Vol.2013-SLDM-160 No.21
Vol.2013-EMB-28 No.21
2013/3/13

高位合成における多面体最適化のためのスレッド構成手法

須田 瑛大[1,a)]　高瀬 英希[1]　高木 一義[1]　高木 直史[1]

概要：高位合成においては，並列実行可能な処理を如何にして自動的に抽出するかが課題となっている．近年，ソフトウェア向けのコンパイラ分野において，多面体最適化と呼ばれる入れ子ループの自動並列化手法が注目されている．これは種々の線形代数学的演算を行うことで入れ子ループ構造内における依存性を解析し，反復空間をタイル分割するものである．本稿では，代表的な多面体最適化アルゴリズムである PLUTO を高位合成に応用することで，並列化された回路を自動設計する手法を提案する．まず，いくつかの論理スレッドを束ねて物理スレッドに割り当てることにより，PLUTO が生成する記述を高位合成向けに変換する手法を提案する．更に，依存性を考慮しつつ RAM へのアクセスを最適化し，RAM のバンド幅を有効に利用する手法を提案する．

キーワード：高位合成，多面体最適化，OpenMP，スレッド構成，入れ子ループ並列化

Threading Method for Polyhedral Optimization in High Level Synthesis

Akihiro Suda[1,a)]　Hideki Takase[1]　Kazuyoshi Takagi[1]　Naofumi Takagi[1]

Abstract: In the field of high level synthesis, there has been an issue on automatic extraction of parallelism. Recently, automatic parallelization methods called Polyhedral Optimization are attracting attention in the field of software compilation. Polyhedral Optimization splits the iteration space of nested loops into tiles by analyzing the dependencies with linear algebra computations. In this research, we propose a method to design parallelized circuits automatically by applying PLUTO, that is the representative Polyhedral Optimization algorithm, to high level synthesis. First, we propose a method to convert descriptions obtained from PLUTO for high level synthesis by assigning multiple logical threads to a physical thread. Furthermore, we propose a method to exploit bandwidth of RAM effectively by optimizing access to RAM considering dependencies.

Keywords: High Level Synthesis, Polyhedral Optimization, OpenMP, Threading, Nested Loop Parallelization

1. はじめに

高位合成においては，処理の並列化により実行サイクル数を削減し，回路の高速化を実現することができる．特に，入れ子ループについては，その処理を並列化できる可能性が高い．しかしながら，入れ子ループは一般にデータ依存性を持つため，人手で並列化を行うことは容易ではない．

近年，ソフトウェア用コンパイラ分野においては多面体最適化（polyhedral optimization）と呼ばれる入れ子ループ自動並列化手法が注目を集めており，既に実用化の段階に入っている．これは，種々の線形代数学的演算を用いて，入れ子ループが持つ依存性を解析することにより，依存性に違反しないように並列化されたコードを自動的に生成する手法である．

多面体最適化を高位合成に適用した研究 [1], [2] もいくつか存在する．しかしながら，これらの研究ではいずれも非一様依存性と呼ばれる複雑なデータ依存性を考慮できていない．また，文献 [1] はオフチップ RAM へのアクセスを考慮していない点にも改善の余地がある．

本稿では，代表的な多面体最適化アルゴリズムである PLUTO を高位合成に応用することで，並列化された回路

[1] 京都大学 大学院情報学研究科
Graduate School of Informatics, Kyoto University
[a)] suda.akihiro.82s@st.kyoto-u.ac.jp

を自動設計する手法を提案する．まず，PLUTO が生成する OpenMP ディレクティヴを高位合成向けに変換し，ハードウェアによるスレッドを構成する手法を提案する．更に，バスのバンド幅を有効活用できるように，オフチップ RAM に配置された配列の複数の要素を一度に読み書きすることにより，並列化された回路の実行に要するサイクル数を削減する手法を提案する．提案手法は非一様依存性にも対応しているため，高位合成における多面体最適化を，従来の手法よりも広範な処理に適用可能になる．

本稿の構成は以下のとおりである．第 2 章にて，実例を用いながら PLUTO アルゴリズムを紹介する．第 3 章にて，多面体最適化によって並列化されたソフトウェアを高位合成に適用するためのスレッド構成手法について述べる．さらに，第 4 章にて，依存性を考慮したバッファ構成手法について述べる．第 5 章にて提案手法を評価したのち，第 6 章にて本研究の結論を述べる．

2. 多面体最適化

近年，入れ子ループに対する多面体最適化理論に関する研究 [1], [2], [3], [4], [5], [6], [7], [8], [9] が盛んとなっている．多面体最適化とは，多面体に対する種々の線形代数学的演算を行なうことにより，入れ子ループにおける並列性の抽出や局所性向上等の最適化を行うアルゴリズムの総称である．この多面体は，ループ変数を要素とするベクトル x 及びその値域を表す行列 A, B とを用いて，$Ax \leq B$ の形で表される．

多面体最適化は 1990 年代初めから研究されてきたが，2000 年代になってソフトウェア分野でのコード生成に関する研究 [5], [6] が進んだ結果，GCC や clang などのソフトウェア用コンパイラにも実装されるようになった．多面体最適化の機能は GCC では GRAPHITE[7] と呼ばれ，2009 年から公式に採用されている．clang では Polly[8] と呼ばれ，こちらも 2012 年から公式に採用されている．しかしながら，高位合成の分野における多面体最適化 [1], [2] は未だ研究の途上にある．

2.1 例題プログラム

文献 [4] にて挙げられている例を図 1 に示す．本章では，このプログラムを例として，多面体最適化による並列化の手順を説明する．

この例のプログラムには以下の特徴があり，文献 [4] のほか，文献 [9] などいくつかの研究で着目されている．

- 非一様依存性を持つ．
- space 方向にまたがる依存性を持つ．

2.2 PLUTO アルゴリズム

多面体最適化による自動並列化手法として，PLUTO アルゴリズム [3],[4] を紹介する．PLUTO アルゴリズムは，

```
for (i=0; i<N; i++) {
  for (j=1; j<N; j++) {
      a[i,j] = a[j,i]+a[i,j-1]
  }
}
```

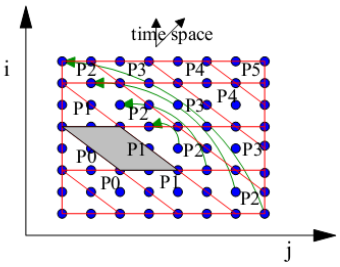

図 1 例題プログラム （文献 [4] Figure 3.11 より引用）

SCoP（Static Control Parts）とよばれる構造を持つプログラムに対して適用することができる．SCoP では，ループ境界，分岐条件，配列のインデックスは全てアフィン式で表される．アフィン式とは，ループ変数の線形結合と定数項との和からなる式である．それ故，SCoP は ACL（Affine Control Loop）とも呼ばれる．

PLUTO アルゴリズムを適用すると，入れ子ループの反復空間はタイルとして分割され，space 方向および time 方向のそれぞれについてベクトル (u_1, w, c_i, c_j) が計算される．space 方向のベクトルは，タイルを各スレッドに割り当てるために用いられ，time 方向のベクトルは，各スレッドがタイルを処理する順番を指定するために用いられる．図 1 中の P0，P1，...，P5 はスレッドを表し，各スレッドは下から上への順でタイルを処理する．

space 方向および time 方向のそれぞれのベクトル (u_1, w, c_i, c_j) について，(u_1, w) はタイル間にまたがる依存性を表す．$u_1 = 0, w = 0$ の場合は依存性は存在しない．$u_1 = 0, w > 0$ の場合は一様依存性（uniform dependence）が，$u_1 > 0, w \geq 0$ の場合は非一様依存性（non-uniform dependence）が存在する．(c_i, c_j) は，space 方向，time 方向の矢印に対応する．以降，space，time それぞれの方向について (u_1, w, c_i, c_j) を求める手順を説明する．

2.2.1 依存性抽出

反復ベクトル $(i, j)^T$ に対し，次の 3 組の依存性及び依存性多面体を抽出することができる．

依存性 1 flow （RAW）：$a[i', j'] \rightarrow a[i, j-1]$
$\mathcal{P}_{e_1} : i' = i, j' = j-1, 2 \leq j \leq N, 1 \leq i \leq N$

依存性 2 flow （RAW）：$a[i', j'] \rightarrow a[j, i]$
$\mathcal{P}_{e_2} : i' = j, j' = i, 2 \leq j \leq N, 1 \leq i \leq N, i - j \geq 1$

依存性 3 anti （WAR）：$a[j', i'] \rightarrow a[i, j]$
$\mathcal{P}_{e_3} : j' = i, i' = j, 2 \leq j \leq N, 1 \leq i \leq N, i - j \geq 1$

依存性 1 の依存性は，定数ベクトル $(0, 1)$ で表すことができる．このように，定数ベクトルで表せる依存性は一様であるという．一方，依存性 2（図 1 中の右下から左上への矢印），依存性 3 は非一様な依存性である．

各依存性について，(u_1, w, c_i, c_j) が満たすべき制約条件は以下のとおりである．

依存性 1 についての制約 タイル化合法性制約（tiling le-

gality constraint）として，

$$(c_i, c_j)\begin{pmatrix} i \\ j \end{pmatrix} - (c_i, c_j)\begin{pmatrix} i' \\ j' \end{pmatrix} \geq 0 \quad \langle i, j, i', j' \rangle \in \mathcal{P}_{e_1}$$

すなわち $c_j \geq 0$ の制約を設ける．

また，一様依存性であるので，通信ボリューム束縛制約（volume bounding constraint）として，

$$(c_i, c_j)\begin{pmatrix} i \\ j \end{pmatrix} - (c_i, c_j)\begin{pmatrix} i' \\ j' \end{pmatrix} \leq w \quad \langle i, j, i', j' \rangle \in \mathcal{P}_{e_1}$$

すなわち $w - c_j \geq 0$ の制約を設ける．

依存性 2，3 についての制約　依存性 2 は非一様であるので，Farkas の補題を用いる必要がある．タイル化合法性制約は，

$$(c_i i + c_j j) - (c_i j + c_j i) \geq 0,$$
$$2 \leq j \leq N, 1 \leq i \leq N, i - j \geq 1$$

となる．これに Farkas の補題，Gauss の消去法，Fourier-Motzkin の消去法を適用すると $c_i - c_j \geq 0$ を得られる．

ボリューム束縛制約は，

$$u_1 N + w - (c_i j + c_j i - c_i i - c_j j) \geq 0, (i, j) \in \mathcal{P}_{e_2}$$

であり，同様にして

$$\begin{aligned} u_1 &\geq 0 \\ u_1 - c_i + c_j &\geq 0 \\ 3u_1 + w - c_i + c_j &\geq 0 \end{aligned}$$

を得られる．

また，対称性により，依存性 3 についての制約は依存性 2 と同様である．

ゼロ解の回避のための制約　ゼロ解を回避するため，

$$c_i + c_j \geq 1$$

の制約を設ける．

2.2.2 space 方向のベクトル

上記の制約に基づき，辞書式順序最小解[*1]

$$\text{minimize}_{\prec}(u_1, w, c_i, c_j)$$

を求めると，$(0, 1, 1, 1)$ を得られる．これはタイルの space 方向ベクトルが $(1, 1)$ であり，スレッド間で 1 本の通信が発生することを意味する．図 1 においては，右上方向の矢印にてベクトルが示されている．

[*1] 辞書式順序（lexicographic order）$\prec$ は，$\vec{x} \prec \vec{y} \equiv \exists p, 1 \leq p < n, (x_{(1\ldots p)} = y_{(1\ldots p)}) \wedge (x_{p+1} < y_{p+1})$ と定義される．

```
#define N 1024
#define ceild(n,d)  ceil(((double)(n))/((double)(d)))
#define floord(n,d) floor(((double)(n))/((double)(d)))
for (t1 = 0; t1 <= floord(3*N-3, 8); t1++){
  lbp = max(max(0,ceild(4*t1-N+1,4)),ceild(8*t1-N-6,16));
  ubp = min(floord(t1,2),floord(N-1,8));
#pragma omp parallel for \
  private(t2, t3, t4, lbt3, ubt3, lbt4, ubt4) \
  shared(a, t1, lbp, ubp)
  for (t2 = lbp; t2 <= ubp; t2++){
    lbt3 = max(8*t2,8*t1-8*t2-N+1);
    ubt3 = min(min(N-1,8*t2+7),8*t1-8*t2+6);
    for (t3 = lbt3; t3 <= ubt3; t3++){
      lbt4 = max(8*t1-8*t2,t3+1);
      ubt4 = min(8*t1-8*t2+7,t3+N-1);
      for (t4 = lbt4; t4 <= ubt4; t4++){
        a[t3][-t3+t4] = a[-t3+t4][t3] + a[t3][-t3+t4-1];
      }}}
```

図 2　PLUTO 0.9.0 により変換されたコード

2.2.3 time 方向のベクトル

space 方向のベクトルが $(1, 1)$ なので，$H_S = (1, 1)$ とおく．I を 2×2 の単位行列として，

$$H_S^{\perp} = I - H_S(H_S H_S^T)^{-1} H_S^T = \begin{pmatrix} \frac{1}{2} & \frac{-1}{2} \\ \frac{-1}{2} & \frac{1}{2} \end{pmatrix}$$

の 1 行目を正規化すると，$(1, -1)$ を得られる．2 行目は 1 行目の符号を逆転したものなので無視する．

制約に $c_i - c_j \geq 1$ を加えて，再び辞書式順序最小解を求めると $(1, 0, 1, 0)$ を得られる．これはタイルの time 方向ベクトルが $(1, 0)$ であることを意味する．図 1 においては，真上方向の矢印にてベクトルが示されている．

2.3 多面体最適化の適用

PLUTO アルゴリズムの実装は，フリーソフトウェアとして広く公開 [10] されている．PLUTO の実装バージョン 0.9.0 を用いると，図 1 の例題プログラムは図 2 のように変換される．ただし，一部の変数の名前は変更している．タイルサイズには 8×8 を指定している．反復変数 `t2` についてのループの並列性が，OpenMP ディレクティヴ `#pragma omp parallel for` により明示される．

OpenMP ディレクティヴを高位合成に適用する手法としては，文献 [11] にて提案されているものが存在する．しかし，以下の 2 つの理由により，PLUTO が生成する OpenMP ディレクティヴには適用できない．

(1) PLUTO が生成するコードでは，図 2 における並列化された反復変数 `t2` の下限 `lbp` は，`t1` の値に応じて変化する．しかしながら，文献 [11] による手法は，並列化された反復変数の下限が 0 以外の場合に対応していない．

(2) 反復回数 `ubp-lbp+1` も同様に変化するが，同手法は，反復回数がスレッド数より小さい場合に対応してい

```
#pragma omp parallel for private(i) shared(B_L, E_L)
/* NOTE: maybe (B_L > E_L). */
for (i = B_L; i <= E_L; i++) S(i);
```

図 3 OpenMP ディレクティヴにより並列性を示されたループ

ない．

3. スレッド構成手法

本章では，PLUTO によって得られる OpenMP ディレクティヴを高位合成用の記述に変換し，並列回路を生成する手法を提案する．提案手法は，2.3 節で示した問題点を解消できるようにスレッドを構成する．

OpenMP ディレクティヴ`#pragma omp parallel for`により並列性を明示されたループの各反復を論理スレッドと呼び，合成される回路における処理単位（PE; Processing Element）を物理スレッドと呼ぶことにする．提案手法は，いくつかの論理スレッドをチャンクとして「束ね」，物理スレッドに割り当てることを問題として扱う．

3.1 スレッド割当戦略

N_L 個の論理スレッドを N_P 個の物理スレッドに割り当てるための戦略として，次の 2 つを提案する．

F 戦略 $\lfloor x/y \rfloor$ の演算を 1 回使う．物理スレッドに割り当てる論理スレッド数 C は，$C = \lfloor N_L/N_P \rfloor$ により決定する．論理スレッドを割り付ける物理スレッド数 J は，$J = N_P$ となる．

C 戦略 $\lceil x/y \rceil$ の演算を 2 回使う．$C = \lceil N_L/N_P \rceil$，$J = \lceil N_L/C \rceil$ となる．

ただし，いずれの戦略でも，$N_L \leq N_P$ の場合は，$C = 1$，$J = N_L$ とする．図 3 のように，ループの反復変数 i が B_L から E_L までの値をとる場合，論理スレッドの個数は $N_L = E_L - B_L + 1$ である．

t 番目の物理スレッド $(0 \leq t < N_P)$ は，$t < J$ の場合は，反復変数 i を b_t から e_t まで変化させながら，割り当てられた論理スレッド $S(i)$ を実行する[*2]．ここで，

$$b_t = B_L + C \times t$$

$$e_t = \begin{cases} E_L & (t = J-1) \\ \max(b_t, b_t + C - 1) & (t < J-1) \end{cases}$$

である．なお，$t \geq J$ の場合は，何もせずに他の物理スレッドの完了を待つ．

3.2 各戦略の有用性

C 戦略は F 戦略よりも除算の回数は多いが，物理スレッドに割り当てる論理スレッド数が多くなるため，全論理スレッドを完了するための反復回数を小さく抑えられることがある点で有用である．例えば，7 論理スレッドを 4 物理スレッドで実行する場合の反復回数は，F 戦略では 4 回，C 戦略では 2 回となる．

F 戦略 $C = \lfloor 7/4 \rfloor = 1$, $J = 4$

- T_0: $S(0)$
- T_1: $S(1)$
- T_2: $S(2)$
- T_3: $S(3), S(4), S(5), S(6)$

C 戦略 $C = \lceil 7/4 \rceil = 2$, $J = \lceil 7/C \rceil = 4$

- T_0: $S(0), S(1)$
- T_1: $S(2), S(3)$
- T_2: $S(4), S(5)$
- T_3: $S(6)$

一方で，9 論理スレッドを 4 物理スレッドで実行する場合は，C 戦略での J が F 戦略よりも小さくなってしまうため，反復回数はいずれの戦略でも 3 回となる．この場合は，除算回数が少ない F 戦略が有用である．

F 戦略 $C = \lfloor 9/4 \rfloor = 2$, $J = 4$

- T_0: $S(0), S(1)$
- T_1: $S(2), S(3)$
- T_2: $S(4), S(5)$
- T_3: $S(6), S(7), S(8)$

C 戦略 $C = \lceil 9/4 \rceil = 3$, $J = \lceil 9/C \rceil = 3$

- T_0: $S(0), S(1), S(2)$
- T_1: $S(3), S(4), S(5)$
- T_2: $S(6), S(7), S(8)$
- T_3: NOP

4. バッファ構成手法

本章では，前章で提案した手法で生成される回路について，オンチップ RAM やレジスタを用いてバッファを構成し，オフチップ RAM へのアクセスを最適化する手法を提案する．PLUTO によって解析される依存性を考慮しつつ，バス幅が許す限りでオフチップ RAM 上の複数の配列要素を読み書きできるようにする．オフチップ RAM から読みこんだ内容はバッファに配置し，オフチップ RAM よりも少ないサイクル数でアクセスできるようにする．

例えば，図 1 の例題プログラムにおいて，オフチップ RAM のバス幅を 64 ビット，配列 `a[N][N]` の 1 要素の大きさを 8 ビットと仮定する．オフチップ RAM のバス幅を最大限に活かすには，`a[N][N]` を 1 要素ずつロード/ストアするのではなく，なるべく連続する 8 要素を一度に処理するようにしたい．一見して，`a[i][j-1]` については連続する 8 要素ずつの読み書きは容易であることがわかる．一方で，`a[j][i]` については，連続する 8 要素ずつ読み書きする手法は自明ではない．本章では，PLUTO により解析される依存性情報を用いて，`a[j][i]` についても連続する 8 要素ずつ読み書きする手法を提案する．

[*2] 物理スレッドについて局所的な変数 (t, b_t, e_t, i) を，小文字で記している．

Vol.2013-SLDM-160 No.21
Vol.2013-EMB-28 No.21
2013/3/13

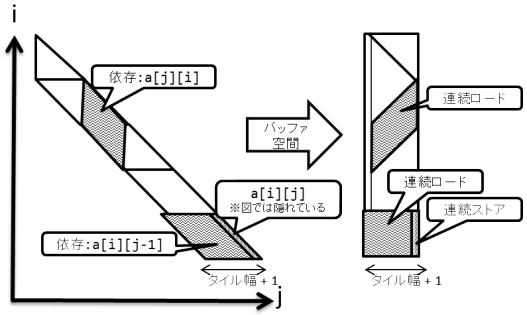

図 4 バッファ空間においてロード/ストアすべき部分

4.1 バッファ空間

図 4 の左側は，論理スレッド 1 個分の反復空間を表している．`a[i][j]` についての計算を行うために，`a[i][j-1]` と `a[j][i]` のタイルを読み込む必要がある．`a[i][j-1]` は space 方向にまたがった依存性であり，`a[j][i]` は非一様依存性である．この非一様依存性は，space 方向をまたぐことなく，論理スレッド 1 個の反復空間内に収まっている．

連続する 8 要素を一度にロード/ストアするため，論理スレッドの反復空間を図の右側のように「歪め」てバッファ空間を構成する．バッファは物理スレッド毎に 1 つ設ける．space 方向の距離 1 の一様依存性のため，バッファの横幅はタイルの幅よりも 1 大きい．提案する手法は，space 方向にまたがって非一様依存性が存在する場合については対応できていないが，そのような入れ子ループは稀であると考えられる．なお，space 方向にまたがる依存性が何ら存在しない場合は問題なく提案手法を適用出来る．

バッファ空間において，`a[i][j-1]` 相当部分は連続ロードを行い，`a[i][j]` 相当部分は連続ストアを行うことができる．`a[j][i]` 相当部分もある程度は連続ロードできる．

4.2 アドレス変換

反復変数が $(i,j) = (t_3, -t_3 + t_4)$ のとき，依存性のために (d_1, d_2) にアクセスする．ここで (d_1, d_2) は (j,i) または $(i, j-1)$ のいずれかである（図 5）．(d_1, d_2) のバッファ内における格納先のアドレスを求める．

(d_1, d_2) についての「タイル左端」の点 $(d_1, tlsd_2)$ を計算すると，$tlsd2 = -t_3 + lbt_4(d_1)$ である．ここで，$lbt_4(d_1)$ は，t_3 を d_1 としたときに定まる，t_4 の下限である．

(d_1, d_2) が「タイル左端」よりも「左」，すなわち $d_2 < tlsd_2$ ならば

$$(bd_1, bd_2) = (d_1, 0)$$

とし，それ以外なら

$$(bd_1, bd_2) = (d_1, d_2 - tlsd_2 + MARGIN)$$

とする．ここで $MARGIN$ は，space 方向の依存性のためにバッファの「左端」に設ける「余白」であり，例では 1 である．

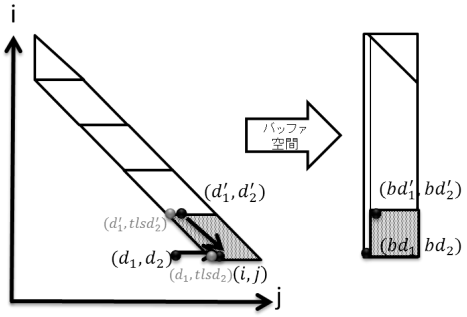

図 5 論理スレッドの反復空間からバッファ空間への変換

よって，(d_1, d_2) の内容をバッファ内に配置するための線形アドレスは，$bd_1 \times (TILEWIDTH + MARGIN) + bd_2$ となる．ここで，$TILEWIDTH$ はタイルの j 方向の幅である．

5. 評価

提案するスレッド構成手法およびバッファ構成手法を図 1 の例題プログラムに適用し，それらの有効性を評価した．評価における高位合成ツールとして，Mentor Graphics 社の Handel-C 5.1 を用いた．

オフチップ RAM の読み書き 1 回には，8 サイクルを要するものとした．バッファは 1 サイクルでアクセスできるオンチップ RAM で構成した．オフチップ RAM への書き込みが全て終了した後に計算結果のチェックサムをとることにより，提案手法適用後のコードに誤りがないことを確認した．配列の大きさは 1024×1024，タイルサイズは 8×8 とした．すなわち，物理スレッド 1 つあたりのバッファの大きさは，$1024 \times (8+1) = 9$K 要素である．

提案手法を適用し並列化した回路の実行サイクル数およびフリップフロップ（FF）数は，それぞれ表 1, 表 2 の通り見積もられる．ただし，表中におけるスレッド数が 1 の列については，提案手法適用前の回路の実行サイクル数や FF 数を記している．実行サイクル数は入れ子ループ部分についてのみのものであるが，FF 数は前処理及び後処理を含む回路全体についてのものである．

バッファ無しの回路では，C 戦略を適用した 2 スレッドの場合で最も実行サイクル数が小さくなり，提案手法適用前の回路より 1.15 倍の性能向上が得られた．しかしながら，これ以上スレッド数を増やしても性能は向上せず，頭打ちになってしまう．

一方，バッファ有りの回路では，C 戦略を適用した 2 スレッドの場合で，提案手法適用前の回路より 1.61 倍，4 スレッドの場合で同 2.85 倍，8 スレッドの場合で同 3.16 倍の性能向上が得られた．提案したバッファ機構を用いることで，より効果的に並列化を実現できることがわかった．

また，バッファ無しの回路では，F 戦略と C 戦略とでサイクル数に有意な違いは無いが，バッファ有りの回路では，

表 1 例題プログラムの合成回路の入れ子ループの処理にかかる実行サイクル数

物理スレッド数	1	2	4	8
バッファ無し (F 戦略)	(77,522,948)	67,302,548	67,568,752	68,049,908
バッファ無し (C 戦略)		67,299,032	67,648,690	68,113,172
バッファ有り (F 戦略)	-	48,174,684	28,466,662	28,172,574
バッファ有り (C 戦略)	-	48,171,344	27,245,394	24,555,284

表 2 例題プログラムの合成回路の入れ子ループ部分を含む FF 数

物理スレッド数	1	2	4	8
バッファ無し (F 戦略)	(1,582)	4,176	6,858	12,222
バッファ無し (C 戦略)		4,207	6,889	12,253
バッファ有り (F 戦略)	-	6,136	10,778	20,062
バッファ有り (C 戦略)	-	6,169	10,811	20,095

物理スレッド数を増やすにつれて C 戦略の方が実行サイクル数が小さくなる．バッファ無しの回路では，オフチップ RAM へのアクセスの衝突が多発するため，C 戦略による反復回数削減の効果が現れにくいものと考えられる．

6. おわりに

本稿では，PLUTO により出力される OpenMP ディレクティヴを含むプログラムを高位合成向けに変換し，並列回路を構成する手法を提案した．さらに，PLUTO により解析される依存性情報を考慮しつつ，バッファを用いてオフチップ RAM のアクセスを最適化する手法を提案した．提案手法を用いることにより，提案手法適用前の回路より 3 倍以上の性能向上を達成した．我々の知る限り，本研究は多面体最適化の高位合成への適用について初めて非一様依存性を扱うものである．

今後の課題としては，より実用的なアプリケーション向けに提案手法を拡張し，評価することが挙げられる．非一様依存性を持つアプリケーションとしては，Cholesky 分解や LU 分解等を挙げることができる．また，文献 [12] によれば SPECfp95 ベンチマークの入れ子ループの 46%以上にも，非一様依存性が含まれているとのことである．

その他，データ再利用性 (data reuse) が存在する場合にはバッファ内容を再利用する手法を考案することや，提案手法を自動的に適用するツールを実装することも視野に入れている．

謝辞 本研究は東京大学大規模集積システム設計教育研究センターを通し，メンター株式会社の協力で行われたものである．

参考文献

[1] Bondhugula, U., Ramanujam, J. and Sadayappan, P.: Automatic Mapping of Nested Loops to FPGAs, *PPoPP '07* (2007).
[2] Wu, G., Dou, Y. and Wang, M.: Automatic Synthesis of Processor Arrays with Local Memories on FPGAs, *FPT '10* (2010).
[3] Bondhugula, U., Hartono, A., Ramanujam, J. and Sadayappan, P.: A Practical Automatic Polyhedral Parallelizer and Locality Optimizer, *PLDI '08* (2008).
[4] Bondhugula, U. K. R.: Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model, PhD Thesis, Ohio State University (2008).
[5] Quillere, F., Rajopadhye, S. and Wilde, D.: Generation of Efficient Nested Loops from Polyhedra, *International Journal of Parallel Programming*, Vol. 28 (2000).
[6] Bastoul, C.: Code Generation in the Polyhedral Model Is Easier Than You Think, *PACT '13* (2004).
[7] Pop, S., Cohen, A., Bastoul, C., Girbal, S., andre' Silber, G. and Vasilache, N.: GRAPHITE: Polyhedral Analyses and Optimizations for GCC, *In Proceedings of the 2006 GCC Developers Summit* (2006).
[8] Grosser, T., Zheng, H., A, R., Simbürger, A., Grösslinger, A. and Pouchet, L.-N.: Polly - Polyhedral Optimization in LLVM, *First International Workshop on Polyhedral Compilation Techniques, IMPACT '11* (2011).
[9] Darte, A. and Vivien, F.: Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs, *PACT '96* (1996).
[10] Bondhugula, U.: PLUTO - An Automatic Parallelizer and Locality Optimizer for Multicores, (online), available from ⟨http://pluto-compiler.sf.net/⟩ (accessed 2013.02.13).
[11] Leow, Y., Ng, C. and Wong, W.: Generating Hardware from OpenMP Programs, *FPT '06* (2006).
[12] Yu, Y. and D'Hollander, E.: Non-uniform Dependences Partitioned by Recurrence Chains, *ICPP '04* (2004).