Vol.2013-ARC-203 No.15
2013/2/1

PEMAP: PErformance Estimatior for Many-core Processors

河合 夏輝[1,a)]　ユリ アーディラ[1,b)]　中村 孝史[1,c)]　田村 陽介[1,d)]

概要：多くのコアを持つ CPU や汎用計算向けの GPU 等，多数の計算ユニットにより並列計算を行うハードウェアが提案，提供されており，その性能を生かすため，ソフトウェアでの対応も求められている．しかし，既存の並列性を考慮していないソフトウェアを並列に動作するにはプログラムの多くの箇所を書き換える必要があり，多くのコストが生じる．従って，並列化を行うかどうかの判断には，並列化した際にどの程度の性能向上が見込めるかが重要である一方，並列ソフトウェアの性能にはプロセッサのアーキテクチャ等の様々な要因が大きな影響を与えるため，精度の高い予測は困難であった．これに対し，我々は，既存のシーケンシャルソフトウェアを並列化した際の性能を推定するツール PEMAP（Performance Estimator for MAny-core Processor）を提案する．本稿では，実用的な性能推定ツールである PEMAP の内部動作を詳細に示すことで，性能予測の自動化における知見を提供する．また，PEMAP が性能予測のために生成したダミー並列プログラムは，元のプログラムと同じ特性を持っていることを確認したため，この評価についても本稿で取り上げる．

1. はじめに

近年，多くのコアを搭載した CPU や，高い汎用計算性能を持つ GPU 等，多くの並列計算を行うハードウェアが用いられるようになっている．一方で，このようなハードウェアの性能を生かすには，並列に動作するよう設計されたプログラム（以降，並列プログラムと呼ぶ）を用意する必要があり，単独のプロセッサ等で実行することを前提に設計されたプログラム（以降，シーケンシャルプログラムと呼ぶ）では十分な性能を得ることができない．

従って，シーケンシャルプログラムを高速に実行する必要がある場合，事前に並列に動作するよう修正する必要がある．

そのため, 今日では, OpenMP[1], CUDA[2], OpenCL[3], OpenACC[4] 等の多くの並列プログラムを記述する言語（以降，並列言語と呼ぶ）やフレームワークが提案，活用されている．一方で，これらを用いてシーケンシャルプログラムを書き換える場合，プログラムの動作を理解した上で多くの部分を書き換える必要があり，多大なコストが必要となる．

1　株式会社フィックスターズ
Fixstars Corp., Shinagawa, Tokyo 141–0032, Japan
a)　kawai@fixstars.com
b)　y_ardila@fixstars.com
c)　nakamura@fixstars.com
d)　tamura@fixstars.com

また，自動でシーケンシャルプログラムを並列化する機能を持つコンパイラ等も提供されている．しかし，十分に高速な並列プログラムを得るには，プログラムをガイドライン等に沿うよう書き直す必要があり [7][8]，並列言語やフレームワークを用いる場合と同様にコストが必要となる．

一方で，並列プログラムの性能は，実行するプロセッサのアーキテクチャ等，様々な要因が大きな影響を与えるため，プログラムの処理内容によっては並列に動作するよう修正しても実行速度を向上させることができないことも考えられる。既存のシーケンシャルプログラムを並列化した際に，どの程度の効果があるのかを事前に把握することができれば，投資に見合うだけの高速化を実現できるか否かを判断するのに役に立てることができる。

そこで、我々は，既存のシーケンシャルプログラムを並列化した際の性能を自動で予測するツール PEMAP: Performance Estimator for MAny-core Processor を提案している．PEMAP のユーザは，C/C++で記述されたシーケンシャルプログラム中のループの前後にアノテーションを挿入した上でビルド・実行することで，PEMAP を起動する．PEMAP は，このループを解析して CUDA C で記述されたダミー並列プログラムを生成して実行する．このダミー並列プログラムは，アノテーションを付与したループを人の手で並列化した並列プログラムと同程度の性能を持つので，ダミー並列プログラムの実行時間をこのループを

ⓒ 2013 Information Processing Society of Japan

並列化した際の実行時間と同程度であると予測する．

PEMAP が生成したダミー並列プログラムは，入力となるシーケンシャルプログラムと異なる出力を生成する代わりに，計算内容と，メモリアクセスパターンと，条件分岐の分岐方向を再現する．CUDA ではメモリアクセス速度は，メモリトランザクションを特定の単位でまとめることの可否によって変わる．そして，この可否はメモリアクセスを行う際のワードサイズや，アクセスされるメモリアドレスによって決まる [2]．PEMAP は，シーケンシャルプログラムでのメモリアクセスパターンを CUDA C 上で再現することで，人の手で書かれた CUDA プログラムのメモリアクセス速度を再現する．

また，シーケンシャルプログラム中の分岐方向によって，メモリアクセスの有無や処理内容が変わることがある．PEMAP は，分岐方向も再現することで，これらの要因もダミー CUDA プログラムに反映される．

本稿では，PEMAP の内部動作を詳述することで，性能予測の自動化に関する知見を提供する．2 章では PEMAP の全体像を示し，3 章では PEMAP が内部で使用する中間表現（IR）とダミー並列プログラムについて詳細を述べる．そして，4 章では自動並列コンパイラ向けのベンチマークである BEMAP を用いた評価について取り上げる．最後に，5 章で関連研究を，6 章で今後の課題を示し，7 章にて本稿のまとめを述べる．

2. 概要

本章では，PEMAP の動作の概要を，その使用方法という観点から示す（**図 1**）．

PEMAP のユーザは，予測の対象とする C/C++ソースコード中の，並列化の対象となるループの前後にアノテーションを付与し（**図 2**）、x86 命令にコンパイルし、実行することで PEMAP を起動する．このアノテーションは，コンパイル時にプリプロセッサにより PEMAP 関数呼び出しと，最適化を回避するコードに展開され，実行時に，予測の対象ループが実行される直前に呼び出される．

呼び出された PEMAP 関数は，予測の対象ループを逆アセンブルした上で解析し，ループレベルコントロールフローグラフ（LCFG: Loop-level Control Flow Graph）を構築する．通常，コントロールフローグラフ（CFG）を用いる解析では，プログラムは基本ブロックを組み合わせた有向グラフで表現される（**図 3**）のに対し，LCFG を用いる解析では，プログラムは基本ブロックとシーケンスを組み合わせた無閉路有向グラフで表現される．そして，このシーケンスも同じ形式のグラフで表現される（**図 4**）．そして，LCFG の各シーケンスを解析し，読み出し，書き換えを行うメモリアドレスやレジスタを明確にする．なお，これらの解析手法については広く知られているため [5][6]，本稿では詳細には触れない．

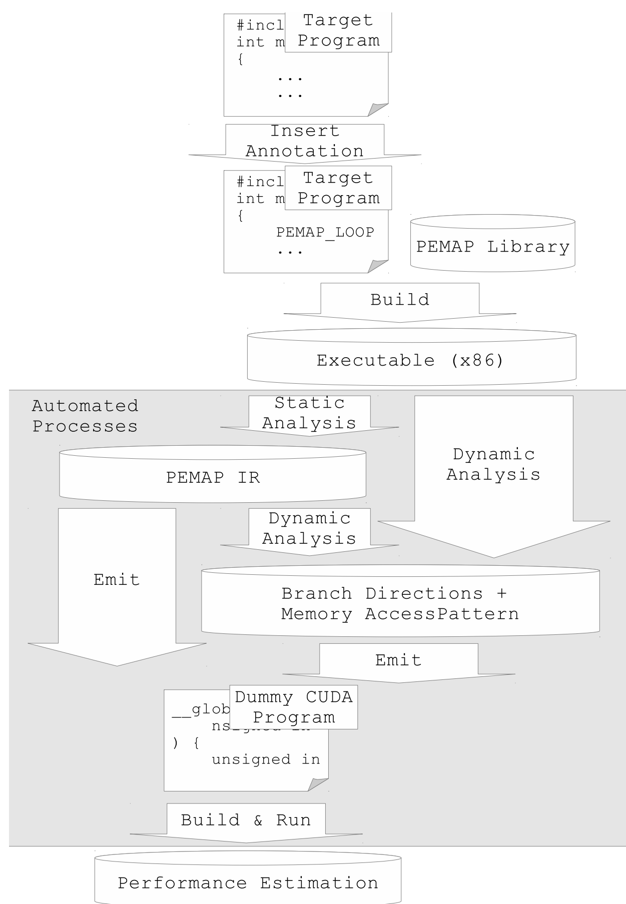

図 1 PEMAP の概要
Fig. 1 An overview of PEMAP.

```
1   #include <pemap.h>
2   void func(int *in, int *out)
3   {
4       PEMAP_LOOP_START; // A PEMAP Annotation
5       for (int i = 0; i < N; ++i) {
6           out[i] = in[i] + 1;
7       }
8       PEMAP_LOOP_END; // A PEMAP Annotation
9   }
```

図 2 アノテーションの例
Fig. 2 An example of PEMAP annotaions

解析後，PEMAP は，対象関数に含まれる機械語命令から，PEMAP が内部で用いる中間表現（Intermediate Representation, IR）を生成する．さらに，IR を解析し，その変換や最適化を行う．IR や、その解析、最適化手法の詳細は 3 章にて述べる．

IR への変換後，対象関数を，動的情報収集の機能を埋め込みながら別領域にコピーする．収集する情報は，条件付き分岐命令による分岐が行われたかどうかと，対象関数を実行する前にアドレスを確定できないメモリ読み出し・書き込みアクセス（以降，動的メモリアクセスと呼ぶ）において実際にアクセスしたアドレスと，対象関数に含まれるループが回った回数である．

条件付き分岐命令の結果とループ回数については、対象

ⓒ 2013 Information Processing Society of Japan

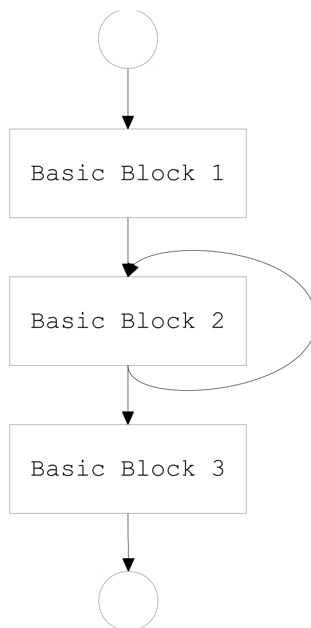

図 3 CFG の例
Fig. 3 An example of CFG.

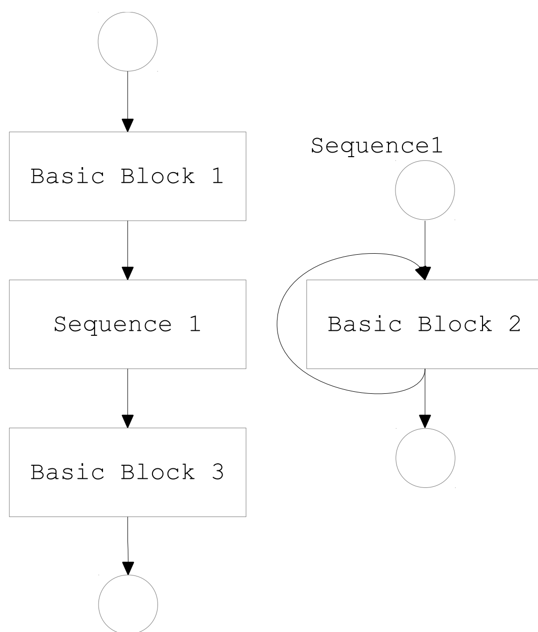

図 4 LCFG の例
Fig. 4 An example of LCFG.

関数が持つ基本ブロックの先頭に情報収集用関数呼び出しを埋め込み、その実行順序を解析することで取得することができる。また、メモリアクセス時のアドレスについては、メモリアクセスを行う命令の直前に収集用関数呼び出しを埋め込むことで取得することができる。

最後に，PEMAP は、IR からダミー並列プログラムを生成して実行することで、プログラムの性能予測を取得する．

3. PEMAP IR

2 章で述べた通り、PEMAP は，機械語で書かれた対象関数を、一度静的解析に向いた IR に変換した上で、CUDA C で書かれたダミー並列プログラムを出力する．また、IR の変換と最適化を行うことで，人の手で並列化したプログラムの性能に近づけている．

本章では，PEMAP が内部で用いる IR と，その変換と最適化について示す．また，IR からダミー並列プログラムへの変換についても示す．

3.1 PEMAP IR の要素

本節では，PEMAP IR を構成する要素を解説する。

最適化前の IR は，ある値を別の値に変換する IR 命令，対象関数が実行される前から定まっている値を表す IR 定数，値をメモリからロードする IR ロード，値をメモリに書き込む IR ストア，前のループ実行回に算出した値である IR ループキャリー，2 つ以上の値から 1 つを選択する IR ファイ関数と，IR ループキャリーが所属する IR ループから構成される．

また，IR の変換，最適化により，複数の値の加減算を表す IR ComAssoc，ループが反復される毎に一定数だけ増減する IR インダクション，複数の値から算出された 1 つの値を表す IR リダクションが生成される．

IR 命令は，3 番地コードの形式で表現されたもので，オペランドとして持つ 1 つか 2 つの値から算出される 1 つの値である．IR 命令は，対象関数の処理内容を再現する役割を持つ．

IR 定数は，対象関数内では変更されない値である．対象関数の外で変更される値であっても定数と呼ぶ点に注意が必要である．IR 定数は、自身がどこに由来するものなのか，つまり，即値として与えられたのかレジスタから渡されたのか等の情報を持つ．IR 定数は，IR 命令のオペランドや，メモリアクセス時のアドレスとして定数が使用されたことを明確にする役割を持つ．

IR ロードは，メモリから読み出した値であり，読み出し元のアドレス値を持っている．IR ロードは，メモリ読み出しのコストをダミー並列プログラム上で再現する役割を持つ．

IR ストアは，メモリに書き込むことを表しており，書き込み先のアドレス値と，書き込む値を持っている．IR ストアは，メモリ書き込みのコストを再現する役割を持つ。また、書き込む値を保持することで，ダミー並列プログラム上で計算する必要のある値を明確にする役割を持つ．

IR ファイ関数は，静的単一代入（Static Single Assignment, SSA）形式において用いられる概念 [9] に由来するもので，複数の値から選択された 1 つの値を表す。IR ファイ関数は、選択される複数の値と選択する基準となる IR

命令を持っている．条件分岐結果によって変わる必要のある値が変わるので，IR ファイ関数はこれを明確にする役割を持つ．

IR キャリーは，ループのある反復回で使用される，前の反復回で計算された値を表しており，初期値と，前の反復回での計算内容を示すバックエッジ値と，所属する IR シーケンスを持っている．IR キャリーは，ループにおけるデータの流れを IR 上で表現する役割を持つ．

IR シーケンスは，2 章で取り上げた LCFG を IR 上で表現するもので，内部に含んでいる IR シーケンスを持つ．IR シーケンスは，IR ストアや IR キャリーに関連した計算を行っている箇所を記録するために必要となる．

IR ComAssoc（交換法則と結合法則を満たす値: Commutative and Associative value）は，複数の値の加減算を表しており，加減算の対象となる値を持っている．IR ComAssoc の役割は IR 命令と同じだが、IR 命令よりも容易に解析を行うことができる．

IR インダクションは，ループが反復される毎に一定の値だけ増減する値を表しており，増分となる値を持つ．IR インダクションはメモリアクセスのアドレスとして頻繁に現れ，また CUDA C のスレッドインデックス等を用いて容易に表現できるため高速に算出することができるため，ダミー並列プログラムの性能を向上させ，人の手で並列化したプログラムの性能に近づける役割を果たす．

IR リダクションは，複数の値から算出された 1 つの値を表しており，初期値と，算出する元となる値を持つ．IR リダクションに対応する処理の例として，総和や総乗の計算や，最大値や最小値の探索等が挙げられる．

IR リダクションの処理は，シーケンシャルプログラムとは異なるアルゴリズムを用いることで，性能を向上させることができる．

この例を示すため，配列に格納された値の総和計算を取り上げる．シーケンシャルプログラムのループを用いる場合，結果を格納するメモリ領域を確保して，その値を初期化して，順次計算対象となる配列の内容を加算する．これと同等の計算を並列に行う場合，結果を格納するメモリを読みだし，加算を行い，結果を書きだす処理を排他的に行う必要がある（**図 5**）ため，複数のスレッドがメモリの 1 箇所へアクセスすることによるボトルネックや，排他処理にかかるコストが大きいというパフォーマンス上の問題が生じる．

これに対して，配列の隣り合う値を加算して，長さが半分の新たな配列を作り出す操作を，配列の要素数が 1 になるまで繰り返すことで，並列計算機において，総和を高速に計算することができる（**図 6**）．

3.2 PEMAP IR の生成

本節では，PEMAP IR を生成する手法について，その詳細を示す．また，生成した IR を変換，最適化して，ダミー並列プログラムの生成に用いることができるようにする手法についても示す．

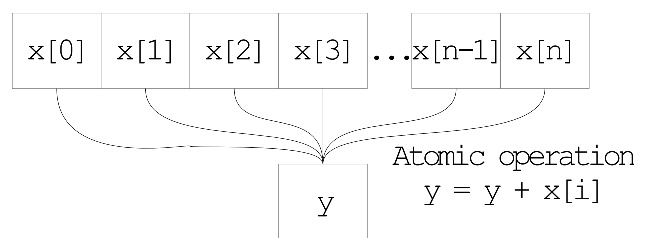

図 5 単純な総和計算
Fig. 5 Simple summation.

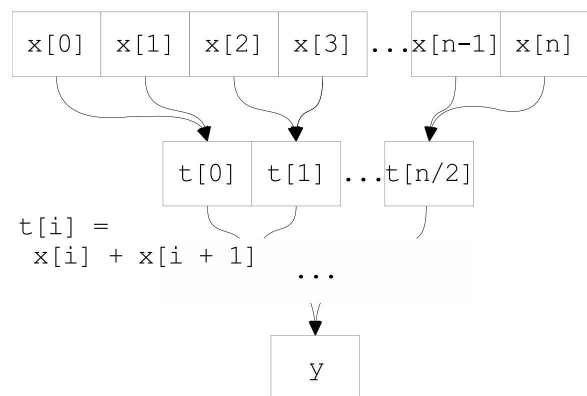

図 6 リダクションによる総和計算
Fig. 6 Summation by reduction.

PEMAP は，対象関数中の各機械語命令の依存関係，つまりある機械語命令が，オペランドとしてどの命令の結果を取っているかを解析した上で，IR 命令に置き換えることで IR を生成する（**図 7**）．対象関数が持つ機械語命令が，他の命令が上書きしていないレジスタの値を参照する場合，その値から IR の定数を生成する．また，命令が即値をオペランドとして持つ場合も定数を生成する．同様に，他の命令が上書きしていないメモリ上の値を参照する場合，そのアドレスから IR ロードを生成する．これとは逆に，ある機械語命令がメモリに値を書き込んで，その後に上書きされることがなければ，そのアドレスと書き込みを行った機械語命令から IR ストアを生成する．

ループ内で領域に格納した値を次のループ反復回で使用するケースが存在した場合，IR キャリーを生成する．IR キャリーの初期値には，そのループが実行される前に該当する領域に値を格納した機械語命令に対応する IR 命令や，該当する領域を示す IR 定数等を割り当てる．バックエッジは，ループ内の命令のうち，その領域に最後に値を格納した命令とする．（**図 8**）

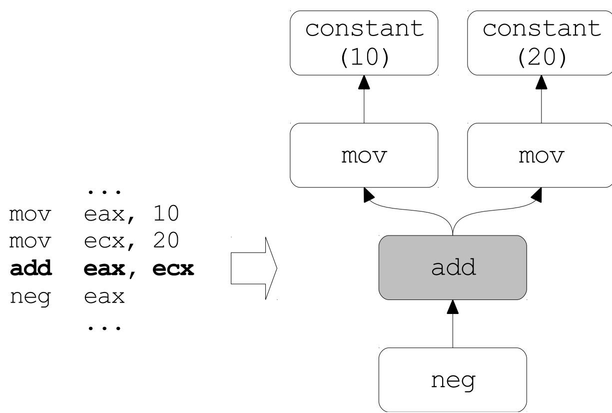

図 7　IR 命令の生成
Fig. 7　Generating an IR-instruction.

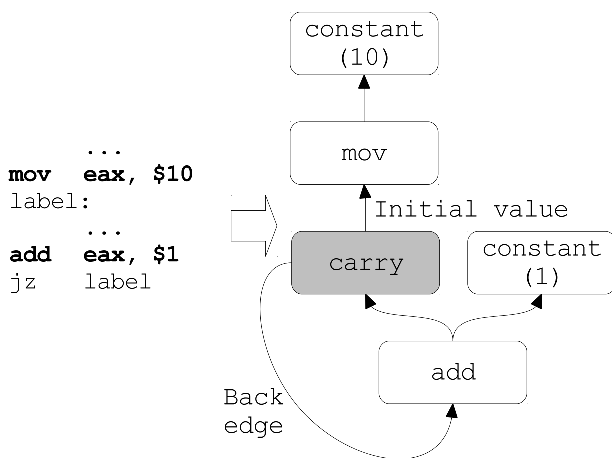

図 8　IR キャリーの生成
Fig. 8　Generating an IR-carry.

条件付き分岐を含むプログラムでは，ある命令のオペランドが複数の命令の結果のいずれかを取ることがある．このような場合，LCFG から支配木を生成して解析することで [5]，オペランドを複数の値から 1 つを生成するファイ関数を生成する．同時に，ファイ関数に，値を選択する条件として，機械語の条件付き分岐命令が参照するフラグを生成した命令に対応する IR 命令を付与する．

IR の変換・最適化で行う処理は，**表 1** の通りである．本稿では，これらの手法のうち，性能予測で重要となるインダクション検出とリダクション検出について詳細に述べる．

インダクション値は，ループの反復回が決まればその値も確定させることができるので，CUDA C におけるスレッドインデックス等を組み合わせて容易に表現できる．一般的に，インダクション値は配列アクセスのインデックスとして用いられることが多いので，ダミー並列プログラム上でメモリアクセスパターンを再現するために重要である．

3.1 節で述べた通り，キャリー値は，その初期値とバックエッジから構成される．インダクション検出では，キャリー値のバックエッジが，そのキャリー値と，キャリーが所属するループを実行する前に確定させることのできる値との加減算のみで表現可能である場合に，そのキャリー値をインダクション値と初期値との和に置き換える．

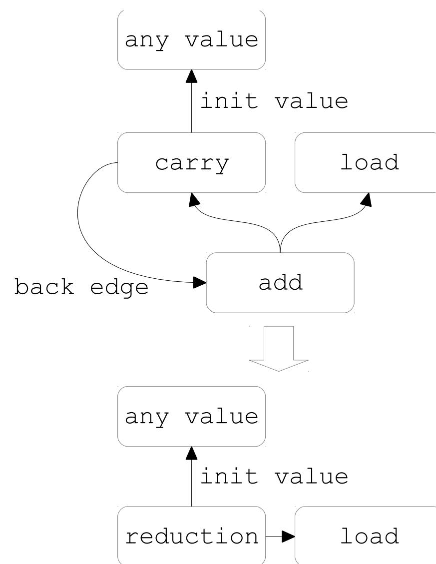

図 9　IR リダクションへの置換
Fig. 9　Replacing to an IR-reduction.

前述の通り，リダクションとは，複数の値から 1 つの値を導出する操作を指す．そして，並列計算機ではリダクション処理を工夫することで計算時間を短縮することが可能であることが，リダクション検出を行う理由である．

PEMAP の IR 生成部は，IR キャリーのバックエッジをたどり，総和や総乗といったリダクション処理のパターンと合致するかどうかを確認することで，リダクションの検出を行っている．例えば，総和の検出では，IR キャリーのバックエッジが，その IR キャリーと何らかの値の加算命令となっていれば，IR キャリーを IR リダクションに置き換える（**図 9**）．最大値や最小値の導出処理の検出では，IR キャリーのバックエッジが，その IR キャリーと何らかの値から選択するファイ関数になっており，選択条件がこれら二つの値の大小比較である場合に，IR キャリーを IR リダクションに置き換える．

3.3　PEMAP IR からダミー並列プログラムの生成

本節では，予測対象のシンプルなプログラム（**図 10**）から得た IR（**図 11**）をダミー並列プログラム（**図 12**）に変換する手順を具体的に示すことで，PEMAP IR からダミー並列プログラムを生成する手順を例示し，IR の各要素の役割を明確化する．

ダミー並列プログラムの生成は，最適化した IR に含まれる IR ストアをダミー並列プログラム中の CUDA カーネ

Vol.2013-ARC-203 No.15
2013/2/1

表 1 変換・最適化処理一覧
Table 1 A list of optimization processes.

変換・最適化処理	内容
冗長命令除去	mov 等の変換を行わない命令を除去する
リアソシエート	複数の加減算命令を 1 つの comassoc にまとめる
キャリー展開	展開可能な IR キャリーを展開する
絶対アドレスロード検出	IR メモリロードのうち常に一定のアドレスを持つものを定数扱いとする
ループ不変値検出	値として扱われる IR の要素ついて，どのループから見ると不変なのかを明確にする
インダクション検出	本文にて詳述
配列検出	IR メモリロード，IR メモリストアのうち，同一の配列にアクセスしているものと見なせるものを検出する
共通部分式除去	IR の要素のうち，同一のものとして扱うことのできるものを 1 つにまとめる
リダクション検出	本文にて詳述

```
1   static int in[1024], out[1024];
2   int main()
3   {
4     int i;
5     PEMAP_LOOP_START;
6     for (i = 0; i < 1024; ++i) {
7       out[i] = in[i] + 1;
8     }
9     PEMAP_LOOP_END;
10  }
```

図 10 予測対象プログラム
Fig. 10 The terget program to estimate.

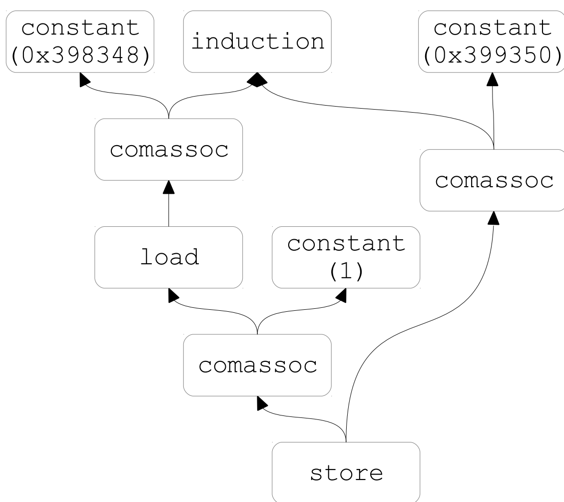

図 11 予測対象プログラム（IR）
Fig. 11 The terget program to estimate (IR).

```
1   #define LOOP_TIMES 1024
2   #define SEQ_COUNT_0 1024
3
4   __global__ void dummy_code(
5     unsigned int *array_0,
6     unsigned int *array_1
7   ) {
8     unsigned int t = (offset + blockIdx.x) * blockDim.x + threadIdx.x;
9     if (t < LOOP_TIMES) {
10      unsigned int t0 = t % SEQ_COUNT_0;
11      unsigned int v0 = 4;
12      signed int v1 = t0 * v0;
13      unsigned int v2 = 0;
14      signed int v3 = +v1+v2+0;
15      unsigned int v4 = 0;
16      signed int v5 = +v1+v4+0;
17      unsigned int v6 = array_0[v5 / 4];
18      unsigned int v7 = 1;
19      signed int v8 = +v6+v7+0;
20      array_1[v3 / 4] = v8;
21    }
22  }
```

図 12 ダミー並列プログラム
Fig. 12 The parallelized dummy program.

ル（以降，ダミーカーネルと呼ぶ）として出力する．その過程で，IR をグラフと見なし，IR ストアが持っているアドレスと値を深さ優先探索しながら出力することで，プログラム全体の計算を再現する．

まず，対象関数を実際に動作させて取得したループ回数を出力する．例では，1 行目の LOOP_TIMES マクロがループ回数である．

次に，カーネル関数内に，ループ回数分だけ実行されるブロックを生成する．例では，6 行目で変数 t にインデックス値を格納し，7 行目で，実際のループ回数よりも小さいインデックスを持つ場合のみ実行される if 文を提供している．また，このブロック内で，各ループのインデックスを再現する．例では，ループは 1 つしか存在しないため，変数 t0 だけを生成している．

ブロックの出力後，IR 内の全ての IR ストアを，配列への書き込みとして再現する．同時に，IR ストアが持っているアドレス値と書き込み値を同じブロック内に出力する．例では，20 行目でメモリ書き込みが生成されており，書き込み値として 19 行目で定義された IR ComAssoc が使用されている．さらに，PEMAP はこの IR ComAssoc をたどり，定数（18 行目）とメモリロード（17 行目）を生成し，メモリロードのアドレスも同様に生成（16 行目）している．この時，IR では大きな定数（0x398348）が存在しているが，これは配列が置かれているアドレスであると考えられるので，0 として扱う（15 行目）．また，アドレスをたどると IR インダクションが現れる．IR インダクションは，インデックス値に定数を乗じたものとみなすこともできるので，12 行目のように出力されている．なお，乗数である整数 4 は，整数のサイズ 4 バイトを表している．

さらに，同様の手順で IR ストアのアドレスを生成することで（13，14 行目），ダミー並列プログラムが完成する．

4. 評価

本稿では，自動並列コンパイラの性能測定のためのベンチマークスイートである BEMAP[10] から，Gray Scale ベンチマークを選択して，PEMAP の評価を行った．

表 2 評価環境
Table 2 Environment.

開発環境	Visual Studio 2010 Professional
最適化オプション	/O1

```
1    #define STACK_ARG_0 1152
2    #define STACK_ARG_1 864
3    #define ABS_LOAD_0 0.114478
4    #define ABS_LOAD_1 0.298912
5    #define ABS_LOAD_2 0.586611
6    #define SEQ_COUNT_0 1152
7    #define SEQ_COUNT_1 864
8
9    __global__ void dummy_code(
10     unsigned char *array_0,
11     unsigned char *array_1,
12     unsigned char *array_2,
13     unsigned char *array_3
14   ) {
15     volatile __shared__ int dummy;
16     unsigned int t = (offset + blockIdx.x) * blockDim.x + threadIdx.x;
17     if (t < LOOP_TIMES) {
18       unsigned int t0 = t % SEQ_COUNT_0;
19       unsigned int t1 = (t / SEQ_COUNT_0) % SEQ_COUNT_1;
20       signed int v1 = t0 * 1;
21       signed int v2 = t1 * STACK_ARG_0;
22       signed int v3 = v1 + v2;
23       unsigned char v4 = array_0[v3];
24       signed int v5 = (signed int)v4;
25       float v6 = v5 * ABS_LOAD_0;
26       unsigned char v7 = array_1[v3];
27       signed int v8 = (signed int)v7;
28       float v9 = (float)v8 * ABS_LOAD_1;
29       float v10 = v6 + v9;
30       unsigned char v11 = array_2[v3];
31       signed int v12 = (signed int)v11;
32       float v13 = (float)v12 * ABS_LOAD_2;
33       float v14 = v10 + v13;
34       unsigned int v15 = (unsigned int)v14;
35       array_3[v3] = v15;
36     }
37   }
```

図 13 An example of a kernel of dummy GPU codes

Gray Scale ベンチマークは，入力された RGB 画像全体のグレースケールの画像を生成するベンチマークであるが，PEMAP のメモリアクセスパターンの評価のため，入力された RGB 画像の一部のみを処理対象とするよう修正を加えた．また，BEMAP のベンチマークは OpenCL で記述されているため，これを CUDA に修正した．その他の環境は**表 2** の通りである．

Gray Scale ベンチマークに横幅 1152 ピクセル，縦幅 864 ピクセルの画像を入力した際に，PEMAP が生成したダミー並列プログラムのカーネルを**図 13** に示す．ただし，自動生成されたコードは可読性が低いため，プログラムの意味を壊さない範囲で修正を加えている．

この結果をみると，配列からのロードが，23 行目，26 行目，30 行目にて行われ，配列への書き込みが 35 行目にて行われている．そして，これらのインデックスは全て 20〜22 行目にて算出されている．そして，SEQ_COUNT_0 マクロに画像の横幅が格納されていることから，18，19 行目で算出されている t0，t1 は，シーケンシャルプログラムのインデックスに相当するものであることがわかる．さらに，STACK_ARG_0 にも画像の横幅が格納されていることを考えると，v1 は画像の横位置を指定するインデックス，v2 は縦位置を指定するインデックスであることが判明する．これらを踏まえると，このダミー並列プログラムは，画像のサイズだけ 3 つの配列に格納された値を順次読み出し，計算処理を行い，1 つの配列に書き出す動作をすることがわかる．

```
1    #define STACK_ARG_0 1152
2    #define STACK_ARG_1 864
3    #define ABS_LOAD_0 0.114478
4    #define ABS_LOAD_1 0.298912
5    #define ABS_LOAD_2 0.586611
6    #define SEQ_COUNT_0 576
7    #define SEQ_COUNT_1 432
```

図 14 An example of a kernel of dummy GPU codes

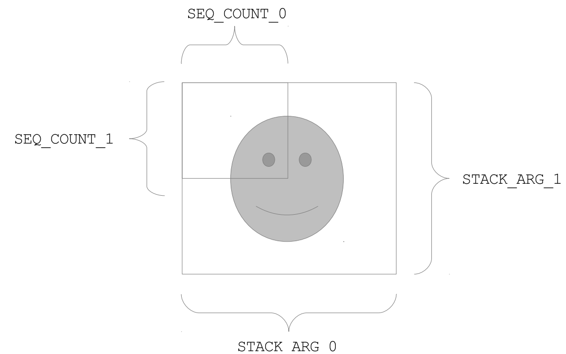

図 15 PEMAP メモリアクセス範囲
Fig. 15 The memory space where the dummy program accessed.

さらに，Gray Scale ベンチマークは常に入力された画像全体を変換する．これに修正を加え，画像の左上 1/4 だけを対象とするよう修正を加えてダミー並列プログラムを生成した結果を，**図 14** に示す．なお，カーネル部分については図 13 と全く同じである．

この結果をみると，SEQ_COUNT_から始まるマクロの値だけ，修正前の半分になっていることが分かる．これらのマクロは，シーケンシャルプログラムのインデックスに相当する値の範囲を決めていることを踏まえると，修正後のダミー並列プログラムでも画像の左上 1/4 相当の領域へのアクセスを再現できていることがわかる**図 15**.

5. 関連研究

GPU 上で動作するプログラムの性能を，実際のハードウェアで動作させることなく予測するモデル [11] や，このモデルを用いて，既存のシーケンシャルプログラムの性能をハードウェア上で実行させることなく予測するためのフレームワーク [12] が提案されている．このフレームワークでは，ユーザは予測の対象となるプログラムから，事前に "Code Skeleton" と呼ばれるものを記述することで，そのプログラムを並列化した際のパフォーマンス予測を得ることができる．

これに対し，PEMAP は，既存のシーケンシャルプログラムからダミー並列プログラムを生成して，実際にハードウェア上で動作させることで性能の予測を行う．そのため，PEMAP のユーザは，対象となる GPU の詳しい仕様

ⓒ 2013 Information Processing Society of Japan

を把握することなく，性能予想を行うことができる．また，PEMAP がユーザに求める操作は，シーケンシャルプログラム中のループにアノテーションを付与することである．従って，ユーザはアノテーションを付与するループとして，性能のボトルネックとなっているものを選択するだけで性能予測することが可能である．性能のボトルネックは既存のプロファイラを用いることで容易に検出することができるので，PEMAP を用いることで、性能予測をより容易に行うことができるといえる．

6. 今後の課題

2 章で述べた通り，PEMAP は，入力プログラムをループ単位で解析してダミー並列プログラムを生成する．従って，1 つのループ内に複数のループが含まれているような複雑な構成をもつ入力に対しては，正しい CUDA カーネルを出力することができない．これに対して，複雑なループをもつ入力プログラムに対しても性能予測手段を提供することが今後と課題である．考えられる手法としては，ループ毎に CUDA カーネルを出力する等が挙げられる．

5 章でプログラムの性能を，実際のハードウェアで動作させることなく予測するモデル [11] 取り上げた．PEMAP のシーケンシャルプログラムの解析により，このモデルでの性能予測に必要なパラメタを確定，もしくは推定することができれば，実際のハードウェアなしで性能の自動予測を実現することができる．このような PEMAP の応用を検討することが今後の課題である．

7. おわりに

本稿では，既存のシーケンシャルプログラムを人の手で並列化した際の性能を予測するツールである PEMAP を紹介し，その内部設計について詳しく述べることで自動での性能予測を行う際に必要となる手法を明確にした．

PEMAP は，予測の対象となるループに対してアノテーションを付与したシーケンシャルプログラムを静的，動的な性質を解析し，人の手でプログラムと同程度の性能をもつダミー並列プログラムを生成することで予測を行うため，より少ない手間で，より多種のハードウェアに対応することができる．

そして，実際に自動並列コンパイラ BEMAP が持つベンチマークを解析し，シーケンシャルプログラムのメモリアクセスパターンを再現することが可能であることを示した．

本稿で詳述した PEMAP を用いることで，並列化により高速化可能であるか否かの判断が難しい分野においても，GPU による高速化を提案することが容易になることを期待することができる．

謝辞　本研究は，一部，独立行政法人 新エネルギー・産業技術総合開発機構 (NEDO) の支援による成果を含んでいます．

参考文献

[1] OpenMP Architecture Review Board: OpenMP Application Program Interface (2011)
[2] NVIDIA Corporation: CUDA C PROGRAMING GUIDE, Version 5.0 (2012)
[3] Khronos Group: OpenCL - The open standard for parallel programming of heterogeneous systems(online), 入手先 ⟨http://www.khronos.org/⟩ (2012.12.20).
[4] OpenACC: OpenACC Directives for accelerations(online), 入手先 ⟨http://www.openacc.org⟩ (2012.12.20).
[5] A. W. Appel: Modern Compiler Implementation in ML, Cambridge University Press (1997)
[6] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman: Compilers: Principles, Techniques, and Tools Second Edition, Prentice Hall(2006)
[7] M. Mase, Y. Onozaki, K. Kimura, H. Kasahara: *Parallelizable C and Its Performance on Low Power High Performance Multicore Processors*, CPC (2010).
[8] Intel Corporation: Intel Guide for Developing Multithreaded Application (2011)
[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck: *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, TOPLAS (1991)
[10] Fixstars Corporation: BEMAP: BEnchmarks for Automatic Parallelizer. 入手先 ⟨https://sourceforge.net/projects/bemap/⟩
[11] S. Hong, H. Kim: *An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness*, ISCA (2009).
[12] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, T. D. Uram: *GROPHECY: GPU performance projection from CPU code skeletons*, SC (2011).

ⓒ 2013 Information Processing Society of Japan