明示的なベクトル・プログラミング - 最も一般的な手法

この記事は、インテル® デベロッパー・ゾーンに掲載されている「Explicit Vector Programming – Best Known Methods」の日本語参考訳です。

明示的なベクトル・プログラミング - 最も一般的な手法

アプリケーションのベクトル化に関心が寄せられているのはなぜでしょうか。答えは、ベクトル化によりパフォーマンスが向上し、消費電力を抑えることができるためです。アプリケーションが CPU 負荷の高い領域を高速に計算できるようになれば、より迅速に CPU を低消費電力状態へ移行できます。

ベクトル演算とスカラー演算のパフォーマンスと消費電力を比較した場合、ベクトル演算は効率的なため、等価なスカラー演算よりも消費電力が少なくて済みます。一方、スカラー演算はサイクルごとにデータを数回処理し、完了までにより多くの命令とサイクルを必要とします。

x86 プラットフォームにより幅の広いベクトルレジスターが導入され、SIMD (Single Instruction Multiple Data) と並列処理 (スレッド化) をサポートするコア数が増加したことにより、ベクトル化は開発者が最適化を行うときに考慮すべき事柄の 1 つとなっています。ベクトル化によるパフォーマンス向上はコアごとにもたらされるため、多くのアプリケーションでパフォーマンスが飛躍的に増加します。これまで、多くの開発者はループのベクトル化をコンパイラーに任せていましたが、プログラミング言語の制約により、コンパイラーはすべてのループをベクトル化できるわけではありませんでした。そこで、ベクトル化でリダクションをサポートする、明示的なベクトル・プログラミング手法が求められました。

- 外部ループ
- ユーザー定義関数を含むループ
- コンパイラーによりデータ依存性が仮定されるが、無害であると開発者が分かっているループ

これらのループをベクトル化することでパフォーマンスが向上し、消費電力も抑えることができます。

この記事では、明示的なベクトル・プログラミングを使用して、ベクトル・プロセシング・ユニット (VPU) を備えた最近のプロセッサーで、CPU 依存のアプリケーションのパフォーマンスを向上する一般的な手法 (BKM) を説明します。多くの場合、スレッドレベルの並列処理と SIMD レベルの並列処理の両方に対応するように、構造の変更を検討すると良いでしょう。

CPU 依存のアプリケーションのパフォーマンスを向上するには、次のステップで作業を進めます。

1. ベースラインのアプリケーション・パフォーマンスを測定します。
2. インテル® VTune™ Amplifier で hotspot 解析と全般解析を実行します。
3. 最も時間を費やしているループ/関数が SIMD 並列処理に適しているかどうか判断します。
4. 明示的なベクトル・プログラミング手法を使用して SIMD 並列処理を実装します。
5. SIMD パフォーマンスを測定します。
6. オプション: アセンブリー・コードを生成してコードの効率を調査します。
7. 上記のステップを繰り返します。

ステップ 1: ベースラインのアプリケーション・パフォーマンスを測定する

最初に、ベクトル化の変更が有効かどうか判断するため、基準となる現在のアプリケーション・パフォーマンスのベースラインが必要です。さらに、開始点からの進捗と最終的なアプリケーション・パフォーマンスを測定するためのベースラインも必要です。これらのベースラインを知ることで、最適化を終了するタイミングを判断できます。

最初のベースラインには、アプリケーションのデバッグビルドではなく、リリースビルドを使用します。リリースビルドには、最終的なアプリケーションの最適化がすべて含まれています。アプリケーションのループや hotspot が時間を費やしていることを理解するには、リリースビルドを使用することが重要です。

リリース・ベースラインでは、シンボル情報が提供され、*simd* (明示的なベクトル化) と *vec* (自動ベクトル化) を除く、すべての最適化が有効になります。simd と自動ベクトル化を明示的に無効にするには、*-no-simd -no-vec* (Linux*)、*/Qsimd- /Qvec-* (Windows*) コンパイラー・オプションを使用します。(『インテル® C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』 (英語) と『インテル® 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』 (英語) を参照)

ベースラインとベクトルバージョンのパフォーマンスを比較し、チューニングによりどれぐらい理論的な最大値に近い値が得られるか調べます。

インテル® VTune™ Amplifier やインテル® Advisor のようなツールや print 文を使用して、ベースラインとベクトルバージョンで特定のループのパフォーマンス (処理時間) を比較すると良いでしょう。

ステップ 2: インテル® VTune™ Amplifier で hotspot 解析と全般解析を実行する

インテル® VTune™ Amplifier を使用して、アプリケーションで最も時間を費やしている関数を特定します。これには、hotspot 解析タイプを推奨します。

最も時間を費やしているアプリケーション領域を特定することで、最も効果が得られる領域の最適化に専念できます。一般に、アプリケーションの合計実行時間の少なくとも 10% の時間を費やしている上位の hotspot や関数を最適化します。次のステップで最適化する hotspot をメモします。(チュートリアル: 「hotspot の検出」 (英語))

マイクロアーキテクチャー解析 (Microarchitecture Analysis) の全般解析 (General Exploration) レポートでは、次の情報が提供されます。

- TLB ミス (コンパイラーのプロファイルに基づく最適化を検討してみてください)
- L1 データ・キャッシュ・ミス (キャッシュの局所性およびストリーミング・ストアの使用を検討してみてください)
- 分割ロードと分割ストア (ターゲット・アーキテクチャー向けのデータ・アライメントを検討してみてください)
- メモリー帯域幅
- アプリケーションで要求されたメモリー・レイテンシー (ストリーミング・ストアとプリフェッチを検討してみてください)

全般解析は、ベクトル化を対象としたチューニングを継続することが有益かどうか判断するのに役立ちます。

ステップ 3: 最も時間を費やしているループ/関数が SIMD 並列処理に適しているかどうか判断する

ベクトル化するループを選択する際に重要なことは、ループ反復のメモリー参照が互いに独立しているかどうかです。(「ベクトルループ内のメモリーの一義化を制御する」および「ループをベクトル化するための条件」を参照してください。)

インテル® コンパイラーの最適化レポート (*-qopt-report -qopt-report-phase=vec*) は、コードの各ループがベクトル化されたかどうかを示します。コンパイラーによる自動ベクトル化を有効にするには、最適化レベル 2 または 3 (*-O2* または *-O3*) を使用していることを確認します。ベクトル化レポートを取得して、ステップ 2 で選択した hotspot に関する情報を調べます。これらの hotspot にベクトル化されなかったループが存在する場合、並列に (例えば、配列で) データの演算、データ処理、文字列計算を行っているかどうか確認します。これらの処理を行っている場合は、ベクトル化の候補となります。ベクトル化の候補が見つかったら、ステップ 4 に進みます。

データ・アライメント

データ・アライメントも、ベクトル化を最大限に活用するために重要です。インテル® VTune™ Amplifier によって分割ロードと分割ストアがレポートされた場合、アプリケーションはアライメントされていないデータを参照しています。データ・アライメントの最適化では、特定のバイト境界上のメモリーにデータ・オブジェクトを生成するようにコンパイラーに指示します。データ・アライメントを行う際は、次の 2 つの点に注意する必要があります。

1. 特定のバイト・アライメント条件で配列を作成する。
2. パフォーマンス・クリティカルな領域に、アライメントを指定するプラグマ/ディレクティブおよび節を挿入する。

アライメントは、データのロード/ストアの効率を高めます。インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2) プラットフォームをターゲットにする場合は、SSE でアライメントされたロード命令が使用されるように 16 バイト境界でアライメントします。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) 命令セットをターゲットにする場合は、32 バイト境界でアライメントします。インテル® AVX-512 命令をサポートするプロセッサーでは、64 バイト境界が最適です。(「ベクトル化の可能性を高めるデータ・アライメント」を参照してください。)

ユニットストライド

コンパイラーによるベクトル化を支援するには、構造体配列 (AoS) やアルゴリズムの最適化ではなく、ユニット・ストライド・メモリー (アドレス・シーケンシャル・メモリーとも呼ばれます) アクセスと配列構造体 (SoA) の使用を検討します。『インテル® C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』 (英語) に例があります (「Using Structure of Arrays versus Array of Structures」を検索)。同様の概念は、Fortran にも当てはまります。

一般に、メモリーを参照するときはユニットストライド方式でデータにアクセスするのが最適です。この方式は、ベクトル化や並列プログラミング手法に適しています。

ベクトル化の成否は、ループ変換 (キャッシュの局所性の情報を参照) やループのアンロールのような、アプリケーションのほかのループ最適化に依存します。

-ip や *-ipo* (Linux*)、*/Qip* や */Qipo* (Windows*) で関数をインライン展開し、ユーザー定義関数を含むループのベクトル化が行われるかどうか確認してみると良いでしょう。これは SIMD 対応関数の使用に代わるアプローチです。どちらのアプローチを使用する場合もトレードオフがあります。

注:
hotspot 解析を実行し、アルゴリズムが計算依存の場合は、後続のステップを実行します。アルゴリズムがメモリー・レイテンシー依存またはメモリー帯域幅依存の場合は、ベクトル化を行うメリットはありません。この場合、キャッシュの最適化やほかのメモリー関連の最適化を行うか、アルゴリズム全体を見直してみてください。*-O3* のようなハイレベルのループ最適化を行うと、キャッシュの局所性問題の解決に役立つループ交換の最適化を適用できることがあります。適用可能な場合、キャッシュ・ブロッキングもキャッシュの局所性向上に役立ちます。キャッシュ・ブロッキング手法については、「ループの最適化: ブロックが必要な場所 (英語) を参照してください。

ステップ 4: 明示的なベクトル・プログラミング手法を使用して SIMD 並列処理を実装する

明示的なベクトル・プログラミングには、OpenMP* 4.0 (英語) のベクトル化ディレクティブのような機能が含まれます。これらは、C/C++ アプリケーションにおけるベクトル化の可能性を表現する、非常に強力で移植性に優れた方法を提供します。OpenMP* 4.0 のベクトル化ディレクティブは、Fortran アプリケーションでも利用できます。これらの明示的なベクトル・プログラミング手法により、コンパイラーにベクトル化するループを指示する手段が提供されます。ベクトル化ディレクティブの候補には、コンパイラーによる依存関係の確認で非常に多くのメモリー参照が行われるループ、リダクションを含むループ、ユーザー定義関数を含むループ、外部ループなどが含まれます。

(OpenMP* 4.0 を使用してアプリケーションで SIMD 機能を有効にする方法については、「Fortran の明示的なベクトル・プログラミング」 (英語) を参照してください。)

明示的なベクトル・プログラミングで利用可能なコンポーネントを次に示します。

SIMD 対応関数

SIMD 対応関数のユーザー生成は、OpenMP* 4.0 以降で提供される機能です。SIMD 対応関数は、呼び出しに応じた SIMD 動作の変更を含む、ユーザー定義関数の SIMD 動作を明示的に示します。『インテル® C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』では、SIMD 対応関数 (英語) の使用について説明しています。『インテル® 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』では、!$OMP DECLARE SIMD (ルーチン名) (英語) について説明しています。

SIMD ループ

SIMD ループの本質は、さまざまな使用法やリダクションなどの記述を含む、ループの SIMD 動作を明示的に表すことです。

従来は、内部ループのみベクトル化のターゲットにされていましたが、OpenMP* 4.0 の *#pragma omp simd* を使用するアプリケーションは、外部ループにも適用できます。

(外部ループでの #pragma simd の使用については、「外部ループのベクトル化」を参照してください。)

ステップ 5: SIMD パフォーマンスを測定する

最適化後のアプリケーションを実行してパフォーマンスを測定します。目標が達成できた場合は、これで完了です。目標が達成できなかった場合は、*-qopt-report-phase=vec* オプションを使用して生成した最適化レポートの SIMD ベクトル化サマリーで、アライメント、ユニットストライドと SoA/AoS の使用、ほかのループ最適化との調整、その他を確認します。

パフォーマンスの測定についての詳細は、「インテル® IA-32/IA-64 命令セット・アーキテクチャーのコード実行時間の測定方法」 (英語) を参照してください。

別のアプローチは、*-profile-xxxx* コンパイラー・オプション (*-profile-functions* (Linux*) や */Qprofile-functions* (Windows*) など) を使用することです。関数またはループの実行時間をプロファイルするインストルメンテーション手法を使用すると、アプリケーションでサイクルが費やされている場所を簡単に確認することができます。インテル® コンパイラーは、アプリケーションにインストルメンテーション・コードを挿入して、さまざまな場所で費やされた時間を収集します。このデータは、最適化チューニングまたは並列化の候補となる hotspot を特定するのに役立ちます。

パフォーマンスを測定する別の手法は、最適化を行った後にインテル® VTune™ Amplifier の hotspot 解析を再実行して結果を比較することです。

ステップ 6 (オプション): アセンブリー・コードを生成して調査する

コンパイラーが生成するアセンブリー・コードを確認して、アプリケーションが適切にベクトル化されたかどうかコードを調査するには、*-S* コンパイラー・オプションを使用してアセンブリー・ファイルを生成します。

ステップ 7: 繰り返す

目的のパフォーマンスを達成するか、適切な候補がなくなるまで、最適化ステップを繰り返します。

次に示すその他の考慮事項は、メモリー・レイテンシー依存またはメモリー帯域幅依存のアプリケーションに利用できます。

その他の考慮事項: ストリーミング・ストア

ストリーミング・ストアは、書き込むデータが以降の処理でしばらく参照されないことが確かである場合、キャッシュ階層を介さずにデータを直接メインメモリーに書き戻す手法です。厳密に言えば、キャッシュを介さない書き込みは、インテル® Xeon® プロセッサーでのみ利用できます。*-qopt-streaming-stores=keyword* は、ストリーミング・ストアを有効/無効にします。

その他の考慮事項: スキャッター、ギャザー、圧縮構造

多くのアプリケーションは、明示的なベクトル・プログラミングの恩恵を受けます。そして、スカラーバージョンに対するパフォーマンスの向上率は、対象のプラットフォームで利用可能なベクトルレーンの数に比例します。しかし、一部のコーディング・パターンやスタイルでは、ベクトル化のパフォーマンスは大幅に制限されます。

ギャザーとスキャッターのコード

```
A[I] = B[Index[i]];          //Gather
       A[Index[i]] = b[i];   //Scatter
```

ギャザー/スキャッターによるベクトル化は最新のインテル® Xeon® プロセッサー・ベースのプラットフォームで利用できますが、ギャザー/スキャッターによるベクトル化のパフォーマンス向上は、多くの場合、ベクトルループ内でユニットストライド形式のロード/ストアを使用するよりもはるかに低くなります。ベクトルループの内部に十分な量の効率良くベクトル化されたその他の演算 (乗算、除算、算術関数の呼び出しなど) がない場合、パフォーマンスはシリアル・パフォーマンスよりも低くなる可能性があります。この問題の唯一の回避方法は、ギャザーとスキャッターを諦めて、ほかのアルゴリズムを使用することです。

構造の圧縮と展開

構造の圧縮と展開は、一般に問題を引き起こします。最新のインテル® Xeon® プロセッサーでは、インテル® コンパイラーは、単純な圧縮/展開形式のループを自動的にベクトル化できます。Fortran による圧縮の例を次に示します。

```
do I =1,N
    if (B(I)>0)
        x= x+1
        A(X) = B(I)
    endif
enddo
```

この例で、変数 **x** は特定の条件で更新されます。このような圧縮構造に !DIR$ SIMD (C/C++ では、#pragma simd) を使用するのは誤りですが、!DIR$ IVDEP (C/C++ では、*#pragma ivdep*) を使用するのはかまいません。

インテル® AVX-512 アーキテクチャーでベクトル化されたループのパフォーマンスを向上するには、*-qopt-assume-safe-padding* (Linux*)、/Qopt-assume-safe-padding (Windows*) コンパイラー・オプションを使用します。(「一般的なベクトル化のヒント」を参照してください。)

参考文献:

コンパイラーの診断メッセージ

- インテル® Fortran コンパイラーのベクトル化診断 (英語) - インテル® Fortran コンパイラーによって生成されるベクトル化レポートの診断メッセージ。インテル® Fortran コンパイラーでベクトル化レポートを取得するには、オプション *-qopt-report[=n] -qopt-report-phase=vec* (Linux* および macOS*) または */Qopt-report[:n] /Qopt-report-phase:vec* (Windows*) を使用します。

- インテル® C++ コンパイラーのベクトル化診断 (英語) - インテル® C++ コンパイラーによって生成されるベクトル化レポートの診断メッセージ。インテル® C++ コンパイラーでベクトル化レポートを取得するには、オプション *-qopt-report[=n] -qopt-report-phase=vec* (Linux* および macOS*) または */Qopt-report[:n] /Qopt-report-phase:vec* (Windows*) を使用します。

記事

- インテル® VTune™ Amplifier の使用 (英語) - インテル® VTune™ Amplifier は、多くのネイティブバイナリーを解析できます。
- ベクトル化および最適化レポート - *-qopt-report -qopt-report-phase=vec* (Linux* および macOS*) または */Qopt-report /Qopt-report-phase:vec* (Windows*) コンパイラー・オプションを使用して、アプリケーションのベクトル化できる領域、できない領域およびその理由を特定します。
- ループをベクトル化するための条件 - ループのベクトル化の条件、コード例、サンプル、アドバイス。
- メモリーレイアウト変換 - 構造体配列 (AoS) から配列構造体 (SoA) へのデータ構造の変更。
- ベクトル化の可能性を高めるデータ・アライメント - データ・アライメントとは、特定のバイト境界上のメモリーにデータ・オブジェクトを生成するようにコンパイラーに指示する手法です。さらに、データが 64 バイトでアライメントされていることが判明している場合、コンパイラーはさまざまな最適化を適用できます。
- 外部ループのベクトル化 - 要素関数とプラグマ/宣言子 SIMD の組み合わせを使用してベクトル化を内部レベルから外部レベルに移動します。
- インテル® マイクロアーキテクチャー向けベクトル化の重要性 (Fortran の例) (英語) - インテル® Fortran コンパイラーのベクトライザーを使用して、SIMD ハードウェアの効率的な利用とメニーコアでのスレッド化の利点により、優れたパフォーマンスを引き出します。
- 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) の parallel_for ブロックのベクトル化 (英語) - 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) の parallel_for ブロック内部にベクトル化しやすいコードを記述します。
- キャッシュ・ブロッキング手法 - キャッシュ・ブロッキングは、データアクセスの再配置により、データのサブセット (ブロック) をキャッシュに読み込み、このブロックに対して操作を行うことで、メモリーから繰り返しデータをフェッチしなくても済むようにします。
- メモリーレイアウト変換 - 実際のアプリケーション向けにデータ構造を配列構造体 (SoA) に変更します。
- 一般的なベクトル化のヒント - ベクトルループ内のユーザー定義の関数呼び出し、要素関数内のユニットストライド方式のアクセス、ベクトルループ内のメモリーの一義化。
- マルチスレッド・アプリケーション開発のためのガイド - スレッド同期やメモリー管理に利用できる、より特殊なチューニング関連の情報。

ドキュメント

インテル® Advisor
インテル® VTune™ Amplifier

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。