Vol.2020-HPC-175 No.7
2020/7/30

宇宙輻射輸送コードARGOTのOpenACCによるGPU実装

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プログラムを 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う．

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．

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

[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

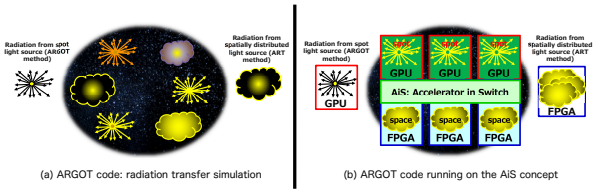

図 1: (a) 宇宙輻射輸送コード ARGOT の概観．(b) AiS コンセプトによる ARGOT コードの実行モデル．

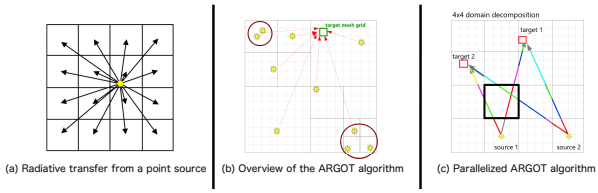

図 2: 点光源からの輻射輸送と ARGOT 法の概観と並列化手法

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行う．

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 [1] [*1] と Authentic Radiative Transfer (ART) 法 [2] 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，**図 2** (a) に示すように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 (b) に示すように光源の分布を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 (b) 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．

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

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．

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

また，ARGOT 法は複数のノードを用いて並列処理することができ，その様子を図 2 (c) 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

[*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．

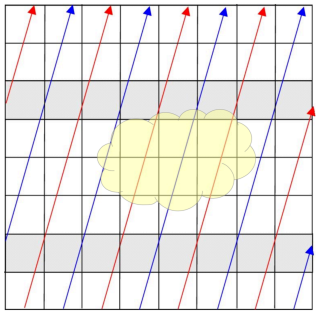

図 3: ART 法で用いられているレイトレーシングの概念図．矢印はレイを表し，黄色の雲は反応を計算するガスを表す．

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 *2．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．ただし，本稿では ARGOT コードは 1 ノードで 実行しているため，この並列化手法は利用していない．この場合，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．

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．

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

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 [3] 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．

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．

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 [4].

*2 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

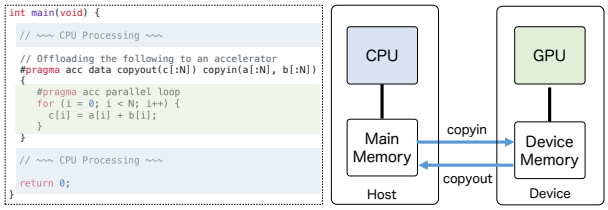

図 4: OpenACC プログラミングモデルの概要

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 [5].

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 [5] 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いるため，(1) については行わない．そして今回は実装の都合上，(2) にあたる最適化のみを行ったためそれに関して説明する．

(2) についての最適化の方針は [6] と同様に GPU カーネルの生成には parallel region および kernels region を適用する．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，本稿ではこの節を省略している．本研究では [6] と同様に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．コンパイラが解析に失敗して並列化できなかった場合や並列性が自明である場合には，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 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．

表 1: 評価環境 (PPX)

CPU	Intel Xeon E5-2660 v4 × 2
CPU Memory	DDR4 2400MHz 64GB (8GB × 8)
GPU	NVIDIA Tesla V100 (PCIe Gen3 x16 card version)
GPU Memory	32 GiB CoWoS HBM2 @ 900 GB/s with ECC
Host OS	CentOS 7.3
Host Compiler	gcc 4.8.5
GPU Compiler	CUDA 9.2.148
PGI Compiler	19.10

4. 評価

4.1 評価環境

通信レイテンシの観点における提案手法の評価には，筑波大学計算科学研究センターで運用中の Pre-PACS version X (PPX) クラスタシステムを用いる．PPX は同センターが開発を計画している PACS シリーズ・スーパーコンピュータ次世代機のプロトタイプシステムであり，Intel FPGA ノードグループ，Xilinx FPGA ノードグループの 2 グルー

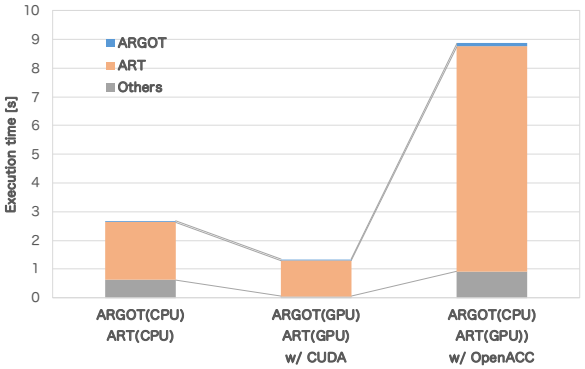

図 5: 問題サイズが 16^3 のときの性能比較

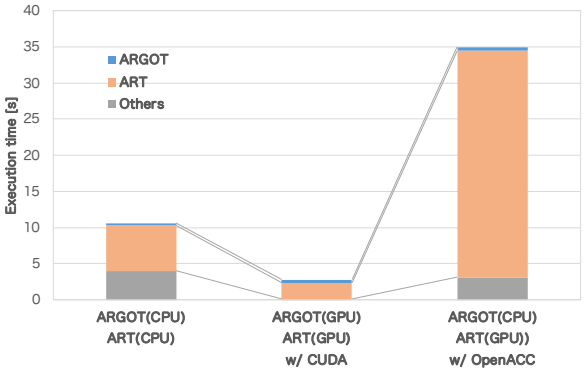

図 6: 問題サイズが 32^3 のときの性能比較

プから構成される．Intel FPGA と Xilinx FPGA は FPGA プラットフォーム比較用に導入され，それらの FPGA をそれぞれ搭載したノードを一体運用しているが，この評価では Intel FPGA のみを利用している．そのため，本節では Intel FPGA を搭載するノードのみの詳細について述べ，それを**表 1** に示す．ノードには，Intel Xeon E5-2660 v4 CPU × 2，NVIDIA Tesla P100 GPU × 1 が搭載されており，CPU-GPU 間は PCIe Gen3 x16 レーンにて接続されている．なお，本評価は 1 ノードのみで行い，Quick Path Interconnect (QPI) を経由する PCIe アクセスによる性能低下を回避するために，GPU 実装の性能評価時は GPU が直接接続されている CPU を用いる．

評価に用いるデータサイズは 16^3 から 128^3 までの間で変化させる．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．

性能評価では，演算時間は CPU 上で計測し，デバイス上で計算を行うためのコスト (カーネルの起動・同期・通信) を含み，GPU・FPGA 間の通信は CPU を経由して実行される．また，本評価では性能の指標として，ARGOT 法と ART 法をどちらも CPU で実行した場合と比較したときのシミュレーション実行速度を用いる．

4.2 ARGOT コードの性能評価

図 5, **図 6**, **図 7**, **図 8** に各問題サイズに応じた CPU 実装 (OpenMP), GPU 実装 (CUDA), ART 法のみを OpenACC 化した版の性能比較を示す．これらの結果は，シミュレーションステップごとの実行時間である．ARGOT (CPU) / ART (CPU) は、いずれのアルゴリズムも OpenMP 実装 (gcc でコンパイル) であることを表し，ARGOT (GPU) / ART (GPU) w/ CUDA は CUDA ベースの GPU 実装，

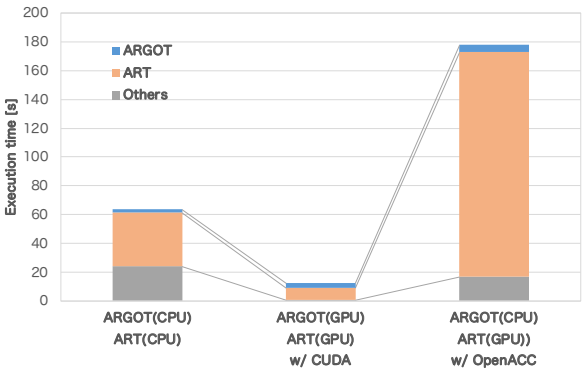

図 7: 問題サイズが 64^3 のときの性能比較

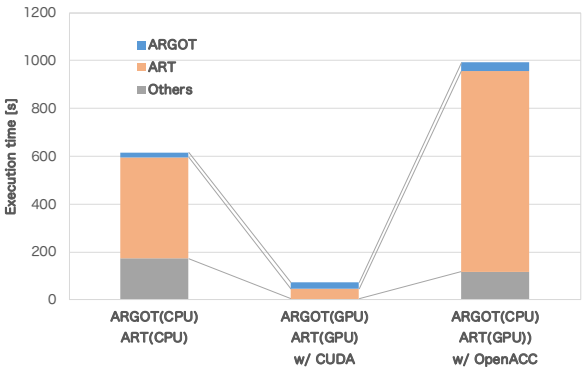

図 8: 問題サイズが 128^3 のときの性能比較

ARGOT (CPU) / ART (GPU) w/ OpenACC は，ARGOT 法は OpenMP ベースの CPU 実装，ART 法は OpenACC ベースの GPU 実装をそれぞれ表す．今回の評価では、単一の Xeon CPU で、CPU の実装は 14 個の OpenMP コア (スレッド) で行い，GPU は NVIDIA Tesla V100 GPU を利用する．

これらの図に示すように，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・冷却の演算はは独立しているため，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 法は，このような状況の恩恵を受けず，GPU ををを使用しても ART 法は ARGOT コードにおける支配的か計算のままである．特に小さい問題サイズでは，GPU の 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，性能向上は僅かであることが分かる．また，CPU に比べてコア自体の性能が低いことや，GPU カーネルの起動や CPU-GPU 間の通信のオーバーヘッドも性能向上を妨げている要因である．

しかし，図 7 や図 8 に示すように，問題サイズが大きくなるにつれて，CUDA ベースの ART 法の性能が大幅に向上していることが分かる．これは，並列性が $O(N^2)$ オーダーで増加し，計算複雑度が $O(N^3)$ オーダーで増加するためである．ART 法は，メッシュに分割された 3 次元空間でのレイトレーシング法に基づいているため，計算量は 3 次関数的に増加する．また、ART アルゴリズムは，あるエッジから別のエッジに進む平行な光線に沿って輻射輸送方程式を解くため，各光線は各 CUDA スレッドに 2 次元的にマッピングされる．したがって，これが並列性の増加が正比例する理由である．すなわち，問題サイズを大きくすることで，SIMD 型プロセッサに本質的に不向きであるという特性や GPU にオフロードするためのオーバーヘッドを押しのけ、CUDA コアの全てを十分に動作させる計算量と並列性が GPU ベースの ART アルゴリズムに現れ始めたため，GPU の性能が向上したということである．

本稿での予備評価は，この恩恵を CUDA ベースの GPU 実装よりも少ないプログラミングエフォートで享受することを視野に入れている．そして上述したように，本稿では ARGOT コードのホットスポットである ART 法を OpenACC でコーディングし，PGI19.10 でコンパイルした．しかしながら，OpenACC ベースの実装は全ての問題サイズにて CPU 実装，GPU 実装どちらの場合においても速度の低下が見られた．これは (2) のホットスポットの並列化を行ったが，正常に並列化できたのはホットスポットの僅か数割の箇所であるためである．また，ホットスポットの大元であるループはコンパイラによる解析が kernels ディレクティブであっても不可能であったため，安直な並列化ができなかったことにも起因している．さらに，本稿では，データ転送の最適化には着手しておらず，全て CUDA ユニファイドメモリ任せの暗黙的なデータ転送であるために，本来であれば不必要なデータ転送も行っている可能性があるため，CUDA ベースの GPU 実装よりもオーバーヘッドが大きくなっていることが予測される．

このように，OpenACC は確かに CPU とアクセラレータとの併用を容易に可能にするプログラミングモデルだが，その最適化には少なからずのプログラミングエフォートが必要であることが示された．今後は，コードのループ構造や利用する変数のメモリ空間を意識してコーディングし，並列化におけるオーバーヘッドを最小化することが今後の課題である．

5. 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，OpenACC は確かに CPU とアクセラレータとの併用を容易に可能にするプログラミングモデルだが，その最適化には少なからずのプログラミングエフォートが必要であることが示された．今後は，コードのループ構造や利用する変数のメモリ空間を意識してコーディングし，並列化におけるオーバーヘッドを最小化することが今後の課題である．

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 2020 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 (B)「再構成可能システムと GPU による複合型高性能プラットフォーム」による．また，本研究の一部は，「Intel University Program」を通じてハードウェアおよびソフトウェアの提供を受けており，Intel 社の支援に謝意を表する．

参考文献

[1] 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).

[2] 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.

[3] 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).

[4] 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).

[5] 星野哲也，　松岡聡: 圧縮性流体解析プログラムの Ope-

nACC 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).
[6] 辻大亮， 朴泰祐，池田亮作，佐藤拓人，多田野寛人，日下博幸：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，技術報告 22，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, 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科, 筑波大学計算科学研究センター／現在，ウェザーニューズ, 筑波大学生命環境科学研究科地球環境科学専攻, 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科, 筑波大学計算科学研究センター (2020).