Vol.2017-HPC-160 No.10
2017/7/26

アプリに特化したSIMD最適化のためのOpenMP仕様拡張の提案とARM SVEを用いた評価

李 珍泌[1,a)]　Francesco Petrogalli[2]　Graham Hunter[2]　佐藤 三久[1]

概要：近年の高性能計算向けプロセッサアーキテクチャにおいてSIMD命令による並列性が増加していく傾向があり、SIMD命令を用いることが対象アーキテクチャの性能を引き出すために必要不可欠になっている。最新のOpenMP言語仕様に導入されたSIMD指示文を用いることによってループ文のベクトル化を助ける情報をコンパイラに与えることができる。しかし、SIMD命令の生成はコンパイラの実装に依存するため、プログラマが命令生成を制御し、最適化を行うことは困難である。本研究では対象アーキテクチャに特化して関数のSIMD命令レベル最適化を行い、OpenMPのSIMDプログラミングモデルで利用するための拡張構文を提案する。プログラマはintrinsic関数などを用いて対象アーキテクチャに特化した最適化を行う。コンパイラがベクトル化するループ文の中で関数呼び出しを見つけた時にそれを自動ベクトル化するのではなく、プログラマが用意したSIMD命令実装に置き換えることによってプログラマが最適と考えるSIMD並列化を実現する。提案手法によってループ文のイテレーションの変換はスレッド並列化と同様、コンパイラによって行われるが、SIMD命令の生成はプログラマが制御することが可能になる。画像処理のコードを用いた性能評価ではコンパイラによる自動ベクトル化コードと比べてプログラマが明示的に与えたSIMD命令実装を用いることで実行命令数を最大70%削減していることで提案手法の有効性を示した。

1. はじめに

近年の高性能プロセッサアーキテクチャはコア数の増加やSIMD命令の幅の拡大によって並列性を増やすことで性能を上げている。プロセッサの電力性能比を改善するためにこのようなトレンドが今後しばらく続くと予想される。最新のIntel社のXeon Phiアーキテクチャは512ビット幅のSIMD命令セットAdvanced Vector eXtensions (AVX-512)をサポートし、60個以上のコアを持つ。ARM社は高性能計算のために設計された新しいSIMD命令セットScalable Vector Extension (SVE)[1]をリリースしている。このようなアーキテクチャを使いこなすためにSIMDレベル並列性を記述するプログラミングモデルが求められている。

OpenMP (OMP)は共有メモリシステム上のスレッドレベル並列化を記述することでコア間の並列実行を行う。*parallel*や*for*指示文はメニーコアアーキテクチャのスレッド並列化にも有効である。その一方で従来の手法ではSIMD命令の活用はコンパイラによる自動ベクトル化によって行われる。コンパイラはループ文のようなプログラムの構造を分析し、イテレーションの間の並列性を見つけることでSIMD命令を生成する。OMP 4.0からSIMD並列化を記述する指示文が導入された。*simd*指示文は対象のループ文がベクトル化可能であることを示す(以後、SIMDループ文)。*declare simd*指示文はSIMDループ文の中で呼ばれる関数に対して指定することができ、対象関数をベクトル化するための情報をコンパイラに与える。これらの指示文は対象とするプログラム構造が安全にベクトル化可能であることを保証するとともに、メモリポインタのalignmentなど、コンパイラの解析を助ける情報を与える。

OMP指示文の情報はベクトル化のためのコンパイラの解析を助け、従来情報不足でベクトル化できなかったコードを扱うことが可能になる。しかし、SIMD命令の選択とコードの生成はコンパイラの実装に依存するため、プログラマが意図したものとは異なるSIMDコードが生成された場合、それを変更・最適化することは困難である。本研究ではプログラマがループ文の中の関数のSIMD命令実装(以後、SIMD関数)をintrinsic関数などの明示的な手段で記述し、OMPのSIMDプログラミングモデルで利用するためのインターフェイスを提案する。そのために、OMP言

[1] 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science
[2] ARM Ltd.
[a)] jinpil.lee@riken.jp

語仕様に新しい指示文 *alias simd* を導入する。SIMD ループ文の中で用いられる関数 (スカラ関数) をプログラマが intrinsic 関数などの SIMD プログラミングモデルによって実装したとする (SIMD 関数)。*alias simd* 指示文はコンパイラのベクトル化処理の中でスカラ関数とプログラマによって与えられた SIMD 関数を関連付ける。*simd* 指示文によってループ文が並列化されるときに、スカラ関数に対応する SIMD 関数が *alias simd* 指示文によって与えられているときにはコンパイラによるベクトル化を行わず、関連付けられた SIMD 関数に置き換えられる。このインターフェイスを使うことによって、ループ文のイテレーションの変換と内部の SIMD 命令生成を区別して行うことができる。ループのイテレーションの変換は *simd* 指示文から与えられた情報からコンパイラが最適なコード変換を行うことができる。しかし、ループ文の中の計算部分のベクトル化は膨大な命令の組み合わせから最適な命令生成を行うことが困難であるため、プログラマが明示的に最適な実装を与えることによって性能向上を得られると考えられる。

提案手法は特定のアーキテクチャに特化したものではないが、本研究では ARM 社の SVE 命令セットをターゲットに検討を行う。SVE は高性能計算のために新しく導入された命令セットであり、固定の SIMD 幅を仮定しない (Vector Length Agnostic、VLA) ことが特徴である。そのために多くの命令がマスク (predicate mask) 付きで実行される。本研究の提案手法は SVE の VLA アプローチを前提とした環境だけでなく、Intel 社の AVX のような固定ベクトル幅を持つ従来の命令セットにも適用できるように検討を行った。これらの命令セットが持つ Application Binary Interface (ABI) と互換性を保つ限り、SIMD 関数の実装はどのようなプログラミングモデルを用いても構わない。本研究では対象アーキテクチャの intrinsic 関数を用いて SIMD 関数の実装を行った。

本稿の構成は次のようである。第 2 章では関連研究を挙げ、本研究との違いを述べる。第 3 章では予備知識として SVE 命令セットの概要や intrinsic 関数について述べる。第 4 章では提案手法である *alias simd* 指示文の言語構文とプログラミングモデルについて述べる。第 5 章では画像合成を行うコードを用いて自動ベクトル化されたコードとプログラマによって与えられた実装の性能を比較する。第 6 章では結論と今後の課題について述べる。

2. 関連研究

明示的な SIMD プログラミングを実現するために様々な手法やプログラミングモデルが提案されている [2]。ARM C Language Extensions (ACLE)[3] は ARM SVE 命令セットの C/C++言語向け intrinsic プログラミングモデルであり、データ型にジェネリックなインターフェイスを提供する。ispc (Intel SPMD Program Compiler)[4] は独自の構文で SIMD レベル並列性を記述するプログラミング言語を実装し、SSE、AVX、AVX-512 などの Intel の SIMD 命令の生成を行う。ispc や Intel array notation[5]、Sierra[6]、Terra[7] などの独自言語を用いた SIMD 並列化プログラミングモデルはコンパイラによる言語構文のサポートが必要である。Boost SIMD[8] は C++テンプレート機能を用いて実現されているため、コンパイラによる構文解析を必要としない。Cyme[9] と Vc[10] はライブラリ形式で実現された SIMD プログラミングモデルである。これらのプログラミングモデルはアーキテクチャに依存する SIMD 命令プログラミングを言語機能やライブラリなどの手法で抽象化している。しかし、特定のベクトル幅のデータ型をサポートし、逐次コードからの移植やベクトル幅の変更にはプログラムの修正が必要である。

3. Scalable Vector Extension の概要

SVE は ARMv8-A アーキテクチャの A64 命令セット向けの新しい SIMD 命令拡張である。将来のハードウェア SIMD 幅の増加をソフトウェアの変更 (再コンパイル) なしで吸収することを目標として設計されたものである。SVE を実装するプロセッサは 128 ビットから最大 2048 ビットまでのベクトルレジスタを持つ（最大ビット数はプロセッサの実装依存、128 ビットの倍数のみサポート)。特定のベクトル幅を仮定しない (VLA) 命令セットを実現するために多くの命令が predicate マスク付きで実行される。

以下に SVE 命令セットの主な特徴を示す：

- 32 個のベクトルレジスタ (Z0-Z31).
- 16 個の predicate マスクレジスタ (P0-P15).
- 動的にベクトル幅を変更可能: 128 から 2048 ビット (利用可能なベクトル幅はプロセッサの実装依存)
- VLA プログラミングモデルを実現 - 同じバイナリを異なるベクトル幅を持つプロセッサで再コンパイルなしで実行することができる

3.1 SVE による VLA プログラミングモデル

リスト 1 に C 言語と SVE アセンブリで記述された配列の足し算のコードを示す。オペランド `p0` は現在のループイテレーションで SIMD 命令が操作を行う SIMD lane の位置をあらわす predicate マスクレジスタである。

図 1 にリスト 1 のループイテレーションが SVE 命令によって制御されるときのレジスタの値の変化を示す。配列 `A`、`B`、`C` のデータ型は `double *`であり、`i` と `N` は `unsigned long int` データ型を持つ。

ループのイテレーションは *whilelo* 命令によって生成される predicate マスクレジスタの値で制御される。図の例では `N` は 12 である。*incd* 命令は現在設定されたベクトル幅を用いてイテレーション変数の値を増加させるため、異なるベクトル幅を持つプロセッサでもバイナリーの修正な

```
; for (i = 0; i < N; i++) { C[i] = A[i] + B[i]; }
; x9,x10,x11 and x12 hold N,A,B, and C, respectively
    mov     x8, xzr
    b       .Lcond
.loop:
    ld1d    z0.d, p0/z, [x10, x8, lsl #3]
    ld1d    z1.d, p0/z, [x11, x8, lsl #3]
    fadd    z0.d, z0.d, z1.d
    st1d    z0.d, p0, [x12, x8, lsl #3]
    incd    x8          ; increase i
.Lcond:
    whilelo p0.d, x8, x9; set p0.d[i] = (i < N)
    b.first .loop       ; execute the loop iteration
                        ; if the first lane is active
```

Listing 1: SVE による配列の足し算

```
unsigned long i = 0;
svbool_t p = svwhilelt_b64_s64(i, N);
svbool_t tp = svptrue_b64();
while (svptest_first(tp, p)) {
  svfloat64_t vec_a = svld1(p, &(A[i]));
  svfloat64_t vec_b = svld1(p, &(B[i]));
  svfloat64_t vec_c = svadd(p, vec_a, vec_b);
  svst1(p, &(C[i]), vec_c);
  i += svcntd();
  p = svwhilelt_b64_s64(i, N);
}
```

Listing 2: ACLE による配列の足し算

256-bit SVE

Iter	x8 (i)	whilelo p0.d, x8, x9 ($i < N$)
0	0	1|1|1|1
1	4	1|1|1|1
2	8	1|1|1|1

384-bit SVE

Iter	x8 (i)	whilelo p0.d, x8, x9 ($i < N$)
0	0	1|1|1|1|1|1
1	6	1|1|1|1|1|1

512-bit SVE

Iter	x8 (i)	whilelo p0.d, x8, x9 ($i < N$)
0	0	1|1|1|1|1|1|1|1
1	8	1|1|1|1|0|0|0|0

図 1　*whilelo* 命令による predicate マスクの値の生成

しでプログラムが動作する。

最初にループ変数 i (アセンブリの x8) を 0 にセットしたあと、*whilelo* 命令に分岐して i の値とループの最後のイテレーションである N (アセンブリの x9) の値を比較する。*whilelo* 命令は比較結果によって predicate マスクレジスタの値をセットする。predicate マスクレジスタの各要素は対応するベクトルレジスタの SIMD lane が有効であるかどうかをあらわす。式 (i < N) ? 1 : 0 を評価し、その結果を SIMD 命令の中の個別のスカライテレーションに対応する p0.d[i] にセットする。

実行するイテレーションが残っていると predicate マスクレジスタの最初の lane から有効化されるので *b.first* 命令によって predicate マスクレジスタの最初の要素が有効である場合、次のイテレーションが実行される。

predicate マスクレジスタはループ文の中の演算にも用いられる。リスト 1 のメモリロード命令 (*ld1d*) やストア命令 (*st1d*) に predicate マスクレジスタを用いることで有効なイテレーションだけがメモリを参照する。このような操作によって固定ベクトル長を持つ命令セットであらわれる scalar tail loop (ベクトル幅より少ない残りのイテレーションをスカラ命令で処理するループ) を必要とせず、SIMD 命令のみでループの実行が可能になる。

ループ変数 i の値は *incd* 命令によって現在の実行ベクトル幅で表現できる double 要素の数だけ増加する。そのあと *whilelo* 命令が実行され、次のイテレーションの有効 SIMD lane をセットし、再度 *b.first* 命令による predicate マスクレジスタの判定を行う[*1]。

3.2 Intrinsic プログラミングインターフェイス

他の SIMD 命令セットと同様、SVE も intrinsic プログラミングインターフェイスを提供する。ARM C Language Extensions (ACLE) は C と C++言語で SVE 命令を利用するためのプログラミングインターフェイスである。リスト 2 はリスト 1 と同等のプログラムを ACLE で記述したものである。SVE の VLA プログラミングモデルにより、*for* 文は predicate マスクの判定を行う *while* 文に置き換わる。*svbool_t* は predicate マスクをあらわすベクトル型であり、*svfloat64_t* は double 型の要素を持つベクトルデータ型である。

predicate マスクの値はリスト 1 と同様、*svwhilelt_b64_s64()* 関数によって生成される。*svptrue_b64()* 関数はすべての SIMD lane が有効な predicate マスクを生成する。各イテレーションの最初に *svptest_first()* 関数を実行し、イテレーションの実行判定を行う。

svld1() や *svst1()*、*svadd()* 関数はリスト 1 のメモリ命令とベクトル演算と同等である。*svcntd()* 関数は現在のベクトル幅による double 型要素の数を返す。ACLE の関数は C++のテンプレートや C11 の *_Generic* で実現されているため、異なるデータ型に対しても同じ関数名を持つジェネリックなインターフェイスを提供する。

4. SIMD 命令最適化のための OpenMP 拡張仕様の提案

第 1 章で述べたように現在の OMP 言語仕様では SIMD

[*1] SVE を用いた VLA プログラミングモデルについて、[11] のホワイトペーパーで様々なサンプルコードが紹介されている。

```
1  #pragma omp declare simd notinbranch          // A
2  int add(int a, int b) {
3    return a + b;
4  }
5  #pragma omp alias simd to(add) simdlen(4)     // B
6  int4_t add_vec4(int4_t a, int4_t b) {
7    return intrinsic_add4(a, b);
8  }
9  #pragma omp alias simd to(add) simdlen(8)     // C
10 int8_t add_vec8(int8_t a, int8_t b) {
11   return intrinsic_add8(a, b);
12 }
13
14 #pragma omp simd simdlen(VL)  // VL is 4 or 8
15 for (i = 0; i < n; i++) {
16   z[i] = add(x[i], y[i]);
17 }
```

Listing 3: *alias simd* 指示文のサンプルコード

ループ文の変換時に命令選択を明示的に行うことはできない。従来のコンパイラの解析では得られなかった情報を指示文によって与えることで SIMD 並列化を可能にするが、SIMD 命令の生成はコンパイラの実装によって行われるため、プログラマが制御することはできない。本章では SIMD ループ文の中で用いられる関数に対応する SIMD 関数を明示的に与える OMP 拡張仕様を提案する。

4.1 提案手法の概要

従来の OMP 言語モデルでは SIMD ループ文の内部の関数の SIMD 命令生成は inline 展開の後に自動ベクトル化するか *declare simd* 指示文を用いて並列化情報を与えることで行われる。提案手法ではコンパイラによるベクトル化を行わず、プログラマがあらかじめ対象関数の SIMD 並列化を行い、*simd* 指示文の変換時に置き換えるように指定する。そのため、プログラミングモデルは *declare simd* 指示文に類似したものになる。*declare simd* 指示文がスカラ関数を対象にして、SIMD 並列化のための情報を与えるものだとすれば、提案手法は対象スカラ関数と関連付けられる (外部にあらかじめ宣言された)SIMD 関数の名前と引数を与えるものである。

以上のプログラミングモデルを実現するために、OMP 言語仕様に新しい指示文 *alias simd* を追加することを提案する。リスト 3 に *alias simd* 指示文のサンプルコードを示す。ベクトルデータ型 (*int4_t*, *int8_t*) と intrinsic 関数 (e.g. *intrinsic_add4()*) は pseudo code である。

alias simd 指示文は特定アーキテクチャ向けに実装された SIMD 関数と逐次コードの中のスカラ関数を関連付ける。リスト 3 の `B` と `C` は逐次コードの *add()* 関数をベクトル型 *int4_t* と *int8_t* を用いて SIMD 並列化したものである。*alias simd* 指示文は *add_vec4()* 関数と *add_vec8()* 関数が *add()* 関数に対応する SIMD 関数であり、スカラ関数とベクトル関数の引数がお互いに ABI に定められた法則によって対応していることを保証する。関数の名前と引数のデータ型を指定する *to* 節と SIMD 関数が用いる命令の SIMD 要素数をあらわす *simdlen* 節を与えることで、SIMD 命令生成時にコンパイラがスカラ関数にマッチングする SIMD 関数を選択できるようになる。現在の言語仕様では同じベクトル幅と名前を持つ SIMD 関数が異なるアーキテクチャ向けに実装されている状況を想定しないため、実装がベンダー毎にマクロ宣言 (e.g. *__AVX__*, *__ARM_NEON__*) によってガードされていると仮定する。

alias simd 指示文による関数の関連付けは *declare simd* 指示文の変換とは独立に行われるため、*declare simd* 指示文によって指定された関数はコンパイラによってベクトル化される。*simd* 指示文による SIMD ループ文の変換時には *alias simd* 指示文で指定された SIMD 関数が優先して選択される。対象ループ文が SIMD 並列化できない場合やマッチングするベクトル幅の SIMD 実装が存在しない可能性があるため、対象関数の逐次コード実装や *declare simd* 指示文の指定は必要である。

図 2 にコンパイラによる SIMD ループ文の並列化と *alias simd* 指示文のコード変換の流れを示す。*simd* 指示文によって対象ループ文がベクトル幅 256 ビット (8 個の integer) でベクトル化されることが指定される。SIMD ループ文のコード変換のために *add()* 関数の SIMD 並列化を行わなければならない。そのために逐次コードの *add()* 関数に *declare simd* 指示文が記述される。*simdlen* 節が与えられていないので、コンパイラは状況に応じて様々なベクトル幅の SIMD 関数を生成する。この例ではベクトル幅 256 ビット (8 個の integer) の SIMD 関数 *add_I8I8I()* が生成される。

その一方で (pseudo) intrinsic 関数による *add_vec8()* 関数が存在するとする。*alias simd* 指示文の *to* 節によって *add_vec8()* 関数を *add()* 関数に関連付ける。多くのアーキテクチャではベクトル幅や命令セット毎に専用の intrinsic 関数が用意されているため、複数の SIMD 関数が同じ関数に関連付けられる。逐次コードの関数に関連付けられた様々な SIMD 関数から対象ループ文のベクトル化にマッチングするものを見つけるために *simdlen* 節を用いて対象 SIMD 関数のベクトル幅を記述する。

コンパイラが図 2 の SIMD ループ文を変換する時、二つの SIMD 関数 (自動ベクトル化された *add_I8I8I()* と intrinsic 関数による *add_vec8()*) が選択可能である。本研究の提案手法ではプログラマによって与えらえる SIMD 関数を優先的に選択し、SIMD 命令を明示的に活用する手段を提供する。

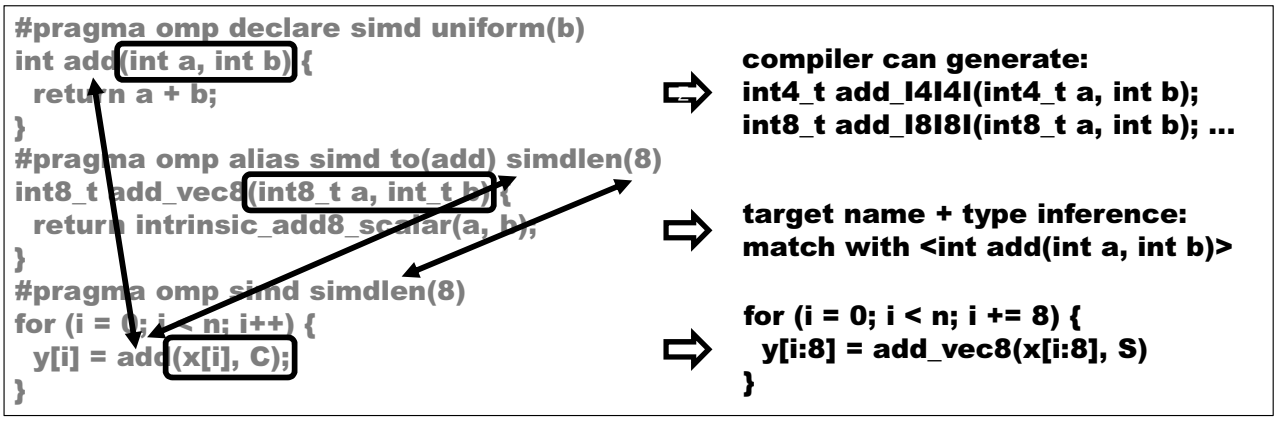

図 2　コード変換の概要

```
#pragma omp alias simd to(name_or_decl ) [clause_list ]
function_definition

name_or_decl := function_name
              | function_declaration
clause := simdlen(integer_expr )
        | inbranch
        | notinbranch
        | linear(linear_list  [: linear_step ])
```

図 3　Alias Simd 指示文の構文規則

4.2　Alias Simd 指示文の構文

図 3 に *alias simd* 指示文の構文規則を示す。指示文は対象 SIMD 関数の完全な定義とともに与えられる。対象命令アーキテクチャのベクトル ABI (ベクトル幅とデータ型) と互換性を持つ限り、プログラミングモデルに制限はない。最初に固定ベクトル幅を持つ従来の SIMD 命令セットを前提に説明を行い、提案手法の言語構文が SVE の VLA アプローチに対応することを述べる。

to 節には関連付けを行う逐次コードの関数の名前か、必要であれば引数のリストを含む関数宣言を与える。逐次関数の名前だけが与えられた場合、コンパイラがベクトル ABI で定義された (関数呼び出しにおける) ベクトルデータ型とスカラデータ型の対応関係から元の関数宣言を推測する。しかし、C++のテンプレートや C11 の *_Generic* によるジェネリックプログラミングモデルを用いることによって同じ名前の関数が複数の宣言を持つ可能性がある場合は完全な関数宣言を与えなければならない。*to* 節は省略不可である。

simdlen 節は対象 SIMD 関数が用いる命令のベクトル幅を記述する。*simd* 指示文の *simdlen* 節やコンパイラが扱う SIMD 命令セットによって SIMD ループ文が利用する命令のベクトル幅が異なるため、正しい SIMD 関数を選択するために SIMD 関数に対応するベクトル幅を *simdlen* 節で記述する。*simdlen* 節が省略された場合、コンパイラが扱う SIMD 命令セットのベクトル幅が指定される。SVE の VLA アプローチは特定のベクトル幅を仮定しないため、*simdlen* 節を指定する必要がない。しかし、SVE を用いる場合でも特定のベクトル幅に特化した SIMD 実装を行うことは可能である。そのような SIMD 関数は *simdlen* 節を用いてベクトル幅を明示的に記述する。

inbranch/*notinbranch* 節は対象 SIMD 関数が条件文の中で用いられるかどうかを記述する。*inbranch* 節が与えられた場合、条件文の判定によってイテレーションの実行が決定する。このような処理を SIMD 命令で行うため、SIMD lane のベクトル要素の実行条件をマスクレジスタに格納し、SIMD 関数の引数として渡す必要がある。このように多くの SIMD 命令の ABI では *inbranch*/*notinbranch* で SIMD 関数の引数が異なるため、それらに対応する SIMD 関数の実装を与えなければならない。しかし、SVE は条件文の処理のみならず、ループ文の制御にも predicate マスクを用いるため、SIMD 関数は常に第一引数として predicate マスクを受け取る。predicate マスクは実行コンテキストとしてループ変数による有効なイテレーションと条件文の判定結果を同時に含む。したがって、SVE を対象にする場合は *inbranch*/*notinbranch* 節によって異なる SIMD 関数を与える必要はない。第一引数の predicate マスクはスカラ関数の宣言には含まれないため、スカラ型の推測にはカウントされない。

linear 節は SIMD 関数の引数が SIMD lane 毎に変化 (step value) する場合に指定する。step value にかかわらず、対象引数は元の (スカラ) データ型を持つ。ベクトルレジスタの SIMD lane で値を展開し、step value によって変化させる処理は SIMD 関数の内部で行われる (関数の中で *linear_step* の値が定数としてあらわれるなど)。したがって、*linear* 節の値に依存する複数の SIMD 関数が存在する。*linear_list* と *linear_step* の構文規則と意味は *declare simd* 指示文のものと同じである。

5.　性能評価

本章では画像処理を行うコードを用いて提案手法の性能評価を行う。アルファマスクを用いて画像合成を行うコードを ACLE intrinsic 関数で実装し、*alias simd* 指示文によ

```
typedef unsigned char uchar;
typedef unsigned short ushort;

#pragma omp declare simd
uchar add_filter(uchar a2, uchar in1, uchar in2) {
  if (a2 > 0) {
    ushort temp = (ushort)in1 + (ushort)in2;
    if (temp > 255) return 255;
    else return (uchar)temp;
  }
  else return in1;
}

uchar out_r[N]; uchar out_g[N]; uchar out_b[N];
uchar in1_a[N]; uchar in2_a[N];
uchar in1_r[N]; uchar in1_g[N]; uchar in1_b[N];
uchar in2_r[N]; uchar in2_g[N]; uchar in2_b[N];

void loop() {
#pragma omp simd
  for (int i = 0; i < N; i++) {
    out_r[i] = add_filter(in2_a[i],in1_r[i],in2_r[i]);
    out_g[i] = add_filter(in2_a[i],in1_g[i],in2_g[i]);
    out_b[i] = add_filter(in2_a[i],in1_b[i],in2_b[i]);
  }}
```

Listing 4: 画像合成カーネルの逐次コード

る関連付けを行ったコードと、コンパイラによる自動ベクトル化コードの比較を行った。本研究ではまだ提案手法をコンパイラによって実現しておらず、自動ベクトル化されたコードと *alias simd* 指示文の変換を考慮してハンドコンパイルした ACLE コードを比較している。逐次コードと ACLE コードともに ARM 社から提供された LLVM コンパイラによってコンパイルし、命令シミュレータによる実行命令数の比較を行った。

5.1 画像合成カーネルの SIMD 並列化

リスト 4 に画像合成カーネルの計算ループ文を示す。逐次コードでは各画素が *unsigned char* データ型 (0 から 255 の値) で処理される。イメージは *red*、*green*、*blue*、*alpha* のチャンネルを持つ。ループの中で *add_filter()* 関数によって画素の各チャンネルの値の合成を行う。

アルファマスクの値が有効 (0 より大きい値) であれば *add_filter()* 関数は二つの入力イメージの画素を合成する。アルファマスクの値が無効 (0) であれば合成を行わず、入力 `in1` の値を返す。合成の結果がデータ型の最大値を超えて overflow する可能性があるため、より大きいデータ型 (*unsigned short*) の変数にコピーして境界 (255) のチェックを行う (clamping)。したがって逐次コードはデータ型の変換と条件分岐を含む。

リスト 5 に ACLE で実装した画像合成カーネルのコードを示す。第 3 章で述べたように、SVE の SIMD 関数は条件

```
#pragma omp alias simd to(add_filter)
svuint8_t add_filter_acle(svbool_t p, svuint8_t a2,
  svuint8_t in1, svuint8_t in2) {
  svuint8_t zero = svdup_n_u8_x(p, 0);
  // a2 > zero
  svbool_t alpha_mask = svcmpgt_u8(p, a2, zero);
  svuint8_t temp = svand_u8_z(alpha_mask, in2, in2);
  return svqadd_u8(in1, temp);
}
```

Listing 5: ACLE による画像合成カーネルの実装と *Alias Simd* 指示文の記述

文の有無にかかわらず、ループ文の制御にも predicate マスクを用いるため、第一引数に predicate マスクを持つ (引数 p)。*alias simd* 指示文を用いて SIMD 関数が *add_filter()* 関数と関連付けられることを記述している。これによってコンパイラはリスト 4 の SIMD ループ文を変換するときに *add_filter()* 関数を *add_filter_acle()* 関数に置き換える。

ACLE 実装は画素の合成に *svqadd_u8()* 関数を用いる。*svqadd_u8()* 関数は *unsigned char* 変数の足し算を行うときに overflow が発生すると計算結果をデータ型の最大値 (255) に調整 (clamping) する。これによって逐次コードで行われた claming のための分岐処理やデータ型の変換を回避することができる。アルファマスクの判定のための分岐処理も predicate マスクを用いることで SIMD 並列化する。*svcmpgt_u8()* 関数でアルファマスクの比較演算を SIMD 化し、有効な SIMD lane だけ足し算を行う。このように SVE 命令を用いることで 8 ビットデータ型の SIMD 演算のみで関数を実装することができる。逐次コードは演算スループットを低下させるデータ型の拡張や、ベクトル化を阻害する分岐処理を含むため、ACLE 実装を SIMD ループ文の中で利用することで計算性能が改善できると考えらえる。

リスト 6 に ACLE で実装されたループ文のコードを示す。ACLE がジェネリックなプログラミングモデルを提供し、VLA アプローチによって異なるベクトル幅でも同じコードを利用することが可能である。しかし、図 6 からわかるようにループの制御は逐次コードからの変更が必要であり、他の命令セットとの互換性が保たれない。リスト 4 では *simd* 指示文によって図 6 と同等 (*add_filter()* の SIMD 実装以外) のコードを得ている。本研究の提案手法はループ制御の変換を OMP のプログラミングモデルによってコンパイラに任せ、計算部分を SIMD 関数を指定することで最適化するアプローチである。このような組み合わせによってコードのポータビリティと性能最適化を両立させている。

5.2 評価結果

図 4 にハンドコンパイルした ACLE 実装 (ACLE) と自

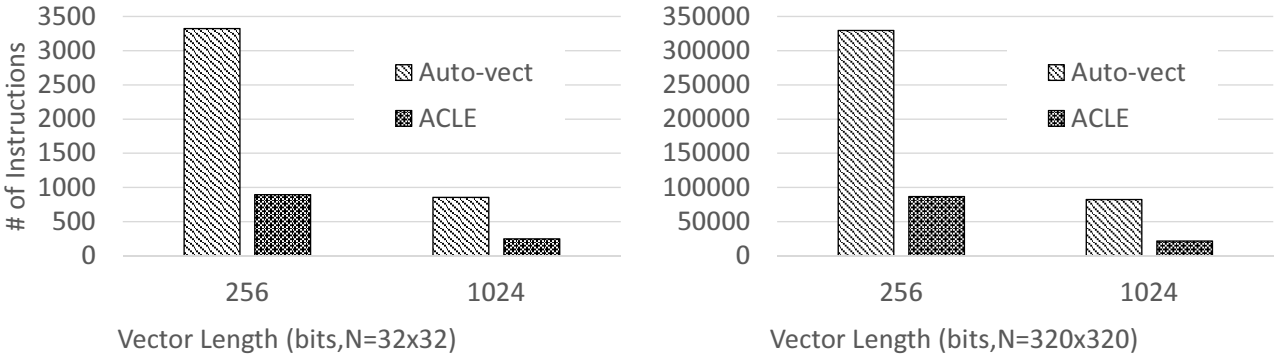

図 4　画像合成カーネルの評価結果

```
void loop() {
  int i = 0;
  svbool_t p = svwhilelt_b8_s32(i, N);
  svbool_t tp = svptrue_b32();
  while (svptest_first(tp, p)) {
    svuint8_t vin1_r = svld1_u8(p, in1_r+i);
    // loads for vin1_{g, b}, vin2_{a, r, g, b}
    svuint8_t vout_r =
      add_filter_acle(p,vin2_a,vin1_r,vin2_r);
    svuint8_t vout_g = add_filter_acle(p, ...);
    svuint8_t vout_b = add_filter_acle(p, ...);
    svst1_u8(p, out_r+i, vout_r);
    // stores for vout_{g, b}

    i += svcntb();
    p = svwhilelt_b8_s32(i, N); }}
```

Listing 6: ACLE によるループ文の制御

動ベクトル化されたコード (Auto-vect) の性能を示す。本研究では特定のハードウェア実装を仮定せず、シミュレータによるループ文の実行命令数の比較を行った。データサイズは 32×32、320×320 ピクセルを用意し、256 ビットと 1024 ビットベクトルレジスタの環境で評価を行った。

評価の結果、自動ベクトル化されたコードが *alias simd* 指示文の変換を考慮してハンドコンパイルしたコードと比べて最大 3.8 倍多い命令を実行している。小さいデータセットでベクトル幅を 1024 ビットに増加させると比率は 3.4 倍に減少する。これは画素データが少ないベクトル命令で処理されるため、ループ制御などの逐次実行部分の比率が増加するためである。

逐次コードは *unsigned char* から *unsigned short* データ型への型変換を行う。そのため、コンパイラは画素合成の足し算を *unsigned short* データ型で行うので計算スループットが ACLE 実装と比べると半分になる。型変換の処理自体もオーバーヘッドになる。clamping やアルファマスクの判定には自動ベクトル化されたコードでも SVE の命令が用いられるが、画像合成の前に predicate マスクのデータ型の変換が行われるため、追加のオーバーヘッドが発生する。その一方で、ACLE 実装は *svqadd_u8()* 関数と predicate マスクを用いることで *unsigned char* データ型の SIMD 演算のみで処理を行うので自動ベクトル化コードで見られるオーバーヘッドは発生しない。

このような最適化はプログラムの意味とアーキテクチャの命令セットを理解することで可能になるものであり、自動ベクトル化で同等の SIMD 命令を生成することは困難である。すべての逐次コードに対して最適な SIMD 命令を生成することは難しいため、本研究の提案手法はどのようなコンパイラ実装であっても有効である。性能評価で示されたコードからわかるように、提案手法を用いることで SIMD 命令の使い方を明示的に制御することが可能である。

6. 結論と今後の課題

本研究では OMP 言語仕様でプログラマによる SIMD 関数実装を明示的に与える *alias simd* 指示文を提案した。*alias simd* 指示文は SIMD ループ文の中で呼ばれる関数のベクトル化をコンパイラ側で行わずに、アーキテクチャ毎に最適化された外部 SIMD 関数に置き換えるように SIMD 関数と逐次コードの関連付けを行う。このようなモデルを用いることによって、ループのイテレーション変換をコンパイラに任せ、命令セットやベクトル幅の変更をコンパイラ側で吸収し、ループの中の SIMD 命令の選択をプログラマが明示的に制御することができる。すべてのコードに最適な SIMD コードを生成するコンパイラを実装することは困難であるため、提案手法のプログラミングモデルを用いることでコンパイラの機能を補う SIMD 命令最適化が可能である。今後の課題として LLVM コンパイラに提案手法を実装し、Intel の命令セットを含む環境で言語モデルの評価や改善を行う。

参考文献

[1] ARM Scalable Vector Extension: https://developer.arm.com/products/architecture/a-profile/docs.
[2] Pohl, A., Cosenza, B., Mesa, M. A., Chi, C. C. and Juurlink, B.: An Evaluation of Current SIMD Programming Models for C++, *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*,

WPMVP '16, New York, NY, USA, ACM, pp. 3:1–3:8 (online), DOI: 10.1145/2870650.2870653 (2016).

[3] ARM C Language Extensions for SVE: https://developer.arm.com/docs/100987/latest/arm-c-language-extensions-for-sve.

[4] Pharr, M. and Mark, W. R.: ispc: A SPMD compiler for high-performance CPU programming, *2012 Innovative Parallel Computing (InPar)*, pp. 1–13 (online), DOI: 10.1109/InPar.2012.6339601 (2012).

[5] Krzikalla, O. and Zitzlsberger, G.: Code Vectorization Using Intel Array Notation, *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '16, New York, NY, USA, ACM, pp. 6:1–6:8 (online), DOI: 10.1145/2870650.2870655 (2016).

[6] Leissa, R., Haffner, I. and Hack, S.: Sierra: A SIMD Extension for C++, *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, New York, NY, USA, ACM, pp. 17–24 (online), DOI: 10.1145/2568058.2568062 (2014).

[7] DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P. and Vitek, J.: Terra: A Multi-stage Language for High-performance Computing, *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, New York, NY, USA, ACM, pp. 105–116 (online), DOI: 10.1145/2491956.2462166 (2013).

[8] Estérie, P., Gaunard, M., Falcou, J., Lapresté, J.-T. and Rozoy, B.: Boost.SIMD: Generic Programming for Portable SIMDization, *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, New York, NY, USA, ACM, pp. 431–432 (online), DOI: 10.1145/2370816.2370881 (2012).

[9] Ewart, T., Delalondre, F. and Schürmann, F.: Cyme: A Library Maximizing SIMD Computation on User-Defined Containers, *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, New York, NY, USA, Springer-Verlag New York, Inc., pp. 440–449 (online), DOI: 10.1007/978-3-319-07518-1_29 (2014).

[10] Kretz, M. and Lindenstruth, V.: Vc: A C++ Library for Explicit Vectorization, *Softw. Pract. Exper.*, Vol. 42, No. 11, pp. 1409–1430 (online), DOI: 10.1002/spe.1149 (2012).

[11] Petrogalli, F.: A sneak peek into SVE and VLA programming (2016).