一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2018-92,CPSY2018-102,RECONF2018-66(2019-01)

Lattice-Boltzmann Method の Intel Programmable Accelerator Card への実装と評価

宮島　敬明†　　上野　知洋†　　佐野健太郎†

† 理化学研究所 計算科学研究センター プロセッサ研究チーム
〒650-0047 兵庫県神戸市中央区港島南町 6-3-5
E-mail: †takaaki.miyajima@riken.jp

あらまし　我々は複数の FPGA を用いた高性能ストリーム計算のための共通プラットフォームの構築を目指し、研究開発を行っている。FPGA を計算プラットフォームとして利用する際の問題点には、設計効率や FPGA 間通信、システムの設計・構築、運用性など多くの解決すべき問題がある。我々は、設計効率を独自のデータフロー言語”SPGen”で、FPGA 間通信を独自の RDMA 実装で、システムの設計・構築を Intel Acceleration Stack (IAS) を採用することで、運用性を仮想環境で解決を試みる。本報告では、プラットフォームの初期段階である、IAS をベースとした単体 FPGA と Direct Memory Access Controller (DMAC) ライブラリの実装と評価を述べる。具体的には、IAS のハードウェア部である Intel PAC 上に、2 つの DMAC モジュールと SPGen で実装した 5 種類の格子ボルツマン法の計算モジュール (LBM コア) を実装した。また、IAS のソフトウェア部である OPAE を利用してホスト CPU から実装した DMAC モジュールを制御する DMA ライブラリを開発した。ベアメタル環境下とコンテナ環境下で DMAC ライブラリのホストメモリと FPGA メモリ間の転送速度、各 LBM コアの Intel PAC 上での実効性能、サイクルカウンタを利用したストール率を測定した。
キーワード　高性能計算、OPAE、Intel PAC、格子ボルツマン法

An implementation and evaluation of Lattice-Boltzmann Method on Intel Programmable Accelerator Card

Takaaki MIYAJIMA†, Tomohiro UENO†, and Kentaro SANO†

† RIKEN R-CCS, Processor Research Team,
6-3-5 Minatojima-Minami, Chuo-ku, Kobe, Hyogo, Japan
E-mail: †takaaki.miyajima@riken.jp

Abstract　We are developing and researching a common platform for high performance stream computing with Field Programmable Gate Array (FPGA). User APIs and hardware modules for the common platform are developed based on Intel Acceleration Stack (IAS). Intel PAC and OPAE are a hardware part and software part of IAS, respectively. In this presentation, an preliminary evaluation of the platform and the current status of our research are shown. We implemented two Direct Memory Access Controller (DMAC) modules and a Lattice-Boltzmann Method (LBM) computing core, a computational fluid dynamics application, on Intel PAC. A control software for DMAC modules and LBM core is developed as well. We evaluated four designs of LBM core and observed that the sustained performance of each design is the same as the theoretical peak performance.
Key words　High Performance Computing, OPAE, Intel PAC, Lattice-Boltzmann Method

1. はじめに

ムーアの法則の停滞により、近年 FPGA が HPC 分野でも注目されつつある。今日のハイエンド FPGA は、14nm などの最先端プロセスを用いて製造され、汎用プロセッサと比べて高速かつ低消費電力な大規模デバイスとなってきた。特に、Intel 社の Arria10 や Stratix10 などの製品では、IEEE 準拠の単精度浮動小数点数を計算する DSP や DDR4 メモリなどが搭載さ

図 1: 冷却ブロックを取り外した Intel PAC の外観写真

れ初めている [1]。また、FPGA は外部インターフェイスを自由に接続することが可能なため、複数カードを用いた際のトポロジが自由にとれる点で GPU よりも優れている。

我々が研究開発中の高性能ストリーム計算のための共通プラットフォームは、複数 FPGA を用いたハードウェア部だけでなく、ドライバやプログラミングモデルや言語を含んでいる。プラットフォーム構築に際して、設計効率の低さを独自のデータフロー言語”SPGen”で、FPGA 間通信を独自の RDMA 実装 [2] で、システムの設計・構築を IAS を採用することで、運用性を仮想環境で解決を試みている。本報告では、プラットフォームの基礎となる IAS を用いた単体 FPGA システムとホスト CPU と FPGA の通信を担う DMAC ライブラリについて取り扱う。FPGA システムの設計構築に際し、ドライバや PCIe などの入出力ポートのなどのシェル（固定機能部）の提供は欠かせない。また、運用性の向上には、パーシャル・リコンフィギュレーション (PR) による回路の切り替えやコンテナを計算リソースの分割や有効利用が欠かせない。現時点のシェルの機能は、IAS で提供される FIM や PCIe ポート、DRAM コントローラに加え、独自に DMAC を提供する。PCIe のドライバや MMIO 経由での読み書きの API は IAS で提供されるものを用いることで、完成度や開発の継続性などの問題を回避する。運用性の問題は、IAS で提供されている PR 機能に加え、コンテナを用いたリソース分割を提供することで解決する。FPGA はボード単位以下でのリソース分割が可能であり、GPU よりもリソース分割の自由度が高い。その反面、パラメータが多く、問題としては難しい。

本報告では、プラットフォームの初期段階の評価として、IAS をベースとした Intel PAC と DMAC ライブラリの実装と評価、コンテナ環境下での予備評価について述べる。評価には、先行研究で設計した並列度の違う 5 種類の LBM コアについて、理論性能と Intel PAC 上での実効性能、ホスト CPU 側のタイマーを用いたアクセラレータ全体としての実効性能の 3 種類を用いた。また、同様の評価をコンテナを用いた仮想環境で実施することで、運用性の予備評価とした。

2. Intel Acceleration Stack (Intel PAC と OPAE)

IAS は、ハードウェア部である Intel Programmable Accelerator Card (Intel PAC) とソフトウェア部である Open Programmable Acceleration Engine (OPAE) から成る、FPGA の開発と運用を目的として設計されたソフトウェア、ファームウェア、ツール群である。図 2.2 に Intel Acceleration Stack における、Intel PAC と OPAE の関係性を示す。図中の薄い水色のボックスが Intel PAC / OPAE で提供される部分であり、黄土色の部分はユーザが設計・実装する部分にあたる。本節では、Intel PAC と OPAE の機能を中心に説明を行う。

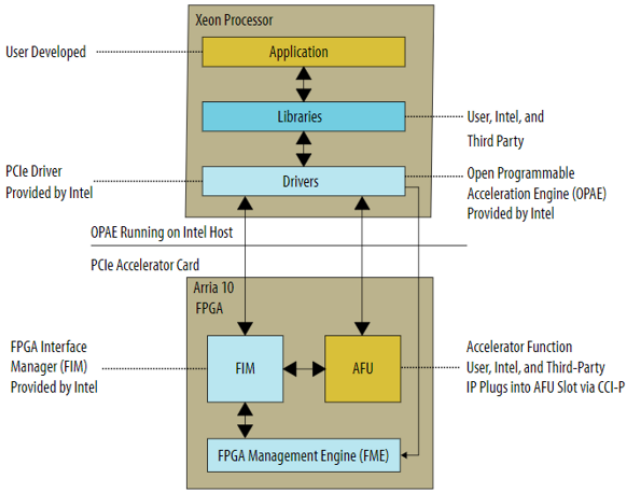

図 2: Intel PAC と OPAE との関係

OpenCL を利用した開発と異なる点として、開発の自由度の高さが挙げられる。Intel PAC と OPAE の組み合わせの場合、専用回路の開発やそれらの接続は既存の手法が利用可能となる。例えば、高位合成言語 Intel HLS を用いて開発した計算コアを任意の DMAC と Platform Designer により接続する、などという自由な構成を取ることが可能である。反面、DMAC や MMIO などと言ったハードウェアの知識だけでなく、Platform Designer などのツールの理解も必須となり、開発効率は低下する。OpenCL では C 言語ベースの記述により計算コアの実装が可能であり、開発効率が高い。また、FPGA カードのベンダーから提供される BSP を利用し、そこに記述された外部インターフェイスは宣言するだけで利用可能となるため、ハードウェアの知識は必須ではない。なお、BSP に記述されていないインターフェイスは、ハードウェアやライブラリを含めて、ユーザが BSP を独自開発しなければならない [3]。OpenCL はソフトウェア開発者が FPGA を使いやすくするために開発され、OPAE はハードウェア開発者が FPGA を使いやすくするために開発されている違いだと言える。

2.1 Intel PAC

Intel PAC はデータセンター向けに設計開発された PCIe 接続の高性能 FPGA カードである。主な構成要素として、図 4 に示すように、Arria 10 GX FPGA や PCIe Gen3 x8 ポート、2 つの 4GB DDR4-2133 メモリ (最大メモリバンド幅 2 × 17 [GB/s]) を搭載する。本稿では言及しないが、QSFP インターフェイスやフラッシュメモリ、USB ポートも搭載している。搭載されている Arria 10 GX (10AX115N2F40E2LG) FPGA は、単精度浮動小数点数用 DSP を 1518 個搭載し、科学技術

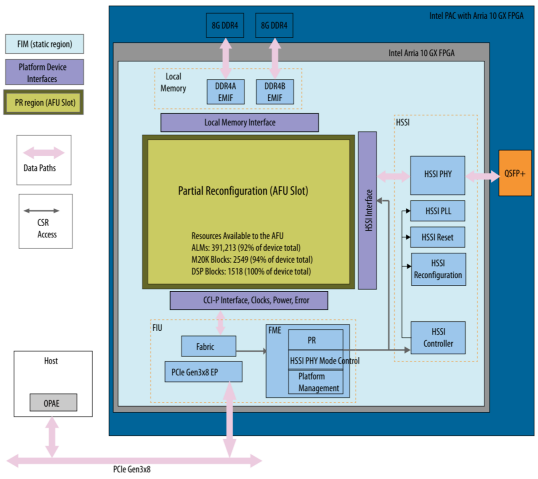

図 3: Intel PAC のブロック図

計算でも充分な性能が期待できる。Intel PAC の大きな特徴の一つに、パーシャル・リコンフィギュレーション (PR) 技術の利用を前提としている点が挙げられる。ユーザが独自の回路を PR 領域に読み込みし、それ以外の I/O 部分などを固定機能とすることで、ボード全体を停止せずに PAC の機能を変更が可能となった。これにより、実運用に耐えるシステムとなっている。FPGA 部には、AFU や FIM と言った Intel PAC 特有の機能を用意され、これらを利用することでアクセラレータとして FPGA を効率よく開発・利用することができる。2.1.1 節と 2.1.2 節でこれらについて説明を行う。

2.1.1 Acceleration Function Unit: AFU

AFU は Arria 10 GX FPGA 上に用意された PR 領域で、ユーザはここに専用回路を実装する。図 3 の黄土色の部分にあたる。PR を用いて時間的に回路を切り替えることで、AFU を超える大きさの専用回路を利用することや使用頻度に応じて回路を入れ替えることが可能になる。AFU に実装する専用回路の設計手法やフローは一般的な FPGA カードと違いはないため、ユーザは既存の FPGA 開発のツールやノウハウを用いて専用回路を設計することができる。4. 節では、評価として当研究チームで研究開発中のストリーム計算専用言語 SPGen を利用して数値流体の計算コアを実装した [4]。また、DMAC と計算コアなどの接続には Intel Platform Designer を利用した。IAS として利用した場合、後述の FIM が必須なため、AFU としては以下のリソースが利用可能である。

- ALMs (Logic): 381,213 (全リソースの 92%)
- M20K (RAM Blocks): 51,004 (全リソースの 94%)
- Block memory bits: 55,562,240 (全リソースの 100%)
- DSP Blocks: 1,518 (全リソースの 100%)

AFU 内の回路の切替には、コマンドラインツール ($fpgaconf$) や OPAE API($fpgaReconfigureSlot$ や $opae :: fpga :: types :: handle :: reconfigure$) を用いる。これは OpenCL の $clCreateProgramWithBinary$ 関数と同等であると考えられるが、OpenCL では.aocx ファイルを読み込むものに対し、OPAE では.gbs 拡張子を持つビットストリーム・ファイルを読み込む。現時点では AFU の回路使用率に関係なく、.gsb ファイルのサイズは 95.5 [MB] で固定され、コマンドラインツールを用いた PR に 1.39 [sec] ほどかかっている。この時間には DDR4 メモリを消去するいる処理も含まれる。

AFU は 256KB のメモリマップド I/O (MMIO) 領域を持ち、ホスト CPU 側から PCIe を経由して制御レジスタを読み書きすることができる。これらの MMIO 領域の読み書きは OPAE で提供されている API を利用することができる。AFU 内の専用回路から DDR4 や QSFP などの外部インターフェイスを利用する際には、DMAC の IP を AFU 内にユーザが用意する必要があり、メモリマップも設定しなければならない。本稿では、2 つの DMAC の制御や計算コアの初期値の設定を行う領域として利用している。

2.1.2 FPGA Interface Unit: FIM

FIM は、DDR4 メモリや PCIe ポート、ネットワークポートなどの FPGA と外部インターフェイスや位相同期回路、Core Cache Interface (CCI-P) と言った固定された機能を持つ、図 4 の薄い水色の部分全体のことを指す。FIM は、AFU のパーシャルリ・コンフィギュレーションや外部ポートの接続・管理を行う FPGA Management Engine (FME) を搭載する。CCI-P は FPGA とホスト CPU を繋ぐプロプライエタリなインターフェイスで、両者のキャッシュのコヒーレンシを保つ役割を持つ。

2.2 OPAE

OPAE は、Intel 社により策定されたオープンソースのライブラリであり、ドライバや API、FPGA のインターフェイスマネージャを含んでいる。ソースコードは github (https://github.com/OPAE/opae-sdk) にて、修正 BSD ライセンスで配布されている。図 の上部に位置し、ユーザアプリケーション (後述の DMAC ライブラリなど) の記述に用いられる。C 言語で書かれた OPAE の API (opae-c) は、大きく 5 つ (Enumeration API, Access API, Event API, MMIO and Shared Memory APIs, Management API) に分類される。本稿では、C/C++ の OPAE APIs を用いて FPGA のリソース管理や TX/RX DMAC の制御、計算コアの初期値の設定などを行っている。本稿では、DMAC 制御ライブラリで $fpgaWriteMMIO64$ 関数を用いてディスクリプタの書き込みを行っている。OPAE では、大きく以下の手順でローカルマシンにある FPGA リソースの管理と計算、MMIO によるデータの書き込みなどを実行する。(1) FPGA 上の計算リソース (AFU) の探索、(2) AFU の利用権の獲得、(3) AFU のレジスタのユーザ空間へのマップ、(4) 共有メモリ空間の宣言と確保、(5)AFU による計算の開始と計算結果の取得、停止、(6) 共有メモリ空間の解放、(7) AFU の利用権の解放 (5) がユーザが実装した専用回路の実行に当たる。2.1.1 節で述べた PR による AFU 上の専用回路の書き換えは、(1) より前にコマンドラインを用いて行うか、(5) で API を用いて行う。

3. ストリーム計算プラットフォーム

ここでは、まず初めに、PAC 単体でのストリーム計算プラットフォームの構成と動作について述べ、次に、開発した DMAC モジュールとライブラリについて述べる。

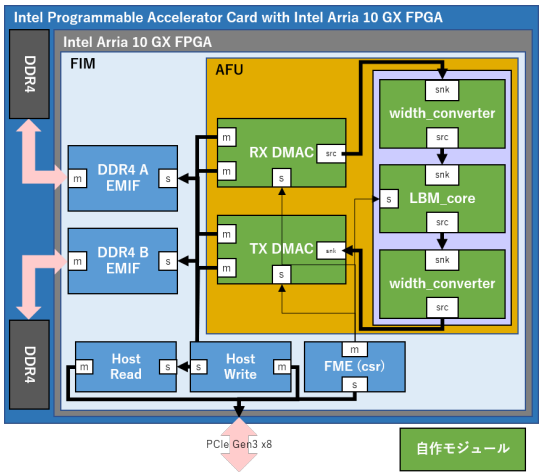

図 4: ストリーム計算プラットフォームに LBM コアを搭載する例

図 4 は、計算コアとして LBM コアを 1 つ搭載した場合のブロック図である。本稿では、DMAC モジュールと DRAM の制御モジュールを加えた計算コア以外の部分を”AFU Shell”と呼ぶ。AFU には、2 つの DMAC (*RXDMAC* と *TXDMAC*) と 1 つの LBM の計算コア (*LBM_core*)、2 つの幅変換モジュール (*width_converter*) が搭載されている。動作は次の様なものとなる。まず、初期データをホストメモリから FPGA メモリ A (EMIF A) に DMA 転送する。次に、読み込み専用 (TX) DMA モジュールが片方の DDR チャネル (EMIF A) から入力データを計算コアにストリームとして流す。LBM コアは入力ストリームを順次計算し、計算結果を書き出し専用 (RX)DMA モジュールにストリームとして流す。RX DMA は受け取ったストリームをもう片方の DDR チャネル (EMIF B) に書き出す。片方の DRAM を読み出し用、もう片方を書き出し用とすることで、メモリバンド幅を有効活用する。

3.1 DMAC モジュール

DMAC モジュールは、ホスト CPU のメモリ空間（ホストメモリ）と FPGA の外部メモリ空間（FPGA メモリ）、および FPGA の外部メモリ空間の間でデータ転送を高速に行うためのハードウェアとして、AFU 上に実装される。ホストメモリへのアクセスには、PCI-Express のメモリ空間を読み書きすることにより行われる。DMAC への転送要求や読み書きアドレスの指定などのディスクリプタの書き込みは、3. 2 節で述べるホスト CPU 側の DMAC 制御ソフトウェアから行われる。TX DMAC は、Intel 社 DMAC IP コアである”Modular SGDMA Write Master Core”を元に、入力ポートを 512bit の Avalon-MM、出力ポートを 512bit の Avalon-ST としたものである。また、RX DMAC は、”Modular SGDMA Read Master Core”を元に、入力ポートを Avalon-ST、出力ポートを Avalon-MM としたものである。2 つの DMAC のリソース消費量は、ALMs: 2910 個, Registers: 3055 個, Block memory bits: 135040, M20Ks: 27 個、DSPs: 0 個となった。入出力ポートが 512bit であるため、計算コアと DMAC の間にはデータ幅変換モジュール *width_converter* を挟む必要がある。

3.2 DMAC 制御ライブラリ

DMAC 制御ライブラリは、前述の 2 つの DMAC のリソース管理や転送データ用のバッファの作成・管理、データ転送のためのディスクリプタの書き込み、状態の監視を行う。本ライブラリは OPAE 付属のサンプルプログラム”Streaming DMA AFU” を元に我々が開発を行っている。以下にデータ転送用のユーザ API を示す。*dma_transfer_type_t* は、次の 4 種類のデータ転送タイプをサポートしている。ホストメモリ→ FPGA メモリ (H2F)、FPGA メモリ→ホストメモリ (F2H)、FPGA メモリ→ FPGA メモリ (F2F)、ホストメモリ→ホストメモリ (H2H)、である。

Listing 1: データ転送用関数 *afuShellDMATransfer*

```
1 fpga_result afuShellDMATransfer(
2   void* dst, // Destination address of data
3   const void* src, // Source address of data
4   size_t count, // Data transfer size [byte]
5   dma_transfer_type_t type // Type of data transfer
6 );
```

afuShellDMATransfer の大まかな処理の流れとして、DMAC のリソース管理や転送データ用のバッファの作成・管理は、利用権を獲得した後、DMA 用バッファの確保、ディクリプタの読み書きアドレスのマッピングを行うハンドラとイベントベースの実装となっている。データ転送のためのディスクリプタの書き込みは、マッピングされたアドレスに対し、MMIO で送信先や送信元のアドレス、データ長、終了判定の有無を書き込む。このディスクリプタは前述の”Modular SGDMA Write/Read Master Core”の”Extended format”に準拠している。状態の監視は、メモリマップされたアドレスを介して CSR を読み込んだり、割り込み信号を監視することで実現されている。また、セマフォを用いた非同期 DMA がホストメモリと FPGA メモリの間で可能である。DMA が読み書きに分かれているため、TX DMAC と RX DMAC にはそれぞれ POSIX スレッドが割当てられており、読み・書きのディスクリプタのキューを監視して、それぞれが別々に動作している。

予備評価として、計算コアを介さず DMAC モジュールを Avalon-ST で直結した状態（計算なし）で、F2F (FPGA メモリ A と FPGA メモリ B との間)、H2F、F2H、H2H でのデータ転送速度を測定した。図 5、図 6 に、それぞれの測定結果を示す。データ転送サイズは、4[byte] を 2 倍しながら 1,073,741,824[byte](=1[GB]) までとした。バンド幅の測定には、転送データサイズをホスト CPU 側のタイマーで測定した時間で割り、10 回の試行の平均から求めた。また、時間測定には *chrono* ライブラリの *clock_gettime* 関数（レイテンシと分解能は共に 100[ns] 程度とされている）を用い、ホスト CPU での DMA の要求を時間測定の開始点とし、DMAC からの終了割り込みを終了点とした。測定時間の中には、転送元 / 転送先とサイズの指定、バッファの書き込み、ディスクリプタの書き込みが含まれる。メモリバンド幅は全ての経路でデータ転送サイズが 1[GB] のときに最大となった。それぞれの経路の最大値は次の通りである。F2F: 11.6[GB/s] (転送サイズ

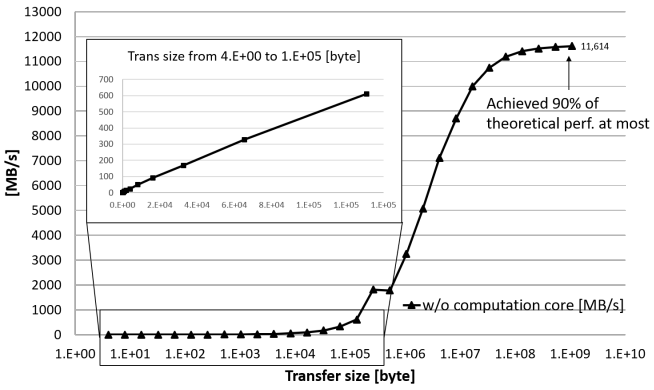

図 5: FPGA メモリ A→FPGA メモリ B 間のデータ転送速度

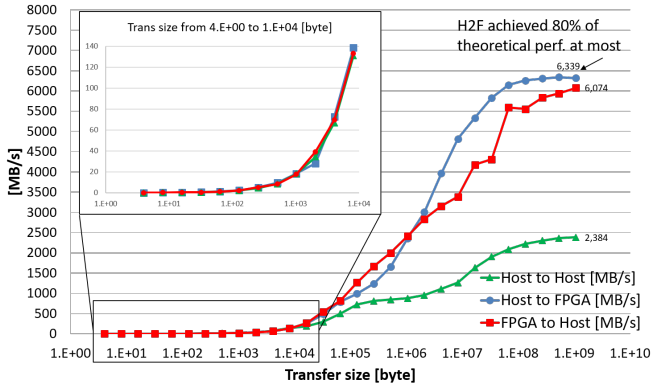

図 6: ホストメモリ⇔ FPGA メモリ A 間のデータ転送速度

1[GB], 効率 90.7%)、H2F: 6.2[GB/s] (転送サイズ 1[GB], 効率 80.6%)、F2H: 5.5[GB/s] (転送サイズ 512[MB], 効率 73.6%)、H2H: 2.3[GB/s] (転送サイズ 1[GB], 効率 30.3%)。H2H でバンド幅が低いのは、入力データと出力データが同じメモリチャネルにアロケートされ、読み込みと書き込みが衝突しているためであると考える。

4. 評　　価

ここでは、3. 節で述べたプラットフォームに、格子ボルツマン法の計算コア (LBM コア) を実装した際の評価と議論を行う。まず、LBM コアの実装について述べ、次にプラットフォーム上での評価と推定を行う。なお、評価環境は次の通りである。

- Intel Acceleration Stack: 1.1 Product, OPAE: 1.2.0
- 【ソフトウェア】 OS: CentOS 7.6 1810, Tools: Intel Quartus Prime Pro 17.1.1, SPGen 2.3, g++/gcc 4.8.5
- 【ハードウェア】 CPU: Intel Core i7-8700K 3.70GHz, Memory: DDR4-2666 32G, Motherboard: ASUSTeK ROG MAXIMUS X APEX

4.1 格子ボルツマン法の計算コア (LBM コア)

格子ボルツマン法は近年数値流体力学の分野で注目されている、非圧縮性粘性流体や気液・液液二相流体に適用可能な数値解析手法である。LBM コアの詳細は [4] に述べられているが、ここでは主要な要素について述べる。LBM コアは FPGA 実装に向けてストリーム計算に最適化されている。1 つの LBM コアを直列に n 個連続的に並べ、パイプラインの様に動作させることで、n タイムステップ先までの時間方向の並列性を利用することになる。例えば、2 個直列した実装では、出力ストリームからは 2 タイムステップを一度に計算した結果が得られる。その際、入力ストリームの幅は一定なため、必要となるメモリバンド幅は n 個直列に並べた場合でも一定のままである。なお、評価では LBM コアの計算結果については検証を行っていない。

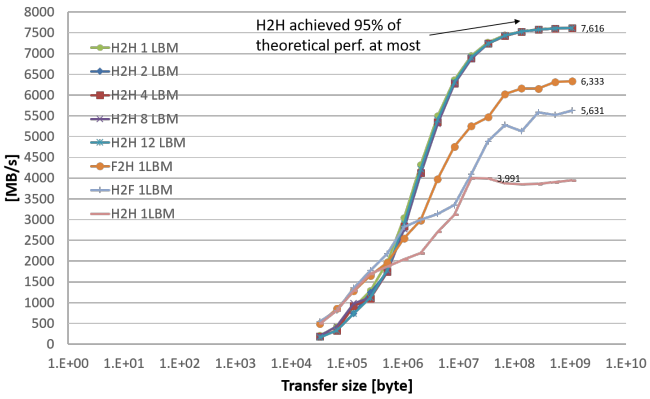

図 7: FPGA メモリ A → LBM コア→ FPGA メモリ B でのデータ転送速度（ベアメタル環境下性能）

表 1: FIM を含めた実装のリソース使用量

LBM コア	AMLs	block memory bits	RAM	DSP
1	53,183	1,779,295	482	84
2	64,835	2,468,350	753	168
4	87,973	3,846,460	1,295	336
8	134,486	6,602,680	2,379	672
12	219,203	8,897,260	1,506	1,008

LBM コアは SPGen で記述され、入出力ストリームの幅は 40 [byte] であり、幅変換モジュールで DMAC からの入出力ストリームの幅 (64 [byte]) との変換を行っている。(入力データは 40 の倍数でなければならない) また合成時の目標周波数を 200 [MHz] に設定した。LBM コアの理論ピーク性能 FN_{ops} は、動作周波数 F に各サイクルでの浮動小数点演算の回数 O_{SPE} を掛けることで算出できる。今回の実装では、F は 200 [MHz]、1 つの LBM コアの O_{SPE} は 131 (70 加算、60 乗算、1 除算) となり、FO_{SPE} = 26.2 [GFlops] となる。評価と性能予測のために、LBM コアを直列に 1,2,4,8,12 個連続に並べた 5 種類の実装を行った。1LBM コアのパイプライン段数は 830 であり、直列に n 個並べたものは n 倍のパイプライン段数を持つ。各実装の理論ピーク性能は、52.4, 104.8, 209.6, 314.4[GFlops] となる。また、1 つの LBM コアに必要なメモリバンド幅は 8 [GB/s] (= 40 [byte] × 200 [MHz]) となる。各回路資源の使用率は表 1 に示すとおりであり、直列に並べられる LBM コアの数は 12 個となった。12 個の際に RAM が全リソースの 88% を利用しているため、これ以上の並列化は不可能であった。

4.2 ベアメタル環境下性能

ベアメタル環境下で、F2F,F2H,H2F,H2H の全経路において Intel PAC 上での実効性能を測定する。評価に際し、時間測定

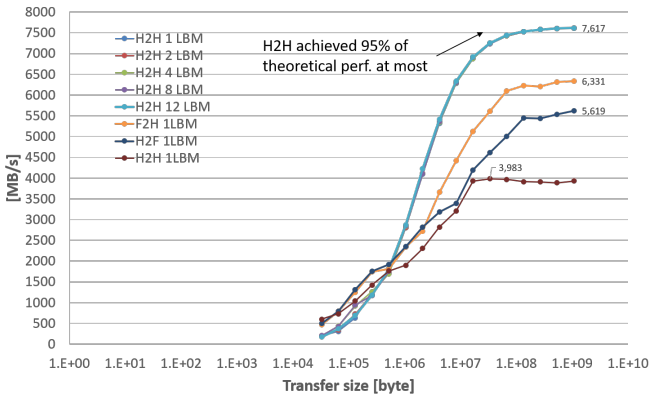

図 8: FPGA メモリ A → LBM コア→ FPGA メモリ B でのデータ転送速度（コンテナ環境下性能）

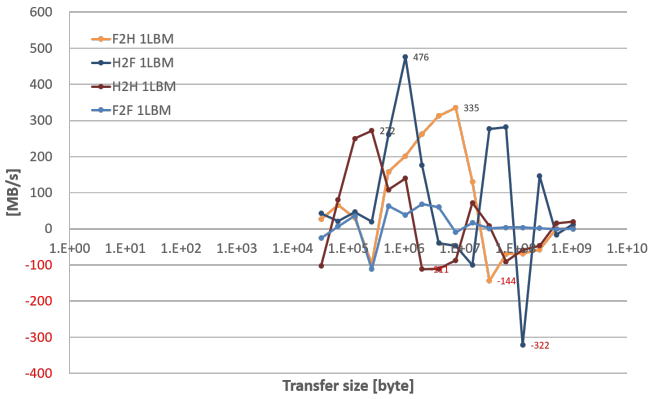

図 9: ベアメタル環境下とコンテナ環境下とのデータ転送速度の差分

にはホスト CPU 側のタイマーを用い、LBM コアの入力ポートにクロック・サイクル・カウンタ (CS カウンタ) を取り付けた。この CS カウンタは、入力データを待つサイクルを含めた全転送カウントと実際にデータが流れた実転送カウントを測定することができる。全転送カウントと実転送カウントの差分は、データ転送が間に合わなかったなどの理由により、LBM コアがストールしているサイクル数と考えることができる。図 7 に、LBM コアを直列に 1,2,4,8 個連続に並べた 4 種類の実装のデータ転送速度の測定結果を示す。転送データサイズが充分に長ければ（測定では 524,160[byte] 以上）であれば、ストール率は 0.03[%] 以下であり、各実装の実効ピーク性能は論理ピーク性能と同値であることがわかった。

4.3 Singularity コンテナ環境下性能

コンテナ環境下で 4.2 節と同様の評価を行った。コンテナサービスには、米 Lawrence Berkeley National Laboratory が研究開発を行っている近年 HPC 分野で注目を集めている”Singularity” [5] を用いた。Singularity は産総研 ABCI [6] や東工大 TSUBAME3.0 などの GPU クラスターでの利用・評価が始まっているが、FPGA での評価は前例がない。評価環境で 4.2 節と異なる点は、Singularity: 3.0.1-147.g0f938c5、gcc/g++: 7.3.1 である。図 8 に転送速度を示し、図 9 にベアメタル環境下とコンテナ環境下での転送速度の差分を示す。値が正であればベアメタル環境下での性能が高く、負であればその逆である。傾向として、データサイズが小さい場合は差分が大きいと言える。また、F2F では差分は最大 100[MB/s] で相対的に影響は小さいが、F2H, H2F, H2H では差分の最大値はそれぞれ 335, 475, 271[MB/s] であった。これは、ホストメモリを読み書きする場合は、3.2 節で述べた様なドライバの処理が重たいためであると考えられる。また、ストール率についてもベアメタル環境下と同様の傾向を示し、各実装の実効ピーク性能は論理ピーク性能と同値であることがわかった。これらの結果から、Singularity コンテナ環境下でもベアメタル環境下と遜色のない性能が発揮できることと結論付られる。

5. 結　　論

本研究報告では、我々が研究開発中の高性能ストリーム計算のための共通プラットフォームの初期評価を行った。Intel PAC 上に先行研究で設計した並列度の違う 5 種類の LBM コアと送信用・受信用の 2 つの DMAC を実装した。また、OPAE を用いて DMAC ライブラリと LBM コアの制御ソフトウェアを作成した。評価として、ベアメタル環境下とコンテナ環境下で DMAC ライブラリのホストメモリと FPGA メモリ間の転送速度、各 LBM コアの Intel PAC 上での実効性能、サイクルカウンタを利用したストール率を測定した。どちらの環境下でも DMAC ライブラリの実効転送性能は論理性能に対し、F2F で最大 90%、H2F,F2H で最大 80%、H2H で最大 30% となった。また、LBM コアで計算を行った場合は F2F で最大 95% の実効性能、ストール率は 0.03% 以下を達成した。ベアメタル環境下とコンテナ環境下での転送性能の差は、最大で 10% でありコンテナ環境下でもベアメタル環境下と遜色ない性能を発揮することがわかった。

文　　献

[1] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manohararajah, I. Milton, T. Vanderhoek and J. Van Dyken: “The stratix^{TM}10 highly pipelined fpga architecture”, Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’16, New York, NY, USA, ACM, pp. 159–168 (2016).

[2] 上野知洋, 土方康平, 佐野健太郎：“大規模 fpga クラスタのための rdma 通信機構”, Technical Report RECONF2018-28, 理化学研究所 R-CCS (2018).

[3] 藤田, 大畠, 小林, 山口, 朴：“Opencl と verilog hdl の混合記述による fpga プログラミング”, Technical Report 16, 筑波大学計算科学研究センター (2017).

[4] K. Sano and S. Yamamoto: “Fpga-based scalable and power-efficient fluid simulation using floating-point dsp blocks”, IEEE Transactions on Parallel and Distributed Systems, **28**, 10, pp. 2823–2837 (2017).

[5] G. M. Kurtzer, V. Sochat and M. W. Bauer: “Singularity: Scientific containers for mobility of compute”, PLOS ONE, **12**, 5, pp. 1–20 (2017).

[6] 佐藤仁, 小川宏高：“Ai クラウドでの linux コンテナ利用に向けた性能評価”, Technical Report 23, 国立研究開発法人産業技術総合研究所, 国立研究開発法人産業技術総合研究所 (2017).