Vol.2020-HPC-173 No.22
2020/3/18

都市気象コードCity-LESのOpenACCによる並列GPU実装とデータ転送最適化

辻 大亮[1,a)]　朴 泰祐[2,3]　池田 亮作[2,†1]　佐藤 拓人[4]　多田野 寛人[2,3]　日下 博幸[2]

概要：HPC システムでは GPU 等のアクセラレータを用いて計算性能を向上させるのが主流になっている．GPU は高いメモリバンド幅と並列計算能力を持ち，メモリバンド幅律速な高性能計算のワークロードに適したアクセラレータである．我々は CPU アプリケーションとして開発されてきた都市気象コード City-LES (Large Eddy Simulation) の GPU 化によってシミュレーションの高速化を行っている．LES における主要な計算はステンシル計算で構成されており，これまでに支配的な複数の関数を GPU 化することで，全体実行では 2GPU 実行で 2CPU 実行に対して 2.97 倍，4GPU 実行では 4CPU 実行の 2.78 倍高速化を達成していた．本研究ではオーバヘッドとなっていた CPU-GPU 間データ移動を削除するために，GPU での高速化が見込めない関数も含めた残る CPU 実行関数を OpenACC によって GPU 化して City-LES のフル GPU 化を行い，筑波大学計算科学研究センターの Cygnus クラスタ上で最大 32 ノードまでの性能評価を行った．2CPU 実行に対する 2GPU 実行では，データ移動の削除によって 6.2 倍の性能を達成した．Strong Scaling では CPU に対して GPU でノードあたり 10-4.7 倍の性能を達成したが，その一方で問題サイズによってスケール性能が制限されていることが確認できた．また，Weak Scaling ではノードあたりで GPU が 9 倍以上の性能を達成し，スケール時の効率も 86%と良いスケール性能を得ることができた．

1. はじめに

近年の HPC システムではアクセラレータを用いて性能向上を図ることが増えている．アクセラレータとしては GPU が広く用いられており，GPU の特徴である高いメモリバンド幅と大量の計算コアによる高い並列計算能力が多くの HPC 向けアプリケーションのワークロードに適するとされている．筑波大学計算科学研究センターでは 2019 年 4 月より，アクセラレータとして GPU だけでなく高性能 FPGA を搭載するスーパーコンピュータ Cygnus を導入している [1]．Cygnus クラスタでは GPU による高性能並列計算を実現しつつ，GPU の苦手とする演算と低レイテンシな通信を FPGA が担う計算モデルが想定されており，GPU を用いるアプリケーションと FPGA の応用が求められている．

一方で，大規模 HPC システムが求められるアプリケーションの一つに LES (Large Eddy Simulation) がある．LES は流体の数値シミュレーション手法の一つであり，高解像度な計算が可能で，気象分野では NICAM[2] のような全球計算ではなく，都市や街などの小スケールの気象シミュレーションに用いられている．筑波大学計算科学研究センターでは日下・池田らによって LES を用いた気象シミュレーションアプリ City-LES が開発されている [3]．City-LES によるシミュレーション結果の例を**図 1** に示す．City-LES は都市気象に特化しており，街路樹の一本一本を 3 次元的に考慮する樹木モデルを採用し，街区内放射計算をラジオシティ法によって高精度で計算し，街区内の熱環境計算を詳細に行えるという特色を持つ．また，City-LES は MPI + OpenMP のハイブリッド並列化が行われている Fortran で記述された CPU アプリケーションであり，スーパーコンピュータによるシミュレーションも行われているが，近年の主流である GPU クラスタを用いたさらなる高速化が望まれている．そこで本研究では City-LES を対象に GPU を用いた高速化を行い，GPU クラスタを活用可能なアプリケーションの開発を目指す．

これまで，City-LES の主要な計算部分のうち，最も割合の大きかった関数と時間積分処理部分を GPU 高速化し，City-LES に組み込んで性能評価を行ってきた [4], [5]．これにより，City-LES の 2GPU 実行で 2CPU に対して 2.97 倍，4GPU で 4CPU に対して 2.78 倍の高速化を達成することができた．その一方で，計算の部分的な GPU 利用に

[1] 筑波大学 システム情報工学研究科コンピュータサイエンス専攻
[2] 筑波大学 計算科学研究センター
[3] 筑波大学 システム情報工学研究科
[4] 筑波大学 生命環境科学研究科地球環境科学専攻
[†1] 現在，ウェザーニューズ
[a)] dtsuji@hpcs.cs.tsukuba.ac.jp

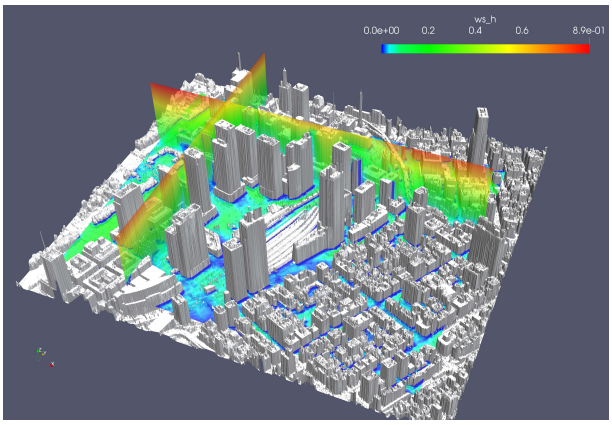

図 1: City-LES のシミュレーション結果の例

よって CPU-GPU 間のデータ移動のオーバヘッドが大きく，これを削除できれば 3.5 倍以上の性能を達成できると予想していた．そこで，本研究ではオーバヘッドとなっていたデータ移動を削除するために，残る City-LES の CPU 実行部分に対して OpenACC を適用することで GPU 実行関数とし，時間発展計算部分のフル GPU 化を行い，Cygnus クラスタにおける性能評価を行う．

2. 関連研究

実際に気象シミュレーションを GPU で高速化した例として，東京工業大学と気象庁が同庁の開発する気象モデルである ASUCA を対象に東京工業大学とフル GPU 化を行っている研究がある [6]．この研究ではフル GPU 化した ASUCA を TSUBAME 2.0[7] 上で実行し，3936GPU による倍精度演算の並列実行で 76.1 TFLOPS を達成している．

小野寺らが行った研究では，GPU による大規模並列実行が可能な LES アプリケーションを開発し，都市部の 10 km 四方を 1 m 格子間隔に解像した高解像度のシミュレーションを行っている [8]．TSUBAME2.0 を用いた性能評価では，単一 GPU 実行によって単精度で 198 GFLOPS を達成し，複数 GPU では単精度で，768 GPU で 115 TFLOPS，1000 GPU で 149TFLOPS を達成している．また，このアプリを用いて東京都心部の実際の建物データを用いたシミュレーションも行っている．

また，理化学研究所計算科学研究センターでは同センターが開発している SCALE (Scalable Computing for Advanced Library and Environment) に含まれている SCALE-LES の GPU 化を行っている [9]．この研究では，GPU 化に伴ってデータ参照の局所性を生かすための計算順序の最適化やカーネルの結合を行い，最終的に CPU に対して GPU で 5 倍以上の高速化を達成している．

このように気象シミュレーションや気象 LES を GPU 実行することで高性能化をする例はいくつもあり，本研究が対象とする City-LES も同様に GPU によって高速化可能であると考えられる．

表 1: 対象とする City-LES の概要 [3]

基礎方程式	非静力ブジネスク近似方程式系
座標系と離散化	直交座標系，Arakawa-C，有限差分法
時間スキーム	3 段階 Runge-Kutta 法 (Wicker and Skamarock 2002)
空間スキーム	2 次，4 次，6 次精度中央差分 3 次，5 次精度風上差分
SGS モデル	TKE-1 方程式モデル (Deadoroff 1980)，Smagorinsky モデル
数値解法	SMAC 法
ポアソン方程式解法	マルチグリッド前処理付き Orthomin(m) 法
短波放射	近藤 (1994)，Dudhia simple(Dudhia 1989)，放射固定
長波放射	近藤 (1994)，RRTM(Mlawer et al. 1997)，放射固定
街区内放射	ラジオシティ法
地表面モデル	Mascart(1995)，フラックス固定
雲物理	warm rain
コード	Fortran90
並列化	MPI (X-Y 方向) + OpenMP (Z 方向)

また，これまでにも二星らや高橋らによって City-LES の GPU アプリケーション化が試みられている [10], [11]．これらの先行研究では，当時の City-LES に対して CUDA/CUDA Fortran を適用し，関数を GPU 化によって高速化しているが，MPI によるマルチ GPU への対応や，計算のフル GPU 化を課題としている．以上の先行研究を踏まえ，これまで CUDA Fortan および CUDA-aware MPI ライブラリを用いて GPU 化と性能評価を行ってきた．本研究では，残る未 GPU 化部分に OpenACC を適用してフル GPU 化することでデータ移動のオーバヘッドを削減した City-LES を実装し，Cygnus クラスタ上で性能評価を行った．

3. City-LES の計算モデル

LES は離散化した格子間隔以上の乱流を直接シミュレーションし，それ以下のスケールの乱流はパラメタライズして計算する数値モデルであり，City-LES の大部分がステンシル計算で構成されている．City-LES の概要を**表 1** に示す．City-LES も主にステンシル計算で構成されており，MPI による地表面と水平な方向である X-Y 方向の領域分割と OpenMP による鉛直上方向である Z 方向のマルチスレッド並列化によってすでに CPU 並列計算機上で高速・高解像度実行が可能になっている．したがって，GPU の持つ高いメモリバンド幅を活かしたより高性能なシミュレーションが達成可能であると考えられる．また，本研究では**表 2** に示した条件に基づいて GPU 化を行う．この条件は City-LES のシミュレーションで頻繁に利用されるもので，GPU による高性能化が達成された際の恩恵が高くなる．

4. City-LES の性能プロファイル

多くの関数から構成されている City-LES の GPU 化をより効果的な関数から行うために，CPU 実行による性能プロ

表 2: GPU 化を行う City-LES モデルの条件

境界条件	周期境界条件
数値解法	SMAC 法（Runge-Kutta3 回目のみ圧力補正）
時間スキーム	3 段階 Runge-Kutta 法
空間スキーム	2 次精度中央差分
SGS モデル	Smagorinsky モデル
建物	建物・植生なし

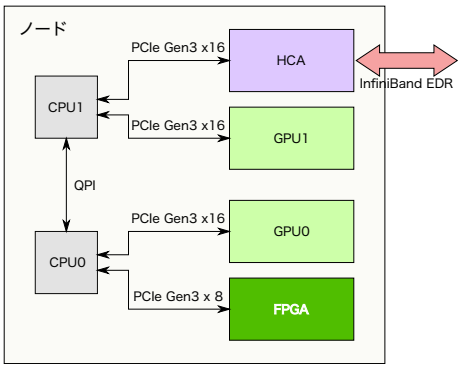

図 2: PPX のノード構成の概要

ファイリングを行った．プロファイリングには筑波大学計算科学研究センターの実験用クラスタ PPX (Pre-PACS-X) のノードを使用した．PPX ノードの構成を**図 2** に，プロファイリング環境を**表 3** に示す．PPX ノードには 2 つの CPU が搭載され，それぞれに PCIe Gen3 で GPU が一台ずつ搭載されている．MPI 通信を含むプロファイリングを行うために，問題領域を 2 MPI プロセスで分割し，各 MPI プロセスでは 1 CPU による 14 スレッド並列実行をした．City-LES の実行時間は計算関数 solve がほとんどを占めているため，その内訳を**図 3** に示す．図 3 中の MGOrthomin_m は，ポアソン方程式をマルチグリッド前処理付きの Orthomin(m) 法で解く関数である [12]．図中凡例における赤字の関数群は前回の [5] までに GPU 化を行った関数群であり，solve 関数の 57%を占める Runge-Kutta 法積分部分と solve 関数の 28%に及ぶ MGorthomin_m 関数部分が該当する．これら関数群の GPU 化によって，それぞれの部分が GPU 上で連続実行可能となっていたが，残る CPU 実行部分とのデータ移動がオーバヘッドとして無視できないものとなっていた．

本研究では，赤字の関数に加えて新たに青字で示した check 関数，mp_driver 関数，smac1 関数，RK-initialization 部分，mp comm および mp pbdy 関数群に対して OpenACC を適用して GPU 関数としている．また，これまでに GPU 化していた赤字の関数群に対しても OpenACC を適用し，性能評価を行っている．また，CPU 実行と GPU 実行での性能比較においては Cygnus クラスタを使用している．

4.1 これまでの実装

前述のように，前回の報告 [5] では計算時間の大きな部分を占めていた MGOrthomin_m 関数や Runge-Kutta 法

表 3: PPX ノード構成と City-LES 性能プロファイリング環境

CPU	Intel Xeon E5-2660 v4(14 cores) x2
GPU	NVIDIA Tesla P100 (PCIe card version) x2
ネットワーク	infiniBand EDR100
ホスト OS	CentOS 7.3
コンパイラ	PGI Compiler 17.10
MPI	MVAPICH2-GDR2.3a
CUDA バージョン	9.0.176
問題サイズ	512x256x128 (256x256x128 x2 プロセス (14 スレッド))
時間発展数	200 ステップ

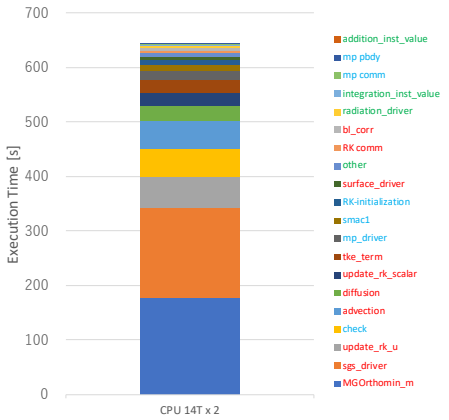

図 3: City-LES solve 関数の性能プロファイル結果

処理部分で呼ばれる関数群を CUDA Fortran 実装し，残りの部分は CPU 実装としていた．CPU 実行となっていた残りの関数は，図 3 に示すように City-LES 全体の中での割合が大きくない関数であったり，GPU を活かしきれるほどの十分な並列性がなかったり，あるいは計算の一部に並列性がないなどの理由で GPU 化されていなかった．これにより，時間発展計算をする上で，物理的に別のメモリを使用する CPU-GPU 間のデータ移動が頻繁に発生し，大きなオーバヘッドとなっていた．**図 4** に，前述した主な関数部分のみを GPU 実装した同じ条件の City-LES を Cygnus クラスタで実行した際の実行時間を示す．それぞれ CPU と GPU の実行時間と，MPI collective 通信時間，MPI 袖通信時間（MPI p2p），CPU-GPU 間のデータ移動時間を示している．図のように，主な関数の GPU 実装によって計算時間が大幅に短縮され，CPU の 441 [s] に対して GPU の 211 [s] と 2 倍以上の性能を発揮できている．一方で，CPU-GPU 間のデータ転送時間は 93.6 [s] と，44%を占める大きなオーバヘッドとなっていることが確認でき，これを削除できればより高い性能を発揮できることになる．

5. OpenACC による GPU 実装

4.1 節で述べたように，これまでの GPU 実装では GPU 実行の主な関数部分と残りの CPU 実装の間で発生するデータ移動が大きなオーバヘッドとなっていた．これを削除するためには残りの CPU 関数を GPU 実装すれば良いが，それらの関数は前述のように GPU に適さないものも

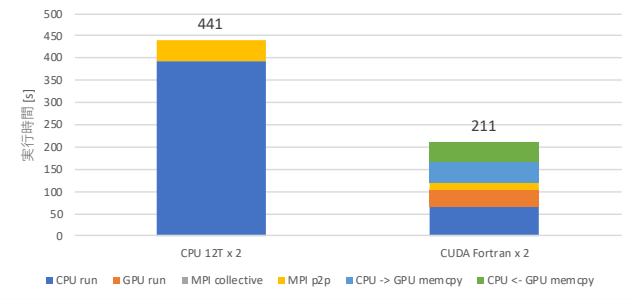

図 4: 主な関数を CUDA Fortran 実装した City-LES との比較

多く，CUDA Fortran 実装するには手間がかかると考えられる．そこで，GPU 化を簡易に行うためにディレクティブベースで容易に GPU 関数を記述できる OpenACC を適用することとする．これまでの研究では CUDA Fortran を用いて GPU 化していたため，CUDA Fortran と OpenACC を併用する形で GPU 化を行う．

5.1 関数およびループの GPU 化

OpenACC ではカーネルの生成には parallel region および kernels region を適用することになる．それぞれ，parallel region がループの並列化方法をユーザ依存とするのに対して，kernels region では該当部分の解析をコンパイラが行って並列化を適用する．また，並列化方法の指定には loop 構文および gang，vector，worker 節を適用する必要があるが，PGI コンパイラでは parallel region と kernels region のどちらを使用しても該当部分を解析して並列化するため，省略したままでの GPU 並列化が可能になっている．そこで，本研究ではカーネルの生成においては GPU 化を簡略化する目的で，コンパイラへの依存度の高い kernels region を適用して自動並列化を行うこととする．また，コンパイラが解析に失敗して並列化できなかった場合に，parallel region の適用や loop 節などの指定によって並列化を行う．

GPU 化した関数およびループで使用する配列変数については，先に CUDA Fortran 実装をしていたため deviceptr 節を使用して CUDA API 定義による変数を用いることとする．また，関数内で定義・使用されている一時配列については，declare create を指定して領域を確保し，present 節で参照を行う．

5.2 MPI 袖通信のパッキング

City-LES では，MPI を用いた X-Y 方向の 2 次元領域分割を採用し，近傍との袖通信には MPI_Type_vector を用いた派生データ型を用いて行う．しかし，GPU 上のデータの MPI_Type_vector で定義した型による通信は，袖領域のストライドアクセスが原因となって性能が悪化する．そこで，本研究では袖通信時にはカーネルによるパック・アンパッキングを行い，連続バッファでの MPI 通信を行うように変更を行う．このパック・アンパックカーネルは前回の発表までに実装していたため，CUDA Fortran 実装となっている．また，カーネル起動時のスレッド構成は，袖領域の各要素に 1 スレッド割り当てて MPI 通信を行う中間バッファに対してコアレッシングするように設定している．

5.3 MGOrthomin_m の GPU 化

City-LES は，ポアソン方程式の求解に V-Cycle によるマルチグリッド前処理を適用した Orthomin (m) 法 (MGOrthomin (m) 法) を用いる [12]．V-cycle を採用するマルチグリッド前処理では，各 Orthomin(m) 法の反復で問題サイズを数回粗くして，粗い問題で大まかに解いた途中解を用いてより細かい問題サイズの途中解を求める処理を繰り返す．OpenACC 実装では単純に kernels region を指定し，3 重ループの内 2 重ループが複数記述されていて解析に失敗した関数については追加で loop independent を指定して回避し，問題サイズを意識せずに GPU 化を行った．また，袖通信時には 5.2 節で述べたようにパック・アンパックを行うように実装した．

なお，[12] とは異なり，マルチグリッド法の平滑化には並列化可能な処理であるという観点から，重み付きヤコビ法による反復を採用している．

5.4 sgs_driver の GPU 化

sgs_driver では SGS (Subgrid Scale) モデルを計算する処理を行う，ステンシル計算によって構成される関数群を呼び出す関数である．したがって，OpenACC 実装ではディレクティブを 5.1 節で述べたように適用して GPU 化を行った．また，sgs_driver では 1 次元の配列要素に並ぶインデックスによって 3 次元データを間接参照するコードが一部含まれている．これらの間接参照に用いられる配列の要素は City-LES の問題設定に依存して決定されるため，実行時に決定される．このような関数に対しても，1 次元配列に対するコアレッシングを期待し，1 次元配列に関する 1 重ループに対してディレクティブを適用して GPU 化を行った．

5.5 check の GPU 化

Check 関数は本研究で新たに GPU 化した関数で，主な目的としてクーラン条件の計算を行う．その一方で，モデルの複数の物理数の全領域における最大最小値を求める機能を持つ．これにより，この関数を単体で GPU 実装した場合，大量の CPU-GPU 間データ移動が必要になるため，この関数は他の関数と同時に実装し，フル GPU 化実装として GPU 実行する必要があった．また，最大最小値計算のために，実行時パラメータによって総数が変化するものの，一つのループ構造の中で 24〜54 個のリダクションを行う必要がある．このような複数のリダクションをまとめ

て行うカーネルを CUDA Fortran で記述するのは労力を要するため，この関数の GPU 化には単純な kernels region でリダクションを記述可能な OpenACC が適していた．

一方で，リダクション結果を格納する変数が全て配列の要素として記述されていたため，そのままのコードではコンパイラによる GPU 化ができないという問題があった．これに対して，コンパイラがスカラ変数に対してはリダクションコードを生成できることを利用して，被リダクション要素の一つ一つをスカラ引数とした関数を新たに作成し，その中で同様のループを kernels region で並列化することで GPU 化を行った．この概要をリスト 1 とリスト 2 に示す．リスト 1 は配列の特定の要素を利用してリダクションを行うループの例である．これに対し，前述の方法でリダクションを行うカーネルを作成した例がリスト 2 である．リスト 1 に対してリスト 2 ではリダクション先がそれぞれスカラ変数として解釈されるため，同じコードのコピーペーストに対して kernels region で並列化可能となる．

5.6 その他の関数の GPU 化

図 3 に示した関数のうち，赤字で示した Runge-Kutta 法処理部分で呼び出される関数群の GPU 化を行った．その多くがステンシル計算で構成される関数であり，これまでと同様にカーネルを作成して GPU 化を行った．一方で，surface_driver は地表の建物等を考慮する関数群であり，問題空間に対する大きさからみて GPU 化には適していない．しかし，CPU-GPU 間のデータ移動の頻度を減らすためにこのような関数でも GPU 化の対象とした．OpenACC を用いた GPU 並列化においては，建物の位置を格納した 1 次元配列による間接参照を行っているループを，1 次元配列に対するコアレッシングを期待して parallel region の指定を指定した．

mp_driver 関数は今回設定した問題条件ではデータコピーを行うため，CUDA Fortran 形式でのデータコピーを記述した．mp pbdy 部や mp comm 部はステンシル計算や袖通信を行うが，他の関数でも使用しているすでに GPU 化できている関数を利用していたため，該当部分および周辺部分を GPU 上メモリを使用するように CUDA Fortran 変数の参照に変更して OpenACC で GPU 化した関数を呼び出すようにした．

smac1 は smac 法の計算を行う関数であるが，通常の 3 重ループや mp pbdy と mp comm のように，他の関数でも使用されている GPU 化済みの関数などで構成されていたため，この関数も同様に使用する変数を CUDA Fortran 変数に変更し，ループにディレクティブを追加して関数は OpenACC で GPU 化したものを呼び出すように変更した．

RK-initialization のような関数ではないその他の部分についても，適宜 OpenACC ディレクティブを適用して GPU 上で計算を行うようにした．

このように，図 3 で示した赤字の関数に加えて青字の関数を OpenACC を用いて GPU 化し，オーバヘッドとなっていた CPU-GPU 間のデータ移動を削除したフル GPU 化 City-LES の実装を行った．

リスト 1: もとの配列要素へのリダクションコード例

```
1  real(8), dimension(n) :: varmax, varmin
2  ...
3  !-- 対象のリダクションループ
4  do k = kms, kme
5    do j = jms, jme
6      do i = ims, ime
7        ...
8        varmax(2) = dmax( varmax(2), array(i,j,k
             ) )
9        varmin(2) = dmin( varmin(2), array(i,j,k
             ) )
10       ...
11     enddo
12   enddo
13 enddo
```

リスト 2: リスト 1 を OpenACC 関数化したリダクションコード例

```
1  subroutine acc_reduction( array, ..., varmax_2
       , varmin_2, ... )
2    real(8), intent(inout) :: ..., varmax_2,
         varmin_2, ...
3    ...
4  #define varmax(X) varmax_ ## X
5  #define varmin(X) varmin_ ## X
6    !-- 対象のリダクションループ
7    !$acc kernels deviceptr( array )
8    do k = kms, kme
9      do j = jms, jme
10       do i = ims, ime
11         ...
12         varmax(2) = dmax( varmax(2), array(i,j
               ,k) )
13         varmin(2) = dmin( varmin(2), array(i,j
               ,k) )
14         ...
15       enddo
16     enddo
17   enddo
18   !$acc end kernels
19 end subroutine
```

6. 性能評価

City-LES について，CPU 実行時と GPU 実行時の実行時間による性能評価を筑波大学計算科学研究センターの Cygnus クラスタ上で行った．**図 5** および**表 4** に Cygnus のノード構成と実行環境を示す．Cygnus のノードには Intel Xeon Gold 6162 が 2 枚搭載され，片方の CPU の下に 2 つの PCIe スイッチを介してそれぞれ NVIDIA Tesla V100 GPU が 2 台ずつ搭載されている．同様に InfiniBand HDR100 のネットワークカードが 2 台ずつ搭載され, FPGA は 1 台ずつとなっている．

はじめに，前回実装した CUDA Fortan 実装版と今回新たに実装した OpenACC 実装版との GPU 実行 City-LES 同士での性能比較を行った．これは公正な比較のために前回までに実装が完了していた図 3 の赤字の関数のみが GPU 化されている．また，スケーリング性能の評価では，Cygnus のノードを最大 32 ノード使用して図 3 の関数のうち赤字と青字の関数を OpenACC で GPU 化したフル GPU 実装の City-LES の性能を評価した．なお，緑色の関数群

表 4: Cygnus のノードの構成および実行環境

CPU	Intel Xeon Gold 6126 (12 cores) ×2
CPU メモリ	DDR4-2666 192GiB (96GiB ×2)
GPU	NVIDIA Tesla V100 (PCIe card version) ×4
ネットワーク	InfiniBand HDR100 ×4
ホスト OS	CentOS 7.6
コンパイラ	PGI Compiler 19.1
MPI ライブラリ	OpenMPI 3.1.4
CUDA バージョン	10.1
コンパイルオプション	-O3 -Mcuda=cc70 -ta=tesla:cc70 -acc -mp -Mextend -mcmodel=medium
時間発展数	200 タイムステップ

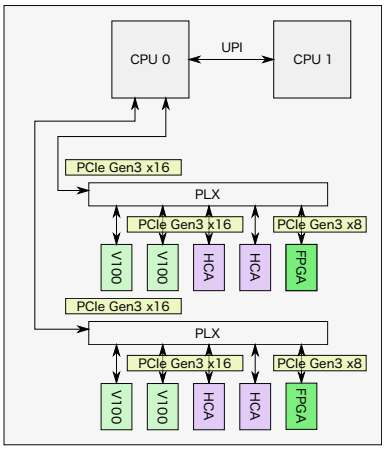

図 5: Cygnus のノード構成図

は出力をしたり，呼ばれる頻度が数十〜数百タイムステップに一度だったりする関係で CPU 実行のままとなっている．図 5 に示したように，Cygnus のノードには 4GPU と 2CPU が搭載されているため，性能評価ではノードあたりの MPI プロセス数を GPU の台数である 4 に設定し，各プロセスでは 2CPU の合計 24 コアを分け合って 6 スレッドの並列実行を行う．また，各プロセス（各 GPU）は同じ PCIe スイッチ下にあるネットワークカードを 1 枚ずつ使うように設定する．

6.1 OpenACC と CUDA Fortran の比較

5 節で述べたように，本研究では残る CPU 実行の関数を OpenACC 実装していたが，今後のコードの修正や最適化が容易になるように，MGOrthomin_m といった主要な関数も簡易な記述で OpenACC 実装を行った．これまで，OpenACC 実装したカーネルは CUDA Fortran 実装のカーネルよりも性能が劣る報告が多くあったため，本研究では新たに OpenACC によって実装した GPU 実行と，前回までに実装していた CUDA Fortan 実装の性能比較を行った．この結果を**図 6** に示す．この比較では，GPU 化部分を [5] 時と同じ Runge-Kutta 法部分および MGOrthomin_m 部分（図 3 における赤字の関数）に限定して，2GPU を使用したノード内実行としている．この実装では部分的な GPU 化

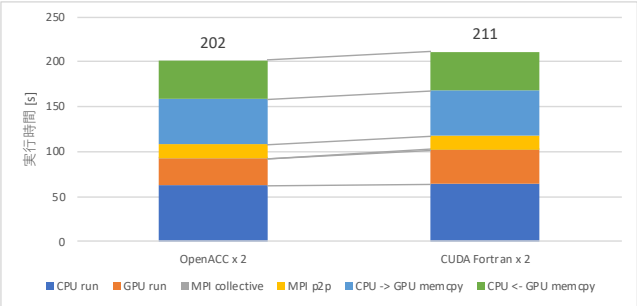

図 6: OpenACC と CUDA Fortran 実装それぞれの実行時間

表 5: advection 関数群のメモリバンド幅性能 [GB/s]
（Unified Cache, L2 Cache, Device Memory）

関数名	OpenACC			CUDA Fortran		
	UC	L2C	HBM2	UC	L2C	HBM2
advection_u_2nd	6908	1396	689	4507	1037	702
advection_t_2nd	3918	1226	706	2579	785	573
advection_scalar_2nd	3950	1225	703	3236	938	618

によって CPU-GPU 間のデータ移動が必要になる．また，問題サイズはプロセスあたり 256×256×128 としている．

それぞれ OpenACC では実行時間が 202 [s]，CUDA Fortran では 211 [s] とほとんど同じになっており，OpenACC を用いても City-LES を CUDA Fortran と同じように高速化できることが明らかとなった．実行時間の内訳では，どちらも MPI 通信時間および CPU 計算時間，CPU-GPU 間データ移動時間はほとんど等しい一方で，数値の差は GPU 計算時間に現れている．OpenACC 実装が高速であった理由としては，CUDA Fortran 実装の最適化不足が挙げられる．CUDA Fortran 実装ではスレッドの構成方法としてスレッドブロック内スレッドを X 方向に 128 スレッドを展開して Y 方向にスレッドブロックを展開していたのに対して，OpenACC ではスレッドブロックが 3 次元に展開され，スレッドブロック内で tailing する構成でカーネルが起動されていた．実際の例を示すと，advection 関数群ではグリッド × スレッドブロック $= (8, 126, 64) \times (32, 4, 1)$ となっていることが確認できた．これにより，**表 5** に示した advection 関数群での実際のメモリバンド幅性能のように，City-LES で多用されるステンシル計算カーネルのキャッシュ効率が改善していたと考えられる．また，**表 6** で示すように，OpenACC 実装したカーネルは CUDA Fortran で実装した場合よりも使用レジスタ数が少ないことも確認できた．これにより，より高い並列度で起動されたカーネルが使用レジスタ数が少なかったことで SM の CUDA コアに対してより高い利用率を発揮できたと考えられる．

表 6: advection 関数群の GPU 化されたカーネルの使用レジスタ数

関数名	レジスタ数		OpenACC / CUDA Fortran
	OpenACC	CUDA Fortran	
advection_u_2nd	96	112	0.86
advection_t_2nd	61	80	0.76
advection_scalar_2nd	72	128	0.56

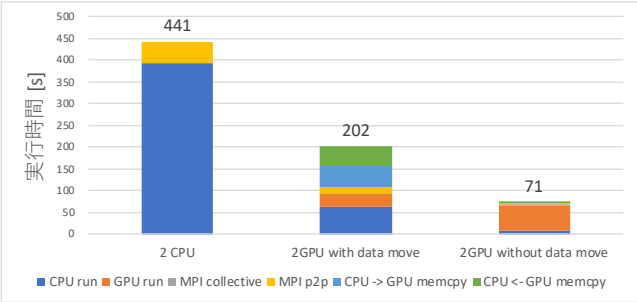

図 7: CPU と OpenACC によるフル GPU 実装の実行時間

6.2 CPU とフル GPU 実装の比較

4.1 節で示したように，これまでの課題は CPU-GPU 間のデータ移動時間を削除することであった．そこで，OpenACC を用いてフル GPU 化によってデータ移動を最適化した City-LES を同じ条件で実行し，CPU との性能比較を行った．この結果を**図 7** に示す．図では 2CPU 実行と，OpenACC による部分 GPU 実行とフル GPU 実行の実行時間を示している．フル GPU 実行時の実行時間は 71 [s] と CPU の 6.2 倍の性能を達成した．また，部分 GPU 実行に対してもオーバヘッドである CPU-GPU 間のデータ移動を削除したことで 2.8 倍高速化することができた．

また，フル GPU 化にあたり，図 3 で示した各関数の性能改善を CPU 実行時とフル GPU 実行時で比較した．ほとんどの関数ではフル GPU 化によって高速に実行できるようになっていたが，radiation_driver，RK-initialization，integration_inst_value，addition_inst_value はそれぞれ 6%，9%，98%，60%性能が低下していた．これは，98%の integration_inst_value に至っては約 2 倍の実行時間がかかってしまうことを示しているが，結果としてデータ移動のオーバヘッドが削減できるため，全体の性能は改善している．

6.3 Strong Scaling

Strong Scaling の結果を**図 8** に示す．これは問題サイズを 1024×1024×128 とし，CPU およびフル GPU 実行での City-LES を 4 ノード 16 MPI プロセスから 32 ノード 128 MPI プロセスまでスケールさせた際の実行時間および並列化効率を示したものである．

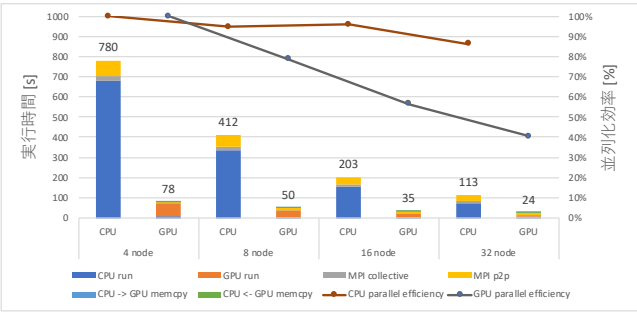

図 8: Cygnus クラスタにおける City-LES の Strong Scaling

4 ノードから 32 ノードへ 8 倍にスケールさせると，CPU では 6.89 倍，GPU では 3.23 倍高速化した．また，4 ノード実行時で CPU に対して GPU では 10 倍，32 ノード実行時には 4.7 倍高速化できた．一方で，並列化効率は 4 ノードに対して CPU では 95%，96%，86%と推移し，GPU では 79%，56%，40%と推移しており，GPU の並列化効率は CPU と比較して劣るものとなっていた．並列化効率が低下した原因としては MPI 通信の増加が挙げられる．図 8 における GPU の通信時間を**表 7** を示す．表 7 のように，GPU 実行ではノード数が増えるにつれて通信時間が増えていることが確認できる．GPU では袖通信時にパッキングを行っているため，問題サイズの縮小とともに効果が弱まりパッキングのオーバヘッドとしての側面が現れたと考えている．また，計算時間は GPU でも 4 ノードに対して 32 ノードで並列化効率が 70%とある程度スケールしていたため，通信時間が相対的に大きくなったことも影響していると考えられる．

表 7: Strong Scaling における GPU の MPI 通信時間 [s]

	4 ノード	8 ノード	16 ノード	32 ノード
MPI collective	1.11	1.60	1.85	1.81
MPI peer-to-peer	2.72	9.97	12.6	10.4

6.4 Weak Scaling

Weak Scaling の結果を**図 9** に示す．図 9 では，プロセスあたりの問題サイズを 256×256×128 とし，CPU およびフル GPU 実行での City-LES を 1 ノード 4 MPI プロセスから 32 ノード 128 MPI プロセスまでスケールさせた際の実行時間を示している．

CPU 実行では 1 ノード実行が 740 [s] であるのに対して，32 ノードでは 841[s] となった．一方，GPU では 1 ノードで 79 [s]，32 ノードで 85 [s] となった．それぞれ CPU に対して，GPU で 9.4 倍と 9.1 倍高速化することができた．また，1 ノードに対する 32 ノード実行時の性能は CPU で 89%，GPU で 86%と，良いスケール性能を示し，GPU でも CPU と遜色のない性能が確認できた．Strong Scaling に対して Weak Scaling ではプロセスあたりの問題サイズが縮小されないため，スケールさせても GPU 上で同等の性能が発揮できた．また，通信時間はスケールにつれて

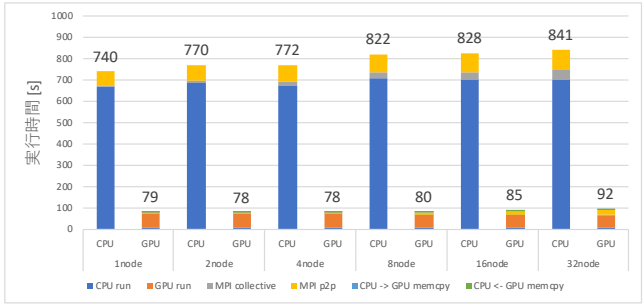

図 9: Cygnu クラスタ s における City-LES の Weak Scaling

ジッタやプロセス数の増加を受けて増加しているものの，問題サイズが縮小されなかったことで袖通信時のパッキングのコストは変化せず，全実行時間に対する割合も大きくなかった結果，性能を維持できたと考えられる．

7. まとめ

本研究では [4], [5] に引き続き，GPU クラスタを活用可能なアプリケーションの開発を目的として，筑波大学計算科学研究センターで開発されている City-LES に対し GPU を用いた高速化を行った．これまで City-LES の主要な関数から GPU 化をしていたが，タイムステップごとに発生する CPU-GPU 間のデータ移動のオーバヘッドが大きかったため，さらなる高速化のために残る CPU 実行部分の GPU 化を OpenACC を用いて簡易に行った．

Cygnus クラスタを用いた CPU とフル GPU 実装の 2CPU と 2GPU での比較では，データ移動のオーバヘッドの削減により，6.2 倍の性能を達成することができた．スケール性能の評価では，4 ノードから 32 ノードまでの Strong Scaling において CPU に対して GPU で 10-4.7 倍の高速化を達成することができた．また，GPU のスケール性能は CPU より劣り，問題サイズの縮小によって制限されていることが確認できた．一方で，1 ノードから 32 ノードまでの Weak Scaling では全てのケースで CPU に対して GPU が 9 倍以上の性能を示し，CPU と遜色のないスケール性能を発揮できていることが確認できた．このように，たとえ部分的に速度が落ちても全データを GPU に置き，基本実行を全て GPU 化することでトータルの性能を向上させることができることを証明した．

今後の課題として，実問題における性能評価が挙げられる．しかし，本研究では City-LES の特色である建物・植生データの利用や，ラジオシティ法による放射計算を無効化していた．これらの計算は建物情報等を用いるため，重い計算となることがわかっており，実問題のシミュレーションを行うために，これらの関数群の性能評価および GPU 化を行う必要がある．また，実問題のシミュレーションではより高解像度な問題あるいはより広範囲の問題を設定することが想定される．そのため，より大きな規模でのスケール性能を評価したいと考えている．さらに，実データによる建物情報を想定する場合には，建物の分布の偏りが想定されるため，その際のスケール性能やロードインバランスの影響を調査したいと考えている．

謝辞 本研究において，OpenACC を積極的に用い，ともかく GPU オフローディングを簡便に記述すればよいということについて，第 170 回 HPC 研究会（SWoPP2019）における議論で東京大学・情報基盤センター・星野哲也助教にご助言を頂いた．そこからフル GPU 化を進め，データ転送量をほぼゼロに削減できたことで大幅な性能向上が達成できた．研究会の場で貴重なご意見を頂いた同氏に深く感謝する．本研究は，文部科学省「次世代領域研究開発」（高性能汎用計算機高度利用事業費補助金）次世代演算通信融合型スーパーコンピュータの開発の一環として実施したものである．また，筑波大学計算科学研究センター・平成 31 年度学際共同利用プログラム課題 「FPGA-GPU 混載プラットフォームにおける HPC アプリケーションとシステム・ソフトウェ アの開発」における複合型演算加速クラスタ Cygnus の利用による．

参考文献

[1] 計算科学研究センター:スーパーコンピュータ - 筑波大学 計算科学研究センター Center for Computational Sciences 入手先 ⟨https://www.ccs.tsukuba.ac.jp/supercomputer/#Cygnus⟩
[2] NICAM : NICAM:非静力学正 20 面体格子大気モデル入手先 ⟨http://cesdweb.aori.u-tokyo.ac.jp/nicam/index.html⟩
[3] Ikeda, R., H. Kusaka, S. Iizuka, and T. Boku.: Development of Urban Meteorological LES model for thermal environment at city scale. 9th International Conference for Urban Climate, Toulouse, France, (July, 2015).
[4] 辻 大亮, 多田野 寛人, 朴 泰祐, 池田 亮作, 佐藤 拓人, 日下 博幸 : 都市気象 LES コードの並列 GPU 環境における高速化．研究報告ハイパフォーマンスコンピューティング（HPC），2019-HPC-168, No.8,（February, 2019）.
[5] 辻 大亮, 多田野 寛人, 朴 泰祐, 池田 亮作, 佐藤 拓人, 日下 博幸 : 都市気象コード City-LES の並列 GPU 実装の最適化と性能評価．研究報告ハイパフォーマンスコンピューティング（HPC），2019-HPC-170, No.39,（July, 2019）.
[6] 下川辺隆史, 青木尊之, 石田純一, 河野耕平, 室井ちあし : メソスケール気象モデル ASUCA の TSUBAME 2.0 での実行. ながれ:日本流体力学会誌 30(2), 75-78, 2011-04-25.
[7] 東京工業大学 学術国際情報センター．TSUBAME2 — [GSIC] 東京工業大学学術国際情報センター．https://www.gsic.titech.ac.jp/tsubame2
[8] 小野寺直幸, 青木尊之, 下川辺隆史, 小林宏光．格子ボルツマン法による 1m 格子を用いた都市部 10km 四方の大規模 LES シミュレーション．ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集 2013, 123-131, 2013-01-08
[9] Mohamed Wahib, Naoya Maruyama: Highly optimized full GPU-acceleration of non-hydrostatic weather model SCALE-LES. IEEE International Conference on Cluster Computing (CLUSTER), Indianapolis, IN, USA, (September, 2013).
[10] 二星義裕, 朴泰祐, 池田亮作, 日下博幸 : 高解像度 LES 計算の GPU による計算加速．ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集 (January, 2012).
[11] 高橋一成, 朴泰祐 : LES による都市気象モデルの GPU クラスタ上での高速化．筑波大学情報学群情報科学類卒業研究論文 (February, 2013).
[12] Tadano H., R. Ikeda, H. Kusaka: Speeding up Large Eddy Simulation by Multigrid preconditioned Krylov subspace methods with mixed precision. The 35th JSST Annual Conference International Conference on Simulation Technology (JSST2016), Kyoto, Japan, (October, 2016).
[13] NVIDIA CORPORATION: PGI — Resources — CUDA Fortran 入手先 ⟨https://www.pgroup.com/resources/cudafortran.htm⟩