Vol.2016-HPC-153 No.1
2016/3/1

FPGAを用いた疎行列数値計算の性能評価

大島　聡史[1,a)]　塙　敏博[1]　片桐　孝洋[1]　中島　研吾[1]

概要：近年，FPGA(Field Programmable Gate Array) に対して新たな高性能計算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を持つハードウェアであるが，プログラミング環境や利用の難しさなどの課題があり HPC 分野における活用はあまり行われていなかった．しかし今日では OpenCL のみを用いて利用可能な FPGA が登場し，様々な HPC アプリケーションを実装・評価できる環境が整ってきている．本稿では疎行列数値計算アプリケーションに対する FPGA の活用に向けて，単純な FEM プログラムの CG 法部分を OpenCL を用いて実装して FPGA 上で実行し，その性能や最適化方法についての評価を行う．

1. はじめに

高速で大規模な科学技術計算の需要に対して．様々な並列計算ハードウェアが利用されている．今日では従来から用いられてきた CPU(Central Processing Unit) に加えて．大量の計算コアを備えたメニーコアプロセッサや．本来は画像処理用のハードウェアである GPU(Graphics Processing Unit) の活用が進んでいる．しかしこれらのハードウェアをさらに高性能化させるうえでは半導体プロセスの微細化の限界に端を発する様々な問題があることが知られており．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消費電力に様々な処理を行うことが可能である．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 を用いて任意の処理を行う．すなわち FPGA のプログラミングを行うためには．Verilog などのハードウェア記述言語のためのプログラミング言語やツールを使う必要があり．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のは困難であるという問題があった．

回路設計技法に詳しくない利用者でも FPGA を扱えるプログラミング環境として．OpenCL[2] が利用され始めている．既に今日のいくつかの FPGA 製品においては．Verilog 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]．

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 [7], [8]．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 [9], [10]．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．現在はこれまでに扱ってきた数値計算カーネルを OpenCL を用いて FPGA 上に実装し．その最適化手法や性能．およびそれらが既存のハードウェア群とどのように違うのかの調査を始めている．本稿ではその実施内容について報告する．

本稿の構成は以下の通りである．2 章では FPGA とその性能最適化手法について述べる．3 章では OpenCL を用いて FPGA 上で動作するプログラムを作成し．いくつかの最適化手法を適用してその性能を評価する．4 章はまとめの章とする．

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

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) MB
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 15.1
	OpenCL SDK

2. FPGA と性能最適化

2.1 FPGA

本研究では，FPGA として Altera 社の Stratix V GS D5 を用いる．Stratix V は本稿執筆の時点で Altera 社のハイエンド最新世代の FPGA の 1 つである．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 [*1][11][12]．本研究ではこの FPGA が搭載された Bittware 社の PCI Express ボード S5-PCIe-HQ (s5phq_d5) (**図 1**) を用いている．

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 Fotran 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．

[*1] 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．

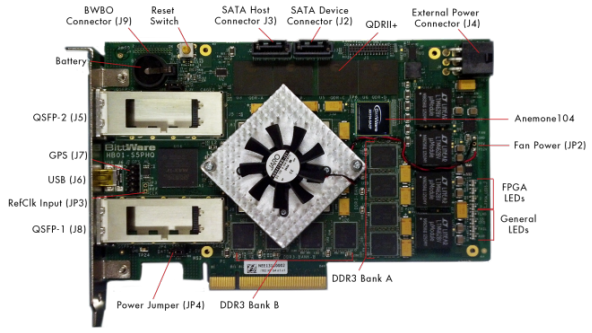

図 1　Bittware 社 S5-PCIe-HQ (Bittware 社提供，ただし本報告で用いるボードには QDR II+は実装されていない)

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．本研究で我々が用いている Altera 社の FPGA においても Stratix V シリーズから OpenCL への対応が始まっており，Verilog などを一切用いることなく，OpenCL のみで FPGA 向けのプログラムが作成可能となっている．

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，本報告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 [*2]．
- 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，カーネル開始直前に FPGA 構成情報（コンフィグレーションデータと言う）をダウンロードする必要があり，ホストから PCI Express 経由で高速に行える必要がある．
- 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

[*2] 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．

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 インタフェースが停止してしまうと，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．

しかしながら依然として，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
  OpenCL で記述されたオフロード機能は，コンパイラの内部で Verilog HDL や IP コアなどのマクロに変換され，論理合成やデバイスへのマッピングなどが行われる．そのためどんなに簡単な記述でも，現時点では 1 回のコンパイルで Intel Xeon E5 (Haswell プロセッサ) を用いても 2 時間以上必要である．
  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．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．性能については，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 OpenCL を用いた FPGA プログラミング

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り，現在は FPGA や DSP(Digital Signal Processor) など様々なハードウェアに利用範囲が広がっている．特に現在の 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．

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[13] と類似点が多い．現在の OpenCL はバージョン 2.0 が最新版である．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．

```
// 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

本稿では Altera 社から提供されている Altera OpenCL SDK[14] を用いて FPGA プログラミングを行う．本 SDK は Stratix V などの Altera 社製品を対象とした OpenCL コンパイラ群であり，Altera 社から 2013 年より提供されている．本 SDK を用いれば OpenCL プログラム (ソースコード) のみから FPGA 上で生成可能なプログラム (構成情報) が実行可能であり，GPU 等を用いる場合と同様に専用の API 関数からカーネル関数を起動するという形式で FPGA を利用する (FPGA に対象とする関数を実行させる) ことができる．

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 [15] や最適化ガイド [16] 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
- 細粒度並列化 (SIMD 化，ベクトル化)
- コード記述レベルの最適化
- ループアンローリング

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

2.2 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．

2.3.2 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．

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．

```
================================================================================================================================
|                                          *** Optimization Report ***                                                         |
================================================================================================================================
| Kernel: cg                                                                                                    | File:Ln   |
================================================================================================================================
| Loop for.body                                                                                                 | [1]:30    |
|   Pipelined execution inferred.                                                                               |           |
--------------------------------------------------------------------------------------------------------------------------------
| Loop for.body5                                                                                                | [1]:37    |
|   Pipelined execution inferred.                                                                               |           |
|   Successive iterations launched every 2 cycles due to:                                                       |           |
|                                                                                                               |           |
|       Pipeline structure                                                                                      |           |
--------------------------------------------------------------------------------------------------------------------------------
| Loop for.body18                                                                                               | [1]:39    |
|   Pipelined execution inferred.                                                                               |           |
|   Successive iterations launched every 8 cycles due to:                                                       |           |
|                                                                                                               |           |
|       Data dependency on variable                                                                             |           |
|       Largest Critical Path Contributor:                                                                      |           |
|           96%: Fadd Operation                                                                                 | [1]:40    |
--------------------------------------------------------------------------------------------------------------------------------
| Loop for.body37                                                                                               | [1]:45    |
|   Pipelined execution inferred.                                                                               |           |
|   Successive iterations launched every 8 cycles due to:                                                       |           |
|                                                                                                               |           |
|       Data dependency on variable BNorm2                                                                      | [1]:46    |
|       Largest Critical Path Contributor:                                                                      |           |
|           96%: Fadd Operation                                                                                 | [1]:46    |
```

図 3 AOC の出力ログ例

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．

3. 性能評価

3.1 対象問題と実行環境

OpenCL を用いて FPGA の性能評価を行った例はいくつか存在し，近年では丸山ら [5], [6] がアクセラレータ向けのベンチマークである Rodinia ベンチマークを用いた際の結果を報告した例などがあげられる．一方で我々はこれまでに研究発表を行ってきたアプリケーション群や ppOpen-HPC プロジェクトにおける各種アプリケーションなどを OpenCL を用いて FPGA 上に実装し評価することを当面の目標としている．しかしながら，これらの対象アプリケーションは OpenCL 化されていないうえに，FPGA に搭載可能なプログラムの規模が限られているため対象アプリケーションそのものを現在の FPGA 上へ実装することは現実的ではない．また FPGA 向けの OpenCL 最適化プログラミングについては 2 章にて述べたように様々な最適化手法があり，現在はどのようなプログラムに対してどのような最適化を行えば良いのかの指針や，具体的なプログラミング方法についての調査や評価が必要な段階である．

以上から，本章では単純なプログラムを対象として幾つかの最適化手法を適用し，その効果を確認する．具体的な対象プログラムとしては，我々の研究対象とするアプリケーションに CG(Conjugate Gradient) 法などの疎行列反復計算が多いことから，実験・演習用に C 言語を用いて作成された単純な一次元の FEM(Finite Element Method) プログラムにおける CG 法部分とする．また一部の最適化に関する評価においては，単純な疎行列ベクトル積プログラムも用いる．なお，計算中に用いる実数のデータ型は全て単精度浮動小数点 (float 型) を用いている．

図 4 に対象とする計算 (CG 法) の処理の概要を示す．ここで，角括弧は行列，波括弧はベクトルを意味する．実行時間の多くは前処理（3 行目）と疎行列ベクトル積（7 行目）に対応する部分に費やされており，含まれる処理の多くは OpenMP などを用いることで容易に並列化が可能な処理である．FPGA 上でループ部分全体を実行し，実行時間を測定する．CPU-FPGA 間の通信については測定範囲に含めていない．実験環境については，Intel Xeon E5 を搭載したサーバに，2 章にて述べた FPGA(Stratix V) を搭載して用いている．

```
1  {r0} = {b} - [A]{xini}
2  loop
3    solve {z} = [Minv]{r}
4    RHO = {r}{z}
5    if ITER=1 {p} = {z}
6    else BETA = RHO / RHO1
7    {q} = [A]{p}
8    ALPHA = RHO / {p}{q}
9    {x} = {x} + ALPHA * {p}
10   {r} = {r} - ALPHA * {q}
11 endloop
```

図 4 CG 法の処理の流れ

以降では，初めに対象アプリケーションを単純に OpenCL 化する方法とその性能について述べた上で，いくつかの最適化手法を適用し，そのプログラムの構成や実行速度，FPGA のハードウェア資源使用量を比較する．

3.2 単純な実装

性能比較のベースとする単純な FPGA プログラムとして，計算部分をそのまま OpenCL 化し必要なデータ転送を行ったものを作成した．より具体的には，CG 法部分を__kernel 接頭辞のついたカーネル関数として切り出し，CPU-FPGA 間で転送が必要な変数を__global 接頭辞のついた変数としてカーネル関数の引数に設定した．これにより，対応するホスト側の OpenCL API 関数 (clEnqueueReadBuffer, clEnqueueWriteBuffer) を用いることで，CPU-FPGA 間で__global 変数の送受信を行うことができる．カーネル関数実行時の clEnqueueNDRangeKernel API 関数呼び出し時におこなう並列度設定については全て 1 を指定しているため，FPGA カーネルは逐次実行される．

本プログラムのコンパイル (FPGA 上で実行されるバイナリを作成する) 時にコンパイラへ与えた主なオプションは`-g -W -v --board s5phq_d5` である．`-g` はデバッグ情報の生成，`-W` は warning の表示，`-v` はコンパイル状況の表示，`--board s5phq_d5` は対象とする FPGA の指定であり，特別な最適化などの指定は行っていない．なお，本コンパイラには CPU 向けのコンパイラでよくみられる`-O2` のような最適化オプションは存在しない．

ところで OpenCL プログラムにおいてカーネル関数を宣言する際には，他の多くの CPU 向けコンパイラ等と同様に，const キーワードにより変数や配列が書き換えられることのないものであることや，restrict キーワードによりポインタの重複がないことを明示することができる．これらを適切に使用することでコンパイラによる最適化がより効果的に行われることが期待できる．そこで，単純な実装に対して const キーワードと restrict キーワードを用いたものを作成した．以下，const キーワードと restrict キーワードを用いていないものを “単純な実装”，用いているものを “逐次実装” と呼ぶことにする．

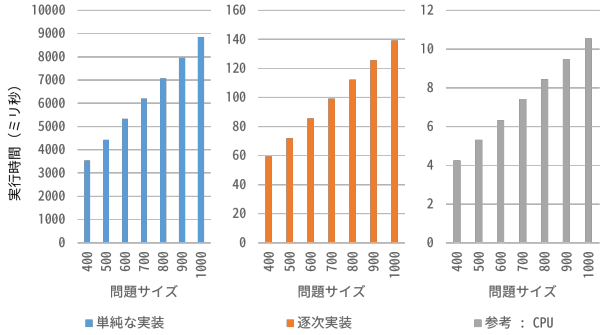

図 5　単純な実装/逐次実装による性能の比較

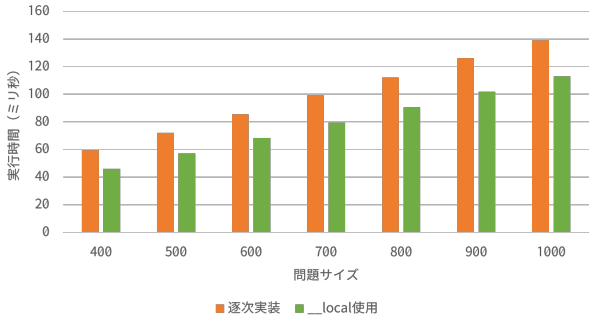

図 6　__local 配列の活用

表 2　コンパイル結果から確認できる回路構成の比較

	単純な実装	逐次実装	__local 化
動作周波数 (MHz)	247.46	269.32	262.12
Logic utilization	60%	68%	39%
Dedicated logic registers	31%	34%	18%
Memory blocks	61%	71%	34%
DSP blocks	2%	2%	2%

図 5 に問題サイズ (求める未知数の数-1 に等しい) と実行時間を示す．比較のため，反復回数は全て 1000 回に固定した．参考として E5-2680 v2 上で逐次実行した際の実行時間も測定した．CPU 用のコンパイラとしては gcc4.4.7，最適化オプションは-O2 を指定した．実行の結果，問題サイズの変化に対する実行時間の延び方の傾向は単純な実装，逐次実装，CPU ともに同様であったが，実行時間には大きな差が生じた．単純な実装のコンパイル時にのみ以下のような警告がでており性能低下の可能性が示唆されていたが，実際に大きな性能低下が観測された．OpenCL を用いて FPGA プログラムを作成する際には restrict キーワードを指定することは必須であると言える．

```
warning: declaring kernel argument with no
 'restrict' may lead to low kernel performance
```

ところで，単純な実装と逐次実装におけるコンパイラ出力結果からカーネル実行時の動作周波数や使用するハードウェア資源量が確認できる．**表 2** に比較結果を示す．なお，“__local 化” については次節で述べる．逐次実装の方が単純な実装よりも動作速度が 1 割程度高速ではあるものの，実行時間の差を説明できるような大きな値の違いは確認できない．静的な情報から実行性能を見積もることは容易ではないことが伺える．

3.3　適切なメモリ種別の指定

今回利用している FPGA にはチップ内に搭載されたメモリとチップ外に搭載された DDR メモリが存在し，前者の方が高速で低レイテンシである．しかし逐次実装では計算中に何度もアクセスする配列も__global 指示子の設定された配列であり，低速な DDR メモリ上に配置されてしまっていると考えられる．そこで，カーネル関数の冒頭で__global 指示子の設定された配列を__local 指示子を設定した配列にコピーし，その後はコピーされた配列のみを用いるという実装を行った．カーネル関数内において__global 配列と__local 配列の間でデータコピーをする分はオーバーヘッドとなるため，それを打ち消すだけの性能向上効果が得られるかが重要となる．

図 5 に問題サイズと実行時間を示す．問題サイズ 400 から 1000 までいずれも一定の性能向上が得られていることがわかる．この結果から，何度もアクセスする配列を__local 配列に移すことは問題サイズに限らず速度向上に寄与する重要であることが確認できた．OpenCL を用いて FPGA プログラムを作成する際には，搭載されているメモリ量などの制限が許すならば，積極的に__local 配列を活用するべきであると言える．

さらに表 2 を用いて本節の実装と前節の実装を比較すると，元々低かった DSP の値以外が大きく減少していることが確認できる．高速な__local 配列を使うことでパイプライン構成上の制約が減り，ロジックとメモリの要求量が低下したものと考えられる．

3.4　細粒度並列化 (SIMD 化，ベクトル化)

細粒度並列化の効果を確認するため，“単純な実装” をもとに構成される演算器の SIMD 長を伸ばしたり演算器の数自体を増やしたりして性能を確認した．これらの変更は，FPGA カーネル関数に対する attribute の指定，カーネル内部のループの初期値・終了値・ステップ値の変更，カーネル呼び出し時の並列度指定によって行った．実装を単純にするため，本節では__local 配列を用いた高速化は適用していない．

はじめに並列処理可能なループを SIMD 実行することを考える．SIMD 化を行うためには `num_simd_work_items` および `reqd_work_group_size` という attribute 値を指定したうえで適切な並列度指定によるカーネル実行をすればよい．ただし，CG 法にはリダクション演算など単純な SIMD 実行には向かない処理が含まれているため，必要に

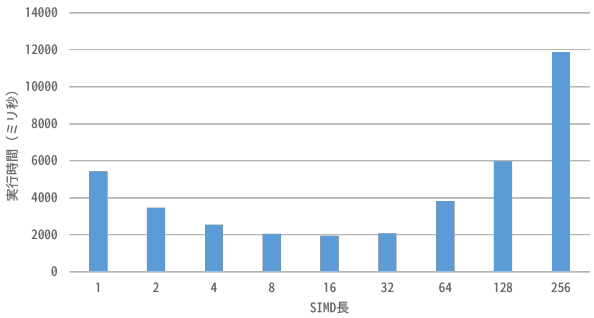

図 7　SIMD 長と実行時間 (コンパイル時に警告あり)

応じてバリア同期関数を挿入した．並列化対象となる各ループ処理については，SIMD 長による均等分割が行われるように初期値・終了値・ステップ値を修正した．

以上のようにして実装したプログラムをコンパイルしたところ，

```
Compiler Warning: Kernel Vectorization:
 branching is thread ID dependent ... cannot vectorize.
Compiler Warning: Kernel 'cg': limiting to 2 concurrent
 work-groups because threads might reach barrier out-of-order.
```

といった警告が表示されてしまった．警告を無視して問題サイズ 1000 にて実行した結果を図 **7** に示す．得られた実行結果 (計算結果の出力値) には問題がなかった．実行時間の傾向を一見すると，適切な SIMD 長を選ぶことで良い性能が得られるという結果が得られているように見えるが，前節までの結果と比べると非常に性能が低いことがわかる．警告上はベクトル化が行えない旨のメッセージであるが，今回適用した SIMD 化が適切に行えていない可能性は大である．

一方，`num_compute_units` という attribute 値を指定することで演算ユニット数を変更することができる．演算ユニット数を増やすことは SIMD 化と比べて FPGA のハードウェア資源を多く消費しやすいため，適切な使い分けが必要である．実際に今回のプログラムでは 2 並列までしか FPGA に収めることができなかった．なお，SIMD 化と組み合わせることも可能であるが，SIMD 化がうまくいかなかったため今回は組み合わせていない．カーネル内部の記述は SIMD 化の場合と同様で良いと考えられるが，コンパイルを行うとやはり

```
Compiler Warning: Kernel 'cg': limiting to 2 concurrent
 work-groups because threads might reach barrier out-of-order
```

という警告が出力され，正しい計算結果を得ることもできなかった．

なお，単純な CRS(Compressed Row Storage) 形式の疎行列に対する疎行列ベクトル積を実装し，行単位の並列化を施したところ，SIMD 化実装ではやはり

```
Compiler Warning: Kernel Vectorization:
 branching is thread ID dependent ... cannot vectorize.
```

が出力されることが確認できた．演算ユニット数を増やした場合には警告が出力されなかった．また各実装において単純なステンシル計算にて用いられるような疎行列を用いて性能を確認したところ，特に並列度が低いときに逐次実行と比べて長い実行時間がかかっており，各演算ユニットが ID を取得する処理自体にも無視できない程度のオーバーヘッドがある可能性が高い．

以上のように，今回実行した範囲では並列化をうまく行うことや，性能向上を得ることができなかった．今回用いている FPGA の仕様上どうしても不可能な処理であるのかという点も含めて引き続き調査中であり，今後の課題としたい．

表 3　回路構成と性能の比較

	逐次実装	最適化後
動作周波数 (MHz)	269.32	285.3
Logic utilization	68%	63%
Dedicated logic registers	34%	31%
Memory blocks	71%	68%
DSP blocks	2%	2%
実行時間 (msec)	139.190	106.951

3.5　コード記述レベルの最適化とアンローリング

2.4 節で述べたように，プログラム中に含まれる各ループ処理の修正やコードの共通化などによりプログラムの性能を向上させることができる．そこで，以下に示す最適化を実施した．

- ループの構成を変更
- 配列変数を一時変数に置換
- 間接配列アクセス部分をアンローリングし，パイプラインステージの長さを揃える

最適化後の実行時間を測定したところ，表 **3** に示すように実行時間が大幅に短縮された．生成された回路の構成からは性能の差となった部分は明確では無いが，動作周波数が向上している点については影響が大きいと考えられる．

4.　おわりに

本稿では FPGA を用いた疎行列数値計算の性能評価に向けて，OpenCL を用いて CG 法カーネルの実装を行い性能を評価した．幾つかの最適化手法を適用して性能を比較し，単純に元プログラムを OpenCL 化するよりも高い性能が得られるケースが確認できた．一方，OpenCL を用いた FPGA プログラミングについては，言語仕様や操作手順的には従来の GPU プログラミングと変わらないため特に困難なものではないが，現状では高速化のための指針を決める難しさ，並列実行の難しさ，実行可能なプログラム規模の小ささ，コンパイル時間の長さといった問題があり満足のいく結果が得られているとは言い難い．今後はさらに最適化を進めるとともに，他のアプリケーションの実装や，他ハードウェアとの性能と最適化手法の比較などを進めていく予定である．

謝辞 日頃より最適化プログラミングについて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，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」の助成を受けたものです．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] 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.

[8] 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/`

[9] 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.

[10] 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.

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

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

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

[14] 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`

[15] Altera Corporation, Altera SDK for OpenCL Programming Guide 15.1, UG-OCL002, 2015.

[16] Altera Corporation, Altera SDK for OpenCL Best Practice Guide 15.1, UG-OCL003, 2015.