

OpenACC と OpenCL の混合記述による GPU-FPGA デバイス間連携

小林 諒平^{1,2} 藤田 典久^{1,2} 朴 泰祐^{1,2}

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、CUDA より抽象度を引き上げたプログラミングモデルである OpenACC と OpenCL の組み合わせにより GPU と FPGA の両演算加速デバイスを連携させ、性能向上を目指す枠組みを示す。

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 に適合するというのではなく、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、この性能向上のボトルネックを引き起こす演算を 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コンセプトを **CHARM (Cooperative Heterogeneous Acceleration with Reconfigurable Multidevices)** と呼称している。その概念実証として、我々は宇宙輻射輸送を解くシミュレーションコードにそのコンセプトを適用することによって、GPU のみを利用した実装と比較して最大 17.4 倍の性能向上を達成したことを報告している [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

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

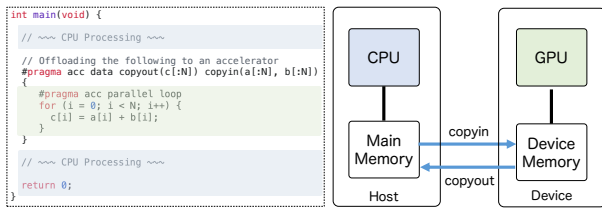


図 1: OpenACC プログラミングモデルの概要

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、アプリケーション開発者がより少ない負担で GPU-FPGA 混載クラスシステムにおける GPU と FPGA の両演算加速デバイスを連携できるようにするプログラミング環境が必要とされる。

そこで本稿では、CUDA と OpenCL との混合プログラミングよりもユーザビリティの高い GPU-FPGA 連携の実現を見据え、その予備評価として、CUDA より抽象度を引き上げたプログラミングモデルである OpenACC と OpenCL の組み合わせにより GPU と FPGA の両演算加速デバイスを連携させ、性能向上を目指す枠組みを示す。

2. 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 [2]。

図 1 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`

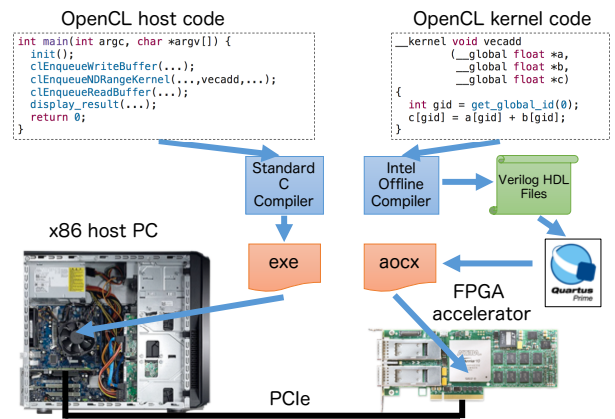


図 2: Intel FPGA SDK for OpenCL のプログラミングモデル。

`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 1 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. Intel FPGA SDK for OpenCL

Intel は OpenCL を用いて FPGA 回路を設計できる開発環境 [3] 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 2 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 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

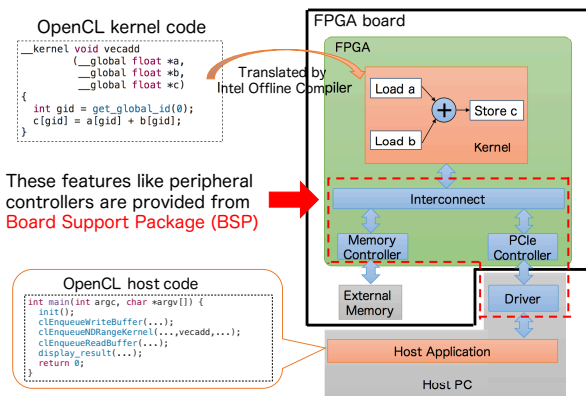


図 3: Intel FPGA SDK for OpenCL プラットフォームの構成図。

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 Verilog 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 3 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

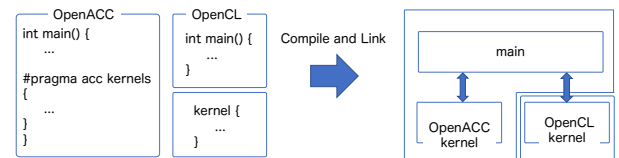


図 4: OpenACC · OpenCL 混合記述によるプログラム構成

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。

4. OpenACC · OpenCL 混合プログラミング

4.1 概要

OpenACC · OpenCL 混合プログラミングでは、第 2 章、第 3 章で述べたように GPU カーネルの記述に OpenACC を、FPGA カーネルの記述に OpenCL を用いている。そして、それは OpenACC プログラム、OpenCL プログラムと別々に動作させてそれらの間をプロセス間通信させるものではなく、1 つのホストプログラム (main 関数) から OpenACC で記述された GPU カーネル、OpenCL で記述された FPGA カーネルの両方呼び出す。

図 4 に、OpenACC · OpenCL 混合記述によるプログラム構成を示す。OpenACC で記述されたカーネルは PGI コンパイラによってコンパイルされるので、OpenACC で記述されたプログラムと OpenCL のホストプログラムのコンパイルおよびリンクには PGI コンパイラを利用し、OpenCL カーネルのコンパイルには第 3 章で述べたように Intel FPGA SDK for OpenCL が提供するオフラインコンパイラを利用している。これによって、1 つのホストプログラムから GPU と FPGA とを制御することが可能となり、GPU と FPGA を制御する CPU は共通のホストメモリ空間を利用できる。

4.2 GPU-FPGA DMA

本節では、OpenACC · OpenCL 混合プログラミングの枠組みにおいて、我々がこれまでの研究で開発した GPU-FPGA DMA 機能 [4] が利用可能であることを示す。この DMA 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ものである。これは、かつて HA-PACS/TCA の開発 [5] 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、開発した GPU-FPGA DMA 機能では

```

1 #define SIZE 1000000
2
3 tcacresult tcaCreateHandleGPU(unsigned long long *paddr,
4                               void *ptr, size_t size);
5
6 int main(void) {
7     uint32_t data[SIZE/4];
8     void* ptr;
9     cudaSetDevice(0);
10    cudaMalloc(&ptr, SIZE);
11
12    unsigned long long paddr;
13    tcaCreateHandleGPU(&paddr, ptr, SIZE);
14
15    printf("paddr = 0x%016llx\n", paddr);
16
17    return 0;
18 }

```

図 5: [4] で報告された 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。

```

1 #define SIZE 1000000
2
3 tcacresult tcaCreateHandleGPU(unsigned long long *paddr,
4                               void *ptr, size_t size);
5
6 int main(void) {
7     void* ptr = malloc(BUF_SIZE);
8     unsigned long long paddr;
9
10    #pragma acc enter data create(ptr[: (SIZE/sizeof(uint32_t))])
11
12    #pragma acc host_data use_device(ptr)
13    tcaCreateHandleGPU(&paddr, ptr, SIZE);
14
15    printf("paddr = 0x%016llx\n", paddr);
16
17    #pragma acc exit data delete(ptr[: (SIZE/sizeof(uint32_t))])
18
19    free(ptr);
20
21    return 0;
22 }

```

図 6: PCIe アドレス空間にマップされた GPU メモリのアドレスを OpenACC 側で取得するコード。赤枠で囲まれたディレクティブを挿入することで、デバイスポインタ `ptr` を `tcaCreateHandleGPU()` 関数に渡すことを指示する。

FPGA が自律的に GPU デバイスのグローバルメモリに対してアクセスする。そのために、FPGA は 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を取得する必要がある。それを OpenACC プログラミングモデルでどのように取得するかが、OpenACC・OpenCL 混合プログラミングの枠組みにおいて GPU-FPGA DMA 機能を利用するうえで最も肝要なポイントとなる。

[4] では、図 5 に示すように、[5] で開発された API である `tcaCreateHandleGPU()` 関数に `cudaMalloc()` で指定したデバイス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 メモリのアドレスが変数 `paddr` に格納される。CUDA ではデバイス側の配列のポインタを明示的に宣言するが、それに対し OpenACC は配列や変数がホスト側、デバイス側を問わず、プログラマ視点から見て一元化して扱えるというのが利点であるため、ホ

スト側の配列とデバイス側の配列と言ったような二つのメモリ上の配列を明示的に宣言する必要が無い。しかし、`tcaCreateHandleGPU()` 関数を呼び出して PCIe アドレス空間にマップされた GPU メモリのアドレスを取得するためにはデバイスポインタが必要であるため、[4] と同様の事を OpenACC・OpenCL 混合プログラミングで実現するためには OpenACC 側で明示的にデバイスポインタを指定する仕組みが必要となる。

そのための鍵となるのが OpenACC と CUDA の相互運用 [6] である。OpenACC で記述されたルーチンと CUDA で記述されたルーチンは、そこで使用されているデータの受け渡しが可能、すなわち OpenACC と CUDA で共通のデバイス側のメモリ (デバイスポインタ) を利用できる。この相互運用性を用いて、[6] の「OpenACC と CUDA C の相互運用 (1)」では、CUDA C で記述されたベクタ加算の GPU カーネルに利用される配列の初期化およびカーネル呼び出しを OpenACC 側で行っている。これを可能にしているのが、OpenACC で用意されている `host_data` ディレクティブである。その clause である `use_device` の引数に OpenACC 側で宣言された配列のポインタをセットすることにより、ベクタ加算のカーネルにデバイスポインタを渡すことを指示している。

この仕組みを利用して、PCIe アドレス空間にマップされた GPU メモリのアドレスを OpenACC 側で取得するコードを図 6 に示す。10 行目の `enter data` ディレクティブの clause である `create` によって GPU メモリを確保し、12 行目の `host_data` ディレクティブの clause である `use_device` の引数に `ptr` をセットすることにより、`ptr` が 10 行目で確保した GPU メモリを指すデバイスポインタであり、それを `tcaCreateHandleGPU()` 関数に渡すことを指示している。これにより、図 5 と同様に、PCIe アドレス空間にマップされた GPU メモリのアドレスが変数 `paddr` に格納される。アドレスの取得後は、[4] で述べられている手順を辿ることによって OpenACC・OpenCL 混合プログラミングの枠組み内でも GPU-FPGA DMA を利用することが可能となる。

5. 評価

5.1 評価環境

OpenACC・OpenCL 混合プログラミングの動作検証およびその枠組みにおける GPU-FPGA 間通信性能の評価には、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

表 1: 評価環境 (PPX)

ハードウェア構成	
CPU	Intel Xeon E5-2690 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
FPGA	BittWare 520N (1SG280HN2F43E2VG)
FPGA Memory	DDR4 2400MHz 32GB (8GB × 4)
ソフトウェア構成	
Host OS	CentOS 7.3
Linux Kernel Version	3.10.0-514.26.2.el7.x86_64
Host Compiler	gcc 4.8.5
GPU Compiler	CUDA 9.2.148
PGI Compiler	19.10
OpenCL SDK	Intel FPGA SDK for OpenCL 19.4.0 Build 64 Pro Edition

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 E5-2690 v4 CPU × 2, NVIDIA Tesla V100 GPU, BittWare 520N FPGA ボードが搭載されており、CPU-GPU 間は PCIe Gen3 x16 レーンにて接続されている。CPU-FPGA 間は、物理的には PCIe Gen3 x16 レーンで接続されているが、本研究で使用している BittWare 520N 用の OpenCL BSP に含まれる PCIe IP コアが PCIe Gen3 x8 までのサポートとなっているため、実際の CPU-FPGA 間のインターフェースは PCIe Gen3 x8 であることに留意されたい。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。

5.2 OpenACC・OpenCL 混合プログラミングの動作検証

OpenACC・OpenCL 混合記述によるプログラム、およびその枠組みで GPU-FPGA DMA が正しく動作していることを検証するための実験について述べる。この実験では、ベクタ加算 $c[i] = a[i] + b[i]$ を実行する OpenACC カーネルが GPU で動作し、ベクタ加算に必要な $b[]$ は FPGA から GPU に DMA 転送され、ベクタ加算の結果で

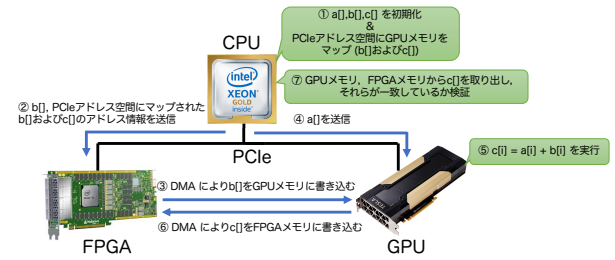


図 7: 実験の概要

```
$ pgc++ -acc -ta=tesla -c acc_cl_main.cc          <- OpenACC・OpenCL 混合プログラムのソースをコンパイル
$ nvcc -c pcie_addr_map.cu                        <- tcaCreateHandleGPU() のソースをコンパイル
$ pgc++ -acc -Mcuda -ta=tesla acc_cl_main.o pcie_addr_map.o <- オブジェクトファイルをリンク
```

図 8: 実験に用いたコードのコンパイル手順

ある $c[]$ が GPU から FPGA に DMA 転送される。

図 7 に実施した実験の概要を示す。まず、ベクタ加算のための配列 $a[]$, $b[]$, $c[]$ を CPU で初期化し、GPU と FPGA 間で DMA 転送される $b[]$ および $c[]$ を 4.2 節で述べた `tcaCreateHandleGPU()` 関数によって PCIe アドレス空間にマップ後、それらに対応するアドレス情報を取得する。それらのアドレス情報および配列 $b[]$ は OpenCL API によって CPU から FPGA に送信され、FPGA (OpenCL カーネル) は配列 $b[]$ を DMA によって GPU のグローバルメモリに書き込む。FPGA から GPU への DMA 後、CPU は `#pragma acc data copyin(a[:numdata])` によって (`numdata` は配列の要素数)、GPU に配列 $a[]$ を送信し、ベクタ加算を実行する OpenACC カーネルを起動する。ベクタ加算の実行後、FPGA (OpenCL カーネル) は、演算結果を格納した配列 $c[]$ を GPU グローバルメモリから FPGA の外部メモリへと DMA 転送する。そして、最後に、GPU グローバルメモリ上に残ったままの $c[]$ を `#pragma acc data copyout(c[:numdata])` で、FPGA の外部メモリに書き込まれた $c[]$ を OpenCL API で、CPU メモリに書き戻し、それらが一致しているか検証する。

この実験を実施した結果、CPU に書き戻したそれぞれのデータが一致することを確認した。これにより、OpenACC・OpenCL 混合記述によって実装されたプログラムが、1 つの main 関数から OpenACC で記述された GPU カーネル、OpenCL で記述された FPGA カーネルの両方を呼び出せ、かつ、我々がこれまでに開発した OpenCL で制御可能な GPU-FPGA DMA がその枠組みの中で正しく動作することが示された。

図 8 に、本実験で用いたコードのコンパイル手順を示す。まず、OpenACC・OpenCL 混合記述によるプログラムソースである、`acc_cl_main.cc` を PGI コンパイラによってコンパイルする。次に、`tcaCreateHandleGPU()` 関数であるソースファイル `pcie_addr_map.cu` を `nvcc` によってコンパイルする。`tcaCreateHandleGPU()` 関数は、内部で CUDA API を利用するので、`nvcc` によるコンパイルが必

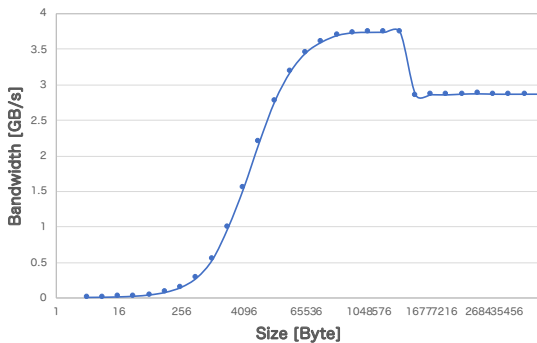


図 9: OpenACC・OpenCL 混合プログラミングにおける GPU-to-FPGA DMA の通信バンド幅

要となる。そして、それぞれのコンパイルで生成されたオブジェクトファイルを PGI コンパイラでリンクする。その際には、OpenACC に必要なシステムライブラリと CUDA に関するシステムライブラリをリンクするためのオプションである `-acc` および `-Mcuda` を付加する [6]。なお、GPU-FPGA DMA するための OpenCL カーネルは、第 3 章で述べた Intel FPGA SDK for OpenCL が提供するオフラインコンパイラによって図 8 とは別にコンパイルされる。

5.3 GPU-FPGA 間通信性能

本節では、OpenACC・OpenCL 混合プログラミングにおける GPU-to-FPGA DMA の通信レイテンシおよび通信バンド幅を評価する。[4] と同様に、OpenCL カーネル内で呼び出し可能なクロック関数を利用しているため、OpenCL カーネルの動作クロック精度で通信性能の計測している。本評価で実装した OpenCL カーネルの周波数は、228.0 MHz で駆動しているため、クロック関数の解像度は 4.3 ns となる。

4 バイトのデータを GPU から FPGA に DMA 転送するのに要するレイテンシを評価した結果、1.66 us であり、[4] より 200 nsec 余分に遅延が発生していることが分かった。これは今回の評価では、[4] よりも測定のためのロジックを OpenCL カーネルに追加したため、その影響を受けていることが考えられる。したがって、その測定ロジックを外し、[4] と同一の条件で測定すれば同様の結果が得られることが期待される。

その証拠に、図 9 に示すように、GPU-to-FPGA DMA の通信バンド幅は OpenACC・OpenCL 混合プログラミングの枠組みであっても、[4] と同様の性能を示していることが分かる。5.1 節で述べたように、本研究で使用している BittWare 520N 用の OpenCL BSP に含まれる PCIe IP コアが PCIe Gen3 x8 までのサポートとなっているため、得られる通信性能は [4] と同様になる。

このように、OpenACC・OpenCL 混合プログラミングの

枠組みであっても、[4] と同様の性能を達成する GPU-FPGA 間 DMA 機能を利用可能であることが確認できた。

6. 関連研究

GPU と FPGA を併用した演算加速というコンセプトは世界中で注目されているトレンドであり、いくつかの関連研究が存在する [7], [8]。

[7] では、我々の先行研究 [4] と同様に PCIe バスに接続された GPU-FPGA 間データ転送手法について提案している。提案手法は 2 種類存在し、1 つは PCIe アドレス空間に FPGA の外部メモリをマップし、GPU が FPGA のメモリに対して `cudaMemcpy` で読み書きするものである。すなわち、FPGA を仮想的な GPU と見せかけ、GPU 間の peer-to-peer 通信によって、GPU-FPGA 間でデータを送受信する。これは、GPUDirect RDMA をサポートしていないコンシューマ向け GPU の利用を前提とした GPU-FPGA 間通信手法である。もう 1 つは、FPGA から GPU へのデータ転送を、我々と同様に GPUDirect RDMA API によって、PCIe アドレス空間にマップされた GPU メモリアドレスを取得し、それを FPGA が DMA する際のデータの送信先として利用しているものに置き換えている。ただし、これらの手法はどちらも CPU が DMA の起動を行っている。対して、我々の開発した GPU-FPGA 間 DMA は FPGA が自律的に DMA の起動を行うため、GPU-FPGA 間 DMA における CPU の介在を極力少なくしている点が大きく異なる。

[8] では、OpenCL C++ のラッパーとして機能するランタイムである EngineCL を FPGA をサポートするように拡張し、CPU・GPU・FPGA がその枠組みで協調計算を実現したことを報告している。また、EngineCL が提供するデバイス間負荷分散アルゴリズムによって、CPU、GPU、FPGA の搭載された計算ノード上で実行させたベンチマークの性能が向上したことを確認している。ただし、1 章で述べたように、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、EngineCL に適合するようにコードを全て 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。そのため、我々は CUDA より抽象度を引き上げたプログラミングモデルである OpenACC と OpenCL の組み合わせにより GPU と FPGA の両演算加速デバイスを連携させるアプローチを採用している。

7. 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、CUDA より抽象度を引き上げたプログラミングモデルである OpenACC と OpenCL の組み合わせにより GPU と FPGA の両演算加速デバイスを連携させ、性能向上を目指す枠組みを示した。

簡易的なプログラムにより、OpenACC・OpenCL 混合プログラミングが正しく動作することを確認し、GPU から FPGA への DMA 転送は OpenACC・OpenCL 混合プログラミングの枠組みであっても、[4] と同様の性能を達成することが確認できた。

今後の研究では、現在我々のターゲットアプリケーションである ARGOT コードを OpenACC・OpenCL 混合プログラミングにより実装し、[1] と同様の演算加速が実現されることを目指す。

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。

参考文献

- [1] Kobayashi, R., Fujita, N., Yamaguchi, Y., Boku, T., Yoshikawa, K., Abe, M. and Umemura, M.: Accelerating Radiative Transfer Simulation with GPU-FPGA Cooperative Computation, *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 9–16 (online), DOI: 10.1109/ASAP49362.2020.00011 (2020).
- [2] 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).
- [3] Overview: Intel FPGA SDK for OpenCL, <https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html>.
- [4] 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).
- [5] 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).
- [6] OpenACC ディレクティブによるプログラミング: 12 章 OpenACC と CUDA C/Fortran との相互運用性, <https://www.softtek.co.jp/SPG/Pgi/OpenACC/012.html>.
- [7] Zhaopeng, S., Kuanjiu, Z., Kai, C. and Shaoqi, H.: PCIe-Based High-Performance FPGA-GPU-CPU Heterogeneous Communication Method, *2020 International Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*, pp. 66–73 (online), DOI:

- 10.1109/IWECAI50956.2020.00020 (2020).
- [8] Dávila Guzmán, M. A., Nozal, R., Gran Tejero, R., Villarroja-Gaudó, M., Suárez Gracia, D. and Bosque, J. L.: Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL, *The Journal of Supercomputing*, Vol. 75, No. 3, pp. 1732–1746 (online), DOI: 10.1007/s11227-019-02768-y (2019).