Vol.2019-HPC-168 No.4
2019/3/5

AVX-512 Intrinsicsで実装されたステンシル計算のScalable Vector Extensionへの展開

廣川 祐太[1,a)]　朴 泰祐[1]　矢花 一浩[1]

概要：我々は電子動力学アプリケーション “SALMON” を開発し，Intel Xeon Phi (Knights Landing, KNL) クラスタ “Oakforest-PACS” での全系実行・性能評価を行ってきた．“ポスト「京」コンピュータ” のプロセッサとして開発が進められている “A64FX” プロセッサは，KNL と非常に近い構成 (メニーコア，高バンド幅メモリ，512 bit SIMD) を採用することから，KNL プロセッサへの実装と最適化はポスト「京」コンピュータの活用に大きく貢献できると期待される．しかし，KNL は Intel AVX-512，A64FX では 512 bit Arm Scalable Vector Extension (SVE) が SIMD 命令として提供されるため，コードの書き換えが必要となる．KNL を活用する実アプリケーションの場合，AVX-512 intrinsics を用いた高度なベクトル最適化の実施が期待され，ユーザは AVX-512 実装の SVE への変換を考えなければならない．本研究では，AVX-512 intrinsics で最適化された SALMON のステンシル計算コードを例として，AVX-512 から SVE への展開を議論する．

1. はじめに

これまで, 我々は電子動力学アプリケーション “SALMON (Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience)” [1] の開発を進め，メニーコアプロセッサ，特に Intel Xeon Phi を対象として最適化を行ってきた．[2] では，筑波大学と東京大学が共同設置する JCAHPC (Joint Center for Advanced HPC, 最先端共同 HPC 基盤施設) にて稼働している Knights Landing (KNL) クラスタ “Oakforest-PACS” [3] での全系実行と性能評価を報告し，大規模メニーコアクラスタにおける十分な性能を示した．現在，我々は “ポスト「京」コンピュータ” を次のターゲットシステムとして，同システムによる大規模電子動力学シミュレーションの実現を目指している．

ポスト「京」コンピュータの開発プロジェクト “FLAGSHIP 2020” では，Arm v8.2-a アーキテクチャベースの “A64FX” プロセッサの開発が進められている [4]．A64FX プロセッサは，(1) 48 コア (12×4) のメニーコア構成，(2) 合計 1024 GB/sec の高バンド幅メモリ HBM2 を 32 GiB 搭載，(3) Arm Scalable Vector Extension (SVE) [5] をサポートし 512 bit 幅の SIMD 演算を提供する．A64FX プロセッサは KNL と非常に近い構成を持っており，KNL へのアプリケーション最適化は 2021 年頃に共用開始を目指しているポスト「京」コンピュータへ向けた準備として非常に重要である．

ここで，Oakforest-PACS (KNL) に対し十分に最適化されたアプリケーションを，ポスト「京」コンピュータ (A64FX) に移植することを考える．A64FX は，12 コアをひとつのグループとして 4 グループを 1 つのチップに実装する NUMA 構成を取り，ノード間ネットワークには 6-dimensional mesh/torus network の Tofu-D を採用する．スレッドとプロセスのアフィニティは性能に大きく影響するが，アプリケーションの並列化は，KNL と同様に従来の OpenMP+MPI ハイブリッド並列を利用できる．しかし SIMD 演算には全く異なる命令が採用されており，コンパイラによる自動ベクトル最適化ではなく，Intrinsics を用いた手動ベクトル最適化を行っている場合には，AVX-512 から SVE への変換と書き換えを考えなければならない．SVE はビット幅が 128 bit 単位で 128–2048 bit の範囲で実装され，SIMD 幅を意識しない計算環境を提供する Vector Length Agnostic (VLA) な SIMD 命令セットである．KNL から A64FX への手動ベクトル最適化コードの移植と考えれば，512 bit SIMD 命令のコード変換と書き換えの問題と捉えることができる．

本研究では高性能計算で多用されるステンシル計算を取り上げ，AVX-512 intrinsics で実装されたコードを SVE intrinsics へ変換する方法について議論する．AVX-512 が高水準な演算をサポートするのに対し，SVE はプリミティ

[1] 筑波大学 計算科学研究センター
[a)] hirokawa@ccs.tsukuba.ac.jp

ⓒ 2019 Information Processing Society of Japan

```
void daxpy_1_1( int64_t n, double da
              , double *dx, double *dy )
{
  for (int64_t i = 0; i < n; ++i) {
    dy[i] = da * dx[i] + dy[i];
  }
}
```

図 1 DAXPY の C 言語実装.

ブな演算を主として提供している．したがって AVX-512 から SVE への変換は，AVX-512 実装互換な処理をどのように複数の SVE 命令で実現するのか，また可能であるかが議論の対象となる．

本稿の構成を次に示す．第 2 章では SVE の概要を紹介し，第 3 章で関連研究を，第 4 章にて本研究で取り上げるステンシル計算の概要と AVX-512 intrinsics の実装詳細を述べる．第 5 章では SVE への変換について述べ，コンパイラによるベクトル最適化との簡易比較を行い，どのように VLA を満たす手動ベクトル最適化実装 (以下 VLA 実装) を行うか議論する．最後に，第 6 章で本研究のまとめと今後の課題を述べる．

2. Arm Scalable Vector Extension (SVE)

SVE は，Arm が提供する Arm v8-a AArch64 アーキテクチャ用の高性能計算向け SIMD 命令セットで，VLA プログラミングモデルを提供する [5]．SVE の演算器を実装するプロセッサは，128 bit の倍数で最小 128 bit から最大 2048 bit のベクトルレジスタを提供する．実際の SIMD 幅は SVE を実装するプロセッサに依存し，VLA 演算を実現するためほぼすべての命令が predicate によるマスク付き命令として提供される．SVE は 32 個のベクトルレジスタ (Z0–Z31) が提供されており，レジスタ数は AVX-512 と同数である (ZMM0–ZMM31)．マスクを行う predicate レジスタは 16 個 (P0–P15) 提供され，128 bit あたり 16 bit の長さで 1 Byte 単位でのマスクが可能である．

Arm は，SVE の C/C++言語用プログラミングインターフェイスとして Arm C Language Extension for SVE (ACLE) を定義している [6]．ACLE は C11 _Generic や C++ template などによりジェネリックなベクトル型とビルトイン関数を提供することで，SIMD 長非依存なプログラミングインターフェイスを提供する．現在，ACLE は Arm C/C++ compiler でしかサポートされていないが，AVX-512 intrinsics と同様に GCC や LLVM/Clang などでのサポートが期待される．

[5] で紹介されているコード例を上げ，ACLE を用いた実装の概要を述べる．**図 1** に，倍精度浮動小数点数ベクトル積和算 (DAXPY) の C 言語実装を示す．`int64_t` は 64 bit 幅の整数型で，サイズ `n` のベクトル `x`，`y` について，

```
void daxpy_1_1( int64_t n, double da
              , double *dx, double *dy )
{
  int64_t i = 0;
  svbool_t pg = svwhilelt_b64(i, n);
  do {
    svfloat64_t dx_vec = svld1(pg, &dx[i]);
    svfloat64_t dy_vec = svld1(pg, &dy[i]);
    svst1(pg, &dy[i]
         , svmla_x(pg, dy_vec, dx_vec, da));
    i += svcntd();
    pg = svwhilelt_b64(i, n);
  } while (svptest_any(svptrue_b64(), pg));
}
```

図 2 DAXPY の ACLE 実装.

スカラ `a` との積和算 `y = a * x + y` を計算する．図 1 を ACLE で実装したコードを，**図 2** に示す．ここで，`sv` のプレフィックスを持つ型と関数はすべて ACLE で定義されており，ハードウェアのベクトル長に依らないジェネリックな名前が用いられている．

まず，`svwhilelt_b64(i, n)` でアクティブな要素を検索し，predicate 配列 `svbool_t pg` に保存する．各ベクトル命令に predicate `pg` を指定して，ベクトルの演算結果をマスクする．`svld1(pg, &dx[i])` は，第 2 引数に指定されたアドレスを始点としてベクトルレジスタ 1 本分をロードし `svfloat64_t dx_vec` に保存する．次に，`svmla_x(pg, dy_vec, dx_vec, da)` でベクトルの積和演算 `dx_vec * da + dy_vec` を計算する．SVE の FMA 演算は 3 オペランドで実装されるため，`svmla_x` では第 1 オペランド (`dy_vec`) の値が破壊される．Intrinsics に指定されたサフィックスは predicate の非アクティブ要素に関する扱いを示し，`_x` は非アクティブ要素に対して操作を行わず，不定値となる．`svmla` の第 4 引数 (`da`) にはスカラ値が許可されており，`svdup_f64_x` によりアクティブな全要素が `da` のベクトルを生成，第 3 オペランドに指定するのと等価である．返却された結果を `svst1` で配列`&dy[i]` にベクトルレジスタ 1 本分をストアする．

`svcntd` は実行している SVE プロセッサの SIMD 幅で，64 bit 要素をいくつ計算できるかを取得できる．つまり変数 `n` のループのステップ数となり，例えば 512 bit SIMD プロセッサの場合は 8 が返却される．再度ループ内で `svwhilelt_b64` を実行し，ループの条件式に `svptest_any` を使い，predicate 内にアクティブな要素が 1 個以上存在するか，すなわちリマインダループも含めて全要素の演算が完了したかを判定する．柔軟な predicate 生成命令により図 2 は原理上，プロセッサがサポートするベクトル長に依存せず，かつシンプルに VLA 演算を実現できる．

ⓒ 2019 Information Processing Society of Japan

```
#pragma omp parallel do collapse(2)
for ik = [1,Nk]
for ib = [1,Nb]
  local l_domain[:,:,:,0]
    = g_domain[1:Nz,1:Ny,1:Nz,ib,ik]

  /* single-thread computation */
  for s = [1,4]
    /* 25 points stencil */
    for ix = [1,Nx]
    for iy = [1,Ny]
    for iz = [1,Nz]
      l_domain[iz,iy,ix,s] = l_domain[...,s-1]

    /* update */
    g_domain[:,:,:] += l_domain[:,:,:,1:4] * c
```

図 3 SALMON における 25 点ステンシル計算の擬似コード.

3. 関連研究

[7] では 7 点と 27 点 3 次元ステンシル計算を SVE で実装・最適化を行い，128, 256, 512, 1024, 2048 bit 長での性能を gem5 シミュレータにより評価している．同研究ではループアンロールやループ融合，データ再利用性の向上といった一般的な最適化が SVE プロセッサに与える影響について示されている．

[8] では SIMD 最適化のための OpenMP 拡張を提案し，SVE ベースで命令数の評価が行われている．OpenMP では SIMD 演算を明示するためのキーワードとして，`omp simd` ディレクティブが定義されている．提案では `omp simd` ディレクティブを拡張して SIMD intrinsics を用いた明示的な部分ベクトル最適化をサポートし，コンパイラ最適化による自動化と SIMD によるプログラマがハンドリングする最適化を取り持ち，性能と実装コストをバランスする．

これらはコンパイラの最適化を中心とした計算カーネルの SVE 実装と最適化手法について論じているが，本研究では AVX-512 intrinsics で高度に最適化されたコードを SVE プロセッサに実装する手段について議論する．

4. AVX-512 を用いたステンシル計算の実装

本節では，本研究で用いる実アプリケーション SALMON のステンシル計算と AVX-512 intrinsics による手動ベクトル最適化実装の概要を述べる．

4.1 ステンシル計算概要

SALMON は Fortran で開発された電子動力学アプリケーションで，TDDFT (Time-Development Density Functional Theory) 計算による第一原理の電子動力学シミュレーションを実現する．TDDFT をベースとした電子の波動関数では，波数空間と実空間の両方が並列化可能だが，

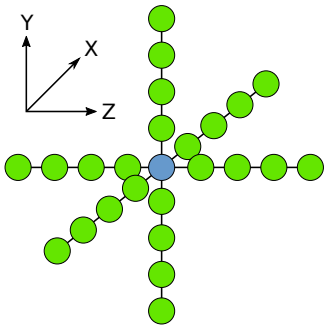

図 4 25 点ステンシル計算のメモリアクセスパターン.

計算量は電子動力学を記述する物質に依存する．例えばナノ構造分子における電子動力学を記述する場合，波数空間と実空間両方の計算量が大きくなるが，古典電磁気学と組み合わせたマルチスケールシミュレーションでは実空間が L2 キャッシュに収まる程度に小さくなる．したがって，ナノ構造分子の計算では両空間のプロセス・スレッド並列化が，マルチスケールシミュレーションでは波数空間のプロセス・スレッド並列化が最適と考えられる．本研究はマルチスケールシミュレーションにおけるステンシル計算を対象とするが，SALMON では差分次数 4 で安定しており，シミュレーションのタイプに依存せず同じ差分次数で計算できる．

図 3 に，ターゲットとする 25 点ステンシル計算の擬似コードを示す．前述の通り，SALMON は波数空間と実空間のどちらも並列化可能だが，本研究の対象計算では波数空間に対し実空間は非常に小さい．したがって波数空間のみを並列化し，ステンシル計算を行う実空間は逐次計算する．擬似コードの中で，`Nk`，`Nb` は波数空間のサイズを示すパラメータ，`Nx`，`Ny`，`Nz` は 3 次元実空間のサイズを示す．`Nk` と `Nb` には依存関係がないため，`omp collapse` を用いて並列化ループサイズを最大化する．並列化されたループの中では，各スレッドは計算する実空間ドメイン (`Nx`，`Ny`，`Nz`) をスレッドローカル配列にコピーする．各実空間 (`l_domain`) に対しステンシル計算が 4 回行われ，4 回すべての出力を用いて電子の波動関数 (`g_domain`) を更新する．25 点ステンシル計算は，周期境界条件で**図 4** に示すように隣接する直交格子を用いて 4 次差分計算を行う．各格子点は倍精度複素数で，512 bit SIMD 演算では 4 要素 (128×4) が同時に計算できる．

4.2 AVX-512 intrinsics 実装

本研究で取り上げる実アプリケーションおよびステンシル計算の実装はオープンソースソフトウェアとして公開されているため，実装詳細は [9] を参照されたい．本節では，どのような最適化が行われているかを述べる．

AVX-512 intrinsics による実装の前に，我々は Knights Corner (KNC) を用いて Fortran コードを最適化し，コン

ⓒ 2019 Information Processing Society of Japan

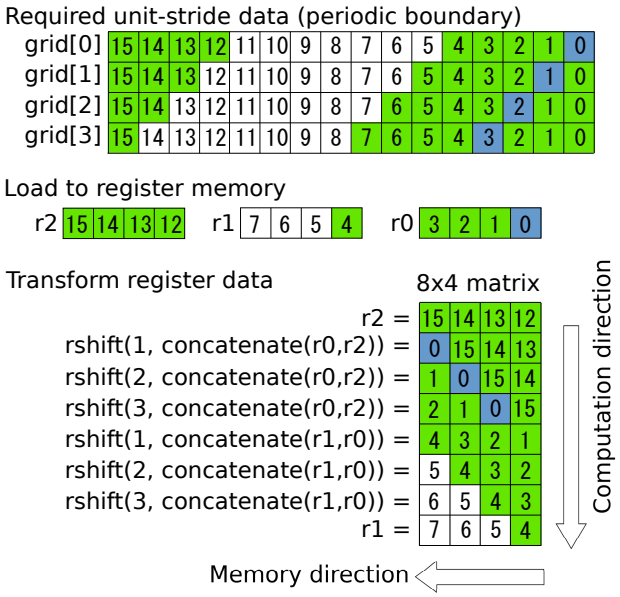

図 5 ステンシル計算の連続領域アクセス最適化.

パイラによるベクトル最適化の促進を行ったが，性能が不十分と判断し早期から SIMD intrinsics を用いた手動のベクトル最適化を行ってきた．特に連続領域である Z 次元のメモリアクセス最適化を行い，メモリ帯域要求を下げ大幅な性能向上に成功した．concatenate shift 命令である `alignr` 命令を用いて，性能が低くボトルネックとなりやすい `gather` 命令 [10] を使わずに連続領域のアクセスを**図 5** の通り最適化した．ここで `rshift` はデータ単位の右シフト命令で第一引数個分ベクトルレジスタを右シフトして下位 4 要素を返し，`concatenate` は 2 つのベクトルレジスタを結合したものを返す関数とする．まず `r0`, `r1`, `r2` の 3 つのベクトルレジスタに必要なデータをロードし，これらだけを用いて必要な近傍点を全て格納した 8×4 の行列を `alignr` 命令により生成する．生成した行列には各格子点が必要とする近傍点が列方向に揃っているため，非常にシンプルな SIMD 演算で，4 つの格子点を同時に計算・更新することが可能となる．この最適化は，Intel が公開したレポート [11] をベースに周期境界条件に対して最適化を行ったもので，[11] とは異なり周期境界条件でなくとも適用可能である．

計算は倍精度複素数で行われるが，最適化によって倍精度複素数の演算は倍精度浮動小数点数との積 (スカラ倍) と加減算のみで実装されている．したがって，計算で必要な演算はすべて倍精度浮動小数点数演算とみなせる．

5. SVE への変換

本節では，AVX-512 intrinsics を SVE intrinsics にどのように変換と書き換えを行うか議論する．はじめに，AVX-512 命令互換で単純置換可能な SVE 命令をまとめ，次に単純置換不可能な処理をどのように SVE intrinsics の組み合わせで実現するかを議論する．

```
svfloat64_t _mm512_set4_pd_sve(d, c, b, a) {
  svfloat64_t x = svdupq_f64(a, c);
  svfloat64_t y = svdupq_f64(b, d);
  return svzip1(x, y);
}
```

図 6 `_mm_512_set4_pd` の SVE 実装.

5.1 AVX-512 互換命令と互換演算の実装

AVX-512 と SVE について，SALMON の手動ベクトル最適化実装で必要な各 intrinsics の対応表を**表 1** に示す．AVX-512 では predicate によるマスク処理が利用できる intrinsic には名称に `_mask_` が付けられるが，SVE はほぼすべての intrinsic で predicate が要求される．したがって，AVX-512 のマスク付き演算との対応は本表では省略している．

基本的なロード・ストアや演算命令は対応する SVE intrinsics が存在するため，単純な置換で解決できる．しかしながら，非互換命令や複数命令で実現されている処理は，SVE intrinsics を組み合わせて実装しなければならない．

非互換命令の例として，`_mm512_set4_pd` を SVE intrinsics でどのように実装するかを考える．ベクトルの要素型が `int32_t` など 128 bit で 4 要素を表現できる場合，`svdupq` で容易に実現できるが，64 bit 整数や浮動小数点数の場合，4 要素は 256 bit 必要となるため，既存命令では処理できない．`_mm512_set4_pd` の SVE 実装を，**図 6** に示す．

`svdupq_f64` で `(a, c, a, c, ...)` および `(b, d, b, d, ...)` と値をコピーしたベクトル `x`, `y` を生成する．ベクトル `x`, `y` を `svzip1(x, y)` でインターリーブコピーし，ベクトル `x`, `y` の最下位ビットから数えて半分の要素を 1 要素ずつ交互にコピーしたベクトルを生成する．図 6 で 4 つのスカラ値で埋めたベクトルを生成できるが，128 bit SVE で実行した場合は `a`, `b` しか参照されない．以上のように，AVX-512 命令を SVE 命令の組み合わせで実現することは容易と考えられる．

5.2 命令の組み合わせで実装する処理

命令の処理効率を考えれば，表 1 で示した非互換命令を SVE で実装するよりも，機能単位でまとめた処理を実装する方が適している．ステンシル計算で行っている処理を機能単位で分けて考えたとき，下記の 2 つを SVE 命令の組み合わせで実現する必要がある．

- 倍精度ベクトルから倍精度複素ベクトルへの変換
- 最適化した倍精度複素ベクトル積

倍精度ベクトルから倍精度複素ベクトルへの変換は，倍精度複素数で表現された各格子点のスカラ倍に必要で，倍精度ベクトルの各要素を実部と虚部にコピーした倍精度

表 1 AVX-512/SVE intrinsics の対応表．T はベクトルの要素型を示す．

AVX-512	SVE	演算内容
_mm512_setzero_T()	svdup_T_z(pg, 0)	ゼロベクトルを生成
_mm512_set1_T(a)	svdup_T_z(pg, a)	スカラ値 a で埋めたベクトルを生成
_mm512_set4_T(d, c, b, a)	–	スカラ値 a, b, c, d で順に埋めたベクトルを生成
_mm512_mask_blend_T(pg, x, y)	svsel(pg, x, y)	predicate pg によってベクトル x, y の要素を混ぜ合わせる
_mm512_alignr_T(x, y, n)	svext(x, y, n)	ベクトル x, y を連結し n 要素ずらしたベクトルを取り出す
_mm512_shuffle_i32x4(x, f)	–	Int32 ベクトル x を f にしたがって 128 bit ブロック単位で並び替える
_mm512_shuffle_T(x, f)	–	ベクトル x を f にしたがって 128 bit ブロック内で並び替える
_mm512_load_T(p)	svld1(pg, p)	アドレス p からベクトルレジスタ 1 本分をロード
_mm512_store_T(p, x)	svst1(pg, p, x)	アドレス p にベクトル x をストア
_mm512_stream_T(p, x)	svstnt1(pg, p, x)	アドレス p にベクトル x をストア (with non-temporal flag)
_mm512_add_T(x, y)	svadd_x(pg, x, y)	ベクトル x + y を計算
_mm512_sub_T(x, y)	svsub_x(pg, x, y)	ベクトル x - y を計算
_mm512_fmadd_T(x, y, z)	svmad_x(pg, x, y, z) svmla_x(pg, z, x, y)	ベクトル x * y + z を計算
_mm512_xor_si512(x, y)	sveor_x(pg, x, y)	ベクトル x xor y を計算

```
/* AVX-512 */
__m512d dcast_to_dcomplex(double const *v) {
  __m512d w = _mm512_maskz_expandloadu_pd(0x55, v);
  return _mm512_movedup_pd(w);
}

/* SVE */
svfloat64_t dcast_to_dcomplex(double const *v) {
  svfloat64_t x   = svld1(svptrue_b64(), v);
  svuint64_t  idx = svindex_u64(0, 1);
  return svtbl(x, svzip1(idx, idx));
}
```

図 7 倍精度ベクトルから倍精度複素ベクトルの変換．

複素ベクトルを生成する．倍精度複素ベクトル積は，ターゲットの計算ではループ中に一度だけ必要で，倍精度複素数 (a, bi) に対し $(a, bi)(0, -i)$ を計算する．式を展開すると $(b, -ai)$ になり，(1) 実部と虚部の入れ替え，(2) 虚部の符号ビットを反転，の 2 ステップで処理できる．本研究では上記 2 つの演算を例として複雑処理の SVE 実装を考え，512 bit 幅での解説を行う．

実部をブロードキャストした倍精度複素ベクトルの生成

倍精度複素ベクトルに対し倍精度ベクトルをかける (スカラ倍を計算する) 場合，512 bit 長でメモリからデータをロードすると，倍精度複素ベクトルは 4 要素であるのに対し，倍精度ベクトルは 8 要素がロードされる．計算には倍精度ベクトルの半分の要素が使用されるため，倍精度ベクトルから各要素の実部と虚部が同値の倍精度複素ベクトルを生成しなければならない．**図 7** に，倍精度ベクトルから倍精度複素ベクトルへの変換方法を示す．AVX-512 ではマスク付きのロード命令で先頭アドレスから 4 要素を実部の位置に読み込み，movedup で実部の値を虚部にコピーする．このとき，512 bit 幅で固定されているため AVX-512 では 8 bit の即値マスクを指定できるが，SVE の predicate では即値指定ができない．

```
svfloat64_t dcomplex_mul_c(svfloat64_t x) {
  svbool_t    pg = svptrue_b64();
  svuint64_t inv, idx, ux;
  inv = svdupq_u64(0, 1ULL << 63);
  idx = svzip1( svindex_u64(1, 2)
              , svindex_u64(0, 2) );
  ux  = svtbl(svreinterpret_u64(x), idx);
  return svreinterpret_f64(sveor_x(pg, ux, inv));
}
```

図 8 式展開した倍精度複素ベクトル積 $(a, bi)(0, -i)$ の実装．

同処理の SVE 実装では，predicate を生成せずに並べ替え命令にあたる svtbl(x, idx) を使用する．svtbl は，ベクトル x をインデックスベクトル idx で並べ替えたベクトルを生成する．インデックスベクトルの生成には，svindex_T が用意されており，開始値と増分を設定できる．svindex_u64(0, 1) とすると，512 bit 幅なら符号なし整数ベクトル (0, 1, 2, 3, 4, 5, 6, 7) が生成される．インデックスベクトルから，svzip1(idx, idx) で idx の前半分をインタリーブコピーしたベクトルを生成する．svzip1 に同じベクトルを指定することで，ベクトルの最下位ビットから前半分の各要素を 2 個ずつ格納したベクトル (0, 0, 1, 1, 2, 2, 3, 3) が生成され，倍精度ベクトルから実部と虚部が同値の倍精度複素ベクトルを生成できる．

ただし，同実装はメモリのアラインメントに依存せずロード命令が実行できると仮定した実装のため，アラインメントを跨いだ時に特別な処理が必要な場合はこの限りではない．

最適化した倍精度複素ベクトル積

式展開し簡略化した倍精度複素ベクトル積 $(a, bi)(0, -i)$ の実装を**図 8** に示す．svdupq_u64(a, b) で，スカラ値 a,

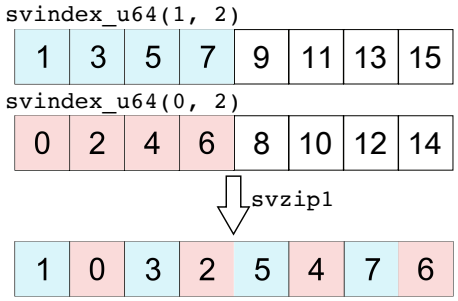

図 9 実部と虚部の並べ替えを行うインデックスの生成.

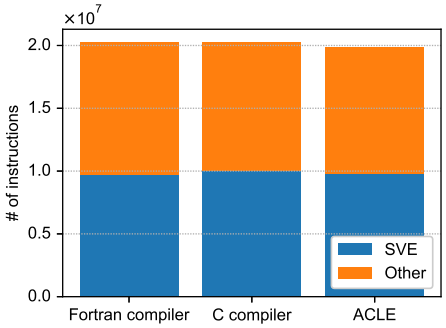

図 10 armie で計測した SVE 命令の実行数 (512 bit SIMD).

b で埋めた符号なし整数ベクトルを生成する．inv には複素数における実部が 0，虚部の最上位ビットが 1 となるベクトルを生成し，虚部の符号ビット反転に用いる．

次に，複素数の実部と虚部を入れ替えるが，前節と同様にインデックスを生成して svtbl と svzip1 を用いた並べ替えを行う．svzip1 に渡す引数は，それぞれ開始値が 1 または 0 で，増分 2 で生成したインデックスベクトルである．2 つのインデックスベクトルを svzip1 でインタリーブすると，**図 9** の青と赤でカラーリングされたベクトルが生成される．生成したインデックスベクトル idx を svtbl に渡して，実部と虚部を入れ替える．

svreinterpret_T は，SVE ベクトル型を別の型に変換する．つまりポインタの読み替えを行うが，SVE のベクトル型はジェネリックな構造体型として定義されるため，読み替えが安全であることを保証しなければならない．符号なし整数型として並べ替えたベクトルに対し，inv を使って XOR 演算により符号ビットを反転する．

5.3 コンパイラによるベクトル最適化との比較

上記の SVE 実装をもとに，Arm HPC コンパイラ 19.0 を用いて Fortran および C 言語実装のコンパイラベクトル最適化との簡易比較を行う．ACLE (SVE intrinsics) をサポートするコンパイラは，現時点で Arm コンパイラしかない．一般的に用いられるオープンソースコンパイラの GCC と LLVM は SVE 命令の出力をサポートしているが，GCC では対象のステンシル計算はほぼベクトル最適化が行われず，LLVM は Fortran コンパイラが未だ開発中で提供されていないため評価と比較が困難である．GCC でのベクトル最適化については，ベクトル化が行われるように実装の書き換えを予定している．

SVE コードの実行には，Cavium ThunderX2 プロセッサ上で Arm instruction emulator for SVE (armie) を使用する [12]．armie は Arm プロセッサ上で SVE 命令を NEON など既存命令の組み合わせでエミュレートするため，高速にエミュレーションが実行できる反面，性能評価はできない．例えば 512 bit SVE のエミュレートをする場合，下記のように-msve-vector-bits=512 を指定するだけで良い．

```
%> armie -msve-vector-bits=512 \
        --iclient libinscount_emulated.so ./a.out
```

--iclient ... オプションにより，指定した実行ファイルの実行命令数と内エミュレートされた命令数をカウントできる．SVE 命令の実行数はエミュレートする SVE の SIMD 幅に依存するため，SIMD 幅が長いほど実行命令数は少なくなる．単純に命令処理数で比較することは困難だが，計算に対し SIMD 演算が積極的に行われているかを判断する材料になる．

Arm コンパイラの最適化オプションは，SVE を用いたベクトル最適化が積極的に行われるようにすべてのケースで-march=armv8.1-a+sve -Ofast を指定する．Arm コンパイラの-Ofast オプションは，Intel コンパイラの-O3 -no-prec-div -fp-model fast=2 に相当する．最適化に-O3 を指定する場合，-ffp-contract=fast オプションを別途指定して FMA 命令をコンパイラが利用することを許可する必要がある．

Arm コンパイラによるベクトル最適化と，ACLE で実装した 512 bit SIMD コードの実行命令数を**図 10** に示す．評価データサイズは実空間格子点が $(16, 16, 16)$，波数空間を 1 とし，一般的なプロセッサの L2 キャッシュに収まる 64 KiB とした．同サイズの倍精度浮動小数点数配列と出力配列を含めると，1 回のステンシル計算で 160 KiB がキャッシュメモリ領域に必要となる．non-temporal store を使用した場合，出力配列を除外して 96 KiB の領域が必要となる．本研究の実装は，Arm Fortran/C コンパイラとほぼ同等水準でベクトル化が行われており，十分なベクトル最適化が実現できていると考えられる．しかしステンシル計算はメモリアクセス性能に律速されるため，キャッシュミスなどのデータ供給待ち時間によって効率的な処理ができていない可能性がある．

5.4 VLA 実装の課題

本研究では 512 bit SIMD 間での実装の変換について議論したが，SVE は本来ベクトル長に依存しない VLA プログラミングモデルを提供する．本節では，VLA プログラミングを実現するための課題について議論を行う．

SVE 命令を組み合わせてベクトル命令の互換処理を実装

するとき，predicate や並び替えに必要なインデックスをどのように生成するかが問題となる．SSE/AVX 命令では，SIMD 幅はアーキテクチャによって決まっていたためビットマスクは即値指定での解決が容易だったが，アーキテクチャがサポートする SIMD 幅に合わせて複数の実装を提供する必要がある．SVE は即値指定ができないかわりに，VLA で実装できれば SIMD 幅に依らず 1 つの実装を使用できる．predicate は生成したインデックスやスカラ値から `svwhilelt` や `svcmpeq` といった比較関数を経由して生成されるため，手動ベクトル最適化においてはどのように複雑なインデックスを生成するかが大きな課題となる．

ステンシル計算の VLA 実装では，連続方向の計算に対して行った差分法の次数とベクトル長に合わせた最適化は非常に困難と考えられる．参照する近傍点が全て直交格子かつベクトル長が差分法の次数以下の場合，本研究でベースとした AVX-512 実装のようにロード命令数を削減可能だが，ベクトル長が差分法の次数より大きい場合には predicate でベクトルレジスタの不要部分の計算をカットする必要がある．しかし，predicate は SVE プロセッサが提供するベクトル長を無為にしてしまうため，可能な限り避けたい．本研究のように周期境界条件の場合，`svext` または `svsplice` を用いて複数のベクトルレジスタを任意のタイミングで結合する必要がある．結合のタイミングは差分次数やベクトル長によって変化するため，ロード命令数を最小化しながらすべてのパターンを網羅することは極めて困難である．

ステンシル計算だけでなく，VLA 実装はすべての計算が SVE の最低 SIMD 幅である 128 bit で計算できるように実装しなければならない．特に，基底となる predicate とインデックスの生成は SIMD 幅に依存せずに生成できなければならない．すべての部分計算についてループを記述し，predicate を計算しながら処理を進める，非常に複雑な実装を要求されることが想像に難くない．[8] で議論されるような手法は，このような煩雑なループの生成をコンパイラに委託して最適化に注力できる解のひとつである．

6. おわりに

本研究では，AVX-512 intrinsics による手動ベクトル最適化が行われたステンシル計算について，Scalable Vector Extension (SVE) への変換と書き換えを議論した．

ステンシル計算は各方向 8 点の近傍点を必要とする 25 点差分計算を周期境界条件で行い，512 bit SIMD を活用した最適化を行っている．本研究では，SVE 実装のターゲットとしてポスト「京」コンピュータの 512 bit SIMD を実装する A64FX プロセッサを挙げた．AVX-512 実装からの変換は，512 bit SIMD 間の命令や処理の変換に相当し，複数の SVE 命令を組み合わせて互換処理を実装し解決した．

Vector Length Agnostic (VLA) を満たす実装を考える場合，すべての計算は最小幅の 128 bit で処理できるように実装する必要があり，Intrinsics を用いた手動ベクトル最適化にかかるコストは非常に高いと考えられる．コンパイラでは実現できない最適化を考えた時，すべての計算パターンを網羅した最適化の実現は非常に困難で，手動ベクトル最適化は従来通りベクトル長に依存した最適化が実施されるのではないかと考えられる．

今後は，我々が最適化を行っているアプリケーション SALMON に本 SVE 実装を組み込み，ポスト「京」コンピュータに向けた準備を加速する．

謝辞 本研究は，JST-CREST 研究課題「光・電子融合第一原理計算ソフトウェアの開発と応用 (課題番号: JPMJCR16N5)」により支援された．本研究の一部は，文部科学省ポスト「京」重点課題 7「次世代の産業を支える新機能デバイス・高性能材料の創成」 (CDMSI) により支援された．

参考文献

[1] M. Noda, S. A. Sato, Y. Hirokawa and *et al.*: SALMON: Scalable Ab-initio Light–Matter simulator for Optics and Nanoscience, *Computer Physics Communications*, Vol. 235, pp. 356–365 (2019).
[2] Y. Hirokawa, T. Boku, M. Uemoto, S. A. Sato and K. Yabana: Performance Optimization and Evaluation of Scalable Optoelectronics Application on Large Scale KNL Cluster, *ISC High Performance 2018: High Performance Computing*, pp. 205–225 (2018).
[3] 最先端共同 HPC 基盤施設: `http://jcahpc.jp/`.
[4] T. Yoshida: Fujitsu High Performance CPU for the Post-K Computer, *Hot Chips 30* (2018).
[5] Arm Scalable Vector Extension User Guide Version 6.11: `https://developer.arm.com/docs/100891/0611`.
[6] Arm C Language Extensions for SVE: `https://developer.arm.com/docs/100987/latest/arm-c-language-extensions-for-sve`.
[7] A. Armejach, H. Caminal, J. M. Cebrian and *et.al*: Stencil Codes on a Vector Length Agnostic Architecture, *In International conference on Parallel Architectures and Compilation Techniques (PACT '18)* (2018).
[8] 李 珍泌，F. Petrogalli，G. Hunter，佐藤三久: 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，情報処理学会研究報告，Vol. 2017-HPC-160, No. 10 (2017).
[9] SALMON (Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience): `http://salmon-tddft.jp/`.
[10] J. Hofmann, J. Treibig, G. Hager and G. Wellein: Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips, *Proceedings of WPMVP'14*, pp. 57–64 (2014).
[11] C. Andreolli: Eight Optimizations for 3-Dimensional Finite Difference (3DFD) Code with an Isotropic (ISO), `https://software.intel.com/en-us/articles/eight-optimizations-for-3-dimensional-finite-difference-3dfd-code-with-an-isotropic-iso`.
[12] Arm Instruction Emulator: `https://developer.arm.com/products/software-development-tools/hpc/arm-instruction-emulator`.

ⓒ 2019 Information Processing Society of Japan