Vol.2019-HPC-172 No.11
2019/12/18

GPU-FPGA 協調プログラミングを実現するコンパイラの開発

綱島隆太[1,a)]　小林諒平[2,1]　藤田典久[2]　中道安祐未[1]　朴泰祐[2,1]　Lee, Seyong[3]　Vetter, Jeffrey[3]
村井均[4]　佐藤三久[4]

概要：近年，高性能コンピューティング（HPC : High Performance Computing）分野におけるトップレベルのマシンには，アクセラレータを搭載した大規模計算クラスタが多く含まれている．高い演算性能とメモリバンド幅を有する Graphics Processing Unit（GPU）がアクセラレータとして主に用いられているが，条件分岐が頻出する処理や多数の演算コアを活用できないような並列性の小さい処理といった GPU の不得手する演算は依然として存在し，それが性能向上の妨げとなっている．このような問題に対し，任意の論理回路をプログラム可能な集積回路である Field Programmable Gate Array（FPGA）に，GPU が不得手とする処理を実行する回路を実装し，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アプローチを我々は試みている．しかしながら， GPU と FPGA の演算カーネルは，それぞれ CUDA と OpenCL といった異なるプログラミング言語で開発する必要があり，このようなマルチリンガルプログラミングは，ユーザーにとって多大な負担となる．そこで本研究では，GPU と FPGA が搭載された計算機システム上にて，両アクセラレータの統合的な制御を可能にする OpenACC を用いたプログラミング環境について検討する．本報告では，OpenACC を用いて記述された一つのプログラムを GPU 用，FPGA 用コンパイラそれぞれに向けたファイルに分割するソース to ソースコンパイラを開発し，最終的にこれらをリンクした単一の実行ファイルにより，両アクセラレータの連携が実現できるか検証を行った．その結果，開発したコンパイラによって，統一したアプリケーションプログラミングインターフェイス（API）で書かれた一つのプログラムから，CPU，GPU，FPGA で連携して演算を行う単一の実行ファイルが生成され，両アクセラレータの連携が実現できることが確認された．

1. はじめに

大規模並列計算システムを用いて科学計算やシミュレーションなどの高速演算を行う HPC 分野では，CPU の他にアクセラレータを搭載することがある．世界的なスーパーコンピュータの性能ランキングである TOP500 の 2019 年 6 月版では，上位 10 機のうち 7 機のシステムでアクセラレータが利用されている [1]．アクセラレータには主に Graphics Processing Unit (GPU) が利用されている．

GPU は本来，画像を描画する際に必要な計算処理を行うプロセッサであるが，CPU に比べて非常に高い並列処理演算性能と電力効率を持っていることから，GPU を様々な汎用計算に用いる General Purpose GPU（GPGPU）がさかんに行われている．しかし，GPU には，条件分岐が頻出したり GPU の多数の演算コアを有効活用できないような部分的なデータ並列性が低下する場合，さらにデータ交換のために GPU 間でノードを跨ぐ通信が発生する場合では，通信レイテンシが増加したり，通信のたびにカーネルとホストのスイッチングが発生するといったデメリットが存在し，これらはアプリケーションの性能劣化の原因となる．

その一方で，近年はアクセラレータとして Field Programmable Gate Array (FPGA) が注目されている．FPGA は，内部の論理回路を再構成可能なハードウェアであり，何度でも回路を書き換えられる．FPGA の利点として以下のようなものが挙げられる．

- アプリケーションに適した回路を自由に設計可能
- パイプライン処理によって高速化が可能
- 実装依存であるが，GPU より省電力化が可能

さらに近年では，FPGA 同士の接続を可能とする光リンクを持つハイエンドモデルも存在し，CPU を介さずに，直接かつ高速にそれらの間での通信を可能とするものも存在す

[1] 筑波大学 システム情報工学研究科
[2] 筑波大学 計算科学研究センター
[3] Oak Ridge National Laboratory
[4] 理化学研究所 計算科学研究センター
[a)] tsunashima@hpcs.cs.tsukuba.ac.jp

ⓒ 2019 Information Processing Society of Japan

る．デメリットとして，Verilog HDL 等のハードウェア記述言語でのコーディングが一般ユーザにとって困難であること，またコンパイル時間が極めて長い（数時間から数日かかる場合すらある）ことが挙げられる．さらに，アプリケーションに利用できるリソースが限られることが HPC への適用上の大きな問題である．これは，FPGA がノイマン型アーキテクチャのようにメモリにプログラムを格納して，行える処理を動的に制御できるプログラミングモデルではなく，計算に必要な全回路が展開され，大きな論理回路空間を必要とすることによる．しかし，回路規模の大きな FPGA が登場してきているため，以前と比較して柔軟な回路設計が可能となっている．

GPU の不得意な計算を FPGA に行わせることで，実アプリケーションのさらなる高性能化が期待できる．そこで，我々は新しいタイプの PC クラスタとして GPU と FPGA を組み合わせたマルチアクセラレータシステムを提案し，GPU-FPGA 協調計算の実現を目指して研究を進めている [2]．

しかし，現状では GPU と FPGA のプログラムはそれぞれ異なるプログラミング環境で開発しなければならない．最も多用されている NVIDIA GPU のプログラミングには CUDA[3] が利用されるのが一般的である．一方で FPGA プログラミングには，Verilog HDL，VHDL といったハードウェア記述言語（HDL: Hardware Description Language）を用いた開発に加え，C++や OpenCL[4] を用いて論理回路を生成する高位合成を行う環境が提供されている．高位合成を FPGA ベンダーが提供していることにより，従来であれば容易でなかったアプリケーション開発者による FPGA の論理回路の実装が現実的になってきている．ただし，現在の開発環境で効果的な GPU-FPGA 協調計算を実現するためには，これらの全く異なる複数の開発環境を組み合わせなければならず，ユーザーにとって非常に負担が大きい．

以上の問題から，本研究では，GPU と FPGA の両方を搭載した計算機において，両アクセラレータ上で実行する処理を，OpenACC のみを用いて記述できるプログラミング環境について検討する．OpenACC は，ディレクティブ形式のプログラミング環境を提供するため，ユーザにとっては上記の開発環境に比べて記述が容易である．本研究ではバックエンドのコンパイラとして，GPU には PGI Compiler，FPGA には米国 Oak Ridge National Laboratory（ORNL）の OpenARC[5] をそれぞれ用いる．OpenARC は OpenACC から FPGA 向け OpenCL コードを出力する，ソース to ソースのコンパイラである．本研究を実現するうえで以下の課題が挙げられる．

- オフロード部を処理するデバイスの明示手法
- 一つのプログラムから各デバイス向けコンパイラへのコード分割用トランスレーター
- マルチデバイス同時実行を行う単一実行ファイルの生成

これまでの報告 [6] では，それぞれ別々に記述された GPU と FPGA 向けの OpenACC プログラムをコンパイル時にリンクし，連携動作の検証を行った．これにより，OpenACC を用いた GPU-FPGA 協調計算が実現可能であることを確認している．本報告では，GPU-FPGA 協調計算を OpenACC により記述された一つのプログラムにより実現するためのプログラミング環境を提案し，各デバイス向けバックエンドコンパイラへのコード分割用トランスレーターとして機能するソース to ソースコンパイラの開発について述べる．

2. 関連研究

[7] では，筑波大学計算科学研究センター（CCS）で GPU 間直接通信機構として提唱された TCA（Tightly Coupled Accelerators）アーキテクチャ [8] を実装した PEACH2（PCI Express Adaptive Communication Hub version 2）による通信処理のオフロードを行っている．PEACH2 は GPU に足りない通信処理を行う FPGA ボードであり，異なるノード間の GPU 同士を直接接続できるようになっている．これによって，ホストの CPU とメモリを介さずに通信を行えるため，転送速度が改善される．Tsuruta らは本来通信機構としてしか用いられていない PEACH2 に追加要素として演算機構を構成し，ノード間通信を行う際に CPU で行う処理のオフロードを行った．N 体シミュレーションを対象に，GPU 間通信の前処理となるツリー法の枝刈り部分を FPGA で処理することにより，CPU の約 7.2 倍の高速化を達成した．

[9] では，OpenCL を用いて FPGA 上での Authentic Radiation Transfer（ART）法の最適化を行っている．その結果，OpenMP を使用した CPU 実装と比較して 6.9 倍高速な性能を達成している．複数の FPGA を使用し並列化した場合は，GPU を超える性能を発揮すると考えられる．[10] では，OpenCL を用いて実装した疎行列数値計算の性能評価と最適化方法について評価している．この中で FPGA 向けのプログラミングでは GPU とは異なる最適化を行う必要があると述べられている．[11] では，FPGA 上に津波シミュレーションの専用回路を構築し，性能評価を行っている．これにはストリーム計算用高位合成コンパイラである SPGen を用いている．

小林らは，GPU-FPGA 複合システムにおけるアクセラレータ間の連携機構ついて検証を行った [12]．計算と通信処理を FPGA で行い協調計算を行わせるため，GPU-FPGA 間直接通信の性能を評価した．結果，従来この通信の起動は CPU が行わなくてはならなかったが，FPGA が主体的に通信を起動できることが確認された．

これらの研究では FPGA を GPU の通信補助手段，あ

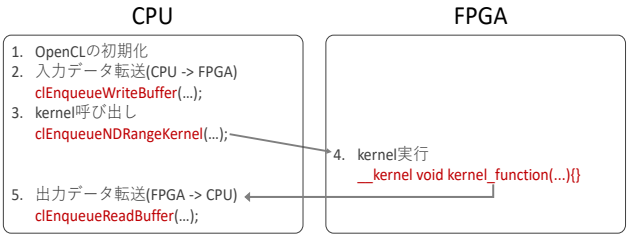

図 1　OpenCL プログラミングの流れ

るいは FPGA のみの演算と通信の融合を試みているが，GPU と FPGA の協調動作を統一したインターフェイスによって記述するためのプログラミング手法については検討されていない．

AiS（Accelerators in Switch）[13] は FPGA を演算と通信に積極的に利用するコンセプトであり，GPU による高速演算を組み合わせることで，理想的な高性能並列処理システムが構築できると考えられる．例えば，通信の遅延を隠蔽するために通信中に FPGA 上で演算を行ったり，GPU の苦手な処理を FPGA が行うことで高速化を図ったりする連携が考えられる．しかし現状，FPGA と GPU の協調計算を効果的に行うためのプログラミングには以下のような課題がある．

- 開発に用いられるプログラミング言語がアクセラレータごとに別々である
- どちらの言語も互換性が無いため移植性が低い
- 単一のプログラムで異なる複数のデバイスを一度に用いることができない
- メモリアクセスやカーネルの並列度などコーディングにおいて考慮すべきことが多い

3. 既存のプログラミング環境

3.1 OpenCL

一般的に FPGA のプログラミングは Verilog HDL や VHDL を用いて行われる．これらの Hardware Description Language (HDL) は，論理回路の動作を定義するために設計された専用言語で，プログラミング言語に似た構文や表記法を用いて，回路に含まれる素子の構成やそれぞれの動作条件，素子間の配線などの記述が可能となっている．しかし，HDL を用いたプログラミングでは回路動作をクロック単位で定義する必要があるため，記述が複雑になり動作検証も困難となる．ハードウェアの素養を持たない開発者にとって HDL で目的の処理を記述することは難しく，HDL を用いた FPGA のプログラミングコストは高い．

OpenCL は，ヘテロジニアスな並列計算機環境に適した並列プログラミングのための言語である．近年では OpenCL を用いた高位合成による FPGA のプログラミング環境が提供されている．OpenCL はハードウェアに近いレベルで API を共通化しており，高度な抽象化が行われていないので，実際に利用する演算プロセッサの特徴に適したパフォーマンスチューニングが可能となっている．OpenCL は C 言語ベースでデバイスコードを実装できるため，HDL を用いた回路生成に比べ，実装コストを低減させることができる．OpenCL プログラミングの流れを**図 1** に示す．OpenCL プログラミングでは，CUDA と同様にホストコードとデバイスコードの 2 種類のプログラムが存在する．ホストコードは，CPU 上で動作し FPGA や GPU といったデバイスを制御するプログラムである．デバイスコードは，デバイス上で動作するプログラムである．しかし，OpenCL では，ホストコードとデバイスコードは別ファイルに記述し，別々にコンパイルする．ホストコードは gcc や Intel Compiler などの C コンパイラでコンパイルし，デバイスコードは専用コンパイラでコンパイルされ論理合成可能なファイルに変換される．なお，OpenCL は GPU でも利用できるものの，HPC で最も利用されている NVIDIA 社製 GPU の演算記述に最適化されているわけではないため，HPC アプリケーションにおいては，CUDA が GPU プログラミング環境のデファクトスタンダードとなっている．また，OpenCL のほうがホストコードの記述が複雑であるため，OpenCL による統一的な記述ではアプリケーション開発者にとってデメリットが多い．

3.2 GPU と FPGA のコンパイル環境の組み合わせ

我々は，GPU と FPGA の計算が混在したプログラムの実装，協調計算が可能であることを確認している [14]．これは，CUDA と OpenCL で記述したプログラムを分割コンパイルし，リンクすることで一つのプログラムとして実行できるようにしている．しかし，現状では GPU と FPGA で用いるプログラミング言語が異なるため，複数言語で記述しなければならず，実装が極めて難しい．CUDA と OpenCL ではプログラミングの作法が異なる部分が存在する．カーネルの並列度の指定方法が異なっており，また，OpenCL では clSetKernelArg() をカーネル引数毎に実行する必要があるため，同じように記述出来ずバグを発生させる原因になる．さらに，2 章で述べたとおり，GPU と FPGA ではプログラミングにおける最適化手法が異なる．FPGA ではリソースが限られているため，GPU のような非常に高い並列度を持ったプログラムの実行は向いていない．したがって，これらのアクセラレータの処理を統一 API により記述できるプログラミング環境を用意する必要がある．

3.3 OpenACC

OpenACC[15] は複数のアクセラレータ向け API である．ディレクティブ形式の言語拡張規格であり，計算科学アプリケーションが標準的に利用する C，C++，Fortran の各言語に対応している．利点として以下が挙げられる．

ⓒ 2019 Information Processing Society of Japan

```
1．転送するデータの指定
   #pragma acc data copyin(in1[0:N], in2[0:N])
   copyout(out[0:N]) {
2．カーネル実行対象の指定
   #pragma acc kernels
3．個々のループに対する指定
   #pragma acc loop independent
       for (i = 0; i < N; i++) {
           out[i] = in1[i] * in2[i];
       }
   }
```

図 2　OpenACC のコード例

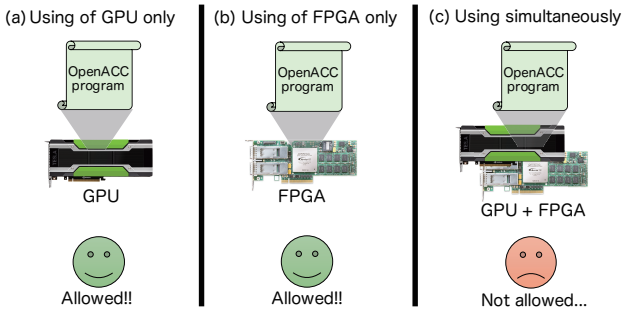

図 3　OpenACC コンパイラの記述対応状況

- 指示文の挿入のみでアクセラレータでの演算が可能
- CUDA と比べ抽象化されておりプログラミングが容易で移植性も高い

OpenACC では，CUDA や OpenCL のようにホストコードとデバイスコードという区別は無い．オフロードさせたい部分にディレクティブの挿入を行うだけで済むため，処理を別々のコードとして分けて記述する手間を省くことができる．

本研究では，OpenACC により，GPU-FPGA 協調計算を実現する開発環境の実現を目指す．

OpenACC のコード例を図 2 に示す．まず，`#pragma acc data` で転送するデータを指定する．`copyin()` でホストからアクセラレータへ転送するデータを，`copyout()` でアクセラレータからホストへ転送するデータを指定する．転送するタイミングはディレクティブを挿入した位置となる．次に，`#pragma acc kernels` でアクセラレータが処理を行う領域を指定する．なお，この指示文のみでも OpenACC のプログラムはアクセラレータでの演算実行を行えるが，指定されていない処理はコンパイラ依存となり，特にアクセラレータとホスト間のデータ転送でオーバーヘッドが発生する可能性があるため，通常は`#pragma acc data` も挿入する．`#pragma acc loop` は個々のループに対する指示を行う．`independent` は，ループにデータ依存性が無いことを示す節である．これを指定しない場合，コンパイラの判断により適切な並列化が行えない場合がある．

3.4　各アクセラレータ向けの OpenACC コンパイラ

GPU 向けの OpenACC コンパイラには，PGI Compiler[16], GCC, Cray Compiler, OpenARC などが存在する．その中でも，PGI Compiler は NVIDIA 傘下の PGI 社が開発するコンパイラであり，NVIDIA 製 GPU に特に最適化されている．

一方で，FPGA に対応した OpenACC コンパイラは市場に存在しない．しかし，研究開発中のものとして OpenARC が存在する．OpenARC は ORNL が開発しているソース to ソースコンパイラであり，OpenACC を FPGA 向け OpenCL へ変換する機能を備える．現在，対応言語は入力が C のみであり，出力のホストコードは C++になる．本研究では，筑波大学計算科学研究センター及び理化学研究所計算科学研究センターと ORNL との共同研究により GPU-FPGA 協調計算環境の実現に OpenARC を利用する．

OpenARC の OpenACC は，FPGA に限らず，GPU やメニーコア CPU 向けのプログラムも出力できる．NVIDIA 製 GPU 向けのコンパイルでは，OpenACC のコードを CUDA へ変換する．しかし，現状では，異なる複数のアクセラレータを同時に用いたプログラムの記述は不可能である．また，OpenACC ディレクティブでは規格上，演算加速したいコード範囲に対してこれを実行する複数のアクセラレータを指定できるディレクティブは存在せず，異なる複数のアクセラレータを一度に利用することはできない．device_type 節というアクセラレータ名を指定できるオプションは存在するが，これは実行するアクセラレータごとに異なる並列実行特性（gang, vector, seq など）を指定するためのものである．また，PGI Compiler では PGI Unified Binary を生成することで演算を行うプロセッサをランタイム関数によって切り替えること自体は可能だが，CPU と NVIDIA GPU にしか対応しておらず，現在，異なる複数のアクセラレータの搭載されたマシンで実行中にオフロード先を切り替えられる実装は存在しない．したがって，図 3 のように，単一のアクセラレータ向けのコードを記述することは可能だが，一つのプログラム内で異なる複数のアクセラレータを使用する書き方はできない．協調計算を実現する場合，FPGA と GPU それぞれのプログラムの記述を行い，分割コンパイルした上でオブジェクトファイルをリンク，グルーコード（互換性の無い部分同士を結合するためだけに働くコード）で繋ぐ必要がある．これはユーザーにとって負担となるため，一つのプログラム内で協調計算を記述可能にし，ユーザの利便性を高める必要がある．

4.　統一インターフェイスによる協調計算記述を可能とするプログラミング環境の提案

本研究では，図 4 に示すコンパイル手法を提案する．まず，OpenACC により GPU，FPGA それぞれの処理を含

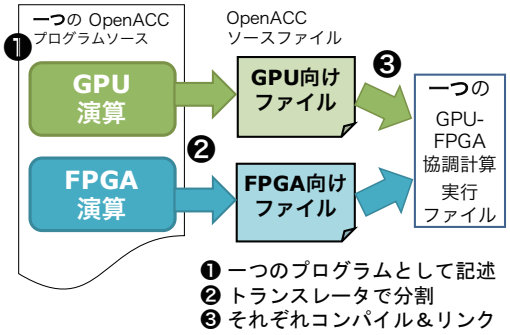

図 4　提案するプログラミング環境の概要

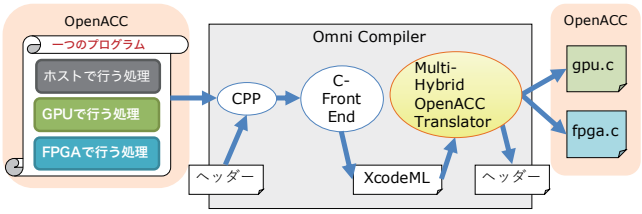

図 5　今回開発したコンパイラの処理の流れ

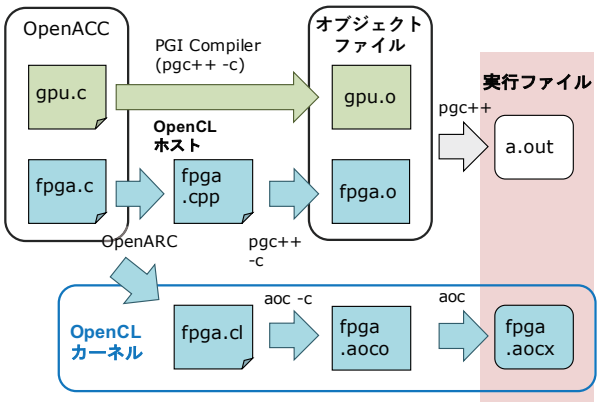

図 6　バックエンドのコンパイル&リンクの流れ

めたプログラムを記述する．次に，記述したプログラムを既存のコンパイラでコンパイルできるように，それぞれのアクセラレータ向けにファイル分割する．このファイル分割の処理は OpenACC の入力から OpenACC のファイルを出力するソース to ソースコンパイルとする．分割されたファイルはそれぞれのデバイスに対応する OpenACC コンパイラで分割コンパイルを行い，オブジェクトファイルをリンクすることで一つのプログラムとして実行できるようにする．

これまでの報告 [6] で，トランスレーターの出力を想定して記述された GPU，FPGA 向けの 2 つの OpenACC プログラムを用い，バックエンドのコンパイルとリンクが可能であることを確認している．バックエンドコンパイラの処理の流れを図 6 に示す．GPU 向けには PGI Compiler，FPGA 向けには OpenARC で OpenACC のコードを分割コンパイルし，それぞれで出力されたオブジェクトファイルをリンクすることで，OpenACC のみの記述で 1 つのプロセスから GPU と FPGA の両者に演算をオフロードするプログラムを記述できる．OpenARC でコンパイル後に出力される FPGA 向け OpenCL プログラムは，Intel 社が提供している Intel FPGA SDK for OpenCL[17] に付属する，OpenCL C で記述されたコードを FPGA の回路情報（ビットストリーム）に変換するコンパイラである aoc コマンドを用いてコンパイルする．実行時に生成した回路情報を FPGA に転送し，演算を行う．

```
1  void fuga() {
2  #pragma accomn ondevice(FPGA)
3  {
4  #pragma acc data copy(a, b)
5  {
6  #pragma acc kernels loop independent
7          for(i=0; i < N; i++)
8                  a[i] = a[i] + b[i];
9  }
10 }
11 }
```

図 7　accomn ondevice の使用例

本章では，図 6 の一番左にある 2 つのファイルを 1 つの OpenACC プログラムから生成するコンパイラについて述べる．この変換を行うために，理化学研究所計算科学研究センターと筑波大学 HPCS 研究室で共同研究を行っている，OpenACC に対応したコンパイラである Omni Compiler[18] を拡張開発し，専用のソース to ソースコンパイラを追加した．Omni Compiler は大規模並列計算機で利用することを想定した XcalableMP[19]，XcalableACC[20]，OpenACC といった指示文を含むコードを対象としたコンパイラであり，XcalableMP 及び XcalableACC のリファレンスコンパイラでもある．現状 Omni Compiler で OpenACC をコンパイルした場合，CUDA または PEZY-SC プロセッサ向け OpenCL 拡張である PZCL を出力する．しかし，FPGA 向け OpenCL へのコンパイルには対応していない．また，最新の GPU アーキテクチャには対応しておらず，PGI Compiler と同程度に最適化されたコードを生成可能にすることは容易ではない．よって，本研究では Omni Compiler をバックエンドコンパイラとして使用していない．以下，開発したコンパイラのうちの本手法のコアな部分である，入力ファイルを解析し各アクセラレータ向けの 2 つのソースファイルに分割する部分を Multi-Hybrid OpenACC Translator (MHOAT) と呼ぶ．MHOAT は Omni Compiler の既存の OpenACC 実装及び OpenARC の入力が C のみに対応することから C を対象としている．

2 つのファイルに分割するに当たり，OpenACC で演算加速するコード範囲をどのアクセラレータで処理するのかを明示する必要がある．しかし，ランタイム関数で対応すると

ディレクティブと異なり可搬性が低くなるため，OpenACC で記述するメリットが少なくなってしまう．そこで，OpenACC の独自拡張ディレクティブとして`#pragma accomn ondevice(DEVICE)` を定義する．”accomn” は通常の OpenACC ディレクティブを示す ”acc” に Omni Compiler 独自拡張であることを示すために ”omn” を追記したものである．DEVICE に入力できる値は現時点では enum 型の `GPU` または `FPGA` となっており，どちらか一方のみ指定できる．このディレクティブの使用例を**図 7** に示す．注意点として，現段階のトランスレーターでは必ずこのディレクティブを main 関数以外の関数の中に記述する必要がある．ここでアクセラレータごとに処理の関数化が必要なのは，分割したファイルをオブジェクトファイルにしてリンクする際にシンボルを結び付けなければならないためである．トランスレーターは変換されたソースコードをディレクティブの書かれた関数ごとに別々のファイルに書き込む．この拡張ディレクティブも通常の OpenACC と同様に演算加速する範囲を波括弧 ”{”, ”}” で囲う．なお，MHOAT は現在開発中であり，現状では OpenACC の全ての書き方には対応していない．for 文の直上のディレクティブには波括弧は不要であるが，それ以外は図中のように連続した行に書く場合でも必ずディレクティブに波括弧が必要である．

開発したコンパイラの処理の流れを**図 5** に示す．Omni Compiler では，まず C Preprocessor（CPP）でヘッダーファイルのインクルードやマクロの展開などを行う．次に C-FrontEnd により C から XcodeML と呼ばれる中間コード形式の XML ファイルに変換される．このファイルを MHOAT で読み込み，解析して 2 つの OpenACC プログラムを出力する．なお，この際にヘッダーから読み込まれたコードを全て削除（一部コメントアウト）した上で，`#include` 指示文を書き戻している．これは OpenARC が図 6 のように入力の C プログラムから OpenCL ホストコードとして C++を出力するため，リンクの際に GPU 向けのファイル，FPGA 向けのファイルのどちらも C++としてコンパイルすることでマングリング問題を回避していることによる．Omni Compiler で実行している CPP は C 用のヘッダーを読み込んでおり，そのまま C++コンパイラに通すとエラーが発生してしまう．また，main 関数や拡張ディレクティブで囲われていないコードは OpenARC が C++に変換する影響を考慮し，全て GPU 向けのファイルに書き込むようにしている．

5. 提案手法の検証

開発中の MHOAT 及び [6] で報告したバックエンドコンパイラの組み合わせによって，統一された API で一つのプログラムとして GPU+FPGA 連携を記述したコードを実際に実行可能ファイルに変換し，GPU-FPGA 協調計算が正しく実行できるか検証を行った．MHOAT は現在開発中

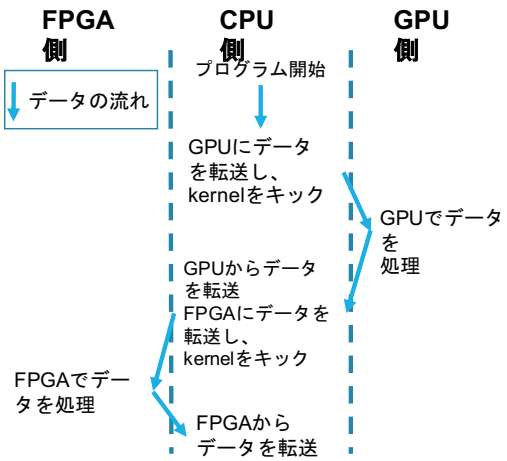

図 8 検証プログラムの処理の流れ

のため，ソースコードを変換して分割ファイル化した結果が，原理的に動作するかを確かめた．

検証に使用したプログラムの動作を**図 8** に示す．プログラムは大きく分けてホストで実行するコードブロック，GPU で実行するコードブロック，FPGA で実行するコードブロックに別れている．まず，ホストでデータの初期値をセットし，GPU で処理を実行する．その後，データを FPGA に渡して FPGA で処理をしたあと，ホストにデータを返し，結果を出力する．なお，データは GPU から FPGA へ直接転送されるわけではなく，GPU から一旦ホストのメモリに転送された後，FPGA に転送される．

プログラムは C で記述した．コンパイル前のプログラムを**図 9** に示す．なお，ここでは問題を単純化するため，複雑な処理はせずに検証しやすいコードとした．GPU ではベクトル加算を行い，FPGA では GPU から転送されたデータに対してベクトル加算を実行する．また，計算結果は CPU 上で計算した結果と GPU，FPGA から返ってきた値が同じであるか，それぞれ比較を行う．また，現状では**図 10**, **図 11** のようにグローバル宣言はヘッダーで行う必要がある．

検証は，2019 年 4 月より筑波大学計算科学研究センター（CCS）で稼働しているスーパーコンピュータ，Cygnus[21]（**表 1**）上で行った．なお，Cygnus には GPU のみが搭載されているノードと GPU と FPGA が混載されているノードが存在するが，今回は GPU と FPGA が両方搭載されている 1 ノード（**図 12**）の CPU0 及び CPU0 に接続されている GPU，FPGA をそれぞれ 1 台ずつ用いた．

MHOAT の出力ファイルの内容を FPGA 側を**図 13** に，GPU 側を**図 14** にそれぞれ示す．図 13 では，`#include` 指示文と，図 9 で`#pragma accomn ondevice(FPGA)` の記述されていた関数のみが出力されていることがわかる．なお，`# 1 "each_block.c"`などは CPP を通したときに追記される linemarker で，MHOAT 内でも行の関係やそれらの

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "acc_ondevice.h"
4  #include "each_block_global_decl.h"
5  int main(int argc, char** argv) {
6      int workers = 16;
7      int gangs = 256;
8      int size = workers * gangs;
9      float* A = (float*) malloc(size * sizeof(float));
10     float* B = (float*) malloc(size * sizeof(float));
11     float* D = (float*) malloc(size * sizeof(float));
12     float* E = (float*) malloc(size * sizeof(float));
13     int i, error=0;
14
15     for (i = 0; i < size; i++) {
16         A[i] = (float) i;
17         B[i] = (float) i * 100;
18     }
19     funcGPU(A, B, D, size);
20     for (i = 0; i < size; i++) {
21         if (D[i] != (float) i + (float) i * 100) error++;
22     }
23     printf("errorGPU:%d\n", error);
24     funcFPGA(D, E, size);
25     for (i = 0; i < size; i++) {
26         if (E[i] != (float) i + (float) i * 100 + (float) i)
               error++;
27     }
28     printf("errorFPGA:%d\n", error);
29     return 0;
30 }
31
32 void funcGPU(float* a, float* b, float* d, int size) {
33 #pragma accomn ondevice(GPU)
34 {
35     int j;
36 #pragma acc data copyin(a[0:size], b[0:size]) copyout(d[0:
       size])
37 {
38 #pragma acc kernels loop independent gang worker(16)
39         for (j = 0; j < size; j++) {
40             d[j] = a[j] + b[j];
41         }
42 } } }
43
44 void funcFPGA(float *a, float *b, int size) {
45 #pragma accomn ondevice(FPGA)
46 {
47     int j;
48 #pragma acc data copyin(a[0:size]) copyout(b[0:size])
49 {
50 #pragma acc kernels
51 {
52 #pragma acc loop independent
53     for (j = 0; j < size; j++) {
54         b[j] = a[j] + (float)j;
55     }
56  } } } }
```

図 9　コンパイル前のプログラム

```
1  enum DEVICE{
2      GPU,
3      FPGA
4  };
```

図 10　デバイスの記述された acc_ondevice.h

```
1  void funcGPU(float* a, float* b, float* d, int size);
2  void funcFPGA(float *a, float *b, int size);
```

図 11　グローバル宣言をするヘッダー (each_block_global_decl.h)

表 1　評価環境

CPU	Intel Xeon Gold 6126 (12C / 2.6GHz) x2
GPU	NVIDIA Tesla V100 (32GiB HBM2 PCIe 3.0 x16) x4
FPGA	Intel Stratix 10 GX 2800 (BittWare 520N[22] PCIe Gen3 x16) x2
OS	CentOS 7.3
GPU compiler	PGI Compiler 19.1
FPGA compiler	OpenARC V0.17 (Oct, 2019)
OpenCL compiler	Intel FPGA SDK for OpenCL 19.1.0.240

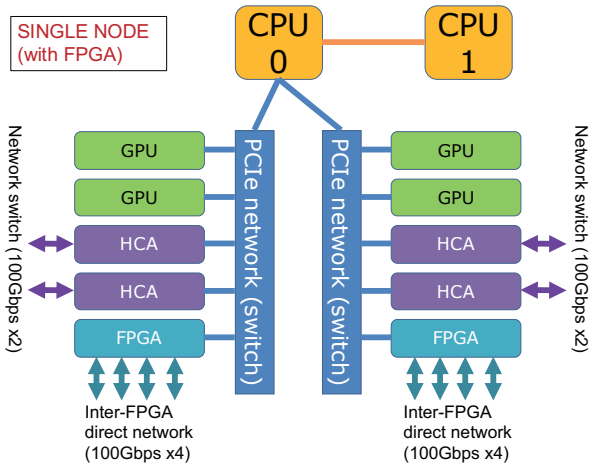

図 12　Cygnus の GPU + FPGA 混載ノードの構成

解析に用いる．バックエンドコンパイラでは再び CPP に掛けられ，これらは無視されるため残っていても問題ない．また，コンパイラの情報と MHOAT の処理の都合上一部削除出来ず残ったヘッダーのコードがコメントアウトされている．これらについてもバックエンドコンパイラでは無視されるため問題ない．図 14 はページに収めるため，コメント行と linemarker を省略し，連続する波括弧を一部一行にまとめて載せている．こちらでは，`#pragma accomn ondevice(GPU)` の記述されていた関数の他，main 関数も出力されている．

これらを図 6 に示したバックエンドコンパイラでコンパ

ⓒ 2019 Information Processing Society of Japan

```
1  /*
2   * Original Source : each_block.c
3   * Language : C
4   * Compiled Time : 2019-11-15 16:14:30
5   * Compiler Info : AccHybridTranslator
6   * Compiler Version : XcodeML/C-FrontEnd 1.3.0
7   */
8  /* 一部のヘッダー読み込み部のコメントアウト（省略）
9  */
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include "acc_ondevice.h"
13 #include "each_block_global_decl.h"
14 # 1 "each_block.c"
15 # 2 "each_block.c" 2
16 # 3 "each_block.c" 2
17 # 4 "each_block.c" 2
18 # 5 "each_block.c" 2
19 # 57 "each_block.c"
20 void funcFPGA(float * a, float * b, int size)
21 {
22 # 61 "each_block.c"
23 ;
24 {
25 int j;
26 # 64 "each_block.c"
27 ;
28 #pragma acc data copyin ( a [ 0 : size ] ) copyout ( b [ 0
       : size ] )
29 {
30 # 66 "each_block.c"
31 ;
32 #pragma acc kernels
33 {
34 # 68 "each_block.c"
35 ;
36 #pragma acc loop independent
37 # 68 "each_block.c"
38 for(j = (0); j < size; j++) {
39 {
40 # 70 "each_block.c"
41 (*(b + j)) = ((*(a + j)) + ((float)(j)));
42 }
43 }
44 }
45 }
46 }
47 }
```

図 13　FPGA での演算

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "acc_ondevice.h"
4  #include "each_block_global_decl.h"
5  int main(int argc, char * * argv)
6  {
7  int workers = 16;
8  int gangs = 256;
9  int size = workers * gangs;
10 float * A = (float * )(malloc(size * (sizeof(float))));
11 float * B = (float * )(malloc(size * (sizeof(float))));
12 float * D = (float * )(malloc(size * (sizeof(float))));
13 float * E = (float * )(malloc(size * (sizeof(float))));
14 int i;
15 int error = 0;
16 for(i = (0); i < size; i++) {
17 {
18 (*(A + i)) = ((float)(i));
19 (*(B + i)) = (((float)(i)) * (100));
20 }
21 }
22 funcGPU(A, B, D, size);
23 # 26 "each_block.c"
24 for(i = (0); i < size; i++) {
25 {
26 {
27 if((*(D + i)) != (((float)(i)) + (((float)(i)) * (100)))) {
28 error++;
29 }}}}
30 printf("errorGPU:%d\n", error);
31 funcFPGA(D, E, size);
32 for(i = (0); i < size; i++) {
33 {
34 {
35 if((*(E + i)) != ((((float)(i)) + (((float)(i)) * (100))) +
       ((float)(i)))) {
36 error++;
37 }}}}
38 printf("errorFPGA:%d\n", error);
39 return 0;
40 }
41 void funcGPU(float * a, float * b, float * d, int size)
42 {
43 ;
44 {
45 int j;
46 ;
47 #pragma acc data copyin ( a [ 0 : size ] , b [ 0 : size ] )
       copyout ( d [ 0 : size ] )
48 {
49 ;
50 #pragma acc kernels loop independent gang worker ( 16
       )
51 for(j = (0); j < size; j++) {
52 {
53 (*(d + j)) = ((*(a + j)) + (*(b + j)));
54 }}}}}
```

図 14　GPU での演算+ホストでの処理

イルし，生成された単一の実行ファイルを実行した結果，GPU，FPGA からホストへ返された値と CPU での演算結果は一致しており，正しく動作していることが確認できた．

6. まとめ

本報告では，GPU と FPGA の両方を搭載した計算機における両アクセラレータでの演算が OpenACC により統一的に記述できるプログラミング環境について提案した．提

ⓒ 2019 Information Processing Society of Japan

案手法を実現するために，一つの OpenACC プログラムに GPU と FPGA の処理を混在させるためディレクティブの定義，及びこれらを記述した一つのプログラムから，既存の GPU 向け，FPGA 向けコンパイラでそれぞれコンパイル可能なソースファイルが出力できるソース to ソースコンパイラである MHOAT を開発し，出力されたコードが正しく計算を行えるか検証を行った．その結果，両アクセラレータを一度に正しく動作させることに成功し，一つの OpenACC プログラムから GPU-FPGA 協調計算を正しく行えるコードが生成できた．以上より，GPU-FPGA 協調プログラミングが可能な統一プログラミング環境が実現できたと言える．

今回十分でなかった機能として，それぞれのアクセラレータで実行したいコードを関数化せずに書けない点，OpenACC の波括弧を省略出来ない点，グローバル宣言をヘッダーに記述しないとエラーが発生する点が挙げられる．これらを完全にサポートしたコンパイラの開発が必要である．また，現在の OpenARC では出力が C++であるにも関わらず C が入力であることから，`extern "C"`を記述することが出来ない．よって，OpenARC で C++を入力できるようにする，専用のディレクティブを用意するなど対策が必要である．さらに，本報告では，GPU-FPGA 間のデータ転送はホストのメモリを一旦介して行っていたが，我々は関連研究で述べた GPU-FPGA 間で直接通信を行う手法についても研究を進めている [12]．これを利用することでさらなる高速化が可能なことから，OpenACC からこのフレームワークを利用可能にすることも課題として挙げられる．

本報告で示したプログラミング環境は筑波大学計算科学研究センターの Cygnus をはじめとする，GPU+FPGA 混載ノードを備えた次世代ハイブリッドクラスタの資源を有効利用するためのユーザーインターフェイスとして重要な技術である．

謝辞

本研究の一部は，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）における研究課題「次世代演算通信融合型スーパーコンピュータの開発」，及び科学研究費補助金・基盤研究 (B)「再構成可能システムと GPU による複合型高性能計算プラットフォーム」（課題番号 18H03246）による．また，スーパーコンピュータ Cygnus の利用は筑波大学計算科学研究センター学際共同利用 2019 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」による．また，理化学研究所計算科学研究センターの中尾昌広博士に感謝する．

参考文献

[1] June 2019 — TOP500 Supercomputer Sites, <https://www.top500.org/lists/2019/06/>
[2] 小林諒平, 藤田典久, 山口佳樹, 朴泰祐：OpenCL と Verilog HDL の混合記述による GPU-FPGA デバイス間連携，情報処理学会研究報告，2018-HPC-167，(2018)．
[3] CUDA とは？ — NVIDIA, <https://www.nvidia.co.jp/object/cuda_whatis_jp.html>
[4] OpenCL Overview - The Khronos Group Inc, <https://www.khronos.org/opencl/>
[5] Lee, S., Lambert, J., Kim, J., Vetter, J. S., & Malony, A. D: OpenACC to FPGA: A Directive-Based High-Level Programming Framework for High-Performance Reconfigurable Computing, (2018).
[6] 綱島隆太，小林諒平，藤田典久，中道安祐未，朴泰祐：GPU-FPGA 協調計算を記述するためのプログラミング環境に関する研究，情報処理学会研究報告，2019-HPC-169，(2019)．
[7] Tsuruta, C., Miki, Y., Kuhara, T., Amano, H., & Umemura, M, : Off-loading let generation to peach2: A switching hub for high performance gpu clusters. ACM SIGARCH Computer Architecture News, 43.4: 3-8. (2016).
[8] Hanawa, T., Fujii, H., Fujita, N., Odajima, T., Matsumoto, K., & Boku, T. Evaluation of FFT for GPU cluster using tightly coupled accelerators architecture. In: Cluster Computing (CLUSTER), 2015 IEEE International Conference on. IEEE, p. 635-641. (2015).
[9] Norihisa Fujita, Ryohei Kobayashi, Taisuke Boku, Yuma Oobata, Yoshiki Yamaguchi, Kohji Yoshikawa, Makino Abe, Masayuki Umemura, : Accelerating Space Radiative Transfer on FPGA using OpenCL, Proc. of HEART2018 (Int. Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies), Toronto, (2018).
[10] 大島聡史, 塙敏博, 片桐孝洋, 中島研吾：FPGA を用いた疎行列数値計算の性能評価．情報処理学会研究報告，2016-HPC-153，(2016)．
[11] 佐野健太郎, 河野郁也, 中里直人, Alexander Vazhenin, Stanislav Sedukhin：FPGA による津波シミュレーションの専用ストリーム計算ハードウェアと性能評価，情報処理学会研究報告，2015-HPC-149（2015）．
[12] 小林諒平，阿部昂之，藤田典久，山口佳樹，朴泰祐：GPU-FPGA 複合システムにおけるデバイス間連携機構，情報処理学会研究報告，2018-HPC-165（2018）．
[13] 藤田典久, 小林諒平, 山口佳樹, 大畠佑真, 朴泰祐, 吉川耕司, 安部牧人, 梅村雅之：OpenCL を用いた FPGA による宇宙輻射輸送シミュレーションの演算加速，情報処理学会研究報告，2017-HPC-161（2017）．
[14] 中道安祐未, 小林 諒平, 藤田 典久, 朴 泰祐：GPU・FPGA 混載ノードにおけるヘテロ演算加速プログラム環境に関する研究, 情報処理学会研究報告, 2019-HPC-168 (2019).
[15] homepage — OpenACC, <https://www.openacc.org>
[16] PGI Compilers & Tools, <https://www.pgroup.com/>
[17] Intel:Intel FPGA SDK for OpenCL, <https://www.intel.co.jp/content/www/jp/ja/software/programmable/sdk-for-opencl/overview.html>
[18] Omni Compiler, <http://omni-compiler.org/>
[19] Masahiro Nakao, Hitoshi Murai, Taisuke Boku, Mitsuhisa Sato: Performance Evaluation for Omni Xcal-

ableMP Compiler on Many-core Cluster System based on Knights Landing, IXPUG Workshop Asia 2018, Tokyo, Japan, (2018).
[20] Masahiro Nakao, Tetsuya Odajima, Hitoshi Murai, Akihiro Tabuchi, Norihisa Fujita, Toshihiro Hanawa, Taisuke Boku, Mitsuhisa Sato: Evaluation of XcalableACC with Tightly Coupled Accelerators/InfiniBand Hybrid Communication on Accelerated Cluster, International Journal of High Performance Computing Applications, (2019).
[21] 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, <https://www.ccs.tsukuba.ac.jp/supercomputer/#Cygnus>
[22] 520N - BittWare FPGA Acceleration, <https://www.bittware.com/fpga/520n/>

ⓒ 2019 Information Processing Society of Japan