Vol.2016-HPC-155 No.40
2016/8/10

FPGAを用いた階層型行列ベクトル積

塙 敏博[1,a)]　伊田 明弘[1]　大島 聡史[1]　河合 直聡[1]

概要：近年，FPGA(Field Programmable Gate Array) に対して新たな高性能計算向けのハードウェアとして注目が集まっている．FPGA は対象とする処理に合わせた最適な回路構成を用いることで高い性能や高い電力あたり性能を得られる可能性がある．さらに OpenCL を用いてプログラムコードの形で記述するだけで，FPGA 上のハードウェアとして容易に実現が可能な環境が整ってきた．
本研究では，階層型行列における行列ベクトル積演算を対象に，FPGA 上に実装を行う．階層型行列は，小さな密行列と低ランク近似行列から構成される．階層型行列ベクトル積を行うには，これら構成行列に依存して入り組んだ処理が必要となる．このような問題に対して OpenCL を用いて FPGA 向けの実装を行い、コードの最適化方法と性能について比較する．

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，CPU 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て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と，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．したがって，次世代の HPC に向けてハードウェア・ソフトウェアの両面からの解決が必要となっている．

高い電力あたり性能を実現しうるハードウェアとして．再構成可能なハードウェアである FPGA (Field Programmable Gate Array) が注目されている．FPGA は回路を動的に再構成することができるため．対象とする問題にあわせて最適な回路を構成することができれば高速かつ低い消費電力で処理を行うことが可能になる．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，極めて困難であった．

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る．そのため HPC 分野における FPGA の活用についても調査・検討が行われつつある [5], [6], [7], [8]．

我々はマルチコア CPU．メニーコアプロセッサ．GPU といった様々なハードウェアを適切に用いて高い並列数値計算性能を得ることや．その技術をライブラリなどの形式で多くの利用者に普及させることに興味を持って研究を行っており．すでに多くの論文発表やソフトウェア・ライブラリの公開などを行っている [9], [10]．また高速なアクセラレータ間通信を実現するために FPGA を用いたノード間通信ハードウェアの開発も行ってきた [11], [12]．さらに我々は数値計算などの HPC アプリケーションに FPGA を活用することにも大きな興味を持っている．

本研究では，階層型行列における行列ベクトル積を扱い，OpenCL を用いて FPGA 上に実装し，その最適化手法について検討し，性能の改善について評価する．

本稿の構成は以下の通りである．2 章では，OpenCL による FPGA プログラミングと性能最適化について述べる．3 章では，階層型行列とその行列ベクトル積について説明

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

する．4 章では，実際に行った最適化と性能評価について述べる．最後に 5 章で本稿をまとめる．

2. OpenCL による FPGA プログラミングと性能最適化

2.1 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．

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り，HPC 分野の研究者からも注目され始めている．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る．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．現在の HPC 分野においては AMD 社の GPU 向けのプログラミング環境として利用されることが多いが，マルチコア CPU はもちろん，メニーコアプロセッサである Xeon Phi や，NVIDIA 社の GPU においても利用可能である．

Altera 社 Stratix V 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，ツールの開発に注力している．一方で，ARM などの組込みプロセッサをハード IP として内蔵している FPGA も登場しており (Xilinx 社 Zynq, Altera 社の Arria SoC など)，これら内蔵プロセッサのためのアクセラレータ機能を OpenCL によって記述できるようにしたものも存在しているが，本研究では前者のみを対象にする．

FPGA をホスト CPU のアクセラレータとして用いる場合，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，Altera 社の 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[*1]．
- 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．

本研究では，FPGA として Altera 社の Stratix V が搭載された，Bittware 社の PCI Express ボード S5-PCIe-HQ (s5phq_d5) (**図 1**) を用いる．

本 FPGA は，**表 1** に示すように，Adaptive Logic Module (ALM) と呼ばれる論理モジュール 172,600 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 6 入力 Look Up Table (LUT) および 2 個の全加算器から構成されている．さらに，FPGA チップ内部には 2,014 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) 8,630 個も使用することができる．また，これらとは別に，整数可変ビット長の Digital Signal Processor (DSP) を持つ．単精度浮動小数点数の演算を扱う場合には，仮数部 27 ビットの加算や乗算器などとして，最大 1,590 個の DSP を使用することができる．ただし Stratix V においては，実際に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する[*2][13][14]．

OpenCL コードのコンパイルを行うには，ユーザは Altera 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．
  (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

[*1] Intel と Altera は共同してプロセッサ間インタコネクトである QPI を用いて結合したプラットフォームを開発中である．
[*2] 次世代の Arria 10, Stratix 10 においては，それぞれ単精度，倍精度浮動小数点演算に対応した DSP が搭載される予定である．

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 (Altera 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．

2.2 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[15] 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

表 1　対象とする FPGA 製品の仕様

FPGA: Altera Stratix V GS D5 (5SGSMD5K2F40C2)	
#Logic units (ALMs)	172,600
#RAM blocks (M20K)	2,014
#DSP blocks	1,590 (27 × 27)
ボード: Bittware S5-PCIe-HQ GSMD5	
DDR メモリ容量	(4 + 4) GB
DDR メモリバンド幅	25.6 GB/sec
PCIe I/F	Gen3 x8
(OpenCL では Gen2 x8 での使用に限定される．)	
ソフトウェア環境	
ツール	Altera 社 Quartus II 16.0.1 OpenCL SDK, Altera Offline Compiler

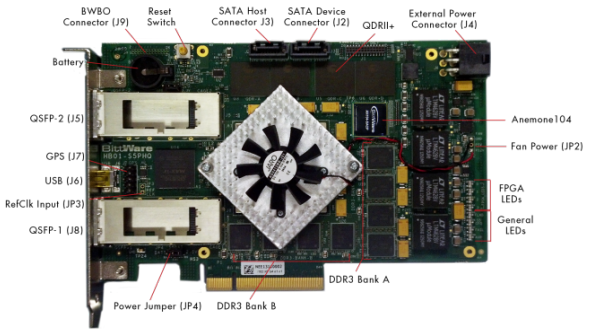

図 1　Bittware 社 S5-PCIe-HQ (Bittware 社提供，ただし本報告で用いるボードには QDR II+は実装されていない)

```
// FPGAデバイスの初期化に関する処理
platform = findPlatform("Altera");
devices.reset(getDevices(platform, CL_DEVICE_TYPE_ALL, &num_devices));
context = clCreateContext(NULL, num_devices, &device, &oclContextCallback, NULL, &status);
queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &status);
// aocxファイルから使いたい関数を読み込む処理
binary_file = getBoardBinaryFile("kernel_source_file_name", device);
program = createProgramFromBinary(context, binary_file.c_str(), &device, num_devices);
status = clBuildProgram(program, 0, NULL, "", NULL, NULL);
kernel = clCreateKernel(program, "kernel_function_name", &status);
// CPU-FPGA間のデータ転送に関する処理
buf = clCreateBuffer(context, CL_MEM_READ_WRITE, N*sizeof(float), NULL, &status);
status = clEnqueueWriteBuffer(queue, buf, CL_FALSE, 0, N*sizeof(float), data, 0, NULL, NULL);
status = clSetKernelArg(kernel, argi, sizeof(cl_mem), &buf);
// カーネル関数の実行
status = clEnqueueNDRangeKernel
         (queue, kernel, 1, NULL, &global_work_size, &local_work_size, NULL, NULL, NULL);
// 実行結果の取得
status = clEnqueueReadBuffer(queue, buf, CL_FALSE, 0, N*sizeof(float), data, 0, NULL, NULL);
// 終了(確保した資源を解放する)に関する処理は割愛

// FPGA上で実行されるカーネル関数(配列を2倍する)の例
__kernel void kernel_function(__global float *restrict data){
  int i;
  for(i=0;i<N;i++){
    data[i] *= 2.0f;
  }
}
```

図 2 FPGA を扱う OpenCL プログラムの例

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 Altera Offline Compiler 16.0 では バージョン 2.0 の一部までをサポートしている．

図 2 に単純な OpenCL プログラムの例を示す．この例は FPGA 上で処理を行う一連の手順の例を示している．具体的には，FPGA 向けのバイナリファイルの読み込み，対象となる関数の設定，入出力変数の設定，データ転送，FPGA 上で実行される関数 (以下カーネル関数) の呼び出し，といった処理が行われている．カーネル関数やその引数については__kernel や__global といった接頭辞が付加されており，その役割がコンパイラにも利用者にもわかりやすい．これらの手順および記述方法は CUDA，特に CUDA Driver API を用いたプログラミングと類似している．しかし，OpenCL と CUDA は似ている部分が多い一方で，プログラム記述とハードウェアとの割り当てや実行モデルの考え方まで同様ではないため，高い性能を持つプログラムを作成するためには FPGA に向けた最適化が必要である．

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．

2.3 最適化

Altera 社の FPGA に向けた最適化プログラミング手法については Altera 社によるプログラミングガイド [17] や最適化ガイド [18] などの公開情報に詳しく紹介されている．本稿では特に

- 適切なメモリ種別の指示
- コード記述レベルの最適化
- ループアンローリング
- 細粒度並列化 (SIMD 化，ベクトル化)

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．

2.3.1 適切なメモリ種別の指定

2.1 節にて述べたように，FPGA 上には複数種類のメモリが搭載されており，また OpenCL には利用するメモリを明示する記述方法が用意されている．コンパイラが最適な回路情報を構成するためには適切なメモリ配置情報を明示的に記述することが重要である．

利用頻度の高い具体的な最適化方法の例としては，本ボードでは__global 接頭辞を付けた配列は DDR メモリ上のグローバルメモリとして確保されるため，__local 接頭辞により RAM 上に確保された配列と比べてアクセス性能が低い．そのため，対象データをローカルメモリ (__local 接頭辞を付けた配列) に一時的に格納して利用するなどグローバルメモリへのアクセスを削減することで性能向上が期待できる．

また，グローバルメモリを使用する際も，今回使用するボードでは，メモリが 2 バンク実装されており，各バンクを適切に使い分けることにより，メモリバンド幅を生かすことができる．

2.4 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．

Altera OpenCL Compiler (AOC) が出力するログの例を**図 3** 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していることがわかる．

また，今回用いた FPGA が比較的ロジックエレメント数が少ないこともあり，ハードウェアの使用量を抑える工夫も必要である．

通常のプログラムであれば，キャッシュの効率なども考

```
================================================================================
                        *** 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 3　AOC の出力ログ例

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．

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．

2.4.1　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．

2.4.2　細粒度並列化 (SIMD 化, ベクトル化)

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の実行には適してはいない．しかし，並列化自体は可能であり，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，`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．ただし対象とするプログラムの構造によってはコンパイラの判断により並列化が行われないことや，必要なハードウェア資源量が多くなりすぎてしまいエラーとなることもあり，適切な値を選択することが必要である．

3.　階層型行列とその行列ベクトル積

3.1　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 4** に階層型行列の一例を示す。図 4 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

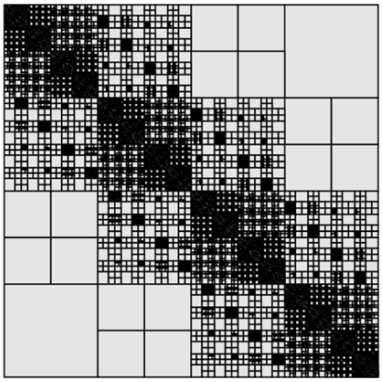

図 4　階層型行列

$N(M)$ 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．

3.2　階層型行列ベクトル積

本稿では、以下のような階層型行列ベクトル積を対象とする．

$$Ax \to y, \quad x, y \in \mathbb{R}^N \tag{5}$$

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

$$A|^m_{s_m \times t_m} \cdot x|_{t_m} \to \hat{y}|_{s_m} \tag{6}$$

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

表 2　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

$$Wm \cdot x|_{t_m} \to c|_{r_m} \tag{7}$$

を計算し、さらに

$$Vm \cdot c|_{r_m} \to \hat{y}|_{s_m} \tag{8}$$

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、

$$\sum_{m \in M} \hat{y}|_{s_m} \to y \tag{9}$$

のようにこれらを統合し、最終的な結果とする。

3.3　対象とするコード

本稿では ppOpen-APPL/BEM ver.0.4.0 に含まれる HACApK 1.0.0 のソースコードを用いる [10]。ppOpen-APPL/BEM 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」[9] の構成要素の 1 つであり、境界要素法 (Boundary Element Method, BEM) の実装において HACApK を用いた階層型行列による計算を行っている [19]。

低ランク近似アルゴリズムについては，ライブラリに含まれている ACA 法ではなく，新たに実装した ACA+法を実装したものを使用している [20].

HACApK のソースコードは Fortran90 によって記述されているが，本研究では，行列ベクトル積計算部分をあらかじめ C 言語化したものを使用している．

3.4　対象とする行列

本稿では**表 2** 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。

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．

4.　実装と評価

4.1　対象問題

我々は，これまで [7] において，FPGA を用いた疎行列の CG 法を対象に FPGA への実装を試みてきた．

本論文では，より局所性を活かすことができる演算として，階層型行列におけるベクトル行列積を対象として，FPGA への実装を行う．

実験環境としては，Intel Xeon E5-2680v2 (IvyBridge) 2 ソケット搭載のサーバを用い，その PCI Express スロットに 2.1 節で述べた Bittware 社の Stratix V 搭載 FPGA ボード S5-PCIe-HQ を接続した．

まず準備として，HACApK の行列ベクトル積サブルーチン `HACApK_adot_body_lfmtx` を，C 言語化したものを用意した．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，それぞれファイルに保存するプログラムを作成した．

次に，上記の C 言語コードをベースにして，FPGA で実行する行列ベクトル積コードを OpenCL によって記述した．併せて，ホスト用コードとして，階層型行列と入力ベクトルをそれぞれ入力ファイルから読み込み，OpenCL カーネルに渡し，FPGA 上のカーネルを呼び出すプログラムを作成した．

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

4.2 実装 0: 単純な実装

基準となる単純な FPGA プログラムとして，ベースとなる C 言語のプログラムをそのまま機械的に OpenCL 化したものを作成した．**図 5** にコード本体を示す．

カーネル関数として関数名に__kernel 接頭辞をつけ，関数の引数には__global 接頭辞をつけて，FPGA ボード上の DDR3 メモリ上に配置するようにした．例外として，zbu については，このプログラム中で途中の計算結果を保存するために用いられているため，__local 接頭辞をつけ，FPGA 内に確保される高速なローカルメモリを用いた．

以下，実装 0 を元に，いくつかの最適化を試した中で，
(1) 最も回路規模が小さい場合
(2) 最も動作周波数が高い場合
(3) 最も処理時間が短い場合
について結果を示す．**表 3** に各実装におけるリソース使用量，**表 4** に，各実装において，各行列を入力として用いた場合の実行時間を示す．実行時間のうち，CPU は，Intel Xeon E5-2680v2 1 コアを用いた場合の実行時間を示す．

実装 0 では，コードは単純であるが，CPU 1 コアに比較して，126 倍もの実行時間を要している．

4.3 実装 1: 最もロジック利用率が小さい場合

図 5 において，7–13 行目と 21–27 行目は，ltmtx が 1 か 2 かの場合分けと，il ループの終端が kt か ndt かの違いのみであり，これらをまとめることで，全体としてループ構造を 1 つ減らすことができる．

```
1     for(ip=0; ip<nlf; ip++){
2       sttmp=st_lf+ip;
3       ndl=sttmp->ndl; ndt=sttmp->ndt;
4       nstrtl=sttmp->nstrtl; nstrtt=sttmp->nstrtt;
5       if(sttmp->ltmtx==1){
6         kt=sttmp->kt;
7         for(il=0; il<kt; il++){
8           zbu[il] = 0.0;
9           for(it=0; it<ndt; it++){
10            itt=it+nstrtt-1;
11            itl=it+il*ndt + sttmp->offset_a1;
12            zbu[il] += a1[itl]*zu[itt];
13        } }
14        for(il=0; il<kt; il++){
15          for(it=0; it<ndl; it++){
16            ill=it+nstrtl-1;
17            itl=it+il*ndl + sttmp->offset_a2;
18            zau[ill] += a2[itl]*zbu[il];
19        } }
20      } else if(sttmp->ltmtx==2){
21        for(il=0; il<ndl; il++){
22          ill=il+nstrtl-1;
23          for(it=0; it<ndt; it++){
24            itt=it+nstrtt-1;
25            itl=it+il*ndt + sttmp->offset_a1;
26            zau[ill] += a1[itl]*zu[itt];
27    } } } }
```

図 5 単純な実装コード

zau に関しては，参照頻度が高いため，ローカルに配列を用意した．1 行目の ip ループ直後で，ループ中で参照する可能性のある zau の要素を全てローカル配列にコピーすることで，グローバルメモリへのアクセス時間を削減した．

また，7 行目の il と 9 行目の it のループの順序を入れ替えることにより，il ループの内側では同じ zu[itt] の値を参照することができるため，メモリアクセスが減少する．

4.4 実装 2: 最も動作周波数が高い場合

実装 1 と同様に，7–13 行目と 21–27 行目をまとめてループ構造を 1 つ減らしただけの修正である．構造が単純であるために，動作周波数を高くすることができたと考えられる．

ローカル配列は使用していないが，実装 1 に比べて，回路規模は増加している．これは，グローバル配列をアクセスする際にキャッシュが生成されるため，それによって却って必要な論理回路が増加しているのではないかと考えられる．

zbu 以外の配列は全てグローバルメモリを参照しているため，動作周波数が最も高いにもかかわらず，メモリアクセス遅延が大きくなっており，実行時間は最も遅くなっている．

4.5 実装 3: 最も処理時間が短い場合

実装 1 と同様の最適化を行った上に，さらに 14–19 行目についても，15 行目の it ループと 14 行目の il ループを入れ替えている．これにより，il ループの内側では，同一の zu[itt] が参照できるため，メモリアクセスが減少している．

この実装については，リソース使用率がかなり低く，かつ動作周波数も高めであり，回路構成においてバランスがよかったものと考えられる．

4.6 結果のまとめ

ローカルメモリを適切に使うことで，元の実装に比べて，10 倍以上の性能改善が見られた．また，ループ順序を入れ替えることで，メモリアクセスが削減でき，高速化につながっている．その結果，最大で 16 倍の性能向上が得られた．一方で，CPU 1 コアと比較すると，現状では CPU の方が 8 倍程度高速である．

ループ内で同一変数への加算がクリティカルパスになっているケースが見受けられた．これを解消するため，ハードウェアとしてはやや冗長になるものの，シフトレジスタを用いてパイプライン的に足し込むような実装を行った．部分的に効果が認められる場合もあったが，全体として比較した場合には，他の最適化による効果の方が大きく，顕著な差が出ていない．

メモリの指定に関しては，読み出しのみの値が格納された配列に対しては，__global の代わりに，__constant という接頭辞をつけることによって，キャッシュの利用が効率化されるとの記載があったが，比較したところ，かなり性能が低下する原因となった．今回は比較的リソースに余裕があり，グローバルメモリのキャッシュに十分リソースが割り当てられたものと考えられる．リソースを限界近くまで使うような場合に効果が現れる可能性がある．

表 3 各実装におけるリソース使用量

	実装 0	実装 1	実装 2	実装 3
Logic utilization	29%	26%	28%	26%
DSP blocks	9	4	6	2
Memory bits	16%	18%	14%	15%
RAM block	608 (30%)	630 (31%)	536 (27%)	560 (28%)
fmax	246.18	244.73	269.25	268.95

表 4 各実装における実行時間 (ms)

行列	実装 0	実装 1	実装 2	実装 3	CPU
100ts	62597.0	5540.9	57661.3	4848.3	494.2
216h	8705.1	808.2	7904.0	684.0	68.7
human_1x1	8762.6	676.9	7962.5	547.3	69.6

今回行った最適化は，これまでに述べてきたことに留まっており，以下の最適化については今後の課題である．

- DDR3 メモリバンクの最適化
- ローカルメモリを使ったさらなるキャッシュブロッキング
- ループアンローリングによるパイプラインステージの調整
- パイプライン多重による高速化
- カーネル分割による高速化

5. おわりに

本稿では，FPGA による階層型行列ベクトル積について，OpenCL による実装を行った．いくつかの最適化を行った結果，最初のコードに比べて，最大で 16 倍高速になった．一方で，現時点ではまだ CPU 1 コアに比べて，1/8 程度の性能である．電力あたり性能を比較すれば FPGA の方が高い可能性はあるが，FPGA に向けた最適化はまだ不十分であると考えられる．

今後はさらに FPGA 向けの最適化を行い，どこまで CPU の性能に近づけることができるか，また電力あたり性能に関しても測定を行う予定である．

謝辞 日頃より議論をさせていただいている東京大学情報基盤センタースーパーコンピューティング研究部門の皆様に感謝します．本研究の一部は，JSPS 科研費 15K00166 の助成を受けたものです．本研究の一部は，科学技術振興機構戦略的創造研究推進事業 (JST/CREST), German Priority Programme 1648 Software for Exascale Computing (SPPEXA-II) の支援を受けています．本研究で用い

た Quartus II のライセンスの一部は，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] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and SatoshiMatsuoka, “Optimizing the Rodinia Benchmark for FPGAs (Unrefereed Workshop Manuscript),” 情報処理学会 研究報告 (2015-HPC-152), 2015.

[7] 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.

[8] 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.

[9] 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. Jitsumoto 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.

[10] 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/

[11] 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.

[12] 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.

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

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

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

[16] Altera Corporation, アルテラ SDK for OpenCL - 概要 https://www.altera.co.jp/products/design-software/embedded-software-developers/opencl/overview.html

[17] Altera Corporation, Altera SDK for OpenCL Programming Guide 16.0, UG-OCL002, 2016.

[18] Altera Corporation, Altera SDK for OpenCL Best Practice Guide 16.0, UG-OCL003, 2016.

[19] 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.

[20] Börm S., Grasedyck L. and Hackbusch W.: Hierarchical Matrices, Lecture Note, Max-Planck-Institut fur Mathematik, (2006).