装した Hybrid OpenCL を用いて，動作検証実験を行った．実験環境として Dell Precision 490 (Xeon 5130 2.0GHz, Memory 2GB, NIC 1Gbps) を 2 台用意し，1 台で OpenCL アプリケーションと Hybrid OpenCL ランタイムを動作させ，もう 1 台でブリッジプログラムと FOXC を動作させた．これらの環境で実行した Hybrid OpenCL システムの結果と従来通りの OpenCL 環境で動作させた結果を比較した．OpenCL アプリケーションには NVIDIA OpenCL に添付されている NBody[4]（多体問題）プログラムや FFT（高速フーリエ変換）プログラムを使用した．検証の結果，NBody や FFT の結果が共に一致することを確認した．

4.2 性 能 評 価

今回，Hybrid OpenCL の性能評価として，NVIDIA NBody アプリケーションによる処理時間の計測を行った．NBody は多体問題を計算するプログラムで，N 体の質量を持つ粒子間の重力を計算し粒子の運動をシミュレートする．NBody では 1 ステップ計算を進めるごとに以下の OpenCL ランタイム API を呼び出す．

- clSetKernelArg()
- clEnqueueNDRangeKernel()
- clEnqueueReadBuffer()

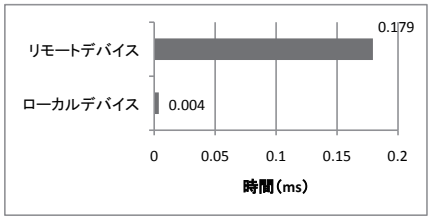
clSetKernelArg() はカーネルへの引数の設定を行う API，clEnqueueNDRangeKernel() は指定したコマンドキューに対してカーネルの実行を表すコマンドを投入する API，clEnqueueReadBuffer() は指定したコマンドキューに対してバッファの読み込みを表すコマンドを投入する API である．ただし，clSetKernelArg() は一度に一つの引数しか設定できない．これらの単一 API の処理時間とともに，各行程の全体の処理時間を計測した．これは，NBody におけるカーネルが引数を 9 つ持つため，clSetKernelArg() API は 9 回呼び出される点．また，clEnqueueNDRangeKernel() は単にカーネルの実行を指示するコマンドをコマンドキューに入れるだけで，実際の計算終了を OpenCL の同期機構を用いて待つ必要がある点から，別途計測する必要がある．また比較のため，これらの計測を，通常の OpenCL アプリケーションのようにローカルデバイスで実行した場合と Hybrid OpenCL を使用しリモートデバイスで実行した場合の 2 パターンの計測を行った．

4.2.1 評 価 環 境

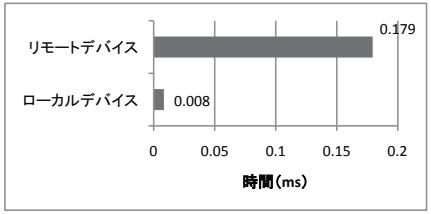
評価環境としては以下のスペックの PC 2 台で行った．

マシン	Dell Precision 490
CPU	Xeon 5130 2.0GHz
メモリ	2GB
NIC	Broadcom NetXtreme BCM5752 (rev 02) 1Gbps
OS	CentOS 5.4 (Linux 2.6.18)

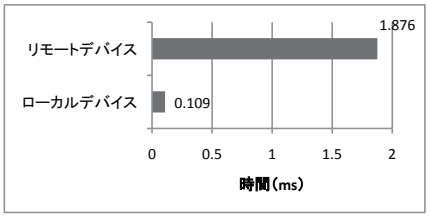
4.2.2 評 価 結 果



(a) clSetKernelArg



(b) clEnqueueNDRangeKernel



(c) clEnqueueReadBuffer

図 7　API あたりの処理時間

評価結果を図 7，図 8 に示す．図 7 は API あたりの処理時間を計測したグラフである．

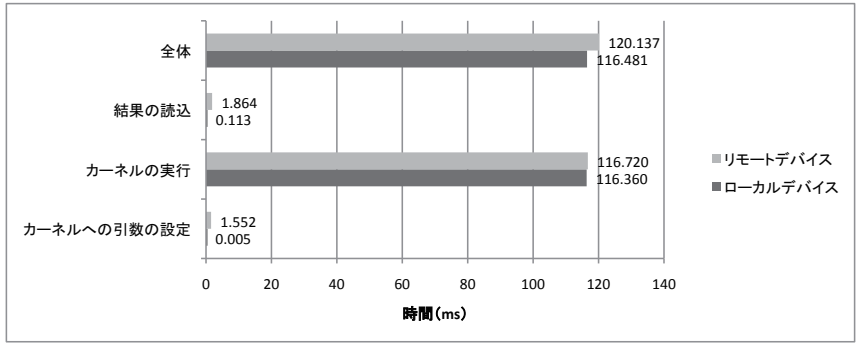


図 8　NBody の計算時間（1 ステップ）

図 8 は NBody の 1 ステップごとの処理時間の平均である．図 8 の全体とはカーネルへの引数の設定，カーネルの実行，結果の読込を含めた全体の処理時間である．カーネルへの引数の設定は clSetKernelArg() を 9 回呼び出すので，リモートデバイスの場合，図 7(a) の結果から，0.179ms × 9 = 1.611ms となり，図 8 カーネルへの引数の設定の 1.552ms と近い値になっている．0.45ms 程誤差があるのは，引数のタイプによってデータのサイズや手続きが異なるためだと考えられる．また，結果の読み込みも clEnqueueReadBuffer() を 1 度呼ぶため，図 7 とほぼ同じ結果となっている．図 8 の全体を見るとローカルデバイスとリモートデバイスでは約 3.6ms 程の速度差となっており，この実験では従来方法と比べて約 3.1% ほどの増加ですんでいる．しかし，clSetKernelArg() ではローカルデバイスに比べて 39 倍，同様に clEnqueueNDRangeKernel() では 17 倍の処理時間がかかっている．

4.2.3 考　　察

実験の結果，リモートデバイスではローカルデバイスに比べ，約 3.1% 程度の速度低下が見られたが，この時間の大半は clSetKernelArg() と clEnqueueNDRangeKernel() の速度低下が原因である．clSetKernelArg() や clEnqueueNDRangeKernel() のレイテンシの一つの原因は OpenCL ランタイム API の戻り値，出力引数を同期的に取得していることにある．これは API 全てがエラーコードを返す仕様になっているためである．バッファの作成などにより OpenCL のリソースハンドルが戻り値として設定されている場合は仕方がないとしても，エラーコードについてはリモートホスト上の OpenCL ランタイムにエラーチェックをさせずに，送信前にローカルの OpenCL ランタイムにエラーチェックを行わせることで，エラーコードの取得を待たずに非同期に API を実行していく方法が考えられる．これにより，clSetKernelArg() などのエラーコードのみが出力となっている API に関しては非同期に実行可能となり，性能向上が見込める．

5. おわりに

本研究では，異なる OpenCL 実装のスケーラビリティーの向上，より多様な並列計算環境の提供，OpenCL アプリケーションの利用価値の向上などを目的として Hybrid OpenCL の開発を行った．Hybrid OpenCL では，既存の OpenCL ランタイムにネットワークレイヤを追加した Hybrid OpenCL ランタイムと，このネットワークレイヤと通信し，異なる OpenCL ランタイムをブリッジするブリッジプログラムを作成することで，ネットワーク上の OpenCL デバイスを扱えるように設計されている．今回，実装を行なったプログラムは約 1 万行となり，実際に動作するかを検証した後，NBody アプリケーションによる評価実験を行った．評価実験の結果，NBody アプリケーションにおいては従来方法と比べ，1 ステップあたり約 3.1% 程度の処理時間の増加で実装することができた．また，この速度低下の主な原因は OpenCL ランタイム API を同期的に処理している点が考えらるが，これらはエラーチェックなどを事前に済ませることで通信を減らし，性能を向上させることができると考えている．

今後はレイテンシが低いネットワーク環境下，例えば InfiniBand を用いての実験や，デバイスの数を増やしてのスケールアップ実験，異なる種類のデバイス，例えば，NVIDIA Tesla，ATI Radeon，マルチコア CPU などを組み合わせた実験，不要な通信を避ける工夫やプログラムのプロファイリング，高速化などを行い，Hybrid OpenCL システム全体のパフォーマンスを向上させていきたいと考えている．

参　考　文　献

1) Luebke, D., Harris, M., Krger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C. and Lefohn, A.: Gpgpu: general purpose computation on graphics hardware, *SIGGRAPH* (2004).
2) Munshi, A.: The OpenCL Specification, *Khronos OpenCL Working Group* (2009).
3) Munshi, A.: OpenCL Parallel Computing on the GPU and CPU, *SIGGRAPH* (2008).
4) Nyland, L., Harris, M. and Prins, J.: Fast N-Body Simulation with CUDA, *GPU Gems 3 - Chapter 31* (2008).