Vol.2012-HPC-133 No.9
2012/3/26

並列言語 XcalableMP のアクセラレータ向け言語拡張の OpenCL 実装

野水拓馬[†1]　高橋大介[†2,†3]　李珍泌[†1]
朴泰祐[†2,†3]　佐藤三久[†2,†3]

高い演算性能を持つ GPU や Intel MIC 等をアクセラレータデバイスとして汎用計算に用いる動きが HPC の分野で活発である，さらに，これらのアクセラレータを搭載したクラスタにおけるプログラミングも注目されている．しかし，アクセラレータを搭載したクラスタ上でプログラムを記述するためには，アクセラレータ専用の記述言語の他に，MPI に代表されるノード間通信インターフェースも用いる必要がある．このような分散並列環境におけるプログラミングの複雑さを解決する手段として，PC クラスタ向けの PGAS 言語である XcalableMP が開発されている．さらに，GPU を搭載したクラスタ向けに，CUDA を用いて XcalableMP の言語拡張も開発されている．本稿では，XcalableMP のアクセラレータ向け言語拡張を OpenCL で実装し，NVIDIA の GPU，AMD の GPU それぞれを搭載したマシンにおいて N 体問題と行列積を用いて評価を行った．その結果，元々CUDA により実装されていた XcalableMP のアクセラレータ向け言語拡張と同等の性能が得られることを確認し，さらに，OpenCL 実装によって XcalableMP で記述したソースコードが異なる計算プラットフォーム間で可搬性を持つことを示した．

1. はじめに

近年，Graphic Proccesing Unit（GPU）等のアクセラレータデバイスを用いた汎用計算が High Performance Computing（HPC）で注目されている．

アクセラレータを用いることにより高い演算性能を得ることが可能であるが，専用のプログラミング環境が必要となる．ベンダーごと，アクセラレータの種類ごとで異なるプログラムを記述する必要があり，ユーザはそれぞれの記述方式を学ぶ必要がある．こういった開発環境の多様化という問題に対し，アクセラレータの種類に依存しないプログラミング環境 OpenCL[1] が Khronos 社と Apple 社によって仕様策定された．OpenCL を用いることで，アクセラレータを用いた演算におけるプログラムの可搬性が期待できる．

さらに，GPU クラスタのようなマルチノード環境において GPU を用いて高い性能を得る動きも活発である．しかし，GPU クラスタのような分散メモリ型並列環境においてプログラミングを行うためには，ノード間通信に Message Passing Interface（MPI），ノード内のアクセラレータに対しては専用の記述言語を用いる必要があり，ユーザに多種の知識が求められ，プログラミングが非常に困難となっている．

こういったプログラミングコストの問題に対し，指示文を追加することでノード間通信を容易に記述できる Partitioned Global Address Space（PGAS）[2] 言語 XcalableMP[3]（以下 XMP と記す）が提案されている．逐次コードに追加された指示文を XMP コンパイラが展開し，ノード間の通信，及びループ文の並列化を行う．また，XMP のアクセラレータ向け言語拡張である XcalableMP Acceleration Device Extension[4]（以下 XMP-dev と記す）も開発されている．

XMP-dev の言語仕様は様々なアクセラレータを対象としているため，XMP-dev を用いることで GPU クラスタ上においても容易なプログラミングが可能となる．しかし，XMP-dev の実装は NVIDIA の GPU の開発環境である Compute Unified Device Architecture（CUDA）[5] で行われている（簡略のため本稿では CUDA により実装されている XMP-dev を XMP-dev/CUDA と表記する）ため，動作する対象は NVIDIA の GPU 上でのみとなっている．

そこで，本研究では XMP-dev の OpenCL 実装（簡略のため XMP-dev/OpenCL と記す）を行った．本稿では XMP-dev/OpenCL を用いて N 体問題の性能を測定し，NVIDIA の GPU と AMD の GPU に両方の計算プラットフォームにおいて同じソースコードでプログラムを実行することができることを確認した．さらに，GPU クラスタ上でノード数の増加に従った性能のスケールも確認できた．

2. XMP-dev

ここでは，本稿で実装した拡張の元となる XMP-dev について簡単な概要を述べる．XMP-dev は，並列言語 XMP のアクセラレータ向け言語拡張であり，GPU を搭載した分散メモリ型並列計算機上で使える生産性の高いプログラミングモデルである．また，XMP の言語

†1 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
†2 筑波大学システム情報系
Faculty of Engineering, Information and Systems, University of Tsukuba
†3 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba

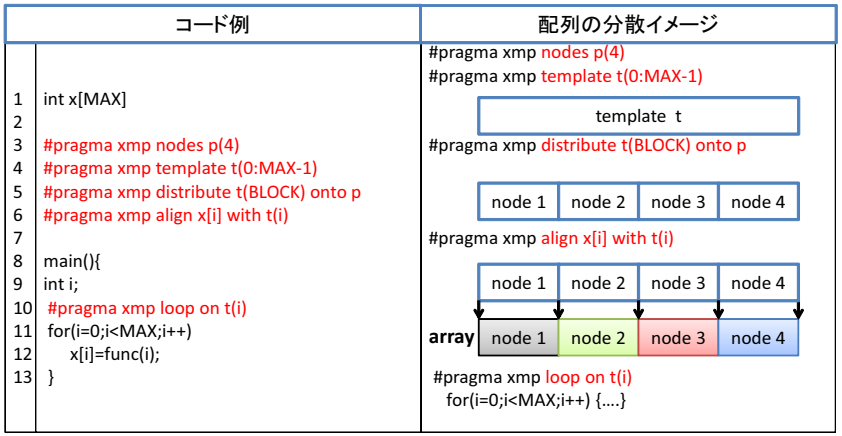


図 1 XMP sample code

仕様は XMP-dev の中でもそのまま利用することができる．逐次コードに指示文を追加することで，アクセラレータを用いたスレッド並列化を簡単に記述する事ができる．

2.1 XMP

XMP の実行モデルについては文献[6] が詳しいが，ここでは XMP で用いられるいくつかの指示文をループ文の並列実行を例に挙げて述べる．XMP は OpenMP[7] のように，データ並列化やタスク並列化で見られる典型的な処理を指示文で記述することにより，逐次コードから少ない変更で並列化を行うことが可能である．しかし OpenMP と異なる点として，分散メモリ型並列計算機をターゲットとした仕様となっているため，XMP の指示文によって分散メモリ型並列計算機でのデータの分散やノード間通信を行うことが可能である．

図 1 に XMP によるデータ並列化のコード例を示す．XMP ではまずノードの集合を定義したのち，データの分散やループ文のワークシェアリングを template と呼ばれる仮想的な index 配列を用いて行う．align 指示文を用いてデータの分散を記述し，loop 指示文でループ文の並列化を記述する．ループの並列実行は，template を実配列にマッピングすることでローカルノードのデータのみ参照し，全ノードで同じプログラムが実行される．

2.2 XMP-dev のプログラミングモデル

ここでは XMP-dev のプログラミングモデルをループ文の並列実行を例に挙げて述べる．XMP-dev は前述の通り XMP のアクセラレータ向け言語拡張であり，XMP で分散された

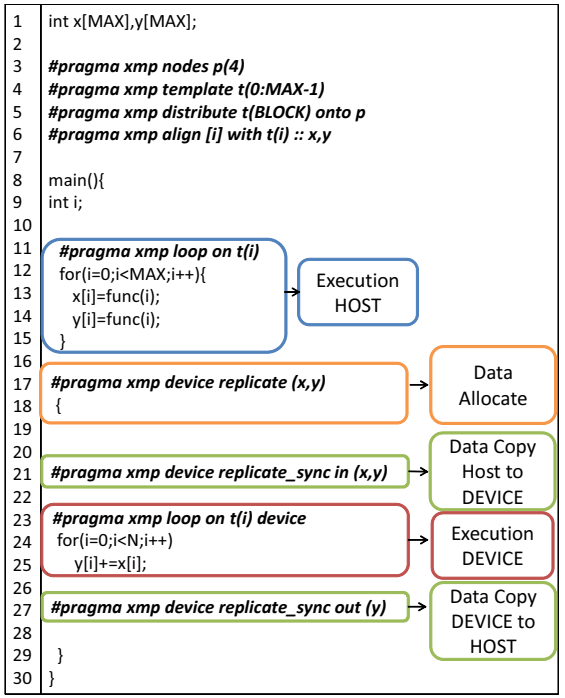


図 2 XMP-dev sample code

配列を各ノードに搭載されている GPU で並列実行させる．

図 2 に XMP-dev によるデータ並列化のコードを示す．キーワード device を含む行が XMP-dev の指示文である．GPU には専用のメモリ（以下デバイスメモリと呼ぶ）が存在するため，GPU に計算を行わせるためにはデバイスメモリでのデータ宣言及びホストメモリからデバイスメモリへの転送を記述する必要がある．XMP-dev では device replicate 指示文でデバイス上でデータの宣言を行い，device replicate_sync 指示文でホスト・デバイス間のデータ通信を記述する．device replicate 指示文で宣言されたデータは，図 2 において 18-29 行目で記述された中でのみ有効である．必要な時のみデータの宣言と開放を行うことで，デバイスメモリを効率よく使うことができる．

23 行目に示す device loop 指示文でループ文の GPU 上での並列実行を記述している．こ

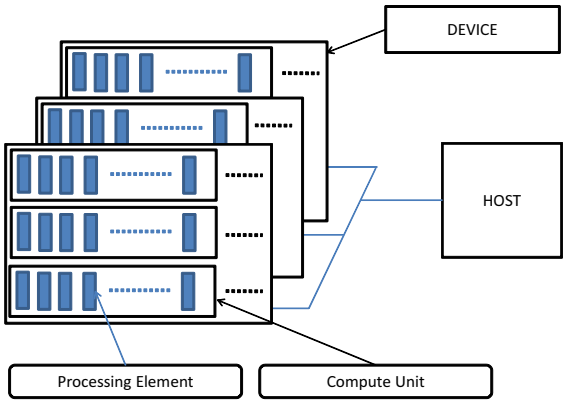


図 3 OpenCL platform

の時，ループのイテレーションがコンパイラによって GPU 上の各スレッドに割り当てられ，ループ内のコードが GPU 上で並列実行可能なコードに変換される．

このように，指示文を追加することで GPU 上での並列実行を可能としている．XMP にはこの他にもリダクションや shadow 領域を用いた通信等のための指示文が実装されており，詳細は文献[6] に記してある．

3. OpenCL

ここでは本稿で実装した拡張仕様に用いた OpenCL の概要を説明する．現在 GPU を用いたプログラムを開発する環境として，NVIDIA から CUDA が，AMD から AMDPP が提供されている．CUDA と AMDAPP は互いに C 言語を拡張した言語であるが，予約語や文法が異なるため，各ベンダの GPU でプログラミングを行うためには，それぞれ別なソースコードを記述する必要がある．このようなソースコードの可搬性の問題に対し，ベンダ及びアクセラレータに依存しない開発環境として OpenCL が開発された．

図 3 に示すように，OpenCL が対象とする計算プラットフォームは制御用ホストと演算用デバイスという二つに分類される．ホストは CPU が担当し，OpenCL の API を呼び出し，デバイスの選択，データの通信，カーネルの発行等の演算制御を行う．デバイスは GPU 等のアクセラレータが担当し，並列計算のためのカーネルを実行する．OpenCL はホストとデバイスが存在するあらゆる計算プラットフォームを対象としており，かつホスト・デバイス間の通信方法も定めていないため，PCI-express だけでなく Ethernet を介したホスト・デバイス間通信なども理論上は可能と考えられる．このように，OpenCL は GPU だけでなく，多種の演算デバイスに対しても同じソースコードを用いて制御することが可能であり，プログラムの再利用性，及び開発効率の改善が期待できる．

4. 本研究の実装事項

ここでは XMP-dev/OpenCL について，CUDA と OpenCL の仕様の差異及び追加したランタイムライブラリ及び関数，そしてコンパイル時の流れについて述べる．

4.1 ランタイムライブラリの追加

OpenCL が CUDA と異なる点として，まずコンテキストの概念が挙げられる．コンテキストは OpenCL の実行モデルにおいてデバイスを扱うための情報群である．OpenCL でプログラムを実行する際，1 つのデバイスに対し 1 つのコンテキストが作成され，デバイスの参照やデータの宣言，デバイスで動作するプログラムの管理が行われる．CUDA においてデバイスの参照はドライバに任せることが可能であるが，OpenCL では自ら明示的に指定しなければならない．そのため，XMP-dev/OpenCL では platform の取得，コンテキストの作成，デバイスの取得に関する初期化ランタイムを追加した．

さらに，コンテキスト内におけるコマンドキューの作成も OpenCL 独自の仕様として挙げられる．デバイスを扱うすべてのコマンドはキューに入れられ，順に実行される（非順序実行も可能である）．複数デバイスを扱うためには複数のコマンドキューをコンテキスト内に作成する必要がある．今回の実装では 1 つのコマンドキューを作成し，1 デバイスのみ扱う仕様となっているが，今後の拡張仕様として複数コマンドキューを立てて GPU と CPU 両方に計算を行わせるといったことも考えられる．

また，CUDA と OpenCL のプログラミングモデルで類似している点として，ホスト側で実行するホストコードとデバイス側で実行するデバイスコードの 2 つに分けて記述するという点が挙げられる．共にホストコードは C 及び C++で記述が可能であり，デバイスコードは C 言語ライクな拡張言語を用いて記述を行う．ホストコードにおいてデータの宣言やデータ転送，カーネル実行等の記述にはランタイム API を用いるため，**表 1** に示すように，CUDA と OpenCL では異なる関数を呼ぶ必要がある．

XMP-dev/CUDA，XMP-dev/OpenCL は共に並列化後のコードにおいて，表 1 に示すそれぞれのランタイム API のラッパー関数を呼ぶ．本研究では OpenCL のランタイム API を実行するラッパー関数の追加を行った．また，OpenCL ではカーネル関数の引数を明示的

表 1 CUDA と OpenCL のランタイム API の一例

関数の用途	CUDA	OpenCL
メモリの確保	cudaMalloc()	clEnqueueCreateBuffer()
データ転送	cudaMemcpy()	clEnqueueWriteBuffer()
カーネル実行	kernel<<<grid,block>>>()	clEnqueueNDRangeKernel()
同期	cudaThreadSyncronize()	clFinish()

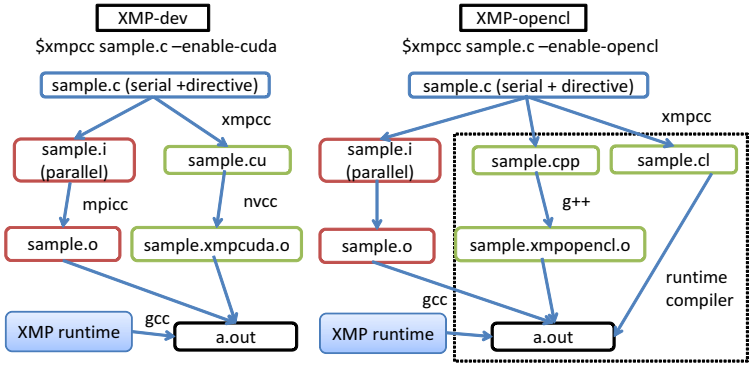


図 4 コンパイルの流れ

に API で指定する必要がある．そのため，引数の指定に関するランタイムも追加を行った．

4.2 コンパイル時の流れ

OpenCL 実装を行うにあたり，XMP-dev/CUDA と同じソースコードを用いて CUDA 版，OpenCL 版の実行ファイルを生成することが必要不可欠となる．そのため XMP-dev の指示文はそのままに，コンパイル時にオプションを指定することで CUDA による並列化コードと OpenCL による並列化コードを生成する．XMP-dev/CUDA と XMP-dev/OpenCL のコンパイル時の流れは**図 4** のようになる．

図 4 において，点線で囲まれている部分が本研究で追加したコンパイラ実装である．

図の補足として，大まかな流れを以下に示す．

(1) MPI 並列版コードとデバイス並列版コードの生成
(2) MPI 並列版コードのコンパイル及びデバイスコードのコンパイル
(3) 1，2 で生成したファイルと XMP ランタイムライブラリのリンク
(4) （XMP-dev/OpenCL のみ）実行時コンパイルによるビルド

まず，逐次コードに指示文を追加した sample.c から，XMP-dev/CUDA では sample.i（MPI 並列版コード）と sample.cu（デバイス並列版コード）を生成する一方，XMP-dev/OpenCL では sample.i，sample.cpp（デバイス並列版コード），sample.cl（デバイス用カーネルコード）を生成する．MPI 並列版コード sample.i では共にデバイス並列コードのラッパー関数を指定し，XMP-dev ランタイム API を呼ぶ．カーネル用コンパイラによるコンパイルは，XMP-dev/CUDA では NVIDIA の GPU 専用のコンパイラを用いるが，XMP-dev/OpenCL では C++コンパイラを用いている．これは OpenCL は実行時にカーネルコードをビルドするため，静的にデバイスコードをコンパイルする必要がないためである．そして XMP，XMP-dev に関するランタイムをリンクし，実行ファイルを生成する．最後に，XMP-dev/OpenCL では実行時コンパイルによって動的にカーネルコードがビルドされ，プログラムが実行される．

5. 性能評価

本稿では重力計算を行う N 体問題，及び行列積を用いて，XMP-dev/OpenCL コンパイラの倍精度浮動小数点演算性能について評価を行った．性能評価のために，NVIDIA の GPU を搭載した最大で 4 ノードのクラスタ，及び AMD の GPU を搭載した 1 ノードのマシンを用いた．本来 AMD の GPU を搭載した GPU クラスタを評価に用いるのが望ましいが，環境がないため本稿では単体ノードのみ測定を行った．測定に用いた環境を**表 2** に示す．また，N 体問題には 1 次元配列を，行列積では 2 次元配列を用いて計算を行わせた．XMP-dev において，1 次元配列と 2 次元配列ではカーネル内のローカルなインデックス変換に使用する関数の数に違いがあるため，多次元配列を用いた場合の検証が必要であると考えられた．そこで，N 体問題の他に，よく知られるベンチマークとして行列積を用いることとした．なお，N 体問題の FLOPS を計算する際は，sqrt 関数を 1FLOP としてカウントしている．これは，NVIDIA と AMD で OpenCL の実装方法が違うため，共に 1FLOP とし，公平性を保つためである．

まず事前評価として，CUDA と NVIDIA 実装による OpenCL のカーネル起動時のコストを比較した．CUDA，OpenCL 共に 1024*1024 スレッド生成するだけの空のカーネルを起動し，1000 回の平均時間を測定した結果を**図 3** に示す．図 3 より，何もしない空のカーネルを繰り返し起動する場合，OpenCL の方が若干起動時間が速いことがわかる．しかし，その差は高々30 micro sec であり，今回用いた N 体問題と行列積のベンチマークにお

表 2 評価環境

	GPU クラスタ	単体マシン
CPU	AMD Opteron Processor 6134*(8core*2) 2.3GHz	Intel Core2Duo E8400 3.00GHz
Memory	DDR3-1333 2GB*2	DDR2-800 2GB*4
GPU	NVIDIA Tesla C2050 (GDDR 3GB)	AMD Radeon HD6970 (GDDR5 2GB)
Network	InfiniBand QDR	N/A
OS	CentOS 6.0	CentOS 6.0
MPI	OpenMPI 1.4.3	N/A
CPU Compiler, option	GCC 4.4.4, -O3	GCC 4.4.4, -O3
Device Compiler	NVIDIA CUDA Toolkit ver4.0 NVCC4.0,OpenCL 1.0	AMD-APP-SDK-v2.5 OpenCL 1.1

表 3 Kernel launch time

	avg [sec]	standard deviation
CUDA	0.000178	6.03e-07
OpenCL	0.000147	1.45e-05

いては，計算時間に対して圧倒的に短いため，カーネルの起動時間が XMP-dev/CUDA と XMP-dev/OpenCL の性能比較の際に明確な差として現れることはないと考えられる．また，OpenCL において標準偏差の値が大きいこともわかる．毎回の起動時間にばらつきが見られたため，以下の測定結果においてもプログラムを繰り返し実行し平均の値としている．

5.1 N 体問題

まず NVIDIA の GPU クラスタ 1node，2node，4node の 3 つの条件で XMP-dev/CUDA と XMP-dev/OpenCL について評価を行う．問題サイズは N 体問題における天体の数であり，16k（k=1024）と 128k の測定結果のグラフを**図 5** に示す．図 5 から，XMP-dev/CUDA，XMP-dev/OpenCL 共にノード数に従った性能向上が見られる．特に，128k の場合は 1node に対して 4node 実行はほぼ 4 倍の性能が得られた．また，XMP-dev/CUDA と XMP-dev/OpenCL の性能に差がないことがわかる．これは実行するデバイスが共に NVIDIA の GPU であることと，そして今回の N 体問題のプログラムでは CUDA と OpenCL においてデバイスコードの最適化の度合いに優劣が無いためであると考えられる．また，問題サイズが小さい場合に 1node に対する 4node の性能が低いことがわかる．この原因としては，XMP-dev の変換コードに問題があるわけではなく，XMP-dev は内部で MPI 通信を行うため，単にノード間の通信コストの問題であると考えられる．

次にプログラミングコストの観点で議論するために，XMP-dev/OpenCL で並列化した

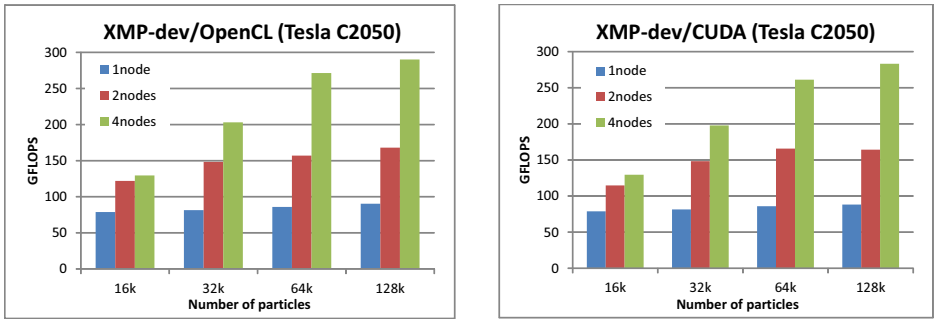


図 5 N-body XMP-dev/CUDA vs XMP-dev/OpenCL

コードと，MPI と OpenCL を用いて記述した自作並列コード（グラフでは hand-coding と記す）の性能比較を行った．**図 6** に NVIDIAGPU クラスタ 4node において，問題サイズを 16k，32k，64k，128k とした場合の測定結果を示す．図 6 から，自動並列化したコードは自作並列コードに比べて極端な性能の低下があるわけではないが，XMP-dev/OpenCL の性能が自作並列コードに比べてやや低下していることがわかる．この性能低下の原因として，ノード間通信を行う際に連続したメモリ領域をまとめる処理や，グローバルなインデックスからローカルなインデックスへ変換する際のオーバヘッドが考えられる．また，XMP-dev/OpenCL は N 体問題を解く逐次コード 121 行に対し，22 行の指示文を追加することで並列化を行っている．それに対して自作並列コードには全体で 278 行必要となる．N 体問題よりもさらに複雑な計算においては，MPI と OpenCL の知識がより求められ，さらにプログラミングコストが増大してしまう．そのため，XMP-dev/OpenCL は逐次コー

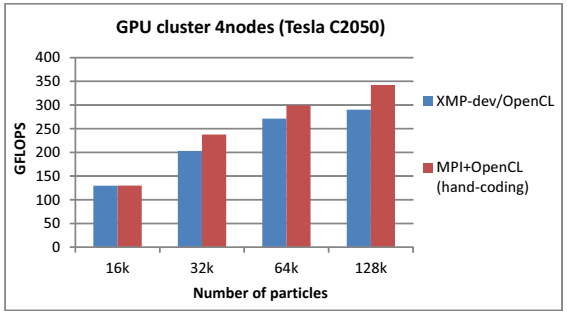


図 6 N-body XMP-dev/OpenCL vs MPI+CUDA(Hand coding)

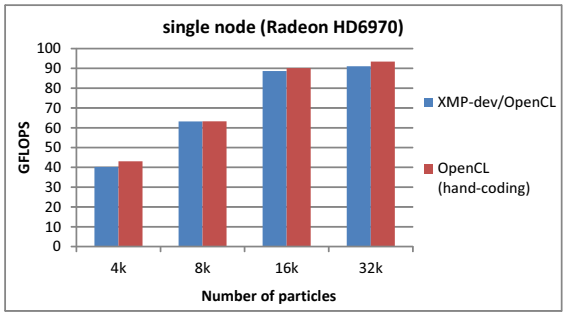
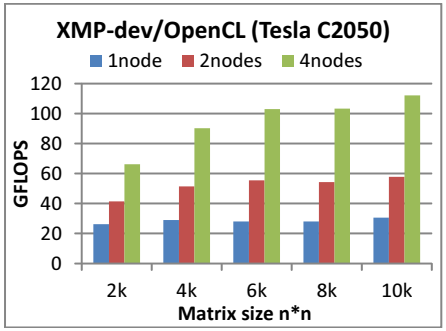


図 7 N-body AMD GPU

ドからの少ない変更で並列化を可能としている点で有益であると考えられる．

最後に，ソースコードの可搬性について検証するために，AMD の GPU 上でも測定を行った．問題サイズ 4k，8k，16k，32k において測定した結果を**図 7** に示す．測定には 1 台のマシンを用いており，XMP-dev/OpenCL を用いて指示文を挟んで書いたプログラムと，OpenCL で書いたプログラムの比較を行っている．図 7 に見られる，XMP-dev/OpenCL の性能が OpenCL で書いたコードに対しやや低下している原因としては，先に述べたインデックス変換コスト等が考えられる．なお，この測定には NVIDIA の GPU 上で実行したコードと同じソースコードを用いている．XMP-dev/OpenCL コンパイラにより同じソースコードを用いて様々なデバイス上で動作するプログラムを実行できることを確認した．この結果から，XMP-dev/OpenCL で可搬性のあるソースコードを生成できることが示された．



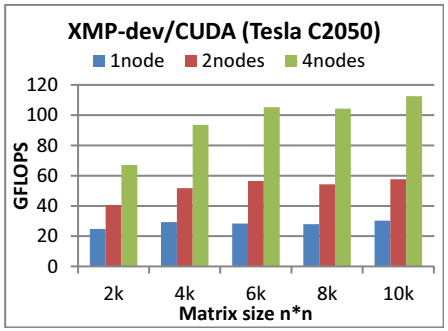


図 8 Matrix Multiplication XMP-dev/CUDA vs XMP-dev/OpenCL

5.2 行 列 積

ベンチマークとして N 体問題の他に行列積を用いて XMP-dev/CUDA と XMP-dev/OpenCL の性能評価を行った．まず NVIDIA の GPU クラスタにおいて，1node,, 2node,, 4node と台数を増やした場合の測定結果を**図 8** に示す．問題サイズは行列のサイズであり，2k，4k，6k，8k，10k とした場合において測定を行った．N 体問題の測定結果と同じく，XMP-dev/CUDA と XMP-dev/OpenCL の比較においてはほぼ同等の性能，及びノード数を増やした場合の性能のスケールが見られる．また，カーネル内で 2 次元配列のインデックス変換を行った場合でも，正しく結果が得られることが示された．なお本稿では N 体問題と行列積でのみベンチマークを行ったため，XMP-dev/CUDA と XMP-dev/OpenCL で同等の性能が得られているが，より複雑なコードを書く必要のある実アプリケーションにおいてはまた違った結果が得られると考えられる．その場合，ベンダーがより力を入れて内部実装を行なっている CUDA を用いた場合，つまり XMP-dev/CUDA を用いてコードを記述した場合の方がより高い性能を得ることができると考えられる．

次に，AMD の GPU を用いて XMP-dev/OpenCL の測定を行った．問題サイズを 1k，2k，3k，4k，5k とした場合の測定結果を**図 9** に示す．行列積においても，XMP-dev/OpenCL の指示文で並列化を行った測定結果と OpenCL のみを用いて記述したコードの結果を比較した．N 体問題と同じく OpenCL のみを用いて記述したコードで良い性能が得られている．これに対しても，先に述べたインデックス変換コスト等が原因として考えられる．また，性能の可搬性の観点から，NVIDIA の GPU と AMD の GPU で同じソースコードを用いた場合にどれ程性能の差異が見られるかという問題がある．本稿の測定結果では，同じソー

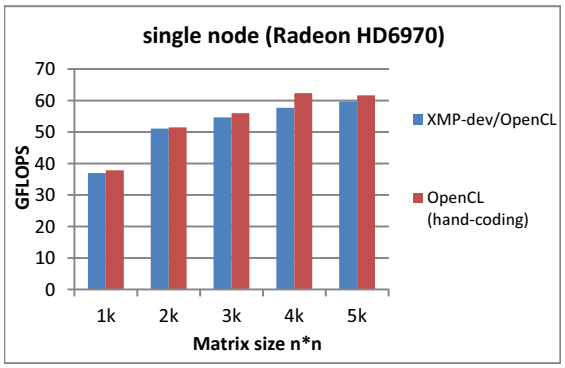


図 9 Matrix Multiplication on AMD GPU

スコードを用いた場合では AMD の GPU（RadeonHD6970）の方が TeslaC2050 よりも良い性能が得られていた．しかし，これら二つの GPU では内部構造が異なるため，別々な GPU においてそれぞれ良い性能を出すためには，アーキテクチャに則した最適化を施す必要がある．したがって，XMP-dev/OpenCL においても，最終的には性能の可搬性までも考慮した実装となることが望ましいと考えられる．

6. 関 連 研 究

OpenCL を用いた GPU のベンチマークは盛んに行われている．Du ら[8] は OpenCL で記述されたソースコードの性能可搬性を，NVIDIA と AMD の両方の GPU を用いて評価した．その結果 OpenCL は両方の GPU でそれぞれに最適化したカーネルを記述することで理論ピーク値の 50%以上の性能を出すことが可能であるが，同じカーネルの性能可搬性は無いことを明らかにした．また，GPU 以外にも，Cell/B.E. のベンチマークも OpenCL を用いて行われている．Breitbart ら[9] は Cell/B.E. の性能評価を行い，OpenCL は Cell/B.E. においても効果的なプログラミングモデルであることを明らかにした．

本研究の他にも，いくつか指示文ベースの GPU 向けプログラミングモデルが開発されている．OpenMP の GPGPU 拡張として OpenMPC[10] と OMPCUDA[11] が挙げられ，これらは逐次コードから OpenMP の拡張指示文によって CUDA コードを生成する．また，PGI 社の GPGPU 向けコンパイラとして PGI Accelelator コンパイラ[12] が挙げられる．PGI Accelelaor コンパイラは静的なプログラム解析によって CUDA カーネルの自動最適化も可能としている．さらに，アクセラレータ向け並列プログラミングモデルとして OpenACC[13] の仕様が発表された．OpenACC は OpenCL と同じくプラットフォームに依存しない，多種なアクセラレータ向けの開発環境である．アクセラレータを用いたプログラミングにおける典型的な処理を指示文ベースで簡単に記述可能で，デバイスコードの最適化まで自動で行う仕様となっている．OpenACC は NVIDIA，PGI，CAPS，Cray らによって仕様策定，開発が行われており，実際に動作するコンパイラも近日発表されると考えられる．

7. 結論と今後の課題

本研究では XMP-dev の OpenCL 実装を行った．2 種類のベンチマークの測定結果から，XMP-dev/OpenCL は XMP-dev/CUDA と同等の性能を得ることができることを示した．今回は 2 種類の単純なベンチマークを用いたが，XMP-dev を実アプリケーションへ適用した場合，XMP-dev/CUDA と XMP-dev/OpenCL において性能の差異が見られることも十分考えられる．また，プログラミングコストの観点からも XMP-dev の有用性を検証することができた．さらに，XMP-dev/OpenCL において，OpenCL の特徴であるデバイスに依存しないソースコードの可搬性も確認した．今回は 2 つの異なる GPU 上でのみ測定を行ったが，多種多様なアクセラレータデバイス上でも正常に動くことが期待できる．今後の課題として，実装の点ではデバイスコードの自動最適化，インデックス変換のオーバーヘッド削減などが挙げられる．また，性能の可搬性という点で，異なるデバイス間でも等しく性能を出すことができるといったアプローチも必要になると考えられる．

謝辞 本研究の一部は，戦略的国際科学技術協力推進事業（日仏共同研究）「ポストペタスケールコンピューティングのためのフレームワークとプログラミング」による．

参 考 文 献

1) : OpenCL-manual, `http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/`.
2) : PGAS - Partitioned Global Address Space Language. `http://www.pgas.org/`.
3) : Specification of XcalableMP, version 0.7a. `http://www.xcalablemp.org/publications.html`.
4) Lee, J., Tran, M.T., Odajima, T., Boku, T. and Sato, M.: An Extension of XcalableMP PGAS Language for Multi-node GPU Clusters, *Proc. Ninth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heteroge-*

neous Platforms (HeteroPar'2011) (2011).
5) : NVIDIA CUDA. http://developer.nvidia.com/category/zone/cuda-zone.
6) Lee, J. and Sato, M.: Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems, *Proc. the 2010 39th International Conference on Parallel Processing Workshops (ICPPW'10)*, pp. 413–420 (2010).
7) : OpenMP.org. http://openmp.org/wp/.
8) Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G. and Dongarra, J.: From CUDA to OpenCL : Towards a Performance-portable Solution for Multi-platform GPU Programming, *Parallel Computing*, No. UT-CS-10-656 (online), available from ⟨http://web.eecs.utk.edu/~library/2010.html⟩ (2010).
9) Breitbart, J. and Fohry, C.: OpenCL - An effective programming model for data parallel computations at the Cell Broadband Engine, *Proc. Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)* (2010).
10) Lee, S. and Eigenmann, R.: OpenMPC: Extended OpenMP Programming and Tuning for GPUs, *Proc. International Conference for High Performance Computing Networking, Storage and Analysis (SC'10)* (2010).
11) Satoshi, O., Shoichi, H. and Hiroki, H.: OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler, *Proc. 6th International Workshop on OpenMP (IWOMP'2010), Lecture Notes in Computer Science, No.6132*, Springer-Verlag, pp.161–173 (2010).
12) : PGI Accelerator Compilers. http://www.pgroup.com/resources/accel.htm.
13) : OpenACC, http://www.openacc-standard.org/.