ベクトル化を利用したデータ・アナリティクス向け LLVM コード生成の最適化

この記事は、インテル® デベロッパー・ゾーンに公開されている「Optimizing LLVM Code Generation for Data Analytics Using Vectorization」の日本語参考訳です。

ビッグデータ・ワークロードの効率を向上する

ビッグデータの爆発的な増加は、データ・アナリティクス・ワークロードの驚異的な増加に拍車をかけています。また、オンライン・トランザクション処理 (OLTP) と比較すると、アナリティクス・ワークロードのクエリーは、多くの場合、複雑で実行時間が長く CPU 依存です。コード生成は、分析データ処理を高速化する有効な手段です。カラム型や式演算子などの実行時に判明するクエリー固有の情報を、コンパイル時に利用可能であるかのように、パフォーマンスが重要な関数で使用できるようになり、効率良い実装が可能になります。

LLVM は、モジュール式の再利用可能なオープンソースのコンパイラー・ライブラリーと関連ツールのコレクションです。LLVM コア・ライブラリーは、最新のオプティマイザーと一般的な命令セット・アーキテクチャー向けのコード生成をサポートします。LLVM は、アナリティクス・エンジンが実行中のプロセス内でジャストインタイム (JIT) コンパイルを実行できるようにし、強力なオプティマイザーを活用します。LLVM ランタイムコード生成は、Apache Impala* や Apache Arrow Gandiva* など、多くのデータ・アナリティクス・エンジンでクエリーの実行パフォーマンスを向上するため採用されているテクノロジーです。

LLVM コード生成エンジンのパフォーマンスは、データ・アナリティクス・エンジン全体のパフォーマンスに重大な影響を与えます。現在の実装は、ベクトル化など、最近の CPU のハードウェア機能を最大限に活用していません。この記事では、実際にインテル® アーキテクチャー・ベースのベクトル化によりランタイムコード生成を最適化し、そこから判明したことについて述べます。

LLVM による問題と高速化

設計を理解する

最適化の手助けを必要とするテストコードから始めます。このテストコードは、インテルの顧客のデータ・アナリティクス・ソフトウェアから抽出されたもので、LLVM ベースのランタイムコード生成を使用したクエリー式評価モジュールで構成されています。図 1 は LLVM エンジンのブロック図です。

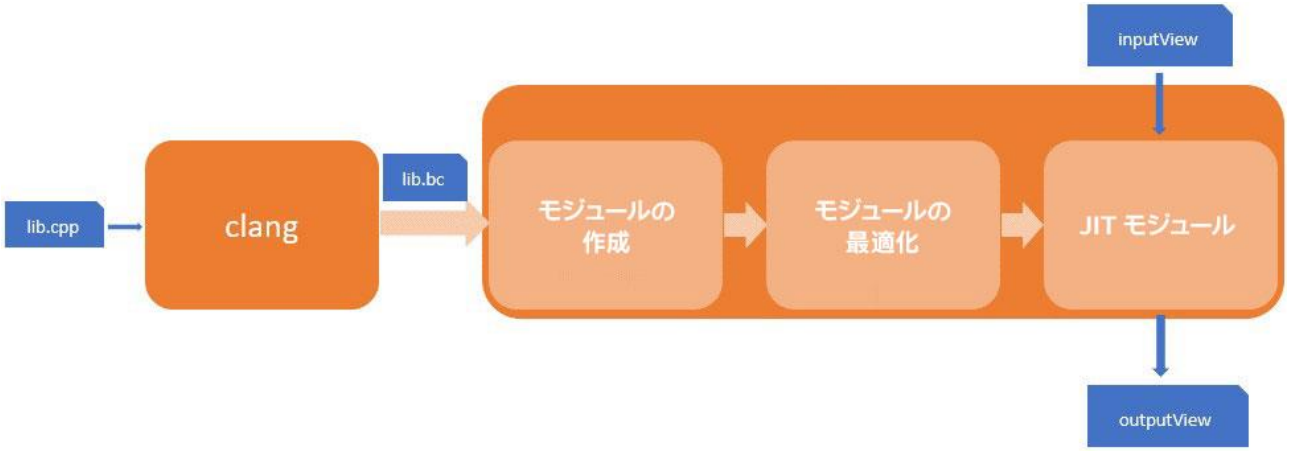

図 1. LLVM エンジンのブロック図

ワークフローには 3 つの段階があります。

1. モジュールの作成。エンジンは、呼び出し元からの SQL クエリーを使用して LLVM 中間表現 (IR) モジュールを作成します。
2. モジュールの最適化。LLVM のオプティマイザーを使用して作成したモジュールを最適化します。オプティマイザーは、パスと呼ばれ、JIT エンジン向けに最適化されたモジュールを出力します。
3. JIT モジュール。LLVM JIT エンジンは、**inputView** からデータを受け取り、最適化されたモジュールで命令を実行します。結果データは、**outputView** として出力されます。

詳しく説明すると、LLVM はコード生成のすべての部分において主に IR を使用します。モジュール作成段階では、エンジンは LLVM の **IRBuilder** を使用してプログラムで IR 命令を生成します。例えば、式評価の場合、**IRBuilder** を使用して、ループを形成する IR とループ内で 1 回式を計算する IR を含む、ループ関数プロトタイプのコードを生成します。

関数には多くの冗長な IR コード (例えば、算術演算やデータの取得/設定など) が含まれるため、プリコンパイルが使用されます。最初に、共通のコード (例えば、図 1 の **lib.cpp**) を C++ で実装して、Clang を使用して IR ファイルにクロスコンパイルします。この IR ファイルをプリコンパイル・ライブラリーと呼びます (例えば、図 1 の **lib.bc**)。コード生成時に、プリコンパイル・ライブラリーをロードして、**IRBuilder** によって生成された IR とリンクして、結合 IR モジュールを生成します。そして、必要なすべてのコードで構成された結合 IR モジュールを LLVM オプティマイザーに送ります。最適化後、IR モジュールは LLVM JIT エンジンで実行されます。

この仕組みは、従来のアプローチと比較して SQL 実行速度を大幅に向上しますが、最近のインテル® CPU アーキテクチャーの機能を最大限に活用しているとは言えません。具体的には、データ処理が行単位で行われているため、ベクトル化されません。

また、最近のインテル® CPU は異なる SIMD 命令(インテル® ストリーミング SIMD 拡張命令 [インテル® SSE] やインテル® アドバンスト・ベクトル・エクステンション [インテル® AVX/インテル® AVX-512]) で、異なるベクトル幅をサポートしています。インテル® CPU の能力を最大限に引き出すには、以下の一連のステップを考慮する必要があります。

プロファイルによってベクトル化の可能性を確認する

テストコードは、いくつかのテストケースで構成されています。典型的なテストケースでは、1000 万行のデータにわたって式 **l_extendedprice*(1-l_discount)*(1+l_tax)** を計算します。

データは、単票先形式でメモリーに格納されており、列ごとに次の 2 つのバッファーがあります。

1. **char* mData**
2. **int8_t* mNull**

mData バッファーはデータを保持し、**mNull** バッファーは **mData** が null かどうかを示します。

テストケースを実装するには、次の操作を行います。

- 最初に、いくつかの列を含むテーブルスキーマを作成し、1000 万行のテーブルデータを生成します。
- 次に、式を使用して関数プロトタイプを生成し、生成した関数を最適化します。
- 最後に、テーブルデータを入力として最適化された関数を評価 (実行) し、実行時間を測定します。

いくつかの手法でテストコードを解析したところ、追加の最適化候補が見つかりました。

最初に、生成された IR と最適化された関数のマシンコードをダンプして確認したところ、SIMD (インテル® AVX/インテル®AVX-512) 命令が使用されていませんでした。次に、インテル® VTune™ プロファイラーを使用してベースライン・プログラムをプロファイルしました。ホットスポット解析はコード生成関数を解析できないため有効な情報を得られませんでしたが、マイクロアーキテクチャー全般レポートはプログラムが CPU 依存であり (図 2)、さらなる最適化が可能であることを示していました。最後に、パフォーマンスの上限を解析しました。ループ内で直接式を計算する最小テストプログラムを C++ で作成し、**Clang -O3** でコンパイルして、実行時間を測定しました。結果は、ベースライン・テストコードの約 1/7 になりました。プログラムの逆アセンブリーを確認すると、期待どおり SIMD 命令でベクトル化されていました。

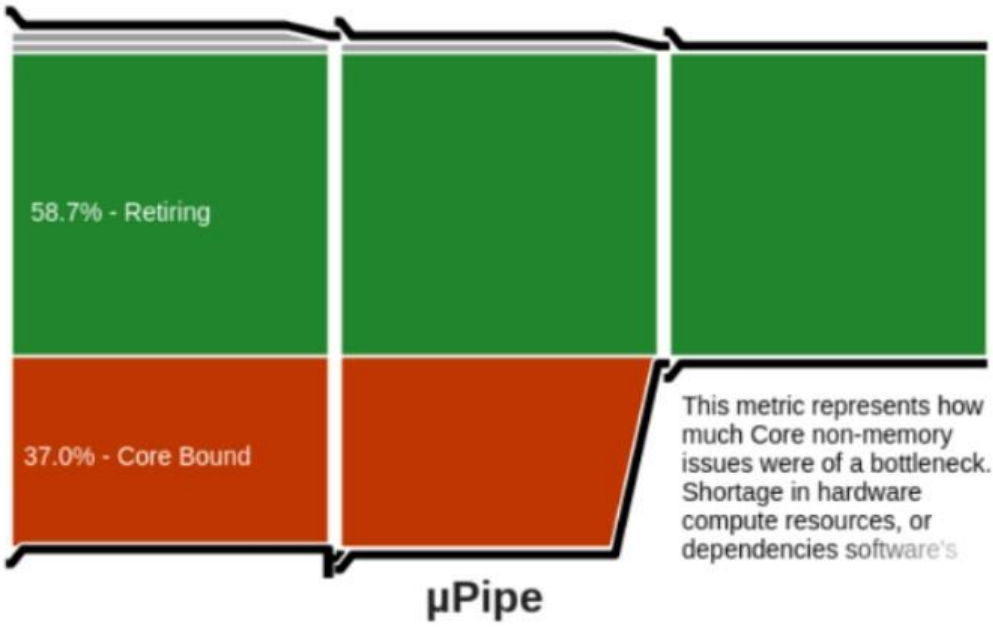

図 2. ベースライン・プログラムのプロファイル

解析によって得られる情報は一貫しており、ベクトル化は、テストコードを最適化する可能性の高いアプローチだと考えられます。

ベクトル化アプローチ

自動ベクトル化には、明示と暗示の 2 種類があります。

1. 明示的なベクトル化では、プログラマーは SIMD 命令を使用してプログラムを実装します。
2. 暗黙的なベクトル化では、コンパイラーが自動的に IR を SIMD に変換します。暗黙的なベクトル化は、コンパイラーによる自動ベクトル化としても知られています。

ここでは、ほかの式にも汎用的に利用できる LLVM のループ自動ベクトル化を使用することにしました。

コンパイラーによる自動ベクトル化アプローチでは、2 つの要因が LLVM のループ自動ベクトル化に影響を与えます。

- まず、LLVM codegen エンジンは、ベクトル化された IR を生成するため適切に設定されている必要があります (Clang コマンドライン・オプション **-Ox** を使用して最適化レベルを指定し、モジュール最適化段階で必要なループのベクトル化パスを使用して正しくベクトル化します)。
- 次に、ソースコードは、ループ・ベクトライザーがベクトル化できる方法で記述されている必要があります。関連する手法 (CPU キャッシュと並列処理を効率良く使用するデータレイアウト、LICM (ループ不変式) を妨げるポインター・エイリアシング問題への対処など) については、ビデオゲーム開発に端を発したデータ指向プログラミングで広く議論されています。

次のセクションでは、この考え方に沿って、サンプルコードを使用して手法の適用方法について詳しく説明します。

LLVM コード生成の自動ベクトル化

基本例から開始する

次のサンプルコードは、一般的な LLVM コード生成処理です。このサンプルコードを使用して、自動ベクトル化向けに **codegen** エンジンを適切に設定する方法を説明します。

plus.cpp と **samplejit.cpp** の 2 つのファイルを作成します。

1. **plus.cpp** は、入力に 10 を足して、合計を出力に格納する式をシミュレーションします。
2. **samplejit.cpp** は、プロトタイプ・モジュールをロードして最適化し、JIT 実行して LLVM コード生成エンジンをシミュレーションします。

以下は、**plus.cpp** のコードです。

```
#define LENARRAY 10240
typedef char ele_t;
ele_t a [LENARRAY], b[LENARRAY];

input plus(ele_t aa[], ele_t bb[], int len)
{
    for(int i=0; i<len; i++) {
        bb[i] = aa[i] + 10;
    }
        return 3148;
}
```

10,240 の入力行に対して SQL 式 **b=a+10** を論理的にシミュレーションして、3,148 を返します。この値は後で確認します。

plus.cpp は、Clang を使用して次のコマンドで IR ファイル (**plus.ll**) にコンパイルします。このステップは、モジュールの作成に相当します。

```
clang -c -emit-llvm -oO plus.cpp -o plus.bb
llvm-dis plus.bc -o plus.ll
```

-Ox コンパイルオプションを使用することで、ベクトル化された IR コードを簡単に生成できますが、ここでは LLVM JIT エンジンを使用してループをベクトル化するため、明示的に最適化レベル **-oO** (最適化なし) を指定します。**plus.ll** から、IR コードはベクトル化されていないことが分かります。

以下は、**samplejit.cpp** のコードです。

```
int main(int argc, char **argv) {
    ...
    // 入力 LLVM IR ファイルをモジュールに変換
    ...
    // MCJIT 実行エンジンをセットアップ
    ...
    // モジュールを最適化
    ...
    // モジュールを mc にコンパイル
    executionEngine->finalizeObject();

    // mc を JIT エンジンで実行
    int (*func_ptr) () = NULL;
    func_ptr = (int(*) ()) executionEngine->getFunctionAddress("_Z4plusPcS_i");
    int res = (*func_ptr) ();
    std::cout << "res: " << res << std::endl;

    return 0;
}
```

最初に、入力 IR ファイル (**plus.ll**) をモジュールとしてエンジンにロードします。次に、エンジンはモジュールを最適化します。その後、モジュールをマシンコードに JIT コンパイルします。関数 **plus()** を JIT エンジンで実行します (**_Z4plusPcS_i** は **plus()** のシンボルです)。変数 res の想定値は 3,148 であり、これは **plus()** の戻り値と一致します。

次に、サンプルコードをビルドして実行します。

```
echo "Building llvmtest ..."
$CPP -o llvmtest samplejit.cpp $CPPFLAGS $LDFLAGS
echo "Running llvmtest ..."
./llvmtest plus.ll
```

LLVM エンジンをベクトル化するには、いくつかの重要なステップがあります。

ループベクトル化の最適化パスを有効にする

最初に、次のループ最適化パスを **passmanager** に追加する必要があります。

```
std::unique_ptr<llvm::legacy::PassManager> pass_manager (new
llvm::legacy::PassManager());
    pass_manager->add(llvm::createFunctionInliningPass());
    pass_manager->add(llvm::createLoopRotatePass());
    pass_manager->add(llvm::createLICMPass());
    pass_manager->add(llvm::createInstructionCombiningPass());
    pass_manager->add(llvm::createPromoteMemoryToRegisterPass());
    pass_manager->add(llvm::createGVNPass());
    pass_manager->add(llvm::createCFGSimplificationPass());
    pass_manager->add(llvm::createLoopVectorizePass());
    pass_manager->add(llvm::createSLPVectorizerPass());
    pass_manager->add(llvm::createGlobalOptimizerPass());
```

パスの順序は重要です。**FunctionInliningPass** は、ループベクトル化パス向けの関数を生成するため、ベクトル化パスよりも前に呼び出されなければなりません。

エンジン向けに CPU 情報をセットアップする

ループベクトル化の幅を指定します。ループベクトル化幅は、SIMD 命令が一度に処理できるデータ量を示します。通常、CPU アーキテクチャーに合わせるべきです (例えば、インテル® SSE の場合は 128 ビット、インテル® AVX-512 の場合は 512 ビットにします)。以下に示すように、**EngineBuilder の setMCPU()** メソッドを使用することを推奨します。このメソッドは、複数世代のインテル® プロセッサーに自動的に適応できます。

```
std::string cpuName = llvm::sys::getHostCPUName() .str();
mEngineBuilder->setMCPU(cpuName);
```

Clang 最適化オプションを設定する

コンパイラーによる自動ベクトル化を利用するには、明示的にコンパイラー・オプションを指定する必要があります。Clang では、最小のコンパイルオプションは **-O2** です。GCC では **-O3** です。ここでは、lib のビルドステップで最適化フラグを有効にする Clang オプションを設定します。LLVM バージョン 5.0 以降では、プリコンパイル lib に optnone フラグが追加されないように **-O1/2/3** オプションを指定すべきです。

上記の設定をすべて適用したら、**samplejit.cpp** をビルドして実行します。結果は **res: 3148** となり、**plus()** 関数の JIT コンパイルと実行が正しく行われたことを示します。最適化後にマシンコードをダンプすると、インテル® AVX-512 命令が使用されていることが分かります。

データ指向プログラミング

上記の設定を実際のコードに適用しても、適切にベクトル化できない可能性があります。実際のプログラムはサンプルコードのように単純ではないからです。複雑な制御フロー、ベクトル化できない型、あるいはベクトル化できない呼び出しなどが原因で、ループをベクトル化できないことがあります。そのため、ベクトライザーが処理しやすい形式でプロトタイプ IR コードを記述する必要があります。CPU 機能を最大限に利用するマシンコードをコンパイラーが生成できるように、ハードウェア指向のコードを記述する方法は、多くの論文とプレゼンテーションで紹介されています。ここでは、ポインターの直接参照と非エイリアス手法を使用します。以下は、実際のユースケースと、ループをインテル® AVX/インテル® AVX-512 でベクトル化する高度な手法の適用方法を示すコード例です。

例: 実際のプログラミング構造のシミュレーション

以下は、実際のプログラミング構造のコード例です。

```
#define LENARRAY 10240
typedef unsigned char nullval_t;
class lvdt
{
public:
    lvdt(unsigned int capacity)
    {
        mNullVal = new nullval_t[capacity];
        for (int i=0; i<capacity; i++)
        {
            mNullVal[i] = rand() %2;
        }
    }
public:
    nullval_t* mNullVal;
};

int main()
{
    int loopcount = 10240;
    srand(time(0));
    lvdt* lvdt1 = new lvdt(LENARRAY);
    lvdt* lvdtout = new lvdt (LENARRAY);

    // ループ
    for (int i=0; i<loopcount; i++) {

        *(lvdtout->mNullVal+i) = *(lvdt1->mNullVal+i);
    }
    ...
    return 0;
}
```

次のコマンドでソースをビルドします (ループはベクトル化されません)。

```
clang ./loopvect.cpp -fno-use-cxa-atexit -emit-llvm -S -02 -mavx512f -c
-Rpass=loop-vectorize -Rpass-analysis=loop-vectorize
```

デバッグ出力には、「loop not vectorized: cannot identify array bounds. (ループはベクトル化されませんでした。配列境界を識別できません。)」と表示されます。これは、ループ内のポインターがエイリアスされていないことを、オプティマイザーが確認できないことが原因です。

ポインターエイリアスを理解するため、簡単なコードについて考えてみます。次のループをベクトル化しても安全であることを証明するには、配列 A が配列 B のエイリアスでないことを示す必要がありますが、コンパイラーは配列 A の境界を識別できません。

```
void test(int *A, int *B, int Length) {
  for (int i = 0; i < Length; i++)
    A[B[i]]++;
}

clang -03 -Rpass-analysis=loop-vectorize -S test.c -o /dev/null

test.c:3:5: remark:
 loop not vectorized: cannot identify array bounds
     for (int i = 0; i < Length; i++)
```

ポインターエイリアスの問題に対応する手法はいくつかあります。A と B のメモリー範囲がオーバーラップしないことが分かっている場合、次のように、ループの前に **#pragma** clang loop ディレクティブを明示的に指定して、コンパイラーに知らせることができます。

```
#pragma clang loop vectorize(enable)
```

同様の効果は、C/C++ コードで関数パラメーターに **restrict** 修飾子を追加することでも得られます。

```
void test (int* restrict A, int* restrict B, int Length)
```

1 つ問題があります。コード生成の例では、プロトタイプ・モジュールは IRBuilder を使用して IR で記述されているため、プラグマや **restrict** キーワードを適用できません。LLVM IR ではどのようにしたら良いでしょうか? LLVM の **noalias** 属性は、C の関数パラメーターの **restrict** 修飾子と同等です。

テストコードは、ループで間接ポインターを使用しているためやや複雑であり、これは LLVM のオプティマイザーがループをベクトル化するのを難しくします。この問題に対処するには、ループ外のバッファーへの直接ポインターを取得し、ループ内で使用します。以下にコード例を示します。

```
int main()
{
    int loopcount = 10240;
    srand(time(0));
    lvdt* lvdt1 = new lvdt (LENARRAY);
    lvdt* lvdtout = new lvdt (LENARRAY);
    nullval_t *p1 = lvdt1->mNullVal;
    nullval_t* pout = lvdtout->mNullVal;
    // ベクトル化可能な更新後のループ
    for (int i=0; i<loopcount; i++) {
        *(pout+i) = *(p1+i);
    }
    ...
    return 0;
}
```

パフォーマンス結果

これらの手法を顧客のベースラインのテストコードに適用することで、すべてのテストケースでベクトル化された式評価関数を生成できました。図 3 は、サーバー上でテストケースのパフォーマンス向上を測定した結果です。

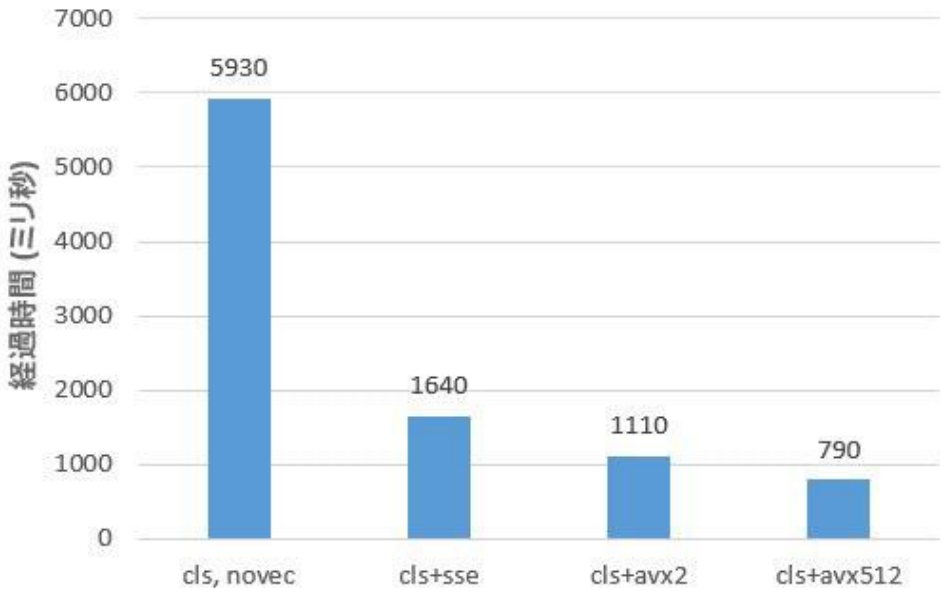

図 3. テストケースのパフォーマンス向上

サーバー構成: インテル® Xeon® Gold 6252 プロセッサー @ 2.10GHz、512GB メモリー、Fedora* 29 (Server Edition)、LLVM: Clang バージョン 7.0.1。法務上の注意書きについては、記事の最後を参照してください。

式 **l_extendedprice * (1 - l_discount) * (1 + l_tax)** の結果が示されています。ベクトル化により、インテル® AVX-512 命令で最大 7.5 倍のスピードアップを達成しています。データ型は、倍精度浮動小数点です。インテル® SSE、インテル® AVX2、インテル® AVX-512 それぞれの実行時間を速成するため、ベクトル幅を制御しました。ほかの単純な式 (例えば、単一演算子) を使用したテストケースでは、10 倍を超えるスピードアップを実現しました。

マイクロアーキテクチャー全般解析を再度実行したところ、コア依存の割合が減少し、メモリー依存が検出されました。インテル® AVX-512 命令は、オリジナルのスカラーバージョンの 8 倍のデータを処理するため、これは妥当であると言えます (図 4)。

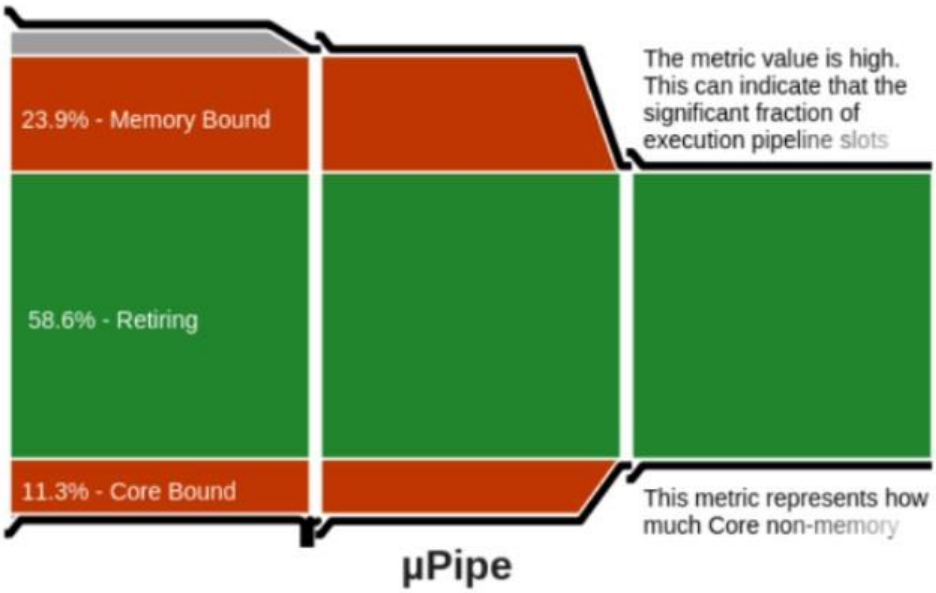

図 4. インテル® AVX-512 命令はオリジナルのスカラーバージョンの 8 倍のデータを処理

データ・アナリティクスを最適化する

この記事では、LLVM ランタイムコード生成の背景と、データ・アナリティクス・エンジンでの実装を紹介しました。また、実用的な例を使用して、codegen を解析して自動ベクトル化する方法も説明しました。ベクトル化により、インテルの顧客から提供されたテストコードでインテル® AVX-512 命令を使用することで、パフォーマンスを大幅に向上できました。

LLVM コード生成の最適化は、データ・アナリティクスのクエリーを高速化する大きな分野です。ここで紹介した手法はほんの一部です。ほかにも、前段階のコード生成や単票データレイアウトなどの手法があり、さらに検討する価値があります。

この記事で紹介した内容が、データ・アナリティクス分野の同様の最適化に役立つことを願っています。

関連情報

インテル® VTune™ プロファイラーは、パフォーマンス・ボトルネックを素早く見つけるのに役立ちます。高度なサンプリングとプロファイル手法により、コードを素早く解析して問題を切り分け、最新のプロセッサー上でパフォーマンスを最適化するための情報を提供します。無料でダウンロードすることも、インテル® Parallel Studio XE ツールスイートの一部として利用することもできます。

製品とパフォーマンス情報

[1] 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