Vol.2022-HPC-183 No.17
2022/3/18

GPU クラスタを用いた宇宙輻射輸送コード ARGOT の OpenACC 実装と性能評価

小林 諒平[1,2]　藤田 典久[1]　山口 佳樹[2,1]　朴 泰祐[1,2]　吉川耕司[1,3]　安部 牧人[1]　梅村 雅之[1,3]

概要：我々は，高い演算性能とメモリバンド幅を有する GPU (Graphics Processing Unit) に演算通信性能に優れている FPGA (Field Programmable Gate Array) を連携させ，双方を相補的に利用する GPU-FPGA 複合システムに関する研究を進めている．GPU・FPGA 複合演算加速が必要とされる理由は，複数の物理モデルや複数の同時発生する物理現象を含むシミュレーションであるマルチフィジックスアプリケーションに有効だと睨んでいるためである．マルチフィジックスでは，シミュレーション内に様々な特性の演算が出現するので，GPU だけでは演算加速が困難な場合がある．したがって，GPU だけでは対応しきれない特性の演算の加速に FPGA を利用することで，アプリケーション全体の性能向上を狙う．しかし，その実装方式は GPU で動作する計算カーネルを CUDA にて，FPGA で動作する計算カーネルを OpenCL にて記述するというような複数のプログラミング言語を用いたマルチリンガルプログラミングであり，そのようなプログラミングモデルはプログラマに多大な負担を強いるため，よりユーザビリティの高い GPU-FPGA 連携を実現するプログラミング環境が必要となる．そのことを踏まえ，本研究ではユーザビリティの高い GPU-FPGA 連携の実現を見据えた予備評価として，初期宇宙における天体形成をシミュレーションする ARGOT コードを OpenACC によって実装し，OpenMP ベースの CPU 実装および CUDA ベースの GPU 実装との 1 ノード利用時の性能評価を実施した．その結果，CUDA ベースの GPU 実装と遜色ない性能を達成することが明らかとなったため，本稿では，GPU クラスタを対象に，その OpenACC 実装をマルチノード・マルチ GPU 化し，その性能評価について報告する．

1. はじめに

GPU (Graphics Processing Unit) は，高い演算性能とメモリバンド幅を有することから，多くの HPC 向けアプリケーションのワークロードを加速させるハードウェアとして広く用いられている．しかし，全てのアプリケーションが GPU に適合するということではなく，ユーザーの意図した通りの演算加速を実現できない場合もある．そのようなアプリケーションのうちの 1 つが，マルチフィジックスシミュレーション (連成解析) である．このアプリケーションは，構造，伝熱，流体，電場などの複数の物理現象の相互作用を考慮したシミュレーションを行う．例えば，石炭の燃焼解析 (化学反応と流体計算) や粗視的と微視的とを組み合わせた分子動力学シミュレーションなどがマルチフィジックスシミュレーションに該当する．複数の物理モデルや複数の同時発生する物理現象を含むマルチフィジックスシミュレーションは，そのシミュレーションの性質上様々な特性の演算がシミュレーション内に出現するので，GPU に不適合な演算が部分的に含まれる可能性がある．そして，アムダールの法則より，そのような演算は性能向上のボトルネックとなる．我々は，このような演算を FPGA にオフロードし，GPU と FPGA とを併用することで，アプリケーション全体の性能向上を実現する手法に関する研究を行っており，現在，宇宙輻射輸送を解くシミュレーションコードを対象とした GPU–FPGA 連携の演算加速について着手している [1].

GPU–FPGA 連携の演算加速の実装方式は，CUDA と OpenCL との混合プログラミングである．すなわち，GPU で動作する計算カーネルを CUDA にて，FPGA で動作する計算カーネルを OpenCL にて記述する．ここで，我々が全ての計算カーネルを OpenCL で記述しない理由は次の 3 つである．まず，既存の HPC アプリケーションの大半は CUDA ベースの実装であるため，コードを全て OpenCL に書き直すというのはプログラマにとって非常に負担が大きい．次に，たとえ，ヘテロジニアスな環境でアプリケーションが動作することを前提としたプラットフォームである OpenCL であっても，GPU と FPGA を同時に利用す

1　筑波大学 計算科学研究センター
2　筑波大学 システム情報工学研究群
3　筑波大学 数理物質科学研究群

ⓒ 2022 Information Processing Society of Japan

るためには，GPU 用の OpenCL コンパイラおよび FPGA 用の OpenCL コンパイラを用いて，それぞれの計算カーネルを分割コンパイル・リンクする必要があるため，それは CUDA および OpenCL のプログラム環境と本質的に同様のことを行っているに過ぎない (場合によっては，両者の OpenCL コンパイラを用いた分割コンパイルはシンボルコンフリクトしてリンクできない恐れがある)．そして最後に，HPC に利用される GPU は NVIDIA 製が大半であり，その GPU アーキテクチャに追従するプログラミングモデルである CUDA を利用した方が，GPU の性能を最大限に引き出すことが容易であるのは想像に難くない．これらの理由により，我々は CUDA と OpenCL との混合プログラミングを採用している．しかし一方，このような複数のプログラミング言語を用いたマルチリンガルプログラミングはプログラマに多大な負担を強いるため，よりユーザビリティの高いプログラミング環境が必要とされる．

現在，我々は GPU-FPGA 混載クラスタシステムにおける両演算加速デバイスを，OpenACC による統一的記述によって利用可能にするプログラミング環境についての研究を推進している．OpenACC は，指示文ベースのプログラミングモデルであるため，アプリケーションのどの部分をどのアクセラレータにオフロードすべきなのかをディレクティブによってコンパイラに明示することができる．そして，米国 Oak Ridge National Laboratory (ORNL) では，GPU 用の計算カーネルに加えて FPGA 用の計算カーネルも OpenACC で記述できるようにするためのコンパイラを開発中である．現在，我々と ORNL は GPU-FPGA 混載クラスタシステムにおける両演算加速デバイスの協調計算の実現という目的の下，共同研究を実施しており，我々はその共同研究の一環として ORNL が開発中のコンパイラを利用することによって，上記のユーザビリティの高い GPU-FPGA 連携の実現を目指している．

以上の背景を踏まえ，本稿では，ユーザビリティの高い GPU-FPGA 連携の実現を見据えた予備評価として，初期宇宙における天体形成をシミュレーションする ARGOT コードを OpenACC によって実装し，OpenMP ベースの CPU 実装および CUDA ベースの GPU 実装との性能評価を実施している．過去に，[2] にて，実アプリケーションにおける OpenACC 実装の性能を報告しているが，実装が不完全であったため CUDA 実装と比べて 3 倍程度性能が低かった．したがって，本稿では，その性能低下の要因を取り除いた OpenACC 実装と CUDA 実装の比較結果を最初に示す．次に，マルチノード・マルチ GPU 化した OpenACC 実装を CUDA 実装と比較し，マルチノード・マルチ GPU 版の CUDA 実装と比較して，ほぼ差がない strong scaling を実現できていることを定量的に示す．

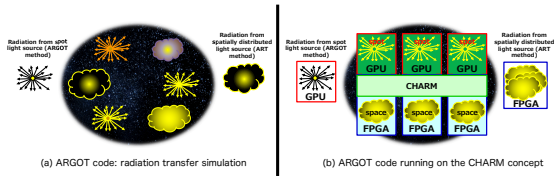


図 1: (a) 宇宙輻射輸送コード ARGOT の概観．(b) CHARM コンセプトによる ARGOT コードの実行モデル．

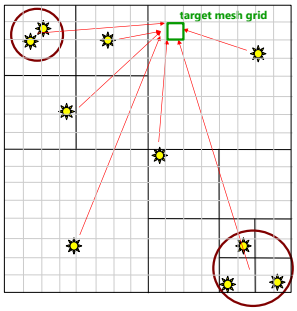


図 2: ARGOT 法の概観

2. 宇宙輻射輸送コード：ARGOT

Accelerated Radiative transfer on Grids using Oct-Tree (ARGOT) は筑波大学 計算科学研究センター (Center for Computational Sciences: CCS) で開発されている宇宙輻射輸送を解くプログラムである．輻射輸送問題は宇宙初期の星や銀河のような天体形成の研究において本質的な要素であり，高速に解くことが求められている．

図 1 (a) に示すように，ARGOT は 2 つのアルゴリズム ARGOT 法 [3] [*1] と Authentic Radiative Transfer (ART) 法 [4] を組み合わせて輻射輸送問題を解く．ARGOT 法のアルゴリズムは点光源からの輻射輸送を計算し，ART 法のアルゴリズムは空間に広がる光源からの輻射輸送を計算する．ART 法は ARGOT プログラムの中で 90%以上の計算時間を占める重要なアルゴリズムであり，本研究では ART 法の演算をこれまでに開発してきた FPGA カーネルを用いて加速させる．また，ARGOT 法の演算には既に ARGOT プログラムに実装されている GPU カーネルを用いることで，図 1 (b) に示すように主要演算部分を GPU と FPGA に適材適所的に機能分散して ARGOT コードを最適化する．

2.1 ARGOT 法

ARGOT 法は，点光源からの輻射輸送を計算するアルゴリズムであり，点光源の数に比例して計算量は増加する．そこで ARGOT 法では，**図 2** に示すように光源の分布を

*1 本論文では，対象とするプログラム名を ARGOT と表記し，その一部であるアルゴリズムを ARGOT 法と表記する．

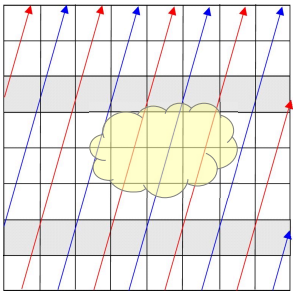


図 3: ART 法で用いられているレイトレーシングの概念図．矢印はレイを表し，黄色の雲は反応を計算するガスを表す．

八分木のデータ構造で扱う．これによって，離れたツリーノード内の光源は単一の光源として扱うことができるため，計算を行う光源の数を N から $logN$ に減らすことができる．あるメッシュグリッド (図 2 の target mesh grid) を対象とした，各点光源からの輻射輸送による光子束は以下の式で求めることができる．

$$f(\nu) = \frac{L(\nu)e^{-\tau(\nu)}}{4\pi r^2} \tag{1}$$

このとき，$L(\nu)$，$\tau(\nu)$，$n(x)$ は振動数 ν での光源の光度，振動数 ν での光学的距離，光を吸収するガス分子の数密度をそれぞれ表し，$\tau(\nu)$ は以下の式で求められる．

$$\tau(\nu) = \sigma(\nu)\int n(\boldsymbol{x})dl \simeq \sigma(\nu)\sum_i n(\boldsymbol{x}_i)\Delta l \tag{2}$$

2.2 ART 法

ART 法では問題空間を 3 次元のメッシュに分割し，その中でレイトレーシングを行うことで輻射輸送の計算を行う．**図 3** に示すように，レイは境界から発射され，それぞれのレイが平行に直進し，反射や屈折はしない．

$$I_\nu^{out}(\hat{\boldsymbol{n}}) = I_\nu^{in}(\hat{\boldsymbol{n}})e^{-\Delta\tau_\nu} + S_\nu(1 - e^{-\Delta\tau_\nu}) \tag{3}$$

式 (3) は ART 法の演算を表し，この式をレイがメッシュを通過する度に計算する．式における ν, I_ν^{in}, I_ν^{out}, $\hat{\boldsymbol{n}}$, $\Delta\tau$, S_ν はそれぞれ周波数，入力放射強度，出力放射強度，レイの方向，メッシュにおける光学的厚み，メッシュの source function を表し，ART 法の計算は全て単精度浮動小数点数を用いて行われる．レイの方向 (角度) は HEALPix アルゴリズム [5] によって求められる．典型的な問題サイズでは，メッシュ数は 100^3 から 1000^3 の規模になり，レイの種類 ((ϕ, θ) の組み合わせの数) は少なくとも 768 方向になる (HEALPix における解像度パラメータ $N_{side} = 8$ の場合)．式 (3) にあるように，ART 法における演算ボトルネックは指数関数である．周波数 ν 毎に 1 回の指数関数の計算が必要であり，周波数の数は問題の設定に依存するが $1 \leq \nu \leq 6$ であり，1 メッシュ通過毎に複数回の指数関数呼び出しを行わなければならない．

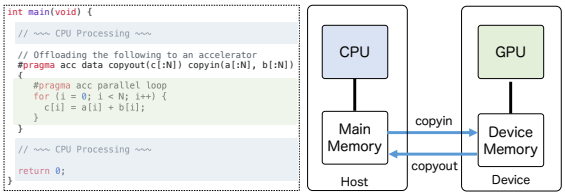


図 4: OpenACC プログラミングモデルの概観

ART 法はレイトレーシングを用いているため，ある 1 つのレイに関する計算は進路に応じて順序通りに計算しなければならないが，異なるレイの間には計算の依存関係がなく並列に計算できる．しかしながら，ART 法を SIMD-like (CPU, GPU など) なアーキテクチャで実装する際には 2 つの問題がある．1 つ目は，メッシュデータに対するメモリアクセスパターンがレイの方向によって様々 (数百〜数千パターン) になることである．複数のレイの計算を SIMD で計算する際に，メッシュデータがメモリ上で連続しない場合があり得る．したがって，キャッシュヒット率の低下や GPU においてメモリアクセスレイテンシの大きさが問題になる．2 つ目に，メッシュに対する積分計算が衝突する可能性があることである．同じメッシュを隣接した複数のレイが通過する (図 3 の灰色のメッシュ部分) 可能性があるため，メッシュ上の複数のレイの効果を重ね合わせる必要があり，これを同時処理するためには，問題を回避するために atomic 演算を用いるか，隣接するレイを同時に計算しない (例えば，図 3 では，赤色のレイと青色のレイに分けて計算している) といった方法が必要となる．ただし，前者の方法では atomic 演算によるオーバーヘッドがあり，後者の方法ではメモリアクセスがより飛び飛びになるオーバーヘッドがある．そして，GPU にとってさらに不利な条件として，各光線の輻射輸送計算はベクトル長が十分でない (ある光線では，すぐにメッシュを抜けてしまう) ことが挙げられる．

こうした ART 法の性質から，我々は CPU や GPU といった SIMD-like のアーキテクチャは ART 法に適さないと考えている．一方で，FPGA はオンチップの内蔵メモリを持ち，低レイテンシ・高バンド幅にランダムアクセスが可能である．それに加えて，FPGA であれば ART 法に最適化したメモリアクセス回路をハードウェアに組み込めるため，ART 法は FPGA での実装に適したアルゴリズムであると考えており，我々は ART 法を高速に計算する FPGA カーネルについて提案している [6]．

3. OpenACC による GPU 実装

3.1 OpenACC

OpenACC は GPU やメニーコアアクセラレータ向けのプログラムを容易に記述することを目的とした並列プログ

ラミング言語規格である．C/C++や Fortran で記述されたプログラムに対し，OpenMP のようなコンパイラ指示文 (#pragma) を挿入することによって，アクセラレータにオフロードすべきプログラムのホットスポットをコンパイラに明示することができる．そのため，アクセラレータのアーキテクチャを意識した低レベルなコードを記述する必要のある CUDA や OpenCL と異なり，既存のソースコードに僅かな修正を加えることでアクセラレータを利用できるプログラミングモデルであることから，これまでに開発された計算科学アプリケーションコードをアクセラレータ環境に移植するための手法として期待が高まっている [7].

図 4 に OpenACC プログラミングモデルの概要を示す．この図では，配列 a と b の要素をそれぞれ足し，配列 c に格納するベクタ加算を実行しているループ部分を切り出して，GPU にオフロードし，残りの処理を CPU が実行している．GPU へのオフロードを行うために，コードに`#pragma acc parallel loop` のコンパイラ指示文を追加している．この`#parallel` ディレクティブは，OpenACC を構成する主要な指示文の一つである並列領域指示文に属し，図のように記述することで，アクセラレータにオフロードされる並列実行領域を指定している．

また，図 4 に示すように OpenACC は，ホストメモリ・デバイスメモリというように 2 つのメモリ空間を配するハイブリッド構成であるため，オフロードされた演算処理に必要なデータをアクセラレータ (デバイス) 側に対してコピーしなければならない．また，処理が終了した後にホスト側へデータを書き戻すことが必要となる．OpenACC では，コードにデータ移動指示文 (`#pragma acc data`) を挿入することによってこれらを実現している．ここで，`copyin(a[:N],b[:N])` は，長さ N の配列 a および b をホストからデバイスにデータ転送を行うことを指示しており，一方，`copyout(c[:N])` は長さ N の配列 c (演算結果) をデバイスからホストにデータ転送を行うことを指示している．なお，この図ではデータ転送をプログラマが明示的にディレクティブを指定してデータのコピーを指示しているが，ディレクティブを指定せずにデータ転送をコンパイラ依存にすることもできる．

このように，演算部分の切り出しやデータ転送を全てコンパイラ指示文で指定でき，ソースコードを直接的に変更することがない．そのため，高いコードのメンテナンス性および移植性を有するプログラミングモデルとなっている．

3.2 ARGOT コードの OpenACC 化

OpenACC を用いた実装を行う際，その実装ステップは [7] に記載されているとおり，以下の様になる．

(1) プロファイリング・ツールを活用し，アプリケーションのホットスポット部位を導出
(2) 並列領域指定指示文である，parallel 指示文，kernels 指示文をコードに追加して，ホットスポット部位を並列化．
(3) data, enter/exit data, update, 指示文などを用いてデータ転送を最適化
(4) loop 指示文による並列粒度の最適化，cache 指示文などによる計算部分の最適化
(5) (1) ∼ (4) の繰り返し

ただし，既にこれまでの研究によって ARGOT コードのホットスポットは判明しており，かつ既存の OpenMP 実装や CUDA 実装より，どのループを OpenACC カーネル化すれば良いかは自明なので，(2) のループ最適化から説明する．

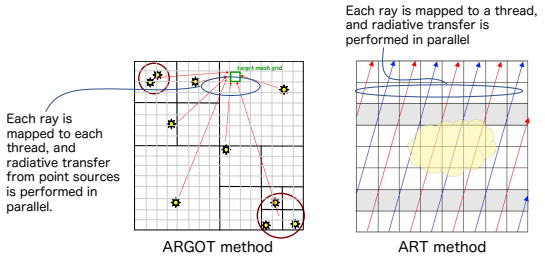


図 5: ARGOT コードの OpenACC 化 (ループ最適化)

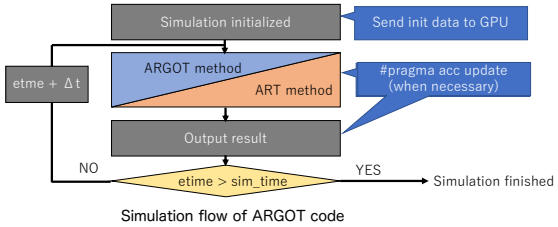


図 6: ARGOT コードの OpenACC 化 (データ転送最適化)

3.2.1 ループ最適化

OpneACC プログラミングでは，GPU カーネルの生成には parallel region または kernels region を適用する．前者ではループの並列化方法がユーザ依存であるのに対して，後者では該当部分をコンパイラが解析して並列化を自動的に行う．本研究では，ARGOT コードの CUDA 実装部分と等価な OpenMP 実装部分 (`#pragma omp parallel for`) を`#pragma acc parallel loop` に書き換えることによって OpenACC 化を行った．指定する並列領域に関数呼び出しが含まれる場合は，その関数に`#pragma acc route seq` を付加している．ループ最適化を適用した ARGOT 法および ART 法の概観を**図 5** に示す．ARGOT 法では，各点光源からの光線を各スレッドにマップして輻射輸送計算を並列に実行し，ART 法では，各光線を各スレッドにマップしてレイトレーシングに基づいた輻射輸送計算を並列に実行する．なお，並列粒度は CUDA 実装と等価になるように，CUDA 実装のブロックサイズおよびスレッド数を参考に `num_gang` と `vector_length` を設定している．

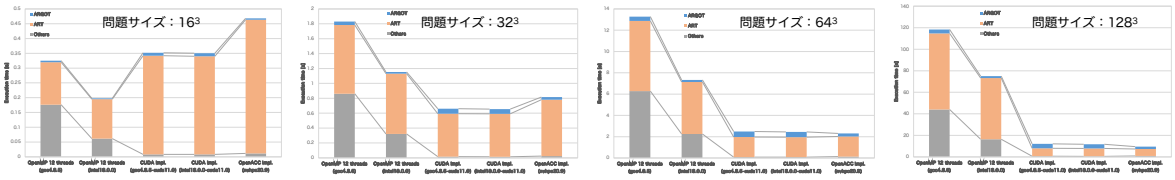


図 7: 問題サイズが 16^3 から 128^3 の場合における 1 GPU 版 OpenACC 実装の性能比較

3.2.2 データ転送最適化

GPU プログラミングでは，ループ構造の並列化と同時に CPU-GPU 間のデータ転送を最小化することが極めて重要である．**図 6** に ARGOT コードのシミュレーションフローを示す．CUDA 実装では，初期化時に ARGOT コードの実行に必要なデータを全て GPU メモリにコピーし，以降は必要に応じて GPU から CPU にデータを書き戻す実装となっている．そのため，OpenACC 実装でも CUDA 実装と同様の挙動になるように，初期化時に`#pragma acc enter data copyin()` にて主要データを GPU メモリにコピーし，ログファイル書き出しのようなホストで実行しなければならない処理にだけ`#pragma acc update host()` を使って，GPU から CPU にデータを書き戻している．

3.2.3 1 GPU を使用した OpenACC 実装の性能評価

OpenACC 化した ARGOT コードの性能を OpenMP ベースの CPU 実装および CUDA ベースの GPU 実装と比較した結果を**図 7** に示す．性能比較には，Intel Xeon Gold 6126 CPU を 1 台，PCIe カード型 NVIDIA Tesla V100 GPU を 1 台使用し，問題サイズは 16^3 から 128^3 までとした．HEALpix アルゴリズムの解像度パラメータ N_{side} は全ての問題サイズで 8 に設定しており，異なる 768 方向のレイを生成する．ここでいう方向とは，球面座標系における偏角 (θ, ϕ) の組合せが 768 種類という意味であって，レイの本数が計算全体で 768 本であるという意味ではない．レイ (平行光) が 768 種の角度で問題サイズに依存した本数分 (N^2) 生成されるため，ART 法の計算量は非常に多いものとなる．図 7 の横軸は使用したコンパイラ，縦軸は 1 シミュレーションステップにおける ARGOT コードの実行時間を示している．実行時間は，`gettimeofday()` 関数を使用して CPU 上にて計測している．

図に示すように，CPU 実装では ART アルゴリズムだけでなく、「Others」の実行も支配的である．この部分は、ARGOT アルゴリズムと ART アルゴリズムの実行結果を基に、主に各メッシュにおける化学反応と輻射加熱・冷却を求めている．各メッシュにおける，化学反応と放射加熱・冷却の演算は独立しているため，CUDA および OpenACC ベースの GPU 実装では多数の演算コアを利用することでその実行を高速化することができる．しかし，ART 法の実行時間は依然として支配的であることには変わらず，ARGOT コードの更なる高速化のためにはこの部分を如何にして演算加速するかが肝要となる[*2]．

CUDA 実装と OpenACC 実装を比較すると，問題サイズが大きくなるにつれて OpenACC 実装の性能が CUDA 実装を最大 20 %程度上回ることが確認された．ただし，CUDA 実装の ARGOT 法では pinned メモリを繰り返し確保するコードが含まれていたため，この影響によるところが大きい．従って，この pinned メモリを初期化時に 1 度だけ確保する方針に変更すれば，CUDA 実装と OpenACC 実装の性能は等価になることが予想される．逆に，問題サイズが小さい場合では，OpenACC 実装の性能は CUDA 実装の 70 % から 80 % 程度となり，これは OpenACC 実装の ART 法の性能が CUDA 実装の ART 法に及んでいないことに起因している．性能が及ばない原因を調査するために，ART 法のコア演算部分のループにおけるコンパイラメッセージを確認したところ，Complex loop carried dependence of ... と表示されており，コンパイラがデータ独立であるかを上手く解析できていないことが分かった．また，16^3 や 32^3 の問題サイズでは，5,120 個の CUDA コアを十分に利用するための並列性が不足しているため，コンパイラの解析能力による影響をより受けたと解釈している．

なお，[2] で報告した OpenACC 実装における ARGOT コードの性能低下は，CPU 実行部分における OpenMP スレッドのオーバーサブスクリプションが主な原因だった[*3]．本稿では，OpenACC 実装における OpenMP スレッドを CUDA 実装と等価になるように設定している．

4. OpenACC 実装 ARGOT コードのマルチノード・マルチ GPU 化

4.1 実装の方針

OpenACC には複数の GPU を使うための機能は存在しないため，複数の GPU を使うためには GPU を呼び出す CPU 側の処理を並列化する必要がある [8]．本稿では， 1 つの MPI プロセスに 1 つの GPU を割り当てる MPI + OpenACC のアプローチを採用する．その実装の概要を**図 8** に示す．

この実装は MPI をベースとしているため，ノード内

*2 我々は，この部分を FPGA を活用して高速化することに成功しているが本稿ではその詳細は割愛する．
*3 CUDA 実装では，OpenMP スレッド数は利用する GPU 数と同じに設定されていた (1 スレッドが 1 GPU を担当する方式)

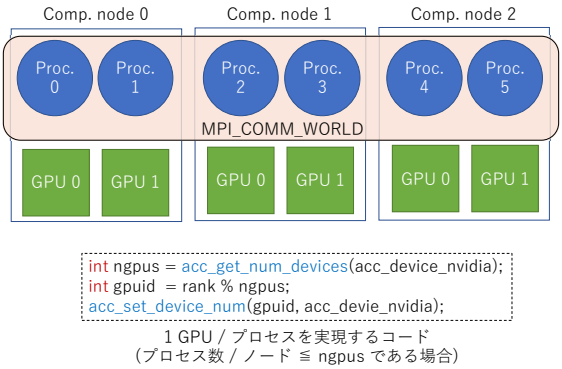


図 8: OpenACC 実装 ARGOT コードのマルチノード・マルチ GPU 化のための実装方針．1 プロセスが 1 つの GPU を担当する．図は [8] を参考に作成．

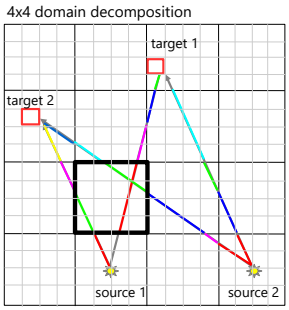


図 9: ARGOT 法のノード並列化の概観．[3] から引用．

だけでなくノード間にまたがって GPU を利用できる拡張性が最大の利点である．それを実現するためのコードが図 8 の下の部分に示されている．まず，1 行目の `acc_get_num_devices()` 関数に，`acc_device_nvidia` を引数として渡すことによって，計算ノード内で利用できる NVIDIA 製の GPU の数を問い合わせる．次に， 2 行目で 各 MPI プロセスのランク番号にしたがって，利用する GPU の番号を計算する．そして，2 行目で求めた GPU の番号を 3 行目の `acc_set_device_num()` 関数に第一引数として渡すことによって，各 MPI プロセスがどの GPU を利用するかの割り当てが完了する．ただし，1 つの MPI プロセスにつき，1 つの GPU という割り当てが正常に動作するためは 1 つの計算ノードにアサインされるプロセス数が計算ノード内で利用できる GPU の数以下でなければならない．なお，この割り当てをしない場合は，計算ノード内の全てのプロセスは 0 番の GPU を利用する．

4.2 ARGOT コードのノード並列化機能

ARGOT コードには，複数の MPI プロセスに対応した並列化機能が実装されている．ARGOT 法および ART 法のノード並列化について以下に述べる．

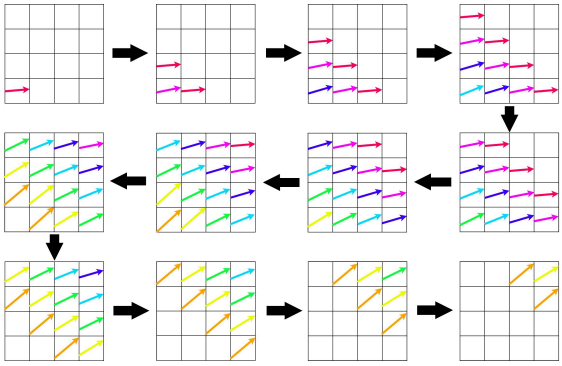


図 10: ART 法のノード並列化の概観．[4] から引用．

4.2.1 ARGOT 法のノード並列化

ARGOT 法のノード並列化の概観を図 **9** に示す．ノード並列化では，シミュレーション空間を各次元に均等に分割する (図では 4 × 4 の domain decomposition)．複数のノードにまたがる光線は，ノード間の境界で「レイセグメント」分割し (図に示すようにセグメント毎に色が異なる)，セグメントの計算が各ノードにて並列実行される [*4]．そして，各セグメントの光学的厚みの計算結果の和を求めて全体の計算結果を求める．なお，1 つの MPI プロセスだけで実行した場合は，source 1 → target 1，source 1 → target 2，source 2 → target 1，source 2 → target 2 の 4 本の光線が「レイセグメント」と見なされ，各光線が各スレッドに割り当てられ並列処理される．

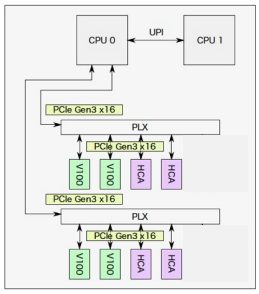
4.2.2 ART 法のノード並列化

ART 法は，3 次元の問題空間をメッシュに分割し，境界面から複数のレイを発射し輻射輸送を計算する．複数の MPI プロセスで領域分割する場合，問題空間はプロセスの数に応じて x, y, z 方向に沿って分割されて割り当てられる (例：2 プロセスの場合は x 方向に沿って 2 分割，4 プロセスの場合は x および y 方向に沿って 2 分割など)．したがって，あるプロセスが担当する問題空間から送出されるレイを，レイの進行方向に位置する次の問題空間を担当するプロセスが受信し，輻射輸送計算を実行することによって複数ノードにおける ART 法が実現される．

複数ノードにおける ART 法の並列化手法として，向きの異なるレイの同時計算が考えられる．しかし，各レイを担当するプロセス順はレイ毎に異なり得るため (たとえ向きが同じであっても)，GPU カーネルの起動制御や MPI 通信の管理が非常に複雑になる．そのため，現在の実装では，Multiple Wave Front 手法 [4] を採用し，レイの方向ごとに順序を制限しつつ，時間並列的なノード並列化を実現している．

*4 各ノードが担当するセグメント同士は互いに独立なので，各セグメントはスレッドに割り当てられ並列処理される

Hardware specification	
CPU	Intel Xeon Gold 6126 x 2
Host Memory	192 GiB (16 GiB DDR4-2666 ECC RDIMM x 6 chan. x 2)
GPU	NVIDIA Tesla V100 (PCIe Gen3 x16) x 4
GPU Memory	32 GiB CoWoS HBM2 @ 900 GB/s with ECC
Network	InfiniBand HDR100 x 4
Software specification	
OS	CentOS 7.9
Host Compiler	gcc 4.8.5
CUDA Toolkit	CUDA 11.0.221 (for CUDA impl.)
OpenACC Compiler	NVIDIA HPC SDK 20.9 (for OpenACC impl.)
MPI	MVAPICH2-GDR 2.3.5



Deneb ノードの構成図

図 11: 評価環境 (Cygnus スーパーコンピュータ)

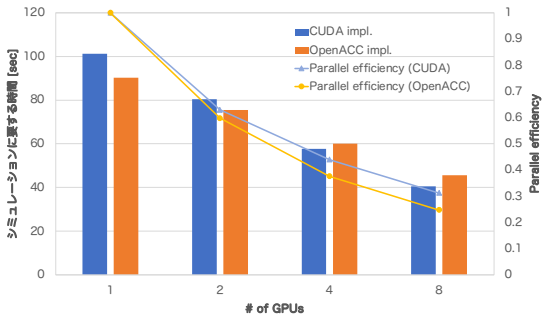


図 12: 問題サイズが 64^3 の ARGOT コードを最大 8 プロセス (8 GPU) で並列実行した場合の性能比較

5. マルチノード・マルチ GPU 版 OpenACC 実装の性能評価

性能評価のプラットフォームとして，筑波大学計算科学研究センターで運用中の Cygnus スーパーコンピュータを用いる．評価環境の詳細を図 **11** に示す．Cygnus は，アクセラレータとして GPU のみを搭載した Deneb ノードと GPU および FPGA を搭載した Albireo ノードの 2 種類の計算ノードから構成されており，本稿における性能評価では 2 台の Deneb ノード (8 GPU) を使用した．

評価に用いる問題サイズは 64^3 とし，それを 1 ～ 8 プロセス (GPU 数) で並列実行した場合に要するシミュレーション時間を CUDA 実装 と OpenACC 実装とで比較評価した．その結果を図 **12** に示す．問題空間は 2 プロセスの場合は x 方向に沿って 2 分割，4 プロセスの場合は x および y 方向に沿って 2 分割，8 プロセスの場合は x, y, z 方向に沿って 2 分割されている．横軸は使用した GPU の数[*5]，左の縦軸は時刻 0.1 秒から 0.2 秒までのシミュレーションに要する実行時間であり，右の縦軸は strong scaling の条件における並列化効率である．なお，CUDA 実装では 1 つのプロセスがノード内全ての GPU を担当する実装方式だったが，`export CUDA_VISIBLE_DEVICES=${MV2_COMM_WORLD_LOCAL_RANK}`を使用し，プロセスから見える GPU の数を 1 つに制限することによって，1 つの MPI プロセスにつき 1 つの GPU がアサインされるようにした．問題サイズ以外の HEALpix アルゴリズムの解像度パラメータのような設定は 3.2.3 節と同じである．

1 プロセスの場合における，CUDA 実装と OpenACC 実装における性能差の原因は，3.2.3 節にて示した ARGOT 法の pinned メモリの確保方法によるものである．そのため，プロセス数を多くすると，プロセスあたりの問題サイズが 32^3 に近づき，CUDA 実装と OpenACC 実装の性能差は逆転する．プロセスあたりの問題サイズが小さい場合における CUDA 実装と OpenACC 実装の性能差は，3.2.3 節にて述べたように，OpenACC コンパイラの解析能力と CUDA コア数に対しての並列性の低さによるものだと考えられる．ただし，CUDA 実装および OpenACC 実装の並列化効率を比較すると，両者の差は数パーセント程度であるため，MPI + OpenACC 実装であっても，MPI + CUDA 実装と同等のノード並列化が実現できている．なお，現時点での OpenACC 実装では CUDA-aware MPI を利用していないため，これを利用することで両者の並列化効率の差は更に縮まることが期待される．

6. おわりに

本稿では，ユーザビリティの高い GPU-FPGA 連携の実現を見据えた予備評価として，初期宇宙における天体形成をシミュレーションする ARGOT コードを OpenACC によって実装し，OpenMP ベースの CPU 実装および CUDA ベースの GPU 実装との性能評価を行った．その結果，1 GPU を使用した OpenACC 実装では，大きい問題サイズの場合においては CUDA 実装とほぼ同等の性能を達成し，小さい問題サイズであっても CUDA 実装の 70% から 80% の性能を達成していることが分かった．また，マルチノード・マルチ GPU 化した OpenACC 実装を CUDA 実装と比較したところ，CUDA 実装よりも若干効率が低いが，ほぼ同等の strong scaling を達成できていることが分かった．この並列化効率の差は，マルチノード・マルチ GPU 版の OpenACC 実装に CUDA-aware MPI を適用することで更に改善されることが期待でき，これが今後の課題として挙げられるまた，実行時間の内訳を tlog のような解析ツールを用いて詳細に解析し，どの部分がノード並列化した際のボトルネックなのかを定量的に明らかにするのも今後の課題である．

謝辞 本研究成果は筑波大学計算科学研究センターの学際共同利用プログラム（Cygnus）における 2021 年度課題「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」を利用して

*5 4 GPU までは 1 ノードで実行した．ノードをまたぐ実行の時は，`-genv MV2_NUM_HCAS 4` を指定し，InfiniBand HDR100 を 4 本利用している．

得られたものである．また，本研究の一部は「高性能汎用計算機高度利用事業」における課題「次世代演算通信融合型スーパーコンピュータの開発」，文部科学省研究予算「次世代計算技術開拓による学際計算科学連携拠点の創出」，及び科学研究費補助金一般 (A)「多重複合演算加速機構を用いた次世代スーパーコンピューティング」による．また，本研究の一部は，NVIDIA 社 丹 愛彦 氏から OpenACC プログラミングの最適化に対する技術的支援を受けたことにより得られたものである．

参考文献

[1] Kobayashi, R., Fujita, N., Yamaguchi, Y., Boku, T., Yoshikawa, K., Abe, M. and Umemura, M.: Multi-Hybrid Accelerated Simulation by GPU and FPGA on Radiative Transfer Simulation in Astrophysics, *Journal of Information Processing*, Vol. 28, pp. 1073–1089 (online), DOI: 10.2197/ipsjjip.28.1073 (2020).

[2] 小林諒平，藤田典久，山口佳樹，　朴泰祐，吉川耕司，安部牧人，梅村雅之：宇宙輻射輸送コード ARGOT の OpenACC による GPU 実装，技術報告 7，筑波大学計算科学研究センター／筑波大学システム情報工学研究群, 筑波大学計算科学研究センター, 筑波大学システム情報工学研究群／筑波大学計算科学研究センター, 筑波大学計算科学研究センター／筑波大学システム情報工学研究群, 筑波大学計算科学研究センター／筑波大学数理物質科学研究群, 筑波大学計算科学研究センター, 筑波大学計算科学研究センター／筑波大学数理物質科学研究群 (2020).

[3] Okamoto, T., Yoshikawa, K. and Umemura, M.: argot: accelerated radiative transfer on grids using oct-tree, *Monthly Notices of the Royal Astronomical Society*, Vol. 419, No. 4, pp. 2855–2866 (online), DOI: 10.1111/j.1365-2966.2011.19927.x (2012).

[4] Tanaka, S., Yoshikawa, K., Okamoto, T. and Hasegawa, K.: A new ray-tracing scheme for 3D diffuse radiation transfer on highly parallel architectures, *Publications of the Astronomical Society of Japan*, Vol. 67, No. 4 (online), DOI: 10.1093/pasj/psv027 (2015). 62.

[5] Gorski, K. M., Hivon, E., Banday, A. J., Wandelt, B. D., Hansen, F. K., Reinecke, M. and Bartelmann, M.: HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere, *The Astrophysical Journal*, Vol. 622, No. 2, pp. 759–771 (online), DOI: 10.1086/427976 (2005).

[6] 藤田典久，小林諒平，山口佳樹，　朴泰祐，吉川耕司，安部牧人，梅村雅之：Optimization on Astrophysical Radiative Transfer Code for FPGAs with OpenCL, 情報処理学会論文誌コンピューティングシステム (ACS)，Vol. 12, No. 3, pp. 64–75 (online), available from ⟨https://ci.nii.ac.jp/naid/170000150497/en/⟩ (2019).

[7] 星野哲也，　松岡聡：圧縮性流体解析プログラムの OpenACC による高速化，技術報告 4，東京大学／東京工業大学, 東京工業大学 (2016).

[8] 星野哲也：OpenACC ではじめる GPU プログラミング中級編 第 1 回，https://www.gdep-sol.co.jp/gpu-programming2-no1.