情報処理学会論文誌 Vol.57 No.1 1–11 (Jan. 2016)

Java 言語ベース高位合成ツールによる高性能計算の実機による検証

矢内 奎太朗[1,a) 長名 保範[2,b) 中條 拓伯[3,c)

受付日 2015年3月4日, 採録日 2015年8月1日

概要：本論文は，Java 言語ベースの高位合成ツール JavaRock-Thrash の持つ，マルチスレッド記述からの並列回路生成機能に注目し，CFD への応用が可能な計算例であるステンシル計算を対象に JavaRock-Thrash で高位合成を行い，CPU と FPGA を用いたシステム上で動作させる場合の回路動作周波数や入出力のインタフェースなどの課題点を明確にすることを目標に，評価実験を行った．結果として，タイミング制約に課題があったものの，マルチスレッドを用いた場合に 1 スレッドでは 62MFLOPS に対して 32 スレッドでは 772MFLOPS となり，約 12 倍の高速化を達成した．

キーワード：FPGA，リコンフィギャラブシステム，高位合成，高性能計算，Java

Evaluation of High Performance Computing with Java based High Level Synthesis on a Real Machine

Keitaro Yanai[1,a) Yasunori Osana[2,b) Hironori Nakajo[3,c)

Received: March 4, 2015, Accepted: August 1, 2015

Abstract: This paper describes evaluation results of stencil calculation with a real machine in order to clear out the problems concerned with operating frequency or input/output interfaces using a Java-based high level systhesis tool JavaRock-Thrash which generates parallel circuits from the description of multi-threading. According to the experimental results, although there are some timing constraints, the generated circuit has achieved 12 times faster as 772 MFLOPS in using 32 threads against a single thread as 62 MFLOPS using multi-threading in high level syntehsizing.

Keywords: FPGA, reconfigurable system, High Level Synthesis, High Performance Computing, Java

1. はじめに

High Performance Computing(HPC) の分野に，流体の運動に関する方程式を元に コンピュータによる計算から解を得ることで流体の動きを解析する数値流体力学 (CFD:Computational Fluid Dynamics) が自動車や航空機設計などにおいて応用されている．CFD では，一般的に大規模な計算を要し，従来スーパコンピュータを利用してきた．最近では，マイクロプロセッサの性能も向上し，またGPU などによる高速化も利用できるようになった．しかしながら，マイクロプロセッサでは高周波数のクロックで駆動することから，消費電力の増大が深刻な課題である．HPC では，GPU によるアクセラレーションも行われているが，GPU によるアクセラレーションも同様に消費電力が高いことが問題点として指摘されている．

この問題の解決策の一つとして，FPGA によるハードウェア・アクセラレーションが注目されている．FPGA は，開発当初 ASIC のプロトタイプとして利用されてきたが，近年，集積度が高まり，動作周波数が向上するとともに，低

1 東京農工大学 工学府
Graduate School of Engineering, Tokyo University of Agriculture and Technology
2 琉球大学 工学部
Department of Engineering, University of The Ryukyus
3 東京農工大学 工学研究院
Institute of Engineering, Tokyo University of Agriculture and Technology
a) yanai@nj.cs.tuat.ac.jp
b) osana@eee.u-ryukyu.ac.jp
c) nakajo@cc.tuat.ac.jp

© 2016 Information Processing Society of Japan

消費電力化が進められてきている．そのため，FPGA を用いて特定用途向けに専用回路を設計することで効率的にアクセラレーションを実現できるようになった．実際に，検索エンジンである Bing のアクセラレータとして FPGA が利用されたり [1]，Amazon EC2 F1 Instance[2] のようにクラウド上での FPGA が提供されているなど，FPGA への期待が 高まってきている．

FPGA を用いてハードウェア・アクセラレーションを行う場合，開発言語としてハードウェア記述言語 (HDL) を用いて記述する場合が多い．しかしながら，HDL は専用のハードウェアを構築するための並列性の記述が可能であるものの，クロックなどの複雑なタイミング制御を考慮する必要があるなど，ハードウェアに関する知識が求められ，これまでソフトウェアを中心に開発を行ってきた HPC プログラマが，大規模なアプリケーションに対し，ハードウェアによる高速化を目指して実装を行う場合に，その開発環境に容易に対応できないといった問題を抱えている．

この課題を解決するために，C 言語や Java 言語などの高級言語を用いて FPGA を設計する手法である高位合成 (HLS:High Level Synthesis) が注目されており，その性能も向上し，フリーで利用できるものも増えてきており，ツールの例として Impulse C[3]，MaxCompiler[4]，Vivado HLS [5] といったものがある．

こういった状況で我々は，Java 言語ベースの高位合成ツール JavaRock-Thrash[6] を開発し，その有効性について検証を行ってきた．JavaRock-Thrash では，追加の構文やデータ型などの拡張なしに Java プログラムから回路を生成できるという特徴を持つ．Java 言語を選んだ理由として，並列に動作させたい処理をマルチスレッドで記述し，そこから並列に動作する回路を生成することができ，ループ展開によるパイプライン化を用いることで性能の良い回路の生成が可能な点が挙げられる．

これまで，Java 言語ベースとした HLS で，マルチスレッド機能による並列回路生成機能については，報告されてはいるものの，これを実際に HPC 向けのプログラムに適用して評価した例はみられない．そこで我々は，JavaRock-Thrash の持つマルチスレッドからの並列回路生成機能に注目し，CFD への応用が可能な計算例であるステンシル計算を対象に高位合成を行い，評価を行った [7].

その評価は，論理合成した結果からサイクル数，動作周波数を取得した結果によるものであり，実機で動作させる場合，実機による実用性を追求するためには，ターゲットシステムの入出力インタフェースを含めた上で配置配線を行った後の回路動作周波数を求め，その性能を明らかにする必要がある．

そこで，RTL シミュレーションでは不明であった，実機動作させる際の課題点を明確にすることを目標に，琉球大学で開発を行っているシステム上で実機で評価実験を行った．本論文では，その実装について述べ，評価した結果について報告する．

Java 言語と HPC の関係については，Heterogeneous System Architecture (HSA) Foundation[8] が Java 言語の有用性に着目しており，AMD も HPC における将来展望について，その重要性を認識している [9]．したがって，Java 言語ベースの HLS によるハードウェア・アクセラレーションの可能性について実機による評価により示すことは，今後の HPC の動向に大しても何らかの影響を及ぼすものと考えられる．

2 章では関連研究を示し，3 章では JavaRock-Thrash の構成を含めシステムの概要について述べる．4 章で評価について示し，5 章でまとめる．

2. 関連研究

高位合成を利用したハードウェア・アクセラレーションの例としてマドリード大学の D. Sanchez-Roman らによる数値流体力学の FPGA 実装 [10] が挙げられる．これは，流体力学のシミュレーションプログラムに対して高位合成を利用して FPGA によるハードウェア・アクセラレーションを行った研究である．この論文では，C 言語ベースの高位合成ツールである Impuluse C を用いてハードウェア化を行っている．回路化した際に，浮動小数点演算に代表される高レイテンシの演算器を自作の低レイテンシの演算器に置換することで高スループットの回路を実現した．また，入力した C 言語で記述されたプログラムに対しても演算順序などの変更などの回路を意識した最適化を行っている．結果として，ソフトウェアとの比較で最大 23 倍の高速化を達成している．

ハードウェア・アクセラレーションの対象である流体シミュレーションプログラムは 4,570 行であり，Impulse C に入力するために書き換えた際の行数が 8,990 行，出力された VHDL のコードが 102,175 行である．この論文では，VHDL で同等な動作をする回路を記述した場合の行数は 33,000 行と見積もっており，高位合成に用いる言語を学習する手間を考慮したとしても，高位合成の利用により開発期間が短縮できるといえる．

3. システムの概要

3.1 高位合成ツール JavaRock-Thrash

JavaRock-Thrash とは Java 言語ベースの高位合成ツールであり，Java のソースから Verilog HDL のソースを生成する．JavaRock-Thrash では次の 3 つの観点から Java 言語に注目して開発さえた．

(1) Java 言語は明示的にポインタを取り扱わないため，ポインタの合成方法を考慮しなくてよい．
(2) 並列処理の記述に Java スレッドが利用できる．
(3) Java のクラスやオブジェクトを HDL のモジュールや

© 2016 Information Processing Society of Japan

サブモジュールに対応させているため，ハードウェアの設計と親和性が高い．

JavaRock-Thrash は，処理速度向上のための機能や回路規模削減のための機能を実装している．高位合成ツールとしての利便性を考え Java 言語の拡張は行っていないが制限はいくつかあり，再帰といった動的な動作のハードウェア化はサポートされていない．また，アノテーションによりハードウェア化専用の機能を実装している．

JavaRock-Thrash は，Java のソースファイルと回路を生成する際の設定を記述する config ファイルを入力ファイルとし，JVM 上で実行可能な Java のクラスファイルと，RTL シミュレーションや論理合成が可能な Verilog HDL ファイルを出力する．詳細については文献 [6] を参照されたい．

JavaRock-Thrash では，各メソッドごとに処理の開始を命令する入力ポート (メソッド名 req) とメソッドが処理中かどうかを示す出力ポート (メソッド名 busy) が作成される [6]．メソッドが処理中は，その処理が終了するまでは新たな入力は受け付けられず，そのため，まとまったデータサンプルを BlockRAM にバッファリングしてから処理を開始する．

JavaRock-Thrash の性能面での特徴を以下に列挙する．

- 単/倍精度浮動小数点の対応
- ループのパイプライン化による時間的並列性の記述（細粒度並列性）
- Java スレッドによる空間的並列性の記述（粗粒度並列性）

高位合成ツールの中には浮動小数点を実装していないものがある．しかしながら，JavaRock-Thrash では単/倍精度の浮動小数点の演算を扱うことが可能であり，浮動小数点演算は CFD に代表されるような科学技術計算には欠かせないものである．さらに並列処理を記述するためにループ展開によるパイプライン化と Java スレッドを実装しているため，複数の演算器による演算密度の向上が期待できる．

JavaRock-Thrash でスレッドを用いた場合の記述例を図 1 に示す．まず，TopClass が SubClass のオブジェクト SubA, SubB を宣言することで，TopClass モジュール内部に 2 つの SubClass モジュールがインスタンス化される．次に TopClass モジュールが Thread クラスを継承した SubClass の start メソッドを Thread として呼び出すことで，それぞれの SubClass の run メソッドが同時に実行される．

スレッドの終了を待つ際にはスレッドの join メソッドを呼び出す．なお，synchronized による排他制御や wait, notify などの機能はサポートしていない．

図 1 のソースコードを JavaRock-Thrash で回路化した際の回路をモジュール図で表すと図 2 のようになる．Block RAM は Dual Port であり，各スレッドの内部へ接続されるポートと親モジュールと通信のためのポートの 2 つが備わっている．JavaRock-Thrash では，スレッド間で直接のデータ通信は行えず，Dual Port RAM を介してデータの授受を行う．たとえば SubA と SubB の間で通信を行う場合は SubA のデータを親モジュールである Top Module に渡し，その後 Top Module から SubB のモジュールへデータを送信する．

これらの機能を使用することで JavaRock-Thrash をハードウェア・アクセラレーションの用途として有効に使うことができる．多数の演算器の並列動作による細粒度並列処理とともに，マルチスレッド機能から高位合成した並列回路動作による粗粒度並列処理を併用することで JavaRock-Thrash は処理性能向上をはかることができる．

JavaRock-Thrash では，ループ展開の展開数をユーザが指定する．しかしながら，展開数を増やせば性能向上は期待できるものの，配置配線後のタイミング制約に支障をきたし，動作周波数に影響を与えることがわかっている．そのためループの展開数については，処理性能と合成した回路の動作性能とのトレードオフを考慮して，繰り返し試行を行い，最適なループ展開数を決定する必要がある．

3.2　マルチスレッドによるステンシル計算の高速化

ここでは，本研究で高速化の対象としたステンシル計算

```
1  public class TopClass {
2    final SubClass SubA = new SubClass();
3    final SubClass SubB = new SubClass();
4    void Thread(){
5       //スレッドの処理開始
6       SubA.start();
7       SubB.start();
8       try{
9         //スレッドの終了待ち
10        SubA.join();
11        SubB.join();
12      }
13      catch(Exeption e){}
14    }
15 }
16 public SubClass extends Thread{
17  public void run(){・・・・
18
19  }
20 }
```

図 1　Java スレッド記述例

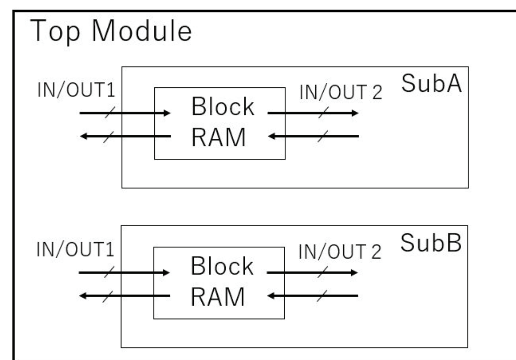


図 2　スレッドのモジュール図

について触れ，ステンシル計算でのマルチスレッドの適用について説明する．

3.2.1　ステンシル計算

HPC の中で数値流体力学や，熱力学や電磁気学など偏微分方程式が支配方程式である計算問題においてステンシル計算が広く利用されている．ステンシル計算とは偏微分方程式の近似解を求める手法の一つである．ステンシル計算では，ある時刻のデータセットを固定された計算パターンにより更新し，次の時刻のデータセットを得る．図 3 に 2 次元正方配列上でのステンシル計算のサンプルコード例を示す．次の時刻におけるすべてのデータの値は，現在の時刻における各データ要素の 4 近傍のデータの要素の値を用いて計算される．

変数 GRID は配列のサイズを示し，正方行列であるため 1 変数となっているが，列と行のサイズが異れば，それぞれ GRID1, GRID2 と異なる変数を割り当てることとなる．

ステンシル計算の例として，拡散方程式がある．拡散方程式とは，熱などの物質の拡散を表現した偏微分方程式である．

1 次元の拡散方程式を離散化や差分法を行った後の方程式は時刻 0 におけるセル A の温度 x_{a0} とそれに隣接するセル B の温度 x_b0 から，時刻 1 におけるセル A の温度 x_{a1} は以下の式で表すことができる．

$$x_{a1} = x_{a0} + k(x_{b0} - x_{a0}) \tag{1}$$

ここで，k は拡散係数である．さらに，中心セル x_c に対する 4 近傍セルの数を 4(上下左右で x_u, x_d, x_l, x_r) とし，k を改めて拡散係数に時間刻みの大きさ Δt を乗じたものとすると，次の時間刻みの中心セル $x_{c\Delta t}$ を解く 2 次元の拡散方程式は以下のように示すことができる．

$$x_{c\Delta t} = x_c + k((x_u - x_c) + (x_d - x_c) + (x_l - x_c) + (x_r - x_c)) \tag{2}$$

この式は 8 つの加算と 1 つの乗算からなり，合計 9 個の演算器が必要となるが，以下の式に変換することができる．

$$x_{c\Delta t} = x_c + k((x_u + x_d + x_l + x_r) - 4x_c)) \tag{3}$$

この変形により 4 つの加算器と 1 つの減算器，2 つの乗算器からなり，合計 7 つの演算器が必要になるので，演算器の削減が可能になる．

本研究では，温度を 100，周囲の温度を 0 とする初期状態を与えた拡散方程式に対して，格子サイズ 100 × 100 としてハードウェア・アクセラレーションを試みた．

```
1  public class ThreadX{
2   for(i=1;i<GRID-1;i++){
3    for(j=1;j<GRID-1;j++){
4     T1[i][j]=T0[i][j]+0.25*(T0[i][j+1]+T0[i][j-1]+
5      T0[i+1][j]+T0[i-1][j]-4.0*T0[i][j]);
6    }
7   }
8   for(i=1;i<GRID-1;i++){
9     for(j=1;j<GRID-1;j++){
10      T0[i][j]=T1[i][j];
11   }
12  }
13 }
```

図 3　ステンシル計算のサンプルコード

3.3　マルチスレッドの適用

前項で述べたステンシル計算を，マルチスレッドで明示的に並列化したコードを Java で記述した．

マルチスレッドによるステンシル計算の概要を図 5 に，擬似コードを図 6 に示す．擬似コード中の変数 `n` は時刻を示し，`i` と `j` は図 4 における任意の格子点の座標を示す．また，座標 `(i,j)` における現在の格子点の値を `T0[i][j]` と示し，次の時刻の格子点の値を `T1[i][j]` と表す．次の時刻の格子点の値 `T1[i][j]` は，周囲の 4 近傍である `T0[i-1][j]`，`T0[i+1][j]`，`T0[i][j-1]`，`T0[i][j+1]` の値を参照し，計算を行うことにより得られる．そして，`T1` の値を `T0` に置き換えた後に次の時刻に遷移する．

まず，ステンシル計算のデータセットを，分割されたあとの配列のサイズが均等になるように複数のブロックに分割する．ここで，分割数は任意に決定でき，分割の方法も分割後の配列のサイズが均等であれば任意の方法でよい．なお，JavaRock-Thrash では配列は BlockRAM 上に配置され，2 次元配列は 1 次元配列として展開される．

次に，メインスレッドから各スレッドに対して分割したデータセットの値のコピーを行う．ここで，ステンシル計算の性質として，ある値を計算する際には 4 近傍の値を用いて計算することから分割したデータセットの袖領域 (図 5 の灰色の部分) のコピーを行う必要がある．そして，スレッドとして分割した領域を並列に動作させた後に，メインスレッド上で各スレッドが保持している袖領域の部分の処理を行い，次のループの実行に移る．任意のループ回数 (変数 `iteration`) が終了したら各スレッドからメインスレッドへ再び計算結果のコピーを行い，計算結果の集約を行う．各スレッドでは，分割したデータセットに対してス

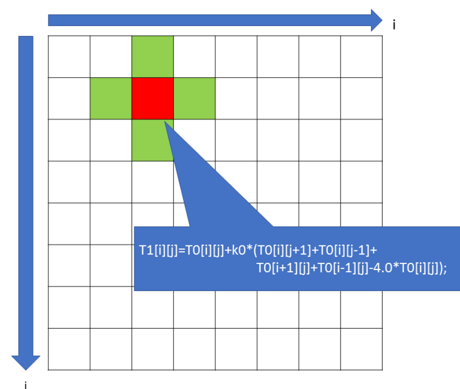


図 4　ステンシル計算の概要図

© 2016 Information Processing Society of Japan

テンシル計算を処理する．ここで各スレッドで行われる計算は JavaRock-Thrash の機能であるループのパイプライン化を用いる．このことよりスレッドによる粗粒度並列化と各スレッド内でのパイプライン実行による細粒度並列化を行うことができるので，並列性の向上が期待できる．

なお，図 5 には，4 スレッドによるデータ分割の例を示したが，8 スレッドに対しては 100 × 100 の行列を 50 × 25 の 8 つの行列に分割した．16 スレッドでは 25 × 25 に，32 スレッドでは 25 × 12 の行列と，25 × 13 の行列をそれぞれ 16 スレッドに割り当ててデータ分割を行った．

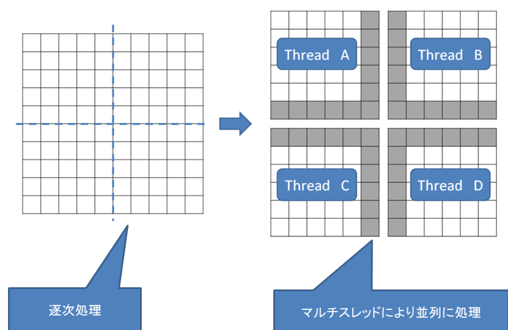


図 5　Java スレッドによるステンシル計算

```
1  public class TopClass{
2    for(n=0;n<iteration;n++){
3      ThreadA.v0=v0;
4      ThreadB.v0=v0;
5      ThreadC.v0=v0;
6      ThreadD.v0=v0;
7     //スレッドとして並列動作
8      ThreadA.run( );
9      ThreadB.run( );
10     ThreadC.run( );
11     ThreadD.run( );
12    //袖領域の同期
13     v0=ThreadA.v1;
14     v0=ThreadB.v1;
15     v0=ThreadC.v1;
16     v0=ThreadD.v1;
17   }
18    //計算結果の集約
19     v0=ThreadA.v1;
20     v0=ThreadB.v1
21     v0=ThreadC.v1;
22     v0=ThreadD.v1
23 }
24 public class ThreadX{
25   for(i=1;i<GRID-1;i++){
26     for(j=1;j<GRID-1;j++){
27       v1[i][j]=v0[i][j]+k0*(u[i+1][j]+u[i-1][j]+
28                  u[i][j+1]+u[i][j-1]-4.0*u[i][j]);
29     }
30   }
31   for(i=1;i<GRID-1;i++){
32       for(j=1;j<GRID-1;j++){
33            v0[i][j]=v1[i][j];
34       }
35   }
36 }
```

図 6　Java スレッドを用いたステンシル計算の擬似コード (4 スレッド)

3.4　システムの構成

JavaRock-Thrash により開発した演算回路の実機による評価のために，琉球大で開発されている図 7 に示すアーキテクチャ構成をとるクラスタシステムのプロトタイプ [15] を用いた．このシステムは，複数の CPU に，アクセラレータとして接続された FPGA 群から構成される並列処理システムである．

複数の FPGA はネットワークにより接続され，FPGA 間にまたがる大規模計算パイプラインを構築してアクセラレータを構成する．CPU 群と FPGA 間もネットワークで接続され，すべての CPU でアクセラレータを共有できる．

このアーキテクチャに基づいた小規模な試作システム上で，既に CPU-FPGA 間，FPGA-FPGA 間の通信と部分再構成による FPGA の回路読み込みのための基本的な仕組みが動作している [16]．図 8 に本研究で用いた試作システムを示す，琉球大で動作しているシステムでは，2 つの FPGA で動作しているが，本研究で用いたシステムはまだ試作段階であるため，CPU と直接接続されている FPGA 単体のみを利用している．用いた FPGA は Xilinx の NetFPGA 1G-CML で，Kintex7 325T-2 FPGA を搭載している．

FPGA の内部の回路のモジュール図を図 9 に示す．ホスト CPU 上では Linux(Ubuntu 16.04) が動作しており，FPGA では，Xillybus コア [17] がホスト側のインタフェースとして組み込まれている．Xilybus コアを用いると，FPGA 上の FIFO が Linux のデバイスファイルとして，read() や write() といった関数によってアクセスが可能である．Xilybus コ

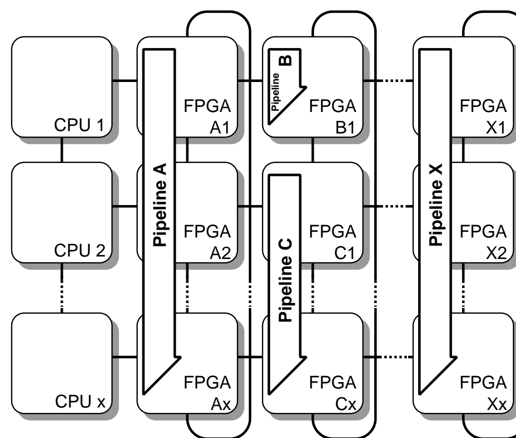


図 7　CPU-FPGA クラスタシステムの構成

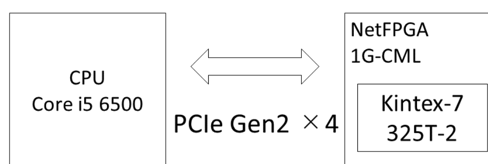


図 8　CPU と FPGA の構成

© 2016 Information Processing Society of Japan

アは Read/Write とともに最大 1,600MB/s の Rev.B core を使用しており，このコアが PCI Express endpoint block を内包している．配布されている Xillybus コアはカスタマイズが可能であり，現状ではホストから FPGA へ転送を行う 64bit の FIFO インタフェースが 2 チャンネル，FPGA からホストへ転送を行う 64bit の FIFO インタフェースが接続されている．FIFO はそれぞれ計算用の入出力として用いられている．FIFO は対応するデバイスファイル名が割り当てられており，これらはユーザープログラムから read() や write() などの入出力関数によって読み書きが可能である．これらはいずれも共通のクロックで動作しており，PCI Express Endpoint と同期する必要があるため，FPGA 内の最大動作周波数は 250MHz となっている．

ホストから転送されたデータは二つの Source FIFO から計算用の PE に送られ，その出力は Sink FIFO を経由してホストに返される．本研究では PE の部分を JavaRock-Thrash により設計した．Source FIFO の読み出しと Sink FIFO の書き込みは PE とは別のコントローラによって制御され，このコントローラによって Source FIFO を二つ用いる場合の読み出しの同期処理や，バックプレッシャの制御を行っている．バックプレッシャの制御について，FIFO には容量制限があるためこの容量を超えないように操作をする必要がある．そのため，SinkFIFO が溢れそうな場合には，溢れる前に Source FIFO の読み出しと PE からの出力を停止する必要がある．

現状バージョンの JavaRock-Thrash では，処理結果を一度 BlockRAM に書き出すため，メモリを経由せずに演算結果を次のモジュールに受け渡すといったストリーム計算に対応していない，

そのため，本システムでは Source FIFO の後と Sink FIFO の手前に BlockRAM を設けている．また，FIFO と BlockRAM への通信部分は Verilog HDL を用いて記述しているが，この部分は JavaRock-Thrash で生成した他の回路と置換することが可能である．これにより，将来的にはユーザーが実行したい回路を Java 言語で記述するだけで，本システムを利用可能にすることができると考えている．

ステンシル計算に必要な情報として，FPGA は CPU から以下に示すデータを受け取る．

- 計算終了までの時間刻み数 (n)
- グリッド内の各セルの値の 2 次元配列

計算のプロセスとしては，まずホスト CPU のプログラムで計算に必要な配列の初期化を行い，FPGA の入力としてこれらのデータをホスト CPU のプログラムから write() で書き込む．ステンシル計算は FPGA の回路で処理を行い，その処理結果を FPGA からの出力として FIFO にあるデータを read() で読み込む．そのため，計算の途中経過をホスト CPU に出力する機能は現在実装していない．これは，FPGA-CPU 間の通信がボトルネックとなっていることが既に判明しているため，演算性能の低下を回避するためである．

4. 評価

4.1 予備評価

予備評価としてマルチスレッドで記述したステンシル計算のプログラムを，JavaRock-Thrash で回路化し，RTL シミュレーションで実行ステップ数を求め，Vivado HLS と比較して評価を行った．
予備評価においては，Xilinx 社の Kintex7 シリーズの XC7K325T を想定し，NC-Verilog を RTL シミュレータとして用い，処理サイクル数を算出し，動作周波数として 250MHz を想定して，得られた実行ステップ数から実行時間として算出した．
評価に用いたステンシル計算は拡散方程式であり，配列サイズは 100×100，ループ回数は 500 とし，スレッド数は 1,2,4,8,16,32 スレッドとした．また，Vivado HLS では，JavaRock-Thrash に入力したプログラムと同等な動作をするプログラムを C 言語で書き直して入力した．この際に，ループや配列にはディレクティブを用いて最適化を行った．なお，比較のために用いた Vivado HLS のバージョンは 2016.3 である．

JavaRock-Thrash と Vivado HLS の評価結果をそれぞれ表 1 と表 2 に示し，実行ステップ数と回路規模を比較したものを，それぞれ図 10 と図 11 に示す．この表によると RTL シミュレーションの場合，マルチスレッド化により処理サイクル数は 95%の削減を達成し，実行ステップ数では Vivado HLS より 27%の削減を達成することができた．
しかしながら，32 スレッドでは 19.2 倍の高速化にとどまっている．この理由としては，ステンシル計算の領域を分割した際に，ある領域のスレッドに対して他のスレッドの領

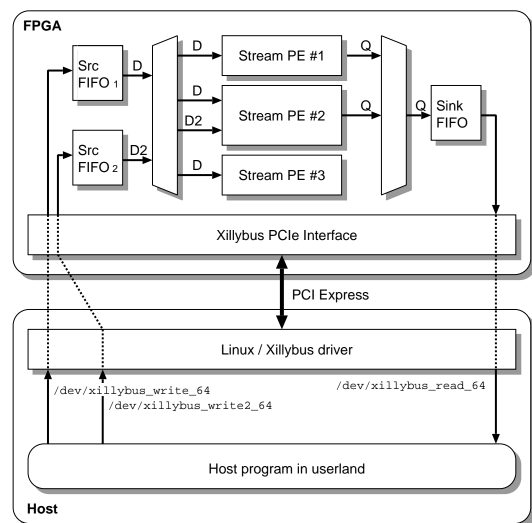


図 9　FPGA の内部のブロック図とホスト側の関係図

© 2016 Information Processing Society of Japan

表 1 予備評価の各スレッドでの比較

	LUT	(%)	FF	(%)	実行ステップ数数	高速化率
1 スレッド	4670	2.29	4333	1.06	129404013	-
2 スレッド	6813	3.34	7551	1.85	66749033	1.94
4 スレッド	12950	6.35	14587	3.58	35936249	3.60
8 スレッド	27423	13.46	29146	7.15	18014084	7.18
16 スレッド	51771	25.40	57510	14.11	10020479	12.9
32 スレッド	88622	43.48	112079	27.5	6730434	19.2
Kintex7 XC7K325T	203800		407600			

表 2 Vivado HLS での評価結果

	LUT	(%)	FF	(%)	処理サイクル数
Vivado HLS	4574	2.24	3344	0.8	24135606
Kintex7 XC7K325T	203800		407600		

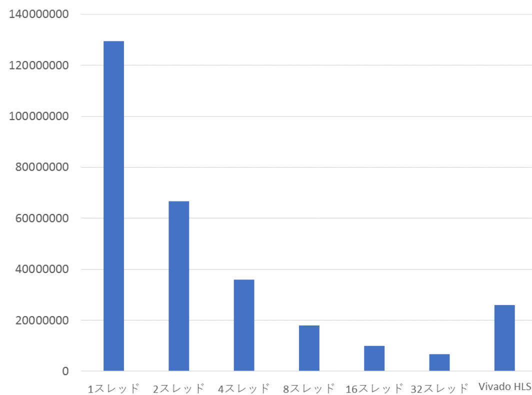


図 10 各スレッドと Vivado HLS の実行ステップ数の比較

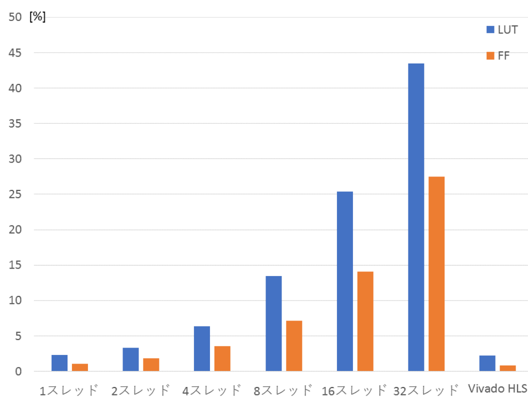


図 11 各スレッドと Vivado HLS で生成した回路規模の FPGA 回路リソースに占める割合

域を参照する必要があり，この処理部分はマルチスレッドで記述した場合も並列化できておらず，逐次処理を行っているためである，もう一つの理由として，BlockRAM がデュアルポートで動作可能ではあるものの，Java 上の Main クラスと各オブジェクトとの配列のアクセスに 1 ポート，各オブジェクト内での配列のアクセスに 1 ポート使用しており，演算に用いる BlockRAM は実質シングルポートとなっている．これより，1 回の演算を行うごとに 5 回 BlockRAM への読み出し処理を行っているためである．この点の改善は今後必要である．

回路規模の比較に関して，JavaRock-Thrash では 1 スレッドあたりの LUT の使用率が，LUT リソース全体の 2.93%を占め，Vivado HLS の LUT の使用率はリソース全体の 2.24%とほぼ同等となった．スレッドを増やせば当然リソース使用率は増加するが，並列化により性能は向上するため，この点は利用する FPGA の規模を考慮したトレードオフとなる．

4.2 評価環境

評価で用いた FPGA は Xilinx 社の Kintex7 シリーズの XC7K325T である．

JavaRock-Thrash で生成したステンシル計算の回路を CPU-FPGA 混在クラスタ上で実装を行い評価した．これまでは，RTL シミュレーションでの検証のみであった．そのため，実機で動作させた際に長大なクリティカルパスが生成されることにより回路が期待した挙動をしない場合や，回路規模が大きすぎるため FPGA に搭載できないといった問題がある可能性がある．また，HPC プログラムなどのハードウェア・アクセラレーションの場合，計算結果の確認を行うためのホスト CPU との協調動作が必要である．そのため，高位合成系で出力した回路以外に入出力のための回路を組み合わせて FPGA に搭載する必要がある．

以上の理由から，実機上で動作可能かどうか検証を行った．今回の評価では，JavaRock-Thrash でのパイプラインの

© 2016 Information Processing Society of Japan

段数を 5 に固定し，スレッドの数を 1,2,4,8,16,32 で変化させた．

評価に用いた元となる Java ソースコードは 92 行からなり，ハードウェア化のためには特別なコードは挿入する必要はなく，並列化を施した後のコードは 898 行となった．並列化により行数が増加したのは，3.3 節で述べたマルチスレッドの適用を行ったためであり，32 個のクラスに分割した際に各クラスにそれぞれ個別の処理を記述しているためである．しかしながら，並列化は一定の手順で記述するものであり，今後は，並列化ツールにより自動化できるものと考えている．

4.3　評価結果

評価結果を表 3 に示し，各スレッドごとの演算性能を比較したものを図 12 に示す．ここで横軸は繰返し回数を示し，縦軸は計算性能を MFLOPS 値で示している．また，各スレッドごとの回路規模を比較したものを図 13 に示す．指定したクロック周波数に対して回路が正しく動作していない可能性がある場合，WNS(Worst Negative Slack) の値が負になってしまう．本評価結果ではすべてのスレッドに対して WNS の値が負になっているが，回路は正しく動作している．

なお、FPGA 上で、100 × 100 のグリッドサイズの 500 時間刻み分の計算には 6,730,434 クロックを所要し，グリッドの 1 セルを更新するのに必要な計算は 7FLOPS である。これから計算すると、250MHz で動作させた場合の理論性能は

$$(250 \times 10^6/6730434) \times (500 \times 100 \times 100 \times 7) = 1.30\text{GFLOPS}$$

となる。文献 [16] における RTL 実装に比べて性能が大きく劣っているのは，文献 [16] ではグリッド全体の計算が終わる前に次の計算をパイプライン的に開始しているためである．現状の JavaRock-Thrash にはその機能を備えておらず，そのために十分な性能を発揮できていない．

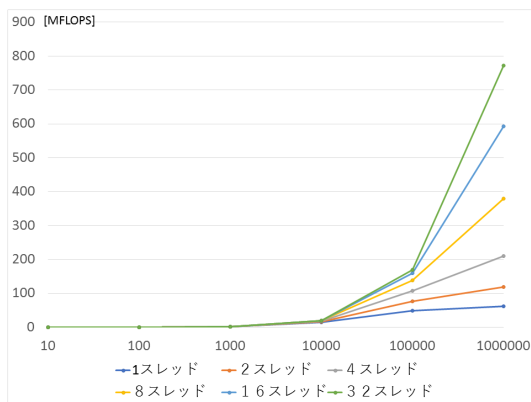


図 12　実機動作における各スレッドの計算性能

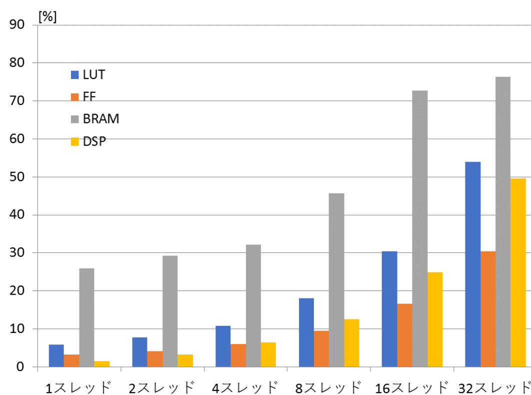


図 13　実機動作における各スレッドごとの回路規模

4.3.1　回路面積の評価

BlockRAM の使用率が高く，またスレッドの増加とともに使用率は増加しているため，64 スレッドを実装する際には，BlockRAM の使用率が FPGA の容量超過となり，64 スレッドを実装することはできなかった．これは，各スレッドで並列に計算を行う必要があるので BlcokRAM を各スレッドに対して用意する必要があるからである．

また，JavaRock-Thrash の内部ですべてのスレッドにおいて WNS(Worst Negative Slack) の値が負になっており，タイミング制約違反を起こしている．Xillybus コアの動作周波数を 250MHz から変更することができないため，クロック周波数を低下させることはできず，Vivado の合成・配置配線における最適化を強力にしても解決することができなかった．

タイミング制約違反を起こしている箇所は JavaRock-Thrash の回路の BlockRAM へのアクセスの部分である．FPGA の内部では BlockRAM は縦に長いため，多くの Block-RAM を 1 つのブロックとして扱った場合，回路が縦に長くなり，動作遅延を引き起こしてしまうことが原因である．現状では，計算結果は正常に出力されているものの，JavaRock-Thrash で出力した回路の BlockRAM の扱いと動作周波数には，課題があるといえる．

4.3.2　計算性能の評価

スレッドの増加とともに計算性能は向上しており，1 スレッドと比較すると 12 倍計算性能が向上している．しかし，現状報告されている同一のシステムを用いて HDL でステンシル計算を実装した場合 [18] は最大 41.93GFLOPS と報告されており，マルチスレッドで明示的に並列化しても HDL の実装と比較すると，まだ高位合成を用いたハードウェア・アクセラレーションには課題があるといえる．現状，JavaRock-Thrash での実装に課題がある箇所としては，ループ展開により計算密度の向上が可能である一方，ループ展開を行うと回路規模が増大しているという点である．このことから，ユーザーが JavaRock-Thrash を用いる場合，ループの展開数やスレッドの数を回路規模や求められる性能に応じて決定する必要がある．

また，CPU と FPGA の転送にかかる時間を算出した．用

ⓒ 2016 Information Processing Society of Japan

表 3 実機での評価結果 (パイプライン段数：5)

	LUT	(%)	FF	(%)	BRAM	(%)	DSP	(%)	MFLOP(n=1M)	WNS
1 スレッド	11924	5.9	13258	3.25	115.5	26.0	13	1.5	62.5	-0.29
2 スレッド	15687	7.7	17057	4.18	130.5	29.3	27	3.2	62.5	-0.30
4 スレッド	22176	10.9	24670	6.05	143.5	32.2	54	6.4	209.6	-0.55
8 スレッド	36722	18.0	38829	9.53	203.5	45.7	105	13	379.1	-0.82
16 スレッド	62011	30.4	67879	16.7	323.5	72.7	209	25	593.0	-1.09
32 スレッド	109948	53.9	124306	30.5	339.5	76.3	417	50	772.64	-1.47
Kintex7 XC7K325T	203800		407600		445		840			

表 4 処理時間の内訳

	処理時間	全体に占める割合 [%]
計算時間 [s]	52.64	60.5
転送時間 [s]	34.37	39.5
全体の処理時間	87.01	100

表 5 OpenMP による Core i7 上でのステンシル計算結果 (GFLOPS)

Size	64	128	256	512	1024
i7-7700K	6.54	8.55	9.79	9.38	6.54
i7-8700K	5.78	8.18	10.01	10.21	10.30

いた条件は 32 スレッドによるマルチスレッド記述により実装し，ループ回数 1,000,000 回における値を用いた．本評価においては，ループ展開数を 5 とした．その理由としては，実機で動作させた際に，6 を超えるループ展開数では，タイミング制約違反により正しい結果が得られなかったためである．

実行処理時間の内訳は，表 4 のようになり，CPU と FPGA の転送時間が全体の 4 割を占めるという結果となった．現状判明していることとして，CPU と FPGA 間の通信インタフェースとして，利用している Xillybus のスループットは，1,600MB/s であるが，転送データ長が短い場合のスループットに課題があるとされている．

この問題は，Riffa[19] といった高性能 PCI Express インタフェースを利用することで解決可能であると考え，その Riffa を用いた場合には 1GB/s を超える転送速度が達成できることが既に確認されている．

4.3.3 ソフトウェア実行との比較

FPGA によるハードウェア・アクセラレーション効果を示すために GPU との比較を試みたが，現状の FPGA システムにおいてはメモリバンド幅がネックとなり，意義のある比較は行えなかった．そこで，Core i7 搭載の PC 上において OpenMP を用いて並列化を行い，複数コアで実行した結果について示す．

メインのデータ構造を以下に示す．ここで `x[]` には初期値が，`wall[]` には境界の情報が入り，`xx[]` にはテンポラリデータを格納する．

```
double x[SIZE][SIZE],
       xx[SIZE][SIZE];
bool wall[SIZE][SIZE];
```

図 14 に実行したプログラムの繰返し実行を行う演算カーネル部分を示す．これを outer loop で並列化を行った．

このカーネル部を含むプログラムを gcc 7 を用いて，-O3 -march=native でコンパイルし実行した結果を表 5 に示す．

十分な性能が出ていないのは `wall[]` が可変であるため，コンパイル時に最適化が効かないことが原因であると考えられる．

4.3.4 Java 言語による設計生産性に関する評価

JavaRock-Thrash では，開発言語である Java 言語に対し，ハードウェア化のための型や文法を拡張しない方針を採り，ソフトウェア開発者にとっても可読性や保守性は高い [6]．また，開発および動作確認に，Eclipse といった一般的な Java の開発環境が利用できる．今回の評価においては，JavaRock-Thrash により生成された HDL コードが膨大なものとなり，RTL シミュレーションにはスレッドを増やすと Vivado のシミュレータでは数時間を要することもあった．しかしながら，JVM 上でソフトウェアとしてアルゴリズムを検証する時間は数秒となった．したがって，計算アルゴリズムの動作確認には長時間を要する RTL シミュレーションを使うことなく，Java で記述したソースを JVM において動作確認ができ，開発期間を短縮することができる．並列化に対するデータ分割については，現在は手

```
1  #pragma omp parallel for
2      for (int i=1; i<SIZE-1; i++){
3        for (int j=1; j<SIZE-1; j++){
4          double l = x[i][j-1]; bool wl = wall[i][j-1];
5          double u = x[i-1][j]; bool wu = wall[i-1][j];
6          double r = x[i][j+1]; bool wr = wall[i][j+1];
7          double d = x[i+1][j]; bool wd = wall[i+1][j];
8          double c = x[i][j];
9
10         if (wl) l = c;
11         if (wu) u = c;
12         if (wr) r = c;
13         if (wd) d = c;
14
15         xx[i][j] = c + (l+u+r+d - 4*c)*k;
16       }
17     }
18
19     for (int i=0; i<SIZE; i++)
20       for (int j=0; j<SIZE; j++)
21         x[i][j] = xx[i][j];
```

図 14 OpenMP によるステンシル計算のカーネル部分

© 2016 Information Processing Society of Japan

作業により分割しているが，その処理自体は複雑なものではなく，自動化は可能である．

5. まとめ

我々はこれまで，Java 言語ベースの高位合成ツール JavaRock-Thrash により，高性能計算を例に，RTL シミュレーションを元に評価を行ってきた．これに対し，本論文では PCI Express を搭載した実機を用いて高性能計算の実装を試み，高位合成の後に配置配線を行った上で，処理性能と動作周波数や回路面積について評価を行った．

文献 [7] では，論理合成のみで行い 484 スレッドまで並列記述を行い，その性能を比較した結果，Vivado HLS に対して 95 倍の性能を示すことを示した．本研究では論理合成だけで性能を予測・決定するのではなく，ターゲットボードを対象に配置配線を行うことで，タイミング制約などを考慮して評価した．さらに PCI Express を通じた PC とのデータ転送などが性能に大きく影響を与えることがわかり，実際にシステムを構築して定量的な数値を示し，FPGA を用いた実用的な計算機システムを実現するための問題点を洗い出した．

生成した回路を実機上で動作させることにより，RTL シミュレーションでは不明であった点を明らかにした．結果として，実機動作でもマルチスレッドの適用により，32 スレッドで実装した場合，単一スレッドと比較して 12 倍の高速化を達成することができた．また，現状の JavaRock-Thrash を実機で動作させる際の課題として，BlockRAM からの読み出しが処理のボトルネックとなっていることや，FPGA 内部でのクロックの遅延により動作周波数を低下させる必要があるということを明らかにした．

今回実装を試みた実機は特殊なインタフェースを備えたものではなく，現在広く利用されている PCI Express を利用したものであり，通信部分インタフェースの書き換えにより，容易に他のシステムへの置き換えは可能であると考えている．しかしながら，高速化のためには BlockRAM を経由しない通信機構が必要であり，Java 言語ベースの HLS を新たに開発を行うためには考慮すべき点である．

今後は，さらにチューニングを行い，実機での実用性を明らかにしていきたい．

参考文献

[1] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Yi Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope: “A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services”, *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*, pp.13–24. IEEE Press (2014.6).

[2] Inc Amazon Web Services. Amazon EC2 F1 インスタンス. https://aws.amazon.com/jp/ec2/instance-types/f1/ (Accessed: 2018-01-26).

[3] Inc Impulse Accelerated Technologies. Impulse C. http://www.impulsec.com/. (Accessed: 2018-01-26).

[4] Maxeler technologies. MaxCompiler. https://www.maxeler.com/products/software/maxcompiler/. (Accessed: 2018-01-26).

[5] Xilinx Inc. Vivado HLS. https://japan.xilinx.com/products/design-tools/vivado/integration/esl-design.html. (Accessed: 2018-01-26).

[6] 小池 恵介，三好 健文，五十嵐 雄太，船田 悟史，中條 拓伯：“Java 言語ベース高位合成ツールによるアクセラレータ開発環境”，電子情報通信学会論文誌 D，Vol.J98-D，No.3，pp.373-383 (2015.3).

[7] Y. Ishikawa, K. Yanai, K. Koike, T. Miyoshi and H. Nakajo: “Hardware Acceleration with Multi-Threading of Java-Based High Level Synthesis Tool”, ACM Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART 2017) ACM Digital Library (2017).

[8] HSA Foundation. http://www.hsafoundation.com/ (Accessed: 2018-11-19).

[9] “AMD は HPC への将来展望を新たにする”, http://www.hpcwire.jp/archives/3355 (Accessed: 2018-11-19).

[10] D. Sanchez-Roman, G. Sutter, S. Lopez-Buedo, I. Gonzalez, F. J. Gomez-Arribas, J. Aracil, and F. Palacios. “High-level Languages and Floating-point Arithmetic for FPGA-based CFD Simulations”, *IEEE Design Test of Computers*, Vol.28, No.4, pp.28–37 (2011.7).

[11] 三好健文: “Fpga 向けの高位合成言語と処理系の研究動向”, コンピュータ ソフトウェア, Vol.30, No.1, pp.76–84 (2013).

[12] Joshua S. Auerbach, David F. Bacon, Perry Cheng, and Rodric M. Rabbah. “Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures”, *OOPSLA* (2010).

[13] 三好健文: “Synthesijer を使った java による FPGA 開発のはじめ方”, 組込みシステムシンポジウム 2014 論文集, Vol.2014, pp.3 (2014.10).

[14] Takefumi Miyoshi: “Synthesijer”, http://synthesijer.github.io/web/. (Accessed: 2018-01-26).

[15] 坂本洋平, 松田紘作, 大久保慎也, 長名保範: “部分再構成による CPU-FPGA 混在クラスタの実現へむけた研究”, 信学技報 リコンフィギャラブルシステム Vol.116, No.210, pp.51–56 (2016.9).

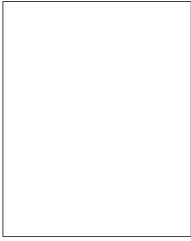
[16] 長名保範, 坂本洋平, 松田紘作, 大久保慎也. “CPU-FPGA 混在クラスタにおけるリモート部分再構成の初期性能評価”, 信学技報 リコンフィギャラブルシステム Vol.116, No.416, pp.25–29 (2017.1).

[17] Xillybus Ltd. Xillybus. http://xillybus.com/. (Accessed: 2018-01-26).

[18] 坂本洋平, 長名保範: “ストリーム計算による拡散方程式の実装と性能評価”, 信学技報 リコンフィギャラブルシステム Vol.117, No.221, pp.13–18 (2017.9).

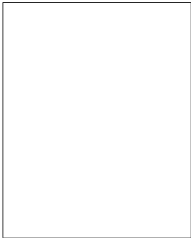
[19] Matt Jacobsen and Ryan Kastner: “RIFFA 2.0: A reusable integration framework for FPGA accelerators”, International Conference on Field Programmable Logic and Applications (FPL) (2013).

© 2016 Information Processing Society of Japan



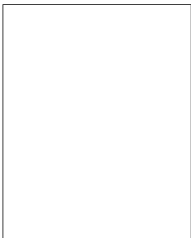
矢内 奎太朗

2016 東京農工大学工学部情報工学科卒業．2017 同大学大学院工学府博士前期課程情報工学専攻修了．現在，スクウェア・エニックス勤務



長名保範

2001 慶應大・理工・情報工卒．2006 同大大学院理工学研究科開放環境科学専攻博士課程修了. 2006 同大学理工・生命情報学科特別研究助手の後，2009 より成蹊大理工・情報科学科助教を経て 2011 より琉球大学工学部助教．リコンフィギャラブルコンピューティング・組み込みシステムに関する研究に従事．電子情報通信学会, IEEE-CS 各会員．博士 (工学).



中條 拓伯　（正会員）

1985 神戸大・工・電気工卒．1987 同大大学院工学研究科電子工学専攻修了．1989 同大学工学部助手の後, 1998 より 1 年間 Illinois 大学 Urbana-Champaign 校 Center for Supercomputing Research and Development (CSRD) にて Visiting Research Assistant Professor を経て，1999 より東京農工大学大学院准教授．プロセッサアーキテクチャ，組込みシステム，リコンフィギャラブルコンピューティングに関する研究に従事．電子情報通信学会，IEEE-CS，ACM 各会員．博士 (工学).

ⓒ 2016 Information Processing Society of Japan