Vol.2019-ARC-240 No.25
Vol.2019-SLDM-191 No.25
Vol.2019-EMB-53 No.25
2019/2/28

マルチターゲット自動並列化コンパイラにおける アクセラレータコスト推定手法の検討

山本 一貴[1,a)] 藤田 一輝[1] 柏俣 智哉[2] 高橋 健[2] Boma A. Adhi[2] 北村 俊明[3] 川島 慧大[4]
納富 昭[4] 森 裕司[5] 木村 啓二[1] 笠原 博徳[1]

概要：アクセラレータを持つコンピュータシステムでプログラムを高速に実行する場合，プログラム中からアクセラレータ実行に適した箇所を適切に選択する必要がある．そのためには，着目しているプログラム部分がアクセラレータで処理可能かどうかのみならず，ホスト CPU で実行する場合に比べてアクセラレータで十分高速に実行可能であるかどうかの判定，すなわち着目部分の正確なコスト推定が重要である．本稿では，複数のアーキテクチャをターゲットとする OSCAR 自動並列化コンパイラにおいて，プログラム中のループのアクセラレータ実行コスト推定手法を提案する．本稿が対象とするコンパイルフローは，入力となる逐次 C プログラムから，アクセラレータ実行対象部抽出を含む自動並列化を行い並列化 C プログラムを生成する OSCAR コンパイラと，並列化されたプログラムから実行オブジェクトを生成する各ターゲット用のコンパイラから構成される．提案手法は，ターゲット用コンパイラで実施されるアクセラレータ用最適化を考慮したコスト推定を OSCAR コンパイラで行うことを特徴とする．各コアがベクトルアクセラレータ（VA）を持つ OSCAR ベクトルマルチコアをターゲットアーキテクチャとして，配列加算，行列積，畳み込み演算，コレスキー分解の 4 つのプログラムを用いて提案手法を評価した結果，最小 1.98%，最大 21.4%の精度で推定可能であることが確認できた．

1. はじめに

画像処理計算・科学技術計算の分野では，アプリケーションの高速化・低消費電力化の需要が増加している．例えば，自動車業界における自動運転車の実用化のためには，カメラから得られる画像情報を高速に解析することで，瞬時に運転動作を決定し，安全な運転を実現する必要がある [1]．また，車内に搭載するためには空冷であることが望ましく，省電力化による発熱の抑制が不可欠である．

一方，これらのアプリケーションはデータ並列性が高く，その高速化に GPU などの SIMD 型アクセラレータが広く使われている．これら既存のアクセラレータは，高い演算性能を実現できる一方，アクセラレータを使ってプログラムを実行するためには CUDA[2] 等の独自の言語やライブラリを使ってソースコードを記述する必要があり，開発コストが増大するという問題点がある．また GPU は，大規模なレジスタファイルへのアクセスや実行スレッドの選択に必要な回路が大きく，消費電力が増大するといった問題点もある [3]．

これらの問題点を解決するため，筆者らは自動並列化に加えてメモリ最適化や電力最適化を実現する OSCAR 自動並列化コンパイラ，及びベクトルアクセラレータを搭載した「OSCAR ベクトルマルチコアアーキテクチャ」を提案してきた [4]．OSCAR コンパイラはマルチターゲットのコンパイラであり，そのコンパイルフローは上記の並列化等の高度な最適化を行う OSCAR コンパイラ本体と，オブジェクトファイルを生成する gcc や llvm 等のターゲットアーキテクチャ用のコンパイラから構成される．また OCAR ベクトルマルチコア用には，OSCAR コンパイラは自動ベクトル化も適用する．これにより，OSCAR ベクトルマルチコアは用途に応じて任意のホスト CPU を汎用コアとして搭載可能である．さらに，逐次のソースコードから自動的にアクセラレータの実行部を決定し，ベクトルアクセラレータでプログラムを実行させることができる．

アクセラレータを持つアーキテクチャでアクセラレータを有効利用するためには，プログラムからアクセラレータ実行に適した箇所を適切に選択する必要がある．そのためには，まず着目しているカーネルループ等のプログラム部分がアクセラレータで実行可能かどうか判定する必要が

1 早稲田大学 基幹理工学部 情報理工学科
2 早稲田大学 基幹理工学研究科 情報理工・情報通信専攻
3 早稲田大学 アドバンストマルチコアプロセッサ研究所
4 オスカーテクノロジー株式会社
5 株式会社 NSITEXE
a) kyamamoto@kasahara.cs.waseda.ac.jp

© 2020 Information Processing Society of Japan

Vol.2019-ARC-240 No.25
Vol.2019-SLDM-191 No.25
Vol.2019-EMB-53 No.25
2019/2/28

ある．しかしながら，選択した部分がアクセラレータで実行可能であった場合であっても，アクセラレータのハードウェア利用効率が低く十分な速度向上を得るのが難しい場合がある．そのため，ホスト CPU に比べてアクセラレータで実行させた場合に十分な速度向上が見込めるカーネルループをコンパイラが選択し，適切なカーネルループのみをアクセラレータで実行させる必要がある．

OSCAR 自動並列化コンパイラでこの選択を行う方法として，対象のカーネルループをアクセラレータと CPU で実行させた場合の推定コストを元に，速度向上が見込めるかどうかを判定する手法があげられる [5]．コスト推定はコンパイラ内部でソースプログラムから得られた中間表現中の各演算処理を積み上げげることにより可能となる．しかしながら，前述したとおり OSCAR コンパイラで並列化・ベクトル化したプログラムは最終的にターゲットアーキテクチャ用コンパイラでコンパイルされオブジェクトファイルが生成される．ここで，ターゲットコンパイラで最適化が実施される場合，コンパイラが生成するターゲット用のアセンブリと，OSCAR コンパイラ内部で保持している中間表現が乖離してしまい，正確なコスト推定ができなくなる可能性がある．

そこで，本稿ではベクトルアクセラレータを対象に，ターゲットコンパイラで実施される最適化を考慮したコスト推定モデルを提案する．提案手法を，配列加算，行列積，畳み込み演算，及びコレスキー分解 [6] の各アプリケーションに適用し，このモデルを使って算出した実行サイクル数の推定値と実測値の比較を行った結果について報告する．さらに，算出されたコスト推定値がアクセラレータ実行カーネルループの選択に有用であるかどうかを評価した結果を報告する．

以下第 2 節では，評価対象の OSCAR ベクトルマルチコアアーキテクチャ及びベクトルアクセラレータの構成について，第 3 節ではベクトルアクセラレータで実行するプログラムのコンパイルフローについて示す．また，第 4 節では提案するベクトルアクセラレータ向けの実行サイクル数推定モデルについて，第 5 節ではモデルを使った評価結果について示す．最後に第 6 節ではまとめについて述べる．

2. OSCAR ベクトルマルチコアアーキテクチャの概要 [4]

本節では，筆者らが高性能・低消費電力化が期待できるアクセラレータとして提案している「OSCAR ベクトルマルチコアアーキテクチャ」を対象に，はじめにその全体構成について述べる．次に，本稿での実行サイクル数推定の対象部分であるベクトルアクセラレータの構成について述べる．

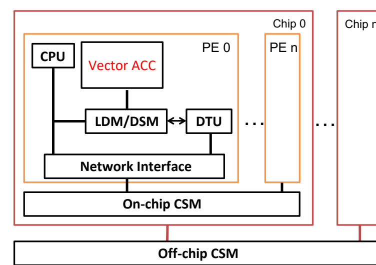

図 1　OSCAR ベクトルマルチコアアーキテクチャ [4]

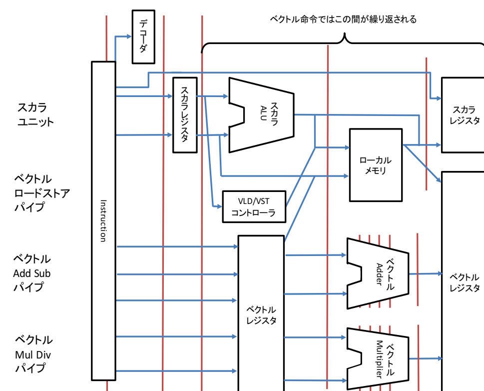

図 2　ベクトルアクセラレータ図

2.1　プロセッサ全体のアーキテクチャ

OSCAR ベクトルマルチコアアーキテクチャは，各プロセッサエレメント (PE) 内にベクトルアクセラレータ (VA) を搭載したアーキテクチャである．この OSCAR ベクトルマルチコアアーキテクチャを図 **1** に示す．

PE の外には，全ての PE からアクセス可能な集中共有メモリ (CSM) があり，容量が大きいため，プログラムに必要な全てのデータを格納することができる．各 PE には，CPU，VA の他に，ローカルメモリ (LDM)，分散共有メモリ (DSM)，データ転送ユニット (DTU) が配置されている．LDM は，自 PE 内からのみアクセスできる高速なメモリであり，各 PE ごとで実行時に使用するデータが格納される．DSM は，自 PE と他の PE からの同時アクセスが可能なメモリ領域であり，タスク間の同期フラグやデータ転送に必要なデータが格納される．DTU は，CPU や VA の動作とは独立に，CSM から LDM にデータ転送を直接行うことができる DMA コントローラである．VA は，自 PE の LDM/DSM にのみアクセス可能であることから，プログラムを実行させる際には実行開始前にあらかじめ DTU を使って CSM から LDM へデータを転送しておく必要がある．

2.2　ベクトルアクセラレータの構成

マルチコア中の各コアが持つベクトルアクセラレータ (VA) は，ベクトル演算によってデータ並列性の利用できるプログラムの高速化・低消費電力処理を目的としている．このベクトルアクセラレータを図 **2** に示す．

© 2020 Information Processing Society of Japan

VA にはベクトル演算器及びスカラ演算器が搭載されている．データレジスタはスカラ整数レジスタ (SR)，スカラ浮動小数点レジスタ (FR)，ベクトルレジスタ (VR)，マスクレジスタ (MR) で構成されている．現在はベクトル命令では，単精度浮動小数点のみ実装されており，8 個の加減算と 8 個の乗除算を同時に行うことが可能となっている．ベクトル演算のベクトル長は，プログラム中でベクトル長の設定命令を実行することで，任意の値にすることができる．また，ベクトル命令はチェイニングによってベクトル演算器間のパイプライン実行が可能となっている．このため，任意のプログラムに対して，ベクトル化による高い演算性能を実現することができる．

3. ベクトルアクセラレータ実行プログラムのコンパイルフロー

本節では，第 2 節で示した「OSCAR ベクトルマルチコアアーキテクチャ」上にあるベクトルアクセラレータを使ってプログラムを実行する際のコンパイルフローについて述べる．

3.1 コンパイルフロー

OSCAR ベクトルマルチコアを利用するためのコンパイラフレームワークを図 **3** に示す．フレームワークは，OSCAR 自動並列化コンパイラ，ホスト CPU 用ネイティブコンパイラ，及び VA 用ネイティブコンパイラとして使用する Clang/LLVM から構成される．OSCAR 自動並列化コンパイラは，逐次 C ソースコードを入力として，ホスト CPU 用並列化 C ソースコードと VA 用ベクトル化 C ソースコードをそれぞれ出力する．ベクトル用 C ソースコード中，ベクトル演算は組込（intrinsic）関数で表現される．

VA 用ベクトル化 C ソースコードは，3.3 節で述べるように VA 用コード生成を行うように拡張された Clang/LLVM を用いてコンパイルを行い，VA 用オブジェクトコードが出力される．ホスト CPU 用コードに VA 用オブジェクトコードを組み込み，GCC などのネイティブコンパイラを使ってコンパイルすることで，OSCAR ベクトルマルチコアで実行させる最終的な実行バイナリを生成する．

3.2 OSCAR 自動並列化コンパイラ

OSCAR 自動並列化コンパイラは，逐次の C ソースコードを入力として解析・コードリストラクチャリングを行った後，再び最適化された C ソースコードを出力する source-to-soruce コンパイラの形式をとっている．これは，マルチプラットホーム化を実現し，様々なハードウェアアーキテクチャ上で OSCAR 自動並列化コンパイラの最適化を適用できるようにするためである．

OSCAR 自動並列化コンパイラでは，まず並列化・ローカルメモリ使用のためのメモリ最適化のための解析とコードリストラクチャリングが行われる．また，ベクトルアクセラレータで実行させるカーネルループを決定するため，ループブロックのベクトル化解析が行われ，ベクトル化可能な場合は自動ベクトル化が行われる [4]．ベクトル化されたブロックを内側にもつループブロックは，VA 実行部のコードとして分離し，ホスト実行部と VA 実行部それぞれにのデータ転送や同期制御のコードを挿入する．これらの解析・コードリストラクチャリングを経て，ホスト実行部用 C ソースコード，及び VA 実行部用 C ソースコードを出力する．

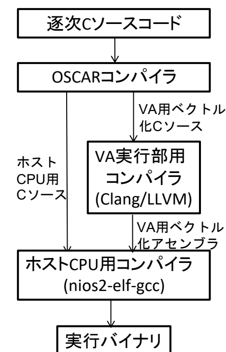

図 3 OSCAR ベクトルマルチコア向けコンパイルフロー

3.3 Clang/LLVM

VA のネイティブコンパイラとして，LLVM バックエンドに VA のターゲットを拡張した Clang/LLVM を使用する．Clang/LLVM では，OSCAR コンパイラによって自動ベクトル化されたベクトル化 C ソースコードを入力として，VA 向けオブジェクトコードを生成する．Clang/LLVM における VA 用ベクトル化 C ソースコードのコンパイル方法の概要を説明する．ベクトル化 C ソースコードを入力として，フロントエンドの Clang によって LLVM の中間表現となる LLVM-IR に変換される．LLVM-IR においては，ベクトル化 C ソースコードにおけるベクトル型の変数は VectorType として表現される．各ベクトル演算に関しては，基本演算かつマスク無しのベクトル演算の場合はベクトル型をオペランドにした命令として，また複雑な演算やマスク有りの演算の場合は Builtin 関数に対応した LLVM-IR Intrinsic 関数の呼び出しとして，それぞれ表現される．

4. 提案するベクトルアクセラレータ向け実行サイクル数推定モデル

本節では，カーネルループをベクトルアクセラレータで実行した場合にかかるサイクル数を，OSCAR コンパイラで推定するためのモデルについて述べる．

実行時間を推定するために必要な要素はハードウェア

© 2020 Information Processing Society of Japan

表 1 命令ごとのデータ依存により発生するペナルティ

演算形式	演算型	命令の種類	ペナルティ
スカラ	整数型	add, sub, shift	0
	整数型	mul	4
	整数型	div	33
	float	add, sub	4
	float	mul	3
	float	div	16
		load, store	2
vector	-	ベクトル長変更	4
	float	add, sub	6
	float	mul	5
	float	div	17
	float	load, store	2

アーキテクチャに基づいた命令実行形式とターゲットコンパイラで行われる最適化の 2 つに分類できる．4.1 節では，アクセラレータでの実行サイクル数を推定するにあたって必要なハードウェアの構成・実行モデルについて記す．4.2 節では，ターゲットコンパイラである LLVM/Clang における最適化，及びこのうち OSCAR コンパイラでサイクル数を推定するために考慮すべき最適化内容について述べる．そして 4.3 節では，OSCAR コンパイラ上で VA 実行サイクル数を推定できる提案モデルの詳細について述べる．

4.1 ベクトルアクセラレータのパイプライン構成・命令実行形式

図 2 にあるように，アクセラレータのパイプはスカラユニット・ベクトルロード/ストアパイプ・ベクトル Add Sub パイプ，及びベクトル Mul Div パイプの 4 つに分かれており，パイプ間の並列実行が可能となっている．

アクセラレータはパイプライン構成となっていることから，サイクルごとに命令をフェッチして実行を開始する．パイプラインハザード発生時には，数サイクル分のストールが発生する．よって，実行サイクル数を推定するためには，対象プログラムを実行した際に発生するストールのサイクル数を推定する必要がある．ストールが発生する要因は，命令間のデータ依存・各パイプの競合・分岐命令の 3 つに大きく分けることができる．このうち，VA で実行させる代表的な命令ごとのデータ依存によるストールサイクル数を表 1 に示す．ベクトル演算のストール数は，エレメント単位でのフォワーディング機能により，ベクトル演算の始めのエレメントが使用可能になるまでのサイクル数を表す．

パイプの競合によって発生するストールは，パイプで実行する命令がベクトル演算の場合，1 つの命令にかかるサイクル数が大きく，同じパイプを使用する次の命令の演算を開始できない場合に発生する．表 2 に，命令ごとのパイプを占有するサイクル数・及び分岐命令によって発生するレイテンシを示す．

表 2 命令ごとのパイプ使用サイクル数

演算形式	演算型	命令の種類	サイクル数
スカラ	整数型	add, sub, shift	1
	整数型	mul	5
	整数型	div	34
	float	add, sub	5
	float	mul	4
	float	div	16
	float	load, store	1
	-	条件分岐命令 (条件確定待ち含む)	7
vector	-	ベクトル長変更	4
	float	add, sub	(vector_length)/8 + 1
	float	mul	(vector_length)/8 + 5
	float	div	(vector_length)/8 + 17
	float	load, store	(vector_length)/8 + 1

4.2 ターゲットコンパイラ LLVM/Clang の最適化

第 3 節の図 3 のコンパイルフローにあるように，OSCAR コンパイラで出力される VA 実行部の C ソースコードは，Clang/LLVM を通すことで VA 用オブジェクトコードに変換される．このようなコンパイルフローをとっていることから，Clang/LLVM を通して得られる実行バイナリは，OSCAR コンパイラが出力する VA 向けソースコードと比べて最適化されたコードに変換されている場合がある．OSCAR コンパイラの段階で実行サイクル数を推定するためには，Clang/LLVM によってどのような最適化がカーネルループに対して適用されるかを，推定の段階であらかじめ考慮しておく必要がある．

今回のモデルでは，Clang/LLVM で行われる最適化のうち，実行時のサイクル数低減への効果が大きい (1)〜(3) の 3 つの最適化を考慮した．

(1) 2 の冪乗の乗算計算をシフト演算に変更
(2) a = b * (ループ誘導変数) → ループ内での演算を a = a + b に変換
(3) ループ回転数が小さい場合のアンローリング

VA で実行させるプログラム中で，(1)〜(3) の最適化が行われる部分を OSCAR コンパイラの段階で特定し，実行サイクル数を推定できるようにした．

4.3 VA 向け実行サイクル数推定モデル

OSCAR コンパイラでは，**図 4** のような逐次の C ソースコードを**図 5** のような VA 向けベクトル化 C ソースコードに変換する．コンパイラ内部では，このベクトル化 C ソースコードを表 3 のような情報として保持する．

提案する VA 実行部実行サイクル数推定手法では，この OSCAR コンパイラが保持する情報にある各演算を VA で実行させる命令として利用する．ループを含むプログラムを VA で実行させた際のサイクル数の推定値は，以下の (1)〜(6) の手順を行うことで取得する．

```
#define N 4096

for( i = 0 ; i < N ; i++){
   c[i] = a[i] + b[i];
}
```

図 4　逐次 C プログラム

```
__pvf vec_a, vec_b, vec_c;

_pt_vlvl(64);
for (i = 0 ; i <= 4095 ; i += 64){
   vec_a = _pt_vld_f(a + i);
   vec_b = _pt_vld_f(b + i);
   vec_c = _pt_vadd_f(vec_a, vec_b);
   _pt_vst_f(c+i, vec_c);
}
```

図 5　ベクトル化 C ソースコード

表 3　図 5 に対応する OSCAR コンパイラ中の情報

ブロック	演算型	演算	演算内容	ベクトル長
BASIC	-	vlvl	vlvl(64)	-
LOOP	int	mul	t1 = i*4	-
	int	add	A = a+t1	-
	vector	load	vec_a←Mem[A]	64
	int	mul	t2 = i*4	-
	int	add	B = b+t2	-
	vector	load	vec_b←Mem[B]	64
	vector	add	vec_c=vec_a+vec_b	64
	int	mul	t4 = i*4	-
	int	add	C = c+t4	-
	vector	store	vec_c→Mem[C]	64
	-	branch	分岐 (回数:64)	-

(1) ループブロック内のデータの中から，次に割り当てる演算を取得する．

(2) 各演算のパイプへの割り当て

 (a) 割り当て先のパイプが他の演算で占有されている場合
 その演算による占有が終了し，パイプが解放された後に割り当てる

 (b) 割り当て先以外のパイプにある前の演算とデータ依存が発生する場合
 表 1 から先行する演算のレイテンシを取得し，その分だけストールさせた後に割り当てる

(3) 割り当てた演算が対象パイプを占有するサイクルを表 2 の値を使用して設定する．スカラ乗算が 2 の冪乗を乗ずる場合，及びループ誘導変数との乗算を行う場合は，4.2 節にあるようにターゲットコンパイラである Clang/LLVM での最適化を考慮して，それぞれシフト演算・スカラ Add のサイクルを設定する．

(4) ループ内のデータの中でパイプに割り当てる演算が残っている場合は (1) からの手順を再び行う．全てのデータのパイプ割り当てが終了している場合は，最後

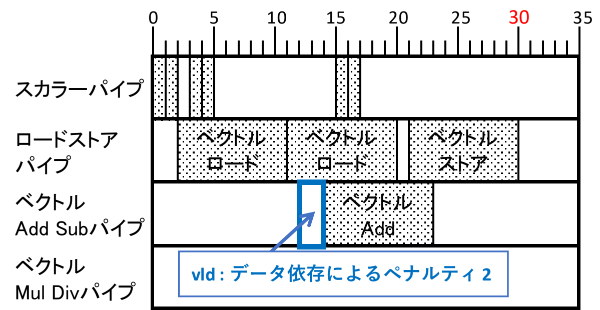

図 6　表 4 の各演算のパイプ割り当て結果

のデータのパイプの占有が終了するまでにかかるサイクル数を取得する．これをベクトル化されたプログラム部分の実行サイクル数の推定値とし，(5) の手順を行う．

(5) (4) で算出した値に表 2 にある分岐コストを加算し，ループ回転数を乗算することでループ全体のサイクル数を算出する．
 4.2 節にあるように，ループ回転数が小さく Clang/LLVM の最適化によりアンローリングされることが推測される場合は，この分岐コストを加えずに回転数のみを乗算する．

(6) ループの外側にある演算のコストを (5) で算出したコストに加算し，プログラム全体の推定サイクル数とする．

例えば，図 5 のようなベクトル化された配列加算プログラムの VA 実行サイクル数の推定は，OSCAR コンパイラ内部で保持している表 3 のデータを利用する．ループブロックの各演算を (1)〜(4) の手順を用いて，スカラー，ロードストア，ベクトル Add Sub，及びベクトル Mul Div の 4 つのパイプに割り当てた結果を図 **6** に示す．図の斜線部分は，表 4 の演算が割り当てられていることを示す．この図から，(4) で求められるループ 1 回あたりの推定実行サイクル数は 30 サイクルに分岐コスト 7 を加えた 37 サイクルとなることがわかる．(5) の手順より，ループ全体の推定実行サイクル数は 37 に回転数 64 を乗算した 2,368 サイクル数となる．最終的なプログラム全体のサイクル数は，(6) の手順により 2,368 にベクトル長変更のレイテンシ 4 を加算した 2,372 サイクルが推定値として算出される．

5. 評価

本節では，配列加算，行列積，畳み込み演算，及びコレスキー分解の 4 つの行列演算プログラムを使って，提案する VA 実行サイクル数推定モデルの精度を評価した結果を示す．また，アクセラレータ切り出し対象のカーネルループをホスト CPU・アクセラレータのどちらで実行させるかを選択する際に，この提案モデルが有効であるかどうかを評価した結果を示す．

© 2020 Information Processing Society of Japan

5.1 提案するモデルの推定精度評価

5.1.1 評価環境

本評価では OSCAR ベクトルマルチコアアーキテクチャが実装された FPGA をエミュレータとして使用した．本エミュレータは Intel/Altera 社の Cyclone V 上に構築している．CPU コアには NIOS II を使用し，動作周波数が 50MHz，命令キャッシュ・データキャッシュは共に 32KB である．また，ベクトルアクセラレータの動作周波数は 40MHz であり，ローカルメモリの容量は 64KB となっている．CPU コア用のターゲットコンパイラには gcc5.3.0 を，ベクトルアクセラレータ用バックエンドコンパイラには Clang/LLVM3.2 を使用した．最適化オプションには-O2 を用いた．

5.1.2 評価アプリケーション

評価には，配列加算の他に，科学技術計算や画像処理計算でよく用いられるプログラムである行列積・畳み込み演算・コレスキー分解の演算プログラムを用いた．コレスキー分解の演算では，実対称行列 A を下三角行列 L とその転置行列 L^T に分解する．各アプリケーションの入力行列の大きさ・データ型についてを以下の表 4 に示す．

表 4　評価アプリケーションの概要

配列加算	Data Size	4096
	Data Type	32bit floating-point
行列積	Data Size	64 x 64
	Data Type	32bit floating-point
畳み込み	Data Size	64 x 64
	Kernel Size	3x3
	Data Type	32bit floating-point
コレスキー分解	Data Size	64 x 64
	Data Type	32bit floating-point

5.1.3 推定精度の評価結果

4 つの評価アプリケーションについて，4 節で示した提案モデルを使って算出したベクトルアクセラレータ実行時の実行サイクル数の推定値と FPGA エミュレータ上で実行して得られた実測値，さらに実測値に対する推定値の誤差率を示した結果を表 5 に示す．各アプリケーションの推定値・誤差率の上段の値は，4.1 節にあるベクトルアクセラレータの実行形式のみを考慮した推定モデル (以下**モデル 1** とする) を使って算出した値を表す．また，下段の値は 4.1 節の実行形式に加えて 4.2 節の Clang/LLVM における最適化を考慮した推定モデル (以下**モデル 2** とする) を使って算出した値を表している．

全てのアプリケーションに対して，**モデル 1** より**モデル 2** の方がより実測値に近い値で推定できることが分かる．よって，ターゲットコンパイラ Clang/LLVM における最適化のうち，演算子の強さの低減化・ループアンローリングの 2 点を考慮することが，精度の高い推定に有効であることが分かる．

表 5　提案モデルによる VA 実行部の推定値と実測値の比較
(推定値/誤差率 : 上段は**モデル 1**，下段は**モデル 2** の値を表す)

プログラム	実測値	推定値	誤差率 [%]
配列加算	2326	2,692	+15.7
		2,372	**+1.98**
行列積	246,601	288,576	+17.0
		262,976	**+6.64**
畳み込み	28,954	42,954	+47.1
		26,846	**−3.99**
コレスキー分解	933,300	1,455,111	+55.9
		1,133,874	**+21.4**

また，配列加算，行列積，及び畳み込み演算のプログラムについては，**モデル 2** を使うことでいずれも高い精度で推定できることがわかる．上の 3 つののアプリケーションについては，OSCAR コンパイラでのコンパイルの段階で，ベクトル長・ループ回転数が一意に決定されるため，精度の高い推定が可能となる．

一方，コレスキー分解のプログラムについては，**モデル 2** の場合でも誤差率が 21.5[%] となり，上の 3 つのプログラムと比べて大きい．これは，コレスキー分解のプログラムに含まれるベクトル演算のベクトル長は，全て外側ループの誘導変数を含む変数で与えられることから，精度の高いコスト推定が難しいためである．評価に用いたプログラムでは，ベクトル長は外側ループの実行に応じて 0 から 63 までステップ数 1 で増加する．このことから，本論文におけるコスト推定の際には，変数で与えられるベクトル長を全て 0 と 63 の中央値である 32 として推定を行った．

5.2 OSCAR コンパイラによるアクセラレータ実行部の決定

VA 実行部の実行サイクル数推定を行う大きな目的は，OSCAR コンパイラ内部でアクセラレータ切り出し対象のカーネルループに対して，VA と CPU で実行させた場合のコストを比較し，速度向上が見込める場合のみアクセラレータで実行させるようにするためである．本研究では，5.1.2 節の 4 つの評価アプリケーションのカーネルループを利用して，提案モデルを使った 1VA 実行の場合の推定実行時間と，既存の OSCAR コンパイラの機能を使って算出される 1CPU 実行の場合の推定実行時間を比較し，1VA 実行の場合の推定速度向上率を算出した．また，各カーネルループを実際に 1VA・1CPU で実行させた際の実測時間，実速度向上率との比較を行った．その結果を**図 7**，**図 8** に示す．

図 7 の評価結果では，速度向上比の実測値が推定値に対して 2 倍以上となっている．これは，NIOS II のキャッシュがダイレクトマップ方式であり，キャッシュミスが多く発生することから，CPU で実行させたの場合の実行サ

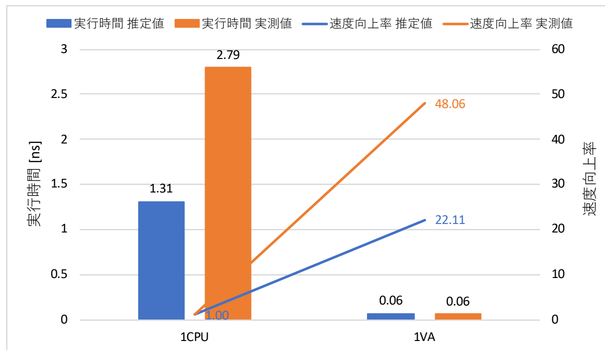

図 7　配列加算プログラムの 1CPU・1VA 実行の評価結果

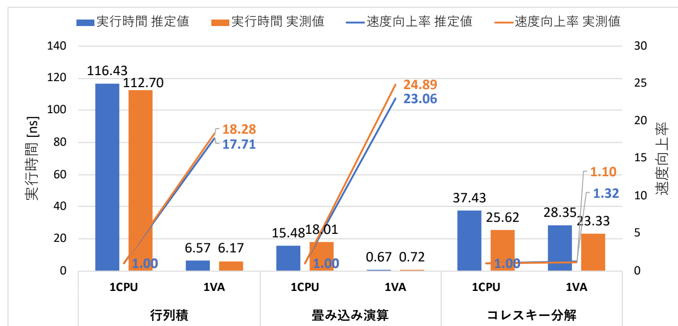

図 8　行列積・畳み込み・コレスキー分解の 1CPU・1VA 実行の評価結果

イクル数を OSCAR コンパラで正確に推定するのが難しいためである．しかし，速度向上比の推定値 22.11 より VA で実行させた場合に十分な速度向上が見込めることが，事前に推定できることが分かる．

図 8 の評価結果では，3 つのアプリケーションいずれも速度向上率の推定値と実測値の誤差が小さく，1CPU に対して 1VA で実行させたときの性能向上の割合を，OSCAR コンパイラにより事前に正確に推定できることが分かる．特に，コレスキー分解のプログラムでは，5.1.3 節で述べたようにベクトル化を行うループ回転数が OSCAR コンパイラの段階で推定することが困難であり，1CPU，1VA それぞれの実行時間の推定値・実測値の誤差は大きい．しかし，CPU 実行の場合のベクトル化を行うループの回転数と，VA 実行の場合のベクトル演算のベクトル長を同一に定めて推定を行うことで，CPU に対する VA の速度向上率の推定値は高い精度で算出できる．

以上の結果より，VA 向け実行サイクル数推定モデルにより算出された推定コストが，OSCAR コンパイラの機能の一つである CPU 実行の場合の推定実行コストと比較することで，アクセラレータの効率的な選択に有効であることがわかった．

なお，コレスキー分解のプログラムについては，コンパイラによる自動ベクトル化ではなく手動ベクトル化によりチューニングされたコードを利用することで，CPU に対して VA で実行させた際に，最大 11.8 倍の速度向上を達成することができる．

6. まとめ

本稿では，OSCAR コンパイラにおけるアクセラレータ実行部の効率的な選択を可能にするため，ベクトルアクセラレータで実行した際の実行サイクル数の正確な推定を可能にするモデルを提案した．

評価により，このモデルはループ回転数がコンパイル時に定まるプログラムについては高い精度で実行サイクル数の推定が可能である一方，回転数が変数で与えられ不定であるプログラムは，推定が難しいことが分かった．また，この VA コスト推定モデルが，OSCAR コンパイラにおいて，アクセラレータ使用時に十分な速度向上が得られる実行部のみを選択するために有効であることが確認できた．

謝辞

本研究の一部は国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務により行われた．

参考文献

[1] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J. et al.: End to end learning for self-driving cars, *arXiv preprint arXiv:1604.07316* (2016).

[2] CUDA: CUDA C PROGRAMMING GUIDE v9.1, NVIDIA Corporation (online), available from ⟨`https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf`⟩

[3] Nowatzki, T., Gangadhar, V., Ardalani, N. and Sankaralingam, K.: Stream-Dataflow Acceleration, *Proceedings of the 44th Annual International Symposium on Computer Architecture*, New York, NY, USA, Association for Computing Machinery, p. 416–429 (online), DOI: 10.1145/3079856.3080255 (2017).

[4] 宮本一輝，牧田哲也，　高橋健，柏俣智哉，　河田巧，狩野哲史，北村俊明，木村啓二，笠原博徳ほか：OSCAR ベクトルマルチコアプロセッサのための自動並列ベクトル化コンパイラフレームワーク，研究報告システムと LSI の設計技術 (SLDM)，Vol. 2018, No. 13, pp. 1–6 (2018).

[5] Hayashi, A., Wada, Y., Watanabe, T., Sekiguchi, T., Mase, M., Shirako, J., Kimura, K. and Kasahara, H.: Parallelizing compiler framework and API for power reduction and software productivity of real-time heterogeneous multicores, *International Workshop on Languages and Compilers for Parallel Computing*, Springer, pp. 184–198 (2010).

[6] Hunger, R.: *Floating Point Operations in Matrix-vector Calculus*, Munich University of Technology, Inst. for Circuit Theory and Signal Processing (2005).

© 2020 Information Processing Society of Japan