

**ストリーミング SIMD 拡張命令 2(SSE2)
を使用した SAXPY/DAXPY**

バージョン 2.0

2000 年 7 月

資料番号: 248600J-001

【輸出規制に関する告知と注意事項】

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

【資料内容に関する注意事項】

・本ドキュメントの内容を予告なしに変更することがあります。

・インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。

・インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。

・本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。

インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。

・いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複製することは禁じられています。

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社 資料センタ

〒305-8603 筑波学園郵便局 私書箱115号

Fax: 0120-47-8832

*一般にブランド名または商品名は各社の商標または登録商標です。

Copyright © Intel Corporation 1999, 2000

目次

| | | |
|-------|------------------------------------|-----|
| 1 | はじめに | 5 |
| 2 | SAXPY および DAXPY | 5 |
| 2.1 | SAXPY と DAXPY のコーディング | 6 |
| 2.1.1 | SIMD 命令のための C++ クラス・ライブラリの使用 | 6 |
| 2.1.2 | インテル® C/C++ コンパイラのベクトル化機能の使用 | 7 |
| 3 | パフォーマンス | 8 |
| 4 | 結論 | 8 |
| 5 | C/C++ によるコード例 | 9 |
| 6 | SSE2 C++ クラスを使用したコード例 | 10 |
| 7 | SSE2 新規ベクトル化コンパイラのコード例 | 11 |
| 付録 A | パフォーマンス・データ | A-1 |
| | パフォーマンス・データの改訂履歴 | A-1 |
| | テスト・システム構成 | A-2 |

改訂履歴

| 改訂 | 改訂履歴 | 日付 |
|-----|------------------------------|---------|
| 2.0 | インテル® Pentium® 4 プロセッサに関する改訂 | 2000年7月 |
| 1.0 | 初版 | 1999年9月 |

参考資料

このアプリケーション・ノートでは次の資料を参考にした。この資料には、ここで取り上げた事項を理解するための有用な情報が含まれている。

Lawson、Hanson、Kincaid、Krogh 著、『*Basic linear algebra subprograms for Fortran usage*』、ACM Transactions on Mathematical Software、Vol. 5、No. 3、308～371 ページ

Dongarra、Moler、Bunch、Stewart 著、『*LINPACK User's Guide*』、SIAM、1979年

インテル、『*C++ SIMD 命令クラス・ライブラリ・リファレンス・マニュアル*』資料番号 693500J、1999年

インテル、『*インテル® C/C++ コンパイラ・ユーザズ・ガイド*』、資料番号 741901J、1999年

1 はじめに

ストリーミング SIMD 拡張命令 2(SSE2、Streaming SIMD Extensions 2)では、SIMD(Single Instruction Multiple Data)倍精度浮動小数点命令、および SIMD 整数命令が IA-32 インテル® アーキテクチャに新しく導入された。倍精度 SIMD 命令による機能拡張の方法は、ストリーミング SIMD 拡張命令(SSE)で導入された単精度 SIMD 命令による機能拡張とよく似ている。128 ビットの整数 SIMD 拡張命令は、64 ビットの整数 SIMD 拡張命令の完全なスーパーセットで、より多くの整数データ型、整数と浮動小数点間のデータ型変換、キャッシュとシステム・メモリの効果的使用をサポートする命令が追加されている。これらの命令は、3D グラフィックス、リアルタイムの物理的な現象、空間的(3D)オーディオ、ビデオ・エンコーディング/デコーディング、暗号化、および科学計算アプリケーションによく使用される演算を高速化する。このアプリケーション・ノートでは、SAXPY/DAXPY と呼ばれるよく知られた線形代数ルーチンを、単精度/倍精度の浮動小数点データに対して実行するコードを説明し、SSE2 を使用して倍精度バージョンを高速化するコード例を示す。単精度バージョンは SSE を使用した従来のもと同じだが、マイクロアーキテクチャの改良により、インテル® Pentium® 4 プロセッサではインテル® Pentium® III プロセッサより高速に実行できる。SAXPY バージョンのコードに関する記述は DAXPY バージョンにも当てはまるものが多い。本書では、両バージョンに当てはまる場合は「SAXPY(DAXPY)」と表記する。

SAXPY(DAXPY)ルーチンは、ベクトルに定数を掛けて別のベクトルに加える計算を行う。これは、線形代数でよく使用される基本的なベクトル操作である。このルーチンは、コンピュータ(特に高速コンピュータ)のパフォーマンスを測る簡単なベンチマークとしてよく使用されている。このアプリケーション・ノートでは、インテル® アーキテクチャ(IA)プロセッサで使用できる SIMD 命令を利用した SAXPY(DAXPY)ルーチンのコーディングについて説明する。ここでは、SIMD 命令のための C++ クラス・ライブラリ、およびインテル® C/C++ コンパイラのベクトル化機能の両方を使用する。

このアプリケーション・ノートのパフォーマンス測定は「完全キャッシュ」環境で行っている。すなわち、すべてのデータが第 1 レベルのプロセッサ・キャッシュ内に収まるようなデータ・サイズを選択した。また、データはすべて 16 バイト境界にアライメントされているものと仮定している。これらの条件は、SAXPY(DAXPY)を使用するときの現実的環境とはいえない。メモリとキャッシュ間のデータ転送はパフォーマンスに大きく影響するし、データ・アライメントも当然と考えることはできないからである。ここに示すコードは、インテル C/C++ コンパイラを使用して、プロセッサの SIMD 機能を利用する方法を示す有益な例として、参考のこと。

2 SAXPY および DAXPY

SAXPY(DAXPY)は、Lawson、Hanson、Kincaid、Krogh による BLAS(Basic Linear Algebra Subprograms)[Lawson, 1979]で定義されたルーチンである。SAXPY とは、単精度スカラー(SA)掛けるベクトル X、プラス、ベクトル Y を表す略語である。DAXPY は、倍精度データに対する同じ操作を表す。数式で示すと次のようになる。

$$Y = a * X + Y$$

ここで、X と Y はベクトル(要素は 1 ~ n)で、a はスカラーである。この計算は、一般に次のような FORTRAN のサブルーチンで定義する。

```
CALL SAXPY (N, A, X, INCX, Y, INCY)
```

ここで、 N はベクトル長を表す。 $INCX$ と $INCY$ は、ベクトルの各要素がメモリ上で連続していない場合に使用する。SAXPY は、一般に連続領域に入っている x と y の 2 つのベクトルに対して実行するので、多くの場合、 $INCX=INCY=1$ である。実際に、LINPACK という BLAS を使用する線形システム・パッケージでは必ず $INCX=INCY=1$ としてこのルーチン呼び出ししている[Dongarra, 1979]。

高速コンピュータ、特にスーパー・コンピュータの世界では、主に科学技術目的で、LINPACK および BLAS のパフォーマンスに関心が集まっており、SAXPY や DAXPY のパフォーマンス測定値がよく引き合いに出される。このアプリケーション・ノートでは、SIMD 命令を使用できるように、 $INCX=INCY=1$ とした。また、SAXPY と DAXPY の各ルーチンは C++ でコーディングしている。

2.1 SAXPY と DAXPY のコーディング

SSE2 を利用するための最も一般的な方法は、ユーザが直接アセンブリ言語でコーディングする方法である。通常は、C/C++ プログラム内にインライン・アセンブリ・コードを書き込む。それに対して、インテル C/C++ コンパイラを使用すれば、はるかに簡単にパフォーマンスを向上させる 2 つの方法がある。1 つは、SIMD 命令のための C++ クラス・ライブラリを使用し、ある程度トランスペアレントに SSE2 を使用する方法である。もう 1 つのより洗練された方法は、インテル C/C++ コンパイラのベクトル化機能を利用して、コードに含まれているベクトル化可能部分を自動検出する方法である。この 2 つの方法を次の 2 つのセクションで説明する。

2.1.1 SIMD 命令のための C++ クラス・ライブラリの使用

アプリケーションでベクトル命令を使用するには、FVEC および DVEC クラスを使用するだけでよい。本書の最後にその例を示す。データが 16 バイト境界にアライメントされていれば、float(double)ポインタを F32vec4(F64vec2)ポインタにキャストするだけで、通常通りにデータにアクセスできる。すなわち、 x と y が F32vec4(F64vec2)ポインタならば、 $y[I] = \text{scalar} * x[I] + y[I]$ とコーディングするだけで、コンパイラは適切なパックド演算命令、およびロードとストアを 128 ビットで行うコードを生成する。ユーザが行うのは、一度に 4(2)要素ずつまとめて処理されるので、それに合わせてループ・カウントを調整するだけである。コード例を見ると、ポインタを上記のようにキャストし、ループ・カウントを適切に調整していることがわかる。ただし、このコード例では、ベクトル長 n が 4(2)の倍数であり、ベクトルの最後に未処理の要素が残っていないものと仮定している。

スカラー $sa(da)$ の初期化に注意してほしい。1 つのデータを初期値として使用すると、その値がパックド変数 $sa(da)$ の 4(2)要素すべてにブロードキャストされる。結局、元のスカラーに関するソース・コードの変更はわずかですむ。実際に、コア・ループはまったく変わっていない。このように、ソース・コードをわずかに変更するだけで新しい命令を使用できるので、プログラマの生産性とソース・コードの読みやすさが大きく向上する。

1 つだけ注意点がある。データが 16 バイト境界にアライメントされている保証がなければ、無条件に float(double)ポインタを F32vec4(F64vec2)ポインタにキャストしてはいけない。データが偶然アライメントされていれば正しく機能し、アライメントされていなければ正しく機能しないコードになってしまう。アライメントはされていないが x と y の 2 つの配列のアライメン

ト状態が同じであれば、アライメントされていないいくつかの要素を先に計算し、残りの要素をアライメントされたデータとして処理できる。x と y のアライメント状態が異なれば、どちらか1つの配列しかアライメント済みデータとしてアクセスできないため、パフォーマンスは著しく低下する。インテル C/C++ コンパイラでは、`declspec` アライメント(この機能の使用方法については、『C/C++ コンパイラ・ユーザズ・ガイド』を参照のこと)を使用して変数を簡単にアライメントできる。x と y のアライメント状態が事前にわからない場合は、この2つのポインタのアライメントを実行時にテストし、`if` 文を使用して適切なバージョンの SAXPY(DAXPY)を使い分けるようにする。

2.1.2 インテル® C/C++コンパイラのベクトル化機能の使用

コードを書き直すには多大なコストがかかることはよく知られている。そのため、研究者たちはこの30年間、元のコードを自動的に「ベクトル化」するコンパイラ・テクノロジーの開発に従事してきた。SAXPY(DAXPY)でこの機能を使用するには、インテル C/C++ コンパイラの `/QxK` および `/QxW` スイッチを使用する。このスイッチは、自動ベクトル化機能を有効にし、コンパイラは SSE と SSE2 を生成する。`/Qvec_verbose3` スイッチを使用すると、どのループがベクトル化可能かを指摘するだけでなく、ベクトル化不可能なループにはその理由を示す診断情報が得られる。これらのスイッチは、プログラマがコードを書き直したり、コンパイラに対して負担の大きいコード変換作業をガイドしたりするのに使用できる。SAXPY(DAXPY)の場合、インテル C/C++ コンパイラが生成するコードのパフォーマンスは、ユーザが手書きしたアセンブリ・コードに匹敵する。

SAXPY(または DAXPY)のような関数の場合、一般にコンパイラはいくつかの条件がわからないとコードを変換できない。この問題は、言語のセマンティクスのため C/C++の方が FORTRAN より深刻である。例えば、渡されたポインタが実際に異なる配列を指すのかわからないので、ベクトライザはこの関数をベクトル化できない。上記の `/Qvec_verbose3` スイッチを使用すれば、2つの配列が依存関係にある可能性が指摘されるかもしれない。プログラマにとっては自明なことでも、コンパイラには深刻な問題である。`ivdep` という `pragma` を使用すれば、このような依存関係を心配せずにコンパイラに指示できる。プログラマは、`for` ループの直前にこの `pragma` を追加する。そうすると、ループがベクトル化され、ベクトル長が4(2)の倍数でない場合の処理を行うコードが挿入される。その場合も、配列がアライメントされているかどうかは未知である。プログラマがこのことを保証するには、`vector aligned` という `pragma` を使用する。

ただし、インテル C/C++ コンパイラは、`ivdep` と `vector aligned` の2つの `pragma` がなくても、SAXPY および DAXPY 関数をベクトル化できる。そのかわり、配列の依存関係とベクトルのアライメント状態を実行時にチェックするコードを挿入する。ベクトル化するコンパイラ・コード例には、このような実行時チェックをスキップするために、これらの `pragma` が入れている。1つのループで、コード例がベースライン・マークを生成するために、C/C++コード例では `novector` という `pragma` を使用して、コンパイラによるベクトル化を禁止している。

ベクトライザのもう1つの重要な機能として、`/QxW` スイッチのかわりに `/QaxW` スイッチが使用できる。`/QxW` スイッチを使用すると、コンパイラは SSE2(および Pentium III プロセッサ命令)を生成する。そのようなコードは Pentium 4 プロセッサでしか実行できないため、従来の IA プロセッサで実行するためには、SSE2 を含まないコードを生成する必要がある。`/QaxW` スイッチを使用すると、コンパイラが自動的にそのようなコードも生成する。すなわち、SSE2 を使用できる関数は2度コンパイルされる。1度は `/QxW` スイッチが指定されたときと同様に

SSE2 を含むバージョンを生成し、もう 1 度は /QxW スイッチが指定されないときと同様に SSE2 を含まないバージョンを生成する。そのようにして生成された「冗長バイナリ」コードは、任意のインテル・プロセッサで実行でき、新しいプロセッサでは高いパフォーマンスを実現し、初期のプロセッサでもパフォーマンスは低いが同等の関数を実行できる。2 つのコード・バージョンの選択は実行時に行われるので、新旧プロセッサが混在するハイブリッド環境でも 1 つのバイナリ・コードで済む。この手法ではバイナリ・コード・サイズは大きくなるが、種々のアプリケーションで単一の「コード・パス」を提供できる。

3 パフォーマンス

パフォーマンスは「完全キャッシュ」環境を前提として測定した。この環境では、SSE2 を使用するコードはすべて同等のパフォーマンスを示した。すなわち、FVEC(DVEC)コードでは、手書きした最良のアセンブリ・コードに匹敵するパフォーマンスが得られた。これは、SIMD 命令のための C++ クラス・ライブラリを使用すると、高品質な手書きのアセンブリ・コードに匹敵するコードが生成されることから、当然の結果である。これに対して、インテル C/C++ コンパイラが生成したベクトル化コードは、FVEC(DVEC)コードほどのパフォーマンスが得られない。これは、(少なくとも執筆時には)インテル C/C++ コンパイラがベクトル化コードにソフトウェア・プリフェッチを挿入しているためである。このテスト環境では、配列がすでにキャッシュに入っているため、プリフェッチを行うとパフォーマンスはかえって低下する。ただし、一般に SSE と SSE2 は同時に複数の命令(SSE の場合は 4 つ、SSE2 の場合は 2 つ)を実行するので、同じコードをベクトル化しない場合よりもはるかに高いパフォーマンスが得られる。

SAXPY の場合、SSE を使用することで、4 倍の高速化が期待できる。Pentium 4 プロセッサでは期待をわずかに下回る高速化を実現できたが、Pentium III プロセッサではメモリ・システムが実行ユニットの速度に付いていけず 2 倍程度の高速化しか得られなかった。SSE を最大限に活用できるように、プロセッサ内部のマイクロアーキテクチャが改良されたことが、この差に現れている。DAXPY の場合、倍精度 SSE2 のおかげで、大幅な高速化が実現できた。

4 結論

まとめると、SSE や SSE2 のような SIMD 命令をコードに取り入れるには、アセンブリ・コードを手書きするというわずらわしい方法にかわって、次の 2 つの方法がある。(1) SIMD 命令のための C++ クラス・ライブラリ(インテル C/C++ コンパイラが持っている内部機能をカバーするライブラリ)を使用する方法。(2) インテル C/C++ コンパイラのベクトル化機能を利用するための pragma を使用する方法。どちらも、インテル® Itanium™ アーキテクチャのプラットフォームで利用できる。(2)の方法は、ソース・コードの移植性が高いという利点がある。その上、自動コード・パス・ソリューションからベクトル化したコードが呼び出せるので、コードのメンテナンスや配布に関する多くの問題を簡単にできる。

5 C/C++によるコード例

```
*
*Saxpy from BLAS, Lawson, Manson, Kincaid, and Krogh (1979)
*
*this compilation from _Linpack Users' Guide_, Dongarra, Moler,
*   Bunch, & Stewart, Siam 1979, appendix A.
*
*These versions are "unit" saxpy (daxpy), meaning they assume stride = 1.
*(note that this is what BLAS requires, and is the most common form)
*
* Assume all vectors are aligned.
*
*/

void usaxpy (int n, float sa, float *sx, float *sy)
{
    if (sa == 0.0) return;

    //The latest intel compilers can now vectorize this loop.
    //Use the novector pragma to prevent vectorization.
    #pragma novector
    for (int i = 0; i < n; i++)
        sy[i] = sa * sx[i] + sy[i];
}

void udaxpy (int n, double da, double *dx, double *dy)
{
    if (da == 0.0) return;

    //The latest intel compilers can now vectorize this loop.
    //Use the novector pragma to prevent vectorization.
    #pragma novector
    for (int i = 0; i < n; i++)
        dy[i] = da * dx[i] + dy[i];
}
```

6 SSE2 C++クラスを使用したコード例

```
/* Assumes vectors are aligned, and that the vector length n is divisible
 * by 4 for saxpy and 2 for daxpy.
 */
#include <fvec.h>
#include <dvec.h>

void usaxpy_fvec (int n, float sa, float *sx, float *sy)
{
    F32vec4 *x = (F32vec4 *)sx, *y = (F32vec4 *)sy;
    F32vec4 a(sa);

    if (sa == 0.0) return;
    n >>= 2;
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}

void udaxpy_dvec (int n, double da, double *dx, double *dy)
{
    F64vec2 *x = (F64vec2 *) dx, *y = (F64vec2 *) dy;
    F64vec2 a(da);

    if (da == 0.0) return;
    n >>= 1;
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

7 SSE2 新規ベクトル化コンパイラのコード例

```
/* Assumes vectors are aligned */
void usaxpy_vec (int n, float sa, float *sx, float *sy)
{
    if (sa == 0.0) return;

    #pragma ivdep
    #pragma vector aligned
    for (int i = 0; i < n; i++)
        sy[i] = sa * sx[i] + sy[i];
}

void udaxpy_vec (int n, double da, double *dx, double *dy)
{
    if (da == 0.0) return;

    #pragma ivdep
    #pragma vector aligned
    for (int i = 0; i < n; i++)
        dy[i] = da * dx[i] + dy[i];
}
```

付録 A - パフォーマンス・データ

パフォーマンス・データの改訂履歴

| 改訂 | 改訂履歴 | 日付 |
|-----|---------------------------------------|------------|
| 2.0 | 1.20 GHz Pentium 4 のパフォーマンス・データに関する改訂 | 2000 年 7 月 |
| 1.0 | 初版 | 1999 年 9 月 |

表 1: SAXPY/DAXPY のパフォーマンス・データ

| パフォーマンス・データ(単位は MFLOPS) | | |
|-------------------------|-----------------------------------|----------------------------------|
| | Pentium III プロセッサ (733 MHz) | Pentium 4 プロセッサ (1.20 GHz) |
| SAXPY: C コード | 360 | 523 |
| SAXPY: FVEC | 837 | 1932 |
| SAXPY: ベクトライザ | 865 | 1476 |
| DAXPY: C コード | 353 | 640 |
| DAXPY: DVEC | N/A | 951 |
| DAXPY: ベクトライザ | N/A | 672 |

表 1 に、「完全キャッシュ」環境を仮定したパフォーマンス測定結果を示す。ベクトル長は、第 1 レベル・キャッシュ内に収まる最大長を選択した。パフォーマンスは、733 MHz Pentium III プロセッサと 1.20 GHz Pentium 4 プロセッサで測定した。測定に使用したシステムの詳細については、A-2 ページの「テスト・システム構成」を参照のこと。ベクトルの各要素に対して 2 つの演算(乗算と加算)を行い、それによって FLOPS 値を計算した。

SAXPY は、ストリーミング SIMD 拡張命令(SSE)を使用することで、4 倍のパフォーマンス向上が期待されたかもしれない。Pentium III プロセッサでは 2 倍程度の向上しか見られなかったが、Pentium 4 プロセッサでは 3.7 倍の高速化が実現できた。SSE を最大限に活用できるように、プロセッサ内部のマイクロアーキテクチャを改良したことが、この差に現れている。最高値として 1.9 GFLOPS が測定されたのに注意のこと。

DAXPY の場合、倍精度 SSE2 により 1.5 倍の高速化を実現できた。最高値は 951 MFLOPS だが、これは単精度の場合の約半分である。これは、DAXPY では 1 つの SIMD 命令で 2 つの浮動小数点演算を行うが、SAXPY では 4 つの浮動小数点演算を行うからである。

テスト・システム構成

表 2: Pentium III プロセッサのシステム構成

| プロセッサ | Pentium III プロセッサ(733 MHz) |
|----------------------|--|
| システム | インテル® Desktop Board VC820 |
| BIOS バージョン | VC82010A.86A.0028.P10 |
| 2次キャッシュ | 256 KB |
| メモリ・サイズ | 128 MB RDRAM PC800-45 |
| Ultra ATA ストレージ・ドライバ | 製品候補 6.00.012 |
| ハード・ディスク | IBM DJNA-371800 ATA-66 |
| グラフィックス | Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR -32MB |
| ビデオ・ドライバのリビジョン | Nvidia Reference Driver 5.22 |
| オペレーティング・システム | Windows† 2000 ビルド 2195 |

表 3: Pentium 4 プロセッサのシステム構成

| プロセッサ | Pentium 4 プロセッサ(1.20 GHz) |
|----------------------|---|
| システム | インテル Desktop Board D850GB |
| BIOS バージョン | GB85010A.86A.0014.D.0007201756 |
| 2次キャッシュ | 256 KB |
| メモリ・サイズ | 128 MB RDRAM PC800-45 |
| Ultra ATA ストレージ・ドライバ | 製品候補 6.00.012 |
| ハード・ディスク | IBM DJNA-371800 ATA-66 |
| ビデオ・コントローラ/バス | Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR -32MB |
| ビデオ・ドライバのリビジョン | NVidia Reference Driver 5.22 |
| オペレーティング・システム | Windows 2000 ビルド 2195 |