Vol.2013-HPC-140 No.40
2013/8/2

並列言語XMP-devにおけるGPU/CPU動的負荷分散機能

小田嶋 哲哉[1,a)]　朴 泰祐[1,2]　佐藤 三久[1,2]　塙 敏博[2]　児玉 祐悦[1,2]　Raymond Namyst[3]
Samuel Thibault[3]　Olivier Aumage[3]

概要：GPU と CPU によるハイブリッドワークシェアリングでは，それぞれの計算リソース間のロードバランスが重要である．我々はこれまでに，この問題に対して PGAS 並列言語である XMP-dev/StarPU コンパイラ及びランタイムシステムを開発している．これによって，GPU/CPU のロードバランスを最適化し，CPU の演算性能を加える事で GPU のみの演算に対して速度向上が原理的に得られることを確認してきた．本稿では，XMP-dev/StarPU コンパイラにアプリケーションの実行中にユーザレベルで動的に負荷バランスを調整する機能及び API を実装した．結果として，いくつかのベンチマークで GPU のみの演算よりも最大で 40%の性能向上を得ることができた．

1. はじめに

近年, 高い演算性能と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）や OpenCL [2] によって GPU を用いたプログラミングが容易になり，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 クラスタが数多く出現し，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（Message Passing Interface）や OpenMP などのフレームワークを組み合わせているため，プログラミングが複雑である．GPU クラスタでは CUDA や OpenCL などによる GPU プログラミングが加わることで，より複雑になり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，演算をオフロードする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 のコア数増加や，SIMD（Single Instruction Multiple Data）命令により，CPU の演算能力は飛躍的に上昇し，計算リソースとしてこれを無視することができない．

一方，大規模分散メモリ環境における次世代並列プログラミング言語として，筑波大学が中心となって，PGAS（Partitioned Global Address Space）並列プログラミング言語 XcalableMP [3]（以降「XMP」と略す）の開発を進めている．このアクセラレータ（特に GPU）向けの拡張仕様として XcalableMP acceleration device extension（以降「XMP-dev」と略す）があり，バックエンドコンパイラに CUDA や OpenCL を用いた XMP-dev/CUDA [4]，XMP-dev/OpenCL [5] が開発されている．しかし，これらは指示された特定のループ処理をすべて GPU にオフロードするもので，GPU と CPU のハイブリッド処理は対象としていない．

これまでの研究 [6], [7] では，INRIA で開発されている StarPU [8] システムに着目し，XMP-dev との統合化を行い，XcalableMP-dev/StarPU（以降「XMP-dev/StarPU」と略す）を提案・実装した．これによって，GPU/CPU 協調計算を高水準並列言語で記述することができ，低いプログラミングコストで計算リソースの有効活用が可能になる．そして，GPU クラスタにおいて GPU のみを計算に利用するよりも，CPU の計算リソースを有効活用し，高い性能を得ることができた．しかし，多くの場合において十分な性能を得ることができなかった．その原因として，タスクサイズと GPU・CPU の性能差が関係していることがわかった．

本稿では，この問題を解決するために XMP-dev/StarPU に新たにデバイスごとに静的・動的にタスクサイズをコントロールする機能を導入する．これによって，問題やサイ

[1] 筑波大学 大学院 システム情報工学研究科
Graduate School of System and Information Engineering, University of Tsukuba
[2] 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[3] Bordeaux Sud-Ouest INRIA research center
[a)] odajima@hpcs.cs.tsukuba.ac.jp

ズによって変わるタスクサイズを，各デバイスの性能に合わせて調節することができ，結果として性能向上が得られることを示す．

2. XMP-dev と StarPU

2.1 XMP の概要

XMP-dev は，アクセラレータを持つ並列計算機向けの XMP の拡張言語仕様である．ここでは，XMP について最小の説明をする．詳細については [9] を参照されたい．

XMP は，分散メモリ型並列計算機上でのプログラミングを行うための PGAS 並列言語である．逐次のプログラムに OpenMP に類似した指示文を挿入することで，データの分散や同期，並列計算を行うことができる．そのため従来の MPI と比較して，少ない記述量で並列化が可能であり，ローカル配列のインデックス変換等に気を使う必要もなく，プログラムのコーディングやデバッグの時間が短縮され，生産性が大幅に向上する．また，XMP では実行単位のプロセスを「ノード」と定義している．XMP では通常，メモリアクセスはローカルメモリのデータに対する参照である．しかし，他ノードのデータを参照するには XMP の指示文を使い，ノード間通信をする必要がある．

2.2 XMP-dev の概要

XMP-dev [4] は，GPU クラスタなどのアクセラレータを搭載した分散メモリ環境向けの並列言語である．XMP-dev が扱うアクセラレータは，ホストと独立したメモリ（以降「デバイスメモリ」と呼ぶ）を持っている．XMP-dev では，XMP にいくつかの指示文を追加することで，ホスト-デバイス間のデータ転送や，デバイス上で loop 文の並列化などを簡潔に記述することができる．これらの指示文と従来の XMP の指示文を組み合わせることで，アクセラレータを持つクラスタ上での並列化が可能になる．CUDA や OpenCL を MPI と組み合わせ使うことなく，プログラムを簡潔に記述できる点が大きなメリットである．

図 1 に XMP-dev のプログラム例を示す．XMP-dev は XMP の拡張仕様であるため，従来の指示文をそのまま利用することが出来る．3〜6 行目は XMP の指示文であり，データの分散などを行う．10〜13 行目は XMP の loop 指示文であり，ホストの CPU 上で実行される．15〜24 行目までが XMP-dev の指示文であり，すべて「#pragma xmp device」から始まる．

15 行目の replicate 指示文は，デバイスメモリへの配列を確保するものである．図 1 では，16〜24 行目のスコープ内において，デバイス上でのメモリ確保が保証されており，スコープから抜けるとデータは解放される．17，23 行目の replicate_sync 指示文は，replicate 指示文のスコープ内で利用することができる．これはホストメモリとデバイスメモリのデータ通信を行う．通信の方向は sync_clause

```
1  int x[N], y[N];
2
3  #pragma xmp nodes p(4)
4  #pragma xmp template t(0:N-1)
5  #pragma xmp distribute t(BLOCK) onto p
6  #pragma xmp align [i] with t(i) :: x, y
7
8  int main() {
9    int i;
10 #pragma xmp loop on t(i)
11   for (i = 0; i < N; i++) {
12     x[i] = func(i); y[i] = func(i);
13   }
14
15 #pragma xmp device replicate (x, y)
16   {
17 #pragma xmp device replicate_sync in (x, y)
18
19 #pragma xmp device loop on t(i)
20     for (i = 0; i < N; i++)
21       y[i] += x[i];
22
23 #pragma xmp device replicate_sync out (y)
24   }
25 }
```

Execution HOST
Data Allocate
Data Copy H -> D
Execution Device
Data Copy D -> H
H : Host, D : Device

図 1 XMP-dev のサンプルコード

で制御する．「in」はホストからデバイスへ，「out」はその逆である．19 行目の device loop 指示文は XMP の loop 指示文同様に，直後の for 文をデバイス上でワークシェアリングする．この for 文は，XMP-dev のコンパイラがデバイスで動作する関数とその関数を呼び出すための関数に変換される．アクセラレータでは，多数のスレッドが動作するため，XMP-dev では 1 スレッドに loop 文の 1 反復の計算を割り当てるように実装されている．

2.3 StarPU の概要

StarPU に関しては文献 [10], [11] に詳しいが，ここでは概要を示す．StarPU では，計算に必要なデータの集合，実行の単位を「タスク」と定義している．StarPU は，このタスクを様々な計算リソースに割り当てたり，タスク間のデータの依存関係を解析し調整したりすることができるランタイムシステムである．対象としている計算リソースにマルチコア CPU，GPU，Cell Broadband Engine などが挙げられる．本稿では，マルチコア CPU，GPU （特に NVIDIA の CUDA が動作する）についてのみ言及する．また，StarPU はタスク間のデータ依存の制御をするために，全ての計算リソースで共有するデータプールに配列データを登録する．計算に必要なデータは，データプールに登録され，計算リソースで共有される．タスクが生成された時に，必要なデータがデータプールから割り当てられ，データの属性（read-only や write-only など）に応じてランタイムが依存関係を管理する．

StarPU は，多数のタスクを各デバイスに割り当てることによって，GPU と CPU 間のロードバランスをとることができる．また，何度かプログラムを実行し割り当てた情

報を保持することで，割り当て方を自動で調整する機能が備わっている．しかし，タスクを割り当てる単位が「CPU core」と「GPU」という単位であるため，デバイス間の性能差が非常に大きい．これが原因で，非常に多くのタスク数がなければ自動調節がうまく機能せず，特に GPU を複数搭載する環境においては性能差が顕著になる．

3. XMP-dev/StarPU の実装

XMP-dev/StarPU の従来実装は [6], [7] に詳しい．ここでは従来の実装の概要を説明する．

3.1 XMP-dev と StarPU

StarPU はノード内におけるデータの管理，データ転送，タスクの生成と発行などを担い，ヘテロジニアスな環境でロードバランスを取ることが潜在的に可能である．しかし，StarPU を使ったアプリケーションの実装は逐次コードから変更する場合，codelet（関数ごとの実行ポリシー）の記述やデータの分割等，プログラミングコストが大きく，ユーザアプリケーションを直接この上で書くことは適切でない．また，StarPU のランタイムでは MPI によるマルチノード上でデータの分散やタスクの実行が可能であるが，マスターノードによって全てのデータ管理やスケジューリングが行われる．そのため，プログラミングにおいてノード番号を指定する必要があるため複雑になりがちである．クラスタなどの分散メモリ環境では更に複雑になることが容易に想像できる．

そこで，我々は XMP-dev と StarPU を組み合わせた XMP-dev/StarPU を提案・実装した．これによって，マルチノード上での GPU/CPU ハイブリッド計算を容易に行うことができるため，機能性や性能の向上が期待できる．

XMP-dev から見たメリット

XMP-dev の device として StarPU を利用する．従来の XMP-dev の実装では，バックエンドは CUDA [4] と OpenCL [5] が用意されていたが，双方とも計算は GPU のみで実行され，CPU は計算に参加しない．そこで，バックエンドのスケジューラとして StarPU を用いることで，GPU と CPU の計算リソースを余すことなく利用することができ，性能向上が見込める．

StarPU から見たメリット

StarPU はプログラミングが複雑になりがちなため，様々なアプリケーションに適用することが難しい．そこで，XMP-dev の指示文で StarPU のデータプールへの登録などを行えるランタイムを作成する．そして，XMP-dev によって生成されたデバイス関数を実行の対象とすることでデバイスでの実行が可能になり，簡潔に GPU/CPU のワークシェアリングが可能になる．

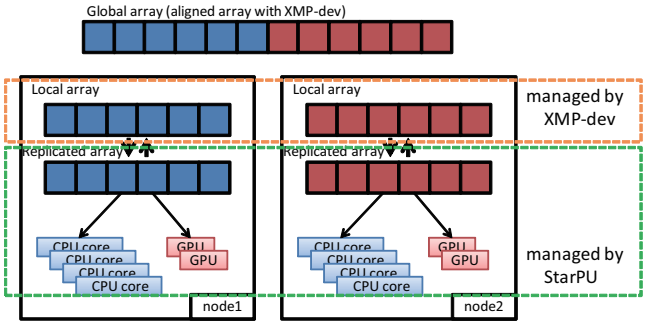

図 2 XMP-dev/StarPU の実行モデルの概要

3.2 従来実装の方針と問題

我々は先行研究において，StarPU を XMP-dev のタスクスケジュールエンジンとして用いた XMP-dev/StarPU を実装した [6], [7]．**図 2** に，XMP-dev/StarPU の実行モデルの概要を示す．XMP-dev/StarPU の処理は以下のようになっている．

- XMP-dev は Global array を分割し，各ノードに Local array として分配する．
- XMP-dev/StarPU のランタイムは，Local array を複製し，Replicated array を生成する．
- Local array は MPI によるノード間通信に利用され，Replicated array は StarPU のデータプールに登録される．
- Replicated array は等分割され，いくつかのタスクを生成する．そして，各デバイスに割り当てられる．
- ユーザはノード間通信などを行う際に，明示的に Local array と Replicated array 間の同期を行う．

先行研究 [6], [7] の実装では，XMP-dev/StarPU の性能は XMP-dev/CUDA に対して 45%程度であり，非常に低かった．この原因として，Replicated array を等分割し，タスクとして GPU と CPU core に割り当てていたため，GPU と CPU core の性能差が影響し，実行時間の差が大きくなってしまったことが挙げられる．特に，タスクサイズが大きい時には GPU は高い性能を得ることができるが，CPU core は GPU に比べ演算性能が低いため実行時間が大きくなり，GPU が空転してしまった．逆にタスクサイズが小さい時は実行時間の差は小さくなるが，GPU の演算性能を十分に引き出すことができず，結果として全体の性能を低下させてしまった．この経験より，一定のサイズの問題では，StarPU を均等サイズのタスクに適用しただけでは最適な性能が得られないという結論に至った．

3.3 動的負荷分散機能による性能向上

従来の XMP-dev/StarPU の実装では，同じサイズのタスクを各リソースに割り当てていたため，GPU と CPU core の性能差によって十分な性能向上を得ることができなかった．そこで先行研究 [6], [7] では，静的に GPU と CPU

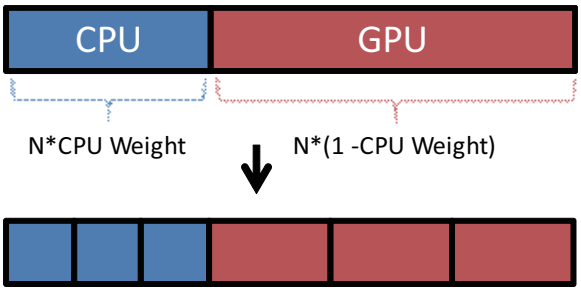

図 3 Replicated array の分割イメージ

core に異なるサイズのタスクを割り当てるようにして，デバイス間の負荷分散の調節を行い，この方向で最適化が原理的に可能であることを確認した．しかし，これらの実装ではタスクサイズの比率はコンパイラのパラメータとしてハードコードされており，ユーザ側で自由に変更することができなかった．

タスクサイズの調整には，GPU と CPU の性能差が大きく影響する．GPU/CPU 協調計算では，GPU と CPU の実行時間をできるだけ近くすることで，待ち時間のオーバーヘッドを最小限にでき，効率的に計算リソースを使うことができる．GPU に割り当てるタスクサイズは大きくすることで，ホスト-デバイス間の転送オーバーヘッドを小さくすることができ，GPU が十分な性能を出すことが可能になる．一方，CPU には大きなタスクサイズを割り当ててしまうと，実行時間が大きくなりすぎてしまう．そのため，GPU の実行時間に応じて CPU のタスクサイズを小さくする必要が出てくる．

この問題を解決するために，我々は “CPU Weight” という値を用いて，loop 文のワークシェアリングにおいて GPU と CPU 間のタスクサイズのバランスを調整することを提案する．タスクサイズのバランスを取るために，まず XMP-dev/StarPU のランタイムで生成した Replicated array を GPU で計算する領域と，CPU で計算する領域に 2 分割する．**図 3** に，Replicated array の分割のイメージを示す．この配列は図 2 の Replicated array である．図 3 の青い部分が CPU，赤い部分が GPU が計算を行う領域である．ここでは，要素数 N の配列を 1 次元分割しており，CPU Weight を用いてそれぞれの領域の割合を調整することができる．CPU Weight は CPU が計算する領域の割合（double 型）を示しているため，0 に近くなるほど CPU の計算量は減る．そして，2 つの領域はそれぞれ StarPU によっていくつかの小さなタスクに分割され，各リソースに割り当てが行われる．図 3 では，それぞれの領域が 3 つに分割されることで，GPU と CPU に割り当てられるタスクサイズが異なり，CPU には小さなタスクサイズを，GPU にはより大きなタスクサイズを割り当てることができる．このようにして，CPU Weight を用いることで GPU と CPU 間のロードバランスを調整することが可能になる．

```
for (int t = 0; t < STEP; t++) {
  // 演算など

  double cpu_time = xmp_cpu_wtime();
  double gpu_time = xmp_gpu_wtime();

  double cpu_ratio =
    cpu_time / (cpu_time + gpu_time) * 100;

  if (cpu_ratio > 50) new_cpu_weight -= 0.01;
  else                new_cpu_weight += 0.01;

#pragma xmp device reset_weight (new_cpu_weight)
}
```

図 4 reset_weight 指示文の例

しかし，最適な CPU Weight を静的に設定することは非常に困難であり，[6], [7] のように同じ問題サイズにおいて値を何度も変えつつ測定する必要が出てくる．そこで，CPU Weight をプログラムの実行中に動的に変化させるために新たな “reset_weight” 指示文を導入する．指示文は以下のとおりである．

```
double cpu_weight;
#pragma xmp device reset_weight (cpu_weight)
```

ユーザは reset_weight 指示文をワークシェアリングを行う loop 文に挿入することで，プログラムの実行特性に応じてユーザが自由に CPU Weight を変更することができる．しかし，図 3 で示しているように，Replicated array は 1 次元分割されているため問題によっては CPU Weight を決定すること難しい．例えば，行列積では問題サイズ N に対して，計算量は $O(N^3)$ になり，CPU Weight を変えることで実行時間が大きく変わりすぎてしまうことがある．このような問題に対しては，プログラム実行中に動的に実行時間のプロファイリングを行い，徐々に CPU Weight を最適な値に近づけていく手法が有効であると考えられる．実際，時間発展をしていくようなシミュレーションでは，TIME STEP ごとに CPU Weight を変更することは可能である．

ここで，**図 4** に reset_weight 指示文の利用例を示す．ワークシェアリングを行う loop 文中で，イテレーションが終了したら次のステップで使う新たな CPU Weight を計算している．XMP-dev/StarPU では，GPU と CPU に割り当てられたタスクの実行時間を得る関数（xmp_cpu_wtime() 及び xmp_gpu_wtime()）が用意されており，ここではそれを用いて CPU の実行時間がトータルの計算時間の何%を占めているかを計算する（cpu_ratio）．そして，その割合が 50%に近づくように，つまり GPU と CPU の実行時間がなるべく等しくなるように CPU Weight を調整してゆく．そして，計算によって新しい CPU Weight を次のステップで利用するために reset_weight 指示文で変更を行う．

表 1　評価環境（HA-PACS）

CPU	Intel Xeon E5-2670 2.6GHz
GPU	NVIDIA Tesla M2090
Main memory	DDR3 1600MHz 128GB
GPU memory	DDR5 6GB / GPU
Interconnection	InfiniBand QDR (2 rails)
OS	CentOS release 6.1 (Final)
CPU compiler	gcc 4.4.5
GPU compiler	CUDA 4.2
MPI	MVAPICH2 1.8.1
# of nodes	2
# of CPU/node	16 cores (8 cores 2 sockets)
# of GPU/node	4

図 4 では，非常に単純なアルゴリズムで CPU Weight を決定しているが，ユーザが最適なアルゴリズムを用いることでより早い収束を得ることも可能である．例えば，1 回ごとの調整幅を最初は大きくし，逆転してしまったらその半分にして微調整する等のアルゴリズムを用いることが考えられる．また，ワークシェアリングを行う loop 文が大きな時間発展ループの中に複数ある時，内部の loop ごとに GPU と CPU の実行時間が異なる場合がある．reset_weight 指示文を用いることで，loop ごとに最適な CPU Weight を設定することができ，全体の性能向上を得ることが期待できる．

4. 性能測定

本評価では，筑波大学の GPU クラスタである HA-PACS を用いる．評価環境は**表 1** に示すとおりである．本稿における評価では 268 台の計算ノード中の 2 ノードを用いた．StarPU は，GPU の通信やカーネル関数の起動などの管理のために 1GPU につき 1CPU core を割り当てる必要がある．そのため，4GPU を計算に用いる場合，計算に参加する CPU core は $16-4=12$ となる．評価に用いるベンチマークは N 体問題と行列積であり，ともに倍精度浮動小数点演算を行う．また，本稿では CPU Weight を最外ループの 1 イテレーションの実行時間によって決定する．特に，本評価は GPU と CPU の負荷バランスについて言及するため，実行時間の測定には MPI 通信の時間を含めていない．

N 体問題に関しては，最外の時間発展の loop 内で CPU Weight を変更する．行列積では，最外に TIME STEP の loop を作り，その中で行列積を何度も繰り返し，1 イテレーションの実行時間によって CPU Weight を決定する．また，行列積では GPU に MAGMA blas [12]，CPU に Gotoblas [13] を導入することで，各リソースの演算性能を出来るだけ引き出すことができると考えている．

まずは CPU Weight の動的変更について評価を行う．HA-PACS には 1 ノードに 4GPU が搭載されており，評価

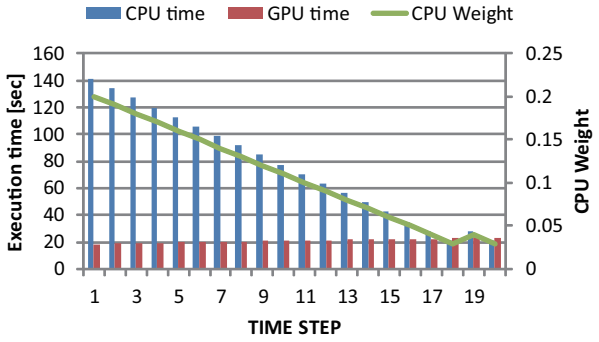

図 5　CPU Weight の推移：N 体問題（粒子数 $N=819200$）

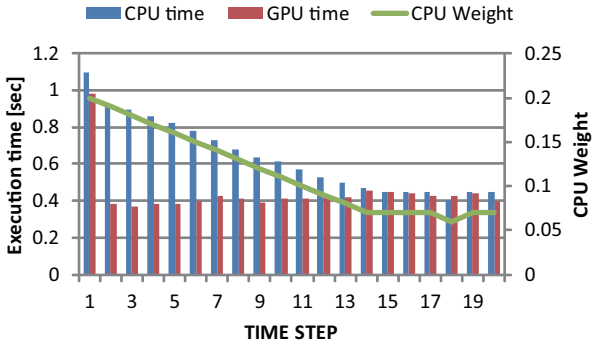

図 6　CPU Weight の推移：行列積（行列サイズ $8192*8192$）

では GPU 数を 4，CPU core 数を 12 とする．CPU Weight の初期値は 0.2 とし，これは図 3 では CPU : GPU = 1 : 4 となる．CPU Weight の決定については図 4 に示したアルゴリズムに則っている．

図 5 及び**図 6** はそれぞれ，N 体問題と行列積の TIME STEP ごとの CPU Weight の推移を示している．図 5，図 6 中の青いバーが CPU，赤いバーが GPU の，それぞれ最も遅いタスクの実行時間を表している．そして緑色の折れ線は CPU Weight を示している．左の縦軸はタスクの実行時間，右の縦軸は CPU Weight，横軸は TIME STEP である．両方のグラフにおいて，GPU の演算性能は CPU core 12 個に対して非常の大きいため，CPU Weight が大きい時に CPU がボトルネックになっていることがわかる．図 4 のアルゴリズムに則って CPU Weight を変更していくと，それぞれの実行時間が均衡していくことがわかる．それに伴って CPU Weight の値も収束し，最終的には最も実行時間の差が小さくなっている．ここでは，実験のために TIME STEP を 20 回で打ち切っているが，どちらのベンチマークでも GPU と CPU の実行時間がほぼ均衡するところまで CPU Weight の調整ができていることがわかる．当然ながら，この収束までの時間はアプリケーションと CPU Weight の更新方法に依存して変わるため，アプリケーションごとに最適な方法を選ぶ必要がある．このようにして，GPU と CPU 間のロードバランスを最適化し，か

表 2 N 体問題：20 STEP 目の実行時間

	1GPU		2GPU		4GPU	
N	time	CPU Weight	time	CPU Weight	time	CPU Weight
102400	1.352352	0.15	0.710417	0.07	0.434995	0.04
204800	5.058176	0.14	2.796911	0.07	1.679012	0.03
409600	20.03512	0.14	11.05233	0.07	6.014442	0.03
819200	78.62307	0.14	42.61331	0.07	23.0175	0.03

表 3 行列積：20 STEP 目の実行時間

	1GPU		2GPU		4GPU	
N	time	CPU Weight	time	CPU Weight	time	CPU Weight
1024	0.004595	0.29	0.005188	0.21	0.003716	0.19
2048	0.023057	0.27	0.016544	0.09	0.011162	0.02
4096	0.163541	0.28	0.116263	0.13	0.060841	0.05
8192	1.184421	0.29	0.691243	0.15	0.444981	0.07
16384	8.276425	0.29	5.167231	0.16	3.674877	0.09

つユーザがプログラムの特徴に応じて CPU Weight を静的に設定することなく，高いレベルの PGAS プログラミング言語で動的に記述することが可能になることがわかった．

表 2，**表 3** には，ノード内の GPU 数を変化せた時の 20 STEP 目の実行時間（ $max(cpu\_time, gpu\_time)$ ）とその時の CPU Weight を示している．それぞれの図は，GPU の数が増えるに従ってタスクの実行時間は短くなっており，全体の性能が向上していることがわかる．表 2 の $N = 102400$ では，1GPU を使うときには CPU は全体の計算の 15%を担当しており，4GPU では 3%である．ノード内の GPU が増えるにつれ GPU のタスクの計算時間は減り，それによって CPU Weight も小さくなっている．1GPU から 4GPU にした時に単純に CPU Weight が 1/4 にならないのは，同時に CPU core 数が 15 から 12 に減り，CPU の計算リソースが減ってしまうためだと考えられる．同様に表 3 の $N = 1024$ では、1GPU において CPU は全体の 30%の計算を担っている。行列積では、GPU・CPU ともに BLAS を用いており，各計算リソースの性能が出やすく，かつ問題サイズが小さいため CPU の割合が増えたと考えられる．特に行列積に関しては，GPU 数が増えた時にも CPU が演算する割合が多く，有効にリソースを使えていることがわかった．

最後に，GPU/CPU 協調計算の性能について述べる．本評価では，GPU のみを計算に利用した結果に対して，それに CPU を加えた場合の速度向上を述べる．**図 7**，**図 8** にそれぞれ XMP-dev/CUDA に対する N 体問題，行列積の 1 TIME STEP の相対性能を示す．相対性能が 1 より大きいと、GPU のみの演算よりも高速であることを示している．図 7 では，粒子数 102400 において約 20%の性能向上が得られた．各 GPU 数において，問題サイズが大きくなると一旦性能が低下するが徐々に性能が向上していくことが読み取れる．これは，問題サイズが大きくなると GPU の演算性能が飽和し演算性能の頭打ちが起き，そこに CPU の演算性能が加わることで GPU のみを使った時に対して性能のアドバンテージが生まれ，性能向上に至ると考えられる．これを確認するためにも，より大きな問題サイズでの

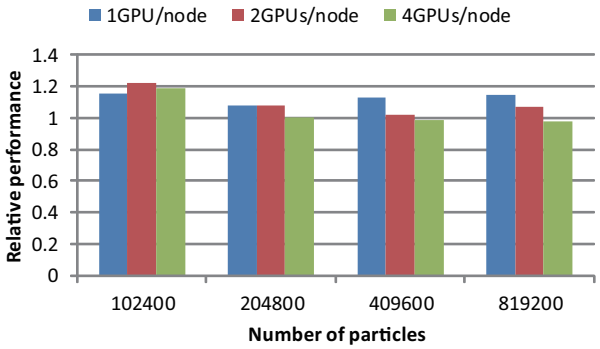

図 7 N 体問題：相対性能

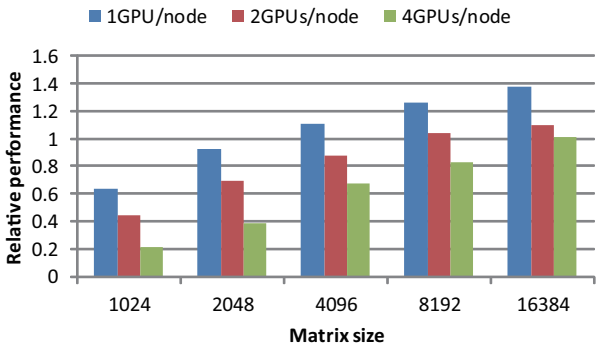

図 8 行列積：相対性能

比較が必要になる．図 8 は行列積の相対性能であるが，問題サイズが小さい時には GPU のみの計算よりも遅くなっている．行列積は 1 TIME STEP の実行時間が短く，配列の分割や集約のオーバーヘッドが見えてしまい速度低下に至ったと考えられる．問題サイズを大きくしていくと，速度向上は最大で約 40%得られた．問題サイズを大きくすれば 1GPU だけでなく 2，4GPU の時にも大きな速度向上が得られると予想できる．本実験では，1 次元分割を用いているため，GPU のメモリ制約によりこれ以上大きなサイズでの検証ができていないが，今後は 2 次元分割での測定を行いスケーラビリティの検証をしたい．

5. 関連研究

アクセラレータ向けのコンパイラとして PGI Accelerator Compilers [14] や HMPP Workbench [15] が挙げられる．これらは GPU を含めた様々なアクセラレータを対象とした指示文を提供する．PGI Accelerator compilers は NVIDIA 社の CUDA が動作する GPU 向けのソースコードを生成することができる．HMPPWorkbench はバックエンドコンパイラとして CUDA や OpenCL を用いているため，逐次のソースコードに指示文を挿入することでマルチコア CPU と GPU などのアクセラレータによるハイブリッドプログラミングが可能になっている．しかし，これはシングルノード内での動作を想定しているため，GPU ク

ラスタのような分散メモリ型の環境には対応していない．XMP-dev/StarPU はこの問題を解決する．

また，MAGMA [12] と呼ばれる NVIDIA の GPU 向けの BLAS（Basic Liner Algebra Subprograms）に StarPU を適用した研究 [16] がある．これはライブラリレベルで GPU と CPU の協調計算を行なっており，言語レベルで GPU/CPU 協調計算を行う XMP-dev/StarPU のフレームワークにも適用が可能だと考えられる．性能に関しても，StarPU を用いることでコレスキー分解を GPU1 台のみ使う実行時間に対して，Intel Nehalem X5550 6cores, NVIDIA FX5800 3 台という環境で最大 4 倍近い速度向上が得られている．XMP-dev/StarPU では基本的にループ分割によるワークシェアリングで記述できる問題については，より柔軟に両者の協調計算を記述できる．ただし，MAGMA で性能が向上している計算ケースについては，XMP-dev/StarPU による実装との比較等の追実験が必要と考えている．

6. まとめ

GPU クラスタ向け並列言語 XMP-dev と GPU/CPU 協調計算を行うための StarPU を組み合わせた XMP-dev/StarPU コンパイラの上で，GPU と CPU の大きな演算性能の差に着目し，適切な負荷分散を動的に行うことができるような拡張を行った．このため，CPU の負荷割り当てを調整する API として指示文を追加し，ランタイムライブラリを追加した．N 体問題や行列積にこの新しい機能を持った XMP-dev/StarPU を適用し，XMP-dev/CUDA に対して約 20%〜40%の速度向上を得ることができた．

今後の課題として，様々アプリケーションや問題サイズやノード数のスケールアップについて評価をし，多次元分割の実装をしてより大きな速度向上を得られるようにする予定である．

謝辞 本研究の一部は JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」，研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」，および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] NVIDIA. CUDA C Programming Guide. http://developer.nvidia.com/nvidia-gpu-computing-documentation.
[2] KHRONOS GROUP. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.
[3] XcalableMP. http://www.xcalablemp.org/.
[4] 李 珍泌, チャン トゥアン ミン, 小田嶋 哲哉, 朴 泰祐, and 佐藤 三久. PGAS 並列プログラミング言語 XcalableMP における演算加速装置を持つクラスタ向け拡張仕様の提案と試作. **情報処理学会論文誌 論文誌トランザクション**, 2011(2):33–50, Apr 2012.
[5] T. Nomizu, D. Takahashi, L. Jinpil, T. Boku, and M. Sato. Implementation of XcalableMP Device Acceleration Extention with OpenCL. In *Multicore and GPU Programming Models, Languages and Compilers Workshop (PLC 2012)*, pages 2394 –2403, May 2012.
[6] 小田嶋哲哉, 李珍泌, 朴泰祐, 佐藤三久, 塙敏博, 児玉祐悦, Raymond Namyst, Samuel Thibault, and Olivier Aumage. GPU クラスタ向け並列言語 XMP-dev における GPU/CPU 協調計算. **情報処理学会研究報告**. *[**ハイパフォーマンスコンピューティング**]*, 2013(25):1–9, Feb 2013.
[7] T. Odajima, T. Boku, T. Hanawa, Jinpil Lee, and M. Sato. GPU/CPU Work Sharing with Parallel Language XcalableMP-dev for Parallelized Accelerated Computing. In *Sixth International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, pages 97–106, Sept. 2012.
[8] StarPU. http://runtime.bordeaux.inria.fr/StarPU/.
[9] 李 珍泌, 朴 泰祐, and 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. **情報処理学会論文誌**, 3(3):153–165, Sep 2010.
[10] A Unified Runtime System for Heterogeneous Multi-core Architecturesq. In E. Cesar, M. Alexander, A. Streit, J. Traff, C. Cerin, A. Knupfer, D. Kranzlmuller, and S. Jha, editors, *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415, pages 174–183. 2009.
[11] C. Augonnet, S. Thibault, and R. Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Technical Report RR-7240, INRIA, March 2010.
[12] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 User Guide. http://icl.cs.utk.edu/magma/.
[13] Texas Advanced Computing Center - GotoBlas2. http://www.tacc.utexas.edu/tacc-projects/gotoblas2.
[14] PGI Accelerator Compiler. http://www.softek.co.jp/SPG/Pgi/Accel/index.html.
[15] HMPP Workbench. http://www.caps-entreprise.com/hmpp.html.
[16] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, Cheaper, Better a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*, volume 2. September 2010.