

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[†] 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。SIMD 命令を効果的に利用するために様々な研究がなされているが、SIMD 命令の適用箇所が明示的でないソースコードに自動的に SIMD 命令を適用することは容易ではない。明示的に SIMD 命令を利用する方法としてアセンブリ言語や組み込み関数が広く用いられているが、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 (API) 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効果的に適用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、本方式を用いて行列積、高速フーリエ変換をプログラミングし、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[†] and HIROKI HONDA[†]

In recent years, many general-purpose processors have extensions of SIMD instruction set for multimedia processing. Various researches about effective use of SIMD instructions have been performed. However, if a source code doesn't have a part that SIMD instructions can be clearly applied to, it is not easy for a compiler to generate SIMD instructions automatically. Assembly language and intrinsic function are widely used as methods for using SIMD instructions explicitly. However, 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 (API) for using SIMD instructions explicitly and developed a tool to apply common description method to a source code effectively. Our method enables a programmer to use SIMD instructions explicitly while keeping code performance portability. We implemented Matrix Multiplication and Fast Fourier Transform 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)}。しかし、SIMD 命令はその特性から動作が複雑であったり、データの配置などの条件を満たしている必要がある。

SIMD 命令を効果的に利用するために様々な研究がなされているが、^{6), 7)} ソースコードから潜在的な適用箇所を見つけだし、自動的に SIMD 命令の性能を引き出すことは容易ではない。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^{4), 5)}。しかし、プログラマがアーキテクチャごとの命令の特性を十分に理解する必要があることに加え、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問題がある。

本研究では SIMD 命令向けの共通記述方式を用いることでソースコードに SIMD 命令を適用する試みを行った。これにより、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、この記述方式を用いて行列積および高速フーリエ変換をプログラミングし、評価を行った。

本稿の構成を以下に示す。第 2 章で本研究で対象となっている SIMD 命令の特徴について述べ、第 3 章で従来の SIMD 命令を用いたプログラミング手法について述べる。第 4 章で共通記述方式と実際のソースコードへの適用の方法を述べ、第 5 章で SIMD 命令

[†] 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

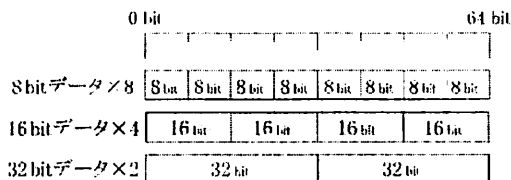


図 1 レジスタの分割パターン

をソースコードに適用するツールの概要を述べる。第 6 章で共通記述方式を用いてプログラミングした行列積および高速フーリエ変換における、実行性能、性能可搬性について評価する。第 7 章で結論と今後の課題を述べる。

2. SIMD 命令

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。代表的なものには Intel の MMX/SSE¹⁾、IBM/モトローラの VMX²⁾、ソニー/東芝の EmotionEngine³⁾ などがある。

これらに搭載されている SIMD 命令セットの殆どは N bit 長のレジスタ (多くの場合、N は 64 または 128) を複数に分割することでデータを並列に処理することができる。これらレジスタは SIMD 命令専用のものが設けられていることが多いが、汎用レジスタをそのままあるいは組み合わせる場合もある。

レジスタの分割方法は対象となるデータの単位サイズによって複数存在する。単位データサイズは 8 bit、16 bit、32 bit 整数と単精度浮動小数点のうち全て、あるいはいずれかがサポートされているのが一般的である。それぞれのデータ分割方法を N = 64 とした例を図 1 に示す。並列に扱える要素の数は $N \div (\text{単位データサイズ})$ で求まる。

データをメモリからロードする場合、通常 N bit のデータを一度にロードする。データをストアする際も同様である。よって、並列処理の対象となるデータはメモリ上で連続に配置されている必要があり、SIMD 命令を効果的に利用するにはデータの配置が非常に重要である。また、複数個の要素が並列で処理されるため、データ間の依存関係にも注意する必要がある。

本研究では SIMD 命令を以下のように分類する。

(1) データ転送命令

この命令はデータをメモリからレジスタにロードする、あるいはレジスタからメモリにストアする命令である。ロード先、ストア先のメモリアライメントに条件が設けられている場合があるので注意が必要である。

表 1 従来のプログラミング手法の一覧

	コード可搬性	コード性能	柔軟性
数値計算ライブラリなどからの利用	△	○	×
コンパイラによる SIMD 命令の自動生成	○	×～△	○
アセンブリ言語、組み込み関数からの利用	×	○	○

(2) 算術演算命令

分割された各データ要素に対して加算、積算などの算術演算を行う。演算結果が隣り合ったデータ要素に影響を及ぼすことはない。

(3) 飽和演算命令

分割された各データ要素に対して飽和演算を行う。飽和演算とは有符号/無符号のデータの上限、下限で演算結果を飽和させる演算である。

(4) 論理演算命令

分割された各データ要素に対して論理積、論理和、排他的論理和などの論理演算を行う。

(5) 比較演算命令

2つのレジスタ間で分割された各データ要素を比較する。比較結果は全てのビットが 1、あるいは全てのビットが 0 という形で表現されることが多い。

(6) その他

SIMD 命令を効果的に使うには上記 5 種類の命令だけでは不足であり、上記に分類されないような命令も必要となる。搅拌命令などがその例である。これらの命令はアーキテクチャごとに挙動が異なることが多い。

3. 従来のプログラミング手法

SIMD 命令を効果的に利用するために様々な研究がされている^{6),7)}。現在、主流であるプログラミング手法は以下の 3 つに大別できる。

(1) 数値計算ライブラリなどからの利用

一部の数値計算ライブラリなど⁹⁾では SIMD 命令を用いた高速化が試みられている。これら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。しかし、ライブラリが利用可能な範囲が限られているため、柔軟なプログラミングには向いていない。

(2) コンパイラによる SIMD 命令の自動生成

ソースコードから潜在的な SIMD 命令の適用箇所を見つけたし、SIMD 命令を生成する手法であ

表 2 API の一覧

分類	代表的な API
算術演算	vadd, vsub, vmul, vmadd
飽和演算	vaddst, vsubst
比較演算	vcmpge, vcmpgt, vcmpge, vmax, vmin
論理演算	vand, vor, vnot, vxor
その他	varray, vsum, vavr

```

for (i = 0 ; i < N ; i++){
  for (j = 0 ; j < N ; j++){
    vo.zero(vo.reg1); /*ゼロ初期化*/
    for (k = 0 ; k < N ; k++) {
      /*積和演算
      reg1 = reg1 + matA[j][k]*matB[i][k] */
      vo.reg1 = vo.vadd(vo.reg1, matA[j]*k,
                        matB[i]*k);
    }
    /* ベクトル要素の和 */
    matC[i][j] = vo.vsum(vo.reg1);
  }
}

```

図 2 API を用いた記述例 (行列積)

る^{4),6)}。自動的な適用が期待できるため、プログラマへの負担を低くすることができる。しかし、2 章で述べたように SIMD 命令の生成にはある程度の条件を満たす必要があるが、ソースコード上では SIMD 命令の生成条件が明らかでないため、プログラマがどのように記述すれば SIMD 命令が生成されるかが不明瞭である。そのため、SIMD 命令を効果的に利用するには適さないと考えられる。

また、コンパイラの作成には大きな労力が必要となる。

(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^{4),5)}。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、コードの可搬性は著しく低下し、アーキテクチャやコンパイル環境が限定されてしまう場合が多い。それぞれの特徴を表 1 にまとめる。

4. 共通記述方式

本研究では SIMD 命令向け共通記述方式を提案する。この記述方式は各 SIMD 命令セット上でコードの可搬性を保ったまま明示的に SIMD 命令を用いることを目的とし、インタフェースは関数型の API として提供する。この API は比較的低いレベルで機能を抽象化し、柔軟性を保つよう設計する。2 章を踏ま

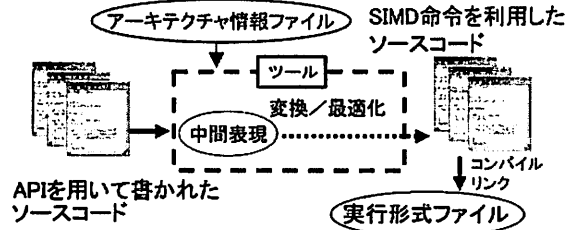


図 3 ツールの動作概要

えた上で API を表 2 のように分類する。

API は C / C++ コード内にプログラマによって記述される。この API を用いたコードは最終的には 1 (スカラ) ~ $N \div (\text{単位データサイズ})$ のいずれかの数のデータが一度に処理される。しかし、命令セットが決定するまではその数は不明である。そのため、プログラマは記述の上ではスカラ演算とみなして記述してよい。しかし、プログラミング時に並列サイズが決定していないと曖昧な記述になってしまう恐れがある。このため、この記述方式では並列サイズにあわせるベクトル変数を提供する。

API を用いて記述した行列積の例を図 2 に示す。

これら API などを用いて記述されたソースコードを各 SIMD 命令セットについて適用を行わなければならない。これはループの変換などを含むため、単純な SIMD 命令の展開だけでは対処することはできない。そのため、ソースコードを読み取り、解析し、SIMD 命令の適用を行わなければならない。5 章でこの適用を行うツールについて述べる。

5. SIMD 命令適用ツールの設計と実装

4 章で述べた API を用いて記述されたソースコードに SIMD 命令を適用するツールについて述べる。このツールは C / C++ の構文を解析し、アーキテクチャの情報をもとに API を SIMD 命令に変換し、それに伴うループの変形や最適化などを行う。変換結果は再び C / C++ のコードに戻され、出力される (図 3)。現在、出力されたコードは GNU C Compiler に対応しており、SIMD 命令は gcc⁸⁾ の拡張インラインアセンブラで出力している。

5.1 アーキテクチャ情報の記述

各アーキテクチャの情報を記述する必要がある。本ツールはこの記述をもとに API を SIMD 命令に変換し、コードを再構築する。ここで記述する情報はレジスタ情報と命令の特性である。命令の特性とはデータ型、並列性、メモリアライメントに関する制限などを

```

(builtin
  (xmm3) ; 出力
  ((xmm1 ((reg xmm) ((float) 128))) ; 使用されるデータ
  (xmm2 ((reg xmm) ((float) 128)))
  (xmm3 ((reg xmm) ((float) 128)))
  (@vmul xmm1 xmm2) ; API フォーマット
  (= xmm3 xmm1) ; 変換ルール
  (asm "mulps @xmm2, @xmm3" xmm3)))

```

図 4 アーキテクチャ情報の記述例

```

input : int *z;
if (z is 16-byte aligned){
  .....
}else {
  .....
}

```

図 5 アライメントの判定を行う条件文生成による
実行コードの選択

指す。

具体例を図 4 に示す。この例は、全体で 128 bit の単精度浮動小数点のデータ列の積を求める API の変換式である。出力とは API の実行結果が出力されるデータである。使用されるデータには、この API で用いられるデータのレジスタセット、データ型、サイズが記述される。API フォーマットは対応する API のフォーマットである。変換ルールには出力されるアセンブラなどが記述される。ここには C/C++ の式や、他の API、中間表現を記述することもできる。

5.2 適用ツールの機能

以下で適用ツールの機能について述べる。

(1) ループベクトル化

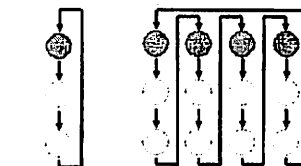
SIMD 命令セットが決定すると、並列で処理されるデータサイズが決定する。ツールはこのサイズをもとにループ内に存在するベクトル演算とスカラー演算を再配置する必要がある。

図 6 にループの再構築と実行オーダーの例を示す。まず、(a) の状態が通常のループの状態である。ループ内の演算をベクトル化するために (b) のようにループ展開を行い API で記述されたベクトル演算を生成する。結果的に (c) のようにベクトル演算と他の演算が再配置され、ループの再構築を行う。

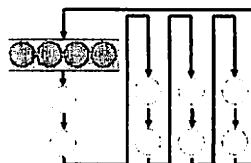
(2) メモリアライメント制御

2 章で述べたようにロードやストアといったメモリ転送を含む命令では対象データのメモリアライメントが条件を満たしていないと実行できないことがある。つまり、対象データのメモリアライメントによって実行する命令を変える必要がある⁷⁾。

近年のコンパイラではコンパイラオプション、あ



(a) 通常のループ (b) 展開されたループ



(c) ベクトル演算を含むループ

図 6 実行オーダーの変化

るいはソースコード内にプラグマや特定のキーワードを付加することで一部のメモリアライメントの保証を行うことができる場合がある。本ツールでは gcc を対象としたキーワードの付加やコンパイラオプションを指定する。

しかし、ライブラリ関数などでメモリを確保した場合、実行時までメモリアライメントの保証を行うことはできない。そのため、実行するコードを動的に変更する必要がある。本ツールでは図 5 のように、実行時に対象となるデータのメモリアライメントを確認するコードを生成し、実行するコードを選択している。

(3) コード最適化

API を SIMD 命令に変換する過程で無駄な転送命令が生じたり、同じ演算が発生する可能性がある。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パフォーマンスに大きく影響する。そのため、ツールはできる限り無駄を省いた、最適なコードを生成する必要がある。本ツールでは以下のような従来のコンパイラで行われているような最適化を行う^{10),11)}。

・冗長式の削除

結果が利用されない式を削除する。

・共通式、共通部分式の削除

計算した式、部分式を再利用し、重複した計算を削除する。

・コピー伝播

代入元の値を代入先にコピー、伝播し必要なレジスタ数を削減する。

(4) レジスタ割付

表 3 評価環境

	CPU	命令セット	OS	gcc のバージョン
1	Xeon 2.80GHz	SSE/SSE2	FedoraCore2	gcc 4.01
2	PowerPC G5 2.30GHz	VMX	MacOS 10	gcc 4.00
3	R5900 V3.1 300MHz	EmotionEngine	PS2 Linux	gcc 2.95

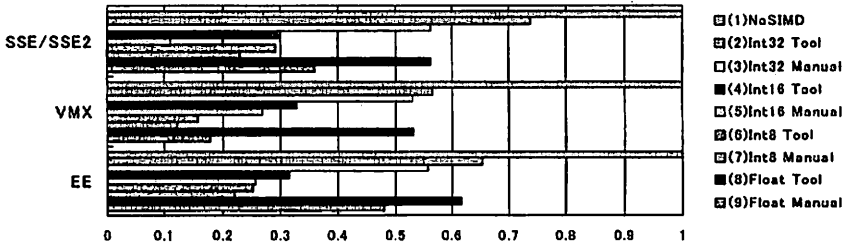


図 7 行列積における実行時間比

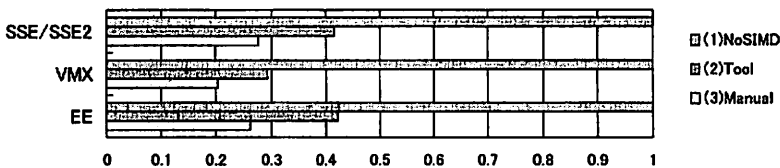


図 8 高速フーリエ変換における実行時間比

SIMD 命令も他の命令と同様にレジスタを用いて演算を行う。高速化のためにはメモリアクセスの回数が少なくなるようにレジスタを割り当てなければならない。しかし、利用可能なレジスタの数には制限があり、その数はアーキテクチャによって異なる。

コード生成以前の中間表現の段階ではすべてのレジスタは対象アーキテクチャに存在するレジスタと同じサイズの仮想レジスタが割り当てられている。これを実レジスタに置き換えるために、古典的な手法であるグラフカラーリングアルゴリズムを用いる。これは各仮想レジスタの生存範囲から干渉グラフを作り、このグラフを利用可能なレジスタの数の色で塗り分けることでレジスタ割り当てを行うものである。もし、利用可能なレジスタ数以上のデータを保持する必要がある場合はメモリ上に保持する。

2 章で述べたように、SIMD 命令では専用のレジスタが利用されることが多い。その場合、他の逐次コードをそれほど乱すことなくレジスタを割り当てることができる。もし、専用のレジスタが無く、汎用のレジスタを利用しなければならない場合はコンパイラとの連携が重要となってくる。本研究で用いている gcc の拡張インラインアセンブラ⁸⁾では

利用したレジスタと破壊したレジスタをコンパイラに伝えることが可能である。そのため、適切なコードを生成することができればインラインアセンブラによるコンパイラのコード最適化の弊害はある程度抑えることが可能である。

6. 提案方式における実行性能、コードの可搬性の評価

ここでは 4 章で提案した API を用いて行列積、高速フーリエ変換をプログラミングし、コードの実行性能、およびコードの可搬性の評価を行う。行列積の対象データは 8, 16, 32 bit 整数および単精度浮動小数点、高速フーリエ変換の対象データは単精度浮動小数点を選択した。対象となる SIMD 命令セットは Intel の SSE/SSE2, IBM/モトローラの VMX, ソニー/東芝の EmotionEngine を選択した。評価は SIMD 命令を用いていないコード、ツールによって SIMD 命令を適用したコード、人手で SIMD 命令を用いて最適化したコードの 3 種類を比較することで行う。

6.1 評価環境

評価を行った環境を表 3 に示す。コンパイラは全て gcc を用いている。ツールには図 4 のような各 SIMD 命令セットの記述を与え、ソースコードは API を用

いて記述した同一の図 2 のような ソースコードを与える。ただし、EmotionEngine では単精度浮動小数点のベクトル演算器を用いるために必要なデバイス制御の記述 (3 行程度) を別途記述している。

6.2 実行性能評価

行列積の実行結果を図 7 に示す。このグラフは SIMD 命令を用いていないコードの実行時間を 1 とした時の実行時間比であらわしている。

提案手法の速度向上は人手による最適化の平均 73 % 程度であった。いずれも人手で最適化を行ったコードには及ばないが、ある程度の速度向上は見られる。人手のコードに劣る主な理由は最適化が十分でないことが大きな要因だと考えられる。特にレジスタを最大限活用するループ展開が十分でなかった。

同様に高速フーリエ変換の実行結果を図 8 に示す。

提案手法の速度向上は人手による最適化の平均 68 % 程度であった。行列積と同様に人手で最適化を行ったコードには及ばなかった。こちらも最適化が十分でないことが大きな要因だと考えられる。

6.3 可搬性評価

行列積、高速フーリエ変換で使われた API に対するアーキテクチャの設定記述の記述量は平均で 750 行程度であった。これは新たな自動ベクタライズコンパイラの作成に必要な記述量と比較するとはるかに少ない。

また、人手で SIMD 命令を記述する場合、命令セットごとに数十行程度の新たなコードなどを記述する必要があるが、API を用いて記述したコードは命令セットごとに新たなコードを記述する必要はない。

提案手法と人手による最適化を命令セット別に比較すると、最も速度向上に差が出たのが、行列積で VMX の 69%、高速フーリエ変換で EmotionEngine の 62 % であった。6.2 で述べたように、行列積、高速フーリエ変換の平均の速度向上比はそれぞれ 73 %、68 % であるので、命令セットの違いにより大きく差が開いたとはいえない。このことから、性能可搬性があるといえる。

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、この記述方式を用いて記述したソースコードに SIMD 命令を適用するツールを開発した。また、行列積、高速フーリエ変換についてこの記述方式を用いてプログラミングし、評価を行った。結果として、人手による最適化ほどの速度向上を得ることはできなかったが、ある程度の速度向上を確認することができた。

また、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、提案方式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た。生成されたコードの性能も命令セットによって人手による最適化と大きく差が生じることはなかった。このことから SIMD 命令を用いたコードとしての性能可搬性を保つことができたと言える。

以下に今後の課題を述べる。

- (1) 人手による最適化に比べ、速度が劣っているのはツールの最適化能力が十分でないと考えられる。よって、最適化能力を強化する必要がある。
- (2) 提案した API が他のアプリケーションや他の命令セットで十分な効果が発揮できるかを確認する必要がある。

参考文献

- 1) Intel Corporation : IA-32 Intel(R) Architecture Software Developer's Manuals.
<http://www.intel.com/design/pentium4/manuals/>
- 2) IBM Corporation : PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual.
<http://www-306.ibm.com/chips/>
- 3) Sony Computer Entertainment, Toshiba Corporation : EE User's Manual.
- 4) Intel Corporation : Intel C++ Compiler Documentation,
https://hpc.msu.edu/doc/icc/main_cls/index.htm
- 5) Apple Corporation : AltiVec Programming
<http://developer.apple.com/hardware/drivers/ve/>
- 6) Aart J. C. Bik, Milind Girkar, Paul M. Grey, Xinmin Tian: Automatic Intra-Register Vectorization for the Intel Architecture, International Journal of Parallel Programming, Volume 30, Issue 2, pp65-98 (2002).
- 7) Peng Wu, Alexandre E. Eichenberger, Amy Wang : Efficient SIMD Code Generation for Runtime Alignment and Length Conversion, CGO '05 pp153 - 164 (2005).
- 8) Free Software Foundation : GCC
<http://gcc.gnu.org/>
- 9) R.Clint Whaley, Antoine Petitet, Jack J.Donggara : ATLAS Project
<http://math-atlas.sourceforge.net/>
- 10) Aho, A. V., Sethi, R., Ullman, J. D. : Compilers - Principles, Techniques, and Tools, Addison-Wesley(1986).
- 11) Steven S. Muchnick : Advanced Compiler Design & Implementation(1998)