コードの性能可搬性を提供するSIMD向け共通記述方式

中西 悠[†,☆] 渡邊 啓正[†]
平澤 将一[†] 本多 弘樹[†]

近年，マルチメディア処理の高速化を目的としてSIMD命令セットを搭載する汎用プロセッサが増えている．これらのプロセッサではアセンブリ言語や組み込み関数を用いて明示的にSIMD命令を記述することによりSIMD命令を効果的に利用することができる．しかし，アーキテクチャに依存した記述であるためコードの可搬性を著しく低下させるという問題がある．本研究では，明示的にSIMD命令を用いる記述方式の共通化をはかり，この共通記述方式を実際にソースコードに変換するツールを開発した．これによってコードの性能可搬性を保ちながら明示的にSIMD命令を利用することが可能となる．また，本方式を用いていくつかの例題を記述し，コードの性能可搬性および性能を評価した．結果として，速度向上と性能可搬性を確認することができた．

Common Description Method of SIMD Instructions for Providing Performance Portability

Yu Nakanishi,[†,☆] Hiromasa Watanabe,[†] Shoichi Hirasawa[†] and Hiroki Honda[†]

In recent years, many general-purpose processors have extensions of SIMD instruction set for multimedia processing. By Descripting SIMD instructions explicitly using assembly language and intrinsic function, a programmer can use SIMD instruction effectively. But, they have a problem on code portability, because of description methods that depend on particular architecture. In our study, we proposed a common description method for using SIMD instructions explicitly and developed a tool to convert common description method into a source code including SIMD instructions effectively. Our method enables a programmer to use SIMD instructions explicitly while keeping code performance portability. We implemented some examples by our method and evaluated their performance and performance portability. As a result, we confirmed speedup and performance portability.

1. はじめに

近年，マルチメディア処理の高速化を目的としてSIMD（Single Instruction Multiple Data）命令セットを搭載する汎用プロセッサが増えている[1)~3)]．これらSIMD命令セットはマルチメディア処理を中心に非常に高い性能を発揮するが，その特性から命令の動作が複雑であるため，通常の命令セットに比べ利用が容易ではない．SIMD命令による速度向上をより容易に得られるように様々な研究がなされており[4)~6),8),9)]，SIMD命令を利用するための様々なアプローチが試みられている．これらは自動ベクトル化コンパイラに自動でSIMD命令を生成させる手法とプログラマがSIMD命令の適用可能箇所を検出し，人手でSIMD命令を記述する手法に大別できる．

現在，自動ベクトル化コンパイラがソースコードから潜在的なSIMD命令の適用可能箇所を見つけだし，自動的にSIMD命令を生成することは容易ではない．これは，従来の命令セットでは要求されなかったデータの並列性やメモリアライメントの条件を満たす必要があることに起因する．

明示的にSIMD命令を利用する方法としてアセンブラや組み込み関数が広く用いられている[8),9)]．しかし，プログラマがアーキテクチャごとの命令特性を十分に理解する必要があることに加え，これらの手法で記述されたコードはアーキテクチャに依存した記述であるため可搬性が失われるという問題がある．

コードの可搬性を向上させる方法としては，アーキテクチャに依存しない可搬性のあるコードと特定の

† 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications
☆ 現在，株式会社東芝
Presently with TOSHIBA Corporation

アーキテクチャに依存したコードを併記する方法がある．しかし，この手法では一部のアーキテクチャにおいてのみ SIMD による速度向上が得られ，他の SIMD アーキテクチャでは SIMD による速度向上が得られない．こうしたコードは性能可搬性がないといえる．本研究における性能可搬性とは SIMD 命令を用いることで速度向上が見込める複数の環境上で，SIMD 命令を活用し速度向上させることができる性質のことを指す．

本研究では明示的に SIMD 命令を記述したコードに性能可搬性を持たせる手法として SIMD 命令向けの共通記述方式を提案する．本方式は，明示的に SIMD 命令を利用し，かつコードの可搬性と性能可搬性を保つことを目的としている．

本研究では本方式の実現手段として共通記述方式向けのコンパイラを実装した．また，共通記述方式を用いて行列積，高速フーリエ変換，色空間の変換のプログラムを記述し，評価を行った．その結果，コードの可搬性および性能可搬性を保てることを確認した．

本稿の構成を以下に示す．2 章で本研究の対象となっている SIMD 命令の特徴と従来の SIMD 命令を用いたプログラミング手法について述べる．3 章では提案する共通記述方式の概要について述べ，4 章で共通記述方式を実現するコンパイラの概要を述べる．5 章で共通記述方式を用いてプログラミングしたアプリケーションにおける実行性能，性能可搬性について評価する．6 章で関連研究について述べ，7 章で結論と今後の課題を述べる．

2. 従来の SIMD プログラミング手法

近年の汎用プロセッサには SIMD 命令セットが搭載されている．これらのプロセッサの多くは 1 つのレジスタを分割し，1 命令で複数のデータを処理するレジスタ分割方式である．本研究ではレジスタ分割方式の SIMD 命令を対象とし，以降に登場する SIMD 命令とはこの方式を用いた SIMD 命令のことを指す．

代表的なレジスタ分割方式の SIMD 命令セットには Intel の SSE や IBM の VMX，SCE/東芝の EmotionEngine がある．これら命令セットの主な処理対象としてはマルチメディア処理があり，画像や音声といった膨大な量のデータに対して単純な演算を実行する用途を中心に用いられている．

現在，SIMD 命令を利用するプログラミング手法はライブラリから間接的に利用する方法と自動ベクトル化コンパイラによる SIMD 命令の自動生成，アセンブリ言語や組み込み関数で直接 SIMD 命令を記述する方法の 3 つに大別できる．以降にそれぞれの特徴を述べる．

2.1 ライブラリからの利用

一部の数値計算ライブラリ[7] では SIMD 命令を用いた高速化が試みられている．これらのライブラリを利用することで間接的に SIMD 命令を利用することが可能である．SIMD 命令に関するプログラミングやその知識はライブラリ製作者に求められ，性能はライブラリ製作者に依存する．ライブラリが既存の場合，容易に SIMD 命令による速度向上が得られる．しかし，SIMD 命令の利用法としては受身であり，利用者が必要とする機能を持つライブラリが存在しない場合には SIMD 命令の利用そのものが困難である．そのため柔軟なプログラミングには向いていない．

2.2 自動ベクトル化コンパイラによる SIMD 命令の自動生成

自動ベクトル化コンパイラがソースコードから潜在的な SIMD 命令の適用可能箇所を見つけだし，SIMD 命令を生成する手法である[4]．現存のアプリケーションのみではなく過去のアプリケーションに対しても SIMD 命令による速度向上が期待できる．

しかし，SIMD 命令を活用するためにはソースコードは SIMD 命令の特徴をとらえた記述をされていなければならない．たとえば，SIMD 命令の演算は連続したメモリに対して行うことを前提としているため，ソースコード上においても連続したメモリに対して演算を行うように記述しなければ SIMD 命令の生成は期待できない．並列に処理されるデータの依存関係も自動ベクトル化コンパイラに対して明確に示しておかなければ SIMD 命令の生成ができない．また，コンパイラによって解釈できる記述が異なるため，生成条件を明確にすることが困難である．たとえば，ある自動ベクトル化コンパイラでは**図 1** の (A) は SIMD 命令を生成することができるが，(B) は生成することがで

```
(A)
  for (i = 0 ; i < N ; i++){
    if (x[i]>y[i]) z[i] = x[i];
    else           z[i] = y[i];
  }

(B)
  for (i = 0 ; i < N ; i++) {
    if (x[i]>y[i])       z[i] = x[i];
    else if (x[i]<=y[i]) z[i] = y[i];
  }
```

図 1 コンパイラによる生成条件の違い
Fig. 1 Difference in generation condition of SIMD instructions according to compilers.

きない．これは同義なコードであるにもかかわらず，後者がコンパイラ内の変換パターンに該当しないためである．このように自動ベクトル化コンパイラの持つ SIMD 命令の生成条件は不明瞭であり，コンパイラがプログラマの意図したとおりに SIMD 命令を生成することは困難である．

以上から，SIMD 命令を効果的に利用する手段としては適していない．

2.3 アセンブリ言語，組み込み関数からの利用

明示的に SIMD 命令を利用する手法としてアセンブリ言語や組み込み関数によるプログラミング手法が広く普及している[8),9)]．この方法はアーキテクチャに密接したプログラミング手法であるため，細やかな最適化を施すことが可能である．しかし，組み込み関数やアセンブリ言語を用いるとアーキテクチャやコンパイル環境が限定され，コードの可搬性が消失する．コードを別の環境に移行する場合はプログラマは新たにアーキテクチャを学習したうえでコードを修正しなければならない．

2.4 従来の SIMD プログラミング手法の問題点

従来の 3 手法を比較する．ライブラリや自動ベクトル化コンパイラは容易に利用可能で，可搬性が高いため開発効率が良い．しかし，SIMD 命令の効果は限定的にしか得られない．より多くのアプリケーションに対し，効果的に SIMD 命令を利用したい場合は，プログラマの意思を反映させやすい手法であるアセンブリ言語や組み込み関数で記述を行う．しかし，コードの可搬性は消失する．

以上から，従来の SIMD プログラミング手法ではコードの可搬性を高く保ったまま SIMD 命令を効果的に利用できない．

3. 共通記述方式の提案

明示的に SIMD 命令を用いることにより，より多くのアプリケーションに対してより効果的に SIMD 命令が利用できる．しかし，既存のプログラミング手法では明示的に SIMD 命令を用いて速度向上を得るにはそれぞれの SIMD 命令セット向けにコードを記述しなければならない．新しいアーキテクチャが登場し，互換性のない新しい SIMD 命令セットが現れると，SIMD 命令による速度向上を得るためにはプログラムの修正を要する．多くのソフトウェア資源を持つ場合，移行にかかるコストは膨大である．

この問題の解決方法として，SIMD 命令に関する記述の共通化を考える．これは，従来 SIMD 命令セットごとに異なった組み込み関数を命令セット間の違いを抽象化したうえで統一化を行ったものとする．

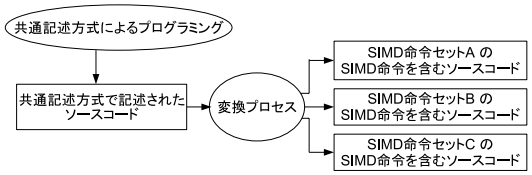


図 2 共通記述方式の概要
Fig. 2 Outline of common description method.

SIMD 命令セット間の違いは命令，ベクトルサイズ，レジスタ数の 3 つの違いに分けることができる．命令，ベクトルサイズ，レジスタをそれぞれ抽象化することで，3 つの違いを隠蔽し，SIMD 命令に関する記述を共通化する．そして，この抽象化された記述を多数の SIMD アーキテクチャ向けに再び具体化することで多数の SIMD アーキテクチャ上で動作させることができる．これによりプログラムの移行に必要な修正の手間を低減させることができ，SIMD 命令セットの違いから発生する問題が解決される．

図 2 に共通記述方式の概要図を示す．図では共通記述方式で記述されたソースコードが変換プロセスを経て SIMD 命令セット A〜C の命令を含むコードに変換される．

本方式ではメモリ上の連続したデータに対して演算処理を行うループ文を対象とする．これは，SIMD 命令があるメモリアドレスから連続したベクトルサイズ分のデータを一度にロード，ストアし演算処理する命令であるため，メモリ上の連続したデータに対して演算処理を行うループ文が最も SIMD 命令の効果を発揮しやすいためである．また，対象のループはナチュラルループであるという条件と，ループ内で処理されるデータ間にデータ依存関係が存在しないという条件を満たす必要がある．なぜならば，ナチュラルループでない場合や依存関係が存在する場合はベクトル命令の処理するベクトルサイズによっては正しい実行結果が得られなくなるためである．ループの回転数が SIMD 命令の並列数の倍数でない場合は，並列数の端数回の演算は SIMD 命令で処理することはできない．この場合は，端数部分のループを分割し，端数部分を逐次命令で実行することで正しい結果が得られる．

提案手法によりプログラマは明示的に SIMD 命令の生成を指示することができる．数値計算などのライブラリ関数よりも低い水準のプログラミングが可能となり，プログラマが必要とする機能の実装可能性が高まる．自動ベクトル化コンパイラは SIMD 命令生成のために依存解析や分岐構造の解析といった高度な解析

を必要とするが，提案手法では依存関係などの条件はプログラマが保証するものとしているため，コンパイラに高度な解析機能を必要としない．これにより，コンパイラの解析機能に依存した変換パターンの曖昧さの問題が解消される．また，提案手法によりコードに性能可搬性を与えることで，組み込み関数やアセンブリ言語で記述されたコードに存在したソフトウェア資源の蓄積の困難さの問題を解消できると考えられる．

以降に SIMD 命令に関する記述の共通化に必要となる命令，ベクトルサイズ，レジスタ数の違いを抽象化する手法を以降に述べる．

3.1 命令の違いに対する抽象化

共通記述方式では命令セットごとの命令の違いを抽象化するために，処理目的を明確にした上で命令の抽象化を行う．

たとえば命令セットごとに異なることが多い命令としてアラインメントが揃っていない場合のロード命令がある．このロードは専用のロード命令を用いる方法と複数に分割してロードし合成する方法がある．専用のロード命令が存在する場合は，命令を変えるだけでロードが可能となる．存在しない場合は，**図 3** のようにアラインメントが揃っているアドレスに対してロード命令を複数回実行し，並べ替え命令を用いて同様の結果を得る．

以上のように同じ目的の処理でも命令セットによって用いられる命令が異なる場合が存在する．しかし，「アラインメントが揃っていないアドレスに対するロード」という目的は同じである．よって目的ごとに抽象化を行い，内部で利用される命令セットごとに異なる SIMD 命令を隠蔽する．このように目的別に機能を抽象化し，関数型の API を定義する．たとえば単精度浮動小数点の積和演算を行う API は以下のように定義できる．vec_float は単精度浮動小数点型を要素に持つベクトル変数を示している．

```
//積和演算 ret = arg1 + arg2 * arg3
vec_float/float*  vmaddf(vec_float/float* arg1,
                         vec_float/float* arg2,
                         vec_float/float* arg3)
```

API は SIMD 命令に変換するプロセスで SIMD 命令列に変換される．引数にポインタが渡された場合はロード命令が生成されベクトルレジスタにポインタの指している先のデータをロードする．積和命令の存在する命令セットでは積和命令に，存在しない命令セットでは積命令と和命令の命令列に変換される．また，代入先がポインタであった場合はストア命令が生成され，ベクトルレジスタの内容をポインタの指している先のデータをストアする．これにより，命令セット間の命令の差異を吸収する．

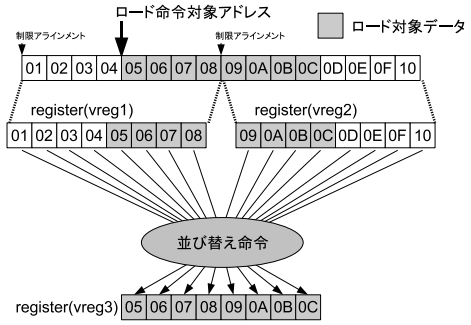


図 3 並べ替え命令によるアライメントの揃っていないロード
Fig. 3 Load for misaligned memory using shuffle instruction.

3.2 ベクトルサイズの違いに対する抽象化

共通記述方式で記述された演算が対象アーキテクチャの SIMD 命令に変換された場合，演算の対象となるデータ列は SIMD 命令によって並列に処理される．しかし，並列に処理されるデータの要素数はベクトルレジスタのサイズと対象データのデータ型によって決定されるため，ベクトルレジスタのサイズを抽象化すると並列数を明記できない．

そこで，プログラムの記述時は SIMD 命令の並列数を明記せず並列数の最低単位である 1 並列（スカラ）で記述を行う手法を考える．関数型 API と SIMD 命令では処理するデータ長が異なるため，API を SIMD 命令に置き換えた後に正しい計算結果を得るための処理が必要となる．たとえば N 並列でデータ処理を行う SIMD 命令に置き換えた場合，N 要素分のデータがその命令で処理されることになる．この処理は，あるメモリアドレスからの連続参照および処理を行うループの N 回分のイテレーションに相当する．よって置き換え後のプログラムにおいてループのイテレーション回数は N 分の 1 となるよう変換することで正しい計算結果を得られる変換となる．

もしループ内に存在する複数の命令の並列数がそれぞれ異なる場合，ループを変換するためにループに含まれる命令の並列数から並列数の最小公倍数を求める．次にそれぞれの命令が最小公倍数分のイテレーションの実行に相当するようにループを書き換えることで記述内容と等しい処理結果となる．たとえば 2 並列の命令と 3 並列の SIMD 命令がループ内に存在する場合，2 並列の命令を 3 命令分，3 並列の命令を 2 命令分配置し，2 と 3 の最小公倍数である 6 回分のイテレーションに相当するように書き換える．処理されるデータ間にデータ依存関係が存在しない場合，実行結果は並列

数やベクトルサイズに依存しないため，この書き換え処理によって実行結果が変化することはない．以上の処理を行うことでベクトルサイズの違いを吸収する．

3.3 レジスタ数の違いに対する抽象化

共通記述方式で記述されたコードを SIMD 命令に変換するプロセスにおけるベクトルレジスタの必要数，利用可能数は SIMD 命令セットによって異なる．よって，抽象化された記述の段階では具体的なレジスタの記述をすることはできない．

共通記述方式ではベクトルレジスタを抽象化したベクトル変数を提供する．ベクトル変数は利用可能なレジスタ数にかかわらず利用することができる．コード記述時およびコードを SIMD 命令に変換するプロセスではすべてベクトル変数を用いる．ベクトル変数は変換プロセスの最終段階で利用可能なベクトルレジスタ数を考慮したうえでベクトルレジスタに変換される．これにより，ベクトルレジスタの数の違いを吸収する．

3.4 提案手法の実現

提案手法を実現するためには共通記述方式で記述されたコードを SIMD 命令に変換するためのアーキテクチャ情報と，その情報をもとに SIMD 命令への変換を行う変換プロセスが必要となる．アーキテクチャ情報は共通記述方式を SIMD 命令に変換するための変換規則と利用可能なレジスタセットの情報からなる．変換規則は関数型の API を同義な SIMD 命令あるいは SIMD 命令列に変換するための規則の集合である．SIMD 命令が受付けるレジスタセットや，メモリアライメントの制限といった SIMD 命令の実行に関する制限の情報も含まれる．また，SIMD 命令を実行するには SIMD 命令が受付けるレジスタを命令に渡す必要があるため，ベクトル変数をレジスタに変換するときに用いられるレジスタ情報が必要となる．

変換プロセスでは共通記述方式を用いて記述されたコードをアーキテクチャ情報をもとに対象アーキテクチャの SIMD 命令に変換する．このプロセスでは，まず共通記述方式で記述されたソースコードとアーキテクチャ情報の解釈を行う．次に，アーキテクチャの情報から得られた変換規則をもとにコード内の共通記述方式で定義された関数型 API を SIMD 命令列に変換する．SIMD 命令を含むループ内の式や他の SIMD 命令の並列数から最小公倍数を求め，各式と命令を 1 イテレーションで最小公倍数分のイテレーションの実行に相当するように書き換える．最後にベクトル変数にベクトルレジスタを割り付け，SIMD 命令を実行可能な状態にする．以上のようなプロセスを実行することで SIMD アーキテクチャ上で実行可能なコードを得ることができる．

表 1 提供される API
Table 1 API provided by our method.

算術演算	vadd@	各ベクトル要素の加算を行う．
	vsub@	各ベクトル要素の減算を行う．
	vmul@	各ベクトル要素の乗算を行う．
	vmull@	各ベクトル要素の乗算を行う（計算結果の下半分を残す．vmul@と同義）．
	vmulh@	各ベクトル要素の乗算を行う（計算結果の上半分を残す）．
	vmadd@	各ベクトル要素の積和演算を行う．
	vadd@st	各ベクトル要素の飽和加算を行う．
	vsub@st	各ベクトル要素の飽和減算を行う．
比較演算 選択演算	vcmpeq@	各ベクトル要素の比較を行う．要素が等しければすべてのビットを 1 で埋める．
	vcmpgt@	各ベクトル要素の比較を行う．片方の要素が多きければすべてのビットを 1 で埋める．
	vcmpge@	各ベクトル要素の比較を行う．片方の要素が大きい，または等しければすべてのビットを 1 で埋める．
	vmax@	各ベクトル要素の比較を行う．要素がの大きいほうを選択しする．
	vmin@	各ベクトル要素の比較を行う．要素がの小さいほうを選択しする．
	vavr@	各ベクトル要素の比較を行い，平均値を計算する．
論理演算	vand	各ベクトル要素で論理積を計算する．
	vor	各ベクトル要素で論理和を計算する．
	vnot	各ベクトル要素で否定を計算する．
	vxor	各ベクトル要素で排他的論理和を計算する．
シフト 演算	vsrl@	ベクトル内の要素を右論理シフト
	vsll@	ベクトル内の要素を左論理シフト
	vsra@	ベクトル内の要素を右算術シフト
その他	varray@	ベクトルサイズにあわせた配列を生成する．
	vsum@	ベクトル内の要素の和を計算する．
	vconv@@	ベクトル内のデータ型を変換する．

@ には対象のデータ型を示す記号が入る．
b=8 bit 整数，h=16 bit 整数，w=32 bit 整数，f=単精度浮動小数点

3.5 共通記述方式の具体例

提案手法で提供している API の一覧を**表 1** に示す．ここでは連続データに対して演算を施すストリーミング処理に対して有効と考えられる演算を選択した．現在，基本的な演算を API としてまとめており，多数の SIMD 命令セットにおいても API の可搬性を保つことが可能であると考えている．しかし，実際に未知の SIMD 命令セットに対して API の可搬性が提供できるかはさらなる検討が必要である．

共通記述方式を用いて記述した行列積の例を**図 4** に示す．このプログラムは SIZE × SIZE の浮動小数型の正方行列の積を求めるプログラムである．vo. とい

```
1: Intput : float **matA, float **matB
2: for (i = 0 ; i < SIZE ; i=i+1){
3:  for (j = 0 ; j < SIZE ; j=j+1){
4:    vo.zero(vo.reg1); /*ゼロ初期化*/
5:    for (k = 0 ; k < SIZE ; k=k+1) {
6:      /*積和演算
7:      vo.reg1 = vo.reg1+matA[j][k]*matB[i][k] */
8:      vo.reg1 = vo.vmaddf(vo.reg1,
9:                          matA[j]+k,  matB[i]+k);
10:   }
11:   /* ベクトル要素の和 */
12:   matC[i][j] = vo.vsumf(vo.reg1);
13: }
14:}
```

図 4 共通記述方式を用いた記述例（行列積）
Fig. 4 Matrix multiplication by common description method.

うプレフィクスがついたものが共通記述方式で定義された関数型 API およびベクトル変数である．それぞれ，vmaddf，vsumf というものが関数型 API であり，reg1 がベクトル変数である．vmaddf は単精度浮動小数点の積和演算を行う API であり，vsumf は単精度浮動小数点を要素とするベクトル内の和を求めている API である．

変換プロセスでは，まず，アーキテクチャ情報から zero，vmaddf，vsumf のそれぞれの API についての変換規則を探す．それぞれの並列数や実行条件を求めたうえで変換を開始する．たとえば vmaddf は matA と matB をロードする命令と積和命令または積命令と和命令の命令列に変換される．ロード命令では matA と matB が指すメモリ上からベクトル変数へベクトルサイズ分のロードを行われる．ベクトル変数に対して積命令，和命令で演算が行われ，結果が reg1 に出力される．命令列の変換後に各命令の並列数に従って各命令を含むループを展開する．積命令，和命令が N 並列であった場合，1 並列である $k = k + 1$ を N 回分の実行に相当する $k = k + N$ に書き換える．最後にツールは変換された命令列に含まれる reg1 のようなベクトル変数に利用可能なベクトルレジスタを割り付け，実行可能なコードにする．

4. コンパイラの実装

共通記述方式で記述されたコードは変換プロセスによって SIMD アーキテクチャ上で動作するコードに変換される．本研究ではこの変換プロセスを実行するコンパイラを設計し，実装した．本コンパイラが変換プロセスを実行することで，同一のソースコードから多種の SIMD 命令セットの命令を含むコードが生成

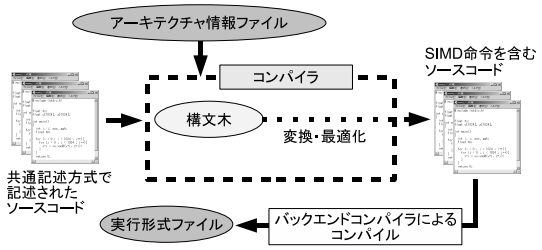


図 5 全体の概要図
Fig. 5 The entire flow chart.

される．

本研究で実装したコンパイラは C/C++の構文と共通記述方式で記述されたコードから，構文木を生成する．コンパイラはアーキテクチャの情報が記述されたファイルをもとに構文木に含まれる関数型 API を SIMD 命令列に変換し，SIMD 命令を含むループの変形や最適化を行う．変換結果は再び C/C++のコードに戻され，出力される．出力されたコードは既存の C/C++コンパイラによってコンパイルされ，実行可能となる（**図 5**）．現在，出力されたコードをコンパイルするバックエンドコンパイラとして gcc に対応している．SIMD 命令は gcc の拡張インラインアセンブラで出力している．

アーキテクチャ情報ファイルとコンパイラの詳細について以降に述べる．

4.1 アーキテクチャ情報ファイル

アーキテクチャ情報ファイルは共通記述方式を SIMD 命令に変換するための変換規則と利用可能なレジスタセットの情報が記述されたファイルである．このファイルは本研究で設計，実装したコンパイラとは独立したファイルとする．コンパイラからファイルを分離しておくことで，ファイルを追加するだけで別の SIMD 命令セットに対応できる．このファイルに記述しておく情報は API の変換規則とレジスタ情報である．レジスタ情報の記述例と API の変換規則の記述例をそれぞれ**図 6**，**図 7** に示す．

図 6 は xmm という名前のレジスタセットの記述である．2〜5 行目に記述されている%xmm0〜%xmm3 がベクトルレジスタの名前である．名前の後に記述されている 128 はベクトルレジスタのビット長である．この例における xmm レジスタセットは 128 bit のベクトルレジスタが 4 つあるレジスタセットであることを示している．

図 7 は単精度浮動小数点積 API の記述である．2 行目は API の結果が代入されるベクトル変数の変数名である xmm3 が記述されている．3〜6 行目は API

```
1:(registerset xmm              ; レジスタセット名
2:             ((%xmm0 128)     ; レジスタ情報
3:              (%xmm1 128)
4:              (%xmm2 128)
5:              (%xmm3 128)))
```

図 6 レジスタ情報の記述例
Fig. 6 Example of register set information.

```
1:(builtin
2:  (xmm3) ; 出力
3:  (; 使用されるデータの情報
4:   (xmm1 ((reg xmm) ((float) 128)))
5:   (xmm2 ((reg xmm) ((float) 128)))
6:   (xmm3 ((reg xmm) ((float) 128))))
7:  (@vmulf xmm1 xmm2) ; API フォーマット
8:  ((= xmm3 xmm1)     ; 変換規則
9:   (asm "mulps @xmm2, @xmm3" (xmm3) (xmm2 xmm3))))
```

図 7 API 変換規則情報の記述例
Fig. 7 Example of API conversion rule.

で使用するデータの情報が記述されている．ここでは float 型のデータを 128 bit 長に 4 個収めたものであるということと，最終的に xmm という名前のレジスタセットのレジスタに変換されることを示している．7 行目は API の名前 vmulf と，引数として受け取るベクトル変数名 xmm1 と xmm2 が記述されている．8〜9 行目が API の変換規則である．8 行目の規則で xmm1 の内容が xmm3 にコピーされる．これは xmm1 を API の変換規則内で破壊しないようにするためである．xmm1 がこの API 以降に利用されない場合は，後に行われる最適化でこの規則は削除される．また，9 行目では mulps という命令が生成されることが示されている．書かれている文字列が実際に出力されるアセンブラコードである．xmm2，xmm3 のような「@」が先頭についたものは，別の変換規則で変換されることを示している．xmm2 や xmm3 はレジスタ割付けによってアーキテクチャ上のレジスタに変換される．また，文字列の後に記述されている 1 つ目の括弧内に示されている xmm3 は mulps で破壊されるベクトル変数名である．2 つ目の括弧内に示されている xmm2，xmm3 は mulps が直前の内容を利用するベクトル変数名を表している．

4.2 コンパイラ

本研究で実装したコンパイラは共通記述方式を含む C/C++言語，およびアーキテクチャ情報ファイルを受付けるパーサを持ち，共通記述方式で記述されたソースコードを対象 SIMD アーキテクチャの SIMD 命令を含むコードを生成する．本コンパイラは C++言語で記述されており，パーサは flex および bison を用いて実装されている．パーサが解析したソースコードは二分木構造の構文木に変換され，コンパイラはこの構文木に対して，変換処理を行う．

変換処理では関数型 API の変換とループの変換が行われる．本コンパイラはアーキテクチャ情報ファイルの記述内容をもとに API を SIMD 命令に置き換える．置き換え後，正しい結果を得るためにループの書き換えを行う．まず，コンパイラはループ内のすべての命令の並列数から並列数の最小公倍数を計算し，それぞれの命令を 1 イテレーションで最小公倍数分のイテレーションに相当するようにループを書き換える．

また，SIMD 命令は密なループで用いられることが多いため，無駄のあるコードは性能に大きく影響する．本コンパイラでは従来のコンパイラで行われているようなコードの冗長性の削除を行い，最適化を行ったコードを生成する．

構文木の段階では変換処理により生成された SIMD 命令はベクトル変数を含んでいる．本コンパイラはグラフカラーリングアルゴリズム[10] を用いてベクトル変数に実レジスタを割り付ける．

最後に，構文木を C/C++のコードに変換しソースコードを出力する．

共通記述方式はメモリへの連続アクセスや依存関係の条件を満たす必要があるが，これらはプログラマによって保証されるため，コンパイラではそれ条件の解析や SIMD 命令の生成が可能であるかの判定は行っていない．

5. 評　　価

ここでは 3 章で提案した共通記述方式を用いて行列積，高速フーリエ変換，色空間の変換のプログラムを記述しコードの可搬性および性能可搬性を評価する．対象となる SIMD 命令セットは Intel の SSE/SSE2[1]，IBM/モトローラの VMX[2]，ソニー/東芝の EmotionEngine[3]（以下 EE）を選択した．

本章ではまず，評価方法について述べる．次に実行結果を示し，以降にそれに対する評価と考察を行う．

5.1 評 価 方 法

評価基準はコードの可搬性およびコードの性能とし，この 2 点の結果から性能可搬性を評価する．コードの可搬性については提案手法で得られたコードが各 SIMD 命令セット上で動作するか検証し評価する．コード性能については SIMD 命令を用いていないコードと提案手法で得られたコード，人手で SIMD 命令を記述したコードの高速化率を比較することで評価する．以上の評価を行うために SIMD 命令を用いていないコード，共通記述方式で記述したコード，人手で

表 2 評価環境
Table 2 Evaluation environment.

CPU	命令セット	gcc version
Xeon 2.80 GHz	SSE/SSE2	gcc 4.01
PowerPC G5 2.30 GHz	VMX	gcc 4.00
R5900 V3.1 300 MHz	EmotionEngine	gcc 2.95

SIMD 命令を記述したコードの 3 種類を用意し，実行性能を比較する．また，人手に頼らない SIMD 命令の生成手法である自動ベクトル化コンパイラとの比較を行った．

評価プログラムとして行列積，高速フーリエ変換，色空間の変換を選択した．行列積は 2 つの正方行列の積を計算し，同サイズの正方行列に出力する．対象データのデータ型は単精度浮動小数点および 8，16，32 bit 整数を選択した．行列積はロード，ストア命令，積，和命令，ベクトルの和計算命令によって構成される．積命令は命令セットごとに SIMD 命令がサポートしているデータ型が違うため，必要に応じてデータ型を変換する変換命令を挿入する．変換命令は並べ替え命令によって実現されるが，並べ替え命令の挙動は命令セットごとに異なるため，適切な命令を選択する必要がある．共通記述方式では，命令セットごとに異なる変換命令は隠蔽される．また，ベクトルの和計算は和命令と並べ替え命令によって構成される．型変換と同様に並べ替え命令の挙動は命令セットごとに異なるため，適切な命令を選択する必要がある．

高速フーリエ変換は単精度浮動小数点のデータ列に対して高速フーリエ変換を行う．これはロード，ストア命令，積，和，差命令，ベクトルの和計算命令によって構成される．行列積と同様にベクトルの和計算命令を構成する並べ替え命令の挙動が命令セットごとに異なるため，適切な命令を選択する必要がある．

色空間の変換は 24 bit RGB カラーの画像を 24 bit YUV カラーに変換するプログラムである．これは RGB 各 8 bit の色情報を 16 bit に変換する変換命令と，ロード，ストア命令，積，和命令で構成されている．ここで用いられる変換命令は演算精度を上げるためのもので，共通記述方式においても API で明示的に記述される．ただし，変換命令を実現する並べ替え命令の挙動は命令セットごとに異なるため，適切な命令を選択する必要がある．

評価を行った環境を**表 2** に示す．gcc の最適化オプションはすべて O3 で行った．人手で SIMD 命令を記述したコードは gcc の拡張インラインアセンブラを用いている．

5.2 実行結果

共通記述方式で記述されたコードを本研究で実装したコンパイラで変換したところ，各命令セットに対して適切なコードが生成された．命令セットによって異なる SIMD 命令がサポートしているデータ型の違いに対しても型変換に必要な適切な並べ替え命令を生成することで対応していることを確認した．また，ベクトルの和計算に用いられる並べ替え命令についても適切な命令が生成されていることを確認した．以上のように，提案手法によって各命令セットに対して実行可能なコードが生成された．

行列積，高速フーリエ変換，色空間の変換についての実行結果をそれぞれ**図 8**，**図 9**，**図 10** に示す．実行結果は SIMD 命令を用いていないコードに対する高速化率で表している．Normal，Manual，Tool はそれぞれ，SIMD 命令を用いていないコード，人手で SIMD 命令を記述したコード，提案手法によって得られたコードの性能を表している．

提案手法によって得られたコードは人手で SIMD 命令を記述したコードには及ばないものの，すべての評価プログラム，すべての命令セットにおいて SIMD 命令による速度向上が得られた．提案手法で得られたコードにおけるそれぞれのプログラムの高速化率は行列積で平均 2.93 倍，高速フーリエ変換で平均 2.72 倍，色空間の変換で平均 1.65 倍であった．

5.3 考察

コードの可搬性について評価する．提案手法で得られたコードを gcc でコンパイルし，実行できることを確認した．このことからコードの可搬性があることが示された．

次にコード性能について評価する．提案手法によって得られたコードは SIMD 命令を用いていないコードと比較して，すべての評価プログラム，すべての命令セットにおいて SIMD 命令を活用した速度向上を得ている．また，人手で記述したコードの高速化率は行列積で平均 4.21 倍，高速フーリエ変換で平均 4.11 倍，色空間の変換で平均 2.06 倍であったことから，提案手法による高速化率は人手で記述したコードの高速化率の 7 割程度得られることが分かる．提案手法で得られたコードと人手で記述したコードを比較したところ，高速化率の差につながるコードの差異はループのアンローリングやそれにともなうレジスタの活用不足，命令スケジューリングの不足であった．今後，本研究で実装したコンパイラでは行われなかったループのアンローリングやレジスタ割付け，命令スケジューリングの最適化といった機能を追加することにより，人手

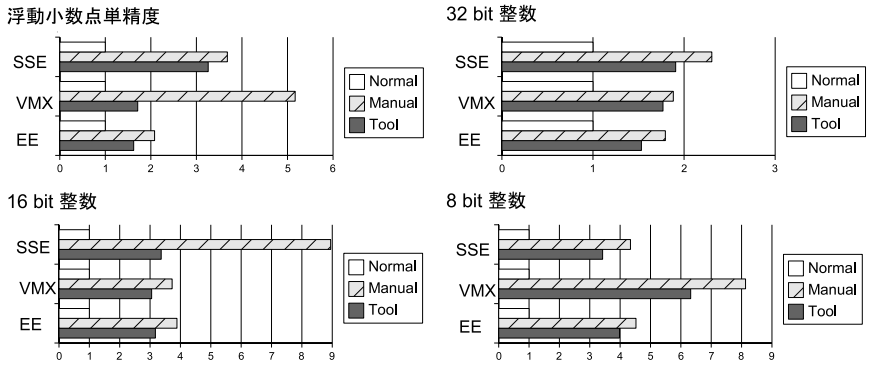


図 8 行列積実行結果
Fig. 8 Result on matrix multiplication.

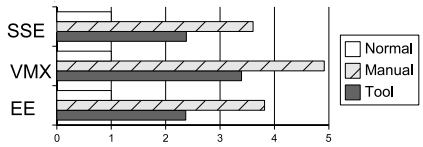


図 9 高速フーリエ変換実行結果
Fig. 9 Result on fast Fourier transform.

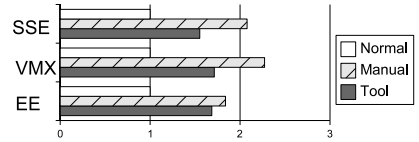


図 10 色空間の変換実行結果
Fig. 10 Result on color conversion algorithm.

で記述したコードとの差を縮小することが可能であると考えている．

以上から，提案手法によって得られたコードは可搬性を持ちつつ，すべての命令セットで SIMD 命令を利用した速度向上を得ている．また，人手で記述したコードには及ばなかったが性能の差は最適化の強化により縮小することができると考えている．このことより提案手法によって得られたコードに性能可搬性があることが示された．

また，人手に頼らない SIMD 命令の生成手法である自動ベクトル化コンパイラとの比較例として Intel プロセッサ上で比較を行った．コンパイラは gcc 4.1.1 を用いた．今回の評価に用いられた行列積，高速フーリエ変換，色空間の変換プログラムを自動ベクトル化向けのオプションを用いてそれぞれコンパイルした．コンパイラには先の評価に用いた SIMD 命令を利用していないコードと同一のコードを与えた．結果的に SIMD 命令が生成されたプログラムは 32 bit 整数の行列積のみであった．SIMD 命令が生成されなかったコードについてはコードを改良することにより，コンパイラが SIMD 命令を生成する可能性はある．しかし，どのように改良すればコンパイラが自動ベクトル化可能なものと判断することができるかは明らかでない．それに対して，提案手法では明確に SIMD 命令の生成を指示することができるため，自動ベクトル化コンパイラが SIMD 命令を容易に生成できないコードに対してもプログラマの意図で SIMD 命令による速度向上が得られた．コード性能は，提案手法で得られたコードの高速化率が 1.91 倍であったのに対し，自動ベクトル化コンパイラから得られたコードの高速化率は 1.84 倍で，提案手法の高速化率が上回った．

6. 関連研究

関連研究としては SIMD 命令の自動生成を試みる自動ベクトル化の分野がある[4]．なかでもマルチプラットフォームをターゲットとした研究[11] が存在しており，コードの再利用による開発の効率化とプログラマの負担軽減が期待されている．

しかし，現状では自動ベクトル化コンパイラが自動生成できる SIMD 命令は限られている．これはソースコードがベクトル化を意識して記述されていないことと，SIMD 命令生成に必要となる複雑なパターンマッチングへの対応が難しいことに起因する．特に，プログラマに SIMD 命令を意識させずにコードを記述させることは，プログラマへの負担が少ない反面，SIMD 命令を活用する機会を奪ってしまう．結果的にアーキテクチャの性能が十分に生かせないという事態を招くため，ベクトル化の重要性が高い問題については高速化手法として不十分である．

7. おわりに

本研究では，SIMD 命令セット向け共通記述方式を

提案し，本方式を実現するためのコンパイラを実装し，3 種類のアーキテクチャ情報ファイルを記述した．本方式を用いて行列積，高速フーリエ変換，色空間のプログラムを記述し，評価を行った．評価は提案手法によって生成されたコードと SIMD 命令を用いないコードと人手で SIMD 命令を記述したコードの 3 つを比較することで行った．対象の SIMD 命令セットは SSE，VMX，EmotionEngine とし，コードの可搬性，性能の可搬性，コード性能の 3 点について評価を行った．

人手による最適化は各命令セットごとに新たに記述しなおさなければならなかったのに対し，提案手法では単一のソースコードから各命令セットの SIMD 命令が適用されたコードを生成することができ，コードの可搬性を示すことができた．また，コード性能については提案手法によって複数のアーキテクチャ上で人手で SIMD 命令を記述したコードの 7 割程度の速度向上を容易に得られることを確認し，性能可搬性が保てることを示した．

以上のことから，本研究で提案した共通記述方式を用いてアプリケーション開発を行うことで他の SIMD 命令セットへ速やかに移行することが可能になる．複数の命令セットを対象としたアプリケーションの開発では SIMD 命令を用いるために生じる開発コストは大きな問題となる．こうしたコストを大幅に減らすことにより，SIMD 命令を用いた高性能アプリケーションの開発効率が改善される．

7.1 今後の課題

本研究では 3.5 節で提示した API を用いてサンプルプログラムを実装し，評価を行った．今後はこれら API で対応可能な SIMD 命令セットやアプリケーションを検証し，評価する必要がある．

また，提案手法の応用として，ヘテロジニアスな環境に対する統一的なプログラミングの提示があげられる．近年，Cell プロセッサのような複数の異なる SIMD 命令を持つマルチコアプロセッサが登場しているが，こうしたプロセッサにおいて統一的な記述方式で各 SIMD 命令セットを活用できない．本方式はアーキテクチャ情報ファイルを各命令セット用に記述することで対応できるが，現在のコンパイラ実装では複数の SIMD 命令セットに対応できていない．バックエンドコンパイラを含むコンパイル環境を複数命令セットに対応させることにより統一的なプログラミングが可能となる．

参 考 文 献

1) Intel Corporation: IA-32 Intel(R) Architecture Software Developer's Manuals. http://www.intel.com/products/processor/manuals/
2) IBM Corporation: PowerPC Microprocessor: VMX Technology Programming Environments Manual. http://www-306.ibm.com/chips/
3) Sony Computer Entertainment, Toshiba Corporation: EE User's Manual.
4) Bik, A.J.C.: Automatic Intra-Register Vectorization for the Intel Architecture, *International Journal of Parallel Programming*, Vol.30, Issue 2, pp.65–98 (2002).
5) Wu, P. and Wang, A.: Efficient SIMD Code Generation for Runtime Alignment and Length Conversion, *CGO'05* pp.153–164 (2005).
6) BuliC, P. and GuStin, V.: Fast Dependence Analysis in a Multimedia Vectorizing Compiler, *Parallel, Distributed and Network-Based Processing* (2004).
7) Whaley, R.C. and Petitet, A.: ATLAS Project. http://math-atlas.sourceforge.net/
8) Intel Corporation: Intel C++ Compiler. http://softwarecommunity.intel.com/isn/home/
9) Apple Corporation: Altivec Programming. http://developer.apple.com/hardwaredrivers/ve/
10) Briggs, Preston, and Cooper, K.D.: Coloring Heuristics for Register Allocation, *16th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Language*, pp.275–284 (1989).
11) Nuzman, D. and Henderson, R.: Multi-platform Auto-vectorization, *The 4th Annual International Symposium on Code Generation and Optimization* (2006).

(平成 19 年 1 月 22 日受付)
(平成 19 年 5 月 7 日採録)

中西 悠

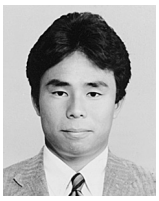
2005 電気通信大学電気通信学部情報科学科卒業．2007 電気通信大学大学院情報システム学研究科博士前期課程修了．同年株式会社東芝に入社．コンパイラ，計算機アーキテクチャ等に興味を持つ．

渡邊 啓正（学生会員）

2003 年電気通信大学情報工学科卒業．2004 年同大学大学院情報システム学研究科博士前期課程修了．同年より同大学大学院情報システム学研究科博士後期課程在学中．計算グリッドのプログラム開発支援環境に関する研究に従事．情報処理学会第 65 回全国大会にて学生奨励賞受賞．

平澤 将一（正会員）

2000 年東京大学理学部情報科学科卒業．2002 年東京大学大学院理学系研究科情報科学専攻修了．2005 年東京大学大学院情報理工学系研究科コンピュータ科学専攻博士課程単位取得退学．2006 年電気通信大学大学院情報システム学研究科助手．2007 年電気通信大学大学院情報システム学研究科助教．高性能計算機システム，並列計算システム，プログラミング言語処理に関する研究に従事．



本多 弘樹（正会員）

1984 年早稲田大学理工学部電気工学科卒業．1991 年同大学大学院理工学研究科博士課程修了．1987 年より同大学情報科学研究教育センター助手．1991 年より山梨大学工学部電子情報工学科専任講師．1992 年より同助教授．1997 年より電気通信大学大学院情報システム学研究科助教授．2007 年より同教授．並列処理方式，並列化コンパイラ，並列コンピュータアーキテクチャ，グリッド等の研究に従事．工学博士．電子情報通信学会，IEEE-CS，ACS 各会員．平成 15 年度山下記念研究賞授賞．