インテル® コンパイラーの浮動小数点演算における結果の一貫性 なぜアプリケーションの答えが常に同じにならないのか?

インテル コーポレーション
ソフトウェア & ソリューション・グループ

Dr. Martyn J. Corden
David Kreitzer

はじめに

ほとんどの実数のバイナリー浮動小数点 [FP] 表現は不正確で、浮動小数点数を含むその演算結果には特有の不確実性があります。浮動小数点アプリケーションのプログラマーは通常、次の目的を持ってプログラミングを行います。

- 精度
  - 正確な演算結果に「近い」結果を生成する
    - 通常は相対誤差で測定し、場合により「最下位桁単位」 (ulp) で測定する
- 再現性
  - 一貫性のある結果を生成する
    - 実行ごとに同じ結果
    - 異なるコンパイラー・オプションで同じ結果
    - 異なるコンパイラーで同じ結果
    - 異なるプロセッサーやオペレーティング・システムで同じ結果
- パフォーマンス
  - できるだけ速く動作するアプリケーションを生成する

これらの目的は、多くの場合互いに矛盾します。ただし、プログラミングの経験を積み、適切なコンパイラー・オプションを使用することで、このトレードオフを制御することができます。

例えば、計算特有の精度を超える再現性があると役立つことがあります。ソフトウェアの品質保証テストでは、演算結果の数学的な不確実性がかなり高いと思われる場合でも、変更の前後で、ソフトウェアがビット単位で「近似」である必要があります。正しいコンパイラー・オプションを使用すると、(最適ではありませんが) 十分なパフォーマンスを保ちつつ、一貫した、非常に近い結果を生成することができます。

浮動小数点セマンティクス

インテル® コンパイラーは、Microsoft®[1] 浮動小数点セマンティクスをベースにしたモデルを実装します。/fp: (Windows®) または -fp-model (Linux* および OS X*) コンパイラー・オプションを使用して、浮動小数点セマンティクスの粒度を選択できます。次のコンパイラー規則を指定することが可能です。

- 安全な値
- 浮動小数点式の評価
- 厳密な浮動小数点例外
- 浮動小数点の縮約
- 浮動小数点演算ユニット (FPU) 環境アクセス

[1] Microsoft® Visual C++® 浮動小数点演算の最適化
http://msdn2.microsoft.com/en-us/library/aa289157(vs.71).aspx (英語)

これらに対応する /fp: (Windows®) または -fp-model (Linux* および OS X*) オプションの引数は次のとおりです。

- precise
  精度に影響しない最適化のみ有効にします。
- source/double/extended
  浮動小数点式の評価で使用する中間結果の精度を指定します。
- except
  厳密な浮動小数点例外セマンティクスを有効にします。
- strict
  FPU 環境アクセスを有効にします。
  FMA (Fused Multiply-Add) 命令のような浮動小数点の縮約を無効にします。
  precise と except が有効になります。
- consistent
  異なるプロセッサーやコンパイラー・オプションで一貫した再現性のある結果を生成します (バージョン 17 以降)。
- fast [=1] (デフォルト)
  精度に影響する最適化を許可します。
  式の評価に使用する精度はコンパイラーが選択します。
  浮動小数点例外セマンティクスを強制しません。
  FPU 環境アクセスを許可しません。
  浮動小数点の縮約を許可します。
- fast=2
  いくつかの追加の近似を許可します。

このコンパイラー・オプションは、/Op や /flt-consistency (Windows®) または -mp や -flt-consistency (Linux* および OS X*) など、インテル® コンパイラーの古いバージョンで実装されているオプションよりも優先されます。

C++ と Fortran で ANSI/IEEE 標準に準拠した浮動小数点値を得るためには、次の設定を推奨します。

/fp:precise /fp:source	(Windows®)
-fp-model precise -fp-model source	(Linux* および OS X*)

C/C++ では、float_control プラグマを指定することで、/fp:precise と /fp:fast (Windows®) または -fp-model precise と -fp-model fast (Linux* および OS X*) と同じ効果が得られます。

```
#pragma float_control (precise, on)
#pragma float_control (precise, off)
```

このプラグマは、コードブロックの前に追加され、そのコードブロックに含まれる関数にも適用されます。

安全な値

SAFE (precise) モードでは、コンパイラーは結果に影響する最適化を行いません。例えば、次のような変換は禁止されています。

$$(x + y) + z \Rightarrow x + (y + z)$$

一般に、この再結合では精度が損なわれる可能性があります。浮動小数点演算の順序を変更 (再結合) すると、異なる中間結果が生成され、それが最も近い浮動小数点表現に丸められることで、最終結果がわずかに異なることがあるためです。

デフォルトは UNSAFE (fast) モードです。通常、「UNSAFE (fast)」モードで生じる差はごくわずかです。しかし、次の 1 つ目の例のように、アルゴリズムに近似値への丸め (大数のわずかな差) が含まれる場合、長いシーケンスの計算では最終結果に大きく影響することがあります。その場合、最終結果の差は、計算の有限精度による結果の不確実性を表しています。

VERY UNSAFE (fast=2) モードは、結果に大きく影響する変換を有効にします。例えば、指数範囲の極値を超える拡張を許可します。

/fp:precise (-fp-model precise) によって無効になる変換の例

- 再結合　例: (a + b) + c ⇨ a + (b + c)
- ゼロフォールド　例: X+0 ⇨ X、X*0 ⇨ 0
- 逆数乗算　例: A/B ⇨ A*(1/B)
- 平方根の近似値
- 超越関数　例: sin(a)、exp(a)
- 突発アンダーフロー (FTZ: Flush-to-Zero)
- RHS の精度を LHS の精度に下げるなど

上記のゼロフォールドの例では、X が無限や NaN (Not a Number、非数) のような特定の値の場合、正しい IEEE 結果が得られません。

ただし、FMA 命令による縮約[2] は、明示的に無効にされている場合、あるいは /fp:strict (Windows®) または -fp-model strict (Linux* および OS X*) が指定されている場合を除き、許可されます。「浮動小数点の縮約」を参照してください。

再結合について

加算と乗算は結合則を満たします。

a + b + c = (a+b) + c = a + (b+c)
(a*b) * c = a * (b*c)

変換前と変換後の式は、数学的には等価ですが、有限精度演算では等価では**ありません**。次のような、ほかの代数恒等式についても同じことが言えます。

a*b + a*c = a * (b+c)

再結合を含む高度な最適化の例として、ループ交換と部分和を使用するリダクション操作のベクトル化があります (「リダクション」を参照)。対応するコンパイラー・オプションは、インテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能ですが、インテル製マイクロプロセッサーにおいてより多くの最適化が行われる場合があります。

ANSI C/C++ 言語標準は、コンパイラーによる再結合を許可していません。括弧がない場合も、浮動小数点式は左から右へ評価されます。インテル® コンパイラーによる再結合は、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) オプションによって無効にすることができます。このオプションを指定すると、精度に影響するほかの最適化も無効になります。また、高度な最適化レベルでは、パフォーマンスが低下することがあります。/fp:fast (Windows®) または -fp-model fast (Linux* および OS X*) オプションを指定すると、インテル® コンパイラーは、括弧がある場合でも、式を再結合する可能性があります。

ANSI Fortran 標準は C 標準よりも制約が少なく、コンパイラーは括弧によって指定された評価順序に従わなければなりませんが、必要に応じて式を並べ替えることができます。インテル® Fortran コンパイラーの /assume:protect_parens (Windows®) または -assume protect_parens (Linux* および OS X*) オプションは、/fp:precise (Windows®) または -fp- model precise (Linux* および OS X*) よりもパフォーマンスへの影響が小さく、標準に準拠した再結合を行います。このオプションは、再結合以外の精度を損なう可能性がある最適化には影響しません。

[2] インテル® マイクロアーキテクチャーは、乗算と加算を 1 つの命令で行うことができます。

同様のオプションとして、インテル® C/C++ コンパイラー 16 以降では、/Qprotect-parens (Windows®) または -fprotect-parens (Linux* および OS X*) を利用できます。このオプションを指定すると、コンパイラーは式の評価順序を決定する際に括弧を尊重し、/fp (Windows®) または -fp-model (Linux* および OS X*) で許可されている括弧で囲まれていない式や部分式は並べ替えます。

Fortran アプリケーションの例

最適化が有効な場合と無効な場合では、アプリケーションの結果が大きく異なりました。そして、その原因は次の式にあることが分かりました。

A(I) + B + TOL

ここで、TOL は非常に小さな正数で、A(I) と B は大数です。最適化を有効にすると、コンパイラーは次のように式を評価します。

A(I) + (B + TOL)

定数式 (B+TOL) は、I のループの入口で 1 回評価されます。しかし、このコードは、A(I) ≈ -B の場合に式が正定値になるように TOL を加算しています。TOL を直接 B に加算すると、有限精度により加算した値が丸められてしまい、A(I) と B が相殺されても正定値になるようにするという役割を果たせません。

この問題を解決する最も簡単な方法は、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) オプションを指定してソースファイルを再コンパイルすることです。これにより、再結合が無効になり、式が左から右に評価されます。パフォーマンスへの影響が小さく、より的を絞った解決方法は、ソースコードで式を次のように変更することです。

(A(I) + B) + TOL

これにより、プログラマーの意図をより明確にし、/assume:protect_parens (Windows®) または -assume protect_parens (Linux* および OS X*) オプションを指定してコンパイルします。

WRF[3] (気象研究/予報モデル) の例

MPI (メッセージ・パッシング・インターフェイス) を利用して、異なる数のプロセッサーでアプリケーションを実行したところ、結果がわずかに異なりました。原因はループ境界でした。MPI プロセスの数に応じて問題が分解され、データ・アライメントが変わったためです。これにより、ループのプロローグ (「ピールループ」)、ベクトル化されたループ本体、ループのエピローグ (「リマインダー・ループ」) に含まれるループ反復が変わりました。プロローグとエピローグで生成されたコードの違いにより、同じデータに対する結果にわずかな差が生じたのです。

この問題を解決するには、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) を指定してコンパイルします。そうすることで、コンパイラーは、ピールループ、リマインダー・ループ、ベクトル化されたループ本体で、一貫性のあるコードと数学ライブラリーの呼び出しを生成します。場合によっては、このオプションはループのベクトル化を妨げることがあります。

リダクション

リダクション・ループ (例えばドット積) の並列実装は、再結合を含む部分和を利用します。そのため、精度に影響します。以下は、浮動小数点リダクション・ループのシリアル実装と並列実装の例です。

[3] http://www.wrf-model.org (英語)

```
float Sum(const float A[ ], int n)
{

    float sum=0;
    for (int i=0; i<n; i++)
        sum = sum + A[i];









    return sum;
}
```

```
float Sum( const float A[ ], int n)
{
    int i, n4 = n-n%4;
    float sum=0,sum1=0,sum2=0,sum3=0;
    for (i=0; i<n4; i+=4) {
        sum = sum + A[i];
        sum1 = sum1 + A[i+1];
        sum2 = sum2 + A[i+2];
        sum3 = sum3 + A[i+3];
    }
    sum = sum + sum1 + sum2 + sum3;
    for (; i<n; i++)
        sum = sum + A[i];
    return sum;
}
```

2 つ目の実装では、SIMD 命令 (例えば、コンパイラーの自動ベクトル化により生成される) または部分和ごとに個別のスレッド (例えば、自動並列化によって生成される) で、4 つの部分和を並列に計算できます。これにより、パフォーマンスが大幅に向上しますが、A の要素が加算される順序が変更されることで丸めによる差が生じ、最終結果がわずかに異なることがあります。このため、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) では、インテル® Cilk™ Plus や REDUCTION 節を含む OpenMP* SIMD プラグマ/ディレクティブによってベクトル化が明示的に指定されている場合を除き、リダクション操作のベクトル化や自動並列化が無効になります。

OpenMP* の並列リダクション操作は、OpenMP* ディレクティブによって指定され、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) で無効になりません。一般に、精度に影響する可能性があるため、プログラマーの責任において使用してください。同様に、MPI ライブラリー呼び出しを含む MPI のリダクション操作もコンパイラーによって制御されず、精度に影響する可能性があります。また、インテル® Cilk™ Plus のレデューサー・ハイパーオブジェクトも精度に影響します。ワークスチール・モデルでは、操作の順序が常に同じになるとは限りません。スケジューリングや MPI プロセスでスレッド数が変更されると、結果がわずかに異なることがあります。場合によっては、同じバイナリーの連続した実行でも操作の順序が異なる可能性があります。

OpenMP* 標準は、部分和の結合順序を指定しないため、ランタイムに決定される順序が変わることで結果にばらつきがでます。インテル® コンパイラー 13 以降では、同じスレッド数とスタティック・スケジューリングでバイナリーを複数回実行した場合に、OpenMP* のリダクション操作が一貫した再現性のある結果を生成するように保証する手段を提供しています。次の環境変数を設定します。

KMP_DETERMINISTIC_REDUCTION=yes (または =on、=true、=1)

この環境変数を設定すると、大きなリダクション操作の精度も向上します。スレッド数が多い場合、KMP_DETERMINISTIC_REDUCTION=yes がデフォルトです。スレッド数が少ない場合は、パフォーマンスに影響する可能性があるため、この設定は使用しません。

インテル® マス・カーネル・ライブラリー (インテル® MKL) とインテル® スレッディング・ビルディング・ブロック (インテル® TBB) は、同じバイナリーを連続して並列に実行する場合に再現性のある結果を生成するための新しい機能を提供しています。詳細は、インテル® Parallel Studio XE に含まれるインテル® MKL ドキュメントの「数値再現性のある結果を得る」セクションとインテル® TBB ドキュメントの「parallel_deterministic_reduce テンプレート関数」を参照してください。

ベクトル化と OpenMP* を有効にするコンパイラー・オプションはインテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能ですが、インテル製マイクロプロセッサーにおいてより多くの最適化が行われる場合があります。

WRF の例 2

同じデータと同じプロセッサーで、(スレッド化されていない) バイナリーを複数回実行すると、わずかに異なる結果が生成されました。

原因は、外部のイベントにより、実行ごとにグローバルスタックの開始アドレスとアライメントが異なるためでした。ローカルスタックのアライメントが変更され、ループのプロローグ、ベクトル化された本体、エピローグに含まれるループ反復が変わりました。そして、ベクトル化されたリダクション操作 (つまり再結合) の順序が変更されたのです。

この問題を解決するには、リダクション操作のベクトル化を無効にする /fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) オプションを指定してコンパイルします。

インテル® コンパイラー 11 以降では、グローバルスタックの開始アドレスはキャッシュライン境界でアライメントされます。これは、アプリケーション内部のイベントが原因で実行ごとにスタックのアライメントが変わる場合 (例えば、現在の日時を格納するため、スタック上に可変長文字列を割り当てる場合) を除き、/fp:fast (Windows®) または -fp-model fast (Linux* および OS X*) を指定した場合でも、前述の実行ごとの結果のばらつきを回避するためです。ヒープ・アライメントの動的な変更も、同様に浮動小数点演算結果に影響します。このようなアライメントの変更は、通常、外部環境に依存するメモリー割り当てによって生じます。

/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) を指定してコンパイルするか、データ配列を明示的にアライメントすることで、浮動小数点演算結果のばらつきを防ぐことができます。インテル® コンパイラー 15 以降では、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) よりもパフォーマンスへの影響が少ない、/Qopt-dynamic-align- (Windows®) または -qno-opt-dynamic-align (Linux* および OS X*) を使用することもできます。

突発アンダーフローと FTZ (Flush-To-Zero)

正規化されていない数[4] (非正規化数) は、浮動小数点の指数範囲をわずかに超えることがあります。非正規化数を扱う計算は、正規化数のみを扱う計算よりも処理に長い時間を要します。デフォルトでは、浮動小数点演算結果が非正規化数の場合、ハードウェアでは 0 に設定されます。/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) を指定すると、精度を維持するため、非正規化数の結果が保持されます。

/fp: (Windows®) または -fp-model (Linux* および OS X*) は、浮動小数点コントロール・レジスターのハードウェア FTZ モードを設定/設定解除する /Qftz または /Qftz- (Windows®) あるいは -ftz または -no-ftz (Linux* および OS X*) を指定してメイン関数やルーチンをコンパイルすると、プログラム全体にわたって無効になります。[5] /fp:fast (Windows®) または -fp-model fast (Linux* および OS X*) のデフォルト設定は、最適化レベル -O1 以上の場合 -ftz です。

x87 演算命令ではハードウェア FTZ を利用できないため、-ftz オプションは無視されます。x87 演算命令は、通常、/arch:ia32 (Windows®) または -mia32 (Linux*) を指定してインテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2) をサポートしない古い IA-32 アーキテクチャー・ベースのプロセッサー向けにコンパイルする場合など、特殊なケースでのみ生成されます。

[4] 非正規化数の概要は、『インテル® コンパイラー・デベロッパー・ガイドおよびリファレンス』の「浮動小数点演算」セクションを参照してください。

[5] /Qftz (Windows®) または -ftz (Linux* および OS X*) オプションは、非正規化数の結果の FTZ を許可しますが、常にゼロにフラッシュされる保証はありません。

浮動小数点式の評価

例:　　a = (b + c) + d

C99 では、FLT_EVAL_METHOD の値に応じて、中間結果 (b+c) の丸め方法が 4 つあります。

評価方法	/fp: (Windows®) または -fp-model (Linux* および OS X*)	言語	FLT_EVAL_METHOD
中間結果の精度	fast	C/C++/Fortran	-1
ソースの精度	source	C/C++/Fortran	0
倍精度	double	C/C++	1
拡張倍精度 (long double)	extended	C/C++	2

/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) が指定され、評価方法が指定されていない場合、インテル® 64 アーキテクチャーとインテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーでは、デフォルトで source が使用されます。IA-32 アーキテクチャー上の C/C++ では、デフォルトで double (Windows®) または extended (Linux*)[6] が使用されます。Fortran では、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) で source のみサポートされます。source、double、extended が指定され、値の安全性が指定されていない場合、値の安全性にはデフォルトで /fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) が使用されます。式の評価方法は、パフォーマンス、精度、再現性、移植性に影響します。特に、異なる精度の表現を繰り返し変換する評価方法の選択は、パフォーマンスに大きく影響します。

浮動小数点演算ユニット (FPU) 環境

浮動小数点演算環境[7] は、浮動小数点コントロール・ワードの設定とステータスフラグで構成されます。コントロール・ワードの設定は、以下を制御します。

- FP 丸めモード (近似、正の無限大方向、負の無限大方向、ゼロ方向)
- FP 例外マスク (不正確、アンダーフロー、オーバーフロー、ゼロ除算、非正規化数、無効な例外)
- FTZ (Flush-to-Zero)、DAZ (Denormals-are-Zero)
- x87[8] のみ: 精度コントロール (single、double、extended)
    - 変更すると意図しない問題が発生する可能性があります。

各例外マスクには、対応するステータスフラグがあります。

FPU 環境へのプログラマーのアクセスは、デフォルトでは許可されていません。

- コンパイラーは、デフォルトの FPU 環境を仮定します。
    - 最近値への丸め
    - すべての FP 例外がマスクされる
    - FTZ (Flush-to-Zero) と DAZ (Denormals-as-Zero) が無効
- コンパイラーは、プログラムが FP ステータスフラグの読み取りを行わないことを仮定します。

ユーザーが (例えば、FP コントロール・ワードを変更するランタイム・ライブラリーの呼び出しなどにより) 明示的にデフォルトの FPU 環境を変更する場合、FPU 環境アクセスモードを設定してコンパイラーに知らせ

[6] -mia32 は、OS X* ではサポートされません。すべてのインテル® プロセッサーは、OS X* でインテル® SSE3 までサポートしています。そのため、-fp-model precise の評価方法は、デフォルトで source になります。

[7] 詳細は、『インテル® コンパイラー・デベロッパー・ガイドおよびリファレンス』の「浮動小数点演算」 > 「浮動小数点の理解」 > 「浮動小数点環境」を参照してください。

[8] x87 浮動小数点演算操作用のコントロール・ワードは異なります。古いプロセッサーをサポートするため /arch:IA32 (Windows®) または -mia32 (Linux*) を指定する場合を除き、通常、x87 FP コントロール・ワードについて考慮する必要はありません。

る必要があります。アクセスモードは、次のいずれかの方法によって、安全な値モードでのみ有効にできます。

- /fp:strict (Windows®) または -fp-model strict (Linux* および OS X*)
- #pragma STDC FENV_ACCESS ON (C/C++ のみ)

これにより、コンパイラーは FPU コントロール設定を不明として扱います。浮動小数点ステータスフラグが保持され、コンパイル時の定数式の評価、浮動小数点演算やその他の演算のスペキュレーションのような特定の最適化が無効になります。[9] コンパイラーに知らせずにデフォルトの浮動小数点演算環境を変更すると、例えば丸めモードの変更により、数学ライブラリー関数などで予測できない結果が生じることがあります。

FPU 環境を変更する例

```
#include <fenv.h>
double x, zero = 0.;
feenableexcept(FE_DIVBYZERO);
    for( int i = 0; i < 20; i++ )
        for( int j = 0; j < 20; j++)
            x = zero ? (1./zero) : zero;
    .....
```

最適化により、「?」の分岐先がループ内に残されたまま、(1./Zero) の計算がループからホイストされ 1 回しか評価されなくなったため、明示的に保護していたにもかかわらず浮動小数点例外が発生しました。デフォルトの FPU 環境ではゼロ除算例外はマークされるため、コンパイラーはこれが安全であると仮定しました。上記の feenableexcept() の呼び出しのように、デフォルトの環境が変更された場合は、/fp:strict (Windows®) または -fp-model strict (Linux* および OS X*) オプションを指定してコンパイルするか、次のプラグマを使用してコンパイラーに知らせるべきです。

#pragma STDC FENV_ACCESS ON (C/C++ のみ)

正しくない例外を引き起こす最適化は、/Qfp-speculation:safe (Windows®) または -fp-speculation safe (Linux* および OS X*) オプションを指定することで、直接無効にすることもできます。これにより、FMA 命令を抑止したり、リダクション操作や数学関数を含むループをベクトル化しないようにする、/fp:strict (Windows®) または -fp-model strict (Linux* および OS X*) の影響の一部を回避することができます。

以下は、よくある例です。

```
double *a, *b;
    for(int i = 0; i < 100; i++)
        if (a[i] != 0.) b[i] = b[i] / a[i]
```

デフォルトの最適化では、コンパイラーはインテル® SSE 命令を使用してこのループをベクトル化します。パックド SIMD 命令を使用して i のすべての値について b[i]/a[i] を評価し、マスクが true の結果のみ b[i] に格納します。この操作は、デフォルトの環境でゼロ除算例外がマスクされているため安全です。この例外が、feenableexcept() の呼び出しや /Qfp-trap:common (Windows®) または -fp-trap =common (Linux* および OS X*) オプションによって明示的にマスクされていない場合、例外はトラップされ、プログラムは終了します。これは、スペキュレーションによって例外が生じる可能性がある場合ベクトル化を行わない /Qfp-speculation:safe (Windows®) または -fp-speculation safe (Linux* および OS X*) を指定してコンパイ

[9] 無効になるほかの最適化:
部分冗長の排除
共通の部分式の排除
不要コードの排除
条件付き変換、例: if (c) x = y; else x = z; ➔ x = (c) ? y : z;

ルすることで回避できます。インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) のような最近の命令セットには、ハードウェアでマスクされる SIMD 命令が含まれています。これらの命令セットでは、上記のようなループをスペキュレーションなしでベクトル化することができます。

厳密な浮動小数点例外

デフォルトでは厳密な例外は無効に設定されており、コードは最適化時にコンパイラーによって並べ替えられます。そのため、ソースコードに記述されているとおりに実行した場合と同じタイミングや位置で浮動小数点例外が発生するとは限りません。これは、インテル® SSE やインテル® AVX のように例外が直ちに通知されない x87演算命令において特に重要となります。

次のいずれかの方法で厳密な FP 例外を有効にすることができます。

- /fp:strict (Windows®) または -fp-model strict (Linux* および OS X*)
- /fp:except (Windows®) または -fp-model except (Linux* および OS X*)
- #pragma float_control (except, on) (C/C++ のみ)

有効にすると、コンパイラーは、浮動小数点演算が例外をスローする可能性を考慮する必要があります。FP 演算のスペキュレーションのような最適化は、分岐しなければ実行されることがない領域で例外が生じる可能性があるため、無効になります。例えば、「if」文を含む特定のループのベクトル化が妨げられる可能性があります。コンパイラーは、ほかの x87 命令の後に fwait を挿入して、FP 例外と例外が発生した命令を紐づけられるようにします。厳密な FP 例外は、安全な値モード (/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*)、あるいは #pragma float_control (precise, on)) でのみ有効にすることができます。値の安全性は、/fp:strict (Windows®) または -fp-model strict (Linux* および OS X*) によって保証されます。

厳密な FP 例外を有効にしても、FP 例外のマスクが解除されるわけではありません。マスクの解除は、関数呼び出し、Fortran では /fpe:0、/fpe-all:0 (Windows®) または -fpe0、-fpe-all0 (Linux* および OS X*)、C/C++ では /Qfp-trap:common、/Qfp-trap-all:common (Windows®) または -fp-trap=common、-fp-trap-all=common (Linux* および OS X*) を使用して、別途行う必要があります。

浮動小数点の縮約

これは、主にインテル® AVX2/インテル® AVX-512 命令セットやインテル® MIC アーキテクチャーで利用可能な FMA (Fused Multiply-Add) 命令の生成に関連しています。FMA 命令の生成は、デフォルトで有効に設定されています。コンパイラーは、乗算と加算を 1 つの命令で行う FMA 命令を生成することがあります。

例: a = b*c + d

この命令を使用することで、高速により正確に計算を行うことができますが、乗算と加算を個別に行う場合と比べて、最後のビット表現が異なることがあります。

浮動小数点の縮約は、/QxCORE-AVX2 や /Qmic (Windows®) または -xcore-avx2 や -mmic (Linux* および OS X*) のようなコンパイラー・オプションを指定して、FMA 命令をサポートするプロセッサー向けに最適化レベル /O1 (Windows®) または -O1 (Linux* および OS X*) 以上でコンパイルする場合、デフォルトで有効になります。これは、古いプロセッサーと浮動小数点演算結果が異なる一般的な原因です。FMA 命令の生成は、次のいずれかの方法によって、ソースファイルまたは関数レベルで無効にすることができます。

- /fp:strict (Windows®) または -fp-model strict (Linux* および OS X*)
- #pragma float_control (fma, off) (C/C++)
- #pragma fp_contract (off) (C/C++)
- !DIR$ NOFMA (Fortran)
- /Qfma- (Windows®) または -no-fma (Linux* および OS X*) (/fp または -fp-model をオーバーライド)

無効にすると、コンパイラーは乗算命令と加算命令を個別に生成し、中間結果を丸める必要があります。FMA 命令の生成は、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) では無効になりません。

インテル® C/C++ コンパイラーは、FMA 組込み関数をサポートするプロセッサー向けに、FMA 用の SIMD 組込み関数 (例えば、_mm256_fmadd_pd() など) の生成をサポートしています。ただし、組込み関数の生成は、コンパイラーによる最適化に依存します。

連続するソース行で乗算と加算を個別に記述しても、FMA 命令の生成を阻止することはできません。デバッグ向けに、「メモリーフェンス」組込み関数を使用し、中間結果をメモリーに格納するように強制するなどして、FMA 命令が生成されないようにすることができます。

```
t = a*b;
_mm_mfence();
result = t + c;
```

math.h の fma() と fmaf() 組込み関数では、丸め処理を 1 回にすべきです。FMA 命令がサポートされていないプロセッサー上でも同じことが言えますが、パフォーマンスに影響します。

FMA 命令は、式の対称性を損なうことがあります。次の例について考えてみます。

```
c = a;  d = -b;
result = a*b + c*d;
```

FMA 命令が利用できない場合、通常、結果はゼロになります。FMA 命令がサポートされている場合、コンパイラーは次のいずれかに変換する可能性があります。

result = fma(c, d, (a*b)) または result = fma(a, b, (c*d))

丸め処理の違いにより、この 2 つの式の結果は、非ゼロまたは異なる可能性があります。

/fp:precise /fp:source (Windows®) または -fp-model precise -fp-model source (Linux* および OS X*) の典型的なパフォーマンスへの影響

/fp:precise /fp:source /Qftz (Windows®) または -fp-model precise -fp-model source -ftz (Linux* および OS X*) は、非正規化数を保持する必要がない通常のアプリケーションでは、パフォーマンスへの影響を抑えつつ浮動小数点演算結果の再現性を向上するため、推奨されます。/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) は、特定の最適化を無効にするため、アプリケーションのパフォーマンスが低下します。パフォーマンスへの影響は、アプリケーションによって大きく異なります。高度なループの最適化が有効なアプリケーションでは、通常、浮動小数点演算の並べ替えが行われるため、パフォーマンスへの影響が最も大きくなります。

SPEC CPU* 2006 の SPECfp* 2006 ベンチマーク・スイートのパフォーマンス評価を使用して、影響について見てみましょう。/O2 または /O3 (Windows®) あるいは -O2 または -O3 (Linux* および OS X*) でコンパイルした場合、/fp:precise /fp:source (Windows®) または -fp-model precise -fp-model source (Linux* および OS X*) によるパフォーマンス低下の相乗平均は 12 ～ 15% でした。テストには、インテル® Xeon® プロセッサー 5650 (6 コア、2.67GHz、24GB メモリー、12MB キャッシュ)、SuSE* Linux* Enterprise Server (SLES) 10 SP2 x64 バージョン、インテル® コンパイラー 12.0 を使用しました。-O3 オプションはインテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能ですが、インテル製マイクロプロセッサーにおいてより多くの最適化が行われる場合があります。

付記

/fp:precise /fp:source (Windows®) または -fp-model precise -fp-model source (Linux* および OS X*) は、/Od (Windows®) または -O0 (Linux* および OS X*) でデバッグビルドでも使用すべきです。/Od (Windows®) または -O0 (Linux* および OS X*) では、評価方法がデフォルトで source になりません。

ここで紹介した浮動小数点モデルは、インテル® MIC アーキテクチャーにも適用できますが、インテル® Xeon Phi™ コプロセッサー x100 ファミリーの実装とは一部異なる点があります。詳細は、「関連情報」セクションにある「インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーの浮動小数点演算の違い」を参照してください。

数学ライブラリー関数

現時点で、log() や sin() のような数学関数[10] の精度や、結果の丸め方法を指定している標準仕様はありません。これらの関数の異なる実装では、精度や丸め方法が異なることがあります。

インテル® コンパイラーは、次のように数学関数を実装します。

- 最適化されたインテルの数学ライブラリー libm (Windows®) または libimf (Linux* および OS X*) の標準呼び出し。これらの呼び出しのほとんどは、Microsoft® C ランタイム・ライブラリー libc (Windows®) または GNU* ライブラリー libm (Linux* および OS X*) の数学関数と互換性があります。
- 後のコンパイルフェーズで最適化可能なインラインコードの生成。
- アーキテクチャー固有の呼び出しシーケンス (例えば、インテル® SSE2 対応の IA-32 アーキテクチャー・ベースのプロセッサーでは SIMD レジスターを利用して引数を渡すなど)
- ベクトル化可能なループについて SVML (libsvml) の呼び出し。

/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) を指定すると、上記の 1 つ目の呼び出しのみに制限されます。/Qfast-transcendentals- (Windows®) または -no-fast-transcendentals (Linux* および OS X*) のように個別のオプションにより制限することもできます。これにより、異なる最適化レベルや異なるコンパイラー・バージョンで、コンパイラーによって生成される呼び出しシーケンスの一貫性を保つことができます。ただし、ライブラリー関数自体の動作の一貫性は保証されません。次の場合、数学ライブラリー関数によって返される値が異なることがあります。

- 異なるコンパイラー・リリース。アルゴリズムと最適化の改善により異なります。
- 異なるランタイム・プロセッサー。数学ライブラリーには、各プロセッサー向けに異なる最適化が適用された関数の実装が含まれています。コードは、実行時にプロセッサーの種類を自動検出し、適切な実装を選択します。例えば、複素数演算を含む関数にインテル® SSE3 命令対応の実装とそうでない実装がある場合、インテル® SSE3 命令対応の実装は、インテル® SSE3 命令をサポートするプロセッサーでのみ呼び出されます。

数学関数の結果の差はわずかです。異なるコンパイラー・リリース、異なるプロセッサー向けに最適化された実装の両方において、想定される差は、標準の数学ライブラリーでは 0.55ulp、ベクトル化されたループで使用される SVML では 4ulp です。

異なるコンパイラー・リリースの数学ライブラリーで、ビット単位の一貫性を強制する直接的な方法はありません。場合によっては、2 つのコンパイラーとより新しいほうのランタイム・ライブラリーを組み合わせて、コンパイラーによって生成されたコードをチェックすることができます。

/Qimf-arch-consistency:true (Windows®) または -fimf-arch-consistency=true (Linux* および OS X*) オプションにより、インテル製マイクロプロセッサーと互換マイクロプロセッサーを含む、同じアーキテクチャーの異なるプロセッサーで数学ライブラリー関数によって返される結果のビット単位の一貫性を保証できます。

[10] 除算関数、平方根関数を除く

このオプションは、IA-32 とインテル® 64、あるいはインテル® 64 とインテル® MIC のように異なるアーキテクチャー間でビット単位の一貫性を保証しません。また、さまざまなプロセッサーで実行できるように高度に最適化されていない関数を呼び出すため、パフォーマンスが低下する可能性があります。

数学関数の結果の丸め方法が指定された標準仕様を採用することで、異なるアーキテクチャー間を含め、浮動小数点演算結果の一貫性が改善されるでしょう。ただし、パフォーマンスは低下するでしょう。

インテル® コンパイラーには、再現性には直接影響しない /Qimf-precision、/Qimf-max- error (Windows®) または -fimf-precision、-fimf-max-error (Linux* および OS X*) のような数学関数の結果の精度を制御するほかのオプションがあります。これらのオプションについては、『インテル® コンパイラー・デベロッパー・ガイドおよびリファレンス』を参照してください。

log() や sin() のような数学関数を含むループのベクトル化は、標準の数学ライブラリーよりもわずかに精度が低い結果を返す異なる数学ライブラリー関数を呼び出すため、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) によって無効にできます。/Qfast-transcendentals (Windows®) または -fast-transcendentals (Linux* および OS X*) によりベクトル化を再度有効にし、/Qimf-precision (Windows®) または -fimf-precision (Linux* および OS X*) によりベクトル化された数学関数の精度を制御できます。

同様に、/fp:precise (Windows®) または -fp-model precise (Linux* および OS X*) は、平方根と除算で正しく丸められた一貫した結果の生成を保証する /Qprec-sqrt と /Qprec-div (Windows®) または -prec-sqrt と -prec-div (Linux* および OS X*) を有効にします。これにより、除算関数や平方根関数を含むループのベクトル化が無効になることがあります。/Qprec-sqrt- と /Qprec-div- (Windows®) または -no-prec-sqrt と -no-prec-div (Linux* および OS X*) によりベクトル化を再度有効にし、/Qimf-precision (Windows®) または -fimf-precision (Linux* および OS X*) により正しいコードシーケンスが生成されるように制御できます。

多くの数学ライブラリー関数は、互換マイクロプロセッサーよりもインテル製マイクロプロセッサーでより高度に最適化されることがあります。インテル® コンパイラーの数学ライブラリーは、選択されたオプション、コード、およびその他の要因に基づいてインテル製マイクロプロセッサーおよび互換マイクロプロセッサー向けに最適化されますが、インテル製マイクロプロセッサーにおいてより優れたパフォーマンスが得られる傾向にあります。

結論

コンパイラー・オプションで、精度、再現性、パフォーマンスのトレードオフを制御することができます。パフォーマンスに与える影響を抑えながら浮動小数点の一貫性と再現性を改善するには、/fp:precise /fp:source (Windows®) または -fp-model precise -fp-model source(Linux* および OS X*) を使用します。[11] 同じアーキテクチャーの異なるプロセッサーにおける再現性が重要な場合は、/Qimf-arch-consistency:true (Windows®) または -fimf-arch-consistency=true (Linux* および OS X*) も使用します。異なるプロセッサーの少なくとも 1 つが FMA 命令をサポートしている場合、最良の再現性を得るためには /Qfma- (Windows®) または -no-fma (Linux* および OS X*) を指定します。

インテル® コンパイラー 17 では、/fp:consistent (Windows®) または -fp-model consistent (Linux* および OS X*) を指定するだけで、上記のオプションがすべて設定され、最良の再現性が得られます。

[11] /fp:source を指定すると /fp:precise も有効になります。

関連情報

- Microsoft® Visual C++® 浮動小数点の最適化: http://msdn2.microsoft.com/en-us/library/aa289157(vs.71).aspx (英語)
- インテル® MKL 条件付き数値再現性: https://software.intel.com/en-us/articles/conditional-numerical-reproducibility-cnr-in-intel-mkl-110 (英語)
- インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーの浮動小数点演算の違い http://www.isus.jp/products/psxe/differences-in-floating-point-arithmetic/
- 『インテル® C++ および Fortran コンパイラー・デベロッパー・ガイドおよびリファレンス』の「浮動小数点演算」セクション
- Goldberg, David: "What Every Computer Scientist Should Know About Floating-Point Arithmetic“ *Computing Surveys*, March 1991, pg.203

最適化に関する注意事項

インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。
本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。

注意事項の改訂 #20110804

付録

インテル® コンパイラーの主な浮動小数点オプション

オプション	説明
/fp:keyword **-fp-model keyword**	**fast**[=**1**|2]、precise、except、strict、consistent、source [double、extended - C/C++ のみ] 浮動小数点セマンティクスを制御します。
/Qftz[-] -[no-]ftz	結果が非正規化数の場合、ゼロにフラッシュします。
その他のオプション	
/Qfast-transcendentals[-] -[no-]fast-transcendentals	高速な数学関数の使用を有効 [無効] にします。
/Qprec-div[-] -[no-]prec-div	浮動小数点除算の精度を上げます。
/Qprec-sqrt[-] -[no-]prec-sqrt	平方根計算の精度を上げます。
/Qfp-speculation keyword -fp-speculation keyword	**fast**、safe、strict、off 浮動小数点演算のスペキュレーションを制御します。
/fpe:0 -fpe0	浮動小数点例外のマスクを解除し (Fortran のみ)、非正規化数の生成を無効にします。
/Qfp-trap:common -fp-trap=common	一般的な浮動小数点例外のマスクを解除します (C/C++ のみ)。
/Qfp-port -fp-port	浮動小数点演算結果をユーザー精度に丸めます。
/Qprec -mp1	比較/超越関数の一貫性を上げます。
/Qimf-precision:name -fimf-precision=name	high、medium、low 数学ライブラリー関数の精度を制御します。
/Qimf-arch-consistency:true -fimf-arch-consistency=true	数学ライブラリー関数が同じアーキテクチャーの異なるプロセッサーで一貫した結果を生成するようにします。
/Qfma[-] -[no-]fma	FMA (Fused Multiply-Add) 命令の使用を有効 [無効] にします。
/Qopt-dynamic-align[-] -q[no-]opt-dynamic-align	実行ごとに浮動小数点演算結果が異なる可能性がある動的データ・アライメントの最適化を有効 [無効] にします。
/assume [no]protect_parens -assume [no]protect_parens	式の評価順序を決定する際にコンパイラーが括弧を考慮すべきか [すべきでないか] を指定します。(Fortran のみ)
/Qprotect-parens[-] -f[no-]protect-parens	式の評価順序を決定する際にコンパイラーが括弧を考慮すべきか [すべきでないか] を指定します。(C/C++ のみ)