Vol.2017-HPC-161 No.10
2017/9/20

階層型行列計算のFPGAへの適用

塙 敏博[1,a)]　伊田 明弘[1]　星野 哲也[1]

概要：近年，単精度浮動小数点の演算器を多数内蔵した FPGA(Field Programmable Gate Array) が登場している．加えて，OpenCL を用いた FPGA 実装が可能になったことで，FPGA を用いた HPC アプリケーションが比較的容易に，また効率的に実現できるような環境が整ってきた．
本研究では，階層型行列法ライブラリである $\mathcal{H}$ACApK を用いたアプリケーションを対象として，OpenCL を用いた FPGA 向けの実装について検討した．階層型行列は，密行列の部分行列を低ランク行列を用いて近似し，大きな密行列全体を多数の小密行列 と低ランク近似行列の集合として表すことで，計算量と必要メモリを削減することができる．階層型行列生成を FPGA 上で実行した結果，AVX2 命令を持つ CPU 1 コアを上回る性能を得ることができた．

1. はじめに

科学技術計算において、高速な演算処理が求められる中で、様々なハードウェアが利用されている。今日では，従来のマルチコアプロセッサに加えて．数多くのコアを備えたメニーコアプロセッサや，画像処理用のハードウェアをベースに科学技術計算に転用した GPGPU などにより，多数の演算コアを用いることによりチップ内部の並列度を向上させていくことで性能を高めている．しかし，近い将来半導体プロセスの微細化が限界を迎える「ポストムーア」時代においては，チップ内では，コア単体性能の向上は見込めず，チップ面積の制約から，コア数を増加させることも困難になると考えられている。

そのような中で, 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で汎用プロセッサに比べて効率よく高い性能を発揮しうると考えられる．

そのため様々な用途に対する FPGA の活用が模索されており．例えばデータセンタ内の処理に FPGA を活用する Catapult[1] などが知られている．また国内の HPC 研究分野における FPGA の活用についても．いくつかの例が存在している [3], [4]．しかし．これまで FPGA を用いて特定の処理を実現するためには．Verilog HDL などのハードウェア記述言語 (HDL) を用いて回路レベルで記述を行う必要があった．従って，FPGA 上で動作する一般的な科学技術計算プログラムを作成するには，アルゴリズムを論理回路レベルで詳細に設計する必要があり，大変困難であった．

その中で，回路設計技術に精通していなくても FPGA 上で動作するハードウェアを設計する方法として，OpenCL が利用されるようになってきた．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 [2]．実際にいくつかの FPGA 製品においては．Verilog HDL などを用いることなく OpenCL のみを用いて汎用のプログラム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検討が進められている [5], [6], [7], [8].

一方，これまでの HPC における FPGA の利用は，GPU などのアクセラレータと同様に, PCI Express を介してホストに接続し，CPU からの処理をオフロードする形態であった．しかし近年では，Intel 社により Xeon CPU と Arria 10 FPGA を QPI で接続した製品が発表されたり [9], [27], OpenCAPI[10], CCIX[11]，Gen-Z[12] のように CPU およびアクセラレータ間の接続プロトコルを標準化する動きがある．

本研究では，文献 [18] における報告に引き続き，階層型行列計算の一部を，OpenCL を用いて FPGA 上に実現し，その feasibility について調査する．

2. 階層型行列計算と $\mathcal{H}$ACApK ライブラリ

2.1 階層型行列

本論文では N 次元実正方行列 $\bar{A} \in \mathbb{R}^{N\times N}$ について考える．本論文では，$\bar{A}$ を部分行列に分割した上で，それ

[1] 東京大学 情報基盤センター
[a)] hanawa@cc.u-tokyo.ac.jp

ら部分行列の大半を低ランク行列で近似したものを**階層型行列** A と呼ぶ．ここで，N 次元正方行列の行に関する添え字の集合を $I := 1, \cdots, N$ 列に関する添え字の集合を $J := 1, \cdots, N$ と表す．直積集合 $I \times J$ を重なりなく分割して得られる集合の中で，各要素 m が I と J の連続した部分集合の直積であるものを M とする．すなわち任意の $m \in M$ は $s_m \subseteq I, t_m \subseteq J$ を用いて $m = s_m \times t_m$ と表される．ある m に対応する $\bar{A}$ の部分行列を

$$A|^m_{s_m \times t_m} \in \mathbb{R}^{\#s_m \times \#t_m} \tag{1}$$

と書く．ここで # は集合の要素数を与える演算子である．階層型行列では、大半の m について $A|^m_{s_m \times t_m}$ の代わりに以下の低ランク表現 $\tilde{A}|^m$ を用いる．

$$\begin{aligned} \tilde{A}|^m &:= V_m \cdot W_m \\ V_m &\in \mathbb{R}^{\#s_m \times r_m} \\ W_m &\in \mathbb{R}^{r_m \times \#t_m} \\ r_m &\leq min(\#s_m, \#t_m) \end{aligned} \tag{2}$$

ここで $r_m \in \mathbb{N}$ は行列 $\tilde{A}|^m$ のランクである。すなわち、低ランク行列 $\tilde{A}|^m$ とは、密行列 $A|^m_{s_m \times t_m}$ を V_m と W_m の積により近似した行列である。**図 1** に階層型行列の一例を示す。図 1 の濃く塗りつぶされた部分行列が $A|^m_{s_m \times t_m}$ に、薄く塗りつぶされた部分行列が $\tilde{A}|^m$ に対応する。

本論文では階層型行列に関する演算，特に行列・ベクトル積に関して論じる．ある階層型行列 A に関するデータ量 $N(M)$ は，m に対応する部分行列に関するデータ量 $N(m)$ を用いて以下のように表される。

$$N(M) = \sum_{m \in M} N(m) \tag{3}$$

$$N(m) := \begin{cases} \#s_m \times \#t_m & \text{m が密行列の場合} \\ r_m \times (\#s_m + \#t_m) & \text{m が低ランク行列の場合} \end{cases} \tag{4}$$

r_m が $\#s_m$ や $\#t_m$ と比べて十分に小さい場合，$r_m \times (\#s_m + \#t_m)$ は $\#s_m \times \#t_m$ と比べて小さい値となり，低ランク行列による表現がデータ量の点で有利となる．その結果，密行列を用いる場合と比べ，行列・ベクトル積等の行列演算に必要な演算量や行列の保持に必要なメモリ量を低減することができる．

2.2 対象とするコード

本稿では，階層型行列計算ライブラリとして，ppOpen-APPL/BEM ver.0.5.0 [14] に含まれる $\mathcal{H}$ACApK ライブラリ利用版のリファレンス実装を用いる．ppOpen-APPL/BEM とは，JST CREST「自動チューニング機構を有するアプリケーション開発・実行環境: ppOpen-HPC」

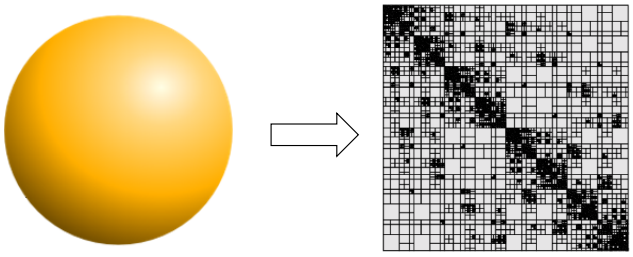

図 1 静電場解析対象と $\mathcal{H}$ACApK により生成される行列分割構造の例

[15] の構成要素の一つであり，境界要素法 (Boundary Element Method, BEM) 用のソフトウェアフレームワークである．

本ソフトウェアでは，境界要素法において係数行列として出現する密行列を $\mathcal{H}$ACApK ライブラリにより近似するリファレンス実装が提供されており，本稿ではこれをベースラインとする．

図 1 は，本稿で静電場解析の対象とする構造物と $\mathcal{H}$ACApK により生成される行列分割構造の例を示したものである．図 1 中で黒く塗りつぶされた領域は小密行列で表現され，それ以外は低ランク近似行列で表現される．

2.3 解析対象

前節で述べたフレームワークが対象としている解析対象は，以下のように想定される．

3 次元空間 $\mathbb{R}^3$ 内の 2 次元領域を $\Omega \subset \mathbb{R}^3$ とする．領域 Ω 上で定義される関数からなるヒルベルト空間を $\mathcal{H}$ とし，その双対空間を $\mathcal{H}'$ とする．関数 $u \in \mathcal{H}, f \in \mathcal{H}'$ として, 次の積分方程式を考える．

$$\int_\Omega \mathcal{G}(x,y)\, u(y)\, \mathrm{d}y = f(x) \tag{5}$$

ここで, $\mathcal{G} : \mathbb{R}^3 \times \Omega \to \mathbb{R}$ は積分核と呼ばれる関数である．解くべき方程式が 3 次元空間 $\mathbb{R}^3$ における微分方程式として定義されている場合, 境界要素法では, 微分演算子に対応するグリーン関数を積分核に選ぶことで, 微分方程式を式 (5) の積分方程式に書きなおすことができる．本稿では積分核 $\mathcal{G}$ が次式で表される場合に解析対象を限定する．

$$\mathcal{G}(x,y) \in \mathrm{span}(|x-y|^{-p}, p > 0) \tag{6}$$

ここで $p > 0$ は正定数である．

完全導体の球体を地面から離して空間に置いた場合の静電場解析の例を**図 2** に示す．この場合，式 (5) のカーネル関数は ϵ を誘電率として，次式に書き下すことができる．

$$\mathcal{G}(x,y) = \frac{1}{4\pi\epsilon}|x-y|^{-1} \tag{7}$$

2.4 階層型行列計算ライブラリ $\mathcal{H}$ACApK の実装概要

以下では，$\mathcal{H}$ACApK ライブラリの実装の概要について

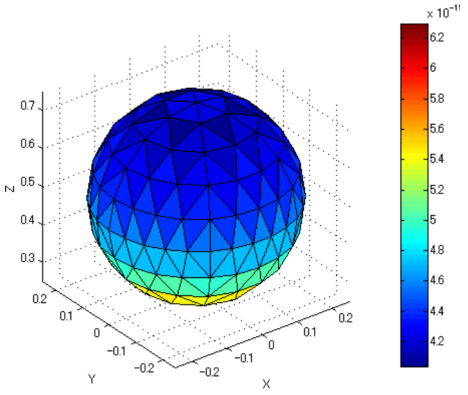

図 2 完全導体の球体における表面電荷密度の計算結果および導体表面の三角要素への分割 [25]

述べる．$\mathcal{H}$ACApK は主に，
(1) 階層型行列の生成
(2) 階層型行列を係数行列に持つ線形方程式のための線形ソルバ
の二つのパートから構成される．

2.5 階層型行列の生成

本稿では FPGA に実装する上で必要な部分のみの説明にとどめる．詳細は [25] を参照されたい．

$\mathcal{H}$ACApK の階層型行列の生成は，以下の 3 ステップにより行われる．
(1) 幾何情報に基づくクラスタリング（**図 3**：左）．
(2) 階層型行列構造の作成（図 3：中央）．この時点では部分行列のフレームを作成するだけで，行列要素は計算されない．
(3) 部分行列の計算（図 3：右）．低ランク近似可能な部分行列については ACA(Adaptive Cross Approximation [26], **図 4**) を用いて近似部分行列を生成し，近似不可能と判定された部分は密行列として計算される．

このうち，ステップ (3) の疑似コードを**図 5** に示す．1 行目の部分行列のループを MPI+OpenMP により並列化している．部分行列の計算開始時点では近似に必要なベクトルの本数（ランク数）k が未知であるため，5 行目では作業用に十分大きな配列を確保している．作業用配列はランク k の確定後に改めて確保した低ランク近似用の領域に値をコピーした後解放することで，メモリの無駄遣いを防ぐ実装となっている．

また，7-8 行目及び 20 行目の列・行ベクトルと小密行列の生成において，要素の計算方法はユーザーが定義するという手法を採用していることが $\mathcal{H}$ACApK の特徴的なところである．**図 6** は，行列要素値を返す擬似関数である．この関数は，$\mathcal{H}$ACApK を使用するためにユーザーが実装する必要があり，低ランク近似を行う際に繰り返し呼び出される．

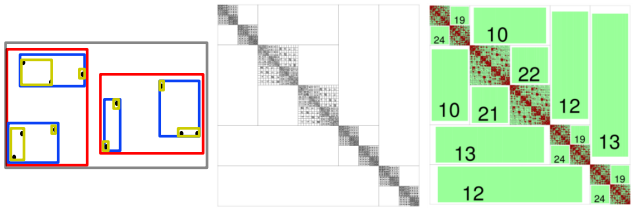

図 3 左：Cluster tree の作成，中央：階層型行列構造の作成，右：部分行列の計算（数字は部分行列の低ランク近似後のランク）

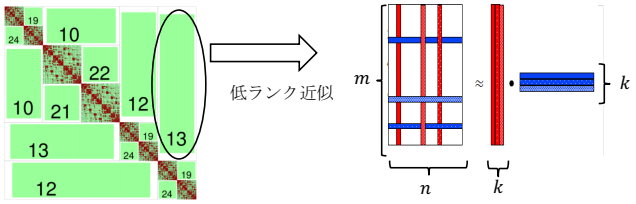

図 4 ACA による低ランク近似．$m \times n$ 行列を k 本の列ベクトルと k 本の行ベクトルの直積により近似する．k が大きくなるほど近似誤差が小さくなることが期待され，k の値により使用メモリ量と近似精度を制御する．

```
1  do i=1, NUM_SUBMTX  !部分行列のループ
2     m = leaf(i)%m !部分行列の列の長さ
3     n = leaf(i)%n !部分行列の行の長さ
4     if leaf(i)%isLowRank then !低ランク近似可能の時
5        allocate(mk_tmp(m,KMAX),kn_tmp(n,KMAX))
6           do k=1, KMAX
7              mk_tmp(:,k) = 列ベクトルの選択・生成
8              kn_tmp(:,k) = 行ベクトルの選択・生成
9              zeps = 近似誤差の計算
10             if zeps < EPS exit
11          enddo
12          leaf(i)%k = k
13          allocate(leaf(i)%mk(m,k))
14          allocate(leaf(i)%kn(n,k))
15          leaf(i)%mk(:,1:k) = mk_tmp(:,1:k)
16          leaf(i)%kn(:,1:k) = kn_tmp(:,1:k)
17       deallocate(mk_tmp,kn_tmp)
18    else !小密行列の時
19       allocate(leaf(i)%mn(m,n))
20       leaf(i)%mn(:,:) = 小密行列の生成
21    endif
22 enddo
```

図 5 部分行列の計算の疑似コード（NUM_SUBMTX：部分行列の数，leaf(i)：i 番目の部分行列，KMAX：ベクトル本数 k の打ち切り値，EPS：必要とする近似精度，leaf(i)%k：i 番目の部分行列の近似に必要であったベクトル本数，leaf(i)%mk/kn：i 番目の部分行列を近似するための k 本の列/行ベクトル，leaf(i)%mn：i 番目の部分行列=小密行列）

2.6 階層型行列ベクトル積

$$Ax \to y, \quad x, y \in \mathbb{R}^N \tag{8}$$

本論文ではこの演算の実施手順として，各部分行列毎に行列積を実行し，その結果を統合することで最終的な結果 y を得るという，最も自然でかつ効率的と考えられる方法を

```
real*8 function HACApK_entry_ij(i,j,st_bemv)
   integer :: i,j
   type(st_HACApK_calc_entry) :: st_bemv
         !計算内容はユーザーが実装
   return HACApK_entry_ij
end function
```

図 6 密行列上の要素番号 i, j が与えられた時，行列要素値を返す関数．ユーザが実装する必要がある．

採用する．密行列により表現されている部分行列 $A|_{s_m \times t_m}^m$ については

$$A|_{s_m \times t_m}^m \cdot x|_{t_m} \to \hat{y}|_{s_m} \tag{9}$$

を計算する。ここで、$x|_{t_m}$ は x の各要素のうち t_m に対応する要素のみを抜き出して生成した $\#t_m$ 個の要素からなるベクトルである。$\hat{y}|_{s_m}$ は $\#s_m$ 個の要素を持つベクトルであり、各要素は y の s_m に対応する要素の部分積の一つとなる。

次に、低ランク表現が用いられている行列 $\tilde{A}|^m$ に関しては、$c \in \mathbb{R}^{r_m}$ として、まず

$$Wm \cdot x|_{t_m} \to c|_{r_m} \tag{10}$$

を計算し、さらに

$$Vm \cdot c|_{r_m} \to \hat{y}|_{s_m} \tag{11}$$

を計算することで、$\tilde{A}|^m \cdot x|_{t_m} = Vm \cdot Wm \cdot x|_{t_m} \to \hat{y}|_{s_m}$ を得る。

それぞれの部分行列について $\hat{y}|_{s_m}$ を計算した後、

$$\sum_{m \in M} \hat{y}|_{s_m} \to y \tag{12}$$

のようにこれらを統合し、最終的な結果とする。

2.7 対象とする行列

本稿では**表 1** に示す 3 つの階層型行列を扱う．これらの行列はいずれも境界要素法を用いた静電場解析において現れる行列である．物体表面の形状はそれぞれ異なるが，積分核はいずれも式 (7) で表される．

表中のリーフ数とは低ランク行列 による近似行列の組数および小さな密行列数の合計数である．階層型行列における近似行列または小さな密行列はツリーにおける葉 (リーフ) に相当するため，本稿ではリーフという呼称を用いている．

表 1 対象とする階層型行列の構成

行列名	100ts	216h	human_1x1
行数	101250	21600	19664
総リーフ数	222274	50098	46618
近似行列個数	89534	17002	16202
小密行列数	132740	33096	20416

3. OpenCL による FPGA プログラミングと性能最適化

3.1 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，例えば C 言語や Fortran を用いれば数行で実装できるような単純な処理を行うだけでも，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．

また研究レベルでは高級言語から Verilog HDL などに変換する試みも数多く行われているが，実用レベルには及ばない．

しかし近年では，OpenCL 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．Intel(旧 Altera) 社の FPGA では Stratix V シリーズから OpenCL への対応が始まっており，Verilog HDL などを直接用いることなく，OpenCL のみで FPGA 向けのプログラムが作成可能となっている．

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．

これまで Intel 社では，ホスト CPU の演算の一部をオフロードするためのアクセラレータとして FPGA を利用できるようなツールを提供している．この場合，FPGA は PCI Express 拡張ボードの形でホストに装着されるのが一般的である．

OpenCL によりオフロード機能を記述して FPGA 上で実行するためには，以下のような機能が必要であり，Stratix V によって初めて実用的になったと考えられる．

- ホストと FPGA 間が PCI Express で直接接続されていること
  アクセラレータとして用いるためには，GPU などと同様に，高速汎用 I/O である PCI Express を用いて接続する必要がある
- FPGA の内部が部分再構成 (Partial reconfiguration) 可能であること
  FPGA の PCI Express インタフェース，ならびに拡張ボードに搭載された DDR メモリインタフェースなどは，ボードが変わらない限り不変であり，特に PCI Express インタフェースが変更されてしまうと，ホストが停止してしまう．したがって，これらのインタフェースを除き，OpenCL のカーネルに相当する範囲だけを再構成できるような部分再構成機能が必要である．
- PCI Express 経由で FPGA 内部が再構成できること
  OpenCL のカーネルとして動作するため，カーネル実行前に FPGA 構成情報（コンフィグレーションデータと言う）をホストから FPGA にダウンロードする必要がある．その際，コンフィグレーションデータは数十 MB を超えるため，PCI Express 経由で転送し，かつデータを受信すると同時に内部を再構成するような仕組みが必要である．

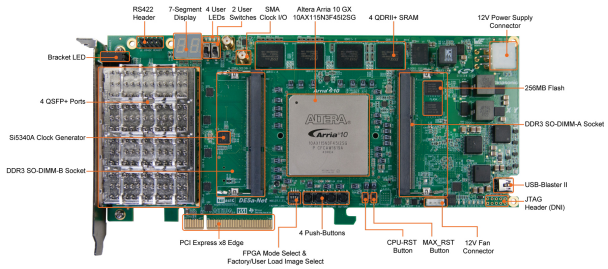

図 7 Terasic 社 DE5A-Net Arria 10 FPGA Development Kit (Teraric 社提供)

本研究では，FPGA として Intel 社の FPGA である Arria 10 GX 1150 が搭載された，Terasic 社の PCI Express ボード DE5a-Net Arria 10 FPGA Development Kit (**図 7**) を用いる．

本 FPGA は，**表 2** に示すように，Adaptive Logic Module (ALM) と呼ばれる論理モジュール 427,200 個で構成されており，各モジュールは 4 個のレジスタ，2 個の 8 入力 2 出力 Look Up Table (LUT) および 2 個の全加算器とマルチプレクサから構成されている．さらに，FPGA チップ内部には 2,713 個の 20Kbit からなる RAM ブロック (M20K) が含まれ，それとは別に，640bit からなる Memory Logic Array Block (MLAB) 20,774 個も使用することができる．

また，これらとは別に，固定小数点および浮動小数点算術演算をサポートする，可変精度 DSP (Digital Signal Processor) を備える．DSP 単体で，単精度浮動小数点の乗算，加減算，積和演算をサポートしている．チップ全体としては，1,518 個の DSP を備え，理論ピーク性能としては，単精度で 1.37 TFLOPS の演算能力を持つ．また，倍精度演算を行うためには，DSP を 4 つと付加回路を用いることで実現できる．旧世代の Stratix V においては，DSP 単体では整数可変ビット長しか扱えなかったため，浮動小数点演算器を実現するためには様々な周辺回路が必要であり，上記の ALM や RAM ブロックも多数消費していたが，Arria 10 以降では，浮動小数点演算をサポートする DSP が搭載されたことで，より多くの浮動小数点演算を実現することができる[*1]．

OpenCL コードのコンパイルを行うには，ユーザは Intel FPGA SDK for OpenCL Offline Compiler (実際には aoc コマンド) を実行するだけである．それによって，内部では以下のような処理が行われる．

- “aoc -c kernel.cl” で行われる処理
  (1) OpenCL のプログラム構造から，パイプラインステージを切り出し，ステートマシンを構成する．

[*1] 次世代の Stratix 10 では倍精度演算を実現する DSP を内蔵するとアナウンスされている．

表 2 対象とする FPGA 製品の仕様および文献 [18] で用いた製品との比較

	本研究で使用するもの	前回 [18]
FPGA	Intel Arria 10 GX E1 (10AX115N2F45E1SG)	Altera Stratix V GS D5 (5SGSMD5K2F40C2)
#Logic units (ALMs)	427,200	172,600
#RAM blocks (M20K)	2,713	2,014
#DSP blocks	1,518 (浮動小数点)	1,590 (整数のみ)
ボード	Terasic DE5a-Net E1	Bittware S5-PCIe-HQ GSMD5
DDR メモリ容量	(4 + 4) GB	(4 + 4) GB
DDR メモリバンド幅	17.1 GB/sec	25.6 GB/sec
PCIe I/F (OpenCL の場合)	Gen3 x8	Gen2 x8
ツール	QuartusPrime Pro 16.1.1	Quartus II 16.0.1

(2) 必要な資源を見積もる．(ロジック使用率，レジスタ使用率，メモリ使用率，DSP 使用率)
(3) PCI Express や DDR3-DRAM インタフェースなどのモジュール，OpenCL を元に Verilog HDL の記述が生成される．
(4) kernel.aoco ファイルの出力

- “aoc kernel.aoco” で行われる処理
  (1) Quartus (Intel 社の FPGA 向け論理合成ツール) を起動し，論理合成，配置配線等を行う．
  (2) FPGA コンフィグレーションデータのビットストリームを含む kernel.aocx ファイルが出力される．

ツールの使い方としては，非常に容易である反面，現状では依然として以下のような課題がある．

- コンパイルに非常に時間がかかる
  aoco ファイルから aocx ファイルに変換する部分では，コンパイラ内部で Quartus ツールを起動して，論理合成および，FPGA デバイスへの配置配線などが行われる．現状ではどんなに簡単な記述でも，1 回のコンパイルで Intel Xeon E5 (Haswell プロセッサ) を用いても 2 時間近くかかる．本来であれば，インタフェース部分などの回路ブロックは不変なはずであり，OpenCL のコードに対応する部分のみを部分再構成で済むはずである．
  今後の FPGA デバイスやツール群の改良により，必要最小限部分の合成やマッピングなどでコンパイル時間が短縮されることが望まれる．
- 設計時にハードウェア資源，性能の予測が難しい
  FPGA 内部に含まれるハードウェア資源をどの程度使用するかをレポートする機能が提供されており，コンパイラに`--report -c` オプションを与えることで利用することができる．また，パイプラインステージの構成についても，クリティカルパスや，依存関係などのメッセージが表示され，性能を改善するためのヒント情報なども含まれている．しかし，FPGA に収まるかどうかの目安にしかならず，最終的には上記の通り，長時間の論理合成の結果を待つ必要がある．動作周波数や，最終的なステートマシンの情報については，事前に予測することはできないため，結果を見ながらトライ&エラーでソースコードの改良を進める必要がある．

3.2 FPGA に向けた OpenCL 記述

OpenCL は C++言語を元にした並列化プログラミング環境であり，接頭辞を用いて関数や変数に対してその実行場所や配置場所といった追加情報を与えるという言語拡張が行われている．またデバイス間でのデータ通信などの機能 (API 関数) も提供されている．言語仕様の策定において GPU での利用が強く意識されていたこともあり，OpenCL のプログラム記述方法や実行モデルは CUDA[21] と類似点が多い．OpenCL を用いた並列化プログラミングは，CPU やメニーコアプロセッサにて広く用いられている OpenMP や GPU 向けの主要な並列化プログラミング環境である CUDA と比べるとプログラム記述量などの点で優れているとは言い難い．しかし OpenCL のみを用いて FPGA プログラミングが行えることは科学技術計算に FPGA を使用したい利用者にとっては大きなメリットである．

現在の OpenCL はバージョン 2.0 が最新版であるが，現在の Offline Compiler 16.1 では バージョン 2.0 の一部までをサポートしている．

通常 OpenCL コードは実行時にコンパイルされるが，FPGA では論理合成に長時間を要するので，事前に offline compiler によりバイナリ（実際にはコンフィグレーションデータ）を生成する必要がある．

FPGA 向けの高性能な OpenCL プログラムを作成するためには様々な最適化を行う必要がある．特に FPGA を使う場合には，ハードウェアの構成自体を利用者がある程度自由に指定できる点が特徴的である．また GPU 向けの OpenCL 最適化プログラミングにおいては，GPU 上の大量の演算器を十分に活用できるように非常に高い並列度を

持つプログラムを記述することが非常に重要である一方，FPGA はハードウェア資源の制約から GPU のような高い並列度には向いていない．そのため同じ OpenCL を用いるものの，FPGA 向けのプログラムには GPU とは異なる最適化プログラミング戦略が必要である．

3.3 最適化

Intel 社の FPGA に向けた最適化プログラミング手法については Intel 社によるプログラミングガイド [23] や最適化ガイド [24] などの公開情報に詳しく紹介されている．本稿では特に

- メモリの使い分け
- コード記述レベルの最適化
- ループアンローリング
- SIMD 化

に着目し，次章では実際にプログラムを作成してその効果を確認する．

3.3.1 メモリの使い分け

OpenCL においては数種のメモリを使い分けることができる．__global 接頭辞を付けた配列は，FPGA ボード上の DDR メモリ上に確保され，ホストとのやりとりなどに使用することができる．

一方，__local 接頭辞を付けた配列は，FPGA 内部の RAM ブロックなどで構成されるオンチップメモリ上に確保される．しかしながら，OpenCL カーネル内部でしか利用することができない．したがって，一旦__global 配列から__local 配列にデータをコピーした後に，__local 配列を計算に使用し，その結果を__global 配列に書き戻す必要がある．

3.4 コード記述レベルの最適化

前節までに述べた最適化手法はプログラムの構造自体を変化させない最適化であった．本節ではプログラムの構造を変化させるようなコード記述レベルの最適化について述べる．

OpenCL コンパイラでは主に for 文や while 文などのループ構造を解析し，ハードウェアレベルのパイプラインに変換する．

Intel FPGA OpenCL Compiler (AOC) が出力するログの例を**図 8** に示す．このログを見ると，実際に for 文を手がかりにして解析を行い，各パイプラインステージに変換していることがわかる．また，各ステージ内で使用される演算器のレイテンシや，クリティカルパスを計算し，自動的にステージを複数サイクルに分割する．

また，FPGA においては，ハードウェアの使用量を抑える工夫も必要である．

通常の CPU プログラムであれば，キャッシュの効率なども考慮して，例えば初回の反復で実行する処理と，その後の残りの反復処理を分離して記述するような場合がある．しかし，ハードウェア資源の制約と，その処理に特化したパイプラインが生成されることを考えると，なるべく共通化できる部分は共通化しておく方がよい場合がある．内部に分岐を含む処理であっても，ハードウェアでは，単にセレクタによって信号線が選択されるだけであり，性能にはほとんど影響がない．また，全体を通してパイプラインの 1 ステージの処理時間が他のステージによって決まるような場合であれば，冗長な計算をしても性能にあまり影響はないため，例えば 0 と掛け算を意図的に行うことで不要な項を削除するなどして，回路を共通化することが可能である．

```
================================================================================
                        *** Optimization Report ***
...
================================================================================
Kernel: hacapk_body
================================================================================
The kernel is compiled for single work-item execution.
Loop Report:
 + Loop "Block1" (file hacapk-calc0.cl line 36)
 | NOT pipelined due to:
 |
 |   Loop structure: loop contains divergent inner loops.
...
 |
 |-+ Loop "Block4" (file hacapk-calc0.cl line 53)
 | | Pipelined with successive iterations launched every 2 cycles due to:
 | |    ...
 | |-+ Loop "Block5" (file hacapk-calc0.cl line 55)
 |     Pipelined with successive iterations launched every 8 cycles due to:
 |    ...
 | |-+ Loop "Block9" (file hacapk-calc0.cl line 62)
 |     Pipelined well. Successive iterations are launched every cycle.
```

図 8 AOC の出力ログ例

これらのことから，逐次実行 (single stream) において高い性能を実現するためには，

- 各 for 文の中に含まれる処理量が，おおよそ均等，または整数倍となり，バランスが取れること
- 共通化できそうな文はまとめること
- メモリアクセスは最小化すること

などが挙げられる．

3.4.1 ループアンローリング

一般的な CPU 向けのループアンローリングは，ループ制御のための命令数を削減するとともに分岐無しで連続実行できる命令数を増加させたり，メモリに対してバースト転送を可能にする効果がある．FPGA においても同様の効果が期待できるうえに，前節で述べたような，ループ単位の計算時間を変化させて計算ブロック毎の計算時間・計算量のバランスを改善しより高速な周波数で動作することを可能とさせる効果もある．

3.4.2 SIMD 化

FPGA は搭載されている資源の制約上，GPU のような非常に高い並列度を持つプログラムがそのまま実行できるわけではない．しかし，同様の並列化自体は可能であり，それによって OpenCL カーネル起動オーバヘッドを低減することができ，資源量にあわせた適切な並列化を行うことで性能向上が期待できる．

OpenCL では clEnqueueNDRangeKernel 関数を用いて FPGA カーネル関数を実行するが，この関数の引数には実行時の並列度を与えることが可能である．FPGA カーネル関数側では実行時に自身の ID を得る API 関数が用意され

ているため，この ID を用いて自身の計算するべき範囲を決めるなどの方法により並列処理が実現可能である．この実装方法は CUDA を用いた GPU プログラミングなどと類似しており，高い性能が期待される並列度には差があるものの，GPU 向けに実装されたプログラムを FPGA 向けに移植する際には低い移植コストにて利用可能な最適化手法であると考えられる．

さらに OpenCL を用いた FPGA プログラミングにおいては，カーネル関数に対して付加できる attribute 情報を用いて並列実行時の動作を制御することができる．たとえば `num_simd_work_items(4)` の指定をすることで SIMD 長が 4 の計算ユニットが作成される．また，`num_compute_units(4)` を指定すれば 4 つの計算ユニットが作成されるが，この場合は，メモリインタフェースが複数必要になってしまうため，FPGA にはあまり適していない．

Offline Compiler は，内部でスレッド ID などを獲得する `get_global_id(0)` などの関数が用いられている場合には NDRange として複数スレッドを想定したコンパイルを実行し，そうでない場合には single stream を想定したコンパイルを行う．

3.5 CPU と統合された FPGA

現在，Intel は，Xeon Broadwell プロセッサと Arria 10 FPGA とを 1 つのチップモジュールにパッケージングした製品をサンプル出荷している [9], [27].

このデバイスにおいては，以下の点でこれまでの PCIe 経由接続の FPGA ボードとは異なる．

- Xeon と FPGA 間の接続が QPI (QuickPath Interconect) PCIe 接続に比べ，バンド幅，レイテンシともに改善が見込まれる．
- Xeon と FPGA 間はキャッシュコヒーレントホスト側のコードで，明示的にバッファを確保するなど，データ管理の必要がなくなる．

この FPGA においても，PCIe 接続ボードと同様に OpenCL を用いてプログラミングできることがアナウンスされている．

4. 実装と評価

4.1 対象問題

我々は，これまで [18] において，階層型行列ベクトル積について Stratix V GX FPGA への実装を試みた．

本論文では，より高い浮動小数点演算性能を持つ Arria 10 GX FPGA を用い，階層型行列ベクトル積の性能について Stratix V の場合と比較するとともに，より計算負荷の重い処理である，階層型行列生成処理への適用について検討する．

実験環境として，Intel Xeon E5-2650v3 (Haswell) 2 ソケット搭載のサーバを用い，その PCI Express スロットに 3.1 節で述べた Terasic 社の Arria 10 搭載 FPGA ボード DE5a-Net Arria 10 FPGA Development Kit を接続した．

表 3 リソース使用量の比較

	今回 (A10)	前回 (S5)
Logic utilization	46,924 (11%)	44,755 (26%)
DSP blocks	4	2
Memory bits	7,164,560 (13%)	6,110,752 (15%)
RAM block	557 (21%)	560 (28%)
fmax	272.1	268.95

表 4 各行列入力に対する実行時間 (ms)

行列	今回 (A10)	前回 (S5)	CPU(IVB)
100ts	9793.5	4848.3	494.2
216h	443.6	684.0	68.7
human_1x1	436.6	547.3	69.6

4.2 階層型行列ベクトル積

$\mathcal{H}$ACApK の行列ベクトル積サブルーチン `HACApK_adot_body_lfmtx` について，FPGA 上での性能を測定した．

FPGA 向けのプログラムとしては，文献 [18] の際に用意した OpenCL プログラムを用いた．これは，このサブルーチンに引数として与えられる，HACApK 階層型行列と入力ベクトルの値を，あらかじめ記録しておいたデータファイルから読み出し，OpenCL カーネルに渡すことで，FPGA 上でオリジナルと同様の計算を実行させる．

オリジナルのコードでは倍精度演算で記述されているが，FPGA 向けの OpenCL 実装においては単精度演算を用いている．

前回の「実装 3: 最も処理時間が短い場合」のコードにおける，リソース使用量，実行時間の比較を行った．**表 3** にリソース使用量，**表 4** に，各行列を入力として用いたときの実行時間を示す．

今回は，同じ実装を使ったにも関わらず，100ts については 2 倍近く遅くなり，他の 2 つの行列では，25%から 55%高速になった．これは，DDR メモリバンド幅が Arria 10 ボードの方が Stratix V ボードより低いことが原因だと考えられる．100ts では小密行列の絶対数が多く，バンド幅が低い影響を受けたものと考えられる．残りの 2 つでは，__local として，オンチップメモリを有効に使えたことにより，性能が向上したものと考えられる．

4.3 階層型行列生成処理への適用

$\mathcal{H}$ACApK ライブラリを用いた境界要素法による静電場解析において，2.5 節で述べた階層型行列生成のうち，ス

表 5 各行列入力に対する face_integral 関数の実行時間 (ms)

行列	A10	CPU(HSW)
216h	1057.4	1160.3
human_1x1	1005.3	1102.1
Sample	5.7	12.5

テップ 3 における部分行列計算の計算のウェイトが高く，必要なデータ量に比べて演算量が格段に大きい．

実際には，図 6 に示した $\mathcal{H}$ACApK ライブラリのユーザが実装する部分，すなわち静電場解析の解析対象から与えられる行列要素値の演算が全体のボトルネックになっている．

そこで，該当する関数 face_integral について，FPGA 向けに OpenCL 実装を行った．

前節と同様に，まずオリジナルの Fortran コードを C 言語に変換し，それを OpenCL に変換した．同時に，関数への入力データとして，各呼び出しごとの，3 要素を持つ変数 ×3 個+スカラ変数 ×3 個をデータファイルに記録した．

その上で，face_integral 関数を同時に複数スレッド処理できるよう，SIMD 化を行った．OpenCL 内では，get_global_id(0) により，入力値を各スレッドが決定して同時に動作する．ホスト側からは，1 回の clEnqueueeNDRangeKernel 関数の呼び出しで，必要な行列生成を（見かけ上）全て並列に動作するように実装した．

OpenCL については，単精度と倍精度それぞれ実装したが，倍精度については，コンパイラのバグで回路の合成ができなかった．

実行時間を**表 5** に示す．比較のため，CPU 1 コアで，単精度，AVX2 命令を有効にした場合の face_integral 関数の実行時間も併せて示す．

その結果，CPU 1 コアの場合に比べて約 10%上回る性能を得ることができた．また，配布パッケージに含まれている Sample データを用いた場合には，CPU 1 コアの 2 倍以上の性能が得られた．これは，FPGA では入力値に応じて最短のパスを通るように最適化されているのではないか，と考えられるが，調査が必要である．

また，今回は Fortran を OpenCL にほぼ単純に変換したのみであり，最適化を検討する必要がある．

5. おわりに

本稿では，階層型行列計算の FPGA への適用について検討し，OpenCL による実装を行った．

階層行列生成については，Arria 10 FPGA により，AVX2 を持つ CPU 1 コアに比べて，最大で約 10%高い性能向上が得られた．但し，CPU 1 ソケットに内蔵されたコア数を考えると，FPGA に向けた最適化は不十分であると考えられる．

今回用いた Terasic 製のボードにおいて，実験中不安定な部分があったり，ボードには搭載されているが，OpenCL からは有効になっていない機能がいくつかある．今後はボードを使うための Board Support Package について，アプリケーションに役に立つ機能を追加するなども検討したい．

さらに今後は，QPI によって CPU と FPGA とが接続された環境において，FPGA の有効活用を検討していく予定である．

謝辞 本研究の一部は，JSPS 科研費 15K00166 の助成を受けたものです．本研究で用いた Quartus II のライセンスの一部は，Intel/Altera 社 University Program によります．

参考文献

[1] Putnam, A. and Caulfield, A.M. and Chung, E.S. and Chiou, D. and Constantinides, K. and Demme, J. and Esmaeilzadeh, H. and Fowers, J. and Gopal, G.P. and Gray, J. and Haselman, M. and Hauck, S. and Heil, S. and Hormati, A. and Kim, J.-Y. and Lanka, S. and Larus, J. and Peterson, E. and Pope, S. and Smith, A. and Thong, J. and Xiao, P.Y. and Burger, D., A reconfigurable fabric for accelerating large-scale datacenter services, 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), pp.13-24, 2014.

[2] OpenCL - The open standard for parallel programming of heterogeneous systems `https://www.khronos.org/opencl/`

[3] 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.

[4] 上野 知洋, 佐野 健太郎, 山本 悟: メモリ帯域圧縮ハードウェアを用いた数値計算の高性能化, 情報処理学会 研究報告 (2015-HPC-151), 2015.

[5] 丸山 直也, Hamid Reza Zohouri, 松田 元彦, 松岡 聡: OpenCL による FPGA の予備評価, 情報処理学会 研究報告 (2015-HPC-150), 2015.

[6] 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.

[7] ウィッデヤスーリヤ ハシタ ムトゥマラ 他：OpenCL を用いたステンシル計算向け FPGA プラットフォーム，情報処理学会 研究報告 (2016-HPC-154)，2016.

[8] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka, “ Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs,” Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’16), 2016.

[9] PC Watch, Intel、Altera の FPGA「Arria 10 GX」を 1 パッケージに統合した Xeon, `http://pc.watch.impress.co.jp/docs/news/752237.html`, 2016 年

[10] OpenCAPI Consortium, `http://opencapi.org`

[11] CCIX Consortium, Connecting Processor Architecture and Accelerators Seamlessly, `https://www.ccixconsortium.com`

[12] Gen-Z Consortium, `http://genzconsortium.org`

[13] K. Nakajima and M. Satoh and T. Furumura and H. Okuda and T. Iwashita and H. Sakaguchi and T. Katagiri and M. Matsumoto and S. Ohshima and H. Jit-

sumoto and T. Arakawa and F. Mori and T. Kitayama and A. Ida and M. Y. Matsuo and K. Fujisawa and et al., ppOpen-HPC: Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning (AT), Optimization in the Real World, pp.15–35, DOI 10.1007/978-4-431-55420-2_2, 2016.

[14] T. Iwashita, A. Ida, T. Mifune, and Y. Takahashi, Software Framework for Parallel BEM Analyses with H- matrices Using MPI and OpenMP, Procedia Computer Science, Vol. 108, pp. 2200 2209 (2017).

[15] ppOpen-HPC — Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning (AT) http://ppopenhpc.cc.u-tokyo.ac.jp/ppopenhpc/

[16] 塙 敏博, 児玉 祐悦, 朴 泰祐, 佐藤 三久, Tightly Coupled Accelerators アーキテクチャに基づく GPU クラスタの構築と性能予備評価, 情報処理学会論文誌（コンピューティングシステム）, Vol.6, No.4, pp.14-25, 2013.

[17] Yuetsu Kodama, Toshihiro Hanawa, Taisuke Boku and Mitsuhisa Sato, “PEACH2: FPGA based PCIe network device for Tightly Coupled Accelerators,” International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2014), pp. 3-8, Jun. 2014.

[18] 塙 敏博, 伊田 明弘, 大島 聡史, 河合 直聡, FPGA を用いた階層型行列ベクトル積, 情報処理学会 研究報告 (2016-HPC-155), 2016.

[19] Altera Corporation, Floating-Point IP Cores User Guide, UG-01058, 2015.

[20] Altera, Stratix V Device Handbook, https://www.altera.com/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf

[21] CUDA Dynamic Parallelism, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism

[22] Intel Corporation, Intel FPGA SDK for OpenCL - 概要 https://www.altera.co.jp/products/design-software/embedded-software-developers/opencl/overview.html

[23] Intel Corporation, Intel FPGA SDK for OpenCL Programming Guide 16.1, UG-OCL002, 2017.

[24] Intel Corporation, Intel FPGA SDK for OpenCL Best Practice Guide 16.1, UG-OCL003, 2017.

[25] A. Ida, T. Iwashita, T. Mifune and Y. Takahashi, “ Parallel Hierarchical Matrices with Adaptive Cross Approx w ima-tion on Symmetric Multiprocessing Clusters, ” Journal of Information Processing Vol. 22, pp.642-650, 2014.

[26] Kurtz S., Rain O. and Rjasanow S.: The Adaptive Cross-Approximation Technique for the 3-D Boundary-Element Method, IEEE Trans. Magn., Vol.38(2), pp.421-424 (2002).

[27] Gupta, P. K. “Accelerating datacenter workloads,” 26th International Conference on Field Programmable Logic and Applications (FPL), 2016.