Vol.2013-HPC-138 No.25
2013/2/22

GPUクラスタ向け並列言語XMP-devにおけるGPU/CPU協調計算

小田嶋 哲哉[1,a)]　李 珍泌[1]　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 クラスタ上でのプログラミングは，様々なプログラミングモデルが直交しており，複雑になってしまうことが多い．本稿では，分散メモリ環境向け高水準並列プログラミング言語である XMP を GPU クラスタ等のアクセラレータを持つ並列計算機向けに拡張した言語仕様 XMP-dev において，GPU と CPU によるハイブリッド協調計算を実現する XMP-dev/StarPU を提案，実装を行った．XMP-dev は，ノード間通信をベースとし，データの分散や GPU へのオフローディングが可能な並列言語である．しかし，CPU を計算リソースとして GPU と並行して用いるには複雑なプログラミングが必要である．これに対し，StarPU をバックエンドのスケジューラとすることで，計算をタスクという単位で GPU や CPU へスケジューリングすることによりワークシェアリングが可能になる．本稿では，実際のアプリケーションに XMP-dev/StarPU を適用することで，GPU のみを計算に利用するときよりも 1.1〜1.2 倍ほどの高速化が可能であることを示した．また，指示文ベースのプログラミングモデルである XMP-dev/StarPU は，通常のプログラミングよりもコストが大幅に削減できることも示した．

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ったことで，HPC（High Performance Computing）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，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 のコア数増加や，SIMD （Single Instruction Multiple Data）命令により，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．

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 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 [*1] [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．そこで，XMP-dev の枠組みで GPU と CPU を負荷分散の対象として同時に利用するハイブリッド処理を検討する．本研究では，INRIA で開発されている StarPU [6] システムに着目し，XMP-dev との統合化を行う．StarPU はランタイムレベルで GPU と CPU へタスクのスケジューリングが可

[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

能である．しかし，StarPU はデータの登録やタスクの発行などプログラム記述量が増えてしまう．そこで，これを XMP-dev に組み込むことで GPU/CPU 協調計算を高水準並列言語で記述することが可能となり，結果として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と考えられる．これによって，GPU と CPU へのデータの分散及び負荷分散をし，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．

本稿では，XMP-dev と StarPU を組み合わせたコンパイラの設計と実装について述べる．また，XMP-dev と StarPU を組み合わせるに当たっての問題について考察し，解決策を提案・実装し，実際のアプリケーションに適用させ，ヘテロジニアスな環境でのハイブリッドプログラムやプログラマビリティについて検討をする．

2. XcalableMP と XMP-dev 及び StarPU の概要

2.1 XcalableMP（XMP）

XMP に関しては文献 [7] に詳しいが，ここでは本稿を理解するための最小限の説明をする．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．

図 1 に XMP のプログラム例を示す．「#pragma xmp」から始まる行が XMP の指示文である．はじめに，nodes 指示文でプログラムを実行するノードを指定する．図 1 では，4 つのノードを用いる．次に，template 指示文でノードにまたがった仮想的なデータの宣言を行う．XMP のデータの分散やループ文の分割には，この template が用いられる．distribute 指示文は，template t を各ノードにどのようにマッピングするかを宣言する．指示文のオプションとしてデータの分割方法を選択でき，ここではブロック分割を行なっている．最後に align 指示文は，template t によって宣言されたデータの分散を実際の配列 x に適応し，データの分割を記述する．これによって，各ノードは割り当てられたデータ分だけをローカルメモリに持つことになる．

loop 指示文は，直後の for 文をノードの集合でワークシェアリングする．ループ文のイテレーションの分割は template t によって設定されている．XMP では，データアクセスはローカルメモリを参照するため，イテレーションの分割とデータの分割の整合性をユーザがとる必要がある．

```
1  int x[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 x[i] with t(i)
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);
13 }
```

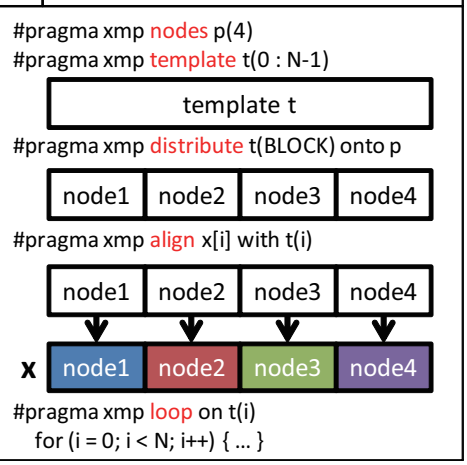

図 1 XMP のサンプルコード

2.2 XMP-dev

我々は，XMP をアクセラレータを持つ並列計算機向けに拡張した言語仕様，XMP-dev [4] を提案している．ここで言う device は，ホスト（CPU）の処理の一部を請け負うアクセラレータを表す．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 2 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 行目は図 1 と同様である．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．

*1 オリジナルの実装は CUDA であり，OpenCL の実装と区別をするために本稿ではこのような表記をする．

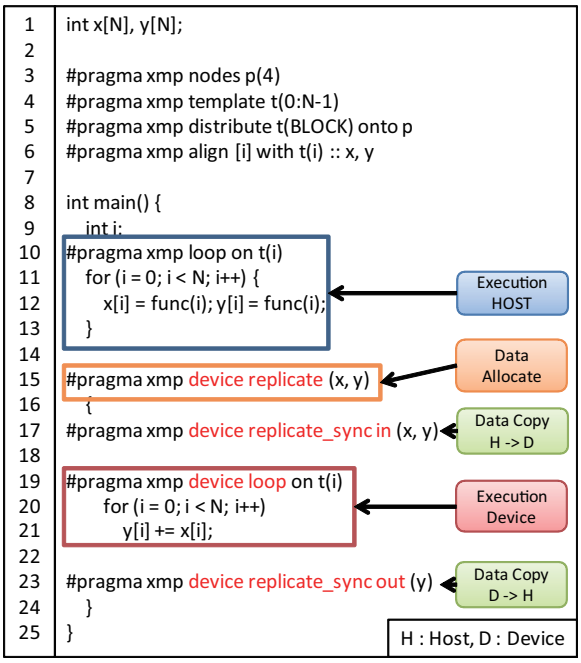

図 2 XMP-dev のサンプルコード

```
#pragma xmp device replicate (list)
```

replicate 指示文はデバイスメモリへ配列を確保するものである．デバイスでの計算に使う配列データは，ユーザが図 2 の 6 行目でホストメモリ，15 行目でデバイスメモリに確保しなければならない．図 2 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 free される．

```
sync_clause := in (list) | out (list)
#pragma xmp device replicate_sync sync_clause
```

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 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．replicate スコープを抜けた後，ホストでデータを参照する必要がある時には，必ず replicate_sync out を使わなければならない．

```
#pragma xmp device loop
    loop-statement
```

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

StarPU に関しては文献 [8] [9] に詳しいが，ここでは本稿を理解するための最小限の説明をする．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はマルチコア 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．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，計算を実行することが可能になる．

2.3.1 メモリ管理

計算に必要なデータは，StarPU のデータプールに登録されている必要がある．StarPU ではこのデータを starpu_data_handle_t という型で登録する．タスクはこの handle を参照することで，正しい値が参照できることが保証されている．StarPU では，計算に必要なデータは最も近いメモリ（CPU はメインメモリ，デバイスはデバイスメモリ）にデータが存在するよう，計算が実行される前に通信が発生する．例えば，あるデバイスで更新した値をホストの CPU で参照するとき，デバイスからホストへのデータ転送が起き，常に最新の値を参照することができる．このデータ参照のポリシーは StarPU のオプションで制御することができる（Read only, Wite only, Read Write, etc...）．また，デバイスで計算した時に使ったデータは再利用性がある可能性が大きいため，ユーザが明示的にデータの破棄をしなければそのままデバイスメモリ上に存在し続ける．これによってホスト-デバイス間の通信を最低限にすることができる．

2.3.2 タスクの実行

StarPU では配列の確保や初期化を行った後，starpu_data_register 関数を使って StarPU のデータプールに登録をする．この状態であれば，StarPU のタスクはデバイス上で実行できるようになる．しかし，このままでは 1 つの計算リソース上でしか実行することができない．そこで，starpu_data_partition 関数を使って配列データを細かく分割する．この分割した単位を「チャンク」と呼ぶ．複数のチャンクをまとめてタスクとし，このタスクの要素数を本稿では「タスクサイズ」と定義する．starpu_task_submit 関数を使って codelet と呼ばれるタスクの動作を制御する構造体に紐付けし，CPU や GPU に計算を割り当てる．これによってヘテロジニアスな環境でのワークシェアリングが可能になる．計算が終わった後に，各デバイスメモリに存在するデータをメインメモリに戻すのが starpu_data_unpartition 関数である．そして starpu_data_unregister を使って StarPU のデータ管理を終了させる．

3. XMP-dev/StarPU の実装

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 の記述やデータの分割等，プログラミングコストが大きいことが問題である．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 を提案する．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 のメリット

XMP-dev の device として StarPU を利用することを考える．現在 XMP-dev の実装において，バックエンドは CUDA [4] と OpenCL [5] がある．双方とも計算は GPU のみで実行されている．そのため，CPU が空転してしまう．そこで，バックエンドのスケジューラとして StarPU を適応する．これによって，GPU/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 のメリット

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応することが難しい．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る．CPU のコードは逐次のコードをそのまま利用することができ，これによって容易に GPU/CPU のワークシェアリングが可能になる．

3.2 XMP-dev/StarPU の実装方針

XMP-dev のコンパイラとランタイムを変更することで，StarPU をバックエンドで動作するスケジューラになるように実装を行う．**図 3** に XMP-dev/StarPU の実行モデルの概要を示す．XMP-dev/CUDA は，XMP の template を用いて Global array を各ノードに分散し，Local array という形で保持している．このデータを用いた計算を GPU にオフローディングすることで，マルチノード上で GPU による計算が可能であった．XMP-dev/StarPU ではノード間のデータの分散は従来通りであるが，Local array をそのまま GPU にオフロードするのではなく，StarPU のスケジューラを通して GPU や CPU に割り当てる．StarPU を用いるにあたって，Local array をいくつかのチャンクに分割し，複数のタスクを生成してスケジュールを行う．これによって GPU と CPU で協調計算が可能になる．

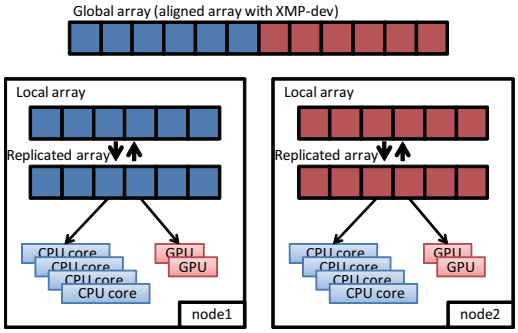

図 3 XMP-dev/StarPU の実行モデルの概要

ここで，図 3 の Replicate array について説明する．Local array を直接複数の StarPU のデータプールに登録することは可能である．しかし，Local array はノード間のデータ交換などに使われているため，この配列を StarPU のデータプールに登録するとコンパイラの様々な場所で acquire-release といったデータ管理の移譲のような制御が必要になる．そこで，簡易に実装するために Local array と同様の配列 Replicate array を作成し，データプールに登録をする．この配列をノード間通信などが必要になった時に Local array と同期することで Local array を分割した時と同等な動作が期待できる．

XMP-dev/StarPU では，XMP-dev/CUDA にはなかった StarPU の制御が必要になる．そのため，指示文の動作が変わる．主な指示文を以下に示す．

#pragma xmp device replicate

XMP-dev/CUDA では，デバイスメモリへの配列の確保を行うための指示文であった．しかし，StarPU ではタスクが発行された時に配列の確保が行われるため直接配列の確保をすることはない．XMP-dev/StarPU では，この指示文で指定された配列は StarPU のデータプールに登録され，同時に Replicate array がホストのメモリ上に確保され，指定されたタスク数に分割される．

#pragma xmp device replicate_sync

XMP-dev/CUDA では，ホスト-デバイス間のデータ転送を行うための指示文であった．しかし，これも StarPU ではタスク実行時に非同期に行われる．XMP-dev/StarPU では device replicate 指示文で登録された配列の Replicate array へのメモリコピーが行われる．従来と同様に「in」と「out」というデータコピーの向きがあり，「in」が Local array から Replicate array へ

表 1 評価環境（HA-PACS）

CPU	Intel Xeon E5-2670 2.6GHz
GPU	NVIDIA Tesla M2090
Main memory	DDR3 11600MHz 128GB
GPU memory	DDR5 6GB
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	16cores (8cores 2sockets)
# of GPU/node	4

のコピー，「out」がその逆である．

#pragma xmp device loop

devic loop 指示文は，GPU のデバイス関数の変換及び GPU へのオフローディングを担っていた．XMP-dev/StarPU ではデバイス関数の他に StarPU で実行するための形式で書かれた関数が 2 つ（GPU，CPU），CPU の計算関数，ホストから呼ばれる関数の計 5 つの関数が for 文 1 つから生成される．この GPU/CPU 用の関数を StarPU の task_submit 関数で実行する対象とすることで GPU と CPU で for 文のワークシェアリングが可能になる．

4. 評価

先行研究 [10] で実装された XMP-dev/StarPU のプログラマビリティと性能を評価する．評価には，筑波大学の GPU クラスタ HA-PACS を用いる．HA-PACS の諸元を **表 1** に示す．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 が搭載されている HA-PACS では計算に使える CPU core は $16-4=12$ となる．また，評価に用いる GPU の数を 1GPU/node とするため，本稿における評価では CPU core 数は 12，GPU 数は 1 に固定する．評価に用いるベンチマークは N 体問題と行列積である．

4.1 プログラマビリティの評価

XMP-dev/StarPU のプログラマビリティについて評価を行う．StarPU は，データの登録や分割，タスクの生成・発行を StarPU の API にそって行う必要がある．これは，実際に MPI や CUDA を駆使してプログラミングを行うよりもコストが小さくなるが，依然としてプログラミングコストは高いままである．そこで，XMP-dev/StarPU を使うことでプログラミングコストがどれほど減るかを確認し，プログラマビリティについて評価を行う．**表 2** に N 体問題を逐次で記述した時，XMP-dev/StarPU の枠組みで記述した時，MPI と StarPU を用いて記述した時（ハードコーディング）のソースコードの行数を示した．ハンドコーディングでは StarPU で GPU を扱うときに CUDA コンパイラによってカーネル関数をコンパイルする必要が有るため，GPU に関係する部分のみファイルを切り分け，分割コンパイルを行う．そのため，ハンドコーディングの行数は 2 つのファイルの合計の値になっている．

表 2 N 体問題のプログラム行

逐次	XMP-dev/StarPU	MPI+StarPU
60 行	85 行	250 行

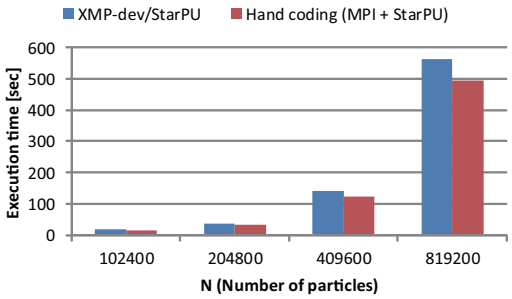

図 4 XMP-dev/StarPU とハンドコーディングの実行時間

表 2 より，XMP-dev/StarPU を用いることで，逐次のソースコードに 25 行追加するだけで，マルチノード上で GPU と CPU の協調計算を行うことができる様になった．これは，ハンドコーディングに比べると非常に少ない手間でプログラミングができることを示している．これによって，XMP-dev/StarPU の GPU/CPU 協調計算プログラミングのコードの生産性が高いことが示される．

性能面に関して，**図 4** に XMP-dev/StarPU が生成したコードと実際に MPI と StarPU を用いてハンドコーディングしたプログラムの実行時間を比較した．これより，若干 XMP-dev/StarPU のほうがハンドコーディングしたプログラムより遅いという結果になった．この差は，配列参照のインデックス計算が大きく影響すると推察される．XMP-dev では，常に Local array を参照させるために Global index から Local index へのインデックスを計算する．文献 [4] でも述べられているようにこのインデックス計算のオーバヘッドが確認されており，XMP-dev のランタイムの改良が必要と考えられる．これについては，本研究の範疇でないため本稿での言及はしないが，オーバヘッドの削減は今後の課題とする．

4.2 性能評価

4.2.1 N 体問題

図 5 に XMP-dev/CUDA に対する XMP-dev/StarPU の相対性能を示した．つまり，GPU のみを計算に利用した時に対する GPU/CPU 協調計算の性能である．縦軸は

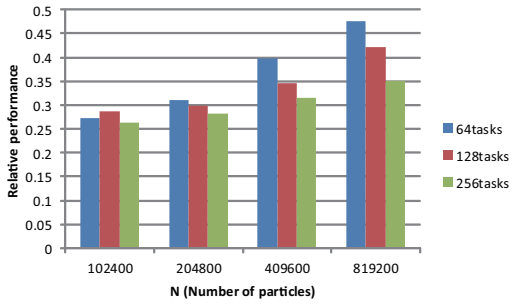

図 5 XMP-de/CUDA に対する XMP-dev/StarPU の性能：N 体問題

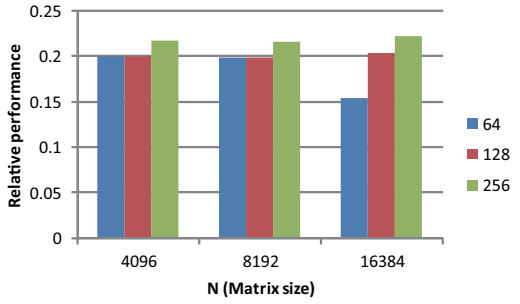

図 6 XMP-de/CUDA に対する XMP-dev/StarPU の性能：行列積

XMP-dev に対する性能，横軸は質点数 N，色違いの 3 本のグラフはそれぞれ Replicated array を何個のタスクに分割したかを示している．図 5 より，XMP-dev/StarPU は XMP-dev に対して最大でも 0.45 倍程度の性能しか出ていないことがわかる．この原因に関しては，タスク数とスケジューリングの自由度が関係していると考えられる．

4.2.2 行列積

行列積のプログラムは非常に単純で，行列 A と B の積を C に代入する．行列積を並列化するとき，行列 A と C を行方向に一次元分割をする．行列 B は列方向から参照されるため，各ノードで全要素が必要になる．そのため，XMP-dev の full shadow 機能を用いて計算ノード及び GPU 間の分散データの同期を取る．また，メモリアクセスが常に連続して起きるように計算部分である 3 重ループのうち最内の k ループと中間の j ループを入れ替えている．

図 6 に XMP-dev/CUDA に対する XMP-dev/StarPU の相対性能を示す．図 6 より，最大で XMP-dev/CUDA に対して 0.2 倍程度の性能しか出ておらず，N 体問題と同じような結果になった．

以上, 2 つのコード例において, 単純に XMP-dev/StarPU での実行を評価した結果，いずれも XMP-dev/CUDA，すなわち単純に GPU のみを使ったオフローディングを行った方が性能が高いという結果になった．次の章では，この原因について解析し，XMP-dev/StarPU で性能を向上させるための方策とその評価を行う．

5. 負荷バランス調整に基づく性能改善

4.2 節での性能評価では，XMP-dev/StarPU は XMP-dev/CUDA に対して，半分以下の性能しか得られなかった．この原因としてタスクサイズとスケジューリングの自由度が関係していると推察される．

5.1 性能低下の分析

StarPU では，データプールに登録した配列を細かい単位に分割する．これを StarPU ではタスクと呼んでおり，スケジューラが GPU や CPU へスケジューリングを行う．しかし，GPU と CPU が混在するヘテロジニアスな環境では先ほど述べたタスクサイズとスケジューリングの自由度の問題が出てくる．CPU コアあたりの演算性能はたかだか十数 GFLOPS であるのに対して，GPU のそれは，NVIDIA の Tesla M2090 において倍精度浮動小数点数演算性能で 665GFLOPS に達し，Fermi アーキテクチャの次の Kepler アーキテクチャ（K20，K20X）においては 1TFLOPS を超えるピーク性能が実現されている．このように，理論ピーク性能比ではあるが，演算性能に数十倍近くの差があるにも関わらず，タスクに割り当てる問題サイズを同じにしてしまうと 1 タスクあたりの実行時間が，GPU と CPU で大きく変わってくるのは明らかである．StarPU では大量のタスクを生成し，それを複数の計算リソースにダイナミックにスケジューリングすることによって，演算性能が高い計算リソースにタスクが多く割り振られることを期待している．非常に大規模な問題では，先行研究 [10] でも述べているようにスケジューリングがうまく動作するが，小規模・中規模であると難しくなる．問題サイズを固定したままタスクサイズを大きくするとタスクの個数が減る．そのため，スケジューリングの自由度は小さくなり，うまく負荷バランスを取ることができない．一方タスクサイズを小さくするとタスクの個数が多くなり，スケジューリングの自由度が上がる．しかし，GPU では小規模のホスト-デバイス間の通信が多発してしまい，その結果，GPU の演算性能が生かしきれなくなってしまう．

5.2 XMP-dev/StarPU の改良

5.1 節での考察により，タスクサイズを CPU と GPU のそれぞれに適したサイズにすることを考える．CPU では小さなタスクサイズ，GPU ではより大きなタスクサイズを割り当てることで GPU の性能十分に引き出しつつ，負荷バランスをとることが可能になると期待できる．現在の StarPU の仕様では，異なるサイズを持つタスクに分割することができない．そのため本実装では，タスクの数だけ data_handle を用意し XMP-dev/StarPU のランタイム内でタスクサイズを計算し，登録することでこれを実現し

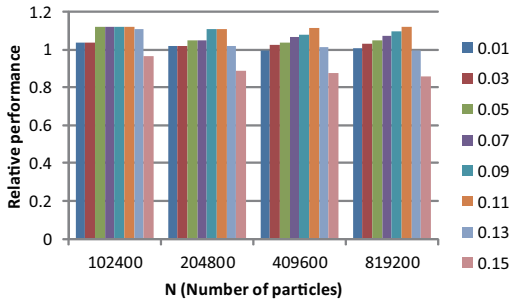

図 7 N 体問題における負荷の均等化

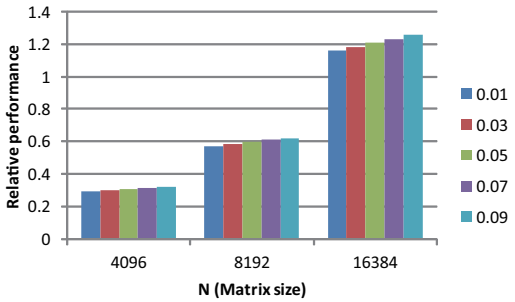

図 9 行列積における負荷の均等化

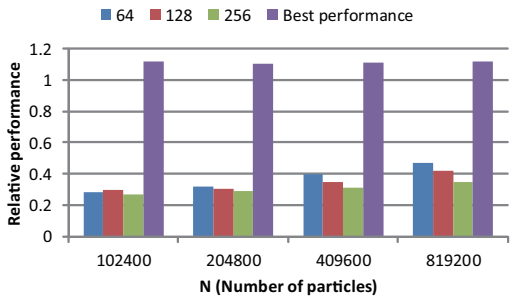

図 8 N 体問題：負荷バランス考慮による性能向上の比較

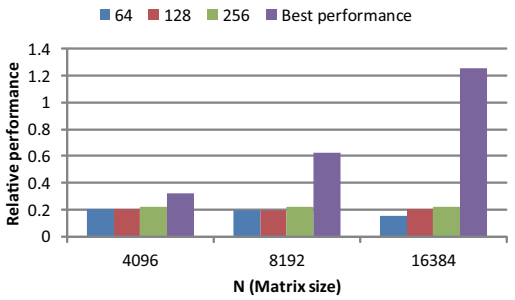

図 10 行列積：負荷バランス考慮による性能向上の比較

ている．文献 [11] では，タスクのサイズを GPU と CPU で変更することで性能を向上させている．そこで，本稿では，各ノードに存在する Replicated array 配列全体の何%を CPU での演算にわりあてるかという割合を「CPU Weight」と定義する．また，負荷バランスについて議論するために，GPU・CPU の分割数は 1 つと 12 で固定，つまり，1TIMESTEP ごとに GPU にも CPU core にも 1 つずつしか割り当てられないように固定化して調節する．

まず，N 体問題の負荷バランスについて考察をする．4.2.1 に示したとおり，XMP-dev/StarPU の性能は GPU のみを計算に用いる XMP-dev/CUDA の高々半分程度しか出ていない．これは，等しいタスクサイズのタスクをそれぞれのリソースに割り当て，かつ十分な数のタスクがないため，結局 GPU の実行時間中に GPU に割り当てられるタスクが枯渇して GPU が空転するという，不均一な負荷バランスが原因であると考えられる．これに加え，問題サイズ N が小さい時には分割した際にタスクサイズが小さすぎ，GPU で十分性能が引き出すことができていない．問題サイズが大きくなるとタスクサイズも比例して大きくなる．これによって，GPU の性能を十分に出せているため性能が向上すると考えられる．問題サイズが大きくなっていくに連れて，タスクサイズが GPU の演算性能をフルに引き出すサイズなるような十分なタスク数を設定できるようになればこの枠組のままで GPU/CPU のハイブリッドプログラムが可能になると推察される．しかし，問題サイズが十分に大きいことが条件であり，図 5 の問題サイズではうまくいかはわかっていない．

そこで，CPU Weight を用いて負荷バランスの調整を行う．**図 7** に CPU Weight を 0.01 から 0.15 まで 0.02 刻みで変更した時の XMP-dev/CUDA に対する相対性能を示す．縦軸は XMP-dev/CUDA に対する相対性能であり，1.0 を超えると XMP-dev/CUDA に対して速度向上が得られていることを示している．横軸は問題サイズである粒子数で CPU Weight を変えた場合の性能を示している．図 7 より，各問題サイズにおいて CPU Weight が 0.11 の時，XMP-dev/CUDA に対して約 1.1 倍の速度向上を得ることができることがわかる．**図 8** は，それぞれの問題サイズにおいて最適な性能が得られる CPU Weight を選んだ場合と，タスクサイズ調整をおこなっていない 4.2 節の結果との比較を示している．横軸は問題サイズ，縦軸は図 7 と同様，XMP-dev/CUDA（GPU のみを計算に使用）の場合に対する相対性能である．このように，計算リソースに最適なタスクサイズの割り当てを行うことで小・中規模の問題サイズにおいて GPU/CPU ハイブリッドによって GPU のみを計算に使った場合に対して，十分ではないものの一定の高速化が実現できることわかった．現在のタスクサイズ制御機能は予備的なもので，コンパイラ内で CPU Weight 及びタスク分割数を直接指定しており，また実用的な実装とはなっていない．CPU Weight は手動で変更し最適な値を見つけていたが，GPU と CPU の実行時間を測定しコンパイラがその値から適した値を与える枠組みを提供することが望ましい．あるいは，動的プロファイリングによる調節ではなく，プログラマが何らかのツールを用いて環境変数やプログラム中の明示的な関数呼び出しによって任意の

CPU Weight を指定可能にすることが望ましいと考えられる．これに関しては今後の課題である．

同様に，行列積の負荷バランスについて考察をする．**図 9** より，行列サイズ 16K（$16 * 1024 = 16384$）の時に CPU Weight を 0.09 とすることにより，同サイズの XMP-dev/CUDA に対して約 1.2 倍の高速化を得ることができた．N 体問題と同様に**図 10** にタスクサイズ調整前と後の XMP-dev/CUDA に対する相対性能を示す．しかし，N 体問題の場合とは違い，問題サイズによっては XMP-dev/CUDA の性能に達する事ができない場合もある．これは問題サイズが小さすぎて StarPU のデータ登録やタスクの生成等のオーバヘッドの割合が大きくなってしまったことが原因だと考えられる．N 体問題では N 個の粒子に対して $O(N^2)$ の計算量があり，GPU への入出力に対する演算性能がある程度確保されるが，行列積では $N * N$ 行列の場合，$O(N^2)$ の入出力量に対して $O(N^3)$ の計算量となるため，入出力オーバヘッドや GPU カーネルの起動等のオーバヘッドを隠しきる事ができない可能性が高いと考えられる．このようなオーバヘッドを無視できるくらいの行列サイズで評価を行うべきであるが，現在の XMP-dev/CUDA の仕様では，行列を一次元方向の分割に分割すると，2node2GPU の構成において，必要なメモリサイズは $((N/2)^2(A) + N^2(B) + (N/2)^2(C)) * 8byte$ となり $N = 19k$ では約 6.1GB 必要になり GPU メモリに載り切らない．現在の XMP-dev の仕様では replicate 指示文や replicate_sync 指示文はホストに宣言された配列と同じサイズのデータしか確保・転送できない．一方，XMP-dev/StarPU ではこのような問題が起きない．その主な理由は，StarPU はタスク単位で計算リソースに計算を割り当てているため，メモリ等のリソース量に対する制約が緩いことがある．

以上のように，行列積問題では，XMP-dev/StarPU の特性を活かせる問題サイズが，比較対象であるオリジナルの XMP-dev/CUDA 側の制約で評価できず，十分な比較が行えなかった．現在の XMP-dev/StarPU は行方向の 1 次元分割のみを想定しているため，一般的に行われている二次元のタイル分割ができない．そのため，二次元分割への対応は様々な種類の問題及び問題サイズを扱う上で重要であり，今後の研究で実現していきたいと考えている．

6. 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 [12] や HMPP Workbench [13] 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．HMPP Workbench 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 [14] 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 [15] 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，XMP-dev/StarPU のフレームワークでも GPU と CPU の協調計算が可能だと推察される．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．

7. 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 コンパイラを提案し，コンパイラとランタイム設計と実装について述べた．そして，GPU と CPU の大きな演算性能の差に着目し，適切な負荷分散と分割タスクサイズの調節を行うことにより，N 体問題や行列積に XMP-dev/StarPU を適用し XMP-dev/CUDA に対して約 1.1〜1.2 倍ほどの速度向上を得ることができた．また，XMP-dev/StarPU の指示文を用いることで MPI+StarPU のプログラムコードよりはるかに少ない量で記述できることを示した．その一方，現在の枠組みでの GPU/CPU 協調計算ではアプリケーションの依存性が強ことと，一定サイズ上の問題サイズでなければ効果が得られないことも確認できた．

本稿では GPU と CPU で計算させるタスクサイズを変えるために CPU Weight という値を用いてそれを実現している．現在の実装では実行中にこの値を変えることができない．そのため，プログラム中でパラメータを変更できるように拡張し，プログラマがプロファイルから得られた結果から変更できるようにしたいと考えている．また，行列積の考察でも述べたが，現在一次元分割のみをサポートしているため，ノード数などをスケールするにあたってこのままでは性能もスケールするのは難しいと考えられる．そのため，二次元分割の実装を進めることでこの問題を解決したいと考えている．

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．また，本研究では，筑波大学計算科学研究センター平成 24 年度学祭共同利用プログラム課題「核融合シミュレーションの GPU 化と性能評価」により，大規模 GPU クラスタ HA-PACS を利用した．同センター並びに関係者各位に謝意を表する．

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, 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, Vol. 5, No. 2, pp. 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)*, pp. 2394 –2403, May 2012.
[6] StarPU. http://runtime.bordeaux.inria.fr/StarPU/.
[7] 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. 情報処理学会論文誌, Vol. 3, No. 3, pp. 153–165, Sep 2010.
[8] 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*, Vol. 5415, pp. 174–183. 2009.
[9] 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.
[10] 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, OlivierAumage. GPU クラスタにおける GPU/CPU ハイブリッド・プログラミング環境. 情報処理学会研究報告, No. 9, Jul 2012.
[11] F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pp. 365–376, 2012.
[12] PGI Accelerator Compiler. http://www.softek.co.jp/SPG/Pgi/Accel/index.html.
[13] HMPP Workbench. http://www.caps-entreprise.com/hmpp.html.
[14] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 User Guide. http://icl.cs.utk.edu/magma/.
[15] 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*, Vol. 2. September 2010.