Vol.2012-HPC-135 No.9
2012/8/1

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

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フレームワークが直交しており，複雑になってしまうことが多い．本研究では，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 の一実装として，GPU と CPU によるハイブリッドワークシェアリングを容易に行うことができる XMP-dev/StarPU を提案し，プロトタイプ実装を行う．XMP-dev は，XMP が本来提供している分散メモリノードへのデータと処理の分割・通信の機能に加え，各ノードでの処理の一部を GPU にオフローディングをすることが可能である．しかし，現在の実行モデルでは GPU にオフロードされた部分はすべて GPU により実行され，CPU との協調計算やワークシェアリングを行うことができない．本研究では，StarPU をバックエンドスケジューラとして用い，計算をタスクという単位で GPU や CPU へスケジューリングをすることで，GPU/CPU のワークシェアリングを可能とする．本稿では，現在開発中の XMP-dev/StarPU のプロトタイプコンパイラと同等の動作をするハンドコンパイルしたコードを用いて重力 N 体問題について評価を行う．結果として，GPU/CPU ワークシェアリングは機能しているが性能向上は十分ではなく，大きな要因は GPU と CPU の性能差に対応する十分な問題サイズを与えることが難しいこと，また，これを改善するために何らかの負荷バランス機能が必要であることがわかった．

1. はじめに

近年, 高い演算性能, 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 で行うプログラミングが容易になったことで，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 や OpenMP など直交するフレームワークを組み合わせているため，複雑なプログラミングが必要である．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 のコア数増加や，SIMD 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．

我々はこれまで，分散メモリシステムを対象とした，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 [4]（以降「XMP-dev」と略す）があるが，これ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．我々は，INRIA で開発されている StarPU [5] 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 へのタスクのスケジューリングが可能であり，これを XMP-dev に組み込むことで言語レベルでの GPU/CPU

[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

協調計算を行うことが可能になる．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 を組み合わせたプロトタイプコンパイラの設計と実装について述べる．しかし，コンパイラは基本的に完成しているが，性能改善のために同等のプログラムによる予備評価を行う．また，XMP-dev と StarPU を組み合わせるに当たっての問題や，ヘテロジニアスな環境でのハイブリッドプログラムについて検討を行う．

2. XcalableMP と XMP-dev の概要

2.1 XcalableMP

XMP に関しては文献 [6] 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．

また，XMP のデータアクセスはローカルのメモリを参照するが，隣接領域に依存する偏微分方程式の差分・陽解法や，ノード間にまたがったデータすべてにアクセスする N 体問題のようなアプリケーションでは，他ノードに存在

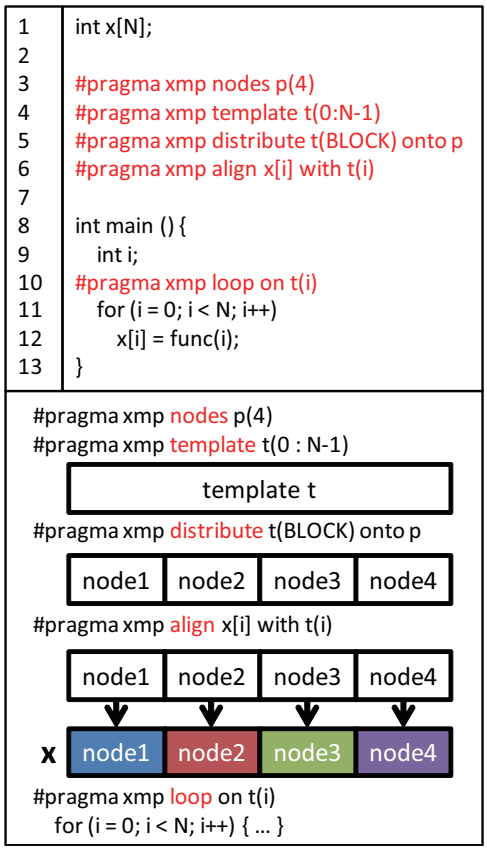

図 1 XMP のサンプルコード

するデータにアクセスする必要がある．そこで XMP では典型的なノード間通信を簡潔に記述するために，shadow 指示文と reflect 指示文を提供している．XMP では，他ノードと重複してデータを持つ領域を「shadow」と定義している．shadow 領域は，他ノードに存在するデータを前もってローカルメモリに確保しておくことで，ローカルメモリ内だけで参照することを可能にする．各ノードでの計算結果を shadow 領域に反映（同期）させる操作は reflect 指示文を用いることで実行される．ユーザは適当なタイミングーに reflect 指示文を挿入し，正しい同期を保証する．reflect 指示文に対応する箇所ではノード間の peer-to-peer 通信が適宜実施される．また，shadow 領域を配列全体に適用することで全てノードが同じ配列を持つことができる．これは「full shadow」と定義されている．full shadow の reflect は MPI_Allgather と同等である．

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，分散メモリ環境におけるノード

間のデータ及び処理の分割という従来の XMP の機能に加え，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．アクセラレータ間のデータ交換はホストメモリを介して行う．そのため，ユーザが XMP-dev の指示文でホスト-デバイス間の通信を指示する必要がある．

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．

```
#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．デバイスメモリサイズを超えた確保を XMP-dev のランタイムはチェックできないため，ユーザがそれを超えないように設定する必要がある．

```
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．

3. StarPU の概要

StarPU に関しては文献 [7] [8] 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，

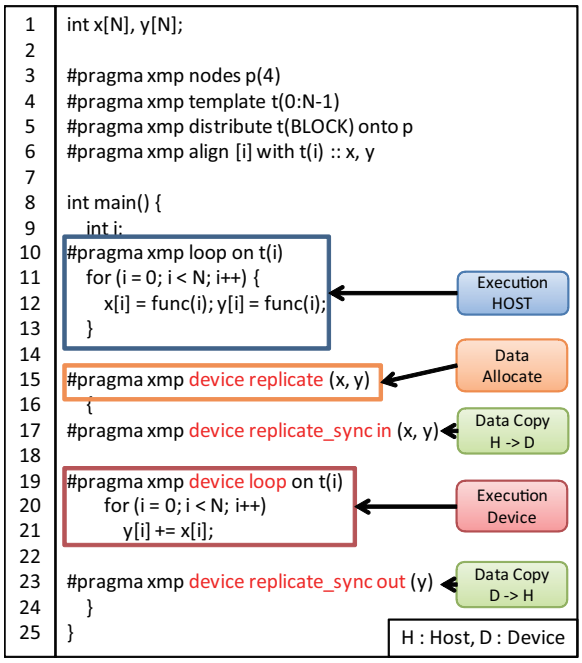

図 2 XMP-dev のサンプルコード

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．

3.1 Codelet

StarPU では，タスクを制御するために「codelet」と呼ばれる構造体を使う．これをタスク生成時にポインタ渡しすることで，どの計算リソースで実行するか，どの関数を実行するか，必要な配列は何かなどの情報を登録する．計算する関数が複数存在する場合，関数ごとに codelet を生成する．codelet の例を以下に示す．この例は，CPU と GPU で並列して実行するようになっている．

```
starpu_codelet cl = {
    .where = STARPU_CPU|STARPU_CUDA,
    .cpu_func = cpu_fnction,
    .cuda_func = cuda_function,
    .nbuffers = 10
};
```

3.2 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 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．

また，データプールに登録されているデータを MPI などの通信を使って他ノードに送りたいことがある．このようにアプリケーションが直接データプールのデータにアクセスしたい時には，starpu_data_acquire 関数でアプリケーションにデータ管理を移譲し，starpu_data_release 関数で管理を StarPU に戻すことができる．

3.3 タスクの実行

図 3 にシングルノードにおける StarPU の動作を示す．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，starpu_task_submit 関数を使って 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．

3.4 タスクサイズとロードバランス

3.3 で触れたように，StarPU は GPU や CPU へタスクのスケジューリングを行う．しかし，ヘテロジニアスな環境では問題が出てくる．それは，チャンクサイズとスケジューリングの自由度である．CPU の演算性能はたかだか百数十 GFLOPS である．一方 GPU は，NVIDIA の Tesla M2090

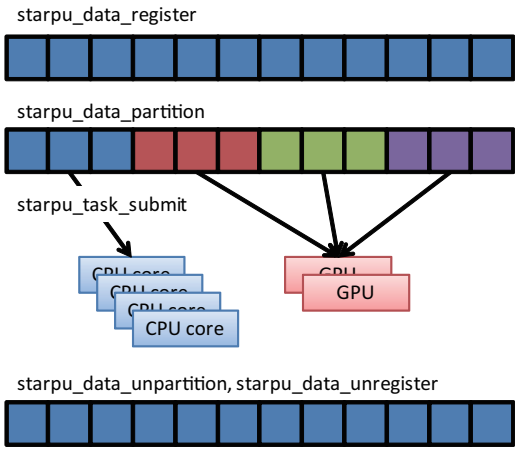

図 3 シングルノード上での StarPU の動作

において倍精度浮動小数点数演算性能は 665GFLOPS に達し，Kepler2 においては数 TFLOPS に達すると言われている．このように演算性能が 5〜6 倍の差があるにも関わらず，チャンクサイズを同じにしてしまうと CPU がボトルネックになってしまう．

そこで，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，この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．チャン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ってしまい，うまく負荷バランスを取ることができなくなる．一方チャン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，小規模のホスト-デバイス間の通信が多発してしまい，オーバーヘッドが大きくなりすぎてしまう．そのため，適切なチャンクサイズを設定する必要がある．

4. XMP-dev/StarPU の概要

4.1 XMP-dev/StarPU の提案

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 [9] 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．

4.2 XMP-dev/StarPU の実装

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 4** に XMP-dev/StarPU の実行モデルの概要を示す．従来の XMP-dev（今後 [4] の実装を区別するために本稿では「XMP-dev/CUDA」と定義する）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．

ここで，図 4 の Replicate array について説明する．Local array を直接複数のチャンクに分解し，それをタスクとすることは可能であるが，Local array はノード間のデータ交換などに使われている．そのため，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 を分割した時と同様な動作が期待できる．しかし，Replicate array は本来冗長なデータであるため今後は Local array から直接タスクの作成が出来るように改良を加えていく予定である．

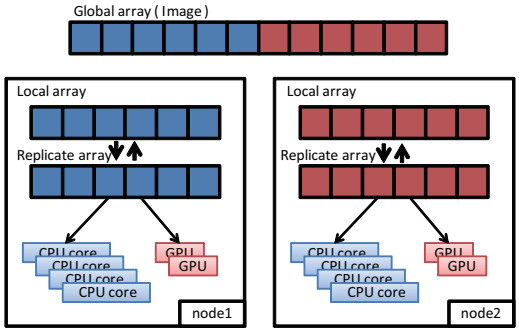

図 4 XMP-dev/StarPU の実行モデルの概要

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，指定されたチャンク数に分割される．

#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．

表 1　測定環境（HA-PACS）

CPU	Intel Xeon E5-2670 * 2 (16cores)
Memory	DDR3 128GB
GPU	NVIDIA Tesla M2090 * 4
GPU Memory	6GB/GPU
CUDA Toolkit	4.1
MPI	OpenMPI 1.4.3
Interconnection	InfiniBand x4 QDR
# of node	4

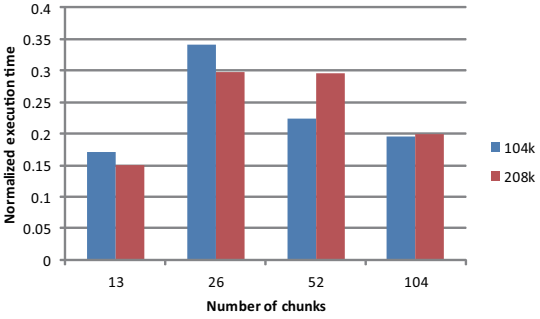

図 5　XMP-dev/CUDA に対する性能の評価．縦軸は XMP-dev/CUDA に対するハンドコンパイルしたコードの相対時間，横軸はチャンク数を表す．

5. 予備評価

XMP-dev/StarPU コンパイラは基本的に完成しているが，性能を向上させるために予備評価が必要である．ここでは，現在のプロトタイプコンパイラが生成するコードと同等のコードをハンドコンパイルによって用意し，これを評価，あるいは一部を改変することで性能への影響を検証する．

5.1 評価と問題

評価に用いた環境は筑波大学計算科学研究センターの GPU クラスタ HA-PACS [10] である．ノードの構成を**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 4 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．そのため，本稿における評価では CPU core 数は 12，GPU 数は 1 に固定する．評価に用いるアプリケーションは重力計算の N 体問題（直接法）である．アプリケーション中に 2 重ループ（i-粒子：最外ループ，j-粒子：最内ループ）がある．並列化する際に i-粒子を分割し，j-粒子はすべて走査する．

はじめに，ハンドコンパイルしたコードが XMP-dev/CUDA に対して速度向上を得ることができるかを評価する．**図 5** に粒子数 104k, 208k において XMP-dev/CUDA

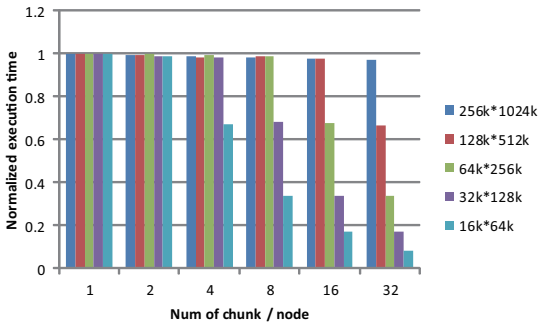

図 6　GPU のチャンクサイズの評価．問題サイズは i-粒子＊ j-粒子，縦軸はチャンク数 1 に対する相対時間，横軸はノード内でのチャンク数を表す．

に対する相対時間を示す．これより，最大でも 35%程度しか性能が出ていないことがわかる．特にチャンク数によって性能の上下が激しい（本稿では，チャンクサイズを割り当てる i-粒子の個数で表すことにする）．これは 3.4 節で触れたように，チャンクサイズとスケジューリングの自由度が影響していると考えられる．チャンク数が 13 の時には計算リソースと同数であるため，各計算リソースに 1 つしかタスクがスケジュールされない．同一のチャンクサイズを持つタスクを GPU と CPU で処理をすると，演算性能に差があるため CPU がボトルネックとなり，GPU が空転してしまい，全体の性能低下に繋がってしまうと考えられる．一方，チャンク数が 104 の時には，1 タスクの実行時間が少なくなり，スケジューリングの自由度は上がるが，チャンクサイズは小さくなってしまう．

GPU は文献 [11] より十分な演算量を与えなければ性能を引き出すことができないことがわかっている．これより，常に GPU が性能を出し続けられるようなチャンクサイズを適切に設定する必要がある．**図 6** に GPU を用いた時の 1 反復あたりの実行時間を相対値で示す．問題サイズを 4 ノードで分割しているため，i-粒子数は 1/4 になっている．これより，性能が落ちている直前のチャンクサイズは $32k/4 = 8k$ や $128k/32 = 8k$ のように 8k であることがわかる．よって，チャンクサイズは $8k(= 8 * 1024 = 8192)$ が最適であると考える．一方 CPU の演算時間は，**表 2** に示すようにチャンクサイズが減少していっても実行時間に変化がないことがわかる．これより，チャンクサイズを 8k に固定しても問題ないと考える．

表 2　問題サイズ $N = 48k$ の CPU の演算時間 [sec]

チャンク数	12	24	48	96
チャンクサイズ	8k	4k	2k	1k
実行時間	358.04	358.0	358.15	358.0

次に，GPU と CPU の演算性能を比較する．比較に用いる問題サイズは CPU のコア数 $*8k(= 12 * 8k = 96k)$ とし，これは各リソースで十分な性能が出る最小の演算量で

ある．比較は CPU12 コアと GPU1 台である．結果を**表 3** に示す．CPU12 コアが 1 回計算する間に GPU は約 7 回同じ量を計算できることがわかる．

表 3 問題サイズ $N = 96$k の演算時間 [sec]

	CPU	GPU	Ratio
実行時間	358.16	49.46	7.24

これより，問題サイズを GPU と CPU の比を考慮し，設定することで GPU と CPU の協調計算が上手くいくことが期待できる．これを検証するために問題サイズ $N = 3\text{m}(= (96\text{k}*1 + 96\text{k}*7)*4 = 96\text{k}*8*4 = 3*1\text{k}*1\text{k})$ とする．そして，チャンクサイズが 8k になるようにチャンク数を 96 とする．ハンドコンパイルしたコードと GPU のみを使った時の実行時間を**表 4** に示す．結果は GPU のみを使った場合と比較して GPU/CPU ハイブリッドプログラムは約 9.6%の性能向上が得られた．しかし，このように一定のチャンクサイズと計算リソースの演算性能を用いて問題サイズを決定すると，問題サイズが大きくなりすぎてしまう．たとえば，HA-PACS のように GPU が 4 枚搭載されているクラスタでは $N = (96\text{k}*1 + 96\text{k}*7*4)*node = 2784\text{k}*node$ と大規模になりすぎてしまう．そのため，小・中規模の問題を計算するにはこの枠組ではうまくいかないことがわかる．

表 4 問題サイズ $N = 3$m の演算時間 [sec]

Hybrid	GPU	Speed-up
2881.44	3158.50	1.096

5.2 チャンクサイズの調整

XMP-dev/StarPU では starpu_data_partition のように登録されたデータを等分割してタスクに渡している．そのため，チャンク数によってチャンクサイズが最適にならないことがある．そして，GPU・CPU のどちらかに負荷が偏ってしまい効率良くハイブリッドプログラムを行うことができない．

そこで，計算リソースにあったチャンクサイズをそれぞれ設定することで，問題サイズがある程度の大きさであればハイブリッドプログラムにおいて速度向上が期待できる．実装は，GPU/CPU が計算する領域をあらかじめ設定し，それぞれをデバイスに合わせたチャンクサイズになるようにチャンク数を設定する．本稿では，GPU はチャンクサイズ 8k を下回らないように，CPU はコア数と同じになるように，それぞれの計算リソースに応じたチャンクサイズを設定し，それに応じたチャンク数でタスクを用意する．ここで，計算領域全体の中で CPU に計算させる領域の割合を「CPU Weight」と定義する．GPU の領域は残りの部分である．最適な CPU Weight の設定は表 3 の ratio の逆数（$= 1/7.24 = 0.138 \fallingdotseq 0.13$）とすればよい．**図 7** に CPU Weight を 0.13 から前後に 0.02 変化させた時の XMP-dev/CUDA に対する相対時間を示す．これより，CPU Weight が 0.13 の時にすべての問題サイズにおいて 4〜6%の速度向上を得ることができた．このように，計算リソースに最適な問題分割を行うことで小・中規模の問題サイズにおいても GPU/CPU ハイブリッドによって高速になる事がわかる．

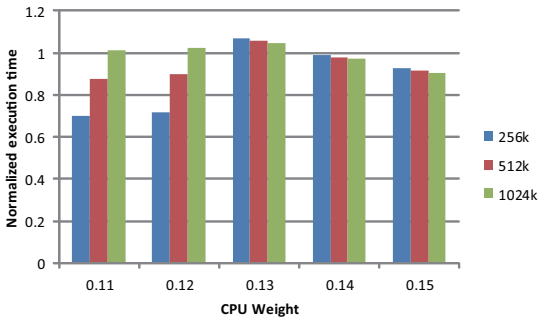

図 7 負荷バランスを考慮による速度向上．縦軸は XMP-dev/CUDA の実行時間に対して正規化したもの，横軸は CPU Weight を表す．

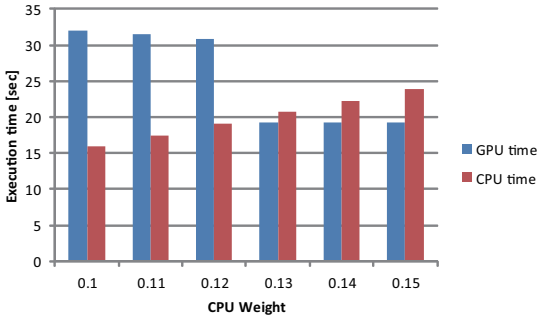

図 8 $N = 256$k における各デバイスの実行時間．縦軸は問題分割後の各リソースの計算時間，横軸は CPU Weight を表す．

また，図 7 の CPU Weigh が 0.11，0.12 で大きく性能が低下していることがわかる．この原因は，**図 8** を見るとわかる．CPU の演算時間は，割り当てられた問題サイズに対して線形に増加している．しかし，GPU は CPU Weight 0.11，0.12 では急激に実行時間が増加している．これは GPU で同時に動くスレッド数が関係していると考えられる．例えば，$N = 256$k，CPU Weight 0.12 の時，GPU に割り当てられる要素数は $N * (1 - 0.12) = 57672$ となる．また，GPU ではチャンクサイズが 8k に近くなるようにチャンク数を設定する．5.1 節よりチャンク数 7 であることがわかるから今回のチャンクサイズは $57672/7 = 8238$（端数は最後のチャンクに割り当てられる）となる．これは 8k(= 8192) を超えている．GPU ではブロックあたりのスレッド数を 256 に設定しているため，8k であれば $8\text{k}/256 = 32$ となる．Tesla M2090 では SM（Streaming

Processor）が 16 基あり，ちょうど 2 回ですべてのスレッドの実行が終了する．しかし，8k を超えるスレッド数では，最後のスレッドが実行を終了するまで 3 回かかる．つまり，8k を少しだけ超えてしまうと全体の実行時間が増加してしまう．これが原因で図 7 では大きな速度低下につながったと考えられる．

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，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．また，これらは OpenACC [14] の規格に準拠している．しかし，現段階ではインタフェースは同じであっても実装は各コンパイラのものが利用されている．

大島らの研究 [15] では，GPU と CPU の協調計算によって GPU のみを演算に使った場合よりも速度向上を得られている．文献中でも論じているように，最適な問題分割割合は CPU と GPU の演算性能の差によって左右される．我々の実装ではこの割合を最適に決定できるようにしなければならないことがわかった．

Agullo らの研究 [16] によると，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．本研究においても，StarPU による GPU/CPU ハイブリッド計算で XMP-dev/CUDA に対し，速度向上が得られることが期待できる．

7. まとめと今後の課題

本稿では，GPU と CPU によるハイブリッドプログラミングを行えるフレームワークとして，XMP-dev と StarPU を組み合わせた XMP-dev/StarPU を提案し，プロトタイプコンパイラ及びランタイムシステムの実装・評価を行った．これによって，単純にチャンクサイズ（タスクサイズ）を統一してしまうと，大規模な問題サイズでなければうまくスケジューリングすることができない事がわかった．そこで，各計算リソースに応じてチャンクサイズを変更することで負荷のバランスを取る手法を提案し，これによって小・中規模問題において GPU/CPU のワークシェアリングがうまくいく事を確認した．

今後の課題として，各リソースに応じたチャンクサイズを設定できるようにコンパイラやランタイムを実装し，評価を行う．その上で，負荷分散の指標をどのように言語に盛り込むかを検討する必要がある．また，本稿の評価では GPU を 1 枚のみ使って測定を行なっていたが，これを HA-PACS の上限である 4 まで使った時に適切にチャンクサイズを指定できるか検証を行う予定である．

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．また，本研究の遂行に当たり HA-PACS を利用させて頂いた筑波大学計算科学研究センターに謝意を表する．

参考文献

[1] CUDA C Programming Guide. http://developer.nvidia.com/nvidia-gpu-computing-documentation.
[2] OpenCL. 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コンピューティングシステム（ACS）, Mar 2012.
[5] StarPU. http://runtime.bordeaux.inria.fr/StarPU/.
[6] 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コンピューティングシステム（ACS）, 2010.
[7] C. Augonnet and R. Namyst. A Unified Runtime System for Heterogeneous Multi-core Architectures. In *Euro-Par 2008 Workshops - Parallel Processing*, 2009.
[8] C. Augonnet, S. Thibault, and R. Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. In *Concurrency Computat.: Pract. Exper.*, Mar 2010.
[9] T. Nomizua, D. Takahashi, J. Lee, T. Boku, and M. Sato. Implementation of XcalableMP Device Acceleration Extention with OpenCL. In *Multicore and GPU Programming Models, Languages and Compilers Workshop (Co-located with IPDPS 2012)*, May 2012.
[10] HA-PACS プロジェクト. http://www.ccs.tsukuba.ac.jp/CCS/research/project/ha-pacs.
[11] 成瀬彰, 住元真司, 久門耕一. GPGPU 上での流体アプリケーションの高速化手法 : 1GPU で姫野ベンチマーク 60GFLOPS 超. 情報処理学会研究報告, Oct 2008.
[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] OpenACC. http://www.openacc-standard.org/.
[15] 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, 2006.
[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*. Sep 2010.