Vol.2013-SLDM-163 No.45
2013/11/29

高位合成における非一様依存性を持つ入れ子ループ向けのバッファ構成手法

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

概要：近年，非一様依存性と呼ばれる複雑なデータ依存性を持つ入れ子ループに対応した自動並列化手法として，多面体最適化が注目されている．多面体最適化は高位合成への適用も可能ではあるものの，PE 間でのオフチップ RAM へのアクセス衝突やオフチップ RAM のバンド幅の低さがボトルネックとなる．本研究では，PE 毎にオンチップバッファを設けることにより，多面体最適化を施した回路においてデータへのアクセスを高速化する手法を提案する．提案手法を適用することにより，PE 数 8 の構成において平均で 5 倍以上の高速化を達成した．

キーワード：高位合成，多面体最適化，OpenMP，メモリ階層

Buffer Construction Method for Nested Loops with Non-Uniform Dependencies in High-Level Synthesis

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

Abstract: Recently, polyhedral optimization has been focused as an automatic parallelization method for nested loops with non-uniform data dependencies. However, off-chip RAM accesses have been the bottleneck for applying polyhedral optimization into high-level synthesis due to their poor bandwidths and access conflicts between PEs. In this report, we propose a method to enable faster data accesses in polyhedral-optimized circuits by constructing on-chip buffers on each PE. The experimental result shows that the buffered circuits with 8 PEs are on average 5 times faster than the original ones.

Keywords: High-Level Synthesis, Polyhedral Optimization, OpenMP, Memory Hierarchy

1. はじめに

近年，入れ子ループ構造の自動並列化手法として多面体最適化（polyhedral optimization）[1], [2] が注目を集めている．多面体最適化は主にこれまでソフトウェアの分野におけるコンパイラ向けの技術として研究されてきたが，高位合成を用いることにより，ハードウェアの設計にも活用することができる．

本稿では，多面体最適化を高位合成に適用するにあたり，プロセッシングエレメント（PE）毎にバッファを設けることでオフチップ RAM へのアクセスを軽減させる手法を提案する．提案手法で構成するバッファ機構は，以下の利点を持つ．

RAM アクセス衝突の回避　オフチップ RAM には，そのポート数以上の PE は同時にアクセスできない．本稿で提案する手法では，PE 毎にバッファを 1 つずつ設けることにより，PE 間でのオフチップ RAM アクセス衝突の発生を軽減する．

バーストアクセス　配列要素の幅がオフチップ RAM のバス幅よりも小さい場合には，連続する複数の要素を 1 回のアクセスで読み書きすることができる．連続する要素をバッファに格納することで，このようなバーストアクセスが可能となる．

[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

```
/* 8-bit unsigned integer */
uint8_t a[N][N];
#pragma scop
for (i=0; i<N; i++)
  for (j=1; j<N; j++)
    a[i][j] = a[j][i] + a[i][j-1];
#pragma endscop
```

(a) pluto_template

```
/* 32-bit fixed-point num (16:16) */
fix16_t a[N][N];
for (k=0; k<N; k++) {
#pragma scop /* scop 1 */
  for (j=k+1; j<N; j++)
    a[k][j] /= a[k][k];
#pragma endscop
#pragma scop /* scop 2 */
  for(i=k+1; i<N; i++)
    for (j=k+1; j<N; j++)
      a[i][j] -= a[i][k] * a[k][j];
#pragma endscop
}
```

(b) lu

```
/* 32-bit fixed-point num (16:16) */
fix16_t a[N][N]
fix16_t b[N][N];
for (i=0; i<N; i++)
#pragma scop
  for (j=0; j<N; j++)
    for (k=i+1; k<N; k++) {
      if (k == i+1)
        b[j][i] /= a[i][i];
      b[j][k] -= a[i][k] * b[j][i];
    }
#pragma endscop
```

(c) strsm

図 1 入力コードの例

データ再利用 複数の反復にまたがってアクセスされる配列要素については，バッファに保持しておくことで再利用が可能となる．再利用を行うことで，オフチップ RAM への重複するアクセスを減らすことができる．

多面体最適化を高位合成の分野に適用した研究 [3], [4] は既にいくつか存在するが，これらの研究では非一様（non-uniform）依存性と呼ばれる複雑なデータ依存性に対応していない．本稿では，バッファ空間を一様部と非一様部とに分割することにより，非一様依存性を持つ入れ子ループにも対応する手法を提案する．

本稿の構成は以下のとおりである．まず第 2 章にて，実例を用いながら多面体最適化手法を紹介する．続いて第 3 章にて，提案するバッファ構成手法の概要について述べる．提案手法のコンパイル時のフローは第 4 章にて，実行時のフローは第 5 章にて述べる．第 6 章にて提案手法を評価したのち，第 7 章にて本研究の結論を述べる．

2. 多面体最適化

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．

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．多面体最適化アルゴリズムおよびそのコンパイラ実装としては，PLUTO[1], [2] が最もよく知られており，GCC[7] や clang[8] などの主要コンパイラにもその派生物が採用されている．

PLUTO は，SCoP （Static Control Parts）と呼ばれるループ記述を含む C ソースコードを入力として受け取り，OpenMP ディレクティヴ（`#pragma omp parallel for`）付きの並列ループ記述を含むソースコードを出力する．SCoP とは，ループ境界，分岐条件，配列の添え字が全てアフィン式で表される入れ子ループ記述のことである．図 1 (a), (b), (c) は SCoP を含む入力ソースコードの例である．

PLUTO を適用すると，元のループの反復空間は平行四辺形のタイルに分割され，論理的な OpenMP スレッドにタイル単位で割り付けられる．タイルの形状は，空間方向のベクトルと，時間方向のベクトルとを用いて表現される．空間方向のベクトルはタイルをスレッドに割り付けるために用いられ，時間方向のベクトルはスレッドに割り付けられたタイルの実行順序を示すために用いられる．

2.1 非一様依存性

図 1 (a) の `pluto_template`[2], [9] は，非一様依存性を含む入れ子ループカーネルの例である．`pluto_template` カーネルに PLUTO を適用すると，図 2 の通り空間方向ベクトルとして $(i, j) = (1, 1)$ が，時間方向ベクトルとして $(1, 0)$ が定められる．同様に，図 1 (b) の `lu` カーネルについては空間方向に $(i, j) = (1, 0)$，時間方向に $(0, 1)$ のベクトルが定められ，(c) の `strsm` カーネルについてはそれぞれ $(j, k) = (1, 0)$，$(0, 1)$ のベクトルが定められる．

また，紙面の都合のため詳述は避けるが，PLUTO は図 3 の通りタイル化された反復空間におけるデータ依存性の解析も行う．`pluto_template` カーネルの場合は空間方向に定数距離 $(1, 0)$ の一様依存性が存在し，時間方向に非一様依存性が存在している．一様依存性とは，依存関係にある配列要素の距離を $(1, 0)$ のように定数ベクトルとして表現できるデータ依存性のことである．一方，非一様依存性とはそのような定数ベクトルを用いて依存性を表現できないデータ依存性のことである．詳細については文献 [2] を参照されたい．

2.2 高位合成への適用

PLUTO によって生成される OpenMP 記述は，容易に高位合成向けの記述に変換することができる [10]．しかしながら，合成される回路の PE 間におけるオフチップ RAM アクセスの衝突のため，単に OpenMP 記述を変換したのみでは十分に並列化の恩恵を受けることができない．既存研究 [3], [4] は配列の要素をオンチップバッファに複製す

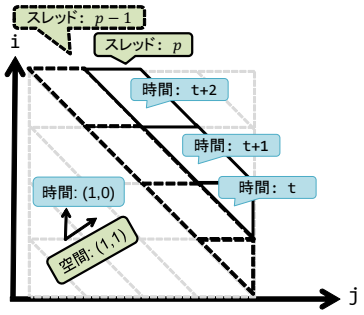

図 2 `pluto_template` カーネルのタイル化反復空間

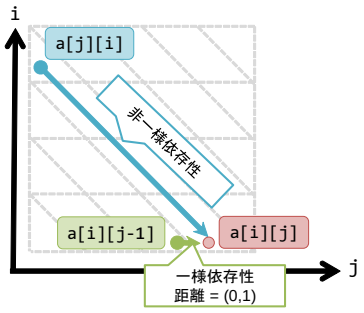

図 3 `pluto_template` カーネルのタイル化反復空間におけるデータ依存性

ることで性能向上を実現している．Wu らの研究 [3] は，PLUTO を高位合成に適用した初めてのものであるが，行列積カーネルおよび行列・ベクトル積カーネルを合成した場合の評価しか行われていない．また，Pouchet らの研究 [4] は実用的なカーネルを用いた評価を行っているが，`a[i][j]` と `a[i][j-1]` とのように添え字が連続する一様アクセスにしか対応していない．さらに，いずれの研究も，非一様依存性を考慮していない．

3. バッファ構成手法の概要

本研究では，非一様依存性を持つ入れ子ループに対応した高位合成回路の高速化手法を提案する．提案手法は，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．提案手法は，第 4 章にて述べるコンパイル時のフローと，第 5 章にて述べる実行時のフローとに分けられる．コンパイル時には，オフチップ RAM から取得した配列要素をバッファ内の何処に複製すべきかを示すバッファマップを作成する．実行時にはこのバッファマップを用いることで，データの整合性を保ちつつ，配列要素をバッファに読み込んだり，バッファの内容をオフチップ RAM に書き戻したりする．

提案手法は，以下の条件を満たすカーネルに対して適用可能である．

(1) 反復空間が 2 次元であること．反復空間が 3 次元以上のカーネルについては，内側の 2 次元を対象とすることができる．
(2) 配列の添え字が 2 次元であること．
(3) `a[b[i]]` のような間接アクセスを含まないこと．

先に見た図 1 (a)，(b)，(c) は提案手法が適用可能なカーネルの例である．なお，同図 (c) の `strsm` カーネルは条件分岐を含んでいるが，コンパイル時のフローにおいては，分岐条件が常に成立するとみなして提案手法を適用する．

4. コンパイル時のフロー

一様部，非一様部それぞれについての反復変数の値域をコンパイル時にキャプチャし，図 5 のように平行四辺形として表す．バッファマップは，この平行四辺形と同じ幅・同じ高さを持つ長方形（図 6）として表される．このバッファマップの作成には，SCoP のデータアクセス文の集合 S と，タイルサイズ (t_h, t_w) とを用いる．

バッファマップの作成フローは，図 4 の通り表すことができる．バッファマップ M は，一様部の集合 M_u と，非一様部の集合 M_n とから構成される．一様部 $m_u \in M_u$ はデータアクセス文毎に設けられ，その文の左辺に現れる配列要素書き込み式や，右辺に現れる一様な配列要素読み込み式を対象とする．一方，非一様部 $m_n \in M_n$ は各文の右辺に現れる，非一様な配列要素読み込み式毎に設けられる．

バッファマップの要素 $m \in \{M_u \cup M_n\}$ は，$LLCoord$，$RUCoord$，$Reusability$ の組として表現できる．ここで，$LLCoord$ および $RUCoord$ は，m のバッファマップ空間上での対応部分における左下座標および右上座標を表す．また，$Reusability = \{NotReusable|InnerReusable|OuterUsable\}$ は m の対応部分の再利用可能性を表現する．$InnerReusable$ は，2 次元 SCoP 内の最外反復変数の値が変化するまでの間，対応部分のバッファ内容を再利用できることを表す．一方，$OuterReusable$ は，SCoP の外側の反復変数の値が変化するまでの間，バッファ内容を再利用できることを表す．

例の `pluto_template`, `lu`, `strsm` それぞれのカーネルについてのバッファマップは式 (1)-(3) の通りとなる．図 6 -8 はこれらを視覚的に示したものである．

$$\begin{aligned} M_u &= \{\{(0,0),(t_h,t_w+8),NotReusable\}\} \\ M_n &= \{\{(t_h,0),(3t_h,t_w+8),NotReusable\}\} \end{aligned} \quad (1)$$

$$\begin{aligned} M_u &= \{\{(0,0),(t_h,t_w),NotReusable\}\} \\ M_n &= \{\{(t_h,0),(t_h+1,t_w),NotReusable\}, \\ &\quad \{(0,t_w),(t_h,t_w+2),InnerReusable\}\} \end{aligned} \quad (2)$$

$$
\begin{aligned}
M_u = \{&\{(0,0),(t_h,t_w),NotReusable\},\\
&\{(0,t_w),(t_h,t_w+2),InnerReusable\}\}\\
M_n = \{&\{(t_h,0),(t_h+1,t_w),NotReusable\},\\
&\{(0,t_w),(t_h,t_w+2),InnerReusable\},\\
&\{(t_h,t_w),(t_h+1,t_w+2),OuterReusable\}\}
\end{aligned}
\tag{3}
$$

なお，バッファマップ表現には書き込み可能か否かを表す要素は含まない．全ての一様部は書き込み可能とし，全ての非一様部は書き込み不可とする．これは，文の左辺に現れる配列要素書き込み式は，一様部の構成にのみ用いているためである．`strsm` カーネルのように一様部と非一様部とに重複要素が存在する場合への対応については第 5 章にて述べる．

4.1 一様部

バッファマップの一様部は，反復変数がタイル 1 つ分の空間を移動する際に，書き込みアクセスベクトルが動く値域と，一様な読み込みアクセスベクトルが動く値域との和集合として定義される．例えば `pluto_template` カーネルに対するバッファマップの一様部は，図 5 の通り，`a[i][j]` の値域と `a[i][j-1]` の値域との和集合として定義される．ただし，実際のバッファマップはオフチップ RAM のバス幅に切り上げた幅とする．オフチップ RAM のバス幅が 64 ビットの場合，`pluto_template` カーネルの配列要素幅は 8 ビットなので，バッファマップは 8 要素分に切り上げた幅とする．`lu` カーネルや `strsm` カーネルについては配列要素の幅が 32 ビットなので，バッファマップは 2 要素分に切り上げた幅とする．

4.2 非一様部

バッファマップの非一様部は，非一様な読み込みアクセスベクトルが動く値域を一様部の大きさに切り上げたものとして一般に定義される．この切り上げ制約により，例えば `pluto_template` カーネルのバッファマップについては，非一様部は元の読み込みアクセスベクトルの値域の約 2 倍の大きさになってしまうが，バス幅の大きさを利用したバーストアクセスが可能となる．

しかしながら，`lu` カーネルや `strsm` カーネルに対してこの一般の定義を適用すると，バッファマップの無駄な部分が大きくなりすぎてしまう (図 7(a))．したがって，このように 2 次元配列の添え字の一方が定数であり，もう一方が一様であるような特殊な非一様アクセスについては，別個の処理を行う．

例えば `lu` カーネルについては，`k` は 2 次元 SCoP 内においては定数として扱われるため，`a[i][k]` 及び `a[k][j]` がこれらの特殊なアクセスに該当する．図 7(b) は，`a[i][k]` 部分を一様部の `a[i][*]` 部分に連結し，`a[k][j]` 部分を `a[*][j]` 部分に連結することで，無駄を減らしたバッファ

入力: 文集合 S およびタイルサイズ (t_h, t_w)
OUTPUT: バッファマップ $M = \{M_u, M_n\}$

マップを初期化する M: $M_u = M_n = \phi$.
for all 文 $s \in S$ **do**
  s の左辺式 a_w に対する一様部 m_u を構成する:
    $m_u = \{LLCoord, RUCoord, Reusability\} = \{(0,0),(t_h,t_w),NotReusable\}$.
  for all s の右辺に現れる一様な式 a_{r_u} **do**
    距離ベクトル $d_u = |a_{r_u} - a_w|$ を用いて m_u を拡張する:
      $m_u = \{\ldots, RUCoord(m_u) + d_u, \ldots\}$.
  end for
  m_u をオフチップ RAM のバンド幅に合わせてアラインする.
  m_u を M_u に追加する: $M_u = M_u \cup m_u$.
  for all s の右辺に現れる非一様な式 a_{r_n} **do**
    if a_{r_n} の 1 個目の添え字が定数であり，もう一方が一様である場合 **then**
      一様な添え字に基づき，新しい一様部 m_u を構成する．m_n のアライメント前の大きさは $(1, t_w)$ とする:
        $m_n = \{MostLUCoord(M), MostRUCoord(M) + (1,0), NotReusable\}$.
    else if a_{r_n} の 1 個目の添え字が一様であり，もう一方が定数である場合 **then**
      一様な添え字に基づき，新しい一様部 m_u を構成する．m_n のアライメント前の大きさは $(t_h, 1)$ とする:
        $m_n = \{MostRLCoord(M), MostRUCoord(M) + (0,1), InnerReusable\}$.
    else if a_{r_n} の添え字が定数である場合 **then**
      大きさ $(1,1)$ の新しい非一様部 m_n を構成し，マップの余りの箇所に配置する:
        $m_n = \{\ldots, \ldots, OuterReusable\}$.
    else
      新しい非一様部 m_n を構成する．大きさは，非一様アクセスベクトルの値域を (t_h, t_w) にアラインしたものとする:
        $z_n = AlignUp(DomainOf(a_{r_n}), (t_h, t_w))$.
        $m_n = \{MostLUCoord(M), MostRUCoord(M) + z_n, NotReusable\}$.
    end if
    m_n をオフチップ RAM のバンド幅に合わせてアラインする.
    m_n を M_n に追加する: $M_n = M_n \cup m_n$.
  end for
end for

図 4 コンパイル時のフロー

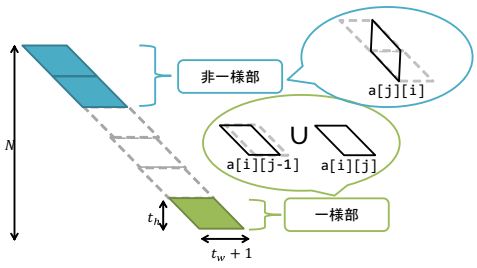

図 5 `pluto_template` カーネルの反復変数の値域

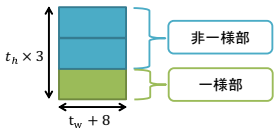

図 6 pluto_template カーネルに対するバッファマップ

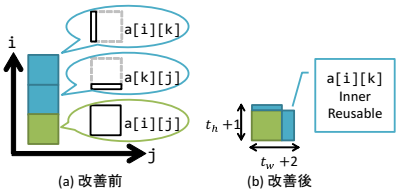

図 7 lu カーネルに対するバッファマップ

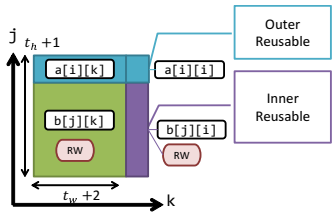

図 8 strsm カーネルに対するバッファマップ

マップを表す．

5. 実行時のフロー

図 9 は，各 PE の実行時のフローを表す．本章では，実行時に PE 間のデータの一貫性を保つ手法について説明する．

5.1 一様部と非一様部と間の一貫性

バッファマップの一様部と非一様部が，バッファチャンクを共有する場合が存在する．ここでのバッファチャンクとは，1 回のオフチップ RAM アクセスでアクセスできる連続した配列要素を意味する．PE はこれらの共有チャンクにアクセスする場合には，一様部のインスタンスを用いるものとする．これは，非一様部は読み込み専用として定義している一方で，一様部は書き込み可能として定義しているためである．

5.2 一様部間の一貫性

一様部が複数存在する場合にもデータ一貫性問題が存在する．この問題については，共有チャンクを更新するたびにオフチップ RAM へのライトスルーを行うことで対処する．従って，共有チャンクを持つ一様部の書き戻しについてはバーストアクセスは行わない．

6. 評価

提案手法を図 1 の例題カーネルに適用し，合成された回路の性能を評価した．いずれのカーネルにおいても配列サイズを表すパラメータ N は 256 に設定した．高位合

入力: バッファマップ $M = \{M_u, M_n\}$

M_n に基づき，外側ループで再利用可能な非一様部をバッファに読み込む．
for 1 個目の反復変数がタイル内を移動する間 **do**
  M_n に基づき，内側ループで再利用可能な非一様部をバッファに読み込む．
  for 2 個目の反復変数がタイル内を移動する間 **do**
    M_n に基づき，残りの非一様部をバッファに読み込む．
    M_u に基づき，一様部をバッファに読み込む．
    for all 文 s **do**
      for all s の右辺に現れる式 a_r **do**
        if アクセスする配列要素に対応するバッファチャンクが $m_u \in M_u$ と $m_n \in M_n$ とにまたがる場合 (5.1 節) **then**
          アクセスする配列要素の内容を，バッファの一様部からレジスタに取り込む．
        else
          アクセスする配列要素の内容をバッファからレジスタに取り込む．
        end if
      end for
      レジスタを用いて s のオペレータを実行し，結果をバッファに書き込む．
      if s の左辺式 a_w が，他の一様部にまたがる場合 (5.2 節) **then**
        更新したバッファチャンクをオフチップ RAM に書き込む．
      end if
    end for
  end for
end for
バッファの一様部の残りを書き戻す．
PE 間でのバリア同期を行う．

図 9 PE の実行時のフロー

成系ならびにシミュレーションツールとしては，Mentor Graphics 社の Handel-C 5.1 を用いた．シミュレーション環境においては，オフチップ RAM へのアクセスには 8 サイクル要するのに対し，オンチップバッファには 1 サイクルでアクセスできるものとした．

図 10 は，提案手法適用後の回路の，手法適用前の回路に対する速度向上（実行サイクル数の比）を表す．提案手法に依るバッファを用いた回路は，PE が 1 個の場合で平均 2.22 倍，PE が 8 個の場合で平均 5.21 倍の速度向上を達成している．一方で，バッファ無しの回路では，PE 数を増やした場合でも，オフチップ RAM アクセス衝突のために性能が向上していない．

PE 数を 8 に固定し，タイルサイズを変化させた場合の速度向上は，図 11 の通り見積もられる．タイルサイズが小さすぎる場合には，バッファが小さくなるために十分にバッファ機構の恩恵を受けられていない．また，タイルサイズが大きすぎる場合には，各 PE がオフチップ RAM へのアクセス権を取得するための待ち時間が増大するために，やはり十分な性能が得られていない．なお，タイルサイズ

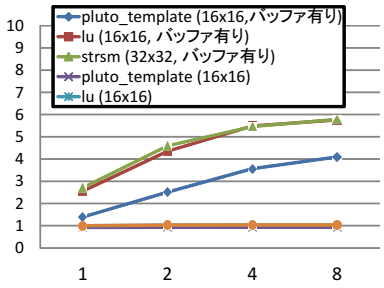

図 10 提案手法適用前の回路に対する速度向上（PE 数を変化させた場合）

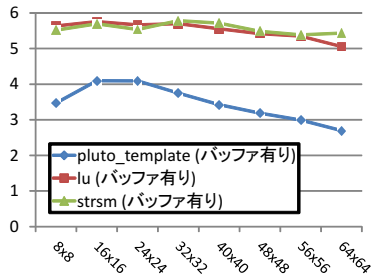

図 11 提案手法適用前の回路に対する速度向上（PE 数を 8 に固定し，タイルサイズを変化させた場合）

表 1 タイルサイズ (t_h, t_w) に対するバッファの大きさ (配列要素数)

カーネル	バッファの大きさ
pluto_template	$(3t_h) \times (t_w + 8)$
lu	$(t_h + 1) \times (t_w + 2)$
strsm	$(t_h + 1) \times (t_w + 2)$

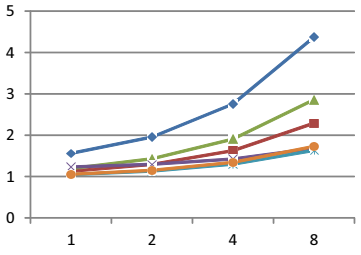

図 12 提案手法適用前の回路に対する NAND ゲート数の増加（PE 数を変化させた場合）

を変化させたときのバッファの大きさは表 1 の通り定められる．バッファの大きさはタイルサイズのみに依存し，配列サイズ N には依存しない．

また，回路の NAND ゲート数の増加は図 12 及び図 13 の通り見積もられる．ゲート数は PE 数に対しては概ね線形であり，タイルサイズに対してはほぼ一定である．

7. おわりに

本稿では，PE 毎にバッファを設けることにより，多面体最適化を適用した回路の性能を向上させる手法を提案した．実用的な算術演算を含む例題に提案手法を適用すると，平均で 5.21 倍の速度向上を達成できた．今後は，最適なタイルサイズを自動的に見積もる手法について研究を行う予定である．

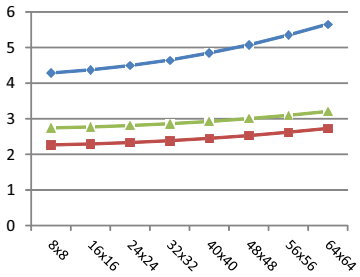

図 13 提案手法適用前の回路に対する NAND ゲート数の増加（PE 数を 8 に固定し，タイルサイズを変化させた場合）

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．本研究の一部は JSPS 科研費 10171422 の助成による．

参考文献

[1] Bondhugula, U., Hartono, A., Ramanujam, J. and Sadayappan, P.: A Practical Automatic Polyhedral Parallelizer and Locality Optimizer, *Proc. of PLDI* (2008).

[2] Bondhugula, U. K. R.: Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model, PhD Thesis, Ohio State University (2008).

[3] Wu, G., Dou, Y. and Wang, M.: Automatic Synthesis of Processor Arrays with Local Memories on FPGAs, *Proc. of the Int'l Conf. on FPT*, IEEE, pp. 249–252 (2010).

[4] Pouchet, L.-N., Zhang, P., Sadayappan, P. and Cong, J.: Polyhedral-based data reuse optimization for configurable computing, *Proc. of the Int'l Symp. on FPGAs*, ACM, pp. 29–38 (2013).

[5] Quilleré, F., Rajopadhye, S. and Wilde, D.: Generation of efficient nested loops from polyhedra, *Int'l Journal of Parallel Programming*, Vol. 28, No. 5, pp. 469–498 (2000).

[6] Bastoul, C.: Code generation in the polyhedral model is easier than you think, *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, pp. 7–16 (2004).

[7] Pop, S., Cohen, A., Bastoul, C., Girbal, S., Silber, G.-A. and Vasilache, N.: GRAPHITE: Polyhedral Analyses and Optimizations for GCC, *Proc. of the GCC Developers Summit* (2006).

[8] Grosser, T., Zheng, H., A, R., Simbürger, A., Grösslinger, A. and Pouchet, L.-N.: Polly - Polyhedral Optimization in LLVM, *Proc. of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)* (2011).

[9] Darte, A. and Vivien, F.: Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs, *Int'l Journal of Parallel Programming*, Vol. 25, No. 6, pp. 447–496 (1997).

[10] 須田瑛大, 高瀬英希, 高木一義, 高木直史: 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, 情報処理学会研究報告, Vol. 2013-SLDM-160, No. 21, pp. 1–6 (2013).