Vol.2020-HPC-177 No.11
2020/12/21

FPGAによる変動精度演算に向けた実装方法の検討

原 忠辰[1,a)]　塙 敏博[2,1]

概要：
高性能計算における計算時間の短縮や消費エネルギー削減に向けて、アプリケーション中の各フェーズで必要とする精度に合わせて計算を行う、変動精度演算が注目を集めている。しかし、現在の CPU や GPU で任意精度を実現するためにはエミュレーションが必要であり、アプリケーション全体の検証は困難であった。そこで本研究では、FPGA 上に高位合成 (HLS: High Level Synthesis) によって任意精度演算器を実装し、ホストからオフロード実行できることを示した。

1. はじめに

ムーアの法則が終焉を迎えつつある現在、高性能計算において電力あたり性能の向上は大きな課題である。

Approximate Computing [13] は、低精度演算などを積極的に利用することにより、全体の実行時間を短縮し、消費電力を削減する試みである。その中で、アプリケーションに対して必要十分な精度で解くために演算精度を動的に変動させる、変動精度 (Transprecision) の研究が行われている。

積極的に混合精度演算を用いる例として、HPL-AI [10] が挙げられる。Top500 等のベンチマークに用いられる HPL (High Performance Linpack) は、倍精度浮動小数点演算 (FP64) のみを用いて解く必要があったが、HPL-AI においては、倍精度演算の場合と同じ解の精度であれば、途中の LU 分解には単精度演算 (FP32) や半精度演算 (FP16) を用いることで、精度を落として高速に計算を行うことができる。これによって、理化学研究所に設置されているスーパーコンピュータ富岳では、2020 年 11 月の Top500 で HPL の性能 R_{max} 442 PFLOPS に対し、HPL-AI では 2.0 EFLOPS を達成し、HPL に対して 4 倍以上の性能向上を得た [9]。このように、FP64 性能に対して、FP32 演算では 2 倍、FP16 演算では 4 倍の性能を持つ、あるいはテンソルコアや行列積向けのアクセラレータなどを備えた CPU や GPU が増えてきており、また、低精度ならばデータ量そのものが減少し、メモリバンド幅やデータ移動の点からも、混合精度、変動精度演算を用いることで高い性能向上を得ることが可能である。

しかし、CPU や GPU では、FP16 や FP32 などあらかじめ決まったビット長の演算器しか持つことができない。実アプリケーションでは、必要な精度が様々に異なるが、必要最小限のビットのみで演算をしようと考えても、ハードウェアが用意していない中間の精度を用いて演算を行うためにはエミュレーションが必要であり、長い実行時間を必要とする。

その一方で，再構成可能なハードウェアとして FPGA (Field Programmable Gate Array) が注目されている．アプリケーションに特化し，カスタマイズされたハードウェアを用いることができる。そのため、CPU や GPU とは異なり、任意精度の演算器が実際に作成可能である。FPGA にあらかじめ搭載されたハードウェア回路 (DSP) で演算可能とは限らなくなるため、かえって遅くなる可能性はあるが、現実的な速度で動作する。これらを用いた時に、実行効率、 精度、反復解法であれば反復回数等がどう変化するかを現実的な処理時間で検証することができる。

近年では回路設計技術に精通していなくても FPGA 上のハードウェアを設計する方法として，OpenCL や C++ を初めとする高位合成 (High-Level Synthesis: HLS) コンパイラも FPGA ベンダから提供されるようになってきた [8, 17]。OpenCL は，マルチコアプロセッサや GPU など，様々に異なるプラットフォーム間での並列処理を容易にプログラムするためのプログラミング言語である。Intel 社の FPGA においては、OpenCL のみを用いて汎用のプログラムのオフロードカーネルを作成することが可能であり，HPC 分野における OpenCL プログラミングについても様々な試みが進められている [19, 23, 16, 18, 24, 22].

そこで本研究では、アプリケーションにおいて変動精度を実現するためのテストベッドとして、FPGA による変動

[1] 東京大学大学院 工学系研究科
[2] 東京大学 情報基盤センター
[a)] ta_hara@cspp.cc.u-tokyo.ac.jp

精度の実装を試みる。ホスト CPU と連携して、FPGA 上でのオフロード実行を対象とし、OpenCL 記述に加えて、変動精度実現部分については、HDL と、C++による記述とを比較する。まず、予備評価として単純な四則演算について実装を試みた後、行列積 (GEMM)、三次元電磁界解析 FD3D について適用した例について紹介する。

2. 浮動小数点数と精度

数値計算を計算機上で行うためには、先ずは実数を量子化し限られたサイズのビット表現する必要がある。この表現の過程を符号化と言う。符号化の方法により表現できる幅、精度が異なる。この符号化の過程で失われた情報により数値計算では必然的に誤差が発生する。

2.1 浮動小数点フォーマット

浮動小数点の符号化には様々な種類がある。ここでは、IEEE 754 標準、これをベースにした Bfloat16 と TF32, Posit 符号化について紹介する。

2.1.1 IEEE 754 標準

IEEE 754 標準は、符号ビット、指数ビット、仮数ビットで構成される。ほとんど全ての CPU や GPU でサポートされている表現形式である。従来は、単精度 (single, FP32) と 倍精度 (double, FP64) のみであったが、後に IEEE 754-2008 によって四倍精度 (quad, FP128) と半精度 (half, FP16) が追加された。

2.1.2 IEEE 754 からの派生

2.1.2.1 bfloat16

FP16 については、指数ビットが 5 ビットと小さいため、表現できる値の桁の範囲が 10 進数で 5 桁程度までしかなく、使いにくい。そこで、FP32 と同じく指数ビットを 8 ビットとし、仮数部を 7 ビットとしたのが bfloat16(以降 BF と呼ぶ) である。特に、機械学習を対象にしたプロセッサにおいて採用されている。[2]

2.1.2.2 TF32

TF32 (TensorFloat32) は、bfloat16 の仮数部を、FP16 の仮数部に合わせて、7 ビットから 10 ビットに拡張したものである。従って、TF32 という呼び方ではあるが、実質は 19 ビットである。NVIDIA Tesla A100 で初めて採用されている。[14]

2.1.3 Posit 符号化 Psit(N,ES)

Posit 符号化は、John L. Gustafson がご案した Unum(Univeral numbers) の三つ目のタイプである [15]。Unum は既存の IEEE754 の限界を克服するためにご案された。IEEE754 の拡張版の Unum のタイプ 1 と、実数の表現範囲を $\pm\infty$ まで拡張した Unum タイプ 2 引き続き、ハードウェアに親密な符号化方が Posit 符号化である。

Posit 符号化には長さ（N）と、標準倍率のベースになる ES と言うパラメタが必要である。ES は標準倍率のベース

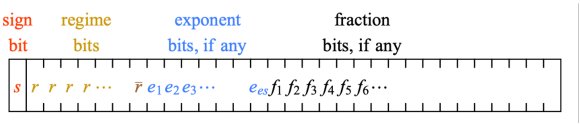

図 1: Posit 符号化のビット構成 [3]

($2^{2^{ES}}$) を決めると同時に、指数ビットの最大長を意味する。Posit 符号化は、**図 1** のように 4 つのパートに構成されている。符号ビット（S)、基準倍率次数になる Regime ビット（G)、指数ビット (E)、そして、仮数ビット（M）がある。Posit は下記の式で数値を表現している。

$$x = \pm useed^k \times 2^e \times f \ where \ useed = 2^{2^{ES}}$$

上記の式の k を決める際に Regime ビットが使われる。Regime ビット決まっている長さはない、代わりに値が反転するまでを数えて k を決める。0 が n 個続いていたら $k = -n$ になり、1 が n 個続いていたら $k = n - 1$ になる。

Posit 符号化では、N ビットの内、符号ビットとして 1 ビット、そして Regime ビットを表現した残りのビットの中で指数ビット最大 ES ビットを、残り余ったビットを仮数ビットとして用いる。

2.2 変動精度

高性能計算では、伝統的に倍精度演算が用いられることが多かったが、適切な精度に落として演算を行うことには以下のような利点がある。

- 近年の CPU, GPU においては、FP64 に対して FP16 の性能が 4 倍、INT8 であれば 8 倍になるなどからわかるように、演算器に要するビット数が少なければ、その分だけ同時計算数を増やすことが可能になる。
- データを表現するのに必要な実際のサイズが小さくなるため、メモリフットプリントや、メモリバンド幅の点で有利になる。
- 回路中を移動するデータ幅が小さくなるため、電力効率が改善する。

ただし、必要な精度が得られているかどうかを確認する必要があり、反復法の場合には反復回数が増えて全体としては性能が悪化する可能性もある。計算によっては、精度保証演算を取り入れることで、誤差を把握することも可能であると考えられる。

一方で、積分計算など加算処理が主になる演算については、四倍精度などの高精度演算を用いた方が誤差の蓄積が減少するため、全体として演算性能や精度の改善につながる可能性も高い。

3. 関連研究

3.1 Deepfloat

Deepfloat[12] は、深層学習において、多様な浮動小数点

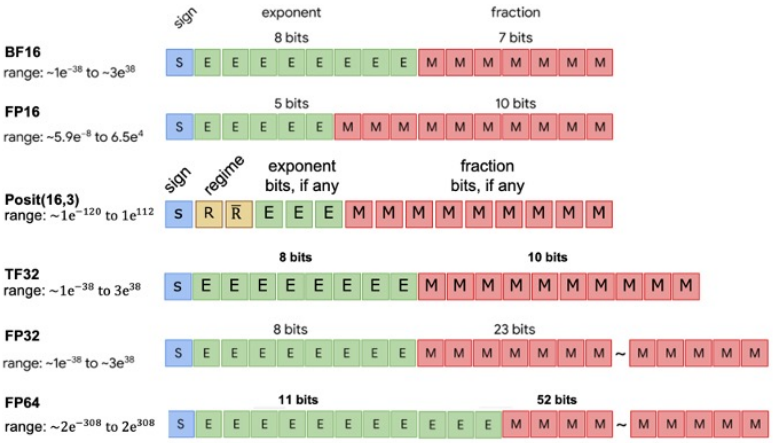

図 2: 多様なビットの符号化方法、上から BFloat16、IEEE-754 16bits 、Posit(16,3)、Tensor Float 32、IEEE-754 32bits、IEEE-754 64bits[2, 3, 14]

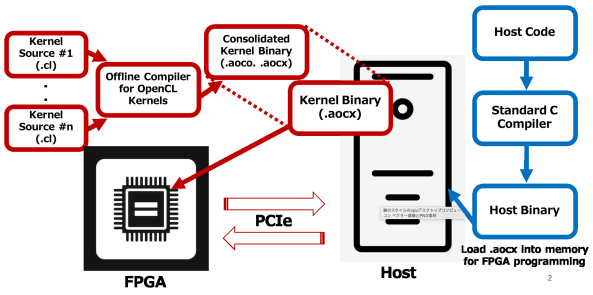

図 3: OpenCL の作業の流れ [4]

の符号化方法を変えながら推論の高速化、電力効率化の可能性を検証した研究である。コアの部分は HDL で記述され、全体としては OpenCL を用いて実装されている。

Deepfloat では、事前に学習された ResNet-50 のパラメータを用いて、推論の正解率を評価している。浮動小数点フォーマットとしては、IEEE 754 に加えて、Posit と Posit の仮数部分をログに拡張した (N, ES, α, β, γ) log 符号化で評価を行っている [12]。

本研究では、C++との比較対象として、Deepfloat の HDL 実装を修正して利用している。実際には IEEE754 のみを対象としているため、Deepfloat の IEEE 754 に関する HDL の実装部のみを使っている。

3.2 FBLAS: Streaming Linear Algebra on FPGA

FBLAS は Intel 社の FPGA を対象とした BLAS（Basic Linear Algebra Subprograms）ライブラリである [1]。FBLAS は HLS モジュール（OpenCL で記述）とホスト API で構成されている。それぞれは、Python で実装されていコード生成器により自動生成される。コード自動生成には、JSON ファイルのデータを基盤としてユーザーがカスタマイズ可能である。

本研究でも FBLAS のように汎用性が高い変動精度演算に向けたライブラリの開発を目指している。

4. 任意精度演算器の実装

本稿では、Intel 社の FPGA 搭載ボードを演算のオフローディングに用いることとし、Intel FPGA SDK for OpenCL Pro Edition[5] を用いて実装を行う。

4.1 OpenCL を用いた FPGA プログラミング

FPGA 内部の論理を設計するためには、従来は Verilog HDL や VHDL といったハードウェア記述言語を用いて記述するのが一般的であり、求められるアルゴリズムにあわせて人手で論理回路レベルに変換する必要があった。そのため、FPGA 上に実装するためには多大な時間と労力が必要であり、様々な HPC アプリケーションに FPGA を活用することは現実的ではなかった.

しかし近年では、OpenCL, C++などを用いた設計ツールが FPGA ベンダーによって提供されるようになってきた。

OpenCL は Khronos グループによって標準化されている並列化プログラミング環境であり、GPU などのアクセラレータ向けに仕様策定や開発が進められたものである [**opencl**]。FPGA 向けのコンパイラでは、OpenCL での記述から内部で Verilog HDL や IP ライブラリを出力し、論理合成を行う。

なお、Intel FPGA では、FPGA 搭載ボードをアクセラレータとして演算のオフローディングに用いる場合、C++の記述は直接扱うことができず、BSP (Board Support Package) と共に、OpenCL を経由してライブラリとして呼び出す必要がある。

OpenCL の作業の流れは**図 3** の通りである。FPGA 側の処理を記述するカーネルソースファイル（図 3 の.cl ファイル）を作成後、コンパイラ（Intel 社が提供している）を用いてカーネルバイナリファイルを合成する。そして、このカーネルバイナリファイルをホスト側で FPGA マウント、呼び出す。

4.1.1 OpenCL におけるライブラリ利用

OpenCL では OpenCL 以外の言語による記述（HDL、C++など）をライブラリとして扱える機能を提供している [4, 12, 1]。ライブラリを利用する作業の流れは**図 4** の通りである。ソースファイルを用いて、OpenCL ライブラリファイル（.aoclib）を合成、このファイルを OpenCL の合成時にリンクすることで OpenCL 側でライブラリを使える。

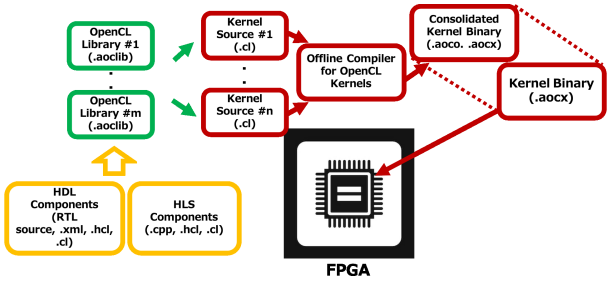

図 4: OpenCL のライブラリ利用時の作業の流れ [4]

4.1.1.1 HDL による OpenCL ライブラリ

ハードウェア記述言語を用いてライブラリを合成する流れは**図 5** の通りである。

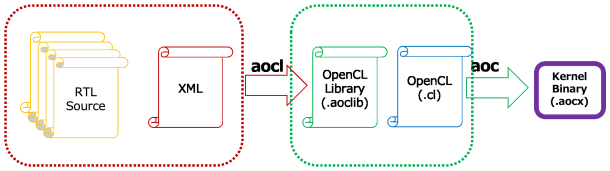

図 5: HDL のカーネルバイナリファイル合成 [4]

図 6 の様に HDL で記述されたコア部分に加え、エミュ

```
module TestModule(
    input logic [20:0] floatIn,
    output logic [20:0] floatOut,
    input clock,
    input resetn,
    input ivalid,
    input iready,
    output logic ovalid,
    output logic oready
);
    // Process
    TestProcess() testProcess(.*);
endmodule
```

図 6: ライブラリとして用いる SystemVerilog ファイルの例

```
AFP21 testName(AFP21 floatIn) {
    // This file is for emulation model
    // So this block is not executing, when FPGA
    really working
    AFP21 floatOut = TestProcess(floatIn);
    return floatOut;
}
```

図 7: エミュレーション用モデルファイルの例

レーション用に**図 7** の OpenCL 記述を追加する。この部分はエミュレーションの為の処理なので、入力、出力の型さえ合えば問題ない。

そして**図 8** の様に、XML を使って、OpenCL 側で呼び出す関数名、HDL モジュール名、入出力などを定義する。

XML に記述した OpenCL 側で使える様に**図 9** の様に OpenCL ヘッダファイルを作成する。

OpenCL 側では**図 10** の様に、OpenCL ヘッダで定義した関数を呼び出すことで使える。

最後に**図 11** の様に合成すると、OpenCL バイナリファイルを合成できる。

4.1.1.2 C++による OpenCL ライブラリ

C++によるフローは**図 12** の通りである。

コアの処理は**図 13** の様に、C++で記述できる。

そして HDL と同じく図 9 の様に、OpenCL 側で呼び出せる OpenCL ヘッダーファイルを作成する。

OpenCL 側では図 10 の様に、ヘッダーファイルで定義した関数を使える。

最後に**図 14** の様に合成すると、OpenCL バイナリファイルを合成できる。

4.2 任意精度浮動小数点演算器の実装

本研究では、IEEE 754 標準に基づく任意精度浮動小数点を実装した。具体的には任意の長さの指数ビットと仮数ビットの浮動小数点を実装した。実装には HDL と C++

```
<RTL_SPEC>
    <FUNCTION name="testName" module="TestModule">
        <ATTRIBUTES>
            <IS_STALL_FREE value="yes"/>
            <IS_FIXED_LATENCY value="yes"/>
            <EXPECTED_LATENCY value="4"/>
            <HAS_SIDE_EFFECTS value="no"/>
            <ALLOW_MERGING value="yes"/>
        </ATTRIBUTES>
        <INTERFACE>
            <AVALON port="clock" type="clock"/>
            <AVALON port="resetn" type="resetn"/>
            <AVALON port="ivalid" type="ivalid"/>
            <AVALON port="iready" type="iready"/>
            <AVALON port="ovalid" type="ovalid"/>
            <AVALON port="oready" type="oready"/>
            <INPUT  port="floatIn" width="21"/>
            <OUTPUT port="floatOut" width="21"/>
        </INTERFACE>
        <C_MODEL>
            <FILE name="sample_model.cl" />
        </C_MODEL>
        <REQUIREMENTS>
            <FILE name="sample.sv" />
        </REQUIREMENTS>
    </FUNCTION>
</RTL_SPEC>
```

図 8: XML によるライブラリ定義ファイルの例

```
AFP21 testName(AFP21 floatIn);
```

図 9: OpenCL ヘッダーファイル（.hcl ファイル）の例

```
#include "test.hcl"

__kernel
__attribute((max_global_work_dim(0)))
void testKernel(
    global AFP21* restrict floatIn,
    unsigned int N,
    global AFP21* restrict floatOut
) {
    for (int i = 0; i < N; i++) {
        floatOut[i] = TestName(floatIn[i]);
    }
}
```

図 10: HDL によるライブラリを呼び出す OpenCL ファイルの例

```
aoc library hdl-comp-pkg sample.xml sample.aoco
aoc library create -o sample.aoclib sample.aoco
aoc -l sample.aoclib sample.cl
```

図 11: HDL ライブラリコンパイル例

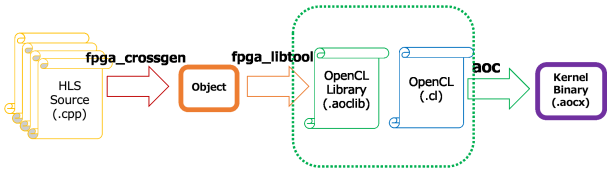

図 12: C++を用いた場合のカーネルバイナリファイル合成 [6]

```
extern "C"
AFP21 testName(AFP21 floatIn) {
    // HLS can describe process using C++
    AFP21 floatOut = TestProcess(floatIn);
    return floatOut;
}
```

図 13: ライブラリ向け C++ファイル例

```
fpga_crossgen --target aoc -DTARGET_OCL sample.cpp
     -o sample.aoco
fpga_libtool --target aoc sample.aoco --create
    sample.aoclib
aoc sample.cl -l sample.aoclib
```

図 14: ライブラリ C++コンパイル例（Quartus 19.4 対象）

```
#include "HLS/hls_float.h"

using AFP21 = ihc::hls_float<8, 12>;

AFP21 testName(AFP21 floatInA, AFP21 floatInB) {
    return floatInA + floatInB;
}
```

図 15: hls_float を用いた実装例

```
typedef unsigned int __attribute__((__ap_int(21)))
      AFP;
```

図 16: ホストと FPGA 間のインタフェース

の二つの方法を使用した。

4.2.1 HDL を用いた実装

本研究では 3.1 章の関連研究で紹介した Deepfloat の HDL コード [11] を用いて実装した。Deepfloat の研究は 28nm プロセスの FPGA を対象としている（実際には Intel 社の Stratix 5 であると考えられる）[11]。それに対し、本研究では Stratix 10 を使用しているため、当 HDL コードは実験対象の FPGA に最適化されている訳ではない。

4.2.2 C++を用いた実装

C++を扱うための Intel 社の HLS コンパイラは、任意精度の浮動小数点（hls_float）もサポートしている [6]。この機能を使う方法は**図 15** の様に hls_float.h を include して、指数ビットと仮数ビットを指定して使用できる。

4.2.3 ホストと FPGA 間インタフェース

OpenCL の SDK では任意精度定数を提供している。この任意精度定数を用いてホスト側で任意の長さのビットのやりとりを行う。実際は**図 16** の様に定義して使用する。

5. 評価

HDL、C++を用いた実装と、比較の為 FP16、FP32、FP64 の場合は OpenCL の基本記述までの、三つを比較してリソースの使用率、スループット、誤差を比較した。

誤差の場合アルゴリズム 1 の様に正規化されて計算を行なった。アルゴリズム 1 の $reference$ としては 32 ビット以下ではホスト側で単精度で計算した計算結果を、32 ビットから 64 ビットでは、倍精度で計算計算した結果を使用した。

Host CPU	Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz, 2 sockets
OS	Red Hat Enterprise Linux Server 7.7
FPGA	Nallatech 520N, 2 cards (Stratix 10 GX 2800)
Quartus Version	19.4.0 Build 64 Pro Edition

(a) Cygnus の計算ノード仕様 [21, 20]

Logic elements (LEs)	2,753,000
Adaptive logic modules (ALMs)	933,120
ALM registers	3,732,480
M20K memory blocks	11,721
M20K memory size	229 Mb
MLAB memory size	15 Mb
DSP blocks	5,760
Peak floating-point performance	9.2 TFLOPS

(b) Intel Stratix10 SX/GX 2800 のリソース情報 [7]

表 1: Cygnus システム概要

	ALUTs	Registers	Logic utilization Size	(%)	DSP Block	(%)	Memory bits Size	(%)	RAM Block	(%)	Clock
FP64	156,227	329,114	195,549	20.96%	4	0.07%	9,977,240	4.16%	1,014	8.65%	279.17 MHz
FP32	151,600	314,754	191,367	20.51%	3	0.05%	10,131,416	4.22%	1,011	8.63%	265.00 MHz
FP16	152,359	297,649	191,885	20.56%	1	0.02%	10,604,312	4.42%	1,011	8.63%	240.63 MHz

(a) OpenCL の基本記述

	ALUTs	Registers	Logic utilization Size	(%)	DSP Block	(%)	Memory bits Size	(%)	RAM Block	(%)	Clock
FP64	163,240	334,749	200,779	21.52%	4	0.07%	9,985,432	4.16%	1,014	8.65%	240.63 MHz
FP32	151,522	302,388	191,362	20.51%	3	0.05%	10,131,416	4.22%	1,011	8.63%	256.25 MHz
FP21	149,938	306,516	189,591	20.32%	1	0.02%	8,812,376	3.67%	887	7.57%	277.50 MHz
FP16	152,527	309,517	191,396	20.51%	1	0.02%	10,604,312	4.42%	1,011	8.63%	256.00 MHz
BF	152,270	312,617	190,135	20.38%	1	0.02%	10,604,312	4.42%	1,011	8.63%	250.00 MHz

(b) HDL

	ALUTs	Registers	Logic utilization Size	(%)	DSP Block	(%)	Memory bits Size	(%)	RAM Block	(%)	Clock
FP64	154,831	303,054	193,711	20.76%	4	0.07%	9,973,144	4.15%	1,009	8.61%	226.00 MHz
FP32	152,790	295,981	191,809	20.56%	1	0.02%	10,130,392	4.22%	1,005	8.57%	231.25 MHz
FP21	149,740	299,293	188,748	20.23%	1	0.02%	8,811,352	3.67%	885	7.55%	268.00 MHz
FP16	151,995	307,218	190,703	20.44%	1	0.02%	10,603,288	4.42%	1,005	8.57%	278.00 MHz
BF	151,877	297,776	190,834	20.45%	1	0.02%	10,603,288	4.42%	1,005	8.57%	259.38 MHz

(c) C++

表 2: 四則演算カーネルのリソース使用率

Algorithm 1 誤差の計算

Require: args $reference, output, N$
  $diff \leftarrow 0$
  $ref \leftarrow 0$
  for $i = 0, \cdots, N-1$ **do**
    $diff \leftarrow diff + (reference[i] - output[i])^2$
    $ref \leftarrow ref + reference[i]^2$
  end for
  $error \leftarrow \frac{\sqrt{diff}}{\sqrt{ref}}$
  return $error$

Algorithm 2 四則演算カーネルアルゴリズム

Require: args N
  $A[N] \leftarrow random()$
  $B[N] \leftarrow random()$
  $C[N] \leftarrow 0$
  for $i = 0, \cdots, N-1$ **do**
    $C[i] \leftarrow A[i] \bigotimes B[i](\bigotimes can\ be\ +, -or\times)$
  end for
  return C

5.1 実験環境

本稿では筑波大学計算科学研究センターが運営しているシステムである Cynus を用いて評価を行なった。

Cygnus の基本的なノード情報は**表 1a** の表にまとめた。使用した FPGA は、Intel 社の Stratix 10GX2800 である。Stratix 10GX2800 のリソーススペックは**表 1b** である。

5.2 基本四則演算を用いた評価

まず基本的な四則演算を実装し、任意精度が正しく動作するかを確認した。

5.2.1 アルゴリズム

確認した演算は足し算、引き算、掛け算の 3 種類で行った。アルゴリズム 2 が実際のカーネルのアルゴリズムである。

5.2.2 FPGA リソース使用率

本 OpenCL バイナリファイルには、精度の変換（拡張、縮小）、足し算、引き算、掛け算の 5 つのカーネルが含まれている。リソース使用率結果は**表 2** の通りである。

凄く単純な計算である為、ビットのサイズが変わっても使用率はそこまで変わっていない。

5.2.3 結果

本実験の際に解いた問題サイズ（アルゴリズム 2 の N に該当）は 2^{21} である。

図 17 で確認できる様に、スループットを比較すると FP32 の OpenCL の結果とそこまで大きな違いはない。最小のスループットは FP64 の HDL の結果で、約 0.85 倍性能低下があった。最大のスループットは FP16 の HDL の結果で、約 1.05 倍の性能向上が得られた。この結果の理由としては単純な計算を行っている為、精度に依らず、リソースの使用量が増えてなかったからだと思われる。

誤差の場合 FP32 以下の場合（FP32、FP21、FP16、BF）

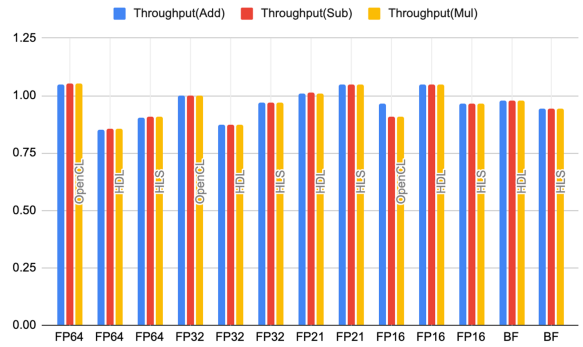

図 17: 四則演算における FP32 の OpenCL を基準として正規化したスループット

		Error Rate(Add)	Error Rate(Sub)	Error Rate(Mul)
FP64 (E:11,M:52)	OpenCL	0.00E+00	0.00E+00	2.49E-08
	HDL	0.00E+00	3.94E-17	2.49E-08
	HLS	1.62E-14	4.30E-14	2.49E-08
FP32 (E:8,M:23)	OpenCL	0.00E+00	0.00E+00	0.00E+00
	HDL	0.00E+00	2.11E-08	0.00E+00
	HLS	0.00E+00	0.00E+00	0.00E+00
FP21 (E:8,M:12)	HDL	6.70E-05	1.07E-04	8.27E-05
	HLS	6.70E-05	1.05E-04	8.27E-05
FP16 (E:5,M:10)	OpenCL	2.68E-04	4.20E-04	3.31E-04
	HDL	2.68E-04	4.29E-04	3.31E-04
	HLS	2.68E-04	4.20E-04	3.31E-04
BF (E:8,M:7)	HDL	2.14E-03	3.44E-03	2.65E-03
	HLS	2.14E-03	3.36E-03	2.65E-03

表 3: 四則演算における正規化した誤差

	ALUTs	Registers	Logic utilization Size	(%)	DSP Block	(%)	Memory bits Size	(%)	RAM Block	(%)	Clock
FP64	445,082	1,052,960	466,455	49.99%	1,030	17.88%	10,701,464	4.46%	1,306	11.14%	312.50 MHz
FP32	120,553	296,621	180,509	19.34%	266	4.62%	8,831,664	3.68%	1,004	8.57%	305.00 MHz
FP16	233,499	477,492	256,399	27.48%	134	2.33%	8,018,120	3.34%	834	7.12%	323.33 MHz

(a) OpenCL の基本記述

	ALUTs	Registers	Logic utilization Size	(%)	DSP Block	(%)	Memory bits Size	(%)	RAM Block	(%)	Clock
FP64	797,113	1,299,718	684,952	73.40%	1,030	17.88%	13,278,360	5.53%	1,713	14.61%	208.33 MHz
FP32	428,088	697,342	388,845	41.67%	262	4.55%	10,740,400	4.47%	1,320	11.26%	265.00 MHz
FP25	333,553	551,224	322,290	34.54%	134	2.33%	9,469,360	3.94%	1,088	9.28%	297.50 MHz
FP24	324,813	488,067	316,505	33.92%	134	2.33%	9,440,688	3.93%	1,096	9.35%	280.00 MHz
FP21	296,614	457,311	291,676	31.26%	134	2.33%	9,126,576	3.80%	1,075	9.17%	297.50 MHz
FP16	258,813	426,929	266,204	28.53%	134	2.33%	8,632,520	3.60%	1,002	8.55%	288.89 MHz
BF	246,474	433,729	252,338	27.04%	134	2.33%	8,632,520	3.60%	1,002	8.55%	323.33 MHz

(b) HDL

	ALUTs	Registers	Logic utilization Size	(%)	DSP Block	(%)	Memory bits Size	(%)	RAM Block	(%)	Clock
FP32	128,940	298,479	183,615	19.68%	262	4.55%	10,547,888	4.39%	1,269	10.83%	320.83 MHz
FP25	278,482	533,024	290,331	31.11%	134	2.33%	12,636,080	5.26%	1,246	10.63%	311.11 MHz
FP24	274,458	503,098	283,216	30.35%	134	2.33%	12,487,088	5.20%	1,258	10.73%	306.25 MHz
FP21	259,805	490,858	271,398	29.09%	134	2.33%	11,792,816	4.91%	1,237	10.55%	326.67 MHz
FP16	220,809	421,623	240,632	25.79%	134	2.33%	9,910,472	4.13%	1,039	8.86%	333.33 MHz
BF	232,541	418,987	245,517	26.31%	134	2.33%	10,480,840	4.37%	1,061	9.05%	341.67 MHz

(c) C++

表 4: GEMM カーネルのリソース使用率

仮数ビットの長さに比例して誤差が小さくなった。FP64 の場合掛け算の場合予想より誤差が大ことが確認できたが、理由は不明である。

5.3 GEMM を用いた評価

次に、GEMM を用いて評価を行なった。

5.3.1 FPGA リソース使用率

リソース使用率の結果は、**表 4** の通りである。

C++の場合リソース容量不足のため FP64 の回路合成に失敗した。そのため**表 4c** は FP32 までしか存在しない。

HDL と C++を比較すると C++のリソース使用率が低く、クラック周波数が高いと言うことを確認できる。

5.3.2 結果

本実験で解いた問題サイズ（アルゴリズム **??**の N,M,P に該当）は $N = 2048$、$M = 1024$、$P = 1024$ である。

図 18 を見ると OpenCL と C++の場合ビット長さによるスループットの変化は、OpenCL の FP64 の約 0.98 倍と C++の BF の約 1.11 倍で大きくない。その理由としては今回対象にした GEMM は FPGA のパイプライン化されてない状態であるからだと思われる。

対して HDL の場合ビットの長さによるスループットの変化が大きい。その理由としてはリソースの使用率が大きくなってクラック周波数が落ちているからだと思われる。

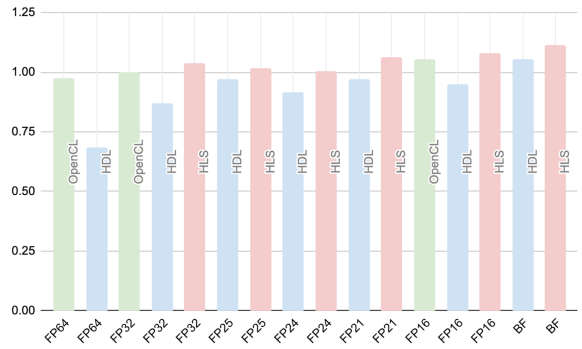

図 18: GEMM 問題サイズ A : $2048 \times 1024, B$: $1024 \times 1024, C$: $2048, \times 1024$ 場合の FP32 の OpenCL を基準として正規化したスループット

		Error Rate
FP64 (E:11,M:52)	OpenCL	8.22E-16
	HDL	4.06E-19
FP32 (E:8,M:23)	OpenCL	4.36E-07
	HDL	0.00E+00
	HLS	0.00E+00
FP25 (E:8,M:16)	HDL	5.48E-05
	HLS	5.48E-05
FP24 (E:8,M:15)	HDL	1.10E-04
	HLS	1.10E-04
FP21 (E:8,M:12)	HDL	1.04E-03
	HLS	1.04E-03
FP16 (E:5,M:10)	OpenCL	1.47E-02
	HDL	1.64E-02
	HLS	1.64E-02
BF (E:8,M:7)	HDL	2.21E-01
	HLS	2.21E-01

表 5: GEMM 問題サイズ A : $2048 \times 1024, B$: $1024 \times 1024, C$: $2048, \times 1024$ 場合の正規化した誤差

誤差に関しては結果で確認できるように、仮数ビットに比例して誤差が小さくなっている事を確認できた。

5.4 Finite Difference 3D を用いた評価

5.2、5.3 章までは比較的簡単な演算を試したので、FD3D アプリケーションを用いた評価を行なった。

5.4.1 アルゴリズム

Finite Difference 3D のアルゴリズムはアルゴリズム 3 の通りである。簡単に説明すると、$size_x, size_y, size_z$ の格子の $time\_steps$ 後の状態を計算するアルゴリズムで、現時点の格子と半径 $radius$ 内での常数を掛け算して次の時点の格子の値を決定している。

Algorithm 3 Finite Difference 3D カーネルアルゴリズム

Require: args $size_x, size_y, size_z, time\_steps, radius$
Ensure: s after $time\_steps$ passed
  $s[size_x][size_y][size_z] \leftarrow random()$
  $buf[size_x][size_y][size_z] \leftarrow 0$
  $c[radius + 1] \leftarrow random()$
  for $t = 0, \cdots, time\_steps$ **do**
    for $x = 0, \cdots, size_x$ **do**
      for $y = 0, \cdots, size_y$ **do**
        for $z = 0, \cdots, size_z$ **do**
          if Points In $radius$ **then**
            $tmp \leftarrow c[0] * s[x][y][z]$
            for $r = 1, \cdots, radius$ **do**
              $tmp \leftarrow tmp + c[r] * s[x - r][y][z]$
              $tmp \leftarrow tmp + c[r] * s[x + r][y][z]$
              $tmp \leftarrow tmp + c[r] * s[x][y - r][z]$
              $tmp \leftarrow tmp + c[r] * s[x][y + r][z]$
              $tmp \leftarrow tmp + c[r] * s[x][y][z - r]$
              $tmp \leftarrow tmp + c[r] * s[x][y][z + r]$
            end for
            $buf[x][y][z] \leftarrow tmp$
          else
            $buf[x][y][z] \leftarrow s[x][y][z]$
          end if
        end for
      end for
    end for
    $s \leftarrow buf$
  end for
  return s

5.4.2 FPGA リソース使用率

リソース使用率の結果は**表 6** の通りである。

C++の場合 FP64 リソース容量不足のため FP64 の回路合成に失敗した。そのため**表 6c** は FP32 までしか存在しない。それ以外にも HLD、C++共通に P21、FP22、FP23 の場合、HDL では FP24 の場合も合成に失敗した。

5.4.3 結果

本実験で解いた問題サイズ（アルゴリズム 3 の $size_x, size_y, size_z, time\_steps, radius$ に該当）は $size_x = 504, size_y = 504, size_z = 504, time\_steps = 5, radius = 4$ である。

	ALUTs	Registers	Logic utilization		DSP		Memory bits		RAM		Clock
			Size	(%)	Block	(%)	Size	(%)	Block	(%)	
FP64	611,397	1,216,236	567,747	60.84%	248	4.31%	44,282,520	18.45%	2,556	21.81%	300.00 MHz
FP32	140,254	287,735	184,671	19.79%	408	7.08%	24,432,856	10.18%	1,610	13.74%	370.83 MHz
FP16	267,130	499,474	274,905	29.46%	48	0.83%	16,051,608	6.69%	1,092	9.32%	358.33 MHz

(a) OpenCL の基本記述

	ALUTs	Registers	Logic utilization		DSP		Memory bits		RAM		Clock
			Size	(%)	Block	(%)	Size	(%)	Block	(%)	
FP64	762,428	758,100	629,644	67.48%	328	5.69%	52,715,672	21.96%	3,048	26.00%	151.25 MHz
FP32	415,715	633,997	370,826	39.74%	88	1.53%	28,580,056	11.91%	1,911	16.30%	230.00 MHz
FP25	336,185	471,834	318,220	34.10%	48	0.83%	23,787,224	9.91%	1,565	13.35%	225.00 MHz
FP16	261,262	404,404	262,412	28.12%	48	0.83%	17,380,120	7.24%	1,216	10.37%	267.50 MHz
BF	247,317	379,742	248,791	26.66%	48	0.83%	17,380,120	7.24%	1,216	10.37%	276.67 MHz

(b) HDL

	ALUTs	Registers	Logic utilization		DSP		Memory bits		RAM		Clock
			Size	(%)	Block	(%)	Size	(%)	Block	(%)	
FP32	144,498	309,743	190,954	20.46%	408	7.08%	28,973,336	12.07%	1,934	16.50%	346.88 MHz
FP25	331,055	586,257	324,376	34.76%	48	0.83%	23,690,904	9.87%	1,557	13.28%	328.13 MHz
FP24	423,683	747,328	391,437	41.95%	48	0.83%	24,094,680	10.04%	1,939	16.54%	224.00 MHz
FP16	247,084	457,258	258,215	27.67%	48	0.83%	17,444,504	7.27%	1,216	10.37%	342.50 MHz
BF	262,488	458,696	262,915	28.18%	48	0.83%	17,448,600	7.27%	1,216	10.37%	334.38 MHz

(c) C++

表 6: FD3D カーネルのリソース使用率

		Error Rate
FP64 (E:11,M:52)	OpenCL	1.42E-15
	HDL	1.48E-16
FP32 (E:8,M:23)	OpenCL	1.53E-07
	HDL	1.50E-07
	HLS	1.50E-07
FP25 (E:8,M:16)	HDL	1.58E-05
	HLS	1.58E-05
FP24(E:8,M15)	HLS	2.53E-05
FP16 (E:5,M:10)	OpenCL	7.40E-04
	HDL	7.44E-04
	HLS	7.44E-04
BF (E:8,M:7)	HDL	9.93E-03
	HLS	9.93E-03

表 7: FD3D 問題サイズ $504 \times 504 \times 504$, 5 *timesteps*, $radius = 4$ の場合の正規化した誤差

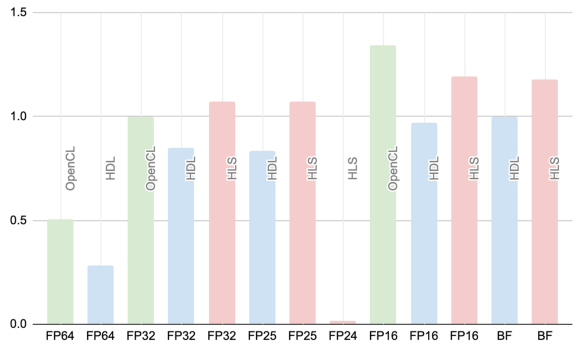

図 19: FD3D 問題サイズ $504 \times 504 \times 504$, 5 *timesteps*, $radius = 4$ の場合の FP32 の OpenCL を基準として正規化したスループット

FD3D はアプリケーションとして、ステンシル計算の特徴上パイプライン化しやすと言う特徴がある。そのため精度による性能の変化が確認できた。

OpenCL の基本記述の場合がその特徴が特に見えていて、倍精度は単精度の約 0.5 倍、半精度は 1.35 倍の性能の変化が有った。単精度計算に特化されている FPGA の特性を考えると、半精度を使う事による計算リソースの削減による性能向上が可能であることが確認できた。この傾向は HDL、C++ともに確認でき、それぞれの FP32 実装を基準とすると、HDL は BF の場合約 1.18 倍、C++は FP16 の場合約 1.11 倍の性能向上が有った。

ごさに関しては結果で確認できるように、仮数ビットに比例して誤差が小さくなっている事を確認できた。

6. 終わりに

本稿では、OpenCL を用いて任意精度を HDL、C++を用いて実装し、四則演算、GEMM、FD3D を用いて評価を行った。その結果、低精度にすると全体的にスループットは向上しているが、アプリケーションによって、その性能向上と最適な符号化方式が実装方法によって異なる。

四則演算の場合には HDL の FP16 の場合が約 1.05 倍、GEMM では、C++の Bfloat16 の場合に約 1.11 倍、FD3D では OpenCL の場合が約 1.35 倍性能向上ができた。この性能向上は Intel の Stratix10 FPGA だと、FP32 に対応する DSP が搭載されているため、FP32 で最適な回路構成されると言うことを考えると有意味な性能向上だと解析できる。

本研究では、ホスト側で任意精度のビットを生成し、FPGA 側に転送している。現状ではこのホスト側の生成速度が遅く、FPGA のカーネル実行時間の百数十倍掛かっている。ホスト側の API の性能向上が課題である。加えて転送する際のパディングが大きくて、仮に FP21 を転送する場合にも、実際には FP32 と同じサイズを転送している。これらを含めて全体の処理を効率化する必要がある。

今後は、上記で述べた課題を解決し、より多様なアプリケーションへの適応、実際の電力測定を行っていく予定である。

謝辞

本研究の一部は，JSPS 科研費 18H03246, 19H04122 の助成を受けたものです．本研究成果の一部は，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) を利用して得られたものです．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実施しました．深く感謝いたします．

参考文献

[1] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. “FBLAS: Streaming Linear Algebra on FPGA”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[2] GOOGLE. *Cloud TPU*. `https://cloud.google.com/tpu/`. [Online; accessed 28-Nov-2019]. 2018.

[3] John L Gustafson and Isaac T Yonemoto. “Beating floating point at its own game: Posit arithmetic”. In: *Supercomputing Frontiers and Innovations* 4.2 (2017), pp. 71–86.

[4] Intel. *Intel® FPGA SDK for OpenCLTM Pro Edition Programming Guide*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl_programming_guide-19-4.pdf`. [Online; accessed 17-November-2020]. 2019.

[5] Intel. *Intel® FPGA SDK for OpenCLTM Pro Edition-Best Practices Guide*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf`. [Online; accessed 9-July-2020]. 2020.

[6] Intel. *Intel® High Level Synthesis Compiler Pro Edition Reference Manual 19.4*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/archives/mnl-hls-reference-19-4.pdf`. [Online; accessed 17-November-2020]. 2020.

[7] Intel. *intel® Stratix® 10 Gx/Sx Product table*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-product-table.pdf`. [Online; accessed 9-July-2020]. 2020.

[8] intel. *Intel® High Level Synthesis Compiler*. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`. [Online; accessed 28-Nov-2019]. 2018.

[9] Piotr Luszczek Jack Dongarra and Yaohung (Mike) Tsai. *”HPL-AI Mixed-Precision Benchmark Results”*. `"https://icl.bitbucket.io/hpl-ai/results/"`. [Online; accessed 20-November-2020]. 2019.

[10] Piotr Luszczek Jack Dongarra and Yaohung (Mike) Tsai. *HPL-AI Mixed-Precision Benchmark*. `https://icl.bitbucket.io/hpl-ai/`. [Online; accessed 20-November-2020]. 2019.

		Throughput	Kernel fmax
FP64	OpenCL	299.580 GFLOPS	312.59 MHz
	HDL	209.760 GFLOPS	209.24 MHz
FP32	OpenCL	306.950 GFLOPS	305.06 MHz
	HDL	267.010 GFLOPS	265.45 MHz
	HLS	318.150 GFLOPS	321.02 MHz
FP25	HDL	297.560 GFLOPS	298.32 MHz
	HLS	312.490 GFLOPS	311.33 MHz
FP24	HDL	280.860 GFLOPS	280.42 MHz
	HLS	308.620 GFLOPS	306.27 MHz
FP21	HDL	297.390 GFLOPS	299.31 MHz
	HLS	325.880 GFLOPS	327.22 MHz
FP16	OpenCL	323.980 GFLOPS	323.62 MHz
	HDL	291.590 GFLOPS	289.26 MHz
	HLS	331.880 GFLOPS	333.88 MHz
BF	HDL	323.860 GFLOPS	324.14 MHz
	HLS	341.600 GFLOPS	341.99 MHz

表 8: GEMM 問題サイズ $A : 2048 \times 1024, B : 1024 \times 1024, C : 2048, \times 1024$ 場合のスループット

		Throughput	Kernel fmax
FP64	OpenCL	1.96 Gpoints / sec	301.11 MHz
	HDL	1.09 Gpoints / sec	151.28 MHz
FP32	OpenCL	3.88 Gpoints / sec	370.91 MHz
	HDL	3.30 Gpoints / sec	230.14 MHz
	HLS	4.16 Gpoints / sec	347.46 MHz
FP25	HDL	3.24 Gpoints / sec	225.83 MHz
	HLS	4.16 Gpoints / sec	328.29 MHz
FP24	HLS	0.07 Gpoints / sec	224.76 MHz
FP16	OpenCL	5.22 Gpoints / sec	358.42 MHz
	HDL	3.76 Gpoints / sec	267.52 MHz
	HLS	4.63 Gpoints / sec	342.81 MHz
BF	HDL	3.89 Gpoints / sec	276.85 MHz
	HLS	4.57 Gpoints / sec	334.67 MHz

表 9: FD3D 問題サイズ $504 \times 504 \times 504$, 5 $timesteps$, $radius = 4$ の場合のスループット

[11] Jeff Johnson. *facebookresearch/deepfloat*. https://github.com/facebookresearch/deepfloat. [Online; accessed 17-Nov-2020]. 2018.

[12] Jeff Johnson. "Rethinking floating point for deep learning". In: *arXiv preprint arXiv:1811.01721* (2018).

[13] Sparsh Mittal. "A Survey of Techniques for Approximate Computing". In: *ACM Comput. Surv.* 48.4 (Mar. 2016). ISSN: 0360-0300. DOI: 10.1145/2893356. URL: https://doi.org/10.1145/2893356.

[14] NVIDIA. *TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x*. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/. [Online; accessed 9-July-2020]. 2020.

[15] Walter Tichy. "The end of (numeric) error: An interview with John L. Gustafson". In: *Ubiquity* 2016.April (2016), pp. 1–14.

[16] Hasitha Muthumala Waidyasooriya et al. "OpenCL-based FPGA-platform for stencil computation and its optimization methodology". In: *IEEE Transactions on Parallel and Distributed Systems* 28.5 (2016), pp. 1390–1402.

[17] xilinx. *Vivado Design Suite - HLx Editions*. https://www.xilinx.com/products/design-tools/vivado.html. [Online; accessed 20-Nov-2020]. 2020.

[18] Hamid Reza Zohouri et al. "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs". In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2016, pp. 409–420.

[19] 丸山直也, 松田元彦, 松岡聡, et al. "OpenCL による FPGA の予備評価". In: **研究報告ハイパフォーマンスコンピューティング** *(HPC)* 2015.44 (2015), pp. 1–5.

[20] 先端共同 HPC 基盤施設（JCAHPC）. **筑波大学 計算科学研究センター 学際共同利用プログラム**. https://www.ccs.tsukuba.ac.jp/wp-content/uploads/sites/14/2020/03/Cygnus.pptx. [Online; accessed 18-Nov-2020]. 2020.

[21] 先端共同 HPC 基盤施設（JCAHPC）. **計算科学研究センター スーパーコンピュータ *Cygnus* の概要**. https://www.ccs.tsukuba.ac.jp/wp-content/uploads/sites/14/2018/12/Cygnus.pdf. [Online; accessed 9-July-2020]. 2019.

[22] 塙敏博, 三木洋平, et al. "宇宙物理アプリケーションのための FPGA 演算オフローディングの検討". In: **研究報告ハイパフォーマンスコンピューティング** *(HPC)* 2019.9 (2019), pp. 1–7.

[23] 大島聡史 et al. "FPGA を用いた疎行列数値計算の性能評価". In: **研究報告ハイパフォーマンスコンピューティング** *(HPC)* 2016.1 (2016), pp. 1–9.

[24] 藤田典久 et al. "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". In: **研究報告ハイパフォーマンスコンピューティング** *(HPC)* 2017.12 (2017), pp. 1–9.