

# GPU クラスタ向け並列言語 XMP-dev における GPU/CPU 動的負荷分散機能

小田嶋 哲哉<sup>1,a)</sup> 朴 泰祐<sup>1,2</sup> 埜 敏博<sup>2</sup> 児玉 祐悦<sup>1,2</sup> Raymond Namyst<sup>3</sup> Samuel Thibault<sup>3</sup>  
Olivier Aumage<sup>3</sup> 佐藤 三久<sup>1,2</sup>

**概要:** GPU クラスタにおける 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 コンパイラおよびランタイムシステムによるアプローチを行っている。これによって、高レベルのユーザ記述だけで、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。本論文では、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および API を実装し、評価を行った。N 体問題および行列積ベンチマークにおいて、GPU/CPU ハイブリッド計算を行った結果、GPU のみによる計算に対して最大で 1.4 倍の性能向上を得ることができた。

**キーワード:** GPU クラスタ, アクセラレータ, PGAS プログラミング, ハイブリッド計算, ワークシェアリング

## GPU/CPU Work Sharing Mechanism on XMP-dev, High-Level Parallel Programming Language for GPU cluster

TETSUYA ODAJIMA<sup>1,a)</sup> TAISUKE BOKU<sup>1,2</sup> TOSHIHIRO HANAWA<sup>2</sup> YUETSU KODAMA<sup>1,2</sup>  
RAYMOND NAMYST<sup>3</sup> SAMUEL THIBAUT<sup>3</sup> OLIVIER AUMAGE<sup>3</sup> MITSUHISA SATO<sup>1,2</sup>

**Abstract:** On the hybrid work sharing among GPUs and CPU cores on GPU cluster, it is an important issue to keep load balance among these resources. We have been developing a PGAS language named XMP-dev and its run-time system named XMP-dev/StarPU toward a solution for this problem. In our research so far, we achieved a certain level of performance improvement by additional CPU power to GPU-only computation based on optimized load balancing. In this paper, we implement a new feature, which can control the load balance dynamically during the application execution in our XMP-dev/StarPU compiler and run-time system. As a result, we achieved up to approximately 40% higher performance with GPU/CPU hybrid execution in N-body computation and Matrix Multiplication benchmarks.

**Keywords:** GPU Cluster, Accelerator, PGAS Language, Hybrid Computing, Work Sharing

### 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

<sup>1</sup> 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  
<sup>2</sup> 筑波大学 計算科学研究センター  
Center for Computational Sciences, University of Tsukuba  
<sup>3</sup> Bordeaux Sud-Ouest INRIA research center  
a) odajima@hpcs.cs.tsukuba.ac.jp

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 などのパラダイム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 に関数単位やループ単位の演算を丸ごと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、PC クラスタコンソーシアムの分科会において、PGAS (Partitioned Global Address Space) 並列プログラミング言語 XcalableMP [3] (以降「XMP」と略す) の開発が進められている。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。

我々は、これまでの研究 [6], [7] において、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・実装してきた。これによって、GPU と CPU によるワークシェアリング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、GPU と併用することにより高い性能を得ることができる。しかし、CPU の全体性能への寄与は部分的であり、アプリケーションによっては十分な性能を得ることができていなかった。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ている。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 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。

本論文では、この問題を解決するために XMP-dev/StarPU に新たにデバイス毎に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、問題の性質や、問題サイズによって変わる最適なタスクサイズを、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。

本論文で提供する手法は、PGAS モデルに基づく高水準言語において、GPU クラスタ上における GPU/CPU ハイブリッドプログラミングおよび両タイプの計算リソースを有効活用するワークシェアリングの枠組みを提供する。これにより、GPU プログラミングに精通していないユーザが、アプリケーションを容易に並列化し、このようなワークシェアリングによる性能向上を享受できる環境を提供することを可能とすることが目的である。

本論文の構成は以下のとおりである。2章では、XMP-dev および StarPU の特徴について述べる。3章では、XMP-dev/StarPU の従来実装やその問題点および XMP-dev/StarPU の新機能である動的負荷分散について述べる。4章では、XMP-dev/StarPU の動的負荷分散についての性能測定により、GPU/CPU ハイブリッドワークシェアリングについて考察を行う。5章では、関連する研究について述べ、6章でまとめと今後の課題とする。

## 2. XMP-dev と StarPU

### 2.1 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 PGAS 並列言語である。XMP-dev が扱うアクセラレータ (以降「デバイス」と呼ぶ) 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 PGAS モデルに基づく並列化が比較的容易に可能になる。CUDA や OpenCL を MPI と組み合わせて使うことなく、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、データの分散や並列実行主体への割り当てを記述する。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

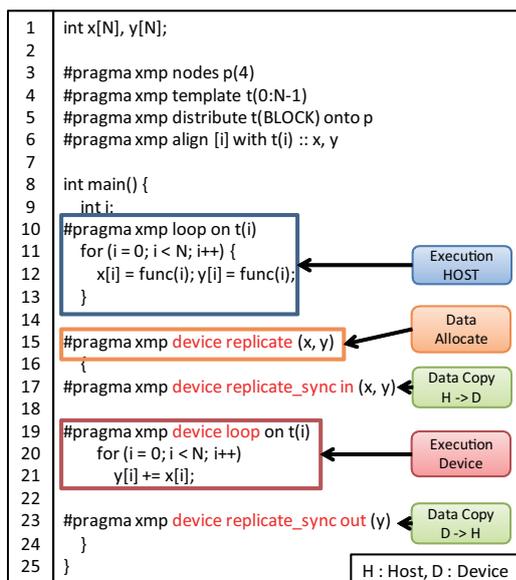


図 1 XMP-dev のサンプルコード

あり、すべて “#pragma xmp device” から始まる。これらについて、以下に記述する。

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 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 文をデバイス上でオフロード実行する。この for 文は、XMP-dev のコンパイラによって、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。GPU に代表されるデバイスでは、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。

## 2.2 StarPU の概要

StarPU に関しては文献 [9][10] に詳しいが、ここでは本論文を理解するための最小限の概略を示す。StarPU では、計算に必要なデータの集合と実行の単位を「タスク」と定義している。StarPU は、このタスクを GPU や CPU 等の様々な計算リソースに動的に割り当てたり、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。本論文では、マルチコア CPU および 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性に基づく実行順序制御やデータの再分散をするために、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。StarPU では、計算に必要なデータはタスクを実行するリソースの最も近いメモリ (CPU はメインメモリ、デバイスはデバイスメモリ) に存在するよう、計算が実行される前にデータの転送を行う。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。

StarPU は、多数のタスクを各計算リソー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し、動的にロードバランスをと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、計算リソース間の性能差が非常に大きい。これが原因で、タスク数が非常に多く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負荷分散は極めて難しくなる。

## 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 (MPI の rank0 相当) 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。

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 ハイブリッド環境におけるワークシェアリング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のため、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。さらに、PGASモデルに基づく分散メモリ環境での並列化が容易に実現できる。

### 3.2 従来実装の方針と問題点

我々は先行研究において、StarPUをXMP-devのタスクスケジューラ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、各タスクに分配され、各デバイスに割り当てられる。
- リソース間のデータの再分散については2.2節のとおりStarPUによって管理されるが、ノード間通信などを行う際には、明示的にLocal arrayとReplicated array間の同期を行う必要がある。

また、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Replicated arrayがホストのメモリ上に確保され、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

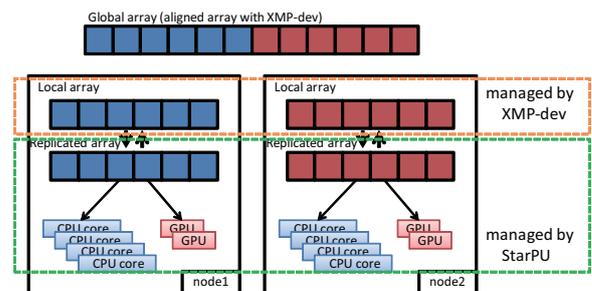


図2 XMP-dev/StarPUの実行モデルの概要

れもStarPUではタスク発行時に行われる。XMP-dev/StarPUではdevice replicate指示文で登録された配列のReplicated 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Replicated 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および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で実行する対象とすることによりGPUとCPUでfor文のワークシェアリングが可能になる。

先行研究[6],[7]の基本的実装では、XMP-dev/StarPUの性能はXMP-dev/CUDAに対して45%程度であり、結果としてGPUのみを利用した計算よりも性能が下がることとなった。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の性能差によって両者の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、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が空転してしまう。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う。この経験より、問題サイズが非常に大きく、多数のタスクに細分化される場合以外では、タスク数が少なすぎる(高速なGPUに十分な数のタスクが与えられず負荷分散がうまくいかない)、あるいはタスクサイズが小さすぎる(GPUの高速性が活かせない)、という理由により性能向上が見込めず、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。

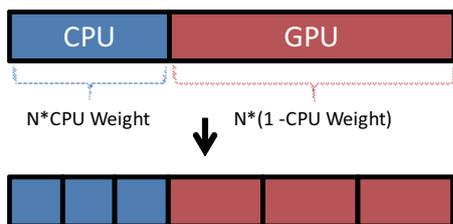


図 3 Replicated array の分割イメージ

### 3.3 動的負荷分散機能による性能向上

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 ワークシェアリングでは、GPU と CPU のタスク当たりの実行時間をできるだけ近くすることができれば、総タスク数が少ない場合であっても、効率的に計算リソースを使うことができる。また、GPU に割り当てるタスクサイズを大きくすることで、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。この両デバイスにおける最適なタスクサイズの差は、問題の性質・問題サイズによって大きく異なるため、従来の固定的な配分比では十分な最適化を行うことは難しい。

この問題を解決するために、CPU core と GPU がそれぞれ担当するタスクサイズの比を示す “CPU Weight” というユーザー制御可能なパラメータを設け、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。これによって、従来の XMP-dev/StarPU の枠組みの中で簡潔に動的負荷分散を記述することができることが期待できる。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 の計算領域である。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を示す 0 から 1 の範囲の実数で、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、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が割り当てられている。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。

しかし、最適な CPU Weight を静的に設定することは非

常に困難である。これをコンパイラやランタイム内部で自動的に決定し、性能向上を得ることは難しいと考えられる。最大の理由は、StarPU におけるタスクへのデータの割り当ては、基本的にデータプールに管理されている配列を単位としているため、タスクの実際の計算量はその配列サイズに対してどの程度のオーダーになっているかはアプリケーションに依存するからである。また、デバイスメモリへのデータ移動のオーバーヘッド等も勘定すると、全体のタスク実行コストをデータのサイズだけから見積もることは極めて難しくなる。そこで我々は、CPU Weight については適当な初期値を与えるが、計算の途中でこの値にユーザが任意に変更可能とする枠組みを提供することにした。すなわち、GPU/CPU へのタスクサイズの割り当てはユーザによって動的に与えられる事ができるものとする。これを実現するために、CPU Weight をプログラムの実行中に動的に再定義する機能として “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 文の直前に挿入することで、プログラムの実行特性に応じて自由に CPU Weight を変更することができる。なお、reset\_weight 指示文のようにアクションを伴う指示文は、API 関数として定義することも可能であるが、その枠組はあくまで XMP-dev/StarPU 言語の中で閉じた形としたいため、指示文形式を取ることとする。

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、先述のようにタスクサイズを実行時間の見積りは難しい。例えば、行列・行列積では行列サイズ  $N$  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可能性がある。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測定を行い、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 を変更することで、ある程度の試行ステップを経た上で、これを最適化することが可能である。

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タイムステップで使う新たな CPU Weight を計算している。ユーザによるロードバランシングを容易にするための補助関数として、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-gpu-wtime() および xmp-cpu-wtime() が用意されており、ここではこれらを用いて 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)。

```

for (int t = 0; t < STEP; t++) {
#pragma xmp device loop on t(i)
  for (i = 0; i < N; i++) {
    // GPU/CPUでワークシェアリング実行されるループ計算の本体
  }

  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 指示文の利用例

そして、その割合が50%に近づくように、つまりGPUとCPUの実行時間に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。

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、loop毎にGPUとCPUの実行時間が異なる場合がある。reset\_weight指示文をこまめに用いることで、loop毎に最適な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。

このような負荷分散制御のためのコードは本来ユーザプログラムに本質的含まれない部分であり、ユーザ自身が記述すべきでないという考えもある。しかし、先述したように、タスクに割り当てられる配列サイズだけからタスク実行時間を予測することは難しく、何らかの動的プロファイリングによる調節が最も有効であると考えられる。ユーザ負担を軽減するため、例えばこういった比較的単純なアニーリング手法をユーティリティ関数として提供する事も考えられる。あるいは、これを標準的な手続きとしてランタイム内に閉じ込め、このような標準的なアニーリングを行うことをデフォルト機能とし、ユーザがより積極的な動的負荷分散制御を行いたい場合はそれを許す、という形の実装にすることも可能である。これについては今後の検討対象とする。

#### 4. 性能測定

本評価では、筑波大学計算科学研究センターで稼働中のGPUクラスタであるHA-PACS [11]を用いる。評価環境は表1に示すとおりであり、本評価では268台の計算ノード中の2から16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

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 CPU/node   | 16 cores (8 cores 2 sockets) |
| # of GPU/node   | 4                            |

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数は $16 - 4 = 12$ となる。評価に用いるベンチマークはN体問題と行列・行列積（以下、単に「行列積」）であり、ともに倍精度浮動小数点演算を行う。また、CPU Weightを最外ループの1イテレーション毎に再計算するようプログラムする。なお、本評価はGPUとCPUの負荷バランスについて着目し、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（例えばLinpackベンチマーク実行中のBLAS (Basic Linear Algebra Subprograms) DGEMMルーチンを何度も呼び出すイメージ)。また、行列積ではGPUにMAGMA blas [12]、CPUにGotoblas [13]を用いることで、小行列の計算を高速化している。本評価の行列積では、行方向に1次元分割をしているため（すなわち、部分行列サイズは行列の1つの方向のみを表し、もう一方は常にNで固定である）、 $C = A \cdot B$ のような演算において、行列BはそのすべてをGPUに転送する必要があり、GPUのデバイスメモリサイズの制約により表5に示す問題サイズが、この環境で解ける最大サイズとなっている。現在、XMP-dev/StarPUコンパイラおよびランタイムシステム自体は二次元分割をサポートしているが、配列をタスクに二次元分割した際にデータが不連続となってしまう。しかし、BLAS自体は引数の配列内のデータが連続領域に格納されていることを想定しているため、結果として二次元分割実装ではBLASを有効に利用できない。そのため本評価では、1次元分割で確実に配列が連続領域であるようにして測定を行っている。今後は、GPUが十分な性能を出せる問題サイズで評価を行うために、これらの問題を解決していく予定である。

まず、CPU Weightの動的変更について評価を行う。本評価ではHA-PACSの2ノードを用い、各ノードのGPU数を4、CPU core数を12とし、ノードあたりのタスク数

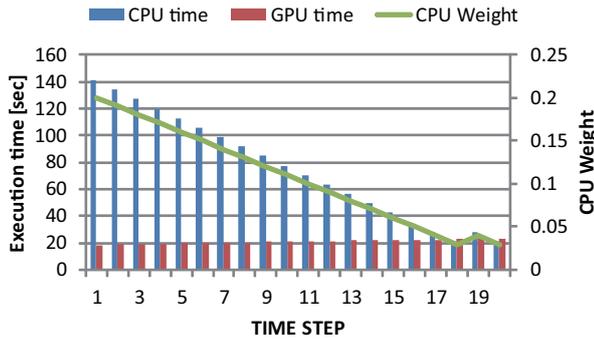


図 5 CPU Weight の推移：N 体問題（粒子数  $N = 819200$ ）

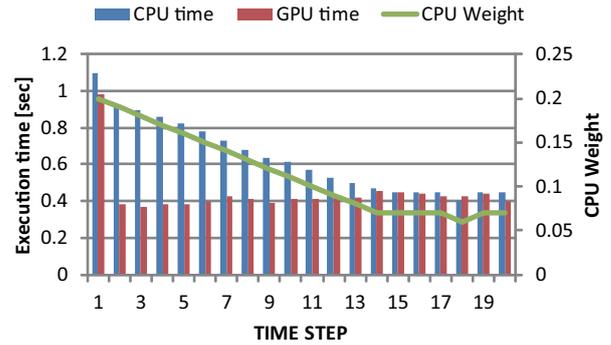


図 6 CPU Weight の推移：行列積（行列サイズ  $8192 \times 8192$ ）

は 16, すなわち各リソースには 1 つだけタスクを割り当てる。CPU Weight の初期値は 0.2 とし, これは図 3 では CPU : GPU = 1 : 4 に相当する。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。この環境で, 最外の時間発展ループを一定回数実行し, CPU Weight の動的変化と共に GPU と CPU の実行時間が近づいてロードバランシングが行われ, 性能が最適化されることを確認する。

図 5 および図 6 はそれぞれ, N 体問題と行列積の TIME STEP 毎の GPU と CPU の実行時間および 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 個に対して非常に大きい。そのため, 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, GPU と CPU の実行時間が徐々に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, 最終的には両実行時間がほぼ均等になっている。N 体問題では, 18 ステップ目, 行列積では 14 ステップ目あたりで GPU と CPU の実行時間がほぼ均衡し, CPU Weight が収束していることがわかる。そして, その段階で全体の実行時間が最短になっていることが示されている。今回の実験では非常に単純な調整アルゴリズムで CPU Weight の自動調整を確認したが, 問題によってはより高速な収束が求められる場合があり, CPU Weight の調整アルゴリズムを工夫する必要があるかもしれない。本実験では, 今回の実装で提供された機能を用い, 原理的に動的負荷分散の調整が可能であることを示した。このようにして, GPU と CPU 間のロードバランスを最適化し, 高水準の PGAS プログラミング言語で簡潔に記述できることを確認した。

表 2, 表 3 に, HA-PACS の 2 ノードを用いて, ノード内の GPU 数を変化させた時の 20 STEP 目の実行時間

表 2 N 体問題：20 STEP 目の実行時間（N：粒子数）

| N      | 1GPU      |            | 2GPU      |            | 4GPU      |            |
|--------|-----------|------------|-----------|------------|-----------|------------|
|        | 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 | 20.035116 | 0.14       | 11.052326 | 0.07       | 6.014442  | 0.03       |
| 819200 | 78.623068 | 0.14       | 42.613314 | 0.07       | 23.017501 | 0.03       |

表 3 行列積：20 STEP 目の実行時間（N：行列サイズ）

| N     | 1GPU     |            | 2GPU     |            | 4GPU     |            |
|-------|----------|------------|----------|------------|----------|------------|
|       | 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       |

( $\max(\text{cpu.time}, \text{gpu.time})$ ) とその時の CPU Weight を示す。表 2 の  $N = 102400$  では, 1GPU を使うときには CPU は全体の計算の 15% を担当しているのに対し, 4GPU の場合には 3% まで減っている。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 の計算時間も短くなる必要があり 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, (StarPU の制約により) 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。また, 表 2 では各タスクの計算量が十分にあるため GPU 数の増加によっておおよそ実行時間が半減していく事がわかる。しかし, 表 3 の  $N = 1024$  のような場合では, 問題サイズが小さいためオーバーヘッドの少ない CPU の計算割合が増えている。行列積では, GPU・CPU とともに 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。 $N = 16384$  では十分な計算量があるため N 体問題同様に表 2 のような変化をしていることがわかる。

最後に, GPU と CPU のワークシェアリング性能について調べる。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調べる。N 体問題および行列積における, 1 TIME STEP における XMP-dev/StarPU の XMP-dev/CUDA に対する相対性能

を図 7, 図 8 に示す。相対性能が 1 より大きければ, GPU のみの演算よりも高速であることになる。ここでは, HAPACS の 2 ノードから 16 ノードを用いて, それぞれノード内の GPU 数を 1 から 4 に変化させて強スケーリングの評価を行う。図 7 では, 全体的に XMP-dev/StarPU で速度向上が得られており, 最大で GPU のみの演算に対して 1.4 倍に性能が向上していることがわかる。また, グラフ中で使用した GPU 数が同じ時, 例えば, 4 ノード 4GPU と 8 ノード 2GPU (どちらも総 GPU 数は 16) の性能は, 4 ノード 4GPU の時よりも, 8 ノード 2GPU の方が高い性能であることがわかる。これらの違いは, 計算に参加する CPU core 数が影響しており, 4 ノード 4GPU では  $48(= (16 - 4) \times 4)$  コア, 8 ノード 2GPU では  $112(= (16 - 2) \times 8)$  コアであり, 8 ノード 2GPU の方が CPU コア数が多い。このことより, CPU の演算リソースが性能向上に寄与していることがわかる。

一方図 8 では, ノード数が少ないとき, 特に 2 ノード 1GPU の行列サイズ 16384 において GPU のみの演算に対して 1.4 倍の性能向上が得られている。しかし, ノード数を増やしていくと徐々に性能が下がり, 8 ノードになると多くの場合において GPU のみの計算よりも遅くなっていることがわかる。行列積における強スケーリングでは, 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。

次にタスクサイズと速度向上の関係について考察する。

表 4, 表 5 に CPU Weight が取束した時における, GPU に割り当てられたタスクサイズ (N 体問題では粒子数, 行列積では部分行列のサイズ) を示す。N 体問題では, ほとんどのケースで GPU のみの場合に対する相対性能が 1 を上回っており, 表 4 では 1.2 倍以上の性能向上が得られた場合をオレンジ色で示し, 相対性能が 1 を下回ったものをブルーで示している。一方行列積は, ほとんどが相対性能で 1 を下回っているため, 表 5 では相対性能で 1 を上回ったものだけオレンジ色で示している。表 4 より, ノード数・GPU 数を増加していくと相対性能が 1.4 倍を超えるオレンジ色の組み合わせが増加, つまり全体の性能向上が得られやすいことがわかる。強スケーリングでは, ノード数・GPU 数を増やしていくに連れて計算リソースに割り当てられる問題サイズは小さくなっていく。そのような状況では, 計算量が大きい時に有利な GPU の性能を引き出すことが難しい。このような時に, CPU の演算性能が加わることで, 性能が伸びにくくなってしまふところを CPU が補っていると考えられる。また, GPU に大きすぎるタスクを渡すと, CPU の演算性能を十分に発揮させることができず大きな速度向上を得ることができない。一方表 5 では, オレンジ色の部分が各ノード数において左下の部分に集まっている。GPU に割り当てられた部分行列サイズをみると, オレンジ色の部分では比較的大きいサイズが割り

表 4 N 体問題: 1GPU に割り当てられた粒子数

| 総粒子数   | 2node  |        |       | 4node  |       |       |
|--------|--------|--------|-------|--------|-------|-------|
|        | 1GPU   | 2GPU   | 4GPU  | 1GPU   | 2GPU  | 4GPU  |
| 102400 | 43520  | 23808  | 12288 | 21504  | 11648 | 6080  |
| 204800 | 88064  | 47616  | 24832 | 43520  | 23808 | 12288 |
| 409600 | 176128 | 95232  | 49664 | 87040  | 47616 | 24832 |
| 819200 | 352256 | 190464 | 99328 | 176128 | 95232 | 49664 |
| 総粒子数   | 8node  |        |       | 16node |       |       |
|        | 1GPU   | 2GPU   | 4GPU  | 1GPU   | 2GPU  | 4GPU  |
| 102400 | 10496  | 5760   | 2976  | 4992   | 2720  | 1392  |
| 204800 | 21504  | 11648  | 6080  | 10496  | 5760  | 2976  |
| 409600 | 43520  | 23808  | 12288 | 21504  | 11648 | 6080  |
| 819200 | 88064  | 47616  | 24832 | 43520  | 23808 | 12288 |

表 5 行列積: 1GPU に割り当てられた部分行列サイズ

| 全体行列サイズ | 2node |      |      | 4node  |      |      |
|---------|-------|------|------|--------|------|------|
|         | 1GPU  | 2GPU | 4GPU | 1GPU   | 2GPU | 4GPU |
| 1024    | 348   | 199  | 101  | 192    | 103  | 57   |
| 2048    | 737   | 445  | 238  | 435    | 243  | 120  |
| 4096    | 1474  | 901  | 496  | 788    | 476  | 253  |
| 8192    | 2908  | 1740 | 972  | 1556   | 911  | 496  |
| 16384   | 5816  | 3440 | 1925 | 2990   | 1781 | 983  |
| 全体行列サイズ | 8node |      |      | 16node |      |      |
|         | 1GPU  | 2GPU | 4GPU | 1GPU   | 2GPU | 4GPU |
| 1024    | 107   | 58   | 28   | 58     | 29   | 12   |
| 2048    | 238   | 124  | 53   | 126    | 63   | 30   |
| 4096    | 465   | 253  | 126  | 253    | 126  | 62   |
| 8192    | 819   | 491  | 250  | 471    | 253  | 124  |
| 16384   | 1576  | 931  | 496  | 860    | 496  | 248  |

当てられているが, 相対性能が 1 を超えていない部分ではこれが小さいことがわかる。これは, 行列積においては全体的問題サイズが, 総リソース量に比べて十分大きくなく, 強スケーリングにおいて, GPU に割り当てられる部分行列サイズが小さすぎ, 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推測される。

## 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, これらの環境で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 は元になっている XMP が PGAS モデルを提供しているため, GPU クラスタ上で容易に並列化が可能である。

XMP-dev/StarPU は, プログラムの記述はオリジナルの XMP-dev/CUDA に準拠している。そのため, 李らの研究 [4] にあるように Laplace 方程式のソルバーについても適用が可能であると考えられる。また, XMP-dev のベースとなっている XMP で記述されたコードも, 一部を変更することで XMP-dev 対応することが原理的に可能である。

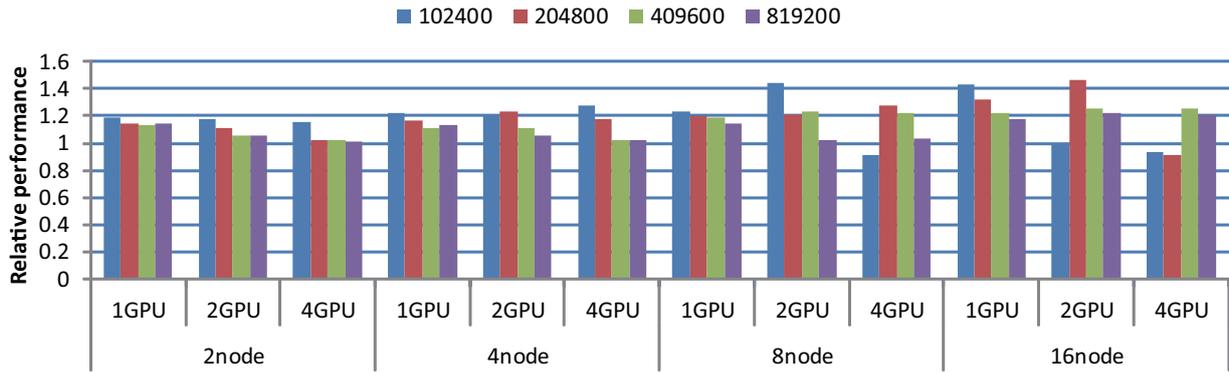


図 7 N 体問題：相対性能

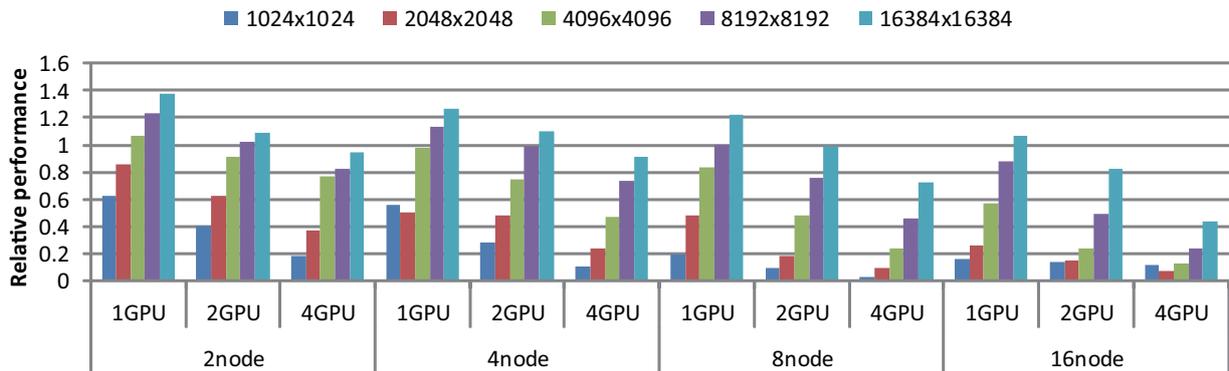


図 8 行列積：相対性能

中尾らの研究 [17] では XMP による共役勾配法の並列化が行われているように、XMP-dev/StarPU は様々なアプリケーションへの対応が可能であると考えられる。

一方、MAGMA [12] における NVIDIA の GPU 向けの BLAS 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 のワークシェアリングを行っており、言語レベルで GPU と CPU によるワークシェアリングを行う 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、より柔軟に両者のワークシェアリングを記述できる。また、BLAS のようなライブラリレベルではなく、ユーザのオリジナルコード全体が並列化・GPU/CPU ワークシェアリングの対象となるため、応用範囲が非常に広い。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

本研究では、GPU クラスタ向け並列言語 XMP-dev と

GPU/CPU ワークシェアリングを行うための 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 に対して最大で約 1.4 倍の性能向上を得ることができた。また、GPU に割り当てるタスクサイズと速度向上率の関係から、GPU が十分に性能を出すことができる計算量を割り当てる必要が有ることがわかった。これより、十分な計算量が確保できなければ GPU のみで計算を行うように XMP-dev/StarPU コンパイラおよびランタイムシステムの変更を行う必要があると考えられる。

今後の課題として、より多くのアプリケーションや問題サイズでの評価を行い、タスクサイズとロードバランスの関係、およびタスクサイズのしきい値について調査をする予定である。

**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。また、本研究では筑波大学計算科学研究センター平成25年度学際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各位に謝意を表す。

## 参考文献

- [1] *CUDA C Programming Guide*. <http://developer.nvidia.com/nvidia-gpu-computing-documentation>.
- [2] 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, チェン トウアン 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, 2011(2):33-50, Apr 2012.
- [5] T. Nomizu, D. Takahashi, J. Lee, T. Boku, and M. Sato. Implementation of XcalableMP Device Acceleration Extension 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, 埴敏博, 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, J. 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] E. Cesar, M. Alexander, A. Streit, J. Traff, C. Cerin, A. Knupfer, D. Kranzlmuller, and S. Jha. A Unified Runtime System for Heterogeneous Multi-core Architectures. In *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415, pages 174-183. 2009.
- [10] 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.
- [11] HA-PACS プロジェクト. <http://www.ccs.tsukuba.ac.jp/CCS/research/project/ha-pacs>.
- [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 - GotoBla2. <http://www.tacc.utexas.edu/tacc-projects/gotoblas2>.
- [14] PGI Accelerator Compiler. <http://www.softtek.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, Sep. 2010.

- [17] M. Nakao, H. Murai, T. Shimosaka, and M. Sato. Productivity and Performance of the HPC Challenge Benchmarks with the XcalableMP PGAS language. In *7th International Conference on PGAS Programming Models*, Edinburgh, Scotland, UK, Oct. 2013.