Vol.2016-ARC-221 No.4
2016/8/8

LLVMを用いた ベクトルアクセラレータ用コードのコンパイル手法

丸岡 晃[1]　無州 祐也[1]　狩野 哲史[1]　持山 貴司[2]　北村 俊明[2]　神谷 幸男[2]　高村 守幸[2]　木村 啓二[1]
笠原 博徳[1]

概要：科学技術計算や画像処理，機械学習の分野を始めとして，アプリケーションの高速化を実現するために各種アクセラレータが利用されている．アクセラレータを有効利用するためには対象アクセラレータに適したプログラムやデータ配置の最適化，ホストとアクセラレータ間のデータ転送や同期などの挿入が必要になるが，これらをプログラマが手動で行うことは困難であり，コンパイラによる自動化が望まれる．筆者等はこれまで OSCAR 自動並列化コンパイラにより，マルチコアプロセッサを対象として自動並列化に加えてメモリ最適化及びデータ転送最適化技術を開発してきた．この OSCAR コンパイラに対し自動ベクトル化技術で多くの実績を持つベクトルプロセッサの技術を取り入れ，さらにベクトルアクセラレータを利用することで，高速化及び低消費電力化を達成しつつプログラムの生産性を大幅に改善することが出来ると考える．本稿では OSCAR コンパイラが対象としてきた OSCAR マルチコアアーキテクチャにベクトルアクセラレータを加えた，プラチナマルチコアアーキテクチャ用の自動並列化・最適化を可能とするコンパイルフローを提案する．提案コンパイルフローでは OSCAR コンパイラによるコンパイル後のベクトルアクセラレータのコード生成に LLVM を利用しており，その実装の詳細も述べる．手動ベクトル化を行った主要カーネルに対してプラチナマルチコアシミュレータを用いて性能評価を行ったところ，1 つの CPU コア及び 1 つのアクセラレータコアを使用した場合，1 つの CPU コアのみによる実行と比較して行列積で 20.06 倍，2DConvolution で 22.23 倍の性能向上が得られることが確かめられた．

1. はじめに

科学技術計算や画像処理，機械学習等の分野を始めとして，アプリケーションの高速化及び低消費電力化が求められている．これらのアプリケーションの多くは高いデータ並列性を含んでいるため，これを活用して高性能化を実現するアクセラレータが利用されている．しかしアクセラレータを有効に利用するためには，アクセラレータのアーキテクチャに適したプログラムやデータ配置の最適化，ホストとアクセラレータ間のデータ転送や同期などといった操作の記述が必要となる．このような操作を手動で行うのは非常に困難であるため，アクセラレータを有効活用しつつソフトウェアの生産性を向上させるために，コンパイラによるこれらの自動化の実現が望まれる．

このような課題を解決するため，アクセラレータを利用するアプリケーションの開発を容易にするコンパイラや開発環境などが提案及び開発されてきた．NVIDIA の CUDA[1] では NVIDIA GPGPU[2] 向けの開発環境及び各種高速ライブラリなどを公開しているが，GPGPU 向けのプログラムを自作する場合は開発者がソースコードを CUDA C/C++言語で記述する必要がある．Chronos Group の OpenCL[3] では様々なヘテロジニアス環境に対する並列プログラミングフレームワークを提供しており，単一プログラムで複数の異なる CPU やアクセラレータ上での実行が可能になるが，性能を引き出すためには対象アーキテクチャに即したアプリケーションの最適化を行う必要が存在する．OpenMP4.0[4] や OpenACC[5] ではディレクティブベースによるアクセラレータに対するオフロード部の指定やデータ転送の指定が可能だが，オフロード部やデータ転送部の指定，アクセラレータに適した最適化やデータ転送のオーバーラップなどは開発者が行う必要がある．

一方，筆者らは OSCAR 自動並列化コンパイラ [6] によって，マルチコアに対する自動並列化や自動電力削減，メモリ配置最適化技術を開発してきた．さらに OSCAR API[7][8] ではヘテロジニアス環境に対する並列化・低消費電力化のための指示文が用意され，OSCAR コンパイラによるアク

[1] 早稲田大学
Waseda University.
[2] オスカーテクノロジー株式会社
Oscar Technology Corporation.

セラレータを含めたタスクジューリングやデータ転送の最適化などが可能であるが，アクセラレータ用プログラムの最適化はソフトウェア開発者が依然として行う必要があった．

そこで，富士通の VP/VPP シリーズ [9][10] で開発されてきたベクトルプロセッサをベースに，組み込み用に短ベクトル長に抑え低消費電力化を図ったアクセラレータを開発し，我が国の産業界がベクトルプロセッサ用に蓄積してきた自動ベクトル化技術を OSCAR コンパイラに追加することによって，プロセッサ間の並列化及びアクセラレータ用プログラムの最適化を自動化し，ソフトウェア生産性の向上を目指す．

本稿では，OSCAR コンパイラが対象としてきた OSCAR マルチコアアーキテクチャ [11] にベクトルアクセラレータを付与した，プラチナマルチコアアーキテクチャ用コンパイルフローを提案する．提案手法では，OSCAR コンパイラによる自動並列化やメモリ最適化，ホストとアクセラレータ間のデータ転送やアクセラレータ制御コードの挿入に加えて，ベクトルアクセラレータ用プログラムの自動ベクトル化を行う．加えて提案コンパイルフローではベクトルアクセラレータのコード生成部に対して LLVM[12] のバックエンドを利用しており，その実装の詳細についても述べる．また手動ベクトル化した主要カーネルに対してプラチナマルチコアシミュレータ上で性能評価を行った結果についても報告する．

以下 2 章では評価対象とするプラチナ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アーキテクチャについて，3 章ではベクトルアクセラレータに対するコンパイル手法について，4 章では性能評価について，そして 5 章ではまとめについて述べる．

2. プラチナマルチコアアーキテクチャ

本章では，本研究で対象とするプラチナマルチコアアーキテクチャ，及びアクセラレータとなるベクトルアクセラレータについて述べる。

2.1 プロセッサアーキテクチャ

プラチナマルチコアアーキテクチャは OSCAR マルチコアアーキテクチャ [11] をベースとし，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) を付与したアーキテクチャである．各 PE は相互接合網で接続され，プロセッサ外部には各 PE 間の共有データが格納される集中共有メモリ (CSM) が接続される．PE は CPU とローカルデータメモリ (LDM)，分散共有メモリ (DSM)，データ転送ユニット (DTU)，そして VA で構成される．プラチナマルチコアアーキテクチャ図を図 1 に示す．

LDM は基本的には自 PE 内のみがアクセスできる高速なメモリであり，各 PE のプライベートなデータが格納される．DSM は自 PE と他 PE の両方から同時アクセス可能なメモリであり，タスク間のデータ転送や同期フラグなどの PE 間で授受されるべき共有データが格納される．DTU は CPU，VA と独立にデータ転送を行うことができる DMA コントローラであり，タスク処理とデータ転送がオーバーラップ可能となっている．CSM は LDM や DSM に比べてアクセス時間が長いメモリだが，容量が大きくプログラム及びデータの全てが格納されている．プログラム実行時には DTU あるいは CPU のデータ転送命令によって，タスク処理前に CSM から各 PE の LDM や DSM に転送することにより，高速なメモリアクセスを実現することができる．VA はベクトル演算を搭載したアクセラレータであり，各 PE に搭載され，PE 内の CPU によって起動される．VA は LDM 及び DSM に対してのみアクセスすることができ，CSM に直接アクセスすることはできない．

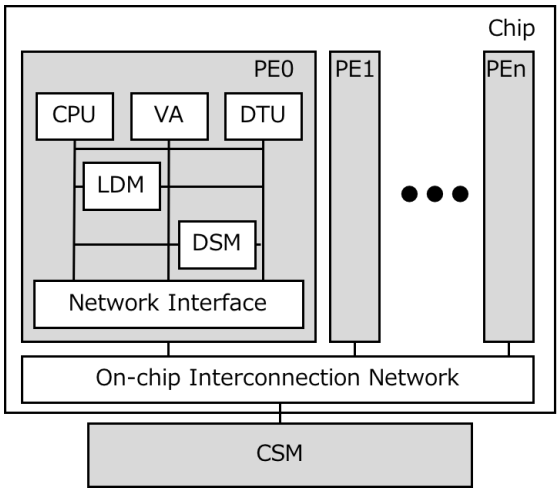

図 1 プラチナマルチコアアーキテクチャ図

2.2 ベクトルアクセラレータアーキテクチャ

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・低消費電力処理を目的とするアクセラレータである．本アクセラレータは CPU 非依存に設計されており，任意のプロセッサコアを CPU として使用することが可能となっている．VA にはベクトル演算器及びスカラ演算器が搭載されており，本評価では 256bit 幅のベクトル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．VR の 1 エントリ当たりのサイズは 256Byte であり，8bit データの場合は 256 エレメント，64bit データの場合は 32 エレメントのデータが搭載可能となっている．

各ベクトル命令は MR を指定することでマスク演算を行うことが可能であり，条件分岐がある場合でも簡単にベクトル化することが可能となっている．また MR のエントリ

0 を指定すると，マスクを使用しないベクトル命令を実行することが出来る．ベクトル長は可変となっており，プログラム中でベクトル長設定命令を実行することによって指定することが可能となっている．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．本ベクトルアクセラレータは組み込み用途も想定しており，スーパーコンピュータ用の長いベクトル長でも組み込み用の短いベクトル長でも高いスループットを実現することができる．

3. 提案するコンパイル手法

本章ではプラチナマルチコアに対して，OSCAR 自動並列化コンパイラ [6] による自動ベクトル化及びアクセラレータ用コード生成と，LLVM[12] による VA 用オブジェクトコード生成手法を提案する．本章で述べた VA のオブジェクトコード生成手法に基づいて，LLVM のバックエンドに対して VA 用ターゲットの実装を行った．

プラチナマルチコアに対するコンパイルフローは図 2 のように，OSCAR 自動並列化コンパイラと，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では 3.1 節に述べるように，逐次 C ソースコードを入力として自動並列化やメモリ最適化，アクセラレータ制御コードの挿入に加えて自動ベクトル化を行い，ホスト CPU 用並列化 C ソースコードと VA 用ベクトル化 C ソースコードを出力する．ホスト CPU 用コード並列化 C ソースコードは GCC や Clang などのホスト CPU 用ネイティブコンパイラによりコンパイルされ，ホスト CPU 用オブジェクトコードが生成される．VA 用ベクトル化 C ソースコードは 3.2 節に述べるように，VA 用コード生成を行うよう拡張された Clang/LLVM によってコンパイルされ，VA 用オブジェクトコードが生成される．最後にホスト CPU 用のオブジェクトコードと VA 用のオブジェクトコードをリンクすることによって，最終的な実行可能バイナリを生成する．

3.1 OSCAR 自動並列化コンパイラ

OSCAR 自動並列化コンパイラでは逐次用 C ソースコードを入力とし，従来の並列化やデータローカライゼーションのための解析やリストラクチャリングに加えて，ベクトル化のための解析やリストラクチャリング，VA 実行部コードの分離，ホストとアクセラレータ間のデータ転送や同期コードの挿入などを行う．

以下 VA 用コード生成に関わる点についてコンパイルの様子の詳細を述べる．まず OSCAR コンパイラでは VA 実行部の検出を行う．VA 実行部はループ並列性解析によりベクトル化可能と解析されたループ，あるいは OSCAR API[7][8] のヒント指示文によって VA 実行部として指定されたループが候補となる．次に VA 実行部に対して命令のベクトル化，及びベクトル化された命令のオペランドをベクトル変数に変換する．その後 VA 実行部の入力及び出力変数を検出し，VA 実行部を入出力変数を引数とした関数として切り出す．その後ホストとアクセラレータ間のデータ転送操作と同期コードを挿入する．

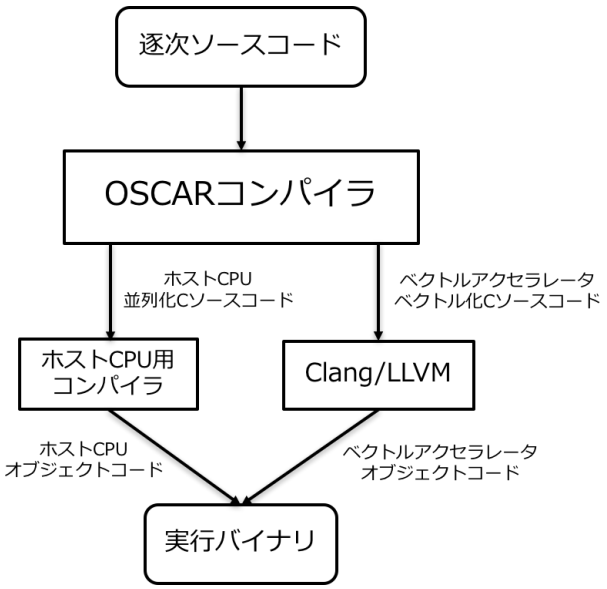

図 2 提案するコンパイルフロー図

プラチナマルチコア用のコンパイルフローにおける OSCAR 自動並列化コンパイラの出力コードは，従来のホスト CPU 用並列化 OSCAR API C ソースコードに加えて，VA 用ベクトル化 C ソースコードの 2 つとなる．VA 用ベクトル化 C ソースコードでは，スカラ処理部は通常の C 言語の文法で記述され，ベクトル処理部はベクトル命令に対応した Intrinsic 関数の呼び出しの形式で記述される．

ベクトル加算を例とした時の入力逐次 C ソースコードイメージを図 3 に，出力ベクトル化 C ソースコードイメージを図 4 にそれぞれ示す．図 4 では，図 3 に示されている入力 C ソースコードがベクトル化された結果，ベクトル長でストリップマイニングされ，さらにその内部で VA のベクトル型の変数宣言と，それらをオペランドとしたベクトル命令に対応する Intrinsic 関数の呼び出しが記述される．

これらのベクトル型や Intrinsic 関数はベクトル化 C ソースコード内で include されているヘッダファイル内で宣言・定義する．ヘッダファイル内の宣言・定義例を図 5 に示す．図 5 のようにベクトル型は__attribute__((__vector_type__())) 宣言によるベクトル型として定義する．これによって，LLVM-IR において VectorType の変数として扱うことが可能となる．Intrinsic 関数の定義は，基本演算かつマスク無しの演算の場合はベクトル型変数の演算として、複雑な演算やマスク有りの演算の場合は演算は Builtin 関数の呼び出しとして記述する．LLVM-IR 上においては，これら Builtin 関数を対応した

Intrinsic 関数へ変換することによってコード生成が可能となる．

```
void vec_add(float* dst,
             const float* src0,
             const float* src1,
             int size)
{
  for (i = 0; i < size; i++) {
    dst[i] = src0[i] + src1[i];
  }
}
```

図 3 入力逐次 C ソースコードイメージ

```
#include <ptintrin.h>
void vec_add(float* dst,
             const float* src0,
             const float* src1,
             int size)
{
  int vr_size = 64;
  __pvf v_dst, v_src0, v_src1;

  for (i=0; i<size; i+=vr_size) {
    v_src0 = _pt_vld_f(&src0[i]);
    v_src1 = _pt_vld_f(&src1[i]);
    v_dst = _pt_vadd_f(v_src0, v_src1);
    _pt_vst_f(v_dst, &dst[i]);
  }
}
```

図 4 出力ベクトル化 C ソースコードイメージ

```
typedef float __pvf
  __attribute__ ((__vector_size__ (2048)));


static __inline __pvf
__attribute__((__always_inline__, __nodebug__))
_pt_vadd_f(__pvf __a, __pvf __b)
{
    return (__pvf)__builtin_pt_vadd_f
      ((__pvf)__a, (__pvf)__b);
}
```

図 5 ベクトル型と Intrinsic 関数の定義コードイメージ

3.2 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，または手動でベクトル化したベクトル化 C ソースコードを入力として，VA の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詳細を説明する．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[13] 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，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，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表現される．

マスク無し加算命令を LLVM-IR 及びアセンブリコードへと変換する例を図 6 に, マスク有り加算命令を LLVM-IR 及びアセンブラコードへ変換する例を図 7 にそれぞれ示す．基本的なマスク無し演算の場合は，図 6 のように LLVM-IR 上では LLVM-IR の Opcode を使用し，VectorType をオペランドとした命令として表現する．この場合，アセンブリコード上ではマスクのオペランドにはマスク無しを指定するため MR0 が割り当てられる．マスク有りの演算の場合では，図 7 のように LLVM-IR 上では Intrinsic 関数の呼び出しとして表記し，MR の割り当てが適切に行われてアセンブリコードが出力される．これによりマスクを使用するなどの複雑なベクトル演算に対しても適切にコンパイルを行うことが可能となる．

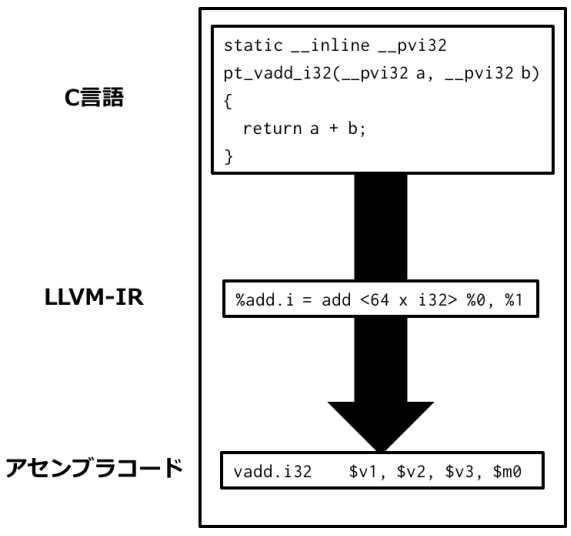

図 6 マスク無し演算におけるコンパイル時のコード変換イメージ

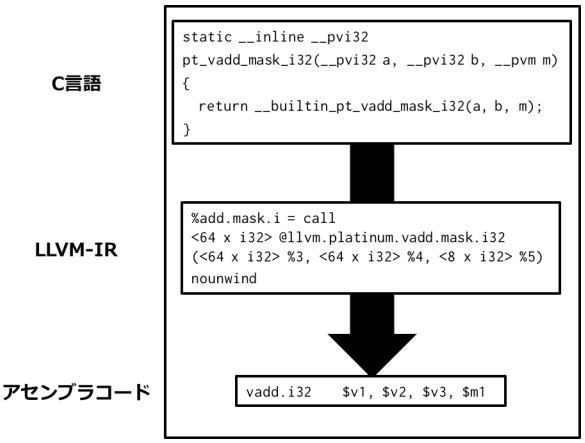

図 7 マスク有り演算におけるコンパイル時のコード変換イメージ

4. 性能評価

本章では 3 章で提案したコンパイルフローのうち，VA のコード生成部を LLVM に拡張実装し，手動でベクトル化を行った計算カーネルの性能をプラチナマルチコアシミュレータ上で評価した結果について述べる．

4.1 評価環境

本評価で使用したプラチナマルチコアシミュレータの構成を表 1 に示す．

CPU には SPARC v9 規格に準拠したプロセッサを使用している．

VA はスカラ命令に関しては加減算ユニットと乗算ユニットがそれぞれ 1 本ずつ，ロードストアユニットが 1 本，ベクトル命令に関しては加減算ユニットと乗算ユニットがそれぞれ 1 本ずつ，ロードストアユニットが 2 本存在し，シングル Issue の構成となっている．各種ベクトル演算ユニット及びロードストアユニットはチェイニングによるベクトル命令間のパイプライン実行が可能となっている．各種ベクトル演算器は 64bit 演算器が 4 個並列に並んでいるため，単一クロックで 256bit 幅の演算が可能となっている．

各メモリのレイテンシは組み込み用途を意識し，LDM と DSM が 1 クロックサイクル，CSM が 60 クロックサイクルとなっている．

上記評価環境のもとで評価対象の計算カーネルを GCC でコンパイルし 1 つの CPU コアのみで実行した場合と，手動ベクトル化した計算カーネルを 3.2 節で述べた手法を実装した Clang/LLVM でコンパイルし 1 つの CPU コアと 1VA コアにおいて実行した場合の性能を比較する．使用したコンパイラ情報を表 2 に示す．

4.2 評価プログラム

評価プログラムとして，DeepLearning を始め各種アプリケーションにて頻出される計算カーネルである行列積と 2DConvolution を使用する．各評価プログラムのパラメータを表 3 に示す．

表 1 プラチナマルチコアシミュレータの構成

CPU	Instruction Set	SPARC v9
	L1 Cache Size	32KB
	L2 Cache Size	512KB
VA	Scalar Int/FP ADD/SUB Unit	1
	Scalar Int/FP MUL Unit	1
	Scalar LOAD/STORE Unit	1
	Vector Int/FP ADD/SUB Unit	1
	Vector Int/FP MUL Unit	1
	Vector LOAD/STORE Unit	2
	Vector Unit Width	256bit
Memory	LDM Latency	1 clock cycle
	DSM Latency	1 clock cycle
	CSM Latency	60 clock cycle

表 2 評価に使用したコンパイラ情報

Compiler	GCC	Clang/LLVM
Version	4.7.2	3.2
Option	-O3	-O2

行列積では入力及び出力配列のサイズは 256x256，データ型は単精度浮動小数点型としている．手動ベクトル化コードでは，C=AxB における B 及び C の列の次元でベクトル化を行っている．

2DConvolution では入力及び出力配列のサイズは 256x256，カーネルサイズは 3x3，データ型は単精度浮動小数点型としている．手動ベクトル化コードでは入力及び出力データの x 次元でベクトル化を行っている．

どちらのプログラムにおいても，入力データは全て予め LDM に格納されている状態から評価を行う．

表 3 評価プログラムのパラメータ

Matmul	Data Size	256x256
	Data Type	32bit Floating-point
2DConv	Data Size	256x256
	Kernel Size	3x3
	Data Type	32bit Floating-point

4.3 評価結果

プラチナマルチコアシミュレータ上で行列積を動作させた場合の性能評価結果を図 8 に，2DConvolution を動作させた場合の性能評価結果を図 9 に示す．それぞれ縦軸は実行クロックサイクル数を示している．

行列積においては 1CPU 実行時では 20200 万サイクルなのに対して，1CPU+1VA 実行では 1007 万サイクルとなっており，CPU と VA が同一周波数であると仮定すると 20.06 倍の性能向上となっている．また 2DConvolution においては 1CPU 実行時では 676 万サイクルなのに対し

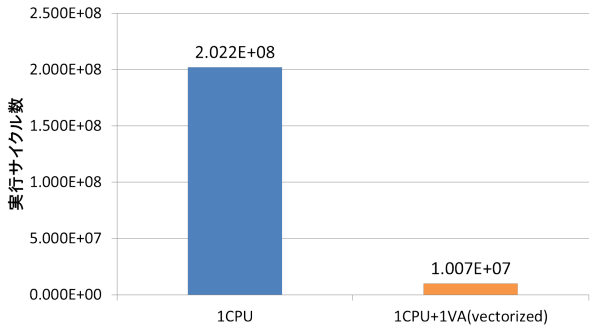

図 8 行列積の 1CPU 実行と 1CPU+1VA 実行における実行サイクル数

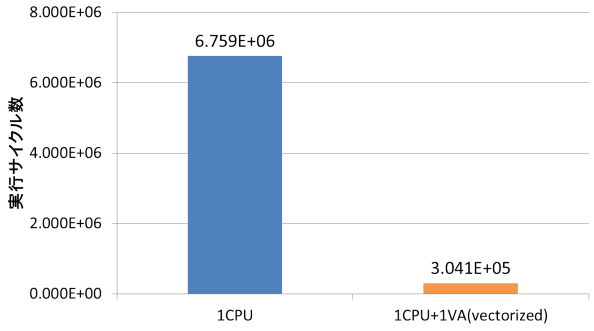

図 9 2DConvolution の 1CPU 実行と 1CPU+1VA 実行における実行サイクル数

て，1CPU+1VA 実行時では 30 万サイクルとなっており，22.23 倍の性能向上となっている．

この結果より VA におけるベクトル実行によってプログラムの性能向上が可能であり，さらにベクトル化 C ソースコードから LLVM によって VA 用オブジェクトコードの生成が可能であることが確認できた．

5. まとめ

本稿では組み込み用途から科学技術計算までの高速化と低消費電力化を目指し，OSCAR マルチコアアーキテクチャにスーパーコンピュータで利用されてきたベクトルアクセラレータを付与したプラチナマルチコアアーキテクチャを対象として自動並列化・最適化を行うコンパイルフローを提案した．本コンパイルフローでは，OSCAR 自動並列化コンパイラにおいて自動並列化やメモリ最適化に加えて自動ベクトル化，アクセラレータの制御やホストとアクセラレータ間のデータ転送の自動挿入を行う．さらに LLVM を用いて，OSCAR コンパイラによって生成されたベクトル化 C ソースコードからベクトルアクセラレータのオブジェクトコードを生成する．本手法のうちベクトルアクセラレータのコード生成部を LLVM に拡張実装し，手動ベクトル化したプログラムをコンパイルしプラチナマルチコアシミュレータ上で評価を行った結果，1 つの CPU コア及び 1 つのアクセラレータコア上での実行において，1 つの CPU コア実行に対して行列積で 20.06 倍，2DConvolution で 22.23 倍の性能向上が得られた．また，本手法によってベクトル化 C ソースコードからベクトルアクセラレータのオブジェクトコード生成が可能であることが確認された．

謝辞

本研究の一部は科研費基盤研究 (C)15K00085 の助成により行われた．

参考文献

[1] NVIDIACorporation: *CUDA Zone*.
[2] Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M. and Buck, I.: GPGPU: General-purpose computation on graphics hardware, *SC '06 Proceedings of the 2006 ACM/IEEE conference on Supercomputing Article* (2006).
[3] KhronosGroup: *OpenCL*.
[4] OpenMP.org: *OpenMP*.
[5] OpenACC-standard.org: *OpenACC*.
[6] Kasahara, H., Honda, H., Mogi, A., Ogura, A., Fujiwara, K. and Narita, S.: A multi-grain parallelizing compilation scheme for OSCAR (optimally scheduled advanced multiprocessor), *Fourth International Workshop Santa Clara* (1991).
[7] 林明宏，和田康孝，渡辺岳志，関口威，間瀬正啓，白子準，木村啓二，笠原博徳：ヘテロジニアスマルチコア向けソフトウェア開発フレームワーク及び API，情報処理学会論文誌コンピューティングシステム (ACS36)，Vol. 5, No. 1, pp. 68–79 (2011).
[8] Kimura, K., lvarez Cecilia, G., Hayashi, A., Mikami, H., Shimaoka, M., Shirako, J. and Kasahara, H.: OSCAR API v2.1: Extensions for an Advanced Accelerator Control Scheme to a Low-Power Multicore API, *7th Workshop on Compilers for Parallel Computing (CPC2013)* (2013).
[9] Tamura, H., Kamiya, S. and Ishigaki, T.: FACOM VP-100/200: Supercomputers with ease of use, *Parallel Computing* (1985).
[10] Miura, K., Takamura, M., Sakamoto, Y. and Okada, S.: Overview of the Fujitsu VPP500 supercomputer, *Compcon Spring '93, Digest of Papers.* (1993).
[11] Kimura, K., Wada, Y., Nakano, H., Kodaka, T., Shirako, J., Ishizaka, K. and Kasahara, H.: Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor, *Proc. of 9th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9)* (2005).
[12] llvm.org: *The LLVM Compiler Infrastructure*.
[13] llvm.org: *clang: a C language family frontend for LLVM*.