Vol.2010-ARC-192 No.22
Vol.2010-HPC-128 No.22
2010/12/17

情報処理学会研究報告
IPSJ SIG Technical Report

OpenCL互換アクセラレータのマルチノード環境における開発負担軽減のためのミドルウェアの実装

設 樂 明 宏[†1] 鎌 田 俊 昭[†1] 山 田 昌 弘[†1]
西 川 由 理[†1] 吉 見 真 聡[†2] 天 野 英 晴[†1]

GPU 等のアクセラレータを搭載したクラスタにおける並列プログラミングは，一般に CUDA や OpenCL 等のノード内における並列ライブラリと，MPI 等のノード間通信のライブラリを組み合わせることにより行われる．よって，2 種類の並列プログラミングの知識・技術の習得が不可欠であり，また，開発の過程において，OpenCL アプリケーションのノード間通信ライブラリ使用に対するコードの変換も必要である．
今回我々のグループは，ネットワーク上の複数のノードに搭載された OpenCL アクセラレータ (以下，アクセラレータ) が仮想的に一つのホストマシンに多数のアクセラレータが搭載されているかのように見せかけ，OpenCL のみで記述できる環境・ライブラリを提案する．このライブラリにより，プログラム開発者は OpenCL のみで開発を行うことができ，1 台のノード用に記述した OpenCL アプリケーションを容易にマルチノード環境で実行することが可能となる．本環境の性能評価の結果，仮想 OpenCL を用いて直交格子法による移流項を計算したところ，最大で 2 ノードを用いて 1.9 倍，3 ノードを用いて 2.5 倍性能の向上を確認した．

The implementation of development-support middleware on multiple-node environment of OpenCL Accelerator

Akihiro Shitara,[†1] Toshiaki Kamata,[†1]
Masahiro Yamada,[†1] Yuri Nishikawa,[†1]
Masato Yoshimi[†2] and Hideharu Amano[†1]

Programming on the cluster with accelerators like GP-GPU tends to be a mixture of intra node parallel library based on CUDA or OpenCL and inter node communication library including MPI. Programmers of such systems must manage two types of parallel programming techniques, and the code translation between for using inter node communication library in OpenCL applications.
Here, a programming environment which can treat multiple OpenCL accelerators attached to each node in a network as if they were connected to a single host machine. An application can be described with only OpenCL library, and programs written for a single node can be easily executed in multiple nodes each of which provides an accelerator. The evaluation results show that...
The evaluation shows that maximum of 1.9 and 2.5 times higher throughput can be obtained for orthogonal grid benchmark using virtual OpenCL environment on two and three nodes, respectively.

1. 背　　景

近年の高性能計算 (HPC) 分野において，汎用の PC 群に加えて Graphic Processing Unit(GPU) をはじめとする，数十〜数百個の演算コアから成るメニーコアアクセラレータを相互接続したヘテロジニアスクラスタが注目を集めている．これらのアクセラレータは，汎用 CPU と比べてピーク性能が高く，コスト対性能比や電力性能比などに優れ，実際に 2010 年 11 月のスーパーコンピュータの TOP500 ランキングでは，上位 5 台中 3 台が NVIDIA の GPU を搭載しているほか[1)]，PowerXCell 8i[2)] のように非対称型マルチコアプロセッサをアクセラレータとして利用するスーパーコンピュータも存在する．さらには，個人や中小規模の組織で構築・運営するクラスタのプラットフォームとしても期待されている．

一方，マルチコアプロセッサ向けのプログラム開発環境も進歩している．特に本来グラフィック用途の GPU を汎用計算に用いる General Purpose of GPU(GPGPU) は広く利用されており，GPU ベンダーの NVIDIA が C ライクなプログラミング環境 CUDA を提供している[3)]．さらに 2009 年には，異なるベンダーのマルチコアプロセッサの統合開発環境である OpenCL が登場し，共通の環境でソフトウェア開発を行うことが可能となった[4)]．

このような状況を受けて，ノード内のメニーコアアクセラレータに対して処理をオフロードすることで高い演算能力を得る，ヘテロジニアスな計算環境が多く登場している．ところが，これらを利用して高効率計算を実現するには，(1) 各プロセッサ独自のプログラミングモデルの理解，(2) ソースコードの最適化の定石などの学習，(3) データフローを考慮した対象問題における粗粒度な処理と細粒度な処理の切り分け，(4) コア間・ノード間のデータ通信の明示的な記述，など，実装には高度な技術を要する．

†1 慶應義塾大学理工学部 〒 223-8522 横浜市港北区日吉 3-14-1
Keio University, 3-14-1 Hiyoshi, Yokohama, 223-8522 Japan
†2 同志社大学理工学部 〒 610-0321 京都府京田辺市多々羅都谷 1-3
Doshisha University, 1-3 Tatara Miyakodani, Kyoto, 610-0321 Japan

© 2010 Information Processing Society of Japan

Vol.2010-ARC-192 No.22
Vol.2010-HPC-128 No.22
2010/12/17

そこで本稿において，我々は，プログラマに対してネットワーク上の複数ノードにそれぞれ搭載されたアクセラレータを仮想的に 1 つのノード内に存在するかのように見せ，多数のノード上のアクセラレータを用いた OpenCL プログラミングを行うことのできる仮想化環境を提案する．

この環境により，以下の利点が得られると考えられる．

- プログラマビリティの向上: OpenCL のプログラミング技術の習得によりネットワーク上の多数の演算ノードに搭載された OpenCL 互換のアクセラレータを利用でき，MPI やその他のノード間通信ライブラリの習得および実装が不要となる．
- 柔軟性: OpenCL 互換のアクセラレータであれば，デバイスの種類を問わず，ネットワーク上のノード数に応じてアクセラレータを指定し柔軟にプログラムを実行することができる．
- 高互換性: 既存の OpenCL アプリケーションプログラムを，極めて少ない変更で多ノードで実行できる

以上を実現し，動作検証 · 評価するにあたり，本研究報告では NVIDIA の GeForce GPU を対象に，GPU を搭載したネットワーク上の複数台の PC 利用して OpenCL アプリケーションを実行する環境の構築について述べる．

なお，当仮想化環境は，LinuxOS の socket 通信を用いており，ノード間通信の遅延の影響によりノード間データ転送量の多いアプリケーションにおいては性能が向上しなかった．当環境の現時点での限界と今後の改善点についても議論する．

以後，第 2 章で関連研究を述べ，第 3 章では OpenCL プログラミングの概念と GPU を用いたクラスタにおける典型的な並列処理について述べる．第 4 章では第 5 章では仮想化環境構造および今回実装したミドルウェアの設計について述べる．第 6 章では評価用アプリケーションを用いた性能評価について述べる．最後に第 7 章で，得られた結果を元に結論を述べる．

2. 関連研究

我々が提案している仮想化環境はネットワーク上の GPU を計算資源として活用するものである．

関連研究として，ネットワーク上ノードの GPU を用いたグリッドコンピューティング環境が挙げられる[5]．これは，ネットワークに接続されている NVIDIA の GPU を搭載したアイドル状態の PC を検出し，GPU のタスクを投入する．GPU を用いたグリッドコンピューティング環境は, 既に科学計算分野で実際に利用されている[6]．

また我々のグループでは，PlayStation3 を用いたクラスタを構築し，その上でのプログラミング開発を支援するスレッド仮想化環境を開発している[7]．これは，ネットワーク上に接続された複数ノードの演算コアを仮想的に単一プロセッサ内にあるかのように見せかけることで多数の演算コアでのマルチスレッドプログラミングを可能にしている．そのため，MPI 等のノード間通信ライブラリを必要としない．

本研究の特徴は

- 既存研究が同一プロセッサないし同一ベンダーのアクセラレータを対象としているのに対し，OpenCL を用いることで異種ベンダーのアクセラレータが使用可能である
- 複数ノード上のアクセラレータが，単一ノード上に搭載されているように見える

の 2 点が挙げられる．

3. GPU アーキテクチャ

本研究では，SOM のプラットホームとなる GPU として，Geforce GTX280, および 9500GT を用いる．GTX280 は GT200 シリーズ，9500GT は G80 シリーズに大別され，特に前者は倍精度浮動小数点演算をサポートしていることが特徴である．その点を除き，両シリーズはハードウェア構成および制御方式が類似しているため，ここでは GT200 シリーズを例にとって解説する．

3.1 グラフィックプロセッサアーキテクチャ

3.1.1 Texture Processor Cluster (TPC)

図 1 に，NVIDIA 製グラフィックカードに搭載されている GT200 シリーズデバイスのアーキテクチャの概観を示す．また以下に主な構成を述べる．

- Texture Processor Cluster (TPC)
- スレッド実行マネージャ
- デバイスメモリ
- L2 キャッシュ

図 1 における Texture Processor Cluster (TPC) は，複数のコアを持つ演算処理部である．GT200 シリーズには GTX280 と GTX260 プロセッサがあり，前者は TPC を 10 個，後者は 8 個持つ．各 TPC 内部には，Texture Filtering Unit(図中の TF) および複数個の Streaming Multiprocessor(SM) を持つ．各 SM の構成を以下に述べる．

- Streaming Processor(以下，SP)(8 個): 単精度型浮動小数演算パイプライン

© 2010 Information Processing Society of Japan

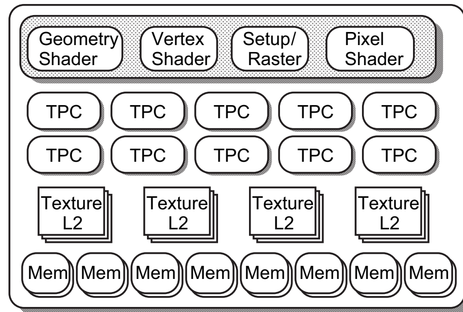

図 1　GT200 シリーズの構造

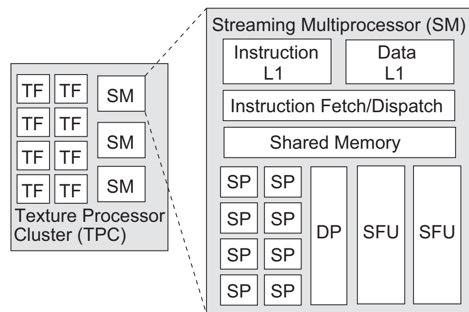

図 2　Texture Processor Cluster(TPC) の構造

- Special Function Unit(SFU)(2 個): 超越関数計算用演算パイプライン
- DP(1 個): 倍精度型浮動小数演算パイプライン
- 共有メモリ: 8 個の SP で共有されるオンチップ メモリ
- 命令およびデータ用 L1 キャッシュ
- プログラムカウンタ

4. スレッド 仮想化環境

アクセラレータを搭載したマルチノード環境における，通常の OpenCL プログラミングの概念を，MPI を用いた場合を例にとり**図 3** に示す．

通常，OpenCL プログラミングをマルチノード環境で行う場合，第一に，1 台のノード上で動作する OpenCL プログラムを，ホスト側とアクセラレータ側のそれぞれに実装する．アクセラレータ上で動作するソフトウェアは，各アクセラレータの持つキューにタスク投入することで実行される (図 3 (1))．1 台のホストに複数のアクセラレータが搭載されている場合は複数のキューが存在し，各キューに対してタスクを投入する．続いて，プログラマは MPI ライブラリを用いてホスト側プログラムを拡張する形でノード間の通信制御を実装する (図 3 (2))．この通常の手法において，プログラマは OpenCL と MPI を組み合わせたソースコードを記述する必要があるため，高いプログラミング技術を要し，デバッグも困難である．

そこで本研究では，ホストとなる 1 台のノード上の CPU プロセスから，他ノード上のアクセラレータの持つキューに対してタスクを投入できるミドルウェアを開発した．これによ

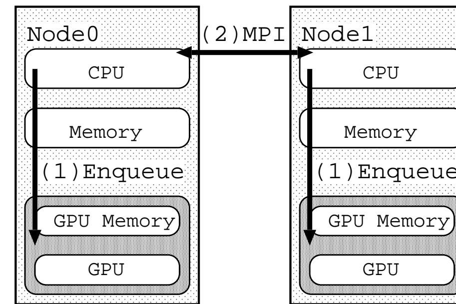

図 3　MPI と OpenCL を組み合わせた通常のプログラミングの概念

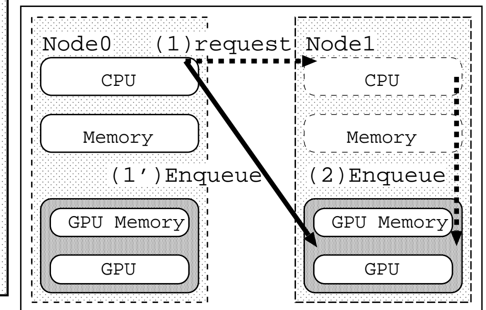

図 4　仮想化時の OpenCL プログラミングの概念

表 1　GPU の諸元

GPU name	GeForce GTX280	GeForce 9500GT
マルチコア数	30	8
マルチコア数/TPC	3	2
コアクロック (GHz)	1.27	1.40
レジスタ/コア	64KB	32KB
共有メモリ/コア	16KB	16KB
メモリクロック (GHz)	1.1	0.8
メモリバス幅 (bit)	512	128
メモリプロセッサ間バンド幅 (GB/sec)	141.7	25.6
メモリ容量	1GB	512MB

© 2010 Information Processing Society of Japan

り，プログラマは MPI ライブラリを用いることなく，他ノードのアクセラレータを用いて OpenCL アプリケーションを実行することができる．

プログラマが記述した OpenCL アプリケーションが**図 4** の Node0 上で実行されるときの，仮想化環境のイメージを図 4 に示す．アプリケーションは Node1 上のサーバプロセスに対し，仮想通信用の API を用いてリクエストを送信する (図 4(1))．そして，リクエストを受信した Node1 のサーバプロセスは，自身のホスト上のアクセラレータのキューに対してタスク投入する (図 4(2))．このとき通信用 API は，内部で Node1 にデータ転送を仲介しているため，通信用 API を実行する Node0 のユーザアプリケーションに対して Node1 のアクセラレータのキューがあたかも直接タスク投入しているかのように見せることができる (図 4(1'))．

5. 設　　計

本研究の環境では，ユーザアプリケーションから他ノードのアクセラレータを制御する．これを実現するために，ユーザアプリケーションがリクエストを送信する仮想 OpenCL API および，各ノード上でリクエストを受けてアクセラレータ制御する仮想 OpenCL サーバプログラムを LinuxOS の socket 通信を用いてそれぞれ実装した．関数は OpenCL API のうち，GPGPU で多く用いられる基本的な約 30 種類を実装した．本章では，これらの設計について述べる．

仮想 OpenCL API の関数が呼び出されるとき，ユーザアプリケーションとサーバプログラムは**図 5** に示される動作をする． ユーザアプリケーションは，ノードの仮想 OpenCL サーバプロセスに OpenCL 関数のリクエストとその引数を送信する．ノード上のサーバプロセスは，ホストからの受信データを元に OpenCL 関数を呼び，実行結果をサーバから受信する．

次に仮想 OpenCL API とサーバプログラムの設計について述べる．

5.1 仮想 OpenCL API の実装

仮想 OpenCL API は，プログラマが実装するアプリケーション上で呼ばれる．これらの関数群は，通常の OpenCL の API と関数名や引数の型名がほぼ同じであるように実装されている．ユーザアプリケーション内で仮想 OpenCL API の中のある関数が呼ばれた時，その関数は他ノード上仮想 OpenCL サーバプロセスで実行される通常の OpenCL API を表す識別データとその引数となるデータを転送する．仮想 OpenCL API は他ノード上での Open CL 関数の実行結果や出力値を受信し，値をユーザアプリケーションに渡す．

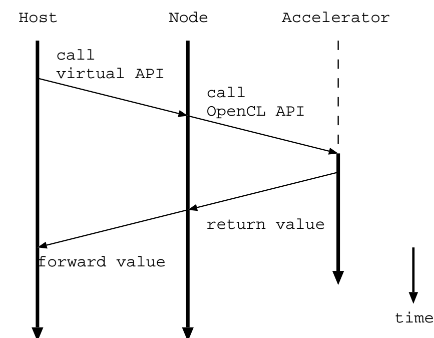

図 5　他ノードへの仮想 OpenCL リクエストの送信

表 2　OpenCL 特有のデータ型とその役割

データ型	役割
cl_platform_id	CUDA や AMD Stream 等のプラットフォーム
cl_context	OpenCL コンテキスト
cl_program	アクセラレータ上のソフトウェア
cl_mem	アクセラレータ上のメモリ領域
cl_device_id	アクセラレータデバイス
cl_kernel	アクセラレータ上のコンパイル済みソフトウェア
cl_command_queue	アクセラレータのキュー
cl_event	カーネル実行やデータ転送等のイベント

5.1.1 サーバ上データのユーザアプリケーションでの保持方法

通常の OpenCL プログラムでは，デバイスやキュー等の識別子データを保持するために OpenCL 特有のデータ型が用いられ，OpenCL の API の引数として指定される．これらのデータ型をユーザアプリケーションで生成される順に，**表 2** に示す．

これらのデータ型は OpenCL のヘッダファイル内で定義されているが，詳細は非公開である．しかし仮想 OpenCL API を実行するユーザアプリケーションは， 複数のノードの持つこれらの型の値を保持する必要があるため，以下の方針を適用した．

- `cl_mem`: メモリ領域へのポインタであるため，ユーザアプリケーション内においてもポインタとして値をそのまま保持する
- `cl_device_id`，`cl_kernel`，`cl_command_queue`，`cl_event`: これらの型で表される 1 つのデータは，1 ノードと一対一対応する．そのため，例えば `cl_device_id` 型の場

© 2010 Information Processing Society of Japan

合，cl_device_id 型に対応した vcl_device_id 構造体を用意する．そして，サーバでの値とデバイスを搭載するサーバに対するソケット識別子を構造体のメンバとして保存し，vcl_device_id へのポインタを cl_device_id として定義する．仮想 OpenCL API 内で呼ばれた cl_device_id はメンバを参照し，ソケット識別子を元にノードを特定してサーバ上で引数となるデータを転送する．

- cl_platform_id，cl_context，cl_program: これらも上記と同様だが，ホストアプリケーションにおいて，これらはあたかも 1 つのノード内に存在する 1 つのデータであるかのように表す必要がある．よって，各ノード上での値とノードのソケット識別子をノードごとに構造体として保持し，構造体同士を線形リストとしてつないで先頭へのアドレスとして表した

5.1.2 アクセラレータ上のプログラムのコンパイル

通常の OpenCL 環境において，アクセラレータで実行されるソフトウェアはホストアプリケーション内でソースコードのまま読み出され，オンラインコンパイラによってコンパイルされる．このことを利用し，ユーザアプリケーションでソースコードを読み出して他ノードに転送し，各ノード内でコンパイルを行う．ノードごとに異なる種類のバイナリデータを実行するアクセラレータが混在する環境においても，それぞれの環境のオンタイムコンパイラをサーバプロセス内で呼ぶことによってアクセラレータごとのバイナリを生成して実行することが可能となる．これにより，アクセラレータ間のソフトウェアの高い互換性を実現する．

5.1.3 仮想 OpenCL API の実行方法

通常の OpenCL 環境においてアプリケーションを開発するとき，プログラマは通常の OpenCL API でホストのプログラムを記述し，OpenCL C を用いてアクセラレータのプログラムを記述する．これらのプログラムを仮想 OpenCL 環境で実行する際の変更点を述べる．

アクセラレータ上のプログラムは，1 ノード，1 アクセラレータの環境用に記述されている場合，1 ノードに複数のアクセラレータが搭載されているときのプログラムに書き換える必要がある．これは CUDA や通常の OpenCL 環境での開発と同様である．しかし，既に 1 ノード内に複数のアクセラレータが存在する場合のプログラムを記述している場合，ユーザアプリケーションからは仮想 OpenCL 環境では他ノードのキューが見えることになるため，アクセラレータ上で動作するプログラムは変更する必要はない．

ホスト上のプログラムには 2 点の変更点があるのみである．

- ヘッダファイルの変更: ホストプログラムで読み込む OpenCL ヘッダを<CL/cl.h>を我々の開発した環境のヘッダに書き換える点である．これにより，通常の OpenCL の API と同名の仮想化関数が呼出される
- clCreateMemObject() と clCreateKernel への引数の追加: これらの関数は，通常の OpenCL においてコンテキストのみを指定することのみで呼出されるが，仮想 OpenCL 環境において cl_context 型のコンテキストは，あたかも 1 つのノード内に存在する 1 つのデータであるかのように表しているため，リクエストを送信するノードとデータが 1 対 1 対応ではない．そのため，cl_device_id 型のデバイス ID を指定することにより，ノードを特定する

これらの変更を必要とする箇所はプログラムの初期段階で呼び出されることが多いため，プログラマには大きな負担とならないと考えられる．

5.2 仮想 OpenCL サーバの実装

ノード上で常駐する仮想 OpenCL サーバプログラムは，ユーザアプリケーションを実行するノードから OpenCL API の実行のリクエストと引数となるデータを受信する．受信されたデータを元に，サーバプログラムは対応する関数を実行し，結果を送信する．

6. 評　価

この章では，仮想 OpenCL 環境の性能評価について述べる．ベンチマークアプリケーションとして直交格子法による移流項の計算プログラム[8] を使用した．

評価は**表 3** に示す，各ノードに 1 枚ずつ NVIDIA GeForce シリーズの GPU を搭載した 3 種類 4 台のノードを用いて行った．また，ノード間接続には Gigabit Ethernet を用いた．

表 3 評価環境

	Node0	Node1, Node2	Node 3
CPU	Intel Core2Quad	Intel Core2Quad	Intel Core2Duo
CPU Clock (GHz)	2.40	2.83	2.40
GPU	GeForce GTX280	GeForce 9500GT	GeForce 9500GT
Host Memory	4GB	8GB	4GB
OS	CentOS 5.5	CentOS 5.5	CentOS 5.5
Host Compiler	gcc 4.1.2	gcc 4.1.2	gcc 4.1.2
CUDA Version	Toolkit 3.2rc	Toolkit 3.2rc	Toolkit 3.2rc
OpenCL Version	OpenCL1.1	OpenCL1.1	OpenCL1.1

© 2010 Information Processing Society of Japan

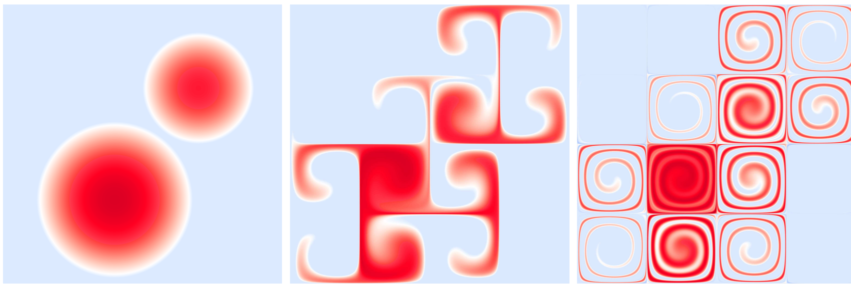

図 6　直交格子法による移流項の計算

6.1　直交格子法による移流項の計算

直交格子法による移流項の計算は，流体の計算の一種で，インクの初期濃度分布と速度場が与えられた時に，格子に区切られた領域のインクがどう移動するか (移流計算の値) を，1 スレッドにつき 1 つの格子に対してステップごとに並列に計算する．計算の様子を**図 6** に示す． 直交格子法の計算では，格子の次のステップの値を計算するのに近傍の格子同士にデータ依存性があるものの，ダブルバッファリング，まとまった区間の格子データを GPU のオンチップのメモリ領域に格納すること，さらに，ホスト-GPU 間の協調による境界データの計算を行うことによって高い性能を得ることができる．本研究では CUDA の最適化された実装を OpenCL に移植し，用いた[9]．

6.2　性 能 測 定

各ベンチマークにおいて，仮想 OpenCL を用いて次のような条件下で実行時間を測定した．

(1)　Node0 のみで CUDA 環境のみで実行したとき (仮想 OpenCL を用いない),
(2)　Node0 のみで OpenCL 環境のみで実行したとき (仮想 OpenCL を用いない),
(3)　Node1 のみで OpenCL 環境のみで実行したとき (仮想 OpenCL を用いない),
(4)　Node1 から Node1 自身の GPU に対してタスクを投入したとき
(5)　Node1 から Node2 の GPU に対してタスクを投入したとき
(6)　Node1 から Node1, Node2 の GPU に対してタスクを投入したとき
(7)　Node1 から Node1~Node3 の GPU に対してタスクを投入したとき

これらの条件下での，直交格子法のプログラムを実行する問題サイズを**表 4** に示す．GPU のメモリ領域は X 方向に連続で，Y 方向で GPU の枚数だけ小さい区間に分割して，それぞれの GPU で次のステップのインクの値を計算する．

表 4　各問題の問題サイズ

問題番号	X	Y
1	256	256
2	1024	1024
3	4096	2048
4	2048	2048
5	4096	4096
6	4096	4096

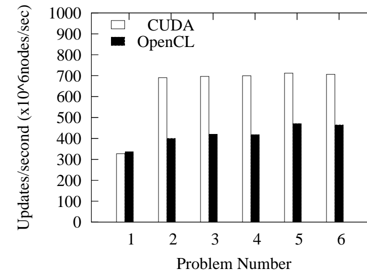

図 7　CUDA と OpenCL における計算性能の比較

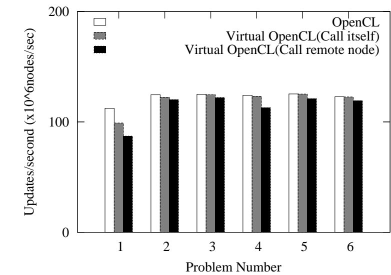

図 8　OpenCL と仮想 OpenCL における計算性能の比較

ここで縦軸は単位時間あたり更新された格子の数であり，単位を $\times 10^6$ 個/秒とする．

6.3　CUDA，OpenCL，仮想 OpenCL の環境による性能比較

それぞれの実行環境における各計算問題の性能を**図 7**，**図 8** に示す．

(1) と (2) のそれぞれの条件において，CUDA と OpenCL の異なる環境による実装と性能の関係を図 7 で比較した場合，CUDA 環境での実装の方が高い性能を示す．プログラムは 1 枚の GTX280 上で実行されているため，GPU 間のデータの交換は行われず，ホスト-アクセラレータ間の同期はステップごとのダブルバッファリングの切り替え時のみに発生する．つまり，OpenCL ではホストがアクセラレータの計算を終了する前に関数呼び出しから戻ってくることで，プログラマは CUDA よりも複雑にホスト-アクセラレータが協調したプログラムを記述することができるが， 1 枚 GPU での実装においては複雑な協調は必要ない．我々の行った CUDA から OpenCL への移植は，アクセラレータ上のプログラムとそれを呼び出すホストプログラムの関数やデータ型，予約語等を置き換えただけであるため，プログラムの構造上の大きな差はない．したがって，図 7 の性能差はアクセラレータの

© 2010 Information Processing Society of Japan

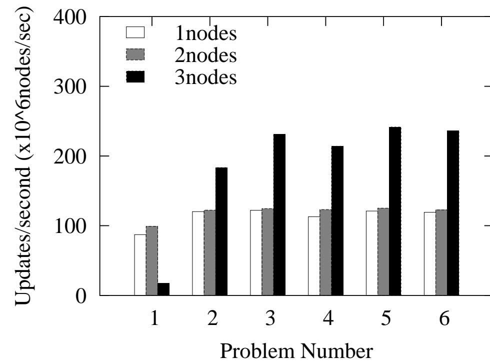

図 9　各問題条件における実行環境と計算性能の関係

プログラムを実行する環境の違いによるものであり，OpenCL 環境では GPU 上の計算に時間を要すると考えられる．

ネットワークを介した OpenCL アクセラレータのもつキューへの仮想的なタスク投入によるオーバーヘッド図 8 によって比較することができる．(3) と (4) において，OpenCL と仮想 OpenCL，それぞれの環境における性能を比較したとき，性能差は小さい．これは，本ベンチマークの計算用データが GPU のメモリ内に全て格納され，ホスト-GPU 間でデータの転送を行わないためである．次に (4) と (5) を比較する．(5) の条件下においても計算用数値データはノード間で転送されない．本環境では，2 台のノード間の通信の遅延は少なく，カーネルの実行やホスト-アクセラレータ間の同期のリクエスト時のデータサイズも数 10 バイト程度であるため，ネットワークを介した仮想的なキューへのリクエストによるオーバーヘッドは，アクセラレータ上で計算を行う時間と比べて小さいと言える．

6.4 ノード数と性能の関係

次にノード数に対する性能向上を評価する．タスクの投入をリクエストするノード数と実行時間の関係を**図 9** に示す．問題 6 において，最大で 2 ノードを用いて 1.9 倍，3 ノードを用いて 2.5 倍性能の向上が得られた．

ノード数が増加するとき，同じサイズの問題であっても，境界データの交換のための転送とホスト-GPU 間の同期が行われる．ノード数が 2，3 と増えたとき，問題サイズが大きいとき，複数ノード上の GPU を用いたことによる性能向上が確認できる．しかし，問題 1 に着目したとき，タスクを投入するノード数が増えるごとに性能は低下する．問題 1 の全体の問題サイズは 256 と小さいため，GPU 間の境界データの交換や同期に必要となる時間が大きい[9]．同様の現象は 6.2 に示した問題 2 においても，(6)，(7) の条件下でのノード数増加においても確認できる．

図 9 において，問題 3~6 に対して (5)~(7) の条件を比較する．1 ノードから 2 ノードに増加したとき，性能も約 2 倍に増加していることが確認できる．一般的に直交格子法はメモリアクセスがボトルネックとなるため，性能の向上が 2 倍に達することは少ない．しかし，OpenCL 環境で実装した場合，カーネルの実行には時間を要することが

しかし，1 ノードから 3 ノードに増加したときは性能は 3 倍には達しない．これは，1 ノードの GPU あたりの問題サイズが小さくなり，計算時間が減少する一方，サーバプログラムを実行するノードにおける 1 ノードあたりのデータ転送量が変わらないためである．すなわち，1 枚の GPU が次のステップの値を計算するのに必要な計算時間が減少する一方，1 組のノード間で境界領域のデータを交換するデータサイズは変わらず，ユーザアプリケーションから見たとき，ユーザアプリケーションを実行するノードのメモリを介して GPU 間で交換するデータの総量は増加すると考えられる．

7. 結　　論

本研究では，OpenCL と MPI を組み合わせたプログラミングの問題を解決するために，他ノード上の OpenCL アクセラレータに対してタスクを投入できるミドルウェア，仮想 OpenCL 環境を実装し，評価した．仮想 OpenCL を用いて直交格子法による移流項を計算したところ，最大で 2 ノードを用いて 1.9 倍，3 ノードを用いて 2.5 倍性能の向上が得られた．

8. 今後の展望

本稿で用いた GPU はグラフィック用途で用いるエントリーモデルであるため，今後の展望としては，より多くの演算コア，大きなサイズのメモリを搭載した GPU による評価を検討している．また，より多くのノード数の増加による性能向上の評価と，ネットワークにおけるデータ通信の性能評価，モデル化も検討している．

参　考　文　献

1) Top500: "Supercomputer sites". "http://www.top500.org/system/10377".
2) Kistler, M., Gunnels, J., Brokenshire, D. and Benton, B.: Programming the Lin-

© 2010 Information Processing Society of Japan

pack benchmark for the IBM PowerXCell 8i processor, *Scientific Programming*, Vol.17, No.1-2, pp.43–57 (2009).

3) NVIDIA: NVIDIA CUDA Compute Unified Device Architecture (2008).

4) NVIDIA: The OpenCL Specification Version: 1.0 (09).

5) Kotani, Y., Ino, F. and Hagihara, K.: "A Resource Selection System for Cycle Stealing inGPUGrids", *Journal of Grid Computing*, Vol.6, No.7, pp.399–416 (2008).

6) : "Folding@home distributed computing". "folding.stanford.edu".

7) 山田昌弘，西川由理，吉見真聡，天野英晴：Cell Broadband Engine を用いたスレッド仮想化環境の提案，信学技報， Vol.110, No.2, pp.27–32 (2010).

8) GPU チャレンジ 2010 実行委員会："GPU Challenge 2010 規定課題マニュアル (ツールキット ver.0.60 対応版)". "http://www.hpcc.jp/sacsis/2010/gpu/".

9) 須藤郁弥，坂内恒介，本田耕一，松田健護， 篠原歩：2GPU による Cubic セミ・ラグランジュ法の高速化，*SACSIS2010 GPU Challenge 2010* (2010).

© 2010 Information Processing Society of Japan