Vol.2020-HPC-173 No.8
2020/3/17

GPU・FPGA 複合演算加速による 宇宙輻射輸送コード ARGOT の性能評価

小林 諒平[1,2]　藤田 典久[1]　中道 安祐未[2]　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．本稿では，マルチフィジックスの例である，宇宙輻射輸送シミュレーションコード ARGOT を対象にする．ARGOT は，点光源と空間に分散した光源の 2 種類の輻射輸送問題を含む．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，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．また，GPU-FPGA 間のデータ転送には，これまでに提案してきた OpenCL から制御可能な GPU-FPGA 間 DMA 転送を利用する．提案手法を評価したところ，GPU と FPGA に適材適所的に機能分散した ARGOT コードは，そうでない ARGOT コードと比較して最大 10.4 倍の性能向上を達成できた．

1. はじめに

高い演算性能とメモリバンド幅を有する GPU (Graphics Processing Unit) を演算加速装置として搭載する CPU-GPU 構成のクラスタが今日の HPC 分野において広く用いられている．このような構成のクラスタで並列処理を実行するためには，複数ノードをまたがる GPU 間の通信において CPU を介した複数回のメモリコピーが必要であり，このレイテンシの増加によってアプリケーションの性能が低下する問題があった．そこで，筑波大学計算科学研究センターでは，演算加速装置間を低レイテンシの通信ネットワークで密に接続する TCA (Tightly Coupled Accelerators) と呼ばれるコンセプトを提唱し，そのための通信機構である PEACH2 (PCI Express Adaptive Communication Hub Ver.2) [1] を独自開発した．コンセプトの実証システムとして，PEACH2 を搭載した HA-PACS/TCA (Highly Accelerated Parallel Advanced System for Computational Sciences/TCA) を運用し，ノードをまたぐ GPU 同士で低レイテンシ通信が実現されていることを確認した．

PEACH2 は FPGA (Field Programmable Gate Array) を用いて開発されており，FPGA とは任意の論理回路を電気的にプログラムすることができる集積回路である．その特性から，アプリケーションに特化した演算パイプラインと内部メモリシステムを実現する回路を FPGA 上に実装してユーザ所望の処理を加速させることが可能である．[2], [3] では，低レイテンシの通信を実行する回路に加えて，GPU が不得手とする処理を実行する回路を FPGA 上に実装し，それを FPGA に適宜にオフロードすることによってアプリケーション全体の性能を向上させる研究事例が報告されている．このような，FPGA に演算をオフロードし，通信機能と連携することによって演算と通信とを融合するコンセプトを我々は AiS (Accelerator in Switch) と呼んでおり，CPU-GPU クラスタ構成である現在の HPC システムの性能を更に向上させる鍵であると睨んでいる．**図 1** に AiS コンセプトの概要を示す．各ノードには GPU と FPGA が搭載され，それらは PCIe バスを介して接続されている．アプリケーションにおける大規模な粗粒度並列処理部分は従来通り GPU が担当しつつ，GPU ではカバーできない並列性の低い演算部分のオフロードおよび高

[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科
[3] 筑波大学 数理物質科学研究科

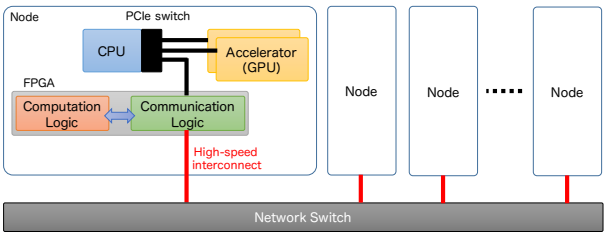

図 1: AiS コンセプトの概要．GPU では粗粒度並列処理を担当する計算カーネルが実行され，FPGA では GPU が不得手とする演算や集団通信を含む高速ノード間通信を担当するカーネルが実行される．CPU はこれらのカーネルの起動および全計算デバイスの調停を行う．

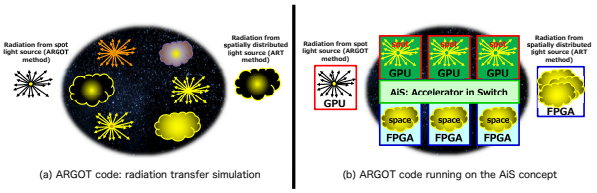

図 2: (a) 宇宙輻射輸送コード ARGOT の概観．(b) AiS コンセプトによる ARGOT コードの実行モデル．

速ノード間通信処理に FPGA を適用することによって，より効率的でレイテンシボトルネックの少ない強スケーリングの実現を目指す．

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 [4] に対して有効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．そのため，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．本稿では，マルチフィジックスの例である，宇宙輻射輸送シミュレーションコード ARGOT を対象にする．ARGOT は，点光源と空間に分散した光源の 2 種類の輻射輸送問題を含む．

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 2** (a) に示すように，ARGOT は 2 つのアルゴリズム ARGOT 法 [5] [*1] と Authentic Radiative Transfer (ART) 法 [6] 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

[*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．

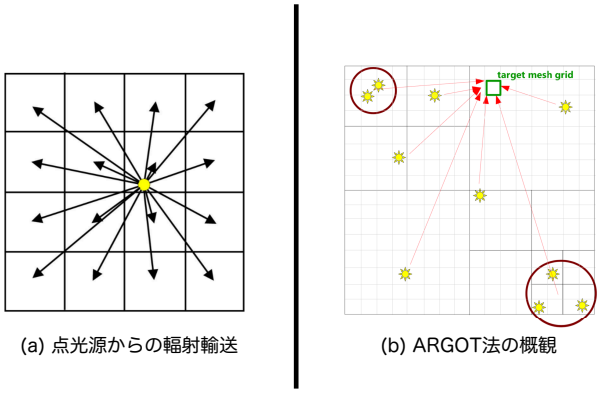

図 3: 点光源からの輻射輸送と ARGOT 法の概観

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 2 (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 3** (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 3 (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 3 (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 4** 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 [*2]．そして，各セグメン

[*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

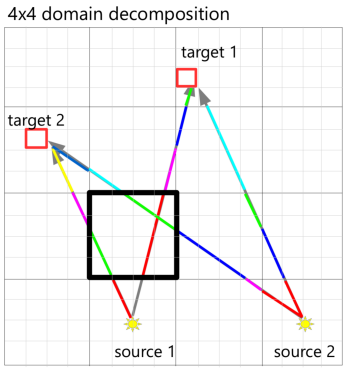

図 4: 並列化した ARGOT 法の概観

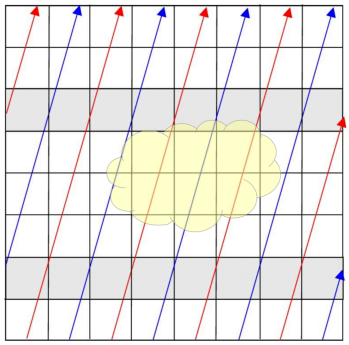

図 5: ART 法で用いられているレイトレーシングの概念図．矢印はレイを表し，黄色の雲は反応を計算するガスを表す．

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 5** 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 [7] 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 5 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 5 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 [8]．

3. ART on FPGA

3.1 Intel FPGA SDK for OpenCL

3.1.1 概要

Intel は OpenCL を用いて FPGA 回路を設計できる開発環境 [9] を提供しており，ART 法の FPGA カーネルの実装はこのツールの利用を前提としている．**図 6** に Intel FPGA SDK for OpenCL におけるプログラミングモデルを示す．ユーザはホスト PC 上で動作するホストコードと

メントはスレッドに割り当てられ並列処理される

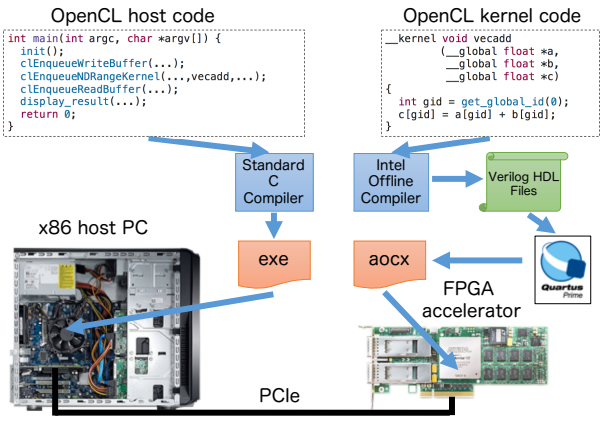

図 6: Intel FPGA SDK for OpenCL のプログラミングモデル．

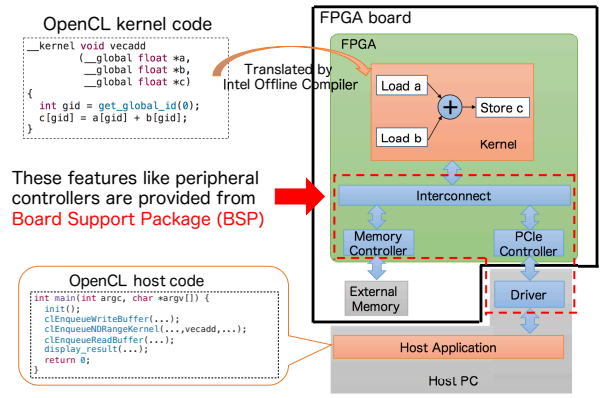

図 7: Intel FPGA SDK for OpenCL プラットフォームの構成図．

FPGA 上で動作するカーネルコードとの 2 種類のコードを記述する．ホストコードは主に OpenCL API (Application Programming Interface) を用いての FPGA のコンフィグレーション，メモリ管理，カーネル実行管理などの FPGA デバイスの制御を担当し，カーネルコードは FPGA にオフロードされる演算を担当する．このプログラミングモデルでは，ホストコードとカーネルコードは別々にコンパイルされ，オフラインコンパイルのみがサポートされている．これは論理合成と配置配線，特に配置配線に数時間要するためである．ホストコードは gcc や Intel Compiler などの標準的な C コンパイラにてコンパイルされ，ホスト PC 上で動作する実行バイナリが生成される．カーネルコードは Intel FPGA SDK for OpenCL に付属している専用コンパイラにて，論理合成可能な Verilog HDL ファイルに変換され，バックエンドで動作する Quartus Prime がその Veirlog HDL ファイルから，FPGA の回路データを含む aocx ファイルを生成する．OpenCL API を用いることで，ホストアプリケーションの実行時に aocx ファイルが FPGA にダウンロード・回路の再構成が行われ，カーネルの実行に必要なデータやカーネルの実行結果などは PCIe バスを介して転送される．

図 7 に Intel FPGA SDK for OpenCL プラットフォームの構成図を示す．C コンパイラによってホストコードからホストアプリケーションの実行バイナリが生成され，Intel FPGA SDK for OpenCL に付属している専用コンパイラによってカーネルコードに記述されている演算をパイプライン処理するハードウェアがカーネルコードから生成される．PCIe コントローラやデバイスドライバ，FPGA デバイスの外部メモリコントローラなどは Bittware や Terasic などの FPGA ボードベンダーから提供される BSP (Board Support Package) に同梱されている．FPGA ボード毎に，FPGA チップや外部ペリフェラル構成は異なる．ボード間のそれらの差異を吸収するために，ボード固有のパラメータや回路は BSP という形で提供され，カーネルコードのコンパイル時に BSP を読み込み利用する．一般的に，OpenCL 対応の FPGA ボードを利用する場合，ボードの開発元から BSP が提供され，ユーザはその BSP を利用して OpenCL を用いた回路開発を行う．そのため，ユーザはホストコードとカーネルコードの実装のみに注力すればよく，たとえ異なる FPGA ボードを利用するとしても，その FPGA ボードの BSP が提供されていれば，既存のコードを移植することが可能である．

3.2 Channel を用いた OpenCL カーネル間通信

“Channel” は Intel FPGA SDK for OpenCL による拡張の 1 つであり，OpenCL カーネル間の通信を行えるようにするものである．Channel の実態はバッファ付き (オプション．なしも可) First-In-First-Out (FIFO) であり，カーネル間に FIFO を通じて通信を行う回路が FPGA 内に生成される．

Channel を使うことの利点は，外部メモリにアクセスすることなく 2 つの OpenCL カーネル間でデータ交換を行える仕組みであることである．一般的な OpenCL の環境において OpenCL カーネル間でデータ交換をする場合は，グローバルメモリを用いるしか選択肢がなかったが，FPGA 環境におけるグローバルメモリは DDR3 や DDR4 の採用が一般的であり，レイテンシやバンド幅の面から性能が期待できない．一方で，2 つのカーネルが Channel で接続されると，グローバルメモリにアクセスすることなく FPGA 内部のデータパスで通信が完了し，低レイテンシ・高バンド幅の通信が行えるようになる．

3.3 実装の概要

図 8 に ART 法の FPGA カーネルの実装の概要を示す．図にあるように，FPGA の中に複数の OpenCL カーネルを実装し，それぞれを Channel で接続して構成されている．図 8 にある “PE Array” は ART 法の演算を実装している OpenCL カーネル群であり，Processing Element (PE),

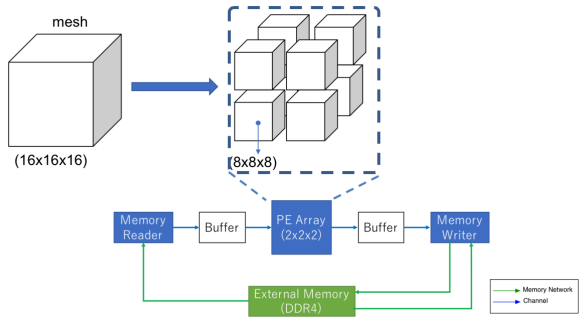

図 8: ART on FPGA 実装の概要.

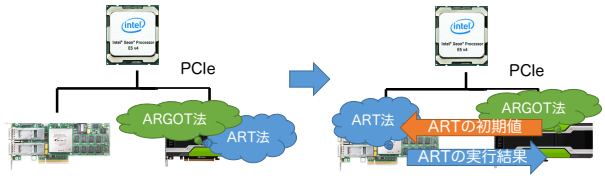

図 9: 実装の方針．ARGOT 法・ART 法をどちらも GPU で実行するコードをベースに，ART 法の部分を FPGA 実装に置き換える．

Boundary Element (BE) から構成され，PE と BE が相互にレイのデータを Channel 経由で通信することで ART 法の計算を行う．

PE は ART 法の演算カーネルを担当するカーネルである．各 PE は図 8 にあるように，1 つの FPGA が担当する問題空間をより小さなブロックに分割し PE に割り当てる．演算用のデータは高頻度にランダムアクセスする必要があるため，Block Random Access Memory (BRAM) を用いて格納しており，それぞれの PE が演算用の BRAM を持つ．BRAM は FPGA 内部に実装されているメモリ (一般的に SRAM である) のことを指し，チップ内に一定のサイズのブロック単位で分散配置されている．BRAM は低レイテンシ・高バンド幅にランダムアクセスでき，非常に高性能であるが，外部メモリと比べて容量が少なく，現時点で最新の FPGA に搭載されている BRAM の容量も高々数十 MB である．BE は PE に対するレイの入出力処理を行うものであり，袖領域に対するレイデータの入出力すなわちレイの初期生成および不要なレイの廃棄と，過去の計算で生成されたレイをレイバッファから読み出す処理，将来の計算で再び用いるためレイバッファに書き出す処理を行う．

4. ARGOT コードにおける GPU・FPGA 連携

4.1 概要

前述したように，本研究では ARGOT コードにおける ARGOT 法を GPU に，ART 法を FPGA にそれぞれオフロードする．その概要を**図 9** に示す．既に ARGOT 法・

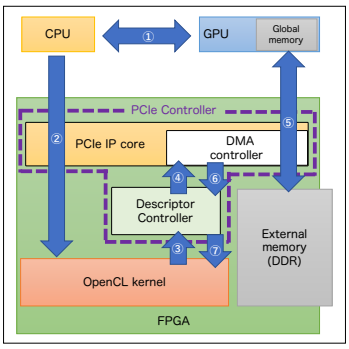

図 10: OpenCL による GPU-FPGA 間データ転送の概略図.

ART 法をどちらも GPU で実装するコードは既に開発されており [5], [6]，そのコードをベースに ART 法の部分を FPGA 実装に置き換えている．ART 法の実行に必要な初期データは GPU で生成した後 FPGA に転送し，ART 法の演算終了後，GPU に演算結果を返信する．そのため，これらのデータを如何にシームレスに GPU・FPGA 間で通信できるかが，効率的な GPU・FPGA 連携を実現するために肝要となる．

我々はこれまでに，GPU デバイスのグローバルメモリと FPGA デバイスの外部メモリ間で CPU を介さずにデータ転送を実現する機能を，PCIe DMA 転送用の IP (Intellectual Property) コアを用いて FPGA 上に実装し，その機能を FPGA ベンダーの提供する OpenCL ツールチェインの仕組みと Veriog HDL とを活用することによって制御する手法を提案している [10]．以降の章にて，我々の提案した GPU-FPGA 間 DMA 転送技術の概要について述べる．

4.2 OpenCL から制御可能な GPU-FPGA 間データ転送

図 10 に，OpenCL から制御可能な GPU-FPGA 間データ転送の概要を示す．この機能は，GPU デバイスのグローバルメモリ，FPGA デバイスの外部メモリを PCIe アドレス空間にマッピングすることで，BSP 内の PCIe コントローラ IP の持つ DMA 機構を用いて双方のメモリ間でデータのコピーを行う．これは，かつて HA-PACS/TCA の開発 [1] において実現した，PCIe 上に接続された GPU と FPGA を PCIe のパケット通信プロトコルを用いて通信させる技術と基本的に同じであるが，この手法では **FPGA が自律的に DMA 転送を起動する**．FPGA から GPU に対しての DMA 転送は以下の手順で実行される．

- CPU 側での設定
  - (1) GPU のグローバルメモリを PCIe アドレス空間にマップ
  - (2) マップしたメモリアドレス情報を FPGA に送信
- FPGA 側での設定
  - (3) GPU メモリアドレス情報を元にディスクリプタ

```
1  #define SIZE  1000000
2
3  tcaresult tcaCreateHandleGPU(unsigned long long *paddr,
       void *ptr, size_t size);
4
5  int main(void) {
6    uint32_t data[SIZE/4];
7    void* ptr;
8    cudaSetDevice(0);
9    cudaMalloc(&ptr, SIZE);
10
11   unsigned long long paddr;
12   tcaCreateHandleGPU(&paddr, ptr, SIZE);
13
14   printf("paddr = 0x%016llx¥n", paddr);
15
16  return 0;
17 }
```

図 11: PCIe アドレス空間へ GPU メモリをマップするコード．12 行目の tcaCreateHandleGPU() 関数で PCIe アドレス空間に GPU メモリをマップし，そのメモリアドレスを paddr に格納する．

を生成し，ディスクリプタコントローラに送信

(4) DMA コントローラにディスクリプタを書き込む

(5) デバイス間 DMA が起動

(6) DMA コントローラが完了信号を発行

(7) OpenCL カーネルで完了信号を検出

なお，CPU 側での設定だが，計算中は FPGA 上に保存された GPU 側アドレス情報やディスクリプタを繰り返し用いるため，①と②は**計算開始時に一度実行するだけで良い**．

4.2.1 PCIe アドレスマッピング

GPU のグローバルメモリを PCIe アドレス空間からアクセスするためには，NVIDIA が提供している API を用いてグローバルメモリを PCIe アドレス空間にマップする必要がある．GPU メモリは CPU 上で動作する CUDA ライブラリや GPU ドライバによって管理されており，この API も GPU ドライバに実装されている．したがって，FPGA から GPU に対して直接通信を行う場合であっても，まず CPU 上で API を用いて PCIe アドレス空間から GPU メモリにアクセスできるように設定しなければならない．そして，DMA 転送を行う際に，GPU を指す PCIe アドレスを DMA 転送先あるいは転送元に指定することで，GPU-FPGA 間の DMA を実現できる．GPU メモリに関する制御には，PEACH2[1] で用いていたカーネルモジュールおよびライブラリを用いる．

PEACH2 で用いていた API を用いた PCIe アドレス空間への GPU メモリのマップ方法を**図 11** に示す．PEACH2 の API である tcaCreateHandleGPU() 関数にホスト側で作成したポインタを渡すことにより，PCIe アドレス空間にマップされた GPU メモリのアドレスである paddr を知ることができる．この関数は，もともと PEACH2 の通信対象とするメモリ領域を識別するためのハンドルを作成する関数であるが，内部的には前述した NVIDIA が提供する Kernel API を用いて GPU アドレスを PCIe アドレスにマップしそのアドレスを取得しており，この手法ではその機能を流用している．

表 1: ディスクリプタの形式．

Bits	Name
[31:0]	Source Low Address
[63:32]	Source High Address
[95:64]	Destination Low Address
[127:96]	Destination High Address
[145:128]	DMA Length
[153:146]	DMA Descriptor ID
[159:154]	Reserved

4.2.2 ディスクリプタの生成

BSP 内の PCIe コントローラは，Intel が自社 FPGA 向けに提供している “Arria 10 Hard IP for PCI Express Avalon-MM with DMA” の IP を利用している．この IP には DMA コントローラ (DMAC: DMA Controller) が内蔵されており，DMAC に対してディスクリプタを書き込むことによって，DMA 転送が行われる．ディスクリプタは**表 1** に示すように特定の形式に従って DMA 転送に必要なデータが格納されている．Source は DMA 転送元 PCIe アドレス，Destination は DMA 転送先 PCIe アドレス，DMA Length は転送長 (ワード単位)，DMA Descriptor ID は転送完了時にどの転送が完了したかを判別するために用いる ID である．このディスクリプタ内の Source や Destination の Address に前節で述べた PCIe アドレス空間にマップされた GPU メモリアドレスをセットすることにより，FPGA は PCIe DMAC を用いて GPU デバイスメモリからのデータ読み出しや GPU デバイスメモリへのデータ書き込みを実行できる．本稿では，PCIe アドレス空間にマップされた GPU メモリアドレスを OpenCL API によって FPGA に送信し，FPGA (OpenCL カーネル) は，受信したアドレス情報を元にディスクリプタを生成し，それを DMAC に書き込むことによって，GPU-FPGA 間データ転送を実行する．

4.2.3 ディスクリプタの書き込み

図 12 に DMAC にディスクリプタを書き込むためのモジュールであるディスクリプタコントローラの構成図を示す．この手法は，OpenCL カーネル内で生成したディスクリプタを I/O Channel API (write_channel_intel 関数) を介してこのモジュールに渡し，ディスクリプタコントローラが受け取ったディスクリプタを DMAC に書き込むことによって GPU-FPGA 間データ転送を実行している．ただし，CPU もホスト-FPGA 間で OpenCL API を用いた DMA 転送 (clEnqueueReadBuffer や clEnqueueWriteBuffer) を実行するためにディスクリプタコントローラを操作する．したがって，それに競合しないように OpenCL カーネルからディスクリプタコントローラに対してアクセスする必要がある．以下にディスクリプタコントローラの動作につい

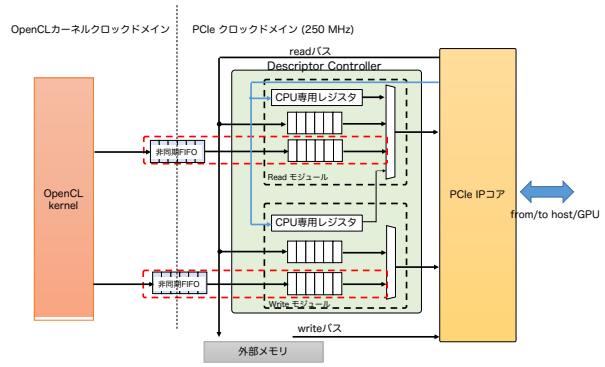

図 12: ディスクリプタコントローラの構成図．赤色の破線で囲まれたコンポーネントを加えることにより OpenCL カーネルからディスクリプタコントローラを操作し，ディスクリプタを DMAC に書き込むことができる．

```
1  #pragma OPENCL EXTENSION cl_intel_channels : enable
2
3  typedef struct __attribute__((packed)) cldesc {
4    ulong src;
5    ulong dst;
6    uint  id_and_len;
7    uint  unused0;
8    uint  unused1;
9    uint  unused2;
10 } cldesc_t;
11
12 channel cldesc_t fpga_dma __attribute__((depth(0)))
       __attribute__((io("chan_fpga_dma")));
13 channel ulong dma_stat __attribute__((depth(0)))
       __attribute__((io("chan_dma_stat")));
14
15 __kernel void fpga_dma(__global float *restrict fpga_mem,
16                         const ulong gpu_memadr,
17                         const uint id_and_len)
18 {
19   cldesc_t desc;
20   // DMA transfer GPU -> FPGA
21   desc.src = gpu_memadr;
22   desc.dst = (ulong)(&fpga_mem[0]);
23   desc.id_and_len = id_and_len;
24   write_channel_intel(fpga_dma, desc);
25   ulong status = read_channel_intel(dma_stat);
26 }
```

図 13: GPU から FPGA への DMA 転送を実行する OpenCL カーネルコード．

て述べる．

ディスクリプタコントローラは FPGA からデータを送信するためのディスクリプタを DMAC に書き込むための Write モジュール，データを受信するためのディスクリプタを書き込むための Read モジュールから構成され，それぞれのモジュールは CPU のみがアクセスできるレジスタ，ホスト-FPGA 間の DMA 転送を実行するためのディスクリプタを格納するための FIFO を有する．ホスト-FPGA 間で DMA データ転送を実行する場合，CPU はまず PIO (Programable IO) アクセスによって，Read モジュール，もしくは Write モジュール内にあるレジスタを操作し，その DMA 転送を実行するためのディスクリプタをホストメモリから FPGA にロードするためのディスクリプタを生成する．そのディスクリプタを Read モジュールから DMAC に書き込むことによって，DMA 転送を実行するためのディスクリプタはホストメモリから読み出され，Read モジュール，もしくは Write モジュール内の FIFO に格納される．その後，FIFO に格納されたディスクリプタを DMAC に書き込むと，ホスト-FPGA 間で DMA データ転送が実行される．

これらの動作を妨げることなく OpenCL カーネルコードから GPU-FPGA 間 DMA データ転送を実行するためには，Read モジュール，Write モジュール内に GPU-FPGA 間 DMA データ転送を実行するためのディスクリプタを格納する FIFO を用意し，プライオリティエンコーダによってそれぞれのモジュールからのディスクリプタの発行を適切に排他制御すれば良い．それらを実行するために図の赤色の破線で囲まれたコンポーネントを Verilog HDL で実装し，ディスクリプタコントローラに付け加えた．なお，OpenCL カーネルとディスクリプタコントローラのクロックドメインは異なるため，OpenCL カーネルコードからディスクリプタコントローラにディスクリプタを送信するためには非同期 FIFO が必要となる．そして [11] と同様に，BSP 内のハードウェアコンポーネントと OpenCL カーネルコードとを関連付けている board_spec.xml を適切に編集することによって，GPU-FPGA 間データ転送を実行する OpenCL カーネルコードを記述することが可能となる．

4.2.4 GPU-FPGA DMA コード例

図 13 は GPU から FPGA への DMA 転送を実行する OpenCL カーネルコードであり，1 行目の pragma は Intel FPGA SDK for OpenCL の独自拡張である channel の有効化をコンパイラに指示するためのものであり，3 ～ 10 行目で DMA コントローラに書き込むためのディスクリプタの構造体を，12，13 行目で I/O Channel 変数である fpga_dma と dma_stat を定義している．GPU から FPGA への DMA 転送なので，ディスクリプタの Source に PCIe アドレス空間にマップした GPU メモリアドレスである gpu_memadr を，Destination に FPGA 外部メモリアドレス (fpga_mem) をセットしている．また，0~127 の id はホスト CPU が利用しているため，OpenCL カーネルで生成されるディスクリプタの id は 128~255 としている．生成されたディスクリプタは write_channel_intel 関数によって，ディスクリプタコントローラにおける Read モジュールに送信され，モジュール内の FIFO でバッファリングされる．その後，適切なタイミングで DMA コントローラに書き込まれ，GPU から FPGA への DMA 転送が実行される．

4.3 GPU-FPGA DMA 機能を組み込んだ FPGA 版 ART

図 8 に示すように，FPGA 版 ART は Memory Reader → PE Array → Memory Writer の順でデータが流れる

```
__kernel void MemoryReader (
    __global art_in volatile* restrict mem,
    // init data of the ART is stored here
    const ulong gpu_memadr
) {
    // Recv init data from GPU
    dma_gpu_to_fpga(gpu_memadr, mem, data_size);

    // ~ send data to the next stage of the pipeline ~
    ...
}

__kernel void MeshWriter (
    __global art_out volatile* restrict mem,
    // rslt data of the ART is written here
    const ulong gpu_memadr
) {
    // ~ get rslt data generated at PE Array ~
    ...

    // Send rslt data to GPU
    dma_fpga_to_gpu(mem, gpu_memadr, data_size);
}
```

図 14: FPGA 版 ART に GPU-FPGA DMA 機能を組み込んだコード例.

演算パイプラインが実装されており，Memory Reader が ART 法の実行に必要な初期データを FPGA の外部メモリから読み込み，PE Array の演算結果を Memory Writer が外部メモリに書き込むことによって処理が終了する．すなわち，Memory Reader に GPU から FPGA への DMA 転送を実行する OpenCL カーネルコードを，Memory Writer に FPGA から GPU への DMA 転送を実行する OpenCL カーネルコードを追加することで，シームレスな GPU・FPGA 間通信を実現できる．

図 14 にそのコード例を示す．前述したとおり，Memory Reader と Memory Writer は OpenCL カーネルとしてそれぞれ実装されている．GPU-FPGA 間 DMA を実行するために必要となる，PCIe アドレス空間にマップされた GPU メモリのアドレス情報は，4 行目および 16 行目でカーネル引数としてセットされており，これらのアドレス情報は図 11 の API を用いて CPU が取得する．これらのアドレス情報と FPGA の外部メモリのポインタ，そして通信データサイズをセットすることによって，Memory Reader カーネルは，GPU から FPGA への DMA 転送 (7 行目) によって mem に ART の初期データを格納し，Memory Writer カーネルは，FPGA から GPU への DMA 転送 (22 行目) によって，mem に格納された演算結果を gpu_memadr の位置の GPU のメモリに書き込んでいる．FPGA 版 ART に組み込んだ DMA 転送性能を評価したところ，[10] で報告されている性能と同等の性能を達成することが確認できた．

4.4 GPU・FPGA 連携 ARGOT コードのコンパイル方法

図 15 に GPU・FPGA 連携 ARGOT コードのコンパイルフローを示す．前述したとおり ARGOT コードにおける ARGOT 法は GPU 実装であり，プログラムは CUDA によって記述されている．そして，ART 法は FPGA にオフロードされ，プログラムは OpenCL で記述される．図に示すように，CUDA コードと OpenCL ホストコードは，nvcc と g++ を用いて分割コンパイルされ，コンパイルされて生成されたオブジェクトファイルは，nvcc でリンクされることによって，実行ファイル (図の a.out) が生成される．そして，OpenCL カーネルコードはホストコードとは別にコンパイルされ，カーネルコードから ART 法を実行する回路情報を含む aocx が生成される．この aocx ファイルは，前述したとおりホストアプリケーションの実行時に OpenCL API を用いることで FPGA にダウンロードされ，その際に回路のコンフィグレーションが行われる．

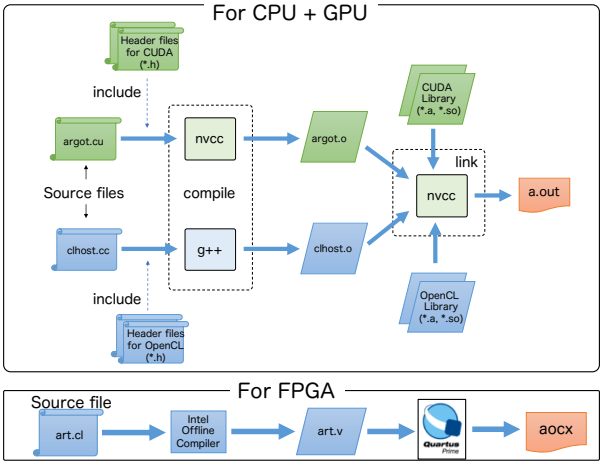

図 15: GPU・FPGA 連携 ARGOT コードのコンパイルフロー.

5. 評価

5.1 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 2** に示す．ノードには，Intel Xeon E5-2660 v4 CPU × 2，NVIDIA Tesla P100 GPU × 2，Mellanox InfiniBand ConnectX-4 EDR HCA × 1，BittWare A10PL4 FPGA ボード × 1 が搭載されており，CPU-GPU 間は PCIe Gen3 x16 レーンにて，CPU-FPGA 間は FPGA ボードの

表 2: 評価環境 (PPX)

CPU	Intel Xeon E5-2660 v4 × 2
CPU Memory	DDR4 2400MHz 64GB (8GB × 8)
GPU	NVIDIA Tesla P100 × 2 (PCIe Gen3 x16 card version)
GPU Memory	16 GiB CoWoS HBM2 @ 732 GB/s with ECC
Host OS	CentOS 7.3
Host Compiler	gcc 4.8.5
GPU Compiler	CUDA 9.1.85
OpenCL SDK	Intel FPGA SDK for OpenCL 17.1.2.304
FPGA	BittWare A10PL4 (10AX115N3F40E2SG)
FPGA Memory	DDR4 2133MHz 8GB (4GB × 2)

仕様のため PCIe Gen3 x8 レーンにてそれぞれ接続されている．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，FPGA と GPU 実装の性能評価時は各デバイス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．ART 法の FPGA カーネルは 8 PE ($= 2^3$) で構成され，そして各 PE は 8^3 メッシュを格納できる BRAM を持つ．すなわち，16^3 サイズの場合は全てのメッシュデータを FPGA の BRAM に格納できる．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．

5.2 FPGA リソース使用量

表 3 に FPGA 実装のリソース使用量を示す．Adaptive Logic Modules (ALMs) と Registers は FPGA の基本要素であり論理回路を構成するために用いられる．M20K は BRAM を表し，DSP は浮動小数点数演算に用いられるブロックである．ART 法の演算は全て単精度で構成されており，指数関数を含め全ての演算は DSP ブロックに実装される．“Freq.” は OpenCL カーネルから生成された回路のクロックドメインにおける動作周波数を表す．

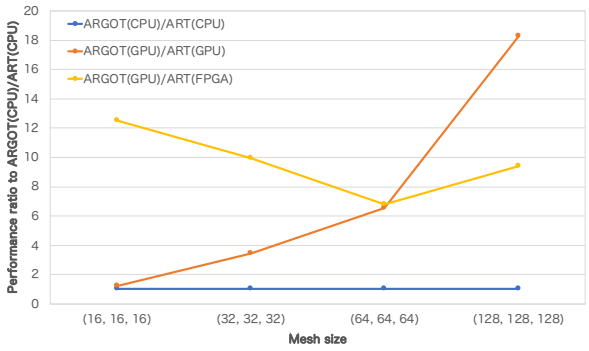

図 16: ARGOT 法と ART 法をどちらも CPU で実行した場合を 1 としたときの ARGOT コードの実行速度向上比．

問題サイズ 16^3 と 32^3 のリソース使用量を比較すると，大きな差があることがわかる．これらの差は前章で述べたレイバッファの制御カーネルによって発生する．16^3 は問題サイズが小さく全てのデータを BRAM に格納することができるため，レイバッファが使われない．したがって，レイバッファに関するカーネルが削除され FPGA に実装されないためである．また，レイバッファ制御カーネルはクリティカルパスにもなっており，問題サイズ 16^3 と 32^3 の比較で約 20MHz 動作周波数が低下している原因であり，今後これらのカーネルの最適化を行わなければならない．

問題サイズ 32^3, 64^3, 128^3 における周波数の差は，OpenCL コンパイラによって発生している．これらのサイズで用いている OpenCL コードは，問題サイズやループカウントなど一部の定数を除いて同じである．コンパイラによって生成される Verilog HDL コードは可読性が低いため，OpenCL コードがどのように回路として表現されているかを知ることが難しく，周波数の差がどのように発生しているかを解析することは困難である．

5.3 ARGOT コードの性能評価

図 16 に CPU 実装，GPU 実装，FPGA 実装の性能比較を示す．ARGOT(CPU) / ART(CPU) は ARGOT 法と ART 法がどちらも CPU 実装，ARGOT(GPU) / ART(GPU) はどちらも GPU 実装，ARGOT(GPU) / ART(FPGA) は ARGOT 法が GPU 実装，ART 法が FPGA 実装でであることを表す．CPU 実装は C 言語ベースであり，OpenMP を用いたスレッド並列処理が適用されている．本稿では，単一の Xeon CPU (14 コア) を利用しており，1 コア 1 スレッドのマッピングでプログラムを実行した．GPU 実装は，CPU 実装をベースとしており，コードは CUDA で記述されている．

表 3: FPGA リソース使用量とカーネルの動作周波数.

Mesh size	# of PEs	ALMs	(%)	Registers	(%)	M20K	(%)	DSP	(%)	Freq. [MHz]
(16, 16, 16)	(2, 2, 2)	167,199	39%	325,496	19%	1,224	45%	800	53%	190.0
(32, 32, 32)	(2, 2, 2)	197,319	46%	382,036	22%	1,299	48%	800	53%	191.7
(64, 64, 64)	(2, 2, 2)	197,296	46%	382,158	22%	1,300	48%	800	53%	163.0
(128, 128, 128)	(2, 2, 2)	197,804	46%	382,221	22%	1,300	48%	800	53%	184.4

ARGOT 法と ART 法をどちらも CPU で実行した場合，すなわち ARGOT(CPU) / ART(CPU) を 1 としたときの ARGOT コードの実行速度向上比を図 16 に示す．ARGOT(GPU) / ART(FPGA) と ARGOT(GPU) / ART(GPU) の性能は全ての問題サイズにおいて，ARGOT(CPU) / ART(CPU) より優れていることが分かる．ARGOT(GPU) / ART(GPU) と ARGOT(GPU) / ART(FPGA) を比較すると，後者の構成はメッシュサイズ 16^3 と 32^3 のそれぞれにおいて，10.4 倍と 2.89 倍の速度向上を達成した．これは，P100 GPU にとってこれらのメッシュサイズが小さすぎるため，3584 CUDA Core に対して十分な演算の並列度が得られないことに起因している．そのため，メッシュサイズが大きくなるにつれて，ARGOT(GPU) / ART(GPU) の性能は向上し，メッシュサイズ 128^3 においては，ARGOT(GPU) / ART(FPGA) の 1.94 倍の性能となる．

しかし，[8] によると，メッシュサイズ 128^3 の場合においても，FPGA の性能は P100 と同等になることが報告されているので，この ARGOT コードの性能比較は先行研究で報告されている通りの性能変化となっていないことが分かる．この原因を調査するために，ART 単体の性能を CPU 実装，GPU 実装，FPGA 実装とで比較した．次節にて，その結果を述べる．

5.4 ART 単体の性能比較

前述したとおり，ARGOT コードにおける FPGA 版 ART は，[8] で提案されたデザインを利用している．しかし，ARGTO コードに [8] を組み込む際に，演算パイプラインの途中にリダクション演算処理を挟み込む必要があることが判明し，本稿の ART (FPGA) はその修正を反映している．その修正が，前節で述べた ARGOT コードの性能に影響を及ぼしていると睨み，本稿では ARGOT プログラムから作成したベンチマークプログラムを用いて ART 単体の性能を評価し，[8] で報告されいているものと比較した．レイデータについては HEALpix ライブラリを用いて ARGOT プログラムと同様の方法で作成するが，ART 法はメッシュデータの値によって計算負荷が変化しないためメッシュデータの入力には疑似乱数を用いており，[8] における評価と同様の方法を採用している．

表 4 に本稿における ART 法と [8] で報告されている ART 法との実行時間の比較を示す．[8] と同様に，“CPU(14C)” は CPU 実装を 14 スレッド (= 1 ソケット) で実行する場合，“CPU(28C)” は 28 スレッド (= 2 ソケット) で実行する場合を表す．GPU 実装は Tesla P100 GPU を利用し，FPGA は BittWare A10PL4 PFGA ボードに搭載されている Arria10 FPGA である．

表 4 に示すように，本稿における CPU 版 ART と GPU 版 ART の性能は [8] と比較して僅かに (最大で 1.5 倍) 低下していることが分かる．これは，[8] の CPU 版 ART と GPU 版 ART にも，本来必要とされる処理 (FPGA 版 ART で必要とされる処理であるリダクション演算と同等の処理) が欠けているためである．しかし，それを踏まえても本稿における FPGA 版 ART の性能低下は CPU 実装や GPU 実装と比べて大きく，[8] の FPGA 版 ART と比べて 3∼4 倍の性能差が発生している．これは，[8] の FPGA 版 ART の Initiation Interval (II) が 1 なのに対し，本稿における FPGA 版 ART の II が 4 であることに起因している．II とは，演算パイプラインが何サイクルに 1 回起動するかを示す指標であり，II が 1 であるなら演算パイプラインは毎サイクル起動しており，スループットが 100%となる．すなわち，II が 4 の場合，演算パイプラインは 4 サイクルに 1 回の起動となるためスループットは 25%となってしまい，性能が劇的に悪化してしまう．

ただしこれは OpenCL コンパイラの解析能力不足によるところが大きく，適切なワークアラウンドを講じれば II が 1 になり，性能が元通りになると期待される．また，メッシュサイズが小さい場合，ARGOT (GPU) / ART (FPGA) の性能は ARGOT(GPU) / ART(GPU) よりも現時点で既に優れているため，II が 1 になればその性能差は更に拡大する．すなわち，問題サイズが小さいほど，FPGA オフローディングは ARGOT コードを加速させる最適解となり，その傾向は強スケーリングとの親和性が非常に高い．さらに，GPU と異なり，Stratix 10 のようなハイエンド FPGA は強力な外部通信リンクが搭載されているので，この特性も強スケーリングとの親和性を向上させている．

6. おわりに

本稿では，本研究では初期宇宙における天体形成シミュレーションで重要な役割を持つ輻射輸送を解く ARGOT プログラムを GPU・FPGA を協調動作させる手法について述べた．ARGOT プログラムの主要演算部分である ARGOT 法と ART 法を GPU と FPGA に適材適所

表 4: 本稿における ART 法と [8] で報告されている ART 法との実行時間 (単位:秒) の比較.

Mesh size	[8] or New ART	CPU (14C)	CPU (28C)	P100	FPGA
(16, 16, 16)	[8]	0.052	0.075	0.055	0.00455
	New ART	0.085	0.12	0.065	0.01839
(32, 32, 32)	[8]	0.293	0.254	0.095	0.040
	New ART	0.395	0.351	0.122	0.145
(64, 64, 64)	[8]	2.13	1.64	0.358	0.335
	New ART	2.46	2.27	0.551	1.37
(128, 128, 128)	[8]	31.3	18.1	2.67	2.63
	New ART	42	26	4.25	9.68

的に機能分散して ARGOT コードを最適化する．提案手法を評価したところ，ART 法を FPGA にオフロードした実行である，ARGOT(GPU) / ART(FPGA) は，ARGOT(GPU)/ART(GPU) と比較して，最大 10.4 倍の性能向上を達成できた．これは，P100 GPU にとってこれらのメッシュサイズが小さすぎるため，3584 CUDA Core に対して十分な演算の並列度が得られないことに起因している．そのため，メッシュサイズが大きくなるにつれて，ARGOT(GPU) / ART(GPU) の性能は向上し，メッシュサイズ 128^3 においては，ARGOT(GPU) / ART(FPGA) の 1.94 倍の性能となることが確認できた．

また，ART 単体の性能を評価したところ，[8] の FPGA 版 ART と比べて本稿における FPGA 版 ART は 3〜4 倍の性能低下が発生していることが分かった．これは，[8] の FPGA 版 ART の Initiation Interval (II) が 1 なのに対し，本稿における FPGA 版 ART の II が 4 であることに起因している．ただしこれは OpenCL コンパイラの解析能力不足によるところが大きく，適切なワークアラウンドを講じれば II が 1 になり，性能が元通りになると期待される．

謝辞 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] Hanawa, T., Kodama, Y., Boku, T. and Sato, M.: Interconnection Network for Tightly Coupled Accelerators Architecture, *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pp. 79–82 (online), DOI: 10.1109/HOTI.2013.15 (2013).

[2] Kuhara, T., Tsuruta, C., Hanawa, T. and Amano, H.: Reduction calculator in an FPGA based switching Hub for high performance clusters, *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4 (online), DOI: 10.1109/FPL.2015.7293985 (2015).

[3] Tsuruta, C., Miki, Y., Kuhara, T., Amano, H. and Umemura, M.: Off-Loading LET Generation to PEACH2: A Switching Hub for High Performance GPU Clusters, *SIGARCH Comput. Archit. News*, Vol. 43, No. 4, pp. 3–8 (online), DOI: 10.1145/2927964.2927966 (2016).

[4] Liu, Z.: *Multiphysics in Porous Materials* (2018).

[5] 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).

[6] 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.

[7] 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).

[8] 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).

[9] Overview: Intel FPGA SDK for OpenCL, https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[10] Kobayashi, R., Fujita, N., Yamaguchi, Y., Nakamichi, A. and Boku, T.: GPU-FPGA Heterogeneous Computing with OpenCL-Enabled Direct Memory Access, *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 489–498 (online), DOI: 10.1109/IPDPSW.2019.00090 (2019).

[11] Kobayashi, R., Oobata, Y., Fujita, N., Yamaguchi, Y. and Boku, T.: OpenCL-ready High Speed FPGA Network for Reconfigurable High Performance Computing, *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, New York, NY, USA, ACM, pp. 192–201 (online), DOI: 10.1145/3149457.3149479 (2018).