Vol.2011-HPC-129 No.12
2011/3/15

並列言語 XcalableMP の GPU 向け拡張

小田嶋 哲哉[†1]　チャン トゥァン ミン[†2]
李　珍泌[†2]
朴　泰祐[†2]　佐藤三久[†2]

高い演算性能及びメモリバンド幅をもつ GPU を搭載した GPU クラスタが高性能計算プラットホームとして広く利用されている．GPU クラスタではプログラミングが非常に複雑になることや，計算負荷が GPU または CPU のどちらかに偏り，計算リソース全体を有効利用しにくいという問題がある．そこで，分散メモリシステム向けの並列言語である XcalableMP を GPU 向けに拡張して，GPU クラスタ等のヘテロジニアス環境に適応させることを検討する．本稿ではその予備評価として，XcalableMP による GPU/CPU 協調計算を行い，典型的な HPC アプリケーションである N 体問題と行列積計算を対象に，GPU と CPU への計算負荷分散による最適化を行い，これらによる協調計算の可能性を検討した．その結果，2 ノード・2GPU のシステム上で GPU に割り当てるデータを 50%から 60%にしたところ，最大で約 1.7 倍の高速化を得ることができた．

Extend to GPU for XcalableMP: A Parallel Programming Language

Tetsuya Odajima,[†1] Minh Tuan Tran,[†2]
Jinpil Lee,[†2] Taisuke Boku[†2] and Mitsuhisa Sato[†2]

As shown in TOP500 List at November 2010, GPU clusters have been recognized as highly cost-effective HPC resources. However, the programming on GPU cluster requires much harder effort than ordinary PC clusters because of complicated heterogeneous coding with combination of CUDA/OpenCL, OpenMP and MPI, for example.

In order to provide a solution for this, we will consider an extension of parallel programming language XcalableMP for GPU cluster computing. In this paper, we propose an textended notation of XcalableMP for data and process distribution in a GPU cluster. We also preliminarily evaluate the performance enhancement by a cooperated computing with GPU and multi-core CPU on typical HPC applications, N-body calculation and matrix multiplication. As a result, we confirmed the maximum of 1.7 times higher performance when we distribute the 50 to 60% of computation to GPU, compared with the case with 100% of computation only by GPU.

1. はじめに

近年，高い演算性能及びメモリバンド幅をもつ GPU（Graphics Processing Unit）を画像処理以外の汎用計算に用いる GPGPU（General-Purpose computation on GPU）が注目されている．特に，NVIDIA 社が提供するプログラミング環境 CUDA[1)]（Compute Unified Device Architecture）によって CPU で行うプログラミングに近い形でのプログラミングが可能になったことで，High Performance Computing（HPC）の様々なアプリケーション分野で GPGPU への対応が進んでいる．これに伴い，GPU を搭載した PC クラスタ（以下「GPU クラスタ」と略す）が数多く出現し，広く利用されるようになった．しかし，現在の PC クラスタはすでに MPI や OpenMP を組み合わせた複雑なプログラミングが必要であり，GPU クラスタでは CUDA などによる GPU プログラミングが加わることで，プログラミングコストの増加が問題になっている．また GPU クラスタでは，GPU を一種の非常に高速な計算加速装置とみなして，CPU から計算するデータを送り，計算が終わったらデータを受け取るという機能分散的なプログラミング手法が一般的である．しかし，これでは CPU の計算リソースを GPU と並行して有効に使用することができない．また，CPU のコア数は年々増加しており，さらに 256bit AVX[2)] 命令等の登場により，GPU と CPU の潜在的な演算能力は徐々に近づきつつあると言える．

そこで，分散メモリシステムを対象とした，指示文ベースで並列プログラミングを行うためのプログラミング言語である XcalableMP[3)]（以下「XMP」と略す）を GPU 向けに拡張することを考える．XMP は PC クラスタを始めとする分散メモリ環境を対象とした並列プログラミング言語であり，データ配列のノード間自動分散やデータ整合性のための通信の自動生成機能を持つ．GPU クラスタにおいては GPU と CPU のアドレス空間は異なり，一種の分散メモリとなっているため，XMP の拡張機能として GPU を扱うことにより，プ

†1 筑波大学情報学群情報科学類
College of Information Science, University of Tsukuba
†2 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

ログラマの負担を大幅に低減することができる．これにより，GPU と CPU へのデータの分散及び負荷分散を行うことで，計算リソースを最大限活用できるプログラミングの支援を行う．

本研究では，XMP による GPU と CPU によるプログラミングのモデルを検討し，XMP の仕様拡張として XMP/GPU を提案する．また，XMP/GPU のコンパイラが生成するプログラムを想定し，XMP，OpenMP，CUDA によるマルチノード上での GPU/CPU 協調計算を行い，XMP/GPU の有用性を評価する．

2. XcalableMP

XMP の詳細に関しては文献[3] に詳しいが，ここでは本稿を理解するための最低限の XMP の機能について述べる．

2.1 実行モデル

XMP の実行モデルは Single Program Multiple Data（SPMD）である．そして，XMP では実行単位のプロセスを「ノード」と表現する．ソースコード上で指示文での指定がない部分は各ノードで同じ処理を行い，XMP によって分割宣言されていないデータは，各ノードで重複したものを保持する．

XMP におけるデータ参照では，明示的な通信指示文がない限り，ローカルメモリにあるデータへの参照がそのまま行われる．他のノードにあるデータへの参照は XMP の指示文が提供する通信記法を用いて明示的に行う必要がある．

2.2 プログラミングモデル

XMP には「グローバルビューモデル」と「ローカルビューモデル」の 2 種類のメモリモデルがある．グローバルビューモデルは，分散メモリシステムで OpenMP-like な指示文によって，並列プログラミングを行うものである．XMP の指示文はデータの分散だけではなく，ループ文のワークシェアリング，barrier や reduction 処理などの集団通信のような並列化手法も提供する．

ローカルビューモデルは，XMP の補助関数の助けを借りてプログラマがメモリのインデックス計算を行うので，より明瞭なメモリイメージを持つことができる．これは，外部のライブラリなどを組み込むときに便利である．XMP ではローカルビューモデルでの通信を容易にするために，片方向通信の構文を提供している．

2.3 XcalableMP による並列プログラミング

2.3.1 template による index 空間の分割

グローバルビューモデルにおける並列プログラミングは，逐次プログラムに適宜指示文を挿入することで実現する．**図 1** にソースコードの例と template にデータ分散の例を示す．

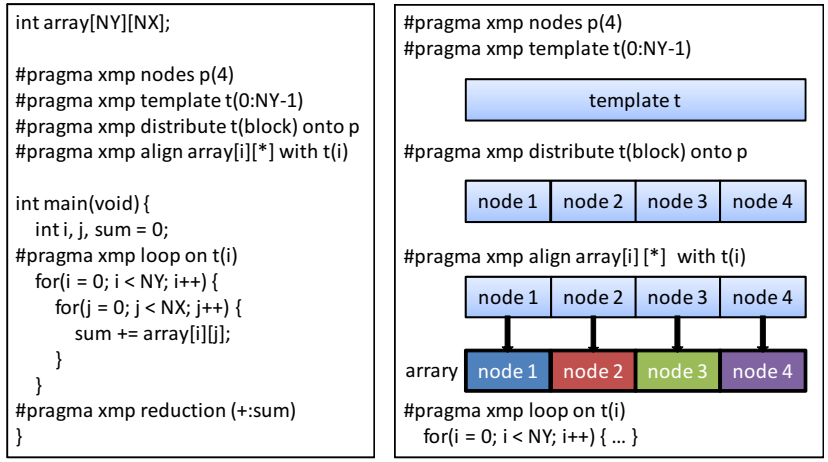
nodes 指示文はプログラムを並列に実行するノードの集合の宣言であり，**図 1(a)** では，4 ノードの集合を p という名前で宣言している．template 指示文は index の集合を表現する仮想的な配列である．ここで言う index は，配列やループ文の添字である．template の宣言からデータの分散までのイメージを**図 1(b)** に示す．XMP ではこの template を用いて「どのデータがどの配列で処理されるべきか」というマッピングの記述を行い，配列の分散や，ループ文のワークシェアリングを行う．distribute 指示文により，template によって宣言された index 空間を，各ノードに分散配置する．現在の仕様では，ブロック分割（block）とサイクリック分割（cyclic）があり，各ノードに不均等に分散する gblock という方法も提案されている．そして，align 指示文で配列の分散を宣言し，index 空間と実際の配列を一致させる．図 1(a) のプログラムの例では，二次元配列が y 軸方向にブロック分割され，分割された配列は各ノードで割り当てられた部分だけメモリに確保される．

2.4 loop 文によるワークシェアリング

loop 指示文は，続くループ文（C 言語では for 文，Fortran 言語では DO 文）が全ノードで並列処理されることを示す．loop 指示文直後のループ文は各ノードでワークシェアリングされる．ワークシェアリングされる配列の分割は template とノード集合を組み合わせることで決定される．図 1 のプログラム例では，添字 i のループ文が全ノードで並列処理される．実際には，インデックス i のうち，そのノードが所有する範囲を求め，ノード上ではループ内でローカル配列のみにアクセスする．

2.5 分散された配列のコピー

分散を宣言した配列の一部またはすべてをコピーしたい時，そのコピーしたい領域の配列がローカルに存在するとは限らない．そのようなときは，ノード間通信により配列を集める必要がある．そこで，XMP では gmove 指示文を用いることで，直後の代入文においてノード間通信の必要があることをコンパイラに知らせる．**図 2** の場合，各ノードに存在する L1，L2 という配列に分散した配列の一部をコピーする．L1 のコピーについて考えると，node3 はローカル配列から代入すれば良いが，他のノードでは通信が発生する．この通信は XMP によって自動的に生成され，プログラマはインデックスの計算などをする必要はない．L2 は複数ノードにまたがった配列を代入する例である．これもまた，XMP が自動的



(a) XMP のプログラム例　　(b) XMP の template

図 1　XMP による記述例と template によるデータ分散

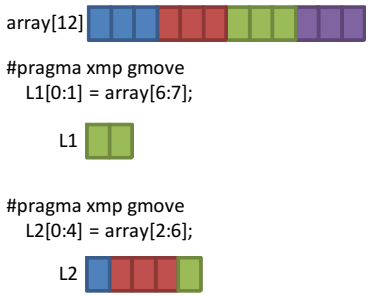


図 2　gmove による配列のコピー

に配列要素を集める通信を生成し，コピーが可能になる．

3. XcalableMP による GPU コンピューティング

ここでは，XMP を GPU クラスタで利用出来るように仕様拡張をした XMP/GPU の提案をし，それによって想定されるプログラムの例を示していく．

3.1 XMP/GPU の基本概念

XMP は分散メモリシステムを対象としたノード間通信のサポートを行う．GPU クラスタではアドレス空間の異なる GPU を一種のノードとみなすことで，GPU と CPU が混在した環境でのデータの分散及びワークシェアリングを XMP に吸収可能であると考える．これに基づき，逐次のプログラムに指示文を挿入するだけで GPU/CPU 協調計算を行うように XMP の GPU 向け拡張を行う．これは XMP に対する言語仕様の拡張であるため，以後，このプログラミング環境を XMP/GPU と呼ぶ．

なお，以下の説明では一般的な GPU クラスタにおける計算ノードを単に「ノード」と呼ぶ．これは，XMP におけるノードの定義とは異なる．XMP では，全ての並列計算リソース（= CPU）が分散メモリアーキテクチャを持つと想定されているため,「ノード=CPU」として定義されていた．これに対し，XMP/GPU では，1 つの計算ノードが 1 つ以上のコアを持つマルチコア CPU と，1 つ以上の GPU から構成される複合アーキテクチャを持つと定義する．これは現在の一般的な GPU クラスタの構成に一致する．よって「ノード≠CPU」であり，従来の XMP とは異なる．

XMP/GPU に求められる機能は以下の 3 つである．

(1) GPU を計算ノード上の新たなリソースとし，各ノードの GPU と CPU へのデータ配列のマッピングを記述．
(2) GPU 及び CPU に割り当てられたデータに align された演算及びループ分割機能．
(3) 計算ノード間のデータ通信の際，GPU 上のメモリを CPU 側のメモリにコピーし，ノード間通信機能を使ってデータ交換する機能．

XMP/GPU は，従来の XMP と同様に template を用いたデータや処理の分散を行う．このため，template 指示文を拡張し，処理を GPU と CPU に分散することを考える．まず，gpunodes 指示文でノード内に存在する GPU の数を指定する．ここではマルチ GPU 環境も想定し，複数の GPU を計算ノード上で宣言可能とする．そして，template に GPU の処理の分散を指定するのが gpudata 指示文である．指定された割合の分だけ GPU が各ノード上に分散された配列の部分的処理を行う．既存の XMP と同様に align 指示文で配列に template を適用することで loop 指示文による処理において，GPU と CPU が並列に動作することを記述可能とする．

図 3 に，要素数 1000 の配列の各要素を 2 乗する計算を，gmove 指示文によるデータ転送と GPU/CPU 協調計算として XMP/GPU で記述した例を示す．6 行目の dist 変数は GPU へのデータの割り当てる割合を指定する．このプログラムでは GPU に 60%，CPU に

```
1  #define N 1000
2
3  int data[N];
4  int a[N], b[N];
5
6  double dist[1] = {0.6};
7  #pragma xmp nodes p(*)
8  #pragma xmp gpunodes g(1) on p
9  //中略
10 #pragma xmp gpudata t1(dist)
11 #pragma xmp align [i] with t0(i) :: data
12 #pragma xmp align [i] with t1(i) :: a, b
13
14 int main(void)
15 {
16   int i;
17
18 #pragma xmp loop on t0(i)
19   for (i = 0; i < N; i++)
20     data[i] = i;
21
22 #pragma xmp gmove
23   a[:] = data[:];
24
25 #pragma xmp loop on t1(i)
26   for (i = 0; i < N; i++)
27     b[i] = a[i] * a[i];
28
29 #pragma xmp gmove
30   data[:] = b[:];
31
32   return 0;
33 }
```

図 3 XMP/GPU のプログラム例

40%のデータを割り当て，計算を実行する．そして，8 行目では gpunodes によってノードに搭載された GPU の個数を与える．10 行目の gpudata では，template t1 によってノードにブロック分割された配列について，dist 変数によって指定された割合で処理をさせる，というデータ分散を各ノードに適用する．

18 行目から 20 行目は，template t0 によって分割された data 配列の初期化を行う．この状態では，data 配列は CPU のメモリ上にしか存在しない．そこで，GPU へのデータ転送を行うために，XMP/GPU では 22，23 行目の gmove 指示文を用いる．図 4 に GPU への割り当てを含めた template の宣言から gmove 指示文の動きを示す．template t1 によって data 配列から a 配列へのコピーが起こり，CPU から CPU へは memcpy が実行され，CPU から GPU へはメモリ転送が実行される．25 から 27 行目までは gmove 指示文によってコピーされた配列 a で CPU と GPU による計算が非同期的に実行される．29，30 行目は，逆方向の通信が実行される．

3.2 現在の XcalableMP における GPU プログラミング

XMP/GPU のコンパイラはまだ未実装であるが，その記述能力と，想定される GPU と CPU による協調計算の有効性の確認が必要である．そこで，XMP/GPU の機能を模擬的に実行するために，既存の XMP 上で CUDA 及び OpenMP による明示的なプログラミングを行う．また，各ノードの処理はシングルスレッドで実行されるため，GPU の制御を CPU

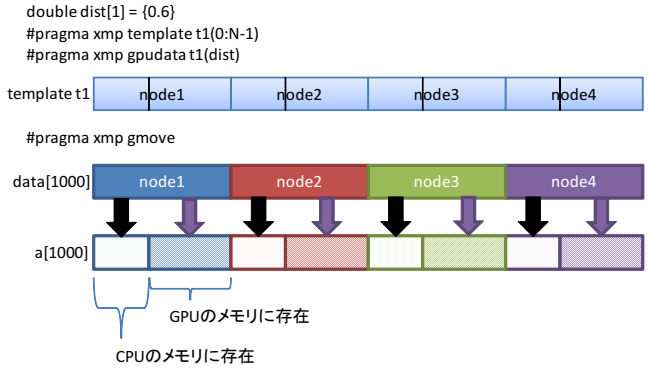


図 4 XMP/GPU の template と gmove 指示分

での計算と並行して行うプログラムを簡潔に記述することができない．そこで，ノードでの計算部分を OpenMP で並列化し，GPU 上でのスレッドによる並列計算を実行するのと同時に，CUDA による GPU 処理の起動も一部のスレッドを用いて実行する．これらを組み合わせることにより，XMP 上での仮想 XMP/GPU 実行をシュミレートし，今後の検討材料とする．

本手法では，各ノード上で自分の受け持つ配列の初期化を行うとともに，GPU のメモリへ必要な領域を確保する．そして，XMP を用いたノード間の通信により配列データの分散・同期を行う．**図 5** に今後の処理の概要を示す．まず，各ノード上で OpenMP によるスレッド生成を行う．本手法では，0 番スレッドが GPU の管理を担い，CPU-GPU 間のデータ通信や，GPU のカーネル関数の呼び出しなど実行する．スレッド生成後，0 番スレッドは GPU での計算に必要なデータを cudaMemcpy 関数によって GPU へ転送し，そして GPU 上での計算を行う関数を呼び出す．ここで，0 番スレッドは GPU の関数の終了を待たず，そのまま他のスレッドと共に継続する CPU の計算を実行するようになっており，GPU の計算を非同期的に実行することで 0 番スレッドを GPU お管理から一時開放し，CPU の計算リソースをすべて利用する．CPU での計算が終了したときに，GPU の処理関数の終了と同期を取り，GPU の計算が終了するまで待つ．GPU の処理が完了したら，0 番スレッドが GPU で計算したデータを CPU 側のメモリに転送する．転送終了後 OpenMP によるマルチスレッド処理を集約し，XMP によるノード間通信によって計算データの同期を取る．

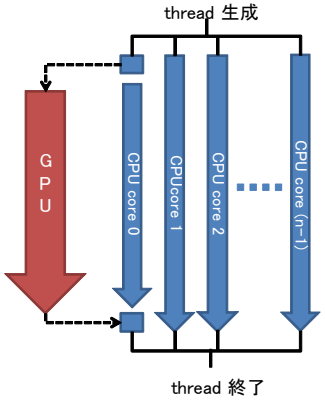


図 5 実行の概念

4. 性能評価

本研究で提案する XMP による GPU/CPU 協調計算の枠組みが正しく動作することを示し，このような協調計算による性能向上を確認するために，典型的な HPC アプリケーションである N 体問題と行列積計算において，問題サイズ，GPU と CPU へのデータ割り当てサイズを変化させたときに，GPU のみを使った場合と比較して速度向上が得られるかを評価する．以降，全体の処理量を GPU へ割り当てるという指標を「GPU 割り当て率」と呼ぶ．GPU のみを使うときは GPU 割り当て率 100%となる．

4.1 評価環境

表 1 に評価に用いる GPU クラスタのノード構成を示す．本評価では 2 ノードを用いたハイブリッドプログラミングの測定を行った．プログラムの分散メモリ部分は XMP，共有メモリ部分は OpenMP，GPU 部分は CUDA を用いる．XMP コンパイラにより生成される計算ノード間の通信は MPI により実行される．

4.2 N 体問題の評価

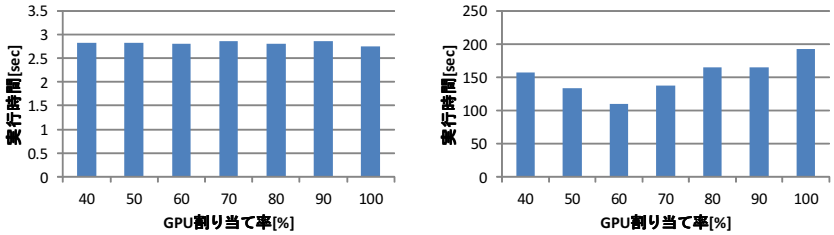
本評価では，N 点の質点間の重力相互作用に基づきニュートンの運動方程式の積分を行うプログラムを作成した．問題サイズである質点数を 10240 点と 102400 点の 2 種類とし，時間方向の反復回数を 20 回としたときの実行時間を測定する．**図 6** に測定したプログラムの時間を示す．N=10240 の場合 (**図 6(a)**)，GPU 割り当て率が 100%以外の時すべての場合

表 1 ノード構成

構成	2 ソケット（8CPU コア）
CPU	Intel Xeon W5590 3.33GHz（Nehalem-EP）
総ホスト・メモリ容量	12GB DDR3（ノード内共有）
GPU	NVIDIA Tesla C1060 1.3GHz
デバイス・メモリ容量	4GB GDDR3
ノード間ネットワーク	InfiniBand (4X QDR)
OS	Linux version 2.6.27.41-170.2.117.fc10.x86_64
CPU コンパイラ	gcc 4.3.2 (-O3)
GPU コンパイラ	nvcc 3.0 (-O3 -arch=sm_13)
MPI	Open MPI 1.4.1

において，GPU 割り当て率 100%の場合よりも速度が低下していることがわかる．これは，十分な問題サイズがなかったため，GPU の計算リソースを使い切ることが出来ず，協調計算を行うときに発生する CPU と GPU の同期などによるオーバーヘッドが速度低下の原因として考えられる．GPU のみを計算に使用すれば CPU との同期が必要ないため，問題サイズが小さい場合には協調計算によるメリットを享受することが出来ない．つまり，協調計算ではオーバーヘッドを無視できるほどの問題サイズが必要になる．これに対し，N=102400 (**図 6(b)**) では，どの GPU 割り当て率においても速度の向上が見られた．GPU 割り当て率 100%の場合との差異をより明らかにするため，GPU 割り当て率 100%の場合の性能を 1 とした速度向上率を算出した．結果を**図 7** に示す．N=102400 の場合，GPU 割り当て率 60%の場合が最高性能で，GPU のみを使った場合に比べて 1.75 倍の速度向上が得られた．

続いて，GPU/CPU 協調計算における GPU と CPU のそれぞれの利用状況を調べるため，GPU と CPU の計算時間の関係を測定した．GPU へのデータ転送及び GPU の関数呼び出し時間は，すべてにおいて全体の実行時間の 0.001%未満であったため，これらのオーバーヘッドは無視できると考えられる．よって，GPU と CPU の計算はほぼ同時に開始しているものとする．**図 8** に GPU の管理をしている 0 番スレッドの CPU の計算時間とノード内の GPU の計算時間を示す．なお，時間測定は CPU 側でしか出来ないため，CPU の演算が GPU よりも先に終了した場合は GPU の計算終了待ち時間の測定が可能であるが，逆の場合はこれが出来ない．そのため，CPU での計算が先に終了するケースについては，GPU での計算に相当する部分を GPU のみで実行させて時間を測定した．ここに示したのは，最も速度向上が得られた GPU 割り当て率 60%を中心とした，50%から 70%における計算時間である．GPU 割り当て率が 60%のときは，GPU と CPU の計算時間ほぼ等しく，負荷分散が最適に行われていることがわかる．ここで，GPU 割り当て率を 50〜70%に増加



(a) 質点数 10240　　(b) 質点数 102400

図 6　N 体問題の実行時間

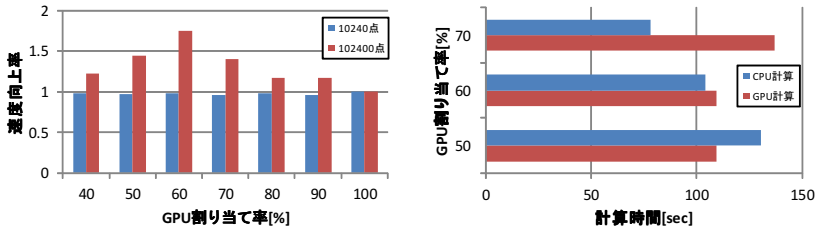
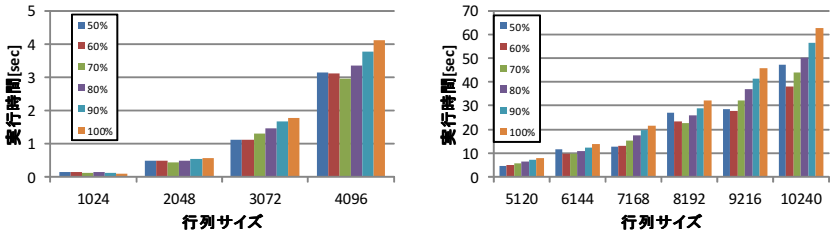


図 7　N 体問題：速度向上率　　**図 8**　質点数 102400 時の計算時間

させた場合，CPU の計算時間は処理量の減少に応じて単調減少しているのに対して，GPU 側は GPU 割り当て率 60%以下では計算時間の短縮が下げ止まっていることがわかる．これは，ある粒度以下の演算処理において GPU 内での処理効率が低下し，計算時間が一定時間以下にはならないことを示している．この点から，N=10240 の場合も同様に計算粒度が不十分であったということが推察される．

4.3 行　列　積

本評価は，サイズ N×N の正方行列 A，B，C について行列積 $C = A \cdot B$ の計算を行う．行列の分割方法は行列 A，C については y 軸方向にブロック分割した部分を各ノードが保持し，行列 B についてはすべてのノードが行列全体を持つものとする．測定する区間は，GPU へのデータ転送後から，すべてのノードでの計算が終了するまでである．**図 9** に，N を 1024〜10240 まで 1024 刻みで変化させて測定した時間を示す (演算量のオーダーが $O(N^3)$ であるため，N=4096 以下の場合と N=5120 以上の場合に分けてある)．行列サイズ



(a) 行列サイズ 1024 から 4096　　(b) 行列サイズ 5120 から 10240

図 9　行列積の実行時間

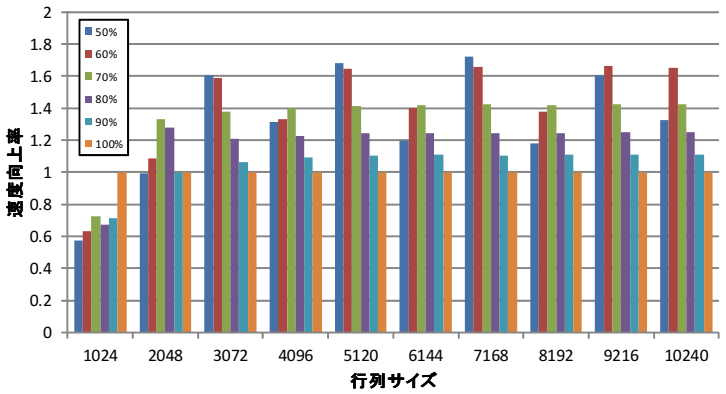


図 10　行列積：速度向上率

が 1024 の時以外ではほとんどの GPU 割り当て率において GPU のみを用いた場合よりも速度が向上していることがわかる．速度向上が得られなかった部分は，N 体問題の N=10240 と同様，問題サイズが小さすぎて GPU 上での演算量が少なく，十分な性能が得られなかったことが原因と考えられる．

図 10 に GPU 割り当て率 100%を 1 とした時の速度向上率を示す．行列サイズ 2048 以上ではすべての GPU 割り当て率において協調計算により速度向上が見られる．しかし，図 10 からは 2 つの特徴的な傾向が見られる．

(1)　行列サイズ 2048，4069，6144，8192 では速度向上率が小さい

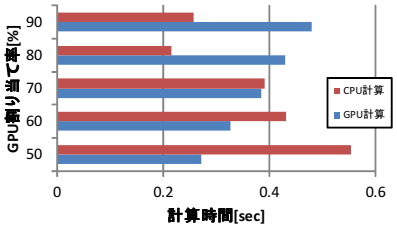


図 11 行列サイズ 2048 時の計算時間

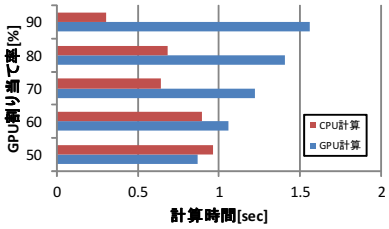


図 12 行列サイズ 3072 時の計算時間

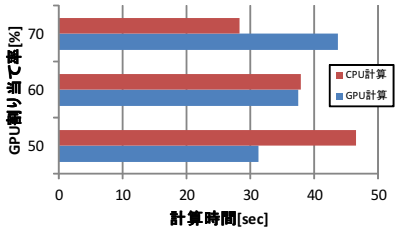


図 13 行列サイズ 10240 時の計算時間

(2) 行列サイズ 9216 以上と未満では最大の速度向上を達成する GPU 割り当て率が異なる

1 つ目の特徴的な傾向について，速度向上率が小さかった行列サイズ 2048 と十分な速度向上が得られた行列サイズ 3092 の GPU と CPU の計算時間を調査する．**図 11** と**図 12** に 0 番スレッドの GPU と CPU の計算時間を示す．測定方法は N 体問題の場合と同様である．図 11 より，GPU 割り当て率 90%の場合の CPU の計算時間が計算量の多い GPU 割り当て率 80%のときの計算時間より長くなっている．この原因は，OpenMP におけるループのスレッド分割の不均衡によるものと思われる．行列積のプログラムにおいて，CPU 側の処理ではキャッシュヒット率を上げるためにタイリングを行っているが，CPU に割り当てられた行列サイズがタイリングの基本ブロックサイズの整数倍数でないとき，OpenMP 上でのループ分割が不均衡になり，演算量が少ないスレッドが出現してしまう．このスレッド間の負荷バランス問題により，CPU の計算リソースを使い切ることが出来ず速度低下が起きたと考えられる．

2 つ目の傾向について，問題サイズが最も大きい N=10240 の GPU と CPU の計算時間を調査する．**図 13** に 0 番スレッドの GPU と CPU の計算時間を示す．図 13 より，行列サイズが大きくなると GPU 割り当て率が小さい時に，CPU での計算時間が飛躍的に大きくなっている．一方 GPU は，問題サイズが大きくなったときの演算時間の増加率は小さい．これによって，GPU 割り当て率 50%では GPU の計算リソースを使い切ることが出来なかったと考えられる．これによって，行列サイズが大きくなると最も速度向上率が高かった GPU 割り当て率 50%から 60%に変化したと考えられる．これは，行列サイズがさらに大きくなるとより GPU 割り当て率が高くなると予想される．

4.4 考　　察

本研究に用いたベンチマークでは，GPU クラスタ上で GPU/CPU 協調計算によって GPU のみを計算に利用した場合と比較して，速度向上を得ることができた．これによって，XMP/GPU により生成されるプログラムによる GPU/CPU 協調計算の有用性を示すことができた．

本研究より，CPU による計算は問題サイズの 40%から 50%ほど，非常に割合の高い物になっていた．これにより，CPU の計算が速度向上に大きな影響を与えていることがわかり，GPU/CPU 協調計算では，CPU の演算が速度向上に大きく影響していたため，積極的に CPU を使用すること必要になると考えられる．

本研究における速度向上に対する主な理由として，測定に使用している GPU が関係していると考えられる．NVIDIA 社の Tesla C1060 は単精度浮動小数点数演算性能は 993GFlops と 1TFlops に近い性能があるが，倍精度浮動小数点演算は 79GFlops と，単精度の場合の 10%未満になっている．本ベンチマークでは倍精度浮動小数点演算をしているので GPU の計算能力が比較的低かったことが影響していると考えられる．NVIDIA の新しいアーキテクチャFermi を搭載した GPU では 515GFlops まで倍精度演算の性能が向上しており，協調計算時の GPU への割合は非常に高くなると予想することができる．

しかし，CPU に関しても次世代マルチコアプロセッサである Intel 社の Sandy Bridge[4] や，AMD 社の Interlagos[5] では，8 から 16 コアに増強されるだけでなく，256-bit SIMD 命令（AVX 命令）の導入によりさらなる高性能化が見込まれている．メモリバンド幅についても，浮動小数点演算の性能向上に比べて相対的には低いものの，現状からさらに着実に向上する予定である．さらには，Intel の Nights Ferry[6] 計画等，メニーコアプロセッサへの移行も計画されており，CPU のソケット当たり性能も着実に向上していくため，本稿で

対象としたような協調計算は今後ますます重要になると考えられる．

5. 関連研究

分散メモリシステムでの並列プログラミングのための言語モデルやライブラリはこれまでに数多く提案されてきた．そのなかでも High Performance Fortran[7] や Unified Parallel C[8] は分散メモリに対する高性能プログラミングの実現を目標としており，本研究との関連が深い．しかしこれらは，ループの並列化や通信をコンパイル時にプログラムの解析により生成，挿入をすることで，データのイメージや通信のタイミングが不明瞭になり，性能チューニングを困難にしてしまうことが問題に挙げられる．一方，XMP はコンパイル時またはランタイムによる完全自動な並列化機能は提供せず，指示文をプログラマによって明示的に記述させる．これにより，コンパイラによるコード変換がプログラマに対し明確になるため，性能チューニングが容易になると考えられる．

また，GPU/CPU 協調計算に関する研究として，大島らの研究[9] があげられる．大島らは，GPU を用いる演算において，CPU の負荷が低いことに注目し，GPU と CPU で並列に演算をすることで GPU のみを使用する時よりも高速化を実現している．また，遠藤らによる研究[10] は，CPU と計算加速装置である GPU 及び ClearSpeed による Linpack ベンチマークにより，CPU のみを使った時と比較して，約 2.5 倍の速度向上を得ている．本研究で行う XMP による GPU/CPU 協調計算においても，GPU と CPU で並列に演算することで，GPU や CPU のどちらか一方を使うときよりも高速化が図れることが期待できる．

6. まとめ

本研究では，XMP の GPU クラスタ向け拡張仕様として XMP/GPU を提案し，その基本記述仕様と，コンパイラ実装によって生成されるべきコード様式について検討した．そして，XMP/GPU が想定するプログラムの評価により，GPU のみで計算した場合に比べて最大で約 1.7 倍の速度向上を得ることができた．これにより，XMP/GPU を用いた GPU/CPU 協調計算の有用性が示されたと言える．また，GPU/CPU 協調計算による性能向上のためには，GPU と CPU の演算時間が近くなるように GPU 割り当て率を最適化する必要があることがわかった．これは，XMP/GPU において GPU/CPU 協調計算を制御する GPU 割り当て率の指示方法が非常に重要であることを示している．

今後の課題として，この評価をもとに様々なアプリケーション，問題サイズによる最適な GPU 割り当て率の決定をする．また，NVIDIA の Fermi アーキテクチャを搭載した GPU で評価し，性能測定をする．また，本研究は XMP で GPU 間のデータ通信や，GPU/CPU 協調計算を吸収するためのランタイムライブラリの実装の予備評価であり，この評価をもとに XMP/GPU コンパイラの実装を進める予定である．

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) CUDA Programming Guide for CUDA Toolkit 3.2. http://developer.nvidia.com/object/gpucomputing.html.
2) Intel. Intel AVX. http://software.intel.com/en-us/avx/.
3) 李珍泌, 朴泰祐, 佐藤三久. 分散メモリ向け並列言語 XcalableMP コンパイラの実装と性能評価. 先進的計算基盤システムシンポジウム SACSIS2010 論文集, pp. 63–70, 2010.
4) Intel. Sandy bridge. http://software.intel.com/en-us/articles/sandy-bridge/.
5) AMD. Interlagos. http://ir.amd.com/phoenix.zhtml?c=74093&p=irol-2010analystday.
6) Intel. Nights Ferry. http://www.intel.com/technology/architecture-silicon/mic/index.htm.
7) High Performance Fortran 言語仕様書 Version 2.0. http://www.hpfpc.org/jahpf/spec/hpf-v20-j10.pdf.
8) UPC Language Specifications V1.2. http://upc.lbl.gov/docs/user/upcspec1.2.pdf.
9) 大島聡史, 吉瀬謙二, 片桐孝洋, 弓場敏嗣. CPU と GPU を用いた並列 GEMM 演算の提案と実装. 情報処理学会論文誌. コンピューティングシステム, Vol.47, No.12, pp. 317–328, 2006-09-15.
10) 遠藤敏夫, 額田彰, 松岡聡, 丸山直也. 異種アクセラレータを持つヘテロ型スーパーコンピュータ上の Linpack の性能向上手法. 情報処理学会研究報告, Vol. 2009-HPC-121, No.24, p.8, 2009-08.