

インターネット・ストリーミング SIMD 拡張命令を考慮したアプリケーション・チューニング

James Abel, Kumar Balasubramanian,
Mike Barger, Tom Craver, Mike Phlipot,
インテル社、マイクロプロセッサ・プロダクツ・グループ(MPG).

インデックス・ワード: SIMD、ストリーミング、MMX®命令、3D、ビデオ、画像処理

摘要

1997年の初頭に、インテルは、新しい命令セットの適用によるソフトウェア・アプリケーションの最適化に取り組むエンジニアリング・ラボを設立しました。このラボでは、商用ソフトウェアのメーカーとの共同作業により、新命令を使って各社のアプリケーションのパフォーマンス向上を図る努力が続けられてきました。2年後に、これらの新命令群は Pentium® III プロセッサの最も重要な新機能、つまりインターネット・ストリーミング SIMD 拡張命令として発表されました。現在では、このラボがコンサルティングを行ったアプリケーションは数多くの商用ソフトウェア・メーカーから出荷されており、インターネット・ストリーミング SIMD 拡張命令の利用によってパフォーマンスの飛躍的な向上を実現しています。この記事では、こうした活動を通じて明らかになった基本原理や概念を紹介します。

インターネット・ストリーミング SIMD 拡張命令の追加により、インテル・アーキテクチャにおける SIMD (Single Instruction/Multiple Data) 処理機能が拡張されました。以前はインテルの MMX®命令を通じて SIMD 整数演算が行われていましたが、これらの新命令では、8 個の新しい SIMD レジスタ・セットを通じて浮動小数点演算を実装します。さらに、インターネット・ストリーミング SIMD 拡張命令では、MMX レジスタ上で機能する新しい整数命令とキャッシュ制御命令が追加され、メモリ・アクセスの最適化が図られています。一般に、3D グラフィックス、デジタル画像処理、モーション・ビデオを使用するアプリケーションでは、新命令の利用によりパフォーマンスの向上が見込めます。

この種のアプリケーションにおいては、データ構成がパフォーマンスに大きく影響します。この記事では、3 種類のデータ並び(AoS、SoA、ハイブリッド)と、それぞれが SIMD 処理性能に及ぼす影響について説明します。また、プリフェッチ命令など、キャッシュ制御命令の影響についても考察を行います。

3D 変換や照明、バイリニア補間、ビデオ・ブロック・マッチング、動き補正といった処理におけるインターネット・ストリーミング SIMD 拡張命令の使用例を取り上げて解説します。これらの例に見られる基本原理は、他の多くのアルゴリズムやアプリケーションにも応用できるでしょう。

はじめに

プロセッサの発表と同時に多数の対応製品が市場投入されれば、ユーザの支持を高める効果があります。対応製品を開発するには、新しいプロセッサの持つ潜在能力を完全に理解することが出発点となります。また、開発の過程で、最大限のパフォーマンスを達成できるように、最適なアルゴリズムを選択することも必要です。Pentium® III プロセッサに関して言えば、インターネット・ストリーミング SIMD 拡張命令に照準を合わせて、こうした活動が 1997 年に開始されました。

この拡張命令群はさまざまなプログラムに幅広く適用できますが、特に 3D グラフィックス、デジタル画像処理、デジタル・モーション・ビデオを使用するアプリケーションで効果を発揮します。この種のアプリケーションで、新しい SIMD 命令を利用して最適化を図る方法を説明することがこの記事の目的です。

アプリケーション全体に対して最適化を図るよりも、特定のアルゴリズムまたはコンポーネントに絞り込んで最適化を行う方が、確実にパフォーマンス向上につながります。VTune™ パフォーマンス拡張環境(参考資料[1])などの分析ツールを使うと、アプリケーション内で最もプロセッサの負荷が高い箇所を簡単に特定できます。特定されたコンポーネントをさらに分析し、分岐の回数を最小限に抑えつつ大量のデータ・セットに対して同様の操作を実行するアルゴリズムを探します。

プロセッサに対するデータの入出力は、最適化に大きく影響します。そこで、この記事では、プリフェッチの効果も含め、さまざまなデータ編制手法を検証します。

また、各アルゴリズムについて、インターネット・ストリーミング SIMD 拡張命令を使用した場合と使用しない場合のコード例を紹介し、2つのバージョンを同じ Pentium III プロセッサ・プラットフォーム上で実行し、パフォーマンスの差を検証します。

データとインターネット・ストリーミング SIMD 拡張命令

このセクションでは、インターネット・ストリーミング SIMD 拡張命令を使ってパフォーマンスを最大限に引き出すために、考慮しなければならない問題を取り上げます。メモリ内でのデータ並びと、プロセッサによるそれらのデータの取り込み方法によって、パフォーマンスに大きな差が出ます。

SIMD 命令とメモリ・アクセス

通常、インターネット・ストリーミング SIMD 拡張命令の浮動小数点命令は、「垂直に」実行されます。つまり、これらの命令は、SIMD レジスタ内の対応する位置、またはそれに相当する位置のデータをメモリから直接ロードして処理されます。レジスタ内の4つの浮動小数点値すべてに対して同じ操作が行われるようにするため、一般に、SIMD レジスタの4つの位置にはそれぞれ、別のループで使用するアルゴリズムの同じ変数を格納するのが最良の方法です。たとえば、 $A[j] = B[j] + C[j]$ というアルゴリズムの場合、1つのレジスタに B を4つ配置し、別のレジスタに C を4つ配置すれば、1度の SIMD 加算演算で4つの A が得られます。

各 SIMD レジスタは小規模なデータ配列であり、これらのレジスタのセットは配列構造(以下、SoA と略す)であると考えられます。

```
struct { float A[4], B[4], C[4]; } SoA;
```

この SoA アプローチが適用できない場合もあります。計算式 $B[j] = B[j-1] + C[j]$ では、反復に依存関係があるため、別の方法を取る必要があります。

Pentium® III プロセッサは、大抵の場合、32 バイト単位のアドレスを通じて、メモリから一度に 32 バイトのデータをロードします。データは 32 バイトごとに L1 キャッシュと L2 キャッシュの「キャッシュ・ライン」内に格納されます。2つのキャッシュ・ラインにまたがってデータのロードや保存を行う命令を頻繁に使用すると、パフォーマンスの大幅な低下を招きます。

インターネット・ストリーミング SIMD 拡張命令の浮動小数点命令のほとんどではメモリ・アクセスの際に 16 バイト単位のアドレスを使用するため、このペナルティを回避できます。movups、movlps、movhps などの命令では、アライメントを指定せずにアクセスすることもできますが、ペナルティが生じる可能性があります。movlps 命令と

movhps 命令では、一度に 8 バイトずつ移動するため、ペナルティなしで 8 バイト境界のアドレスにアクセスできます(これに対し、movups では一度に 16 バイトのデータを移動します)。

プリフェッチ命令の使用

プリフェッチ命令は、必要なデータをすぐに使用できるよう準備しておく働きをするため、CPU の処理スピードに依存するアルゴリズムで使うと効果的です。プリフェッチ命令では、必要となるデータを事前にロードすることにより、ロードのレイテンシが軽減され、CPU でメモリ帯域幅を効率的に使用できるようになります。Pentium III プロセッサでは、キャッシュ・ラインへの書き込みと同時にデータをキャッシュにロードできるため、プリフェッチによりデータ格納時のレイテンシも減らせます。

また、利用できるメモリ帯域幅に依存するアルゴリズムにおいても、プリフェッチ命令は有効です。たとえば、prefetchnta 命令では、L1 キャッシュにだけデータをロードするため、L2 キャッシュへのロードによって生じるオーバヘッドを回避できます(通常のロード/ストアでは、データは L2 キャッシュにもロードされます)。

再度すぐに必要になる可能性が低いデータは、L1 キャッシュにだけロードすれば、L2 キャッシュにすでに置かれているデータを書き換えずに済みます。データをむやみに書き換えると、ペナルティはさらに大きくなります。キャッシュ・ラインの内容が変更されていると、内容をいったんメモリに書き出し、再度必要になったときにロードし直さなければなりません。

prefetcht2 命令は、L1 キャッシュに収まりきらないデータ・セットを L2 キャッシュにロードする場合などに使用できます。これを使うと、CPU スピードに依存するアルゴリズムを実行する間も、データに対してランダムにアクセスできるようになります。時間がたつにつれて、より多くのデータが L2 キャッシュに蓄積されます。

プリフェッチの効果を最大限に高めるには、ループをアンロールして(アルゴリズムのマルチパスをループの反復ごとに行うようにする)、反復のたびに、各変数が格納されている同じキャッシュ・ラインをプリフェッチして使用するようにします。

データ並びと SIMD アルゴリズムのパフォーマンス

SIMD 演算では SoA が最も自然な並びであるため、同様にメモリ内のデータもこの順序であることが望ましいと思われます。

```
struct
{
    float A[1000], B[1000], C[1000];
} SoA_data;
```

このアプローチがうまく機能するケースもあります。しかし、構造体メンバの数が多くなると、SoA を使うとメモリ・アクセスのパフォーマンスが低下する場合があります。PC のメモリ・システムでは、高速アクセス用に、限られた数のメモリ・ページ(一般に 4KB のブロック)だけを「オープン」にしておくことができます。メンバの数がこの数字を超えた場合、SIMD 演算で 4 つの値のセットを使用する際にメモリ領域のあちこちから参照しなければならず、メモリ・サブシステムではページの「再オープン」にかなりの時間を費やすことになります。

SIMD 命令を使用しないプログラミングでは、AoS (Array of Structures) データ並びが一般に使われています。

```
struct
{
    float A, B, C;
} AoS_data[1000];
```

AoS データ・セットによるシーケンシャルな処理では、メモリ内の隣接したデータを参照するため、SoA の場合に見られる「オープン・ページ」の制約を回避できます。しかし、このデータ並びは SIMD 演算に適した順序でないことが明らかです。

これを解決するには、一般に、AoS データを SIMD レジスタにロードしてからデータ再構成(「シャッフリング」)命令を使って SoA 形式に変換する方法が使われます。このプロセスは、データ並びの「転置」であると考えられます。図 1 を参照してください。

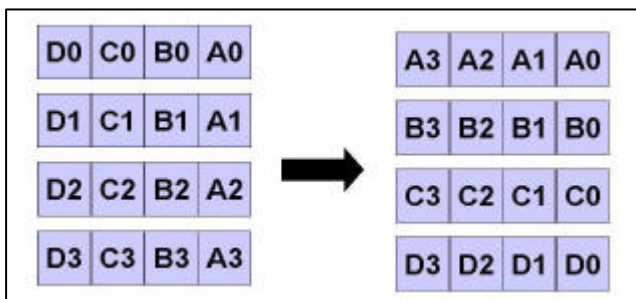


図 1: AoS から SoA への転置

転置を行うためにパフォーマンスにいくぶん影響が生じますが、大抵の場合このアプローチは効果的です。また、既存のデータが AoS 形式の場合、既存のプログラムのインターフェース仕様が AoS データだけに対応している場合、シーケンシャルではなくランダムにデータにアクセスする場合など、他の要因によって AoS データ並びを使わざるを得ない場合には、これが唯一の現実的な解決方法でしょう。

Pentium III プロセッサでは、メモリから一度に 32 バイトのデータをキャッシュにロードします。AoS 構造体メンバの中に現在の演算で必要ないものが含まれていても一括して

引き出されるため、メモリ・バスのオーバーヘッドが生じパフォーマンスが制限されます。

データのキャッシングでは、次の処理ステップで必要になるまでデータをキャッシュに留めておくことにより、こうしたオーバーヘッドを相殺できる場合があります。しかし、より大きなデータ・セットの場合、キャッシュに収まりきれないこともあり、次に使用する前にデータが書き換えられてしまうことがあります。一般に、同時に使用する頻度の高いものだけを AoS 構造体のメンバに加えるのがよいでしょう(このやり方は非 SIMD コードの場合にも有効です)。

AoS の使用を義務づける外部要因がないのであれば、AoS の特性であるデータの隣接性を確保しつつ SoA のロード順をサポートできるようなデータ並びを使用するのが望ましいと言えます。このような「ハイブリッド」データ並びの例を示します。

```
struct
{
    float A[8], B[8], C[8];
} Hybrid_data[125];
```

この並びでは、SoA の場合と同様に、プロセッサはどのメンバ配列からも 4 個の値を同時にロードできます(movaps などを使用します)。同じインデックスを持つ構造体メンバは、直に隣接していなくても、十分近い位置にあるため通常は同じメモリ・ページに格納されます。

ハイブリッド構造体が 32 バイトにアライメントされている場合、データも 32 バイト単位で移動します(4 バイトの float サブアレイにそれぞれ 8 つのエントリがあるため)。このことは、16 バイト・アライメントを必要とするインターネット・ストリーミング SIMD 拡張命令や、特定のメンバが 1 つでも含まれているキャッシュ・ライン全体をプリフェッチする場合などに便利です。大きなデータ・セットをシーケンシャルに処理する場合にも、確実に効果が得られます。

図 2 は、ドット積アルゴリズムのパフォーマンスにおける SIMD 命令とデータ並びの影響を示しています。

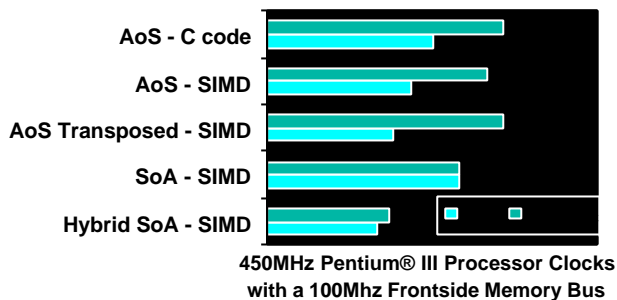


図 2: データ並びとパフォーマンス

ドット積はそれぞれ、3つの要素(xyz)からなる2組の大量のベクトル・データ間で計算したものです。これらはすべてC言語で書かれており、インターネット・ストリーミング SIMD 拡張命令を実装するにあたって、Intel®コンパイラに含まれる最適化された「組み込み関数」を使用しました。いずれの場合も、プリフェッチ命令を使用して、メモリ帯域の使用率を最適化しています。棒グラフの「xyz」は3つのメンバ・データだけを含む構造体でテストした結果を表し、「xyzabc」はドット積では使用しない3つの構造体メンバをデータ・セットに追加してテストした結果を表しています。

Hybrid SoA を使う方法では、総合的に最も高いパフォーマンス結果が出ています。AoS アルゴリズムでは、同じ構造内に余分なメンバが含まれるとパフォーマンスが大幅に低下しています(メモリの使用効率も低下します)。SoA および Hybrid SoA アルゴリズムでは、構造体に余分なメンバを加えてもほとんど影響はありません。インターネット・ストリーミング SIMD 拡張命令の使用に関しては、AoS の xyz アルゴリズムではドット積をシーケンシャルに計算した場合(AoS - SIMD)にいくぶんパフォーマンスの向上が見られ、処理前にデータを SoA 形式に転置するとさらにパフォーマンスが向上しました。SoA アルゴリズムは全面的にメモリの制約を受けたため、SIMD に適したデータ並びであるにもかかわらずパフォーマンスの向上にはつながっていません。

ブロック処理操作でのメモリ使用効率の最適化

一般に、ブロック処理アルゴリズムでは、大量のデータをシーケンシャルに読み込み、データに変更を加えてから別の大きなブロックに書き込みます。たとえば、画像データを RGB 形式から YUV 形式に変換するには、RGB コンポーネントをシーケンシャルに読み込み、それに対応する YUV コンポーネントの値を計算してから別の配列に書き込むという作業が必要になります。もっと単純な例を挙げると、ブロック・コピーなどがあります。

このようなアルゴリズムの多くでは大量のメモリを必要とするため、あらゆる手段を使ってデータ・フローを最適化

することが非常に重要です。これらのアルゴリズムに共通する特性は、一般に、いったん処理したデータを再利用しないという点です。このようなケースでは、キャッシュに結果を保存しても意味がなく、そればかりか他の必要なデータを書き換えてしまう危険性もあります。

ストリーミング・ストア命令(movntps、movntq)を使うと、キャッシュを経由せずに結果を目的のメモリ・バッファに直接書き込みできます。ただし、これらの命令は、ライト・コンバイニング(write combining)を有効に活用できるようなケースでなければ効果がありません。メモリや L2 キャッシュへのアクセスがあるたびに、ライト・コンバイニング・バッファが一定量に達する前に書き出される原因になり、メモリ・バスの使用効率が悪くなります。このパフォーマンス・ペナルティを回避するために、ストリーミング・ストア命令で結果を書き出す間、データに対する読み込みを L1 キャッシュだけに限定する必要があります。

ブロック処理アルゴリズムでこれを行うには、prefetchnta を使ってサブブロック・データ(通常 4KB 程度)を L1 キャッシュに読み出すためのループを追加します。通常処理のループでは、この命令に従って L1 キャッシュに置かれたサブブロック・データを読み込み、ストリーミング・ストア命令を使って結果を書き出します。両方のサブブロック・ループの外側にあるループでは、すべてのサブブロックを合わせてデータ・セットを構成します。この方法を使うにあたって、問題が1つ生じます。プリフェッチ命令は、prefetch で指定された仮想メモリ・ページが、Pentium III プロセッサの TLB (Translation Lookaside Buffer)によって物理メモリ・ページにマップされる場合でなければ機能しません。一般に、TLB の更新は、最初にそのページにアクセスしたときに行われます。TLB のエントリの数は限られているため(64 など)、最近使われていないページ・マッピングは TLB から書き出されてしまっている可能性があり、プリフェッチが機能しないこととなります。

プリフェッチ命令を確実に機能させるには、ソース・データを 4KB メモリ・ページごとに読み込み、その直後に prefetch ループを開始する必要があります。TLB の初期化に多少の時間がかかりますが、prefetch ループの直前に読み込みが行われれば、プリフェッチの多くは問題なく迅速に実行されます。このことは、さまざまな方法で調整できます。たとえば、prefetch ループの開始点よりも 4KB 先のアドレスを読み込むようにすれば、有効なデータを確実に読み込むことができます。

データをキャッシュに収まるサイズのサブブロックに分ける方法は、キャッシュに収まらないデータを何度も転送するような場合にも効果的です。たとえば、一連の処理ステップで、前のステップの結果を順に次のステップで使用するような場合に有効です。

この操作を別々に処理する場合、中間結果をそのつどメモリに書き込み、次のステップでメモリからロードし直さなければなりません。代わりに、キャッシュに収まる複数の小さなブロックに分けて転送することにより、メモリ・バスのトラフィックを最小限に抑えることができます。

3D アプリケーションのチューニング

一般に、3D ジオメトリ・エンジンには、変換、照明やシェーディング、クリッピング、カリング、透視補正モジュールなどさまざまなコンポーネントがあります(参考資料 [2])。どのコンポーネントを最適化すべきかを見極めるのは非常に困難です。この記事の冒頭でも紹介したように、変換や照明に使われる関数は、インターネット・ストリーミング SIMD 拡張命令の使用によりパフォーマンスの向上が期待できます。

変換や照明の関数は膨大な計算を必要とし、隣接した大量のデータに対して同じ操作を行う内部ループで構成されるため、SIMD の効果が大きいと言えます。プリフェッチ命令を使うと、ループで使用するデータを事前にロードしておくことができます。新しい近似命令では、照明ループでのレイテンシの長い平方根や除算の操作を減らすことができ、変換ループで透視補正を行う際にも役立ちます。最後に、照明ループ内で値の範囲を調整する操作でも、新しい min/max 命令を使うことにより、反復の際に発生する予測不能な 2 種類の分岐を排除できます。

この後に、それぞれの最適化手法の詳細について説明します。どのケースについても、データ並びとアライメントについては前のセクションを参照してください。

3D 変換

3D 変換を実行するには、 4×4 の変換マトリックスと 4 つの要素からなるベクトル値を掛け合わせます。ベクトル値は頂点要素である X 、 Y 、 Z と定数の 1 で構成されます。一方、変換マトリックス自体はシーンの個々のオブジェクトごとに計算されます。この操作では、中間値 X' 、 Y' 、 Z' 、 W を算出します。場合によっては、 W の値をそのまま使って中間ベクトルを正規化し(透視変換)、最終値の X'' 、 Y'' 、 Z'' を生成することもあります。4 つめの要素の最終結果は常に 1.0 であるため、 W による除算は無視してもかまいません。

3D 変換の最適化

変換処理では、特定のメッシュのすべての頂点に対して同じ変換マトリックスを適用し、また大量のデータを処理するため、SIMD プログラミングによる効果が高いと言えます。インターネット・ストリーミング SIMD 拡張命令を利用して最適化を図るには、SIMD レジスタの利点をフルに活用する必要があります。1 個のレジスタに X_0 、 X_1 、 X_2 、 X_3 という 4 つの X 値をロードし、別のレジスタにはそれぞれ 4 つの Y 値、4 つの Z 値をロードします。

次に、マトリックス要素の最初のデータ・セットを 4 つめのレジスタにロードします。この操作には、2 種類の方法があります。1 つめの方法では、マトリックス要素をそれぞれ単精度浮動小数点値としてメモリに格納し、movss を使ってレジスタの一番下の位置に読み込み、shufps 命令を通じて 4 個に複製します。もう一方の方法では、要素を 4 個の同じ浮動小数点値の配列として格納し、32 バイト境界に揃えた状態で、movaps 命令を通じて一度に 4 個ずつ読み込みます。2 番めの方法の方が効果的です。マトリックス全体についてこの方法で格納すると構造体のサイズは 64 バイトから 256 バイトに急増しますが、パフォーマンスの向上を考えると些細な問題です。

同様のやり方で、マトリックス要素 m_{01} 、 m_{02} 、 m_{03} をメモリからロードします。レジスタ内のすべてのデータを SIMD 形式にすることにより、変換処理は、3 つの乗算命令とそれに続く 3 つの加算命令で結果が出るというシンプルなものになります(図 3 を参照)。 X' の値を求めるための頂点データを Y の値の計算にも使用することがわかっているため、マトリックス・データが格納されているレジスタの内容を上書きするような命令を使います。一連の計算が完了すると、中間値 X_0' 、 X_1' 、 X_2' 、 X_3' は 32 バイト単位の出力バッファに書き込まれます。

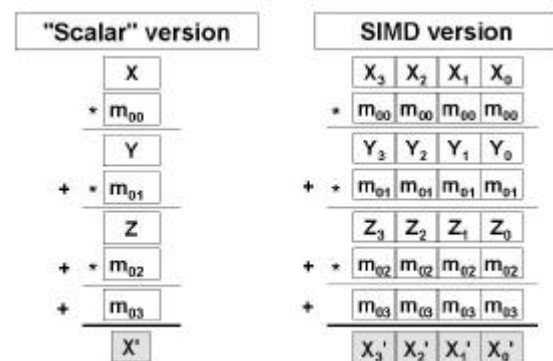


図 3: 通常の変換では 1 個ずつ値を計算するのに対し、SIMD の変換では一度に 4 個の値を算出します。

次に、マトリックス要素 m_{10} 、 m_{11} 、 m_{12} 、 m_{13} をロードして、最初の 4 つの Y の値についてこのプロセスを繰り返します。さらに、 Z' および W' について、それぞれ対応するマトリックス要素をロードして計算を行います。最初の変換ステップの最終結果として、16 個の中間値が算出されます。この時点で透視変換を行うのであれば、rcpps 命令を使うと非常に効果的です(詳細については 3D 照明の最適化のセクションを参照)。図 4 は、Pentium III プロセッサ固有のコードを前述の方法で最適化した場合と、同じアルゴリズムを標準 C で実装した場合のパフォーマンス結果を比較したものです。

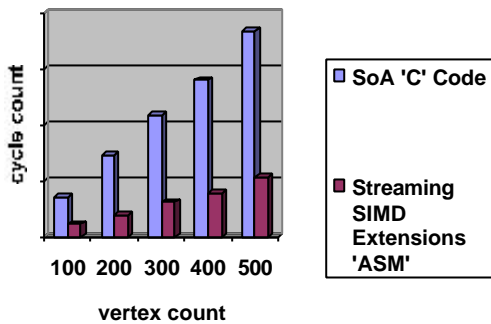


図 4: 変換のサイクル・タイム -- Pentium® III プロセッサに最適化されたアセンブリ・コードと標準的な C コードの比較

3D 照明

3D グラフィックス・アプリケーションで最も広く使われている光源として、ポイント・ライトが挙げられます。特定の頂点にポイント・ライトを適用するには、最初にその頂点から光源へのベクトル値を計算します。ベクトルの長さが算出されると、これを使って頂点から光源へのベクトル値を正規化します。正規化されたベクトル値から、現在の頂点に関する拡散要素を計算します。最後に、その頂点の全体的な色の値を計算し、値が[0.0, 1.0]の範囲内であることを確認します。この範囲を超える値は、上限または下限どちらかの値に「強制補正」する必要があります。頂点の色の値が算出され調整を済ませたら、レンダリング時に使用できるようメモリに格納されます。

3D 照明の最適化

光源からの距離を計算するには、標準的な二乗の和の平方根を使います。ここでも SIMD レジスタをフルに活用し、各レジスタに 4 つの頂点値をロードして同じ光源から 4 つの頂点までの距離を同時に計算します。次に、光源のベクトル値を正規化するために、算出済みの距離でベクトル値を除算します。簡単なお見えですが、これらのステップには平方根と除算という時間のかかる浮動小数点演算が必要とされ、それぞれの演算を完了するのに 36 クロック・サイクルほど要します。その上、これらの命令は「パイプライン化されない」ため、実行中の命令がリタイアするまでは、他の命令を実行することができません。

この問題を解決するために、インターネット・ストリーミング SIMD 拡張命令には、`rsqrtps` と `rcpps` という逆近似命令が用意されており、これらを利用すれば同等の量の演算を短時間でこなせるようになります。これらの命令では、ハードウェア・テーブル・ルックアップの利用によりレイテンシを 2 クロック・サイクル減らすことができ、また、命令のパイプライン化のおかげで、実行中でも他の操作を開始できます。ただし、トレードオフとして、真の単精度命令では仮数の精度が 23 ビットまで保証されるのに対し、これらの命令では 11 ビットまでしか保証されませ

ん。ほとんどの 3D アプリケーションでは 11 ビットで十分ですが、アプリケーションによってはこれ以上の精度が必要とされる(または推奨される)場合があります。

Newton-Raphson 法は、近似計算で失われた精度を回復するための数値演算アルゴリズムです。シングルパスの Newton-Raphson で、精度は倍の 22 ビットまで回復し、次のパスで 23 ビットまで回復します。近似計算の後にこの操作を行うことにより、処理を大幅に高速化できます。

$$\text{rcp}'(a) = 2 * \text{rcp}(a) - a * \text{rcp}(a)^2$$

$$\text{rsqrt}'(a) = (0.5) * \text{rsqrt}(a) * (3 - a * \text{rsqrt}(a)^2)$$

各距離の二乗の逆数と計算済みの光源ベクトル値を掛け合わせるにより、正規化された 4 つの方向ベクトル値が生成されます。これらの値は、その後頂点の色の diffuse コンポーネントを計算する際に使用します。

拡散要素の計算では 2 つのベクトル値の内積を求めます。計算結果が負の数値になる場合もあります。一般に、グラフィックス・アプリケーションでは、負の数値を避けるために「ゼロより小さい」(less-than-zero)チェックを行い、結果が「true」であれば値を「0」に設定します。繰り返しのたびにこの種の予測不能な条件分岐が行われるとパフォーマンスのボトルネックになります。この分岐を排除するには、4 個の diffuse の結果を格納するレジスタと 4 つの 0.0 値をもつレジスタに対して `maxps` 命令を実行します。`max` 命令を実行すると、負の値は 0 に変更され、正の値であれば変更されずそのまま残ります。

頂点の最終的な色を計算する際にも、同様の最適化手法が使えます。ここでは、計算結果の値が 1.0 を超える傾向があります。`minps` 命令を実行すると、しきい値を超える値は 1.0 に変更され、条件分岐を使わず済みます。最適化された照明アルゴリズムと、標準 C コードによる照明関数の比較を図 5 に示します。

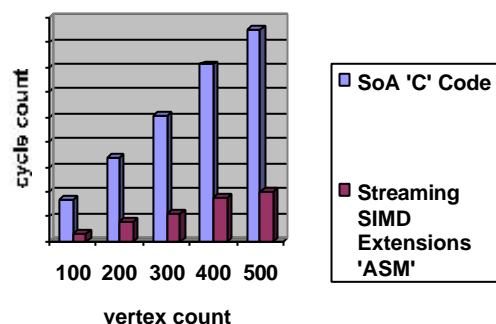


図 5: 照明のサイクル・タイム -- Pentium® III プロセッサに最適化されたアセンブリ・コードと標準 C コードの比較

デジタル画像処理

一般に、デジタル画像処理アプリケーションは、小規模な数値演算を大量のピクセル・データに対して行うアルゴリズムで構成されています。また、各ピクセルは Red、Green、Blue、Alpha という 4 つの値で構成されています。このため、デジタル画像処理アプリケーションでは、MMX®テクノロジーの利用によりパフォーマンスが飛躍的に向上します。しかし、最新の画像処理アプリケーションでは高精細のビデオ/グラフィックス機能が豊富に取り入れられる傾向にあり、パフォーマンスに対する要求はますます高まっています。Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令を利用すれば、こうしたパフォーマンス要求にも余裕で応えられます。この後のセクションでは、デジタル画像処理アルゴリズムにおいて、MMX テクノロジーで達成されたパフォーマンスの向上をさらに強化するこれらの新機能について説明します。

整数 SIMD 拡張命令

SIMD 画像処理アルゴリズムを実装するにあたっては、MMX®レジスタ内のデータを並べ替える必要性が頻繁に生じます。整数 SIMD 拡張命令には、そのような頻繁に発生する処理のパフォーマンスを強化するためのシャッフル命令 (pshufw) が含まれています。たとえば、アルファ・サチュレーションでは、効率的な SIMD 命令の実装により、R、G、B すべての要素について対応するアルファ値との比較を並行処理できるようになります。このためには、図 6 に示すように、アルファ値自体を別の MMX レジスタ内で複製する必要があります。

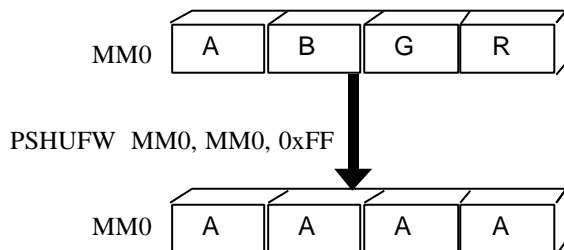


図 6: アルファ値のブロードキャスト

MMX テクノロジーではこの操作に 3 つの命令を必要としますが、新命令ならば 1 つだけで済みます。

特定の画像処理アルゴリズムを SIMD にマッピングする過程でデータに依存する分岐があると、パフォーマンスの低下を招くことがよくあります。たとえば、RGBA の中間値を計算した後で、R、G、B、A のいずれかの値が特定のしきい値に満たない場合、さらに別の計算が必要になることがあります。通常の MMX 命令では、条件チェックを行うと、MMX レジスタ内で複数のマスク・パターンが生じます。しかし、必要なビットをこうしたマスク・パターンからアドレッシング用のレジスタに抽出すると、大抵の場合きわめて効率が悪くなります。場合によっては、こ

のことが原因で、SIMD の実装によって得られるパフォーマンスの向上が相殺されてしまうこともあります。新命令 `pmovmskb` を使うと、このような問題を解決できます。この命令では、必要なビットを MMX レジスタのマスク・パターンから抽出し、アドレッシングに使うレジスタに配置します。

ヒストグラム関連のアルゴリズムに見られるようなテーブル・ルックアップ操作は、デジタル画像処理のパフォーマンスに大きく影響します。これらのアルゴリズムでは、算出された R、G、B の値を、それぞれのカラー・ルックアップ・テーブルでのインデックスとして使用します。MMX テクノロジーでは、算出した RGBA の値が MMX レジスタに格納され直接的にはアドレッシングに使用できなくなるため、この操作の実装は困難でした。それぞれの値をアドレッシング用の適切なレジスタに抽出し、テーブルからそれに対応する内容を取り出し、再度 MMX レジスタに書き込むという操作は効率が悪く、パフォーマンスの低下を招きます。整数 SIMD 拡張命令には、こうしたアルゴリズムのパフォーマンスの強化につながる 1 組の命令 (`pinsrw/pextrw`) が含まれています。

新しい整数 SIMD 拡張命令には、これらの命令のほかにも、頻繁に使用する画像処理アルゴリズムのパフォーマンスを強化する命令がいくつか用意されています。たとえば、SIMD *unsigned multiply* 命令を使うと、MMX テクノロジーでは実装が困難であった特定のフィルタ処理も容易に実装できます。また、アルファ・サチュレーションにおける値の範囲チェックに役立つ `minimum/maximum` 命令や、あらゆる条件チェックに役立つさまざまな比較演算子が幅広く用意されています。

SIMD 浮動小数点命令

現在の画像処理操作には、主に固定小数点式の整数計算が使用されています。しかし、最新の画像処理アプリケーションでは、多彩なグラフィックス機能を駆使し高精細画質が要求されるケースがますます増えてきています。このようなアルゴリズムでは、インターネット・ストリーミング SIMD 拡張命令の SIMD 浮動小数点機能の利用により、パフォーマンスの飛躍的な向上が見込まれます。すでに浮動小数点命令をもとに実装されているアルゴリズムの場合でも、浮動小数点演算のパフォーマンスをさらに強化することにより、大抵のユーザ要求にリアルタイムで応えられるようになります。また、計算の過程においても浮動小数点表示の精度が高められるため(固定小数点表示では 16 ビットまで)、優れた画質を実現します。さらに、浮動小数点形式でアルゴリズムを実装することにより、浮動小数点演算の処理を減らすことができます。このため、コードの作成、デバッグ、保守といった作業が容易になり、生産性が格段に向上します。画像処理アルゴリズムでは、基本となるデータ・オブジェクト (RGBA ピクセル値) は整数データ・タイプです。しかし、これまで以上に精度の高さとプログラミングの容易さが要求される現在においては、浮動

小数点を使ってデータの変換を実装するケースが増えてきています。こうしたアルゴリズムでは、SIMD 浮動小数点機能の利用によりパフォーマンスが飛躍的に向上します。この後、バイリニア補間の例を挙げて、これらの SIMD 浮動小数点命令の使い方と使用に伴うパフォーマンス上のトレードオフについて説明していきます。

バイリニア補間の例

表示画像の各ピクセルの RGBA 値は、ソース画像での隣接する 4 つのピクセルの RGBA 値をもとに、バイリニア補間を使って計算されます(図 7 参照)。

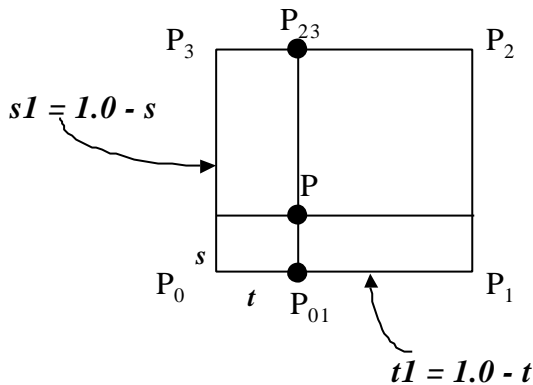


図 7: バイリニア補間

ピクセル P のコンポーネント R は、次のように計算します。

$$R_{01} = tI * R_0 + t * R_1$$

$$R_{23} = t * R_2 + tI * R_3$$

$$R = sI * R_{01} + s * R_{23}$$

これらの計算式を見るとわかるように、バイリニア補間のステップは、3 組のリニア補間の計算で構成されています。リニア補間では、R、G、B、A それぞれの値について、乗算を 2 回、加算を 1 回行います。もちろん、SIMD を利用した場合は、4 つの RGBA コンポーネントを同時に計算できます。SIMD 浮動小数点機能の利用によってパフォーマンスの向上が見込まれるため、これらの計算を浮動小数点で行う場合を考えてみましょう。この場合、最初に、RGBA のピクセル値を通常のバイト表示から浮動小数点表示に変換する必要があります。ここで必要になるステップを図 8 に示します。ソース画像の 4 つのピクセルそれぞれについて、この変換処理を行います。このステップでは、MMX レジスタと新しい Pentium® III プロセッサのレジスタを使うため、MMX®テクノロジーと SIMD 浮動小数点命令の両方を使用することに注目してください。浮動小数点 SIMD ユニットと整数 SIMD ユニットで並行して処理を行うため、4 つのピクセルをまとめて変換処理でき、ハードウェアの利用効率を高められます。

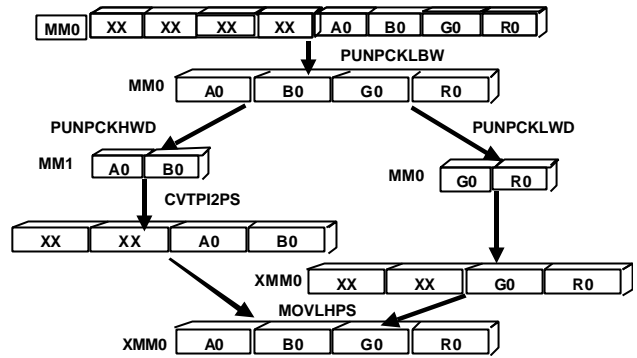


図 8: バック化によるバイト値から浮動小数点値への変換

このようにデータ・タイプを変換した後では、リニア補間での実際の乗算・加算ステップが比較的容易になります(図 9 参照)。算出されたピクセルの RGBA 値は浮動小数点形式であるため、整数タイプに戻す必要があります。ここでは、図 8 と同様のステップが必要になります。

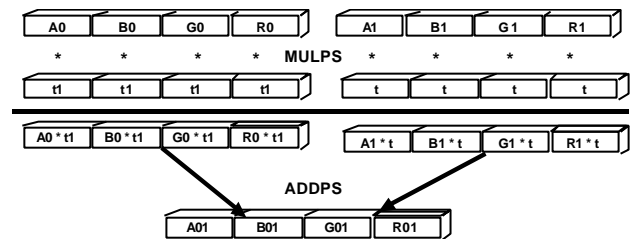


図 9: リニア補間

この種のアルゴリズムを分析すると、本質的に、およそ 9 個の基本命令が必要であることがわかります(3 つのリニア補間についてそれぞれ MULS が 2 つと ADD が 1 つ)。SIMD 浮動小数点を使ってこれを実装する場合、さらに 29 個の命令が追加されることとなります(4 つのソース・ピクセルをそれぞれバイト値から浮動小数点値に変換する 6 種類の命令と、計算結果の表示ピクセルを浮動小数点値からバイト値に変換する 5 個の命令)。しかし、多くのアプリケーションでは、ピクセルのバイリニア補間以外にも、照明効果などいくつかの浮動小数点演算を行います。そのような場合、バイト値と浮動小数点値の変換によるオーバーヘッドが、こうした他の浮動小数点演算すべてに影響を及ぼします。このため、SIMD 浮動小数点命令を使ってパフォーマンスを強化することには大きな意味があります。

キャッシュ制御命令

一般に、画像処理では大量のデータ・セットを扱うため、キャッシュの利用効率を高めればパフォーマンスの大幅な向上につながります。インターネット・ストリーミング

SIMD 拡張命令には、ハードウェア・リソースをより効率的に利用しキャッシュ汚染を最小限に抑えるためのキャッシュ制御命令がいくつかあります。データの使用頻度に応じて異なるプリフェッチ命令を使って、事前にデータをメモリからキャッシュにロードします。たとえば、タイル・ベースのイメージング・アーキテクチャでは、プロセッサの実行ユニットが特定のタイルのデータを処理する間、メモリ・サブシステムは次に必要となるタイルのデータをプリフェッチすることができます。同様に、最終的に表示されるピクセル値が算出されると、ストリーミング・ストア命令を使って、それらの値をキャッシュにロードせずに直接メモリに格納することもできます。このため、すでにキャッシュに格納されているデータが別の計算に必要な場合に、キャッシュの内容が更新される可能性を最小限にとどめられます。

これらのキャッシュ制御命令を最大限有効に活用するには、プリフェッチすべきデータ・セットの識別、プリフェッチしたデータを置くキャッシュ・レベル、プリフェッチを行うタイミングなどについて注意を払う必要があります。

ビデオ・コーデック

MPEG コードやデジタル・ビデオ(DV)コードなどのビデオ・コーデックでは、インターネット・ストリーミング SIMD 拡張命令の利用によりパフォーマンスの向上が見込めます。パフォーマンス向上が見込まれる操作の例を表 1 に示します。

	PSADBW	PAVG	Prefetch, Streaming Stores
Encode	Motion Estimation	Motion Estimation, Motion Compensation	Color Conversion, Motion Compensation
Decode		Motion Compensation	Color Conversion, Motion Compensation

表 1: ビデオ・コーデックでのインターネット・ストリーミング SIMD 拡張命令の使用

動き推定

動き推定では基本的にブロック・マッチングが使われます。ブロック・マッチング関数のアウトプットである Sum of Absolute Difference (Sum of Absolute Distortion と呼ばれる)の値を計算するには、次の公式が使われます。

$$SAD = \sum_{i=0}^{15} \sum_{j=0}^{15} |Block_{ref}[i][j] - Block_{pred}[i][j]|$$

図 10 は、動き推定を計算するしくみを示しています。

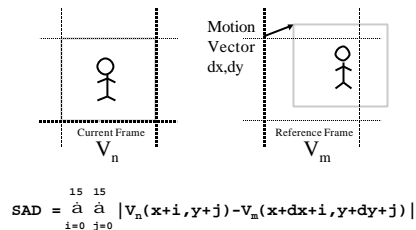


図 10: ブロック・マッチング

dx および dy は、評価の対象となる動きベクトルです。異なる dx と dy の組み合わせを使って複数のブロック・マッチング操作を行うことにより、動き推定を行います。SAD の値が最小になるものが最も有効な動きベクトルです。

インターネット・ストリーミング SIMD 拡張命令には、ブロック・マッチングの高速化に役立つ新しい命令、psadbw が含まれています。この命令で行われるオペレーションを図 11 に示します。

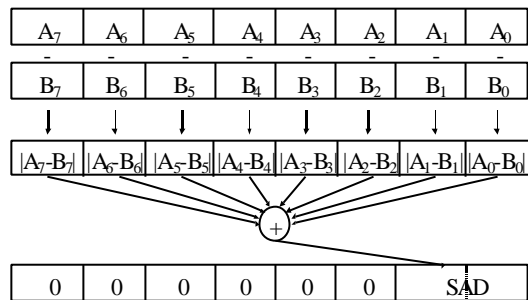


図 11: PSADBW

ブロック・マッチングは、図 12 に示すように PSAD 命令を使って実装します。この操作を行うためのコードの一覧を表 2 に示します。このコードを使用すると、MMX® テクノロジーを使用した場合に比べて、最大で 2 倍のパフォーマンス向上を達成します。

ブロック・マッチングでのメモリ・アクセスの特性により、32 バイト境界にまたがるデータをロードする際に、データはキャッシュ・ラインに収まるように分割されます。これは、dx と dy の値が 1 ずつ変化するためです(アドレスは一度に 1 バイトずつ変化します)。データは一度に 8 バイトずつロードされます。

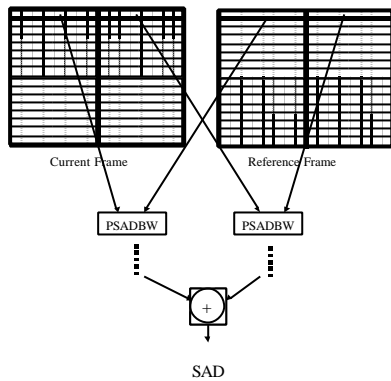


図 12: PSADBW の利用によるブロック・マッチング

```

psad_top:                // 16 x 16 block
matching
    // Do PSAD for a row, accumulate
results
    movq mm1, [esi]
    movq mm2, [esi+8]
    psadbw mm1, [edi]
    psadbw mm2, [edi+8]

    // Increment pointers to next row
    add esi, eax
    add edi, eax

    // Accumulate diff in 2 accumulators
    paddw mm0, mm1
    paddw mm7, mm2

    dec ecx                // Do all 16 rows of
macroblock
    jg psad_top

    // Add partial results for final SAD
value
    paddw mm0, mm7
    
```

表 2: ブロック・マッチング

階層形式の動き推定は、複雑な演算を簡素化するための手法としてよく使われます。この方法を使うと、より有効な動きベクトルの値が得られる場合があります。サブサンプリングのしくみを図 13 に示します。

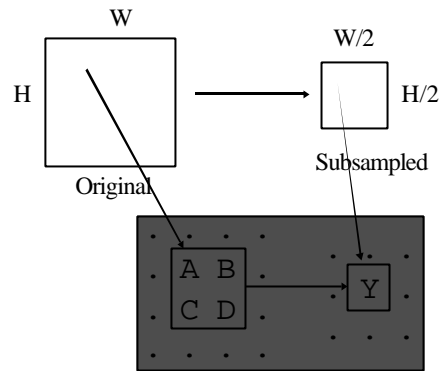


図 13: 階層形式の動き推定でのサブサンプリング

pavg 命令を用いることで、元画像のサブサンプリングを高速化できます。pavgb 命令で行われるオペレーションを図 14 に示します。

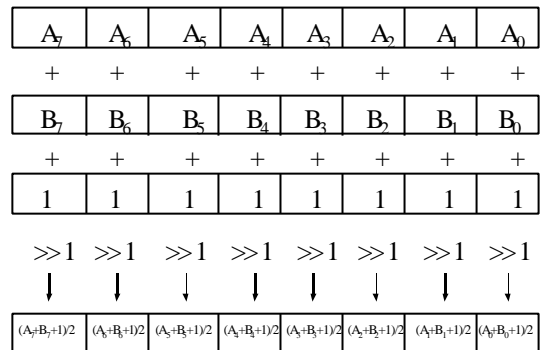


図 14: PAVGB

インターネット・ストリーミング SIMD 拡張命令には、pavgw 命令も用意されています。この命令は、pavgb 命令と同じように機能しますが、4 個の 16 ビット値で平均値の算出を行います。

特筆すべきは、この新命令を使うと加算を行う際の精度が向上することです(pavgb では 9 ビット、pavgw では 17 ビット)。このため、オーバーフロー・エラーを回避できます。平均値が算出されると(2 で除算した後)、その後の計算では元の数値と同程度の精度を確保できます(8 ビット/16 ビット)。

pavg 命令では一度に 2 つの値を処理しますが、pavg 命令を 3 個使って 4 つの値の平均値を算出することもできます。これを擬似コードで表すと次のようになります。

$$Y = \text{pavg}(\text{pavg}(A, B), \text{pavg}(C, D)) - 1$$

この値は、サブサンプリングで一般に使われる $(A + B + C + D + 2)/4$ の計算に近い数値になります。ただし、近似値の計算では、値の 87.5% は正確に一致しますが、残りの 12.5% には 1LSbit (least significant bit) を上限とする誤差が生

じます。動き推定においては、通常この程度の誤差は許容範囲内であると言えます。

動き補正

動き補正(MC)は、ビデオ・デコーダとエンコーダの両方で使われます。デコーダでは iMC (inverse motion compensation)を行い、デコーダでは MC と iMC の両方を行います。これらの計算では、特にエンコーダでの精度がきわめて重要です。エンコーダのローカル・デコーダで、高品質デコーダのオペレーションをトラックする必要があります。MCでは、双方向 B フレームで 2 つの値の補間が必要な場合があります。これを MPEG 標準で表すと次のようになります。

$$Y = (A + B + 1) / 2$$

movq 命令では、これとまったく同じ計算を行います。

インターネット・ストリーミング SIMD 拡張命令には、プリフェッチ命令とストリーミング・ストア命令が用意されています。動き補正(MC)で頻りにメモリをアクセスする場合、prefetch 命令は MC を高速化します。prefetchnta 命令と prefetcht0 命令はいずれも高速化に貢献しますが、どちらの命令がより効果的であるかはデコーダの実装方法によって異なります。デコードされた画像をグラフィックス・カードのメモリに書き込むタイプのデコーダの場合、movntq 命令を使うと、再度使用することのないデータがキャッシュに留まることがないため、処理の高速化に効果があります。

離散コサイン変換

離散コサイン変換(DCT)および逆離散コサイン変換(iDCT)は、ビデオ・コーデックで使われます。デコーダでは iDCT を使い、エンコーダでは DCT を使います。エンコーダにローカル・デコーダがある場合、通常は iDCT を使います。これらの処理では、インターネット・ストリーミング SIMD 拡張命令の利用により高速化が見込める場合がありますが、高速化の度合いはアプリケーションによって大きく異なります。SIMD 浮動小数点命令を使うと、DCT/iDCT の計算がきわめて正確に行えます。ただし、MMX®テクノロジーに見られる SIMD 整数命令を使っても、IEEE 1180-1990 (参考資料[3])標準に準拠します。一般に、家電の場合は、SIMD 整数命令を使うと十分な精度と速度が得られます。業務用または最先端のコーデックには、SIMD 浮動小数点命令の使用をお勧めします。インテルのイメージ処理ライブラリを利用すれば、どちらの場合も簡単に実装できます。

可変長エンコード

エンコーダでは、離散コサイン変換(DCT)と量子化を行った後、その値をもとにビット・ストリームを生成する必要があります。このプロセスは可変長エンコード(VLE)と呼

ばれます。特に B フレームの場合などは、0 の値を検出して「読み飛ばす」必要が頻りに生じます。こうした 0 の値を処理する際は、pmovmskb 命令によって 8 個の値を求めます。pmovmskb 命令を使ってこの操作を行う場合の例を表 3 に示します。

pxor	mm7,mm7	// zero mm7
movq	mm0,[esi]	// get eight Q values
pcmpeqb	mm0,mm7	// find zeros
pmovmskb	eax,mm0	// 8 flags into eax

表 3: 可変長エンコード

eax が 0xff を保持する場合、8 個すべての値が 0 であることを示します。

カラー変換

カラー変換は、エンコーダとデコーダの両方で使われます。エンコーダでは、そのままではエンコードできない形式のデータを受け取る場合がしばしばあります(クロマ・フォーマットの違い、インターリーブ・データと二次元データの違いなど)。また、デコーダでは、デコードされた画像を、別のカラー空間を使ってグラフィックス・カードのメモリに書き込まなければならない場合があります。このような場合にも、カラー変換が必要になります。

エンコーダで行うカラー変換は、一般に大量のメモリを必要とします。この操作では、画像データをメイン・メモリからロードし(ビデオ・キャプチャ・カードから直接メモリ・アクセスするなど)、何らかの(大抵はシンプルな)計算を実行し、そのデータをメモリに書き出します。prefetchnta 命令では、L2 キャッシュをバイパスすることにより、カラー変換に要する時間を短縮します。非テンポラルなプリフェッチ命令は、カラー変換に最も適したプリフェッチ命令である場合が多いですが、これはコーデックによる入力を繰り返す必要がないからです。ストア命令では、通常のストア(movq など)でデータを保存でき、画像はカラー変換を行った後に L2 キャッシュに格納されます。

まとめ

データをメモリに格納する順序と、そのデータがプロセッサとキャッシュ間でどのように移動するかによって、アプリケーションのパフォーマンスに大きく影響します。SIMD 命令では総合的にハイブリッド・データ・オーダーが最も効果的ですが、アプリケーションで通常の AoS オーダーを使用せざるを得ない場合は、データを SIMD レジスタ内で SoA オーダーに転置してから処理するのが最も有効な方法です。プリフェッチ命令は、メモリ・レイテンシを低減し、メモリ帯域の使用効率を高める効果があります。大きなデータ・ブロックを処理する場合は、データを Pentium® III プロセッサのキャッシュに収まるサイズのサブセットに分割すると、メモリ・オーバーヘッドの増大を防ぐことができます。

3D 変換では、パックド・データを 4 つ格納できる SIMD レジスタとメモリを効率的に使用することにより、パフォーマンスに大きな差が出ます。変換コードをテストした結果、Pentium III プロセッサに最適化されたアセンブリ・コードを使うと、標準 C コードの場合に比べて 3~3.7 倍のパフォーマンス向上が見られました。3D 照明においても、近似値命令と分岐排除命令の使用により、パフォーマンスが格段に(最大で 4 倍)向上するという結果が出ています。

整数 SIMD 拡張命令は、一般的な画像処理アルゴリズムを SIMD 形式で実装するのに役立ち、MMX®テクノロジーの利用によるパフォーマンス向上をさらに強化します。同様に、浮動小数点を扱うアルゴリズムでは、浮動小数点 SIMD 命令を有効活用することにより精度とパフォーマンスが向上します。また、固定小数点演算の処理を減らすことができるため、コードの開発/検証が容易になります。Intel®パフォーマンス・ライブラリ集(参考資料[4])に含まれる高性能イメージ処理ライブラリには、こうしたさまざまな手法が取り入れられています。

インターネット・ストリーミング SIMD 拡張命令を利用することにより、動き推定、動き補正、可変長エンコード、カラー変換といったビデオ・コーデックでよく使われる関数の処理を高速化できます。新しい psadbw、pavgb、pavgw 命令や、プリフェッチ命令、ストリーミング・ストア命令などは、特にビデオ・コーデックで効果を発揮します。テストの結果、動き推定では 2 倍、エンコード・アプリケーション全体では 1.3 倍の高速化が達成されています。

謝辞

この記事で取り上げた最適化の概念は、インテルのマイクロプロセッサ・ラボ、フォルサム・デザイン・センター、デベロッパ・リレーションズ&エンジニアリングなどのグループに属する組織、および所属エンジニアによる取り組みをベースにしてまとめたものです。

参考資料

- [1] J. Wolf 著 『VTune™ パフォーマンス拡張環境の利用による、Pentium® III プロセッサのインターネット・ストリーミング SIMD 拡張命令を活用するためのプログラミング手法』、インテル・テクノロジー・ジャーナル、1999 年第 2 四半期号
- [2] A. Watt 著 『3D Computer Graphics 2nd Edition』
- [3] 『IEEE Circuits and Systems Society, IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform』、IEEE Std. 1180-1990.
- [4] <http://developer.intel.com/vtune/>

著者紹介

James Abel、将来のインテル・プロセッサ向けのソフトウェア・アプリケーションを担当。インテルに入社以来 10 年間、インテルのソフトウェア Dolby® Digital デコーダ、エンベデッド・マイクロコントローラ設計、Design Automation など、さまざまなソフトウェア/ハードウェアの開発に携わる。1983 年にイリノイ州ブラッドリー大学でエンジニアリングの理学士号を取得。1991 年にアリゾナ州立大学でコンピュータ・サイエンスの理学修士号を取得。

電子メール・アドレス: james.c.abel@intel.com

Kumar Balasubramanian、ソフトウェア・メーカ各社に働きかけ、インテルの新しいプロセッサの機能を活用したアプリケーションの開発を支援する。これまでに、インターネット・ストリーミング SIMD 拡張命令を統合したビジネス・アプリケーションをいくつか手がける。インテルに入社以来 7 年間、インテルの CAD エンジニアリング組織で主導的役割を果たし、インテル・アーキテクチャ・ラボと共同で MMX®テクノロジーを取り入れた最初のアプリケーションの開発にも携わる。ダートマス大学でコンピュータ・エンジニアリングの理学修士号を取得。

電子メール・アドレス: kumar.balasubramanian@intel.com

Mike Barger, 1997 年にインテルに入社し、ソフトウェア・パフォーマンス・ラボに所属。入社以来、PC 向け 2D/3D グラフィックス・アプリケーションのパフォーマンス・チューニングを担当。これまでに、数々の 3D ゲーム・タイトルや MPEG モーション・ビデオを手がける。プリガム・ヤング大学で電気工学の理学士号を取得。

電子メール・アドレス: michael.l.barger@intel.com

Tom Craver、3D グラフィックス・ソフトウェア・メーカ各社のドライバ・ソフトウェアをインテルの最新プロセッサに最適化するための支援に努める。以前は、ケーブル・モデムやインテル DVI マルチメディア・テクノロジー対応製品のドライバ/ユーザ・インターフェース・ソフトウェアの開発と検証を担当。インテル入社以前はデビッド・サーノフ研究所(ニュージャージー州プリンストン)の研究者であり、それ以前には AT&T ベル研究所に在籍。イリノイ大学で物理学とコンピュータ・サイエンスの理学士号を取得し、パーデュー大学で理学修士号を取得。

電子メール・アドレス: tom.r.craver@intel.com.

Mike Philpot、デスクトップ向けソフトウェア・メーカに働きかけ、インテルの最新プロセッサの機能を各社のアプリケーションに統合するための支援に努める。最近では、インターネット・ストリーミング SIMD 拡張命令を活用した 3D ゲームの開発を支援。インテルに入社以来 10 年間、デジタル・ビデオ圧縮技術やケーブル・モデムをはじめとするテクノロジー分野で、エンジニアからマネージャまでさ

さまざまな職務を経験。ゼネラル・モーターズ工科大学で機械工学の理学士号を取得、ミシガン大学でコンピュータ・エンジニアリングの理学修士号を取得。

電子メール・アドレス: mike.p.phlipot@intel.com