間接呼び出しと仮想関数の呼び出し：インテル® C/C++ コンパイラー 17.0 によるベクトル化

最新のインテル® C++ コンパイラーでサポートされたベクトル化方式の SIMD 対応関数の間接呼び出し

Hideki Saito インテル コーポレーション 主任エンジニア、Serge Preis 同主任エンジニア、
Sergey Kozhukhov 同シニア・ソフトウェア・エンジニア、Xinmin Tian 同主任エンジニア、
Clark Nelson 同主任エンジニア、Jennifer Yu 同シニア・ソフトウェア・エンジニア、
Sergey Maslov 同シニア・ソフトウェア・エンジニア、Udit Patidar 同プロダクト・マーケティング・エンジニア

Parallel Universe 22 号では、OpenMP* 4.0 仕様の SIMD サポート機能について取り上げました。OpenMP* 4.0 の主要な機能の 1 つは、ユーザー定義関数 (SIMD 対応関数) のベクトル化が可能なことと、ベクトル化されたコードからベクトル化された SIMD 対応関数の呼び出しが可能なことです。SIMD 対応関数は、通常の関数およびサブルーチンでベクトル命令を使用して多くの要素のデータ並列コンテキスト (ベクトルループの内部または別の SIMD 対応関数の内部) で同時に操作を行うことにより、**SIMD** ハードウェアのプログラミング環境を大幅に向上します。

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。

これまでプログラマーにより記述されていた、実行ファイルでスカラー関数呼び出しをベクトル化された関数に置き換えるメカニズムは、どのスカラー関数が呼び出されたか判断するコンパイル時の静的な情報に依存していました。その結果、多くの場合、ベクトル化された関数は直接呼び出しのみに制限されていました。

関数ポインターや仮想関数呼び出しによる間接関数呼び出しは、C++ のような現代のプログラミング言語の大きな特徴です。関数ポインターの主な利点は、実行時のポインターに基づいて実行する関数を選択する簡単な方法を提供して、アプリケーション・コードを単純化できることです。間接関数呼び出しの性質により、実際に呼び出される関数は実行時に決定されます。そのため、インテル® C++ コンパイラー 16.0 までは、これらの呼び出しがベクトル化の前に直接呼び出しに最適化されない限り、ベクトル化されたコンテキスト内の間接呼び出しはすべてシリアル化されていました。コンパイラーはこれらの間接呼び出しをベクトル化できず、SIMD 対応ハードウェアのベクトル機能を活用できないため、間接的に呼び出された SIMD 対応関数を実行すると大幅なボトルネックが発生していました。

間接関数呼び出しは C++ のような現代のプログラミング言語の大きな特徴です。

インテル® C++ コンパイラー 17.0 (インテル® Parallel Studio XE 2017 に含まれる) では、ベクトル化方式の SIMD 対応関数の間接呼び出しがサポートされました。また、SIMD 対応関数のポインターを宣言する新しい構文も追加されました。互換性規則のセットは、コンパイル時にスカラー関数の同一性を実際に解決するのではなく、異なるマッピングでスカラー関数をベクトル化します。SIMD 対応関数の間接呼び出しを宣言して使用する例を次に示します。

1 つの SIMD 対応関数ポインターに、ポインターにより呼び出されるターゲット関数で利用可能なすべてのバージョンに対応した、複数の vector 属性を関連付けることができます。間接呼び出しを検出すると、コンパイラーは実引数の種類と関数ポインターで宣言されているベクトルバージョンを比較して、最適なバージョンを選択します。ベクトル関数ポインター宣言と間接呼び出しを含む**図 1** の例について考えてみます。

```
// ポインター宣言

// ユニバーサルだが 3 つのループとも最も遅い定義と一致
__declspec(vector)

// 最初のループと一致
__declspec(vector(linear(in1), linear(ref(in2)), uniform(mul)))

// 2 つ目と 3 つ目のループと一致
__declspec(vector(linear(ref(in2))))

// 2 つ目のループと一致
__declspec(vector(linear(ref(in2)), linear(mul)))

// 3 つ目のループと一致
__declspec(vector(linear(val(in2:2))))

int (*func)(int* in1, int& in2, int mul);
int *a, *b, mul, *c;
int *ndx, nn;
...
// ループの例
   for (int i = 0; i < nn; i++) {
       /*
        * ループ内で 1 つ目の引数はリニアに変更される
        * 2 つ目の参照もリニアに変更される
        * 3 つ目の引数は不変
        */
       c[i] = func(a + i, *(b + i), mul);
   }

   for (int i = 0; i < nn; i++) {
       /*
        * 1 つ目の引数の値は予測不可
        * 2 つ目の参照はリニアに変更される
        * 3 つ目の引数はリニアに変更される
        */
       c[i] = func(&a[ndx[i]], b[i], i + 1);

   }

   #pragma simd private(k)
   for (int i = 0; i < nn; i++) {

       /*
        * ベクトル化により private 変数は配列に変換される：
        * k->k_vec[vector_length]
        */
       int k = i * 2;

       /*
        * 1 つ目の引数の値は予測不可
        * 2 つ目の参照と値はリニアであると見なせる
        * 3 つ目の引数の値は予測不可
        * (__declspec(vector(linear(val(in2:2)))) が
        * 一致する 2 つのバージョンの中から選択される）
        */
       c[i] = func(&a[ndx[i]], k, b[i]);
   }
```

1 ベクトル関数ポインター宣言と間接呼び出しを含むループ

SIMD 対応関数の直接または間接呼び出しでは、仮引数としてスカラー引数が指定されている部分に配列を与えます。並列コンテキスト内で SIMD 対応関数を間接的に呼び出すことも可能です。**図 2** は、コンパイラーに特殊なベクトル命令を生成させることで命令レベルの並列処理を実現する 2 つの構文を示しています。

```
__declspec(vector)
float (**vf_ptr)(float, float);

// 配列 a､b､c 全体に対して処理を行う
a[:] = vf_ptr[:] (b[:],c[:]);

/*
 * 配列表記構造で n (長さ) と s (ストライド) を
 * 指定する
 */
a[0:n:s] = vf_ptr[0:n:s] (b[0:n:s],c[0:n:s]);
```

2 並列コンテキスト内で SIMD 対応関数の間接呼び出しを行うサンプルコード

インテル® C++ コンパイラー 17.0 ではベクトル化方式の SIMD 対応関数の間接呼び出しがサポートされました。

インテル® C++ コンパイラー 17.0 (この記事の執筆時点では製品化前のバージョンを使用しています) とインテル® Xeon® プロセッサー E3-1240 v3 を使用したユニット・テスト・パフォーマンスの結果を**図 3** に示します。ユニットテストは、ループで **FUNC()** 関数を呼び出して合計リダクション操作を実行します。関数は直接呼び出し (シリアル化またはベクトル化) あるいは間接呼び出し (シリアル化またはベクトル化) です。ループの反復回数 NTIMES は、ウォールクロック時間で 1 秒よりも長く実行するテストを設定します。**図 4** は間接呼び出しのベクトル化のオーバーヘッドを示しています。ベースラインは、SIMD 対応関数機能を使用しないで関数を直接呼び出した場合です。最初に、“直接、ベクトル” の結果を見てみましょう。SIMD 対応関数機能を使用すると、4 つの倍精度計算が行われるため、パフォーマンスは約 4 倍になるはずです。TINY 計算テストモードでは、関数呼び出しのシリアル化のオーバーヘッドは SMALL/MEDIUM モードよりも大きいため、相対的なパフォーマンスは約 5 倍でした。

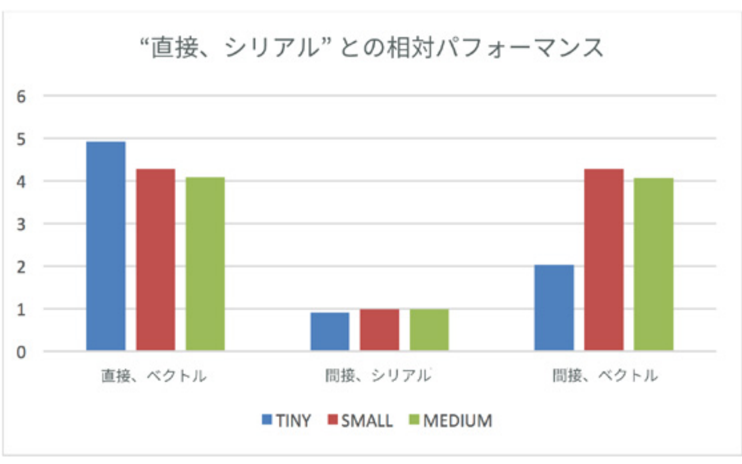
次に、“間接、シリアル” の結果を見ると、関数の間接呼び出しをシリアルに行った場合、(直接関数を呼び出した場合よりも) オーバーヘッドは小さくなっています。最後に、“間接、ベクトル” の結果を見ると、間接ベクトル関数呼び出しのオーバーヘッドは TINY モードでは大きくなっていますが、計算量を増やすと小さくなります。すべてのベクトル要素が同じ関数を呼び出しているため、これは最高のシナリオと言えます。

```
double sum=0;
#pragma omp simd reduction(+:sum)
for (i=0;i<NTIMES;i++){
    sum += FUNC(i);
}
#ifdef SIMD_ENABLED_FUNC
#pragma omp declare simd processor(core_4th_gen_avx)
#endif
__declspec(noinline)
double unit_test0(double x){
    return COMPUTE(x);
}
```

3 インテル® C++ コンパイラー 17.0 でベクトル化された仮想関数のユニット・テスト・パフォーマンス測定に使用したサンプルコード。
サンプルコードは合計リダクション操作を実行します。

テストモード	COMPUTE(x)	NTIMES
TINY	x+1	2,000,000,000
SMALL	exp(sin(x))	200,000,000
MEDIUM	log(exp(sin(x)))	200,000,000

ラベル	関数	SIMD_ENABLED_FUNC	SIMD_ENABLED_FUNC_PTR
直接、シリアル	unit_test0	X	N/A
直接、ベクトル	unit_test0	◯	N/A
間接、シリアル	funcptr0[0]	X	N/A
間接、ベクトル	funcptr0[0]	◯	◯

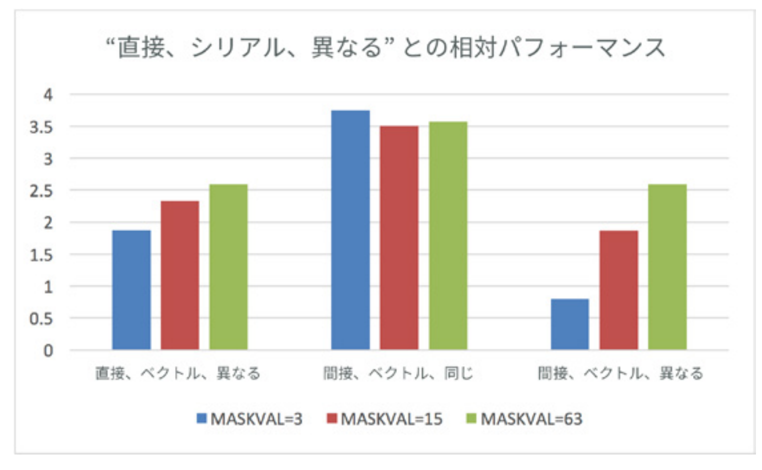


4 間接呼び出しによるベクトル化のオーバーヘッド

図 5 は分岐呼び出しのオーバーヘッドを示しています。この例で、**`FUNC()`** は **`unit_test0()`** または **`unit_test1()`** の直接呼び出しまたは間接呼び出しです (ループのインデックス値に依存します)。ここでは、SIMD 対応関数の間接呼び出しのオーバーヘッドに対する分岐のオーバーヘッドを強調するため、TINY 計算モードを使用しました。ベクトル長が 4 の場合、SIMD 対応関数呼び出しは MASKVAL=3 で 1 つのベクトル反復ごとに、MASKVAL=15 で 4 つのベクトル反復ごとに、MASKVAL=63 で 16 のベクトル反復ごとに、それぞれ分岐します。棒グラフの中央のセット "間接、ベクトル、同じ" は興味深い結果を示しています。このセットは 2 つの異なる関数ポインターを使用していますが、関数ポインターの実際の内容は同一であるため、最後の呼び出しは分岐せず、コンパイラーがその点を利用したことがパフォーマンスに反映されています。棒グラフの左のセット "直接、ベクトル、異なる" および右のセット "間接、ベクトル、異なる" と比べて、分岐呼び出しを間接的に呼び出すオーバーヘッドは実際の分岐により減少しています。

ラベル	関数	SIMD_ENABLED_FUNC	SIMD_ENABLED_FUNC_PTR
直接、シリアル、異なる	((i & MASKVAL) ? unit_test0 : unit_test1)	X	N/A
直接、ベクトル、異なる	((i & MASKVAL) ? unit_test0 : unit_test1)	◯	N/A
間接、ベクトル、同じ	((i & MASKVAL) ? funcptr0[0] : funcptr0[1])	◯	◯
間接、ベクトル、異なる	((i & MASKVAL) ? funcptr1[0] : funcptr1[1])	◯	◯

SIMD 対応仮想関数呼び出しの使用モデルは SIMD 対応メンバー関数呼び出しの使用モデルに似ていて、パフォーマンス特性は前述した間接呼び出しの例に似ています。コードの計算負荷の高い hotspot に間接呼び出しが含まれていて、ベクトル化が可能な場合、このインテル® C++ コンパイラー 17.0 の新機能を試してみると良いでしょう。



5 分岐呼び出しのオーバーヘッド

関連情報 (英語)

SIMD 対応関数

SIMD 対応関数ポインター

ベータ版 インテル® PARALLEL STUDIO XE 2017 を評価する

詳細 >

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。