Vol.2021-ARC-243 No.30
Vol.2021-SLDM-193 No.30
2021/1/26

ループ平坦化による LLVM/Polly におけるループ融合の促進

伊澤　昇平† 　外處　尭之† 　瀬戸 謙修† 　立岡 真人‡ 西田 嘉人‡

†東京都市大学工学研究科 〒158-8557 東京都世田谷区玉堤 1-28-1

E-mail: †g2081205@tcu.ac.jp

‡株式会社ソシオネクスト

要約　現在の高位合成技術では、生成されるハードウェアの性能を向上するために、コード最適化を必要とすることがある。外側ループシフト(Outer Loop Shifting, OLS)と呼ばれる多面体モデルベースのループ融合技術を用いることで、多重の完全ネストループを構築できるが、その技術を LLVM/Polly 上で適用するとコード制約が必要となる。提案手法 MF3 により、入力コードを 1 重ループに平坦化した後でループ融合することで、OLS の制約を解決し、より多くの例題を完全ネストループに変形することを目的とする。提案手法 MF3 は、OLS をループ平坦化したループ構造と同等の回路を生成した。回路性能について、最適化前よりも総実行サイクルを最大 37.3%削減できた。しかし、面積は最大 19.5%、消費電力は最大 23.3%増加した。MF3 は、LLVM/Polly において、OLS よりも多くの例題を完全ネストループに変形することが可能であることが分かった。

キーワード　ループ融合、LLVM/Polly、高位合成

Abstract　The current High-Level Synthesis requires code optimization to improve the performance of the generated hardware. A polyhedral model-based loop fusion technique called Outer Loop Shifting (OLS) can be used to build fully nested loops, but applying that technique on LLVM / Polly requires code constraints. By using the proposed method MF3, the input code is flattened into a non-nested loop and loop fusion is performed to overcome the OLS limitation and transform more examples into fully nested loops. The MF3 produced a circuit equivalent to a loop structure as that by the OLS followed by loop-flattening. As for circuit performance, the total execution cycles was reduced by up to 37.3% compared to those before optimization. However, area increased by up to 19.5% and power consumption increased by up to 23.3%. We found that MF3 can transform more examples than OLS into fully nested loops.

Keywords　Loop fusion, LLVM/Polly, High-Level Synthesis

1. はじめに

C 言語などの高級言語から HDL(Hardware description language)を自動生成する高位合成技術では、自動生成されるハードウェアの性能を十分に保つために高位合成前にコード最適化を行う必要がある。高位合成向けの最適化には、ループ最適化、メモリアクセス最適化などが存在する。ループ最適化は、for、while ループなどのサイクル数を削減する。メモリアクセス最適化は、配列のアクセス数を削減する。本稿では、ループ最適化に着目する。

ループ最適化の例として、ループ融合やループ平坦化が挙げられる。まず、ループ融合の例を図 1 に示す。

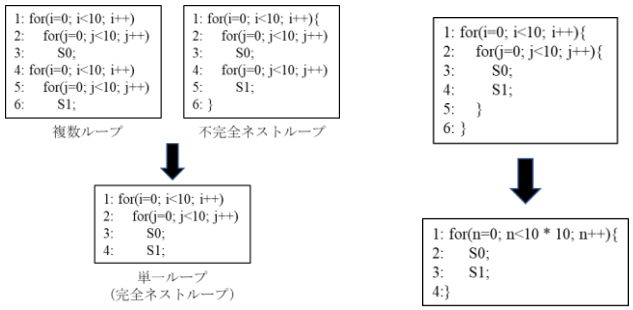

図 1.(左)ループ融合の例
(右)ループ平坦化の例

ここで、S0、S1 は実行文を示す。ループ融合は、図 1(左)上のような複数ループや、入れ子構造の不完全ネストループを図 1 (左)下のように単一ループ(完全ネストループ)の形に融合し、サイクル数を削減する最適化である。図 1(左)の例では、サイクル数が 200 回から 100 回に削減されていることが分かる。また、図 1(左)上では、S0、S1 が逐次実行されているのに対して図 1(左)下では、並列実行可能になる。

次に、ループ平坦化の例を図 1(右)に示す。ループ平坦化は、図 1(右)上の多重ループを図 1(右)下の 1 重ループに平坦化させる最適化である。多重ループは、ループ変数 i=0 のときのループ変数 j のループ処理が完了しないと i=1 にインクリメントされないため、ループ内に待機時間が存在する。ループ平坦化により、待機時間を除去することができ、さらに、完全ネストループを構築することができる。高位合成において、完全ネストループにパイプライン化を適用することで実行文の並列性が極めて高くなる。

本稿で用いるループ融合ツール LLVM/Polly[1][2][3][4] では、データの依存関係のあるループをループ融合する場合、不完全ネストループを生成する問題点がある。これを解決する方法としてループ融合ツール Pluto を

Vol.2021-ARC-243 No.30
Vol.2021-SLDM-193 No.30
2021/1/26

用いた先行研究 OLS[5](Outer Loop Shifting, 外側ループシフト)が提案された。OLS は図 1(左)下のような多重の完全ネストループを生成するため、ループ平坦化を行うことでより性能向上することが期待される。しかし、LLVM/Polly で OLS を適用する場合、複数の制約が発生する。この制約については、3.1 節で述べる。

2.1 節で述べる多面体モデルによるコード最適化では、多次元スケジュール（以下、スケジュール）を変更するステップ 1 と、変更したスケジュールに基づいたコードを完全ネストループとして出力するステップ 2 が存在する。Pluto は、ステップ 1 ができ、ステップ 2 はオプションを付けることで完全ネストループを構築できる。しかし、Polly では、ステップ 1 は Pluto と同様にできるが、ステップ 2 はオプションを付けることでは対応できないという問題点がある。Polly は Pluto よりも対応する入力コードが広いため、Polly でステップ 2 を解決することでより多くの入力コードに対応できる。本研究では、Polly を用いた場合のステップ 2 の問題点を解決する。

よって、本稿では、LLVM/Polly 上で OLS を適用する際の制約と、多面体モデルにおけるコード最適化のステップ 2 を解決する手法として、複数の多重ループを単数で 1 重の完全ネストループに変換する手法を提案する。そして、OLS を平坦化したループ構造と提案手法を高位合成により比較を行い、提案手法の有用性を検証することを目的とする。

2. 多面体モデル

本節では、LLVM/Polly 内の変換方式である多面体モデルと多面体モデルの重要な概念である *schedule* について述べる。

多面体モデルは、様々なループ変換を柔軟に適用できるプログラム表現である。LLVM/Polly は、入力コードを多面体モデルに変換し、多面体モデルに様々な最適化を適用する。最適化された多面体モデルは C 言語等にコード変換して出力される。

schedule は、Si = (i0 , i1 , … , iN)のような多次元ベクトルで表される。多次元ベクトルの各次元(i0~iN)は、定数とループ変数で表され、左側から順に実行される。例えば、多重ループでは、外側ループが多次元ベクトルの左側に位置する。*schedule* における多次元ベクトルには、定数のみで表されるスカラ次元と定数およびループ変数で表されるループ次元が存在する。スカラ次元は、ループの実行順序を決定するために使用され、ループ次元は、実行されるループを表す。3.1 節で述べる先行研究 OLS、第 3 章で述べる提案手法 MF3 では、*schedule* を変更してループ変換を行っている。

3. 先行研究

3.1. 外側ループシフト（OLS）

OLS(Outer Loop Shifting)は、ループを分割する 0 以外の値を持つスカラ次元を外側のループ次元に移動させ、ループ次元間に存在するスカラ次元を削除する。そして、ループ融合を行い、多重の完全ネストループを構築する手法である。図 2 にサンプルコードと *schedule*、図 3 に Polly にあらかじめ実装されているループ融合を適用したコードと *schedule*、図 3 のループ融合したコードに OLS を適用したときのコードと *schedule* を図 4 に示す。

```
1: for(i=0; i<N; i++)  //schedule：(i0, i1, i2)
2:   for(j=0; j<N; j++)
3:     B[i]=B[i] + A[i][j]              //S0：(0, i, j)
4: for(i=0; i<N; i++)
5:   for(j=0; j<N; j++)
6:     D[i] = D[i] + C[i][j] + B[i]     //S1：(1, i, j)
```

図 2. サンプルコードと *schedule*

```
1: for(i=0; i<N; i++) {          //schedule：(i0, i1, i2)
2:   for(j=0; j<N; j++)
3:     B[i]=B[i] + A[i][j]              //S0：(i, 0, j)
4:   for(j=0; j<N; j++)
5:     D[i] = D[i] + C[i][j] + B[i]     //S1：(i, 1, j)
6: }
```

図 3. Polly のループ融合結果

```
1: for(i=0; i<N; i++)                   //schedule(i0, i1, i2)
2:   for(j=0; j<N+1; j++){
3:     if(i<=N-1)
4:       B[i]=B[i] + A[i][j]            //S0：(i, 0, j)
5:     if(i>=1)
6:       D[i] = D[i] + C[i][j] + B[i-1]  //S1：(i+1, 0, j)
7: }
```

図 4. OLS 後のコードと *schedule*

図 2 は、1~3 行目、4~6 行目がそれぞれ 2 重ループの完全ネストループで構成されている複数の多重ループである。図 3 では、図 2 の複数ループが融合され、不完全ネストループを構築している。図 4 は、図 3 のコードに OLS を適用したもので、完全ネストループを構築している。また、*schedule* において、図 2 は i0 がスカラ次元、i1、i2 がループ次元である。図 3、図 4 では、i1 がスカラ次元、i0、i2 がループ次元である。

図 3 の Polly に実装されているループ融合は、先述した通り不完全ネストループを生成し、ハードウェアの性能向上が制限されてしまう問題点がある。この問題点を解決するために、OLS を適用する。しかし、Polly を用いて、OLS を適用する場合、以下のようなコーディング制約が発生する。

1. ループ融合対象のループが隣接している
2. ループ融合対象のループは完全ネストループ
3. 各ループのループ回数が同一

条件 1 について、多面体モデルの最適化は、SCoP(Static Control Parts)ごとに行われる。ループ融合対象のループが隣接していない場合、SCoP に含まれない。Polly の仕様のためユーザー側でループを隣接させて記述する必要がある。条件 2 について、不完全ネストループ

Vol.2021-ARC-243 No.30
Vol.2021-SLDM-193 No.30
2021/1/26

に対してループ融合を行うと完全ネストループを構築できない可能性がある。条件分岐を用いるなど、ユーザー側で完全ネストループを構築する必要がある。条件 3 について、ループ回数が異なるループに対して OLS を適用するとループが分割され、不完全ネストループを構築する可能性がある。そのため、LLVM/Polly 上で OLS を適用する場合、各ループのループ回数が統一されている例題に限定される。本稿では、より多くの例題に対応させるために、条件 3 を解決する手法を提案する。

3.2. MFF(Modulo Free Flatten)

本節では、提案手法 MF3 でループ平坦化を行う際に用いるループ平坦化手法について述べる。

MFF[6]は、ループ平坦化後に新しく生成されるループ変数から、元のループ変数の値を計算する部分に発生する剰余算器をその他の演算器に置き換える手法である。図 5 にサンプルコード、図 5 に MFF 手法を適用したコードを図 6 に示す。

```
1: for(i=0; i<8; i++)
2:     for(j=0; j<9; j++)
3:         for(k=0; k<11; k++)
4:             c[i][j] =  c[i][j] + a[i][k] * b[k][j]; //S0
5: for(i=0; i<8; i++)
6:     for(j=0; j<12; j++)
7:         for(k=0; k<9; k++)
8:             d[i][k] = d[i][k] + c[i][j] * e[j][k]; //S1
```

図 5.サンプルコード

```
1: for(x=0; x<8*9*11; x++){
2:     i = x / (9 * 11);
3:     j = x / 11 – i * 9;
4:   k = x – i * (9 * 11) – j * 11;
5:       c[i][j] =  c[i][j] + a[i][k] * b[k][j]; //S0
6: }
7: for(y=0; y<8*12*9; y++){
8:     i = y / (12 * 9);
9:     j = y / 9 – i * 12;
10:  k = y – i * (12 * 9) – j * 9;
11:      d[i][k] = d[i][k] + c[i][j] * e[j][k]; //S1
12: }
```

図 6.MFF 手法適用コード

図 5 は、4 行目の演算結果を 8 行目で用いる依存関係のあるループである。また、各ループでループ回数が異なる例題である。図 5 のサンプルコードに MFF 手法を適用することで、図 6 のようにそれぞれの多重ループが 1 重ループに平坦化される。MFF 手法を適用しないループ平坦化では、図 6 の 3,4 行目、9,10 行目のループ変数計算部分の演算器に剰余算器が用いられる。剰余算器は、乗算器などの演算器よりも規模が大きいため、高位合成において回路面積が増加する原因になる。図 6 のように減算器と乗算器に置き換えることによって、回路面積増加を抑えることができる。

4. 提案するループ融合手法（MF3）

本節では、前節 3.1 で述べた OLS のコーディング制約の条件 3 を解決する手法として、MF3 を提案する。MF3(Modulo Free Flatten Fusion)は、ループ融合前に対象のループをそれぞれ MFF 手法により平坦化する。その後、ループ融合することで完全ネストループを構築する手法である。MF3 の変換フローを図 7 に示す。

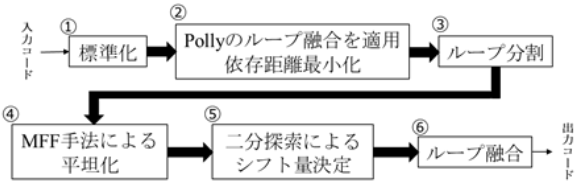

図 7. MF3 の変換フロー

まず、①のフローで図 5 のような入力コードに対して最適化を適用しやすくする標準化を行う。その後、②で Polly のループ融合を適用して最内側ループの依存距離を最小にする。依存距離とは、配列などに保持されたデータが次に使用されるまでの間隔のことである。③のループ分割では、Polly のループ融合を適用して生成された不完全ネストループを複数で多重の完全ネストループに変換する。④のフローで MFF 手法による平坦化で図 5 のような複数多重の完全ネストループを図 6 のような 1 重の完全ネストループに変換する。⑤で生成された複数の 1 重ループを融合するために二分探索アルゴリズムを用いてシフト量を算出する。シフト量は、入出力依存関係のある実行文を並列に実行させるために、実行文の実行をどのくらい遅らせるかを決定する値である。図 6 では、最初のループの実行文(5 行目)の c 配列が、次のループの実行文(11 行目)に使われている。最後のフロー⑥ループ融合により、並列実行が可能でも、データが準備されていなければ、正しく計算されない。データが準備されるまで、実行文の実行を開始しないためにシフト量が必要になる。シフト量は、入力コードに配列の添え字を統一することやループ分割を行い、各ループで 1 つの実行文にする書き換えを行うことによって最小の値が算出される。シフト量を算出後、ループ融合を行い、単数で 1 重の完全ネストループに変換されたコードを出力する。出力されたコードに対して、入力コードの演算や配列の添え字計算部分を追加することで図 8 のようなコードを生成する。

```
1: for(x=0; x<873; x++){
2:     i = x / (9 * 11);
3:     j = x / 11 – i * 9;
4:   k = x – I * (9 * 11) – j * 11;
5:   if (x <= 791)
6:       c[i][j] =  c[i][j] + a[i][k] * b[k][j]; //S0
7:   y = x – 10;
8:   i1 = y / (12 * 9);
9:   j1 = y / 9 – i * 12;
10:  k1 = y – I * (12 * 9) – j * 9;
11:  if (x >= 10)
12:      d[i1][k1] = d[i1][k1] + c[i1][j1] * e[j1][k1]; //S1
13: }
```

図 8.MF3 手法を適用したコード

ここで、図 8 の 5 行目の条件文は、6 行目の実行文がオーバーフローしないための制御文である。7 行目はシフト量を考慮したループ変数を計算するコードで、8、9、10 行目はシフト量を考慮した配列の添え字計算

である。11 行目の条件文は、シフト量の分だけ、実行分 S1 の実行を遅らせている。

図 5 の場合では、S0 の実行に 792 回、S1 の実行に 864 回ループするため総ループ回数は 1656 回になる。MF3 を適用した図 8 のコードにすることで総ループ回数は 873 回となった。総ループ回数が削減できた理由として、図 6 で逐次実行していた各実行文が並列実行可能になったためである。図 5 は、OLS が適用できないループ回数の異なる例題である。MF3 により図 5 を図 8 のように完全ネストループを構築することができることが確認できた。よって、提案手法 MF3 は、LLVM/Polly に先行研究 OLS を含めることではループ融合できなかったループ回数が異なる例題を完全ネストループに変換できた。さらに、元のコードを平坦化して融合することにより、実行文を並列実行可能にするため、ループ回数を削減することが可能である。

5. 実験結果

5.1. 実験準備

本章では、5.2 節で先行研究 OLS を平坦化した場合(融合→平坦化)と、提案手法 MF3(平坦化→融合)の差異でコード出力の変化を確認する。5.3 節で最適化前コードと OLS+平坦化適用コード、MF3 適用コードの高位合成結果を示す。5.4 節で OLS を適用できない例題に対し、MF3 を適用した結果を示す。

本稿の実験は、サイクル数を最小にする高位合成のオプションを使用して実験を行った。また、商用の高位合成ツールにて単一スレッドの回路を合成し、ループパイプライン化を行った際の性能と、論理合成を行った際の面積や消費電力への影響を評価した。テクノロジライブラリはプロセスルールが 45nm のものを使用した。

5.2. ループ変換結果

MF3 は、複数の多重ループを単数の 1 重ループに変換しているが、先行研究の OLS は、複数の多重ループを融合して単数の多重ループに変換している。高位合成を行う際に条件を公平にするために OLS の結果を MFF により平坦化した。

本節では、MF3 手法 (平坦化ののち融合)と OLS 後に平坦化させる場合(融合ののち平坦化)の順序の差異で、出力されるループ構造の変化を確認する。確認する例題として、OLS、MF3 共に完全ネストループを構築することが可能な mms、gemver、tce の 3 つを使用した。mms(matrix multiplication solver)は、行列乗算を 1 回、行うプログラムである。gemver は、BLAS(Basic Linear Algebra Subprograms)からの複数の行列ベクトル乗算を行うプログラムである。tce は、畳み込み演算を 4 回行うプログラムである。一例として、mms をそれぞれ MF3 適用したもの、OLS を適用し平坦化したものをそれぞれ図 9,図 10 に示す。

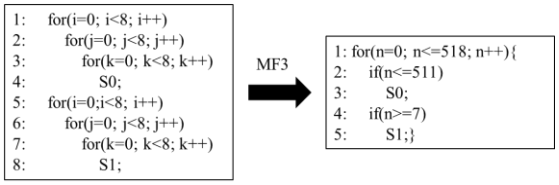

図 9. MF3 適用結果(平坦化ののち融合)

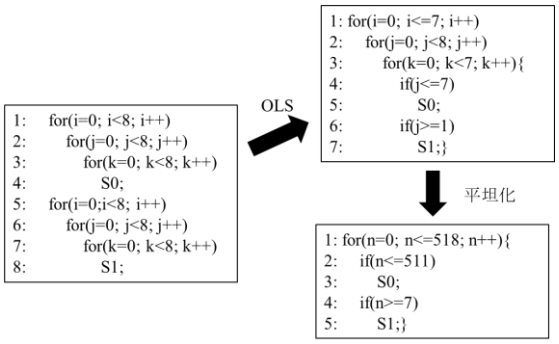

図 10. OLS の平坦化結果(融合ののち平坦化)

図 9、図 10 の結果から、MF3 を適用して平坦化後に融合したループ構造と、OLS による融合ののちに平坦化を行ったループ構造は同一になることが分かった。これは、最適化前のコードにおいて、あらかじめ配列の添え字を統一すること(例えば、S0、S1 にそれぞれ a[i][j]と a[k][j]がある場合、どちらか一方に添え字を揃えること)や、各ループに 1 つの実行文に書き換えるループ分割を行うことにより、どちらの手法でも最小のシフト量を算出することができたため、同一の構造になったと考える。その他の例題 gemver、tce も出力されるループ構造は同一であった。

したがって、各ループのループ回数が同じ場合、融合をして平坦化する手法と平坦化して融合する手法は、手順に関わらず、同一のループ構造になると考える。

5.3. 高位合成結果

本節では、5.2 節で用いた 3 つの例題を最適化し、単数で 1 重の完全ネストループを用いて高位合成を行った。その結果を表 1 に示す。ここで、動作周期は 5.0ns である。横軸の No fusion は最適化前のオリジナルコード、OLS＋MFF は先行研究 OLS に MFF 手法を適用したコード、MF3 は提案手法を適用したものである。縦軸の総実行時間は、サイクル数と動作周期をスラックで減算した値の積で表され、プログラムの演算処理が全て完了するまでの時間を示す。スラックは回路の余裕度を示す。総実行サイクルから下の項目は、すべて No fusion を 100%とした時の百分率を示し、100 よりも小さい値の場合、No fusion よりも回路性能が向上する、100 よりも大きい値の場合、回路性能が悪化することを意味する。本稿では、No fusion と最適化後の総実行サイクルと面積比、消費電力比の善悪を比較す

Vol.2021-ARC-243 No.30
Vol.2021-SLDM-193 No.30
2021/1/26

表1. 各手法の高位合成結果

	mms			gemver			tce		
	No fusion	OLS+MFF	MF3	No fusion	OLS+MFF	MF3	No fusion	OLS+MFF	MF3
総ループ回数	1024	519	519	256	71	71	131072	36856	36856
サイクル数	3135	2658	2658	2143	2173	2173	380927	266213	266213
動作周期(ns)	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
総実行時間(ns)	3499	3081	3081	1924	2030	2030	622816	390268	390268
面積(μm^2)	3270	3426	3426	5238	5797	5797	5507	6399	6399
スラック(ns)	3.884	3.841	3.841	4.102	4.066	4.066	3.365	3.534	3.534
消費電力(nW)	384666	407963	407963	645983	758795	758795	519020	639779	639779
総実行サイクル(%)	100	88.1	88.1	100	105.5	105.5	100	62.7	62.7
面積比(%)	100	104.8	104.8	100	110.7	110.7	100	116.2	116.2
消費電力比(%)	100	106.1	106.1	100	117.5	117.5	100	123.3	123.3
サイクル×面積比(%)	100	92.3	92.3	100	116.7	116.7	100	72.8	72.8
サイクル×消費電力比(%)	100	93.4	93.4	100	123.9	123.9	100	77.2	77.2

る。そして、総実行サイクルと面積比の積で表されるサイクル×面積比、総実行サイクルと消費電力比の積で表されるサイクル×消費電力比で回路性能を総合的に評価する。

本稿の実験では、No fusion よりスラックの値が著しく低下する直前の開始間隔で実験を行った。

表 1 の横軸の OLS+MFF、MF3 の各項目の値は同値であった。これは、5.2 節より、OLS+MFF と MF3 のループ構造が同一になるため、回路性能が同等になることが確認できた。

最適化後の総実行サイクルは、mms では 11.9%、tce では 37.3%、No fusion より削減したが、gemver では 5.5%、No fusion よりも悪化した。用いた 3 つの例題は、完全ネストループを構築して並列実行が可能になったが、mms と tce では、ループパイプライン化による並列実行の効果によりサイクル数が削減できたと考える。一方、gemver では、ループ平坦化により、配列アクセスが増加するため、並列実行の開始間隔が長くなり、サイクル数を削減できなかったと考える。gemver のような配列アクセスが増加する例題においては、メモリアクセス最適化を適用することで総実行サイクルの短縮が期待される。

最適化後の面積は、mms では 4.8%、gemver では 10.7%、tce では 16.2%、No fusion よりも増加した。提案手法 MF3 は、単数で 1 重の完全ネストループに変換されるため、図 8 のように新たにループ変数が生成される。このループ変数のビット数増加によるレジスタの増大、元のループ変数を計算して保持するための変数の増加が面積増加の原因と考える。また、並列実行を可能にするために比較器などのリソースが増加したことも面積増加の一因であると考える。gemver や tce は、mms よりも面積増加率が高いが、これは、動作周期制約内で処理を完了させるために大型演算器を用いたこと、特にループ平坦化、融合したことで新たにループ変数のビット幅が増加したことが原因と考える。リソース増加に伴い、消費電力もそれぞれ増加したと考える。

最適化後のサイクル×面積比は、mms では 7.7%、tce では 27.2%、No fusion より改善したが、gemver では 16.7%、No fusion より悪化した。mms と tce では、総実行サイクルの削減率が面積増加率より上回っているため、回路性能は改善した。gemver では、総実行サイクルが増加したこと、さらに、配列アクセスが増加したことが加わり、回路性能は悪化した。

最適化後のサイクル×消費電力比は、mms では 6.6% 、tce では 22.8%、No fusion より改善したが、gemver では 23.9%、No fusion より悪化した。mms と tce では、時間比×面積比と同様に総実行サイクルの削減率が消費電力比よりも上回っているため、回路性能は改善した。gemver では、総実行サイクルも消費電力比も悪化するため、総合的に悪化した。

以上の結果より、提案手法 MF3 は、単数で 1 重の完全ネストループを構築することができ、また、OLS+MFF と各項目が同値になることを確認した。また、サイクル数を削減できることが分かった。一方、サイクル数とスラックはトレードオフの関係にあり、最適解を見つけ出す必要があることが分かった。MF3 は、比較的単純な行列計算をハードウェアで行う場合においてループ平坦化、融合は有効である。しかし、回路面積や消費電力が増加するという問題点があることが分かった。面積や消費電力の増加を抑えることが今後の課題である。

5.4. ループ融合結果

本節では、LLVM/Polly 上で様々な例題に対して OLS、MF3 それぞれの手法を適用した。その結果を表 2 に示す。

表 2. ループ融合結果

	mms	tce	dct	AES (decrypt)	vgg19	2mm	2mm_d	3mm	3mm_d	atax	bicg
OLS	○	○	×	○	×	×	×	×	×	×	×
MF3	○	○	○	○	○	○	○	○	○	○	○
	doitgen	doitgen_d	mvt	gemm	gemver	gemver_opt	gesummv	symm	syrk	syr2k	trmm
OLS	×	×	-	-	○	○	-	×	-	-	-
MF3	○	○	○	○	○	○	○	×	○	○	○

ここで、○は、完全ネストループを構築できた場合、

×は構築できなかった場合を示す。ーは、依存関係解消のために OLS を必要としない場合など条件が変わる場合を示している。表 2 の横軸は、用いた例題名を示す。「_d」は各ループのループ回数が異なる場合を示す。ループ融合結果の具体的な一例として、mms_d のコードに対して OLS を適用したものを図 11 に示し、MF3 を適用したものを図 12 に示す。

```
1:   for(i=0; i<8; i++)
2:     for(j=0; j<9; j++)
3:       for(k=0; k<11; k++)
4:         S0;
5:   for(i=0;i<8; i++)
6:     for(j=0; j<12; j++)
7:       for(k=0; k<9; k++)
8:         S1;
```

```
1:   for(i=0; i<=7; i++)
2:     for(j=0; j<=12; j++)
3:       if(j >= 9){
4:         for(k=0; k<=8; k++)
5:           S0;}
6:       else{
7:         for(k=0; k<=10; k++){
8:           S1;
9:           if(j >= 1 && k <= 8)
10:            S2;}
11:        }
```

図 11. mms_d の例題(左)と OLS コード(右)

```
1:   for(i=0; i<8; i++)
2:     for(j=0; j<9; j++)
3:       for(k=0; k<11; k++)
4:         S0;
5:   for(i=0;i<8; i++)
6:     for(j=0; j<12; j++)
7:       for(k=0; k<9; k++)
8:         S1;
```

```
1: for(n=0; n<=889; n++){
2:   if(n<=791)
3:     S0;
4:   if(n>=26)
5:     S1;}
```

図 12. mms_d の例題(左)と MF3 コード(右)

表 3. mms_d の高位合成結果

	mms_d	
	No fusion	**MF3**
総ループ回数	**1656**	**890**
サイクル数	**5663**	**5024**
動作周期(ns)	**5.0**	**5.0**
総実行時間(ns)	**6456**	**23266**
面積(μm^2)	**3456**	**4491**
スラック(ns)	**3.860**	**0.369**
消費電力(nW)	**376029**	**469049**
総実行サイクル(%)	**100**	**360**
面積比(%)	**100**	**130**
消費電力比(%)	**100**	**125**
サイクル×面積比(%)	**100**	**468**
サイクル×消費電力比(%)	**100**	**450**

mms_d は、LLVM/Polly における OLS のコーディング制約の条件 3 を満たさないループ回数が異なる例題である。図 11 より、ループ回数が異なる例題に対して OLS を適用すると、図 11(右)の 4、7 行目のようにループが分割され、不完全ネストループを構築した。一方、図 12 より、MF3 を適用することで完全ネストループを構築した。例として、mms_d の高位合成結果を表 3 に示す。

表 3 の結果から、MF3 は No fusion よりも総実行サイクルが 360%、面積は 30%、消費電力は 25%増加した。実行時間増加の理由としては、ループパイプライン化による並列実行が可能になったが、スラックの値が No fusion よりも 10 分の 1 になり、処理が完了する時間が増加したためだと考える。面積や消費電力が増加した理由も 5.3 節と同様にリソース増加によるものと考える。ループ回数の異なるループを平坦化、融合する場合、ループの添え字計算部分をビットシフト演算に置き換えることにより、スラックが改善される場合がある。

ループの融合結果より、LLVM/Polly において、MF3 は、OLS で対応できないループ回数が異なる例題に対してループ融合が可能になるため、OLS よりも多くの例題を融合することが可能になることが分かった。しかし、ループ回数の異なる例題の高位合成結果は、ループ回数が異なる例題に対して総実行サイクルが悪化し、面積や消費電力は悪化したことを確認した。また、スラックの値は、5.3 節の結果よりも低下した。ループ回数が異なる例題に対して MF3 を適用した際のスラックの低下は今後の課題と考える。

6. 結論

本稿で提案したループ融合手法 MF3 は、データ依存関係のあるループを平坦化し融合することで複数の多重ループを単数で 1 重の完全ネストループに変換することを可能にした。ループ回数が同じ例題の場合、MF3 で構築する構造と先行研究 OLS を平坦化した構造は、どちらも同一の完全ネストループを構築することが分かった。回路性能としては、総実行サイクルは最大 37.3%削減できた。しかし、面積は最大 16.2%、消費電力は最大 23.3%増加することが分かった。回路面積と消費電力の増加を抑えることや、スラックの低下を改善することは今後の課題である。また、MF3 は、LLVM/Polly 上で OLS を適用できないループ回数が異なる例題に対して、完全ネストループを構築することができた。

文　献

[1] The LLVM Compiler Infrastructure, https://llvm.org/. (参照 2020-10-13)

[2] C. Lattner, V.A dve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, CGO '04 , pp.75, 2004.

[3] Polly: LLVM Framework for High Level Loop and Data Locality Optimizations, http://polly.llvm.org/. (参照 2020-10-13)

[4] Yuta Kato, Kenshu Seto, “Loop Fusion with Outer Loop Shifting for High level Synthesis,” IPSJ Transactions on System LSI Design Methodology, Vol. 6, pp.71 75, 2013.

[5] 石川大輔ら,”高位合成における多重ループに対するパイプライン処理時のサイクル数オーバーヘッド削減を行うループ平坦化ツールの開発”vol. 117, no. 455, VLD2017-97, pp. 49-54, 2018