高位合成におけるループパイプライン化の検討

五十嵐 雄太[1]　老子 裕輝[1]　中條 拓伯[1]

概要：流体のシミュレーションのように実行時間が増大してしまうプログラムに対して，FPGA を用いたハードウェアアクセラレーションが行われている．また，FPGA を設計する際に，HDL ではなく高級言語を用いる高位合成に注目が集まっている．高速化するには並列性を抽出する必要があり，高位合成においてループ部分をパイプライン化することで並列度を高めることができる．本論文では高位合成ツール JavaRock-Thrash を用いた，ループのパイプライン化を実装した．流体のソルバに対して実装を行うことで，ハードウェア化に際して必要な機能を検証する．

キーワード：高位合成，FPGA，アクセラレーション，数値流体力学

A Study on Loop Pipelining in High Level Synthesis

Yuta Ikarashi[1]　Yuki Oigo[1]　Hironori Nakajo[1]

Abstract: A hardware acceleration using an FPGA is used for a program which increases execution time such as simulation of fluid. In designing hardware in an FPGA, though high level synthesis is currently attracting hardware designers instead of HDL. In order to speed up, exploiting parallelism by pipelining a loop part is expected to speed up a target algorithm in using high level synthesis. A strategy of pipelining a loop using JavaRock-Thrash is introduced in this paper. We have verified necessary functions in implementing hardware by applying it to a fluid solver.

Keywords: High Level Synthesis, FPGA, Acceleration, Computational Fluid Dynamics

1. はじめに

学術界および産業界において計算機の果たす役割は年々増している．特に，科学技術分野における活用方法の一つして High Performance Computing (HPC) があげられる．HPC のターゲットに，流体の運動に関する方程式を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) がある．CFD においては，大規模計算機を利用してでも，膨大な時間を要し，実行結果を得るのに数日を要することもある．

近年, HPC において書き換え可能な Field Programmable Gate Array(FPGA) を用いた，ハードウェア・アクセラレーションが注目され，さまざまな研究機関において計算処理の高速化が試みられている．

FPGA は，開発当初は ASIC のプロトタイプとして用いられてきたが，近年集積度や動作周波数が向上したことに伴い，低消費電力化が進められ，ネットワーク機器，組込み機器などにおいても利用されるようになり，さらには演算器を多数並べるなどにより，ハードウェア・アクセラレーションにおいても活用されるようになってきた．この FPGA を用いて特定の用途に向けた専用回路を設計することで，より効果的にアクセラレーションを実現することが可能となった．

これまでは FPGA の内部回路を設計する際にはハードウェア記述言語 (HDL) を用いる方法が一般的であった．しかし，ハードウェア・アクセラレーションに FPGA を使用する場合，ソフトウェア技術者にとって HDL の習得が FPGA を扱う上での一つの障壁となっている．そこで近年注目を集めているのが高位合成による設計方法である．これは C 言語や Java 言語などの高級言語を用いて FPGA 設計を行う方法であり，高位合成ツールとして Impluse-C[1], MaxCompiler[2]，JavaRock[3] などがある．

この流れの中で，当研究室において JavaRock と同じゴールである，型の追加や構文の拡張を導入せず Java プログラムに手を加えずに HDL に変換することを目指し，浮動

[1] 東京農工大学
Tokyo University of Agriculture and Technology

小数点演算機能やループ展開といった最適化を強化したJavaRock-Thrashの開発を行い，評価を進めてきた [4], [5].

以上の背景からハードウェア・アクセラレーションを目的として，高位合成ツール JavaRock-Thrash (JRT) を用いて流体力学ソルバ FaSTAR[6] を FPGA 上で設計・実装を試みた．しかしながら，現状の JRT を用いて実装したところ，十分なアクセラレーション効果を得ることはできなかった．その原因を解析したところ，プログラムの実行時間の多くを占めるループ部分において，並列性を抽出できておらず，全体の処理サイクル数が増加していることがわかった．

この評価結果を受けて，本論文ではループ部分で，十分な並列性を抽出する方法を考案し，ループ部分のパイプライン化を行った．本論文では，2 章で関連研究について述べる．次に 3 章で JRT を用いた実装について述べる．次に 4 章で提案手法であるループのパイプライン化について述べ，評価を行う．最後にまとめと今後の課題について述べる．

2. 関連研究

1 章で述べたように FPGA の利用において，ハードウェア・アクセラレーションのために，HDL だけではなく高位合成も使用されるようになった．ここでは，その研究例についていくつか述べる．

2.1 FPGA よるハードウェア・アクセラレーション

文献 [7] では CFD パッケージ UPACS[8] を対象として高速化を行っている．対象となるプログラムの処理内容の分析を行い，実行時間の割合が大きい箇所から順次 FPGA に実装を行うことで効率よくハードウェアによる高速化を目指している．ハードウェアにより高速化する際にはデータの流れをパイプライン化することが必要であり，その際に考慮しなければならないことはデータの依存関係であり，それによりパイプラインの段数が決まる．Xilinx 社の Virtex-5 上に実装した結果，Intel Core 2Duo（2.66GHz）によるソフトウェア実行との比較において約 209 倍の実行速度向上に成功している．しかしながら，これは FPGA の設計に HDL を用いており，提案方法では高位合成を用いている点で異なる．

文献 [7] では HDL を用いて FPGA 設計を行っているが，高位合成を用いて FPGA 設計を行いアクセラレーション達成した例として，文献 [9] と文献 [10] がある．それぞれ高位合成ツールとして Impulse C と Max Compiler を使用し，ともにアクセラレーションとしての効果が得られており，文献 [9] ではソフトウェアと比較して約 23 倍の高速化に成功している．高速化の要因として，HDL を用いて独自の IP コアを作成したことがあげられる．そのほかにも高位合成ツールには独自ループ展開やループパイプライン化を行う機能が備わっているものもあり，それを活用したことも高速化の一因である．さらに高位合成ツールを用いたことで開発期間の短縮につながったことも述べられている．我々の提案する方式は，既存の高位合成ツールの機能のみではなく機能の拡張をすることでアクセラレーションの効果を向上させている．

2.2 ループパイプライン化

ハードウェア化する際にはパイプライン化は必須のものであり，ループをパイプライン化することで処理の高速化を実現することができる．ループをパイプライン化する研究はこれまでにも行われている [11]．これまでは完全入れ子ループに対してのパイプライン化のみ実装が行われていたのに対し，文献 [12] では不完全入れ子ループに対してもパイプライン化を実装する方法を提案している．完全入れ子とはループの中に単一のループのみの構造であるが，不完全入れ子とはループの内部に複数のループが存在するものを指す．文献 [12] では，パイプライン化を行わない場合は Data Flow Graph(DFG) に対して一つのステートマシンでデータパスを管理している．一方でパイプライン化の際は，パイプラインステージごとにステートマシンを用意することでデータフローがパイプライン動作している．これによりおよそ 10 倍の実行速度の向上が得られた．しかしながら，本論文と文献 [12] では対象にしている高位合成ツールが異なるため，別のアプローチをとる必要がある．

3. 高位合成ツールによる CFD コードのハードウェア実装

高位合成を用いたハードウェア・アクセラレーションとして，JRT を用いて CFD パッケージである FaSTAR の一部を実装し，性能の検証を行った．

3.1 CFD パッケージ FaSTAR

FaSTAR とは宇宙航空研究開発機構によって開発された CFD パッケージである．非構造格子を扱うことのできる圧縮性流体解析ソルバであり，航空機や宇宙機などの空力解析に代表されるような数値流体力学を得意としている．今回はその一部である勾配計算を行うモジュールを対象に実装および評価を行う．このモジュールは First Loop，bdface Loop，grad_states Loop の 3 つのループで構成されており，各ループの実行時間を計測したところ First Loop が最も大きい実行時間の割合を占めていた．したがってこの First Loop の部分を対象にハードウェア化を行った．

このモジュールの特徴としては不完全入れ子ループが存在していることである．ループに対する高速化の処理として高位合成ツールの機能であるループ展開を行うことができる．しかし，一般的なループ展開は入れ子のループがあった場合最も内側のループに対してのみ実行することが

できるものである．従って FaSTAR のこのモジュールをはじめとする，その他の CFD コードにも存在する入れ子の外側のループ，あるいは全体に対してはループ展開を実行することができない．完全入れ子の場合はループつぶしを行うことで入れ子の構造を解消することができるが，不完全入れ子ループの場合は複雑で，2 章で述べたように独自に対策をとる必要がある．

3.2 高位合成ツール JavaRock-Thrash

JRT とは Java から Verilog HDL への変換を行う高位合成ツールである．JRT では次の 3 つの観点から Java 言語に注目して開発している．

(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．

ただし，new や malloc のように動的な振る舞いをする動作は記述できず，基本的には逐次処理に倣った仕様になっている．特徴として入力の Java に変更を加えることなく扱えることや，ループ展開による時間的並列性，Java スレッドによる空間的並列性を持つ Verilog HDL のソースコードが生成できることがあげられる．

3.3 ループ展開を用いた実装

まずはじめに FaSATR に対して JRT を用いてハードウェア化し Xilinx Artix7(XC7A100T) を対象に論理合成を行った．その結果を**表 1** に示す．Intel Core i5（3.10GHz）によるソフトウェア実行と比較して，FPGA で実行した場合は実行時間が低下している．通常の場合とループ展開時を比較すると，ともにループ展開時の結果は動作周波数の低下がみられる．一方でループ展開をしなければ処理サイクル数が多くなり，処理時間が増大してしまう．

高速化が実現できない原因は並列性の抽出をループ展開のみで行っている点である．ループ展開によってループ部分がパイプライン的に動作しているように見えるが，実際はループの境界が存在している．ループ展開を行うと一連の処理の中での処理サイクル数が増えてしまい，各処理サイクルにおける状態を管理するためにステートマシンの状態数が増加する．それにともなって回路構造が複雑になり，動作周波数の低下を招く．これはループ展開の展開数が増えるとより顕著になる．

またハードウェア化した箇所は不完全入れ子ループであり，繰り返し回数が多いのは外側のループという構造になっている．最も内側のループは繰り替えし回数が少なく，ループ展開による高速化の恩恵が少なくなっている．したがって文献 [12] のように，単一のループを最適化するのではなくて，入れ子構造のループに対しても最適化を実行して処理時間を短縮させる方法を考える必要がある．

表 1 CFD コードのソフトウェアと JRT の比較
Table 1 Comparison of the software and the JRT of CFD code

	Fmax[MHz]	処理サイクル数	処理時間 [s]
ソフトウェア	—	—	166×10^{-3}
JRT(通常)	179	1345×10^{12}	7.51×10^{6}
JRT(展開)	171	909×10^{12}	5.31×10^{6}

4. ループパイプライン化

これまでの実装評価で，ループ展開を行っただけではアクセラレーションの効果が得られないことがわかった．そこで動作周波数を低下させずに処理サイクル数を削減する手段としてループをパイプライン化する方法をとる．パイプライン化する手法として整数線形計画法やモジュロスケジューリングがある [13]．しかし，整数線形計画法は解を得るのに時間がかかることがあるのでモジュロスケジューリングを用いてパイプライン化を実装した．

図 1 にパイプライン化の例を示す．縦軸の数字がステップ数であり，この例は x[i] = a[i] + b[i] + c[i] という式を繰り返していることを示している．通常では 6 ステップ目が終了してから次の繰り返し処理が始まるのに対し，パイプライン化することにより 3 ステップ目から次の繰り返し処理を実行することができる．このパイプライン化を行うことでスループットが向上し，処理サイクル数を削減できる．パイプライン化はループの回数に依存しないため，ループ展開のような動作周波数の低下を回避できる．

4.1 モジュロスケジューリング

モジュロスケジューリングとはループをパイプライン化するスケジューリング手法の一つである．次に示す手順でスケジューリングが行われる．

(1) 非パイプラインの DFG を用意する．
(2) 開始間隔を決め，空の資源予約表を作成する．
(3) 資源予約表を DFG の各ノードを埋める．
(4) 相対スケジュールを決定する．

非パイプラインにスケジューリングされた DFG と使用演算器数を入力とし，出力として開始間隔と相対スケジュールを得る．開始間隔と相対スケジュールを示したものが図 1 である．開始間隔とは数クロックサイクルごとに次の繰り返し処理を開始するかを決定するものである．相対スケジュールとはループ内での各ノードの実行開始ステップ数である．

実際にモジュロスケジューリングを行った結果の資源予約表とスケジューリング結果の例を**図 2** に示す．資源予約表の列はノードの種類，行の大きさは開始間隔，表中の数字は各ノードの相対スケジュールである．左側の DFG

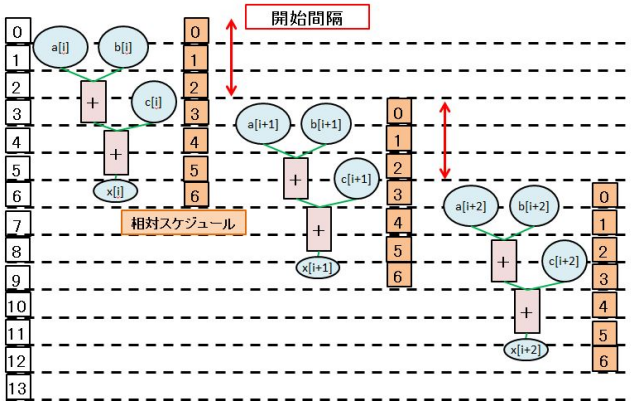
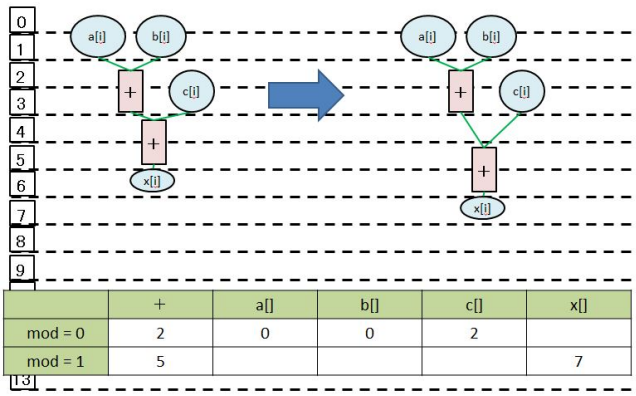


図 1　開始間隔と相対スケジュール
Fig. 1　Explanation of iteration interval and iterative schedule



	+	a[]	b[]	c[]	x[]
mod = 0	2	0	0	2	
mod = 1	5				7

図 2　モジュロスケジューリングの例
Fig. 2　Example of modulo scheduling

にモジュロスケジューリングを適応すると右側の DFG になる．まず左側の DFG において，先行ノードを持たない a[i]，b[i]，c[i] のうち元々の開始ステップが小さい a[i]，b[i] からスケジューリングを行う．始めは資源予約表は空なので対象ノードの開始ステップの剰余を計算し，mod=0 の行に相対スケジュールを記録する．同様にほかのノードも表に埋めていくが，加算器は 2 つあり，2 回目に使用されるものは mod=0 の箇所が埋まっているので次の行の mod=1 に割り当てられる．このようにしてスケジューリングを行っていき，すべてのノードの相対スケジュールが決定するまで続ける．もし資源予約表の行が足りなくなったら開始間隔を大きくして再度スケジューリングを行う．

4.2　JavaRock-Thrash への適応

前節で述べたモジュロスケジューリングのアルゴリズムを JRT へ適用する．JRT は Java ファイルとコンフィグファイルを入力として，Verilog HDL ファイルと DFG の XML ファイルを出力する．パイプライン化では JRT が出力する 2 つのファイルを元にループ部分の再スケジューリングを行いパイプライン化する．その後そのスケジューリング結果を JRT の出力した Verilog HDL のループ部分に

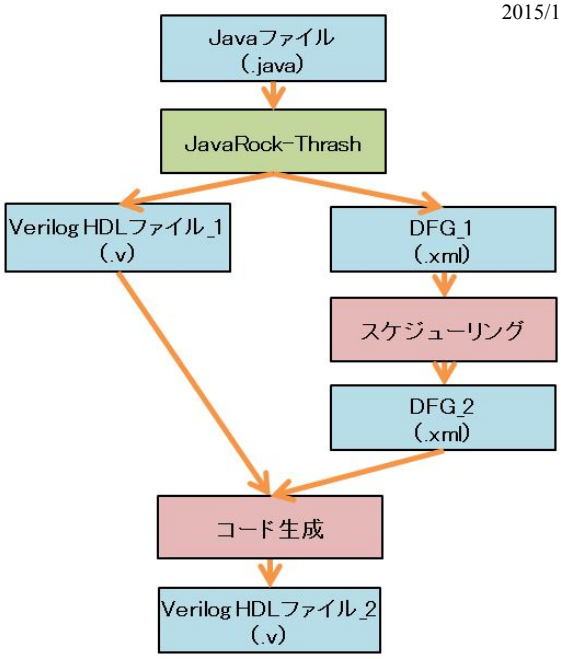


図 3　パイプライン化機構の構成図
Fig. 3　Diagram of pipeline system

反映させる．今回実装した機構の構成図は**図 3** のようになっている．大きく分けて再スケジューリングを行うスケジューリング部と Verilog HDL のソースコードを変更して出力するコード生成部からなる構成である．

4.2.1　スケジューリング部分

JRT の DFG が持つ情報のうち，モジュロスケジューリングで使用する情報は以下のものがある．

- ノード番号
- ノードの種類（配列，変数，演算器）
- 優先度
- 先行ノードの有無
- レイテンシ
- 実行開始ステップ数

ノードの種類によってスケジューリングの方法が異なるため，各ノードを識別するためにノード番号とノードの種類を把握する必要がある．DFG から上記の情報を読み込んだら，次に各ノードの種類と数をもとに資源予約表のサイズを決定する．続いて，図 2 の例で示したように各ノードを資源予約表に埋めるため，優先度が高く先行ノードを持たないノードから順次スケジューリングを実行する．資源予約表に記録する相対スケジュールは，レイテンシと実行開始ステップ数を基準にして決定する．全てのノードを資源予約表に埋めたら，スケジューリングが終了である．

4.2.2　コード生成部

前項のスケジューリングによって生成された DFG の結果を JRT の生成した HDL ファイルに反映させる．今回実装したのはループ展開と同様で入れ子を含まないループのみを対象としている．データパスは変更せず，ステートマシンの部分を変更することでデータの入出力のタイミングを変えることでパイプライン化を実装した．

JRT は DFG のノードごとにステートマシンを設けて実行のタイミングを管理している．通常はカウンタが特定の値になったら実行されるが，パイプライン化の際には実行されるタイミングも変動させることができるように変数で管理する必要がある．このレジスタの動作としては初期値として相対スケジュールが与えられて，対象処理が実行されるごとに開始間隔の大きさだけ値を増やす仕組みにした．そうすることで一定間隔ごとに特定の処理を実行することができ，パイプラインが完成する．

5. 評価

以上のループパイプライン化を JRT に適応した形で実装し評価を行った．

5.1 結果

まずはループのパイプラインを加算と代入のみの単純な式で実装した結果が**表 2** である．対象とした FPGA は Artix7(XC7A100T) であり論理合成ツールには XilinxISE14.7 を用いた．各種のリソース量と動作周波数，処理サイクル数を比較した．パイプライン化を行うことで動作周波数を低下させることなく，処理サイクル数を 50%削減できている．またリソース量もループ展開と比較してパイプラインの方がレジスタで 53%，LUT で 89%少なくなっている．ループに関する高速化を行わなかった場合と比較して約 2 倍の実行速度を得ることができた．

6. 考察

パイプライン化の効果について考察し，今回の実験を通して得られた今後の課題について言及する．

6.1 パイプライン化によるリソース量と動作周波数

この実験ではループの回数を 16 に設定していたが，実際の CFD コードではループの回数がさらに多くなることが考えられる．したがって必要なループ展開の展開数も増え，さらに動作周波数が低下することが考えられる．また論理合成ツールの性能を考えた場合，ループ展開により回路が複雑になるとそれだけ論理合成に時間を要する．これより高位合成による開発期間の短縮といった目的を満たせなくなる．これら 2 つの時間に対する課題を解決するためにもループのパイプライン化は有効であるといえる．

6.2 入れ子ループのパイプライン化

今回実装したのは入れ子の内側のループに対してのみの実装であった．しかし，それによるハードウェア・アクセラレーションの効果は小さい．したがってさらに大局的な最適化を考慮する必要があり，不完全入れ子ループのパイプライン化に対応する必要がある．しかしながら，文献 [12] をそのまま JRT に適応するのは難しい．その理由として文献 [12] では FMSD を用いて回路の生成を行っており，この高位合成専用の実装方法だからである．

```
//loopA
for(int i=0; i¡N; i++){
    //loop1
    for(int j=0; j<N1; j++){
        ...
    }
    //loop2
    for(int j=0; j<N2; j++){
        ...
    }
    //loop3
    for(int j=0; j<N3; j++){
        ...
    }
}
```

図 4 不完全入れ子ループの例
Fig. 4 Example of irregular nested loop

JRT において実装するには，2 つのことを考慮する必要がある．ひとつは全体を最適なパイプラインにするために個々のループをスケジューリングするということ．もうひとつはループ間のデータの依存関係を明確にすることである．1 つ目は個々にループのパイプライン化を行ったとして，入れ子ループの全体をパイプライン化する際には資源制約があるため必ずしも最適になるとは限らない．従って個々のループのスケジューリングも同時に変更する必要がある．二つ目は JRT の性能によるものである．JRT では単一ループ内でしかスケジューリングができないので，並列なループ間でのスケジューリグを可能にする必要がある．

不完全入れ子ループの例を**図 4** に示す．この例では loopA の内側に 3 つのループ loop1，loop2，loop3 が存在している．これらのループが依存関係がなく独立している場合はループ融合により完全入れ子ループに変換し，その後ループつぶしをすることで入れ子を解消することが可能となる．しかし，各ループに依存関係がある場合，パイプライン化を実現するためには loop1〜loop3 を一つの処理のまとまりと考える必要がある．

6.3 メモリアクセス

高位合成を用いて科学技術計算を行っていくことを考えた場合，大容量のデータを扱うことが必須となる．その場合 FPGA の内部メモリだけでは容量が足りず，そのため外部メモリも使用する．そうすると内部メモリのアクセスと比較して速度の遅い外部メモリのアクセスがボトルネックとなる可能性が高いので，メモリアクセスの方法についても考慮する必要がある．

パイプライン化を行う際に配列にアクセスするためのポートが一つしかないというのはリソースの制約条件になるので，開始間隔が大きくなってしまう．現状では多くの

表 2　ループ展開とパイプラインの比較
Table 2　Comparison of the loop unrooling and the pipelining

	reg	LUT	Fmax[MHz]	処理サイクル数	処理時間 [μs]
JRT	224	289	358	4120	11.5
JRT（展開）	421	3386	178	2076	11.7
パイプライン	199	367	358	2076	5.80

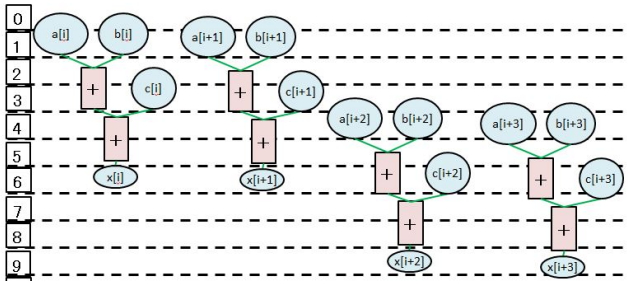


図 5　マルチポートを持つ配列のスケジューリング
Fig. 5　Scheduling of multi port array

処理系で配列のポートが 1 つのものを扱っているため，この課題が残っていると考えられる．そこで解決方法として配列の複製を用意したりマルチポートの配列を用いる方法 [14] がある．その他にメモリアクセスの高速化手法として，パイプライン化のスケジューリングの際に，アクセス順序にも注目して行うことでメモリアクセスの空間的局所性を高めることができ，ボトルネックとなるメモリアクセスを高速化する方法もある．

図 1 の例では配列の入出力ポートが一つしかないため，同一 step 上に同一の配列をスケジューリングすることができず，そのことが開始間隔が大きくなる要因となっていた．しかしマルチポートをもつ配列を設計した場合，**図 5** のようにスケジューリングすることが可能となる．アクセスするタイミングが同時になるので演算器の使用量は増加するかもしれないが，処理サイクル数を減らすことが可能となる．その結果処理時間の短縮を期待できる．

7. まとめ

本論文では高位合成によるハードウェア・アクセラレーションについて述べた．プログラムのループ部分をパイプライン化し，並列性を抽出することで十分な効果を発揮できることがわかった．ただし CFD コードには単一のループの他，入れ子のループも多数存在することからそれらに対応したパイプライン化を実装することが求められる．

今後は入れ子のループに対するパイプライン化を実装することで，局所的な回路ではなく実用的なプログラム全体に対してもハードウェア・アクセラレーションとして高位合成を使うことができる．また，他の高位合成ツールと性能比較を行い評価する必要がある．

謝辞　本研究の一部は，文部科学省特別経費「持続可能社会にむけた知的情報空間技術の創出」および JSPS 科研費基盤研究 (C) 25330067 による支援を得た．ここに記して感謝する．

参考文献

[1] David Pellerin and Scott Thibault. *Practical fpga programming in c*. Prentice Hall Press, 2005.
[2] Maxeler Technologies. Maxeler technologies. https://www.maxeler.com/products/software/maxcompiler/.
[3] JavaRock. JavaRock. http://javarock.sourceforge.net/.
[4] 小池恵介, 榎戸健二, 船田悟史, 三好健文, 中條拓伯. JavaRock-Thrash の実装. 組込みシステムシンポジウム論文集, pp. 41–48, 2013.
[5] 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. 98, No. 3, pp. 373–383, 2015.
[6] Atsushi Hashimoto, Keiichi Murakami, Takashi Aoyama, Manabu Hishida, Shinji Ohno, Masahide Sakashita, Paulus Lahur, and Yukio Sato. 高速流体ソルバ FaSTAR の開発.
[7] 隆哲横山, 博和森下, 保範長名, 直行藤田, 英晴天野. FPGA による UPACS サブルーチンの高速化 (リコンフィギャラブル応用). 電子情報通信学会技術研究報告. RECONF, リコンフィギャラブルシステム, Vol. 109, No. 26, pp. 103–108, may 2009.
[8] 山本一臣. CFD 共通基盤プログラム UPACS の開発. 第 14 回数値流体力学シンポジウム, 2000, 2000.
[9] 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. *Design Test of Computers, IEEE*, Vol. 28, No. 4, pp. 28–37, July 2011.
[10] 啓福井, 昌宏藤田. 高位合成ツールを利用したハードウェアアルゴリズムの最適化 (最適化技術, システム設計及び一般). 電子情報通信学会技術研究報告. VLD, VLSI 設計技術, Vol. 111, No. 40, pp. 51–56, may 2011.
[11] A. Morvan, S. Derrien, and P. Quinton. Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, Vol. 32, No. 3, pp. 339–352, March 2013.
[12] 崇竹中, 一敏若林, 優佳中越. 不完全ネストループに対するループパイプライン (暗号と高位設計, システムオンシリコンを支える設計技術). 電子情報通信学会技術研究報告. VLD, VLSI 設計技術, Vol. 111, No. 450, pp. 37–42, feb 2012.
[13] 中田育男. コンパイラの構成と最適化. 朝倉書店, 2009.
[14] Ameer M.S. Abdelhadi and Guy G.F. Lemieux. Modular Multi-ported SRAM-based Memories. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pp. 35–44, New York, NY, USA, 2014. ACM.