

**ストリーミング SIMD 拡張命令 2(SSE2)を使用した
Viterbi デコーディングでの
隠れマルコフ・モデルの評価**

バージョン 2.0

2000 年 7 月

資料番号: 248671J-001

【輸出規制に関する告知と注意事項】

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

【資料内容に関する注意事項】

・本ドキュメントの内容を予告なしに変更することがあります。

・インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。

・インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。

・本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。

インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。

・いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複製することは禁じられています。

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社 資料センター

〒305-8603 筑波学園郵便局 私書箱115号

Fax: 0120-47-8832

*一般にブランド名または商品名は各社の商標または登録商標です。

Copyright © Intel Corporation 1999, 2000

目次

1	はじめに	5
2	Viterbi デコーディング・アルゴリズム.....	5
2.1	Viterbi デコーディング・アルゴリズムを使用して HMM を評価するアプリケーション	8
2.2	Viterbi デコーディング・アルゴリズムのコーディング.....	10
2.2.1	実現技法	14
3	パフォーマンス	15
3.1	パフォーマンスの向上/改善	15
4	結論	16
5	機能コードの例	16
6	一般的な最適化を使用したインテル® Pentium® III プロセッサ SIMD コード	18
7	状態固有の最適化を使用したインテル® Pentium® III プロセッサ SIMD コード	21
8	一般的な最適化を使用したインテル® Pentium® 4 プロセッサ SIMD コード.....	27
9	状態固有の最適化を使用したインテル® Pentium® 4 プロセッサ SIMD コード.....	30
付録 A	パフォーマンス・データ	A-1
	パフォーマンス・データの改訂履歴.....	A-1
	テスト・システムの構成.....	A-3

改訂履歴

改訂番号	改訂履歴	改訂時期
2.0	インテル® Pentium® 4 プロセッサに関する更新	2000年7月
1.0	初版	1999年9月

参考資料

このアプリケーション・ノートでは次の資料を参考にした。これらの資料には、本書で取り上げた事項を理解するための背景情報が記載されている。

- 『A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition』 Rabiner, L.R., *IEEE の会報*, 1989年2月, Vol. 77, No. 2
- 『Speech Recognition: No Longer a Dream but Still a Challenge』 Quinnet, Richard A., *EDN*, 1995年1月19日号, 41～46ページ
- 『ストリーミングSIMD 拡張命令(Streaming SIMD Extensions)を使用した Viterbi デコーディングでの隠れマルコフ・モデルの評価』、インテル・アプリケーション・ノート AP-811、Copyright 1998 (<http://developer.intel.com/vtune/cbts/strmsimd/811down.htm>を参照)
- 『Using MMX™ Instructions to Implement Viterbi Decoding』 インテル・アプリケーション・ノート AP-569、Copyright 1996、<http://developer.intel.com/drg/mmx/appnotes/ap569.htm>
- 『Dynamic Programming』 第7章、7-1～7-24ページ、インテル音声/信号認識ライブラリ
- 『Fundamentals of Speech Recognition』 Rabiner, L.R., および Juan, B., Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1993年、340ページ

1 はじめに

ストリーミング SIMD 拡張命令 2(SSE2、Streaming SIMD Extensions 2)では、SIMD(Single Instruction Multiple Data)倍精度浮動小数点命令、および SIMD 整数命令が IA-32 インテル®アーキテクチャに新しく導入された。倍精度 SIMD 命令による機能拡張の方法は、ストリーミング SIMD 拡張命令(SSE)で導入された単精度 SIMD 命令による機能拡張とよく似ている。128 ビットの整数 SIMD 拡張命令は、64 ビットの整数 SIMD 拡張命令の完全なスーパーセットで、より多くの整数データ型、整数と浮動小数点間のデータ型変換、キャッシュとシステム・メモリの効果的使用をサポートする命令が追加されている。これらの命令は、3D グラフィックス、リアルタイムの物理的な現象、空間的(3D)オーディオ、ビデオ・エンコーディング/デコーディング、暗号化、および科学計算アプリケーションによく使用される演算を高速化する。SSE2 の 128 ビット整数 SIMD 拡張命令は、XMM レジスタを使用して 128 ビットのデータを一度に処理でき、Viterbi デコーディング・アルゴリズムと隠れマルコフ・モデルの評価などの重要なアルゴリズムのコードの性能を、インテル® MMX®テクノロジー命令と SSE を使用した従来のコードより向上させる。このアプリケーション・ノートでは、SSE2 を使用して Viterbi アルゴリズムをコーディングし、隠れマルコフ・モデルを評価する方法の説明とそのコード例を示す。

アプリケーション・ノート AP-569 『Using MMX™ Instructions to Implement Viterbi Decoding』では、MMX®命令を使用して、スカラ・コードに対してパフォーマンスを 2 倍に向上させる方法について説明した。アプリケーション・ノート AP-811 『SSE を使用した Viterbi デコーディングでの隠れマルコフ・モデルの評価』では、SSE を使用して一度に 4 つのデータ要素を操作する(SIMD 幅を 2 倍に拡張することによって、さらにパフォーマンスを向上させる方法について説明した。このアプリケーション・ノートでは、SSE2 を使用して、SSE を使用したコードと比較してパフォーマンスを大きく向上させる方法について説明する。

2 Viterbi デコーディング・アルゴリズム

この章は、AP-811 の第 2 章に基づいている。この章では、Viterbi デコーディング・アルゴリズムを使用して隠れマルコフ・モデル(HMM)を評価する方法について簡単に説明する。本書の 4 ページに示した参考資料を使用して、これらの事項について詳しく学習することをお勧めする。AP-811 の 2.2 節には、隠れマルコフ・モデルに関する基礎知識が記載されている。

HMM (λ)は、 $j = 1, 2, \dots, n$ [5]の番号が付いた n 個の状態のマルコフ連鎖で構成される。遷移とは、ある状態(i)から別の状態(j)に移ることである。HMM は、パラメータ・セット $\{A, B, \pi\}$ によって、次のように記述される。

- A は、遷移確率と呼ばれるマトリックス $\{a_{ij}\}$ である。 a_{ij} は、状態 i から状態 j への遷移の確率である。
- B は、出力確率と呼ばれるマトリックス $\{b_i(k)\}$ である。 $b_i(k)$ は、状態 i にあるときに観察 k を生成する確率である。
- π は、初期確率と呼ばれるベクトル $\{\pi_i\}$ である。 π_i は、状態 i から始まる確率である。

発生する可能性のある遷移のタイプに応じて、各種の HMM 構造を使用できる。この文書では、制約ジャンプ・モデルについてのみ検討する。図 1 に示すように、このモデルでは状態遷移が制限され、状態 i から、状態 i 、 $i+1$ 、または $i+2$ への遷移だけが起こり得る($j > i+2$ の場合と $j < i$ の場合は、 $a_{ij} = 0$)。

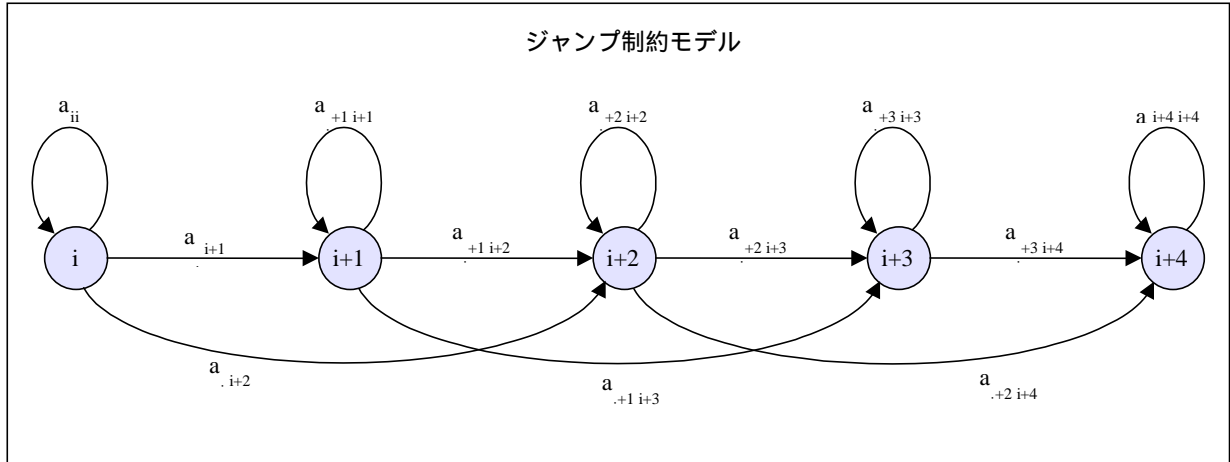


図1: ジャンプ制約隠れマルコフ・モデル

このアプリケーション・ノートでは、離散 HMM についてのみ検討する。離散 HMM では、1 つの状態が M 個の異なる観察シンボルを生成できる。各観察シンボルは、モデル化されるシステムの特定の物理的出力に対応する。個々の観察シンボルは、 $V = \{v_1, v_2, \dots, v_M\}$ として示される。観察シンボルのインデックスは、観察ベクトル O による物理的出力のシーケンスを記述する。

$O = \{O_1, O_2, \dots, O_T\}$ 、ここで、 T はシーケンス内の観察の数

観察ベクトル内の個々の観察は、システムの出力に最もよく一致する観察シンボルのインデックスである。離散 HMM では、出力確率 $b_i(k)$ は、状態 i にあるときに観察シンボル v_k が生成される確率である[1](図 2 を参照)。

$b_i(k) = P(\text{時間 } t|q_t = \text{状態 } i \text{ での } v_k)$ 、ここで、 $k = O_t$ 、

$$1 \leq i \leq N,$$

$$1 \leq k \leq M$$

5つの状態を持つ離散 HMM が与えられ、各状態に以下の3つの確率が関連付けられる。

π - 初期確率

A - 遷移確率

B - 出力確率

このモデルが3つの観察シンボルを持つジャンプ制約モデルだとすると、確率マトリックスは次のようになる。

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ 0 & a_{22} & a_{23} & a_{24} & 0 \\ 0 & 0 & a_{33} & a_{34} & a_{35} \\ 0 & 0 & 0 & a_{44} & a_{45} \\ 0 & 0 & 0 & 0 & a_{55} \end{bmatrix} \quad B = \begin{bmatrix} b_1(1) & b_2(1) & b_3(1) & b_4(1) & b_5(1) \\ b_1(2) & b_2(2) & b_3(2) & b_4(2) & b_5(2) \\ b_1(3) & b_2(3) & b_3(3) & b_4(3) & b_5(3) \end{bmatrix} \quad \pi = \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \\ \pi_4 \\ \pi_5 \end{bmatrix}$$

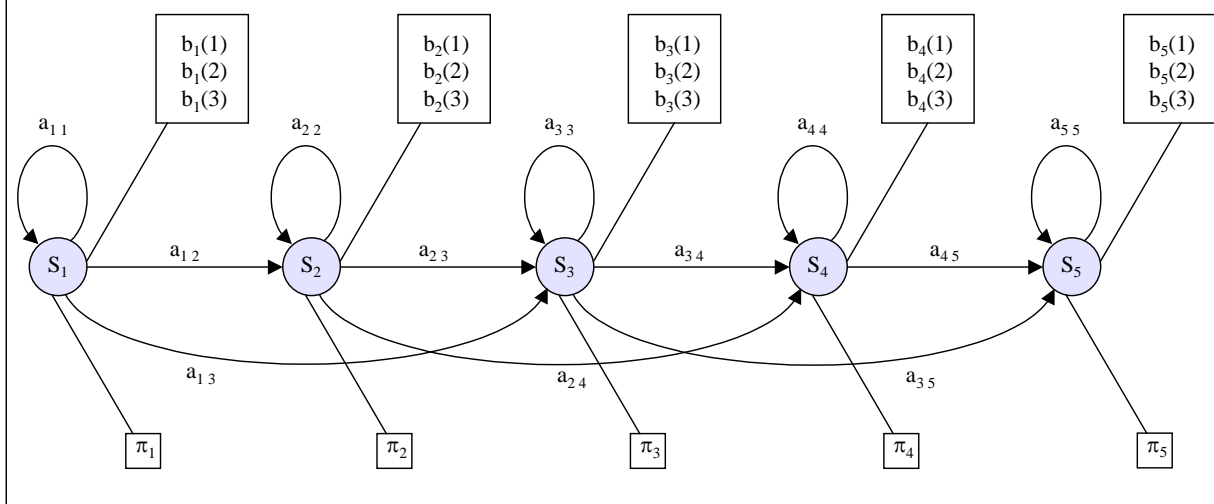


図2: 離散隠れマルコフ・モデルのパラメータ

HMM は、音声認識に使用できる。発声(単語または音節)によって生成する音響スペクトルのシーケンスは、一連の状態遷移によってモデル化される[2]。観察ベクトル O で表す音声サンプルが与えられると、特定の HMM によって観察シーケンスが生成される確率 $P(O|\lambda)$ を計算できる。このモデルでは、異なる状態シーケンスから同じ観察シーケンスが生成される場合があるが、各状態シーケンスは異なる確率に関連付けられる。最も良い確率の状態シーケンスは、その観察シーケンスを生成する可能性が最も高いパスである。この状態シーケンスは、最適パス(P^V)と呼ばれる。

$$P^V = \max[P(Q|O, \lambda)], \text{ ここで、 } Q \text{ は } O \text{ を生成できる状態シーケンスである。}$$

$$P(O|\lambda) \approx P^V$$

HMM 上で最適パスを徹底的に検出しようとするのは現実的ではない。Viterbi デコーディング・アルゴリズムは、最適パスを効率的に見つける方法である。このアルゴリズムを使用して、最適パスとそのパスの状態シーケンスを見つけられる。最適パスの状態シーケンスを見つけるのを、バックトレースと呼ぶ[6]。このアプリケーション・ノートに記載されたコードは、最適パスを解くだけであり、バックトレースは行わない。

Viterbi アルゴリズムを使用して、観察シーケンス O と HMM モデル λ に基づいて、最適パス (P^V) を見つけられる。以下に Viterbi アルゴリズムの説明を示す[1]。

- 時間 t での 1 つのパス上の最適スコア(最高確率)として、 $\delta_t(i)$ を定義する。このスコアには、最初の t 個の観察が含まれる。状態シーケンスは S_i で終了する。

$$\delta_t(i) = \max_{q_1, q_2, \dots, q_{t-1}} P(q_1 q_2 \dots q_t = S_i, O_1 O_2 \dots O_t | \lambda)$$

および

$$\delta_{t+1}(j) = [\max_i \delta_t(i) a_{ij}] * b_j(O_{t+1})$$

- Viterbi アルゴリズム

- 1) 初期設定:

$$\delta_1(i) = \pi_i b_i(O_1) \quad 1 \leq i \leq N \quad (1)$$

- 2) 再帰:

$$\delta_t(j) = \max_{i=j, j-1, j-2} [\delta_{t-1}(i) a_{ij}] * b_j(O_t) \quad \begin{array}{l} 2 \leq t \leq T \\ 1 \leq j \leq N \end{array} \quad (2)$$

- 3) 終止:

$$P^V = \max_{1 \leq i \leq N} [\delta_T(i)] \quad (3)$$

2.1 Viterbi デコーディング・アルゴリズムを使用して HMM を評価するアプリケーション

隠れマルコフ・モデルの 1 つのアプリケーションは、分離語認識器(Isolated Word Recognizer: IWR)[1] (276-277) である。Rabiner は、システムの語彙内の各単語 w について、各 HMM をどのように設計し、訓練するかを説明している。IWR は、最初に入力音声信号(話された単語)の特徴解析を実行する(図 3 を参照)。特徴解析は、音声信号を符号化スペクトル・ベクトルの時間シーケンス(観察シーケンス)に変換する。すでに説明したように、それぞれの観察は、物理的出力に最もよく一致する観察シンボルのインデックスである。この音声認識システムでは、観察シンボルは、符号化スペクトル・ベクトルである。入力信号から観察シーケンスを得ると、特定の HMM で観察シーケンスが生成される確率 $P(O|\lambda)$ が、すべてのモデルについて計算する。 $P(O|\lambda)$ を求めるために、Viterbi アルゴリズムで各 HMM を評価して、HMM ごとに最適パスを見つける。最適パスは、HMM 内の最も可能性の高い状態シーケンスが、特定の観察シーケンスをどの程度よく記述しているかを表す。最も良い確率のモデルが見つかり、話した単語は認識したと見なされる。

$$w^* = \operatorname{argmax}_{1 \leq w \leq W} [P(O|\lambda^w)]$$

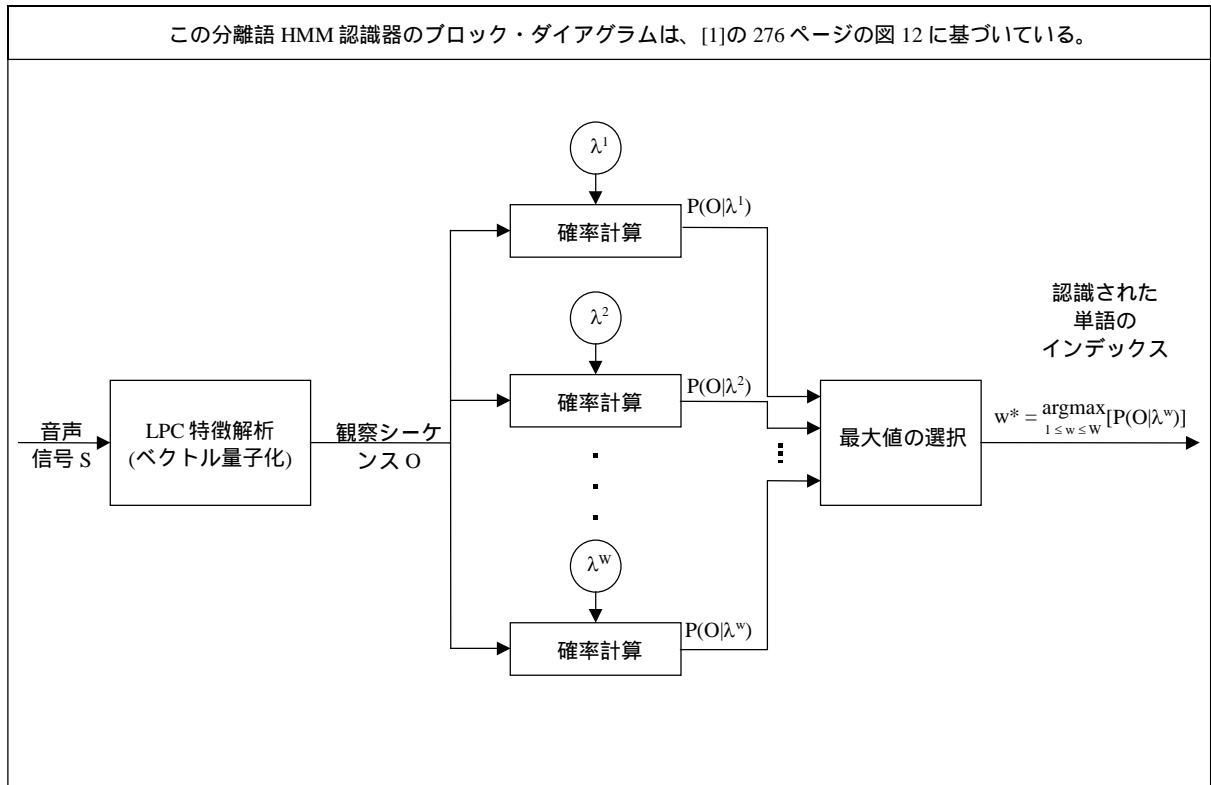


図3: 分離語認識器のブロック・ダイアグラム

2.2 Viterbi デコーディング・アルゴリズムのコーディング

Viterbi アルゴリズムを構成する 3 つの式をコーディングする際は、両辺に負の対数を使用する。負の対数を使用するのは、スケーリングの問題を避けるために、すべての乗算を正の数の加算に変換するからである。以前のアプリケーション・ノート([3]、[4])の表記を使用すると、Viterbi アルゴリズムの 3 つの式は次のようになる。

1) 初期設定:

$$\text{Dist}(j,1) = \text{Pi}(j) + \text{bProb}(j,k(1)) \quad 1 \leq j \leq N \quad (4)$$

2) 再帰:

$$\text{Dist}(j,t+1) = \min_{i=j,j-1,j-2} [\text{Dist}(i,t) + \text{aProb}(j,i)] + \text{bProb}(j,k(t+1)) \quad \begin{array}{l} 1 \leq t \leq T \\ 1 \leq j \leq N \end{array} \quad (5)$$

3) 終止:

$$\text{Dist}^v = \min_{1 \leq j \leq N} \text{Dist}(j,T) \quad (6)$$

ここで、

$$\begin{aligned} \text{Dist}^v &= -\log(P^V) \\ \text{Dist}(j,t) &= -\log(\delta_t(j)) \\ \text{aProb}(j,i) &= -\log(a_{ij}) \\ \text{bProb}(j,k(t)) &= -\log(b_j(O_t)) \\ \text{Pi}(j) &= -\log(\pi_j) \end{aligned}$$

アルゴリズムの処理時間の大部分は、式 5 に費やされる。この式は、次のように展開できる。

$$\text{Dist}(j,t+1) = \min \left[\left\{ \text{Dist}(j,t) + a\text{Prob}(j,j) \right\}, \left\{ \text{Dist}(j-1,t) + a\text{Prob}(j,j-1) \right\}, \left\{ \text{Dist}(j-2,t) + a\text{Prob}(j,j-2) \right\} \right] + b\text{Prob}(j,k(t+1)) \quad \begin{matrix} 1 \leq t \leq T \\ 1 \leq j \leq N \end{matrix} \quad (7)$$

この式は、図 4 に示すように、SIMD 形式に変換できる。

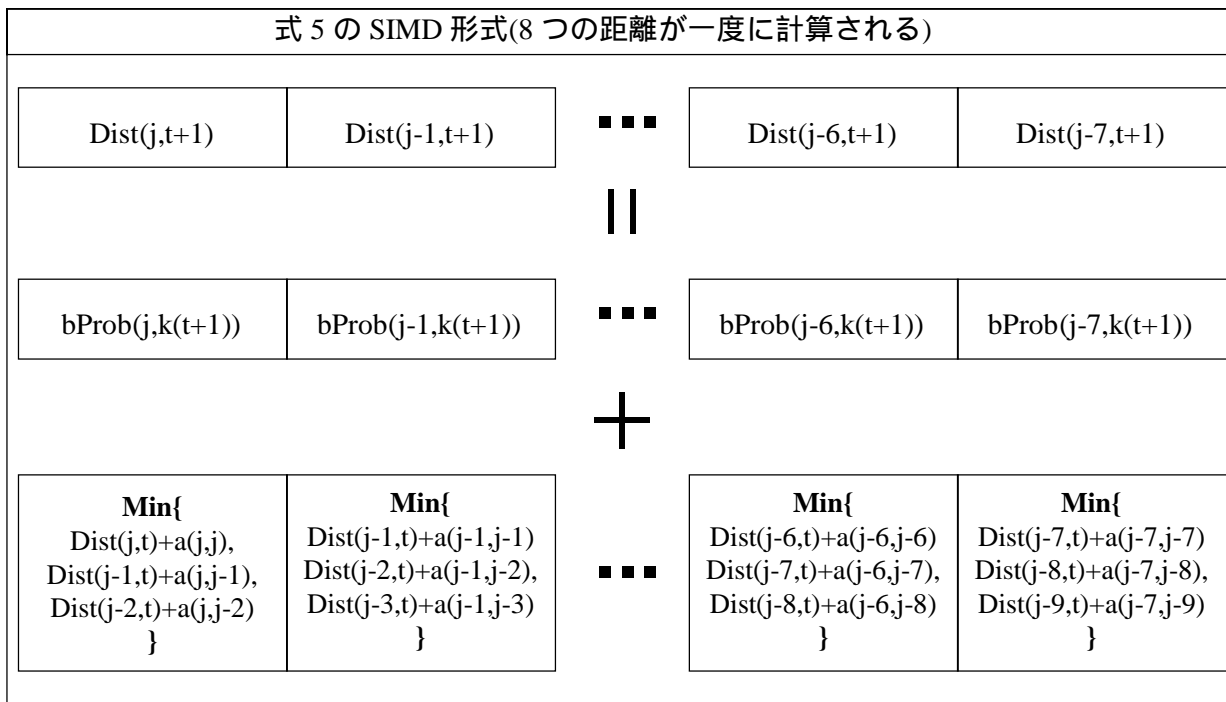


図4: SIMD 形式による式 5

注: 簡単にするために、aProb の要素は aProb(x,y) から a(x,y) に短縮されている。また、一番左の列は、式 7 のスカラ・バージョンである。

式 5 の SIMD 形式を使用するために、j の減少する値(j = N, N-1, ..., 1)を使用して、距離が計算される。インテル® C++ SIMD クラス・ライブラリ(SIMD 演算対応)を使用して、次のように式 5 の SIMD 形式を SSE2 で表現できる。

```
for(...)
{
    //load variables dist8_1 and dist8_2
    sum0 = sat_add(dist8[j],a8[j]);
    sum1 = sat_add(dist8_1,a8_1[j]);
    sum2 = sat_add(dist8_2,a8_2[j]);

    min0 = simd_min(sum2,simd_min(sum1,sum0));
    dist8[j] = sat_add(b8[j],min0);
}
```

Viterbi アルゴリズムのステップ 2 で処理時間の大部分が費やされるが、アルゴリズムのステップ 1 とステップ 3 も、SIMD 命令を使用して最適化ができる。インテル C++ SIMD クラス・ライブラリ(SIMD 演算対応)を使用してコーディングされた Viterbi デコーディング・アルゴリズムについては、第 6 章を参照のこと。

第 5 章 ~ 第 9 章の Viterbi デコーディング・アルゴリズムのコードをよく理解するには、関数に渡されるパラメータを理解する必要がある。このアプリケーション・ノートで使用される関数は、以下の引数を必要とする。

1. ObsVect 音声信号を観察のシーケンスとして記述する観察ベクトル。各観察は、観察シンボルに対応するインデックスである。
2. ObsLen 観察シーケンスの長さ(T)。
- 3,4,5. a, a_1, a_2 3つのベクトルを使用して、遷移確率マトリックス(aProb)を表す。ジャンプ制約モデルの aProb マトリックスは、1行当たり3つの0でない要素を持つ(vh 2を参照)。この3つの0でない要素は、状態 j の3つの可能な遷移 aProb(j,j)、aProb(j,j-1)、および aProb(j,j-2)を表す。ベクトル“a”は、aProb(j,j)の遷移確率を表す。ベクトル“a_1”は、aProb(j,j-1)の遷移確率を表す。ベクトル“a_2”は、aProb(j,j-2)の遷移確率を表す。これらのベクトルについては、後で詳しく説明する。
6. bprob 出力確率。これは配列へのポインタの配列である。ポインタを保持する配列のサイズは、観察シンボルの数 M である。各ポインタが指す配列のサイズは、状態の数 nStates である(図 5 を参照)。
7. Pi 初期確率ベクトル。
8. nstates HMM 内の状態の数。
9. DistBuffer 累積された距離を格納する一時的なバッファ。

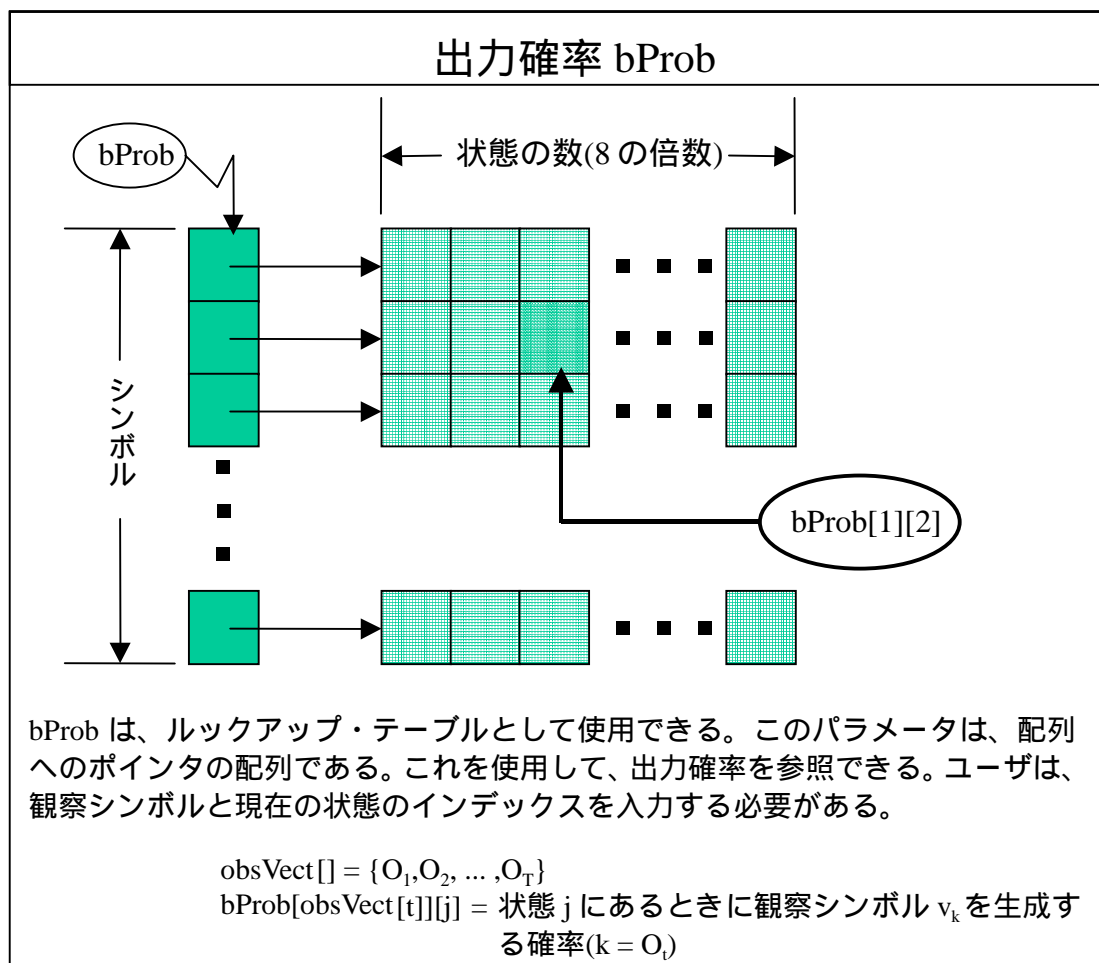


図5: 出力確率(bProb)のコードの説明図

このアプリケーション・ノートでは、 $aProb(j,j)$ 、 $aProb(j,j-1)$ 、および $aProb(j,j-2)$ を表す 3 つのベクトルを使用して、 $aProb$ マトリックスをコーディングする。

ベクトル a: $a_[] = \{a(j,j), a(j-1,j-1), a(j-2,j-2), \dots, a(j-n,j-n)\}; //aProb(j,j)$

ベクトル a₁: $a_1[] = \{a(j,j-1), a(j-1,j-2), a(j-2,j-3), \dots, a(j-n,j-n-1)\}; //aProb(j,j-1)$

ベクトル a₂: $a_2[] = \{a(j,j-2), a(j-1,j-3), a(j-2,j-4), \dots, a(j-n,j-n-2)\}; //aProb(j,j-2)$

$aProb$ マトリックスは、アライメントの合わないロードを避けるために、3 つのベクトルを使用してコーディングされる。このコードでは、ユーザがすべての確率マトリックス/ベクトルを 8 の SIMD 幅に合わせてパディングしているとする。つまり、HMM に 6 つの状態が含まれる場合、プログラマが 8 の SIMD 幅に合わせて $distBuffer$ 、 Pi 、 $aProb$ 、および $bProb$ 配列をパディングしているとする。このパディングによって、不要な分岐が避けられ、余分な要素を処理するコードを除去できる。

2.2.1 実現技法

式 5 の SIMD コードは、1 回の反復当たり 8 つの距離を処理するために、アライメントの合わないロードを少なくとも 2 回実行する必要がある。距離ベクトルが 16 バイトにアライメントされているとすると、アライメントの合ったロード(`movdqa` 命令)を使用して、最初の 8 つの距離をロードできる。しかし、2 番目と 3 番目のアライメントの合わないロードには、`movdqu` 命令を使用する必要がある。

```

movdqa xmm0, [edi+eax]    //Dist(j-7,t)...Dist(j,t)
movdqu xmm1, [edi+eax+2]  //Dist(j-8,t)...Dist(j-1,t), unaligned load
movdqu xmm2, [edi+eax+4]  //Dist(j-9,t)...Dist(j-2,t), unaligned load

paddsw xmm0, [ebx+eax]    //Dist8      += aProb8
paddsw xmm1, [ecx+eax]    //Dist8_1   += aProb8_1
paddsw xmm2, [edx+eax]    //Dist8_2   += aProb8_2

pminsw xmm0, xmm1
pminsw xmm0, xmm2        //Dist8=Min(Dist8,Dist8_1,Dist8_2)

paddsw xmm0, [esi+eax]    //Dist8 = Dist8 + bProb
movdqa [edi+eax], xmm0

```

HMM に含まれる状態の数が少ない場合は、アライメントの合わないロードのために問題が起こる可能性がある。つまり、書き込みの直後にアライメントの合わない読み出しを使用すると、データが転送されない。この場合、メモリ階層内のデータが更新されるまで、アライメントの合わない読み出しの実行を待機する必要がある。

アライメントの合わない読み出しによるストール(ストア・フォワードのペナルティ)を避ける 1 つの方法は、レジスタ内での距離の保持である。アライメントの合わない読み出しを避けるもう 1 つの方法は、アライメントの合った読み出しを 2 回実行し、シフトと OR 演算によってデータを結合することである。これらの手法は、インライン ASM、インテル® C/C++コンパイラ組み込み関数、またはインテル C/C++クラス・ライブラリ(SIMD 演算対応)を使用してコーディングできる。これらの最適化手法を、以下の各種のコード例で示す。

1. 機能コード

C++で作成された、最適化されていない機能コード。このコードを使って、Viterbi アルゴリズムを理解できる。

2. 一般的な最適化を使用したインテル® Pentium® III SIMD クラス・コード(PIVEC_GEN)

PIVEC_GEN は、インテル C/C++クラス・ライブラリ(SIMD 演算対応)の SIMD 整数ベクトル・クラス(IVEC)を使用して、Viterbi アルゴリズムを実行する。このコードは Pentium III プロセッサ向けであるため、`Is16vec4`(4 つの要素の 16 ビット符号付き整数ベクトル・クラス)が使用される。このクラスは、64 ビット MMX テクノロジ・レジスタと、MMX テクノロジ命令および SSE を使用する。このコードは、アライメントの合わない読み出しを、2 回のアライメントの合った読み出し、2 回のシフト、および 1 回の OR 演算で置き換える。この例は、(HMM 状態の数に依存しない)一般的な最適化を

示す。アライメントの合ったロードによって、ストア・フォワードのペナルティが避けられる。

3. 状態固有の最適化を使用した Intel® Pentium® III SIMD クラス・コード (PIVEC)

PIVEC も、`Is16vec4` クラスを使用して Viterbi アルゴリズムを実行するコードである。このコードは、ストア・フォワードのペナルティを回避し、さらに HMM 状態の数に固有の最適化を使用する。使用される最適化手法については、コード内のコメントを参照のこと(第 7 章)。

4. 一般的な最適化を使用した Intel® Pentium® 4 SIMD クラス・コード (WIVEC_GEN)

WIVEC_GEN は、Intel C/C++ クラス・ライブラリ (SIMD 演算対応) と `Is16vec8` クラス (8 つの要素の 16 ビット符号付き整数ベクトル・クラス) を使用する。このコードは、Pentium 4 プロセッサ向けに最適化されている。`Is16vec8` クラスは、XMM レジスタ (128 ビット・レジスタ) と SSE2 を使用する。このコードは、PIVEC_GEN コードと同じ一般的な最適化手法を使用する。つまり、アライメントの合わないロードを、2 回のアライメントの合ったロード、2 回のシフト、および 1 回の OR 演算で置き換えて、ストア・フォワードのストールを回避する。

5. 状態固有の最適化を使用した Intel® Pentium® 4 SIMD クラス・コード (WIVEC)

WIVEC は、`Is16vec8` ベクトル・クラスを使用し、ストア・フォワードのペナルティを避けるように最適化され、さらに HMM 状態の数に固有の最適化手法を使用する。使用される最適化手法については、コード内のコメントを参照のこと(第 9 章)。これは、このアプリケーション・ノートで説明する 5 種類のコードのうち、最も効率的なコードである。この Pentium 4 プロセッサ向け IVEC コードを使用すれば、最適な Pentium III プロセッサ向け IVEC コードと比べて、処理速度が大きくアップする。

3 パフォーマンス

3.1 パフォーマンスの向上/改善

Viterbi デコーディング・アルゴリズムについては、アプリケーション・ノート AP-569 と AP-811 ですでに説明した。SSE2 を使用して、上記の文書で説明したコードをさらに最適化できる。SSE2 は SIMD 幅を拡張する (同時に処理できる要素の数が 4 つから 8 つに増える)。SIMD 幅の拡張は、SSE2 による高速化の主要な要因である。`pminsw` 命令も、スカラー・コードに対する処理速度のアップに非常に効果的である。`pminsw` 命令は、AP-811 の SSE コードにも使用されている。

ストア・フォワードのペナルティの回避も、処理速度を大きくアップさせる (2.3.1 項を参照)。アライメントの合わないロードを避けるために、状態遷移確率は 3 つの別々のベクトルに変換される。確率ベクトルおよびマトリックスのパディングによって、余分な要素を処理するコードと分岐が不要になり、さらに処理が高速化される。さらに、HMM 状態の数に固有の最適化も使用できる (第 7 章と第 9 章を参照)。

4 結論

SSE2 は、Viterbi アルゴリズムを使用した HMM の評価のパフォーマンスを大きく向上させる。SSE2 による高速化の主要な要因は、SIMD 幅の拡張と `pminsw` 命令である。その他の最適化手法には、ストア・フォワードのペナルティを避ける、ベクトルのパディングによって余分な要素を処理する分岐を減らす、HMM 状態の数に固有の最適化手法を使用するなどがある。

5 機能コードの例

C++ で作成された、最適化されていない機能コードを以下に示す。このコードを使って、Viterbi アルゴリズムを理解できる。

```
short const MaxShort    = 0x7FFF;
//Signed 16-bit integers are used.  Thus positive numbers are saturated to
//0x7FFF
inline short add_with_saturate(short x, short y)
{
    return((((int)x+(int)y > MaxShort) ? MaxShort : (x+y));
}

inline short minx(short a, short b)
{
    return((a<b)?a:b);
}

//functional code
short hmm_kernel::functional(unsigned int *obsVect, int obsLen,
    short *a,short *a_1,short *a_2, short **bProb, short *pi, int nStates,
    short *distBuffer)
{
    short    minDist;
    short    *b = bProb[obsVect[0]];
    short    *dist = distBuffer;
    short    *dist_1 = &distBuffer[1];
    short    *dist_2 = &distBuffer[2];
    short    sum_0;
    short    sum_1;
    short    sum_2;
    short    min0;

    int      i = 0;

    if (nStates == 0)
```

```
    return(0);

//step 1: Initialization
for( i = 0; i < nStates; i++)
    {dist[i] = add_with_saturate(pi[i],b[i]);}

//Step 2: Recursion
//Loop for each observation or obsCount
for(i = 1; i < obsLen; i++)
{
    b = bProb[obsVect[i]];

    //OneStateLoop
    for(int j = 0; j < nStates; j++)
    {
        sum_0 = add_with_saturate(dist[j+0],a[j]);
        sum_1 = add_with_saturate(dist_1[j+0],a_1[j]);
        sum_2 = add_with_saturate(dist_2[j+0],a_2[j]);

        min0 = minx(sum_2,minx(sum_1,sum_0));
        dist[j+0] = add_with_saturate(b[j+0],min0);
    }
}

//Step 3: Termination
// Get Minimum Distance
minDist = distBuffer[0];
for (i = 1; i<nStates; i++)
{
    minDist = minx(minDist,distBuffer[i]);
}

return(minDist);
}
```

6 一般的な最適化を使用したインテル® Pentium® III プロセッサ SIMD コード

PIVEC_GEN は、インテル C/C++ クラス・ライブラリ (SIMD 演算対応) の SIMD 整数ベクトル・クラス (IVEC) を使用して、Viterbi アルゴリズムを実行する。このコードは Pentium III プロセッサ向けであるため、Is16vec4 (4 つの要素の 16 ビット符号付き整数ベクトル・クラス) が使用される。このクラスは、64 ビット MMX テクノロジ・レジスタと、MMX テクノロジ命令および SSE を使用する。このコードは、アライメントの合わない読み出しを、2 回のアライメントの合った読み出し、2 回のシフト、および 1 回の OR 演算で置き換える。この例は、(HMM 状態の数に依存しない) 一般的な最適化を示す。アライメントの合ったロードによって、ストア・フォワードのペナルティが避けられる。

```
//Pentium(R) III processor - Ivec implementation - Uses the Streaming
//SIMD Extensions (Generic Optimization-avoids the store-forwarding penalty)
short hmm_pivec_gen::TheCode(unsigned int *obsVect, int obsLen, short *a,
                              short *a_1, short *a_2, short **bProb, short *pi,
                              int nStates, short *distBuffer)

{

    short      minDist;
    int i = 0;

    Is16vec4 *b4 = (Is16vec4 *) (bProb[obsVect[0]]);
    Is16vec4 *dist4 = (Is16vec4 *) distBuffer;
    Is16vec4 d4, d4next, d4_1, d4_2;
    Is16vec4 *a4 = (Is16vec4 *) a;
    Is16vec4 *a4_1 = (Is16vec4 *) a_1;
    Is16vec4 *a4_2 = (Is16vec4 *) a_2;
    Is16vec4 sum0, sum1, sum2, min0, min1;
    Is16vec4 *pi4 = (Is16vec4 *) pi;

    if (nStates == 0)
        return(0);

    int Pack4States = nStates >> 2;

    ///ERROR IF LESS THAN 4 STATES OR IF STATES ARE NOT A MULTIPLE OF 4///
```

```

//assert(Pack4States == 0 || (nStates % 4) > 1)
if (Pack4States == 0 || (nStates % 4) > 1)
    return(-1);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Step 1 of Viterbi Algorithm: Initialization
for( i = 0; i < Pack4States; i++)
    {dist4[i] = sat_add(pi4[i],b4[i]);}

//Loop for each observation or obsCount
for(i = 1; i < obsLen; i++)
{
    b4          = (Is16vec4 *)(bProb[obsVect[i]]);

    //Step 2 of Viterbi Algorithm: Recursion
    //FourStateLoop
    for(int j = 0; j < Pack4States; j++)
    {
        d4      = dist4[j];
        d4next = dist4[j+1];

        d4_1 = ((I64vec1)d4 >> 16);
        d4_1 |= ((I64vec1)d4next << 48);
        d4_2 = ((I64vec1)d4 >> 32);
        d4_2 |= ((I64vec1)d4next << 32);

        sum0 = sat_add(d4,  a4[j]);
        sum1 = sat_add(d4_1,a4_1[j]);
        sum2 = sat_add(d4_2,a4_2[j]);

        min0 = simd_min(sum2,simd_min(sum1,sum0));
        dist4[j] = sat_add(b4[j],min0);

    }

}

//Step 3 of Viterbi Algorithm: Termination
// Get 4 Minimum Distances
min0 = dist4[0];
for(i = 1; i < Pack4States; i++)
{
    min0 = simd_min(min0, dist4[i]);
}

```

```
}  
  
// Get Minimum Distance  
  
min1 = (Is16vec4)_m_pshufw(min0, 0x0E); //min0 = m3 m2 m1 m0  
min0 = simd_min(min0, min1); //min1 = xx xx m3 m2  
min1 = (Is16vec4)_m_pshufw(min0, 0x01); //min0 = xx xx m1 m0  
min0 = simd_min(min0, min1); //min1 = xx xx xx m1  
minDist = _m_pextrw(min0, 0); //min0 = xx xx xx min  
//minDist = min  
  
_m_empty();  
return(minDist);  
}
```

7 状態固有の最適化を使用したインテル® Pentium® III プロセッサ SIMD コード

PIVEC も、Is16vec4 クラスを使用して Viterbi アルゴリズムを実行するコードである。このコードは、ストア・フォワードのペナルティを回避し、さらに HMM 状態の数に固有の最適化を使用する。

```
//Pentium(R) III processor - Ivec implementation - Uses the Streaming
//SIMD Extensions (State-Specific Optimizations)
//For comparison with the WIVVEC implementation, this implementation is
//only optimized for states that are a multiple of 8

short hmm_pivec::TheCode(unsigned int *obsVect, int obsLen, short *a,
                          short *a_1, short *a_2, short **bProb, short *pi,
                          int nStates, short *distBuffer)
{
    short    minDist;
    int i = 0;

    if (nStates == 0)
        return(0);

    int Pack4States = nStates >> 2;

    ///ERROR IF LESS THAN 4 STATES OR IF STATES ARE NOT A MULTIPLE OF 4///
    //assert(Pack4States == 0 || (nStates % 4) > 1)
    if (Pack4States == 0 || (nStates % 4) > 1)
        return(-1);
    //////////////////////////////////////

    Is16vec4 *b4 = (Is16vec4 *) (bProb[obsVect[0]]);
    Is16vec4 *dist4 = (Is16vec4 *) distBuffer;
    Is16vec4 d4, d4next, d4_1, d4_2;
    Is16vec4 d4_1st, d4_1_1st, d4_2_1st;
    Is16vec4 d4_2nd, d4_1_2nd, d4_2_2nd;
    Is16vec4 *a4 = (Is16vec4 *) a;
    Is16vec4 *a4_1 = (Is16vec4 *) a_1;
    Is16vec4 *a4_2 = (Is16vec4 *) a_2;
    Is16vec4 sum0, sum1, sum2, min0, min1;
    Is16vec4 sum0_1st, sum1_1st, sum2_1st, min0_1st, min1_1st;
    Is16vec4 sum0_2nd, sum1_2nd, sum2_2nd, min0_2nd, min1_2nd;
```

```
Isl6vec4 *pi4 = (Isl6vec4 *) pi;

//For comparison with the WIVVEC implementation, this implementation is
//only optimized for states that are a multiple of 8
switch (Pack4States)
{
case 2: //8 States
    {
        //step1: Initialization of 8 states
        //Optimization: Do two stores instead of a loop
        d4_1st = sat_add(pi4[0], b4[0]);
        d4_2nd = sat_add(pi4[1], b4[1]);

        //step2: Loop
        //Optimization: Try to keep the data in the registers by:
        //Optimization1: Use only one loop instead of outer & inner loop.
        //Optimization2: There are only 8 states, thus combining the data
        //                  with the third load is unnecessary and has been
        //                  commented out.
        for( i=1; i < obsLen; i++ )
        {
            b4 = (Isl6vec4 *) (bProb[obsVect[i]]);

            //////////////// Do the 1st Packed 4 Distance ////////////////
            d4_1_1st = ((I64vec1)d4_1st >> 16);
            d4_1_1st |= ((I64vec1)d4_2nd << 48);
            d4_2_1st = ((I64vec1)d4_1st >> 32);
            d4_2_1st |= ((I64vec1)d4_2nd << 32);

            sum0_1st = sat_add(d4_1st, a4[0]);
            sum1_1st = sat_add(d4_1_1st, a4_1[0]);
            sum2_1st = sat_add(d4_2_1st, a4_2[0]);

            min0_1st = simd_min(sum2_1st, simd_min(sum1_1st, sum0_1st));
            //store result of first packed 4 distance
            d4_1st = sat_add(b4[0], min0_1st);

            //////////////// Do the 2nd Packed 4 Distance ////////////////
            d4_1_2nd = ((I64vec1)d4_2nd >> 16);
            //d4_1_2nd |= ((I64vec1)d4_3rd << 48); - not needed in this case
            d4_2_2nd = ((I64vec1)d4_2nd >> 32);
            //d4_2_2nd |= ((I64vec1)d4_3rd << 32); - not needed in this case
```

```
sum0_2nd = sat_add(d4_2nd, a4[1]);
sum1_2nd = sat_add(d4_1_2nd, a4_1[1]);
sum2_2nd = sat_add(d4_2_2nd, a4_2[1]);

min0_2nd = simd_min(sum2_2nd, simd_min(sum1_2nd, sum0_2nd));
//store result of second packed 4 distance
d4_2nd = sat_add(b4[1], min0_2nd);
}

//optimization: No need to use a loop
//Step 3 of Viterbi Algorithm: Termination
// Get 4 Minimum Distances
min0 = simd_min(d4_1st, d4_2nd);
break;
}

case 4: //16 States
{

Isl6vec4 sum0_3rd, sum1_3rd, sum2_3rd, min0_3rd, min1_3rd;
Isl6vec4 sum0_4th, sum1_4th, sum2_4th, min0_4th, min1_4th;
Isl6vec4 d4_3rd, d4_1_3rd, d4_2_3rd;
Isl6vec4 d4_4th, d4_1_4th, d4_2_4th;

//step1: Initialization of 16 states
//optimization: No need to use a loop
d4_1st = sat_add(pi4[0], b4[0]);
d4_2nd = sat_add(pi4[1], b4[1]);
d4_3rd = sat_add(pi4[2], b4[2]);
d4_4th = sat_add(pi4[3], b4[3]);

//step2: Loop
//Optimization1: Use only one loop instead of outer & inner loop.
//Optimization2: There are only 16 states, thus combining the data
//                with the fifth load is unnecessary and has been
//                commented out.
for( i=1; i < obsLen; i++ )
{
    b4 = (Isl6vec4 *) (bProb[obsVect[i]]);
```

```
////////// Do the 1st Packed 4 Distance //////////
d4_1_1st = ((I64vec1)d4_1st >> 16);
d4_1_1st |= ((I64vec1)d4_2nd << 48);
d4_2_1st = ((I64vec1)d4_1st >> 32);
d4_2_1st |= ((I64vec1)d4_2nd << 32);

sum0_1st = sat_add(d4_1st, a4[0]);
sum1_1st = sat_add(d4_1_1st, a4_1[0]);
sum2_1st = sat_add(d4_2_1st, a4_2[0]);

min0_1st = simd_min(sum2_1st, simd_min(sum1_1st, sum0_1st));
//store result of first packed 4 distance
d4_1st = sat_add(b4[0], min0_1st);

////////// Do the 2nd Packed 4 Distance //////////
d4_1_2nd = ((I64vec1)d4_2nd >> 16);
d4_1_2nd |= ((I64vec1)d4_3rd << 48);
d4_2_2nd = ((I64vec1)d4_2nd >> 32);
d4_2_2nd |= ((I64vec1)d4_3rd << 32);

sum0_2nd = sat_add(d4_2nd, a4[1]);
sum1_2nd = sat_add(d4_1_2nd, a4_1[1]);
sum2_2nd = sat_add(d4_2_2nd, a4_2[1]);

min0_2nd = simd_min(sum2_2nd, simd_min(sum1_2nd, sum0_2nd));
//store result of second packed 4 distance
d4_2nd = sat_add(b4[1], min0_2nd);

////////// Do the 3rd Packed 4 Distance //////////
d4_1_3rd = ((I64vec1)d4_3rd >> 16);
d4_1_3rd |= ((I64vec1)d4_4th << 48);
d4_2_3rd = ((I64vec1)d4_3rd >> 32);
d4_2_3rd |= ((I64vec1)d4_4th << 32);

sum0_3rd = sat_add(d4_3rd, a4[2]);
sum1_3rd = sat_add(d4_1_3rd, a4_1[2]);
sum2_3rd = sat_add(d4_2_3rd, a4_2[2]);

min0_3rd = simd_min(sum2_3rd, simd_min(sum1_3rd, sum0_3rd));
//store result of first packed 4 distance
d4_3rd = sat_add(b4[2], min0_3rd);

////////// Do the 4th Packed 4 Distance //////////
```

```

d4_1_4th = ((I64vec1)d4_4th    >> 16);
//d4_1_4th |= ((I64vec1)d4_5th << 48); - not needed in this case
d4_2_4th = ((I64vec1)d4_4th    >> 32);
//d4_2_4th |= ((I64vec1)d4_5th << 32); - not needed in this case

sum0_4th = sat_add(d4_4th,  a4[3]);
sum1_4th = sat_add(d4_1_4th,a4_1[3]);
sum2_4th = sat_add(d4_2_4th,a4_2[3]);

min0_4th = simd_min(sum2_4th,simd_min(sum1_4th,sum0_4th));
//store result of second packed 4 distance
d4_4th = sat_add(b4[3],min0_4th);

}

//Step 3 of Viterbi Algorithm: Termination
// Get 4 Minimum Distances
//optimization: No need to use a loop
min0 = simd_min(d4_1st, d4_2nd);
min0 = simd_min(min0, d4_3rd);
min0 = simd_min(min0, d4_4th);
break;
}
default:
{

//Step 1 of Viterbi Algorithm: Initialization
for( i = 0; i < Pack4States; i++)
    {dist4[i] = sat_add(pi4[i],b4[i]);}

//Loop for each observation or obsCount
for(i = 1; i < obsLen; i++)
{
    b4          = (I16vec4 *)(bProb[obsVect[i]]);

//Step 2 of Viterbi Algorithm: Recursion
//FourStateLoop
for(int j = 0; j < Pack4States; j++)
{
    d4          = dist4[j];
    d4next      = dist4[j+1];

```

```

    d4_1 = ((I64vec1)d4 >> 16);
    d4_1 |= ((I64vec1)d4next << 48);
    d4_2 = ((I64vec1)d4 >> 32);
    d4_2 |= ((I64vec1)d4next << 32);

    sum0 = sat_add(d4, a4[j]);
    sum1 = sat_add(d4_1, a4_1[j]);
    sum2 = sat_add(d4_2, a4_2[j]);

    min0 = simd_min(sum2, simd_min(sum1, sum0));
    dist4[j] = sat_add(b4[j], min0);

}

}
//Step 3 of Viterbi Algorithm: Termination
// Get 4 Minimum Distances
min0 = dist4[0];
for(i = 1; i < Pack4States; i++)
{
    min0 = simd_min(min0, dist4[i]);
}
break;
}
}
// Get Minimum Distance

min1 = (Is16vec4)_m_pshufw(min0, 0x0E); //min0 = m3 m2 m1 m0
min0 = simd_min(min0, min1); //min1 = xx xx m3 m2
min1 = (Is16vec4)_m_pshufw(min0, 0x01); //min0 = xx xx m1 m0
min0 = simd_min(min0, min1); //min1 = xx xx xx m1
minDist = _m_pextrw(min0, 0); //min0 = xx xx xx min
//minDist = min

_m_empty();
return(minDist);
}

```

8 一般的な最適化を使用したインテル® Pentium® 4 プロセッサ SIMD コード

WIVVEC_GEN は、インテル C/C++ クラス・ライブラリ (SIMD 演算対応) と `Is16vec8` クラス (8 つの要素の 16 ビット符号付き整数ベクトル・クラス) を使用する。このコードは、Pentium 4 プロセッサ向けに最適化されている。`Is16vec8` クラスは、XMM レジスタ (128 ビット・レジスタ) と SSE2 を使用する。このコードは、PIVVEC_GEN コードと同じ一般的な最適化手法を使用する。つまり、アライメントの合わないロードを、2 回のアライメントの合ったロード、2 回のシフト、および 1 回の OR 演算で置き換えて、ストア・フォワードのストールを回避する。

```
//Streaming SIMD Extensions 2 (SSE2) - Ivec implementation (Generic
Optimization)
short hmm_wivec_gen::TheCode(unsigned int *obsVect, int obsLen, short *a,
                             short *a_1, short *a_2, short **bProb, short *pi,
                             int nStates, short *distBuffer1)
{

    short    minDist;
    int i = 0;

    Is16vec8 *b8 = (Is16vec8 *) (bProb[obsVect[0]]);
    Is16vec8 *dist8 = (Is16vec8 *) distBuffer;
    Is16vec8 d8, d8next, d8_1, d8_2;

    //transition probability:
    // a8 - the prob. of moving back to the same state
    // a8_1 - the prob. of moving to the next state (to state j-1)
    // a8_2 - the prob. of moving to state j-2
    Is16vec8 *a8 = (Is16vec8 *) a;
    Is16vec8 *a8_1 = (Is16vec8 *) a_1;
    Is16vec8 *a8_2 = (Is16vec8 *) a_2;

    Is16vec8 sum0, sum1, sum2, min0, min1;
    Is16vec8 *pi8 = (Is16vec8 *) pi;

    if (nStates == 0)
        return(0);

    int Pack8States = nStates >> 3;
```

```

/////ERROR IF LESS THAN 8 STATES OR IF STATES ARE NOT A MULTIPLE OF 8/////
//assert(Pack8States == 0 || (nStates % 8) > 1)
if (Pack8States == 0 || (nStates % 8) > 1)
    return(-1);
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Step 1 of Viterbi Algorithm: Initialization
for( i = 0; i < Pack8States; i++)
    {dist8[i] = sat_add(pi8[i],b8[i]);}

//Step 2 of Viterbi Algorithm: Recursion
//Loop for each observation or obsCount
for(i = 1; i < obsLen; i++)
{
    b8          = (Is16vec8 *) (bProb[obsVect[i]]);

    //EightStateLoop
    for(int j = 0; j < Pack8States; j++)
    {
        d8 = dist8[j];
        d8next = dist8[j+1];

        //Remove the unaligned loads by combining two aligned loads together
        d8_1 = (Is16vec8)_mm_srli_si128((I128vec1)d8, 2);
        d8_1 |= (Is16vec8)_mm_slli_si128((I128vec1)d8next,14);
        d8_2 = (Is16vec8)_mm_srli_si128((I128vec1)d8 ,4);
        d8_2 |= (Is16vec8)_mm_slli_si128((I128vec1)d8next,12);

        sum0 = sat_add(d8, a8[j]);
        sum1 = sat_add(d8_1,a8_1[j]);
        sum2 = sat_add(d8_2,a8_2[j]);

        min0 = simd_min(sum2,simd_min(sum1,sum0));
        dist8[j] = sat_add(b8[j],min0);
    }
}

//Step 3 of Viterbi Algorithm: Termination
// Get 8 Minimum Distances

```

```
min0 = dist8[0];
for(i = 1; i < Pack8States; i++)
{
    min0 = simd_min(min0, dist8[i]);
}

// Get Minimum Distance
//min0 = m7 m6 m5 m4 m3 m2 m1 m0
//min1 = xx xx xx xx m7 m6 m5 m4
//min0 = xx xx xx xx m3 m2 m1 m0
min1 = (Is16vec8) _mm_shuffle_epi32(min0, 0xE);
min0 = simd_min(min0, min1);
//min0 = xx xx xx xx m3 m2 m1 m0
//min1 = xx xx xx xx xx xx m3 m2
//min0 = xx xx xx xx xx xx m1 m0
min1 = (Is16vec8)_mm_shufflelo_epi16(min0, 0xE);
min0 = simd_min(min0, min1);
//min0 = xx xx xx xx xx xx m1 m0
//min1 = xx xx xx xx xx xx m1
//min0 = xx xx xx xx xx xx min
min1 = (Is16vec8)_mm_shufflelo_epi16(min0, 0x1);
min0 = simd_min(min0, min1);
//minDist = min
minDist = _mm_extract_epi16(min0, 0);

return(minDist);
}
```

9 状態固有の最適化を使用したインテル® Pentium® 4 プロセッサ SIMD コード

WIVVEC は `Is16vec8` ベクトル・クラスを使用し、ストア・フォワードのペナルティを避けるように最適化され、さらに HMM 状態の数に固有の最適化手法も使用する。使用する最適化手法については、コード内のコメントを参照のこと。これは、このアプリケーション・ノートで説明する 5 種類のコードのうち、最も効率的なコードである。この Pentium 4 プロセッサ向け IVEC コードを使用すれば、最適な Pentium III プロセッサ向け IVEC コードと比べて、処理速度が大きくアップする。

```
//Streaming SIMD Extensions 2 (SSE2) - Ivec implementation (State-Specific
//Optimizations)
short hmm_wivec::TheCode(unsigned int *obsVect, int obsLen, short *a,
                          short *a_1, short *a_2, short **bProb, short *pi,
                          int nStates, short *distBuffer1)
{

    short    minDist;
    int i = 0;

    Is16vec8 *b8 = (Is16vec8 *) (bProb[obsVect[0]]);
    Is16vec8 *dist8 = (Is16vec8 *) distBuffer;
    Is16vec8 d8, d8next, d8_1, d8_2;

    Is16vec8 d8_1st, d8_1_1st, d8_2_1st;
    Is16vec8 sum0_1st, sum1_1st, sum2_1st, min0_1st, min1_1st;
    Is16vec8 d8_2nd, d8_1_2nd, d8_2_2nd;
    Is16vec8 sum0_2nd, sum1_2nd, sum2_2nd, min0_2nd, min1_2nd;
    Is16vec8 d8_3rd, d8_1_3rd, d8_2_3rd;
    Is16vec8 sum0_3rd, sum1_3rd, sum2_3rd, min0_3rd, min1_3rd;

    //transition probability:
    // a8 - the prob. of moving back to the same state
    // a8_1 - the prob. of moving to the next state (to state j-1)
    // a8_2 - the prob. of moving to state j-2
    Is16vec8 *a8 = (Is16vec8 *) a;
    Is16vec8 *a8_1 = (Is16vec8 *) a_1;
    Is16vec8 *a8_2 = (Is16vec8 *) a_2;

    Is16vec8 sum0, sum1, sum2, min0, min1;
```

```

Isl6vec8 *pi8 = (Isl6vec8 *) pi;

if (nStates == 0)
    return(0);

int Pack8States = nStates >> 3;

////ERROR IF LESS THAN 8 STATES OR IF STATES ARE NOT A MULTIPLE OF 8////
//assert(Pack8States == 0 || (nStates % 8) > 1)
if (Pack8States == 0 || (nStates % 8) > 1)
    return(-1);
////////////////////////////////////
switch (Pack8States)
{
case 1:
    {
        //step1: Initialization of 8 states
        //Optimization - removed loop
        d8_1st = sat_add(pi8[0], b8[0]);

        //step2: Loop
        //Optimization Goal: Keep all data in registers by:
        // Optimization 1) Use only one loop instead of inner & outer loop
        // Optimization 2) No need to combine data with 2nd load.
        for( i=1; i < obsLen; i++ )
        {
            b8 = (Isl6vec8 *) (bProb[obsVect[i]]);

            //////////////// Do the 1st Packed 8 Distance ////////////////
            d8_1_1st = (Isl6vec8)_mm_srli_si128((I128vec1)d8_1st, 2);
            //d8_1_1st |= (Isl6vec8)_mm_slli_si128((I128vec1)d8_2nd,14);
            d8_2_1st = (Isl6vec8)_mm_srli_si128((I128vec1)d8_1st ,4);
            //d8_2_1st |= (Isl6vec8)_mm_slli_si128((I128vec1)d8_2nd,12);

            sum0_1st = sat_add(d8_1st, a8[0]);
            sum1_1st = sat_add(d8_1_1st,a8_1[0]);
            sum2_1st = sat_add(d8_2_1st,a8_2[0]);

            min0_1st = simd_min(sum2_1st,simd_min(sum1_1st,sum0_1st));
        }
    }
}

```

```

        //store result of first packed 8 distance
        d8_1st = sat_add(b8[0],min0_1st);

    }

    //Step 3 of Viterbi Algorithm: Termination
    // Get 8 Minimum Distances
    // Optimization: No loop needed
    min0 = d8_1st;
    break;
}

case 2:
{
    //step1: Initialization of 8 states
    //optimization: removed loop
    d8_1st = sat_add(pi8[0], b8[0]);
    d8_2nd = sat_add(pi8[1], b8[1]);

    //step2: Loop
    //Optimization Goal: Keep all data in registers by:
    // Optimization 1) Use only one loop instead of inner & outer loop
    // Optimization 2) No need to combine data with 3rd load.

    for( i=1; i < obsLen; i++ )
    {
        b8 = (Is16vec8 *) (bProb[obsVect[i]]);

        //////////////// Do the 1st Packed 8 Distance ////////////////
        d8_1_1st = (Is16vec8)_mm_srli_si128((I128vec1)d8_1st, 2);
        d8_1_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,14);
        d8_2_1st = (Is16vec8)_mm_srli_si128((I128vec1)d8_1st ,4);
        d8_2_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,12);

        sum0_1st = sat_add(d8_1st, a8[0]);
        sum1_1st = sat_add(d8_1_1st,a8_1[0]);
        sum2_1st = sat_add(d8_2_1st,a8_2[0]);

        min0_1st = simd_min(sum2_1st,simd_min(sum1_1st,sum0_1st));
        //store result of first packed 8 distance

```

```

d8_1st = sat_add(b8[0],min0_1st);

////////// Do the 2nd Packed 8 Distance //////////
d8_1_2nd = (Is16vec8)_mm_srli_si128((I128vec1)d8_2nd, 2);
//d8_1_2nd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_3rd,14);
d8_2_2nd = (Is16vec8)_mm_srli_si128((I128vec1)d8_2nd ,4);
//d8_2_2nd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_3rd,12);

sum0_2nd = sat_add(d8_2nd, a8[1]);
sum1_2nd = sat_add(d8_1_2nd,a8_1[1]);
sum2_2nd = sat_add(d8_2_2nd,a8_2[1]);

min0_2nd = simd_min(sum2_2nd,simd_min(sum1_2nd,sum0_2nd));
//store result of first packed 8 distance
d8_2nd = sat_add(b8[1],min0_2nd);
}

//Step 3 of Viterbi Algorithm: Termination
// Get 8 Minimum Distances
min0 = simd_min(d8_1st, d8_2nd);
break;
}

case 3:
{
//step1: Initialization of 8 states
//optimization: removed loop
d8_1st = sat_add(pi8[0], b8[0]);
d8_2nd = sat_add(pi8[1], b8[1]);
d8_3rd = sat_add(pi8[2], b8[2]);

//step2: Loop
//Optimization Goal: Attempt to keep most data in registers by:
// Optimization 1) Use only one loop instead of inner & outer loop
// Optimization 2) No need to combine data with 4th load.
for( i=1; i < obsLen; i++ )
{
b8 = (Is16vec8 *)(bProb[obsVect[i]]);

////////// Do the 1st Packed 8 Distance //////////
d8_1_1st = (Is16vec8)_mm_srli_si128((I128vec1)d8_1st, 2);
d8_1_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,14);

```

```
d8_2_1st = (Is16vec8)_mm_srli_si128((I128vec1)d8_1st ,4);
d8_2_1st |= (Is16vec8)_mm_slli_si128((I128vec1)d8_2nd,12);

sum0_1st = sat_add(d8_1st, a8[0]);
sum1_1st = sat_add(d8_1_1st,a8_1[0]);
sum2_1st = sat_add(d8_2_1st,a8_2[0]);

min0_1st = simd_min(sum2_1st,simd_min(sum1_1st,sum0_1st));
//store result of first packed 8 distance
d8_1st = sat_add(b8[0],min0_1st);

////////// Do the 2nd Packed 8 Distance //////////
d8_1_2nd = (Is16vec8)_mm_srli_si128((I128vec1)d8_2nd, 2);
d8_1_2nd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_3rd,14);
d8_2_2nd = (Is16vec8)_mm_srli_si128((I128vec1)d8_2nd ,4);
d8_2_2nd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_3rd,12);

sum0_2nd = sat_add(d8_2nd, a8[1]);
sum1_2nd = sat_add(d8_1_2nd,a8_1[1]);
sum2_2nd = sat_add(d8_2_2nd,a8_2[1]);

min0_2nd = simd_min(sum2_2nd,simd_min(sum1_2nd,sum0_2nd));
//store result of first packed 8 distance
d8_2nd = sat_add(b8[1],min0_2nd);

////////// Do the 3rd Packed 8 Distance //////////
d8_1_3rd = (Is16vec8)_mm_srli_si128((I128vec1)d8_3rd, 2);
//d8_1_3rd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_4th,14);
d8_2_3rd = (Is16vec8)_mm_srli_si128((I128vec1)d8_3rd ,4);
//d8_2_3rd |= (Is16vec8)_mm_slli_si128((I128vec1)d8_4th,12);

sum0_3rd = sat_add(d8_3rd, a8[2]);
sum1_3rd = sat_add(d8_1_3rd,a8_1[2]);
sum2_3rd = sat_add(d8_2_3rd,a8_2[2]);

min0_3rd = simd_min(sum2_3rd,simd_min(sum1_3rd,sum0_3rd));
//store result of first packed 8 distance
d8_3rd = sat_add(b8[2],min0_3rd);

}

//Step 3 of Viterbi Algorithm: Termination
```

```

    // Get 8 Minimum Distances
    //optimization: removed loop
    min0 = simd_min(d8_1st, d8_2nd);
    min0 = simd_min(min0, d8_3rd);

    break;
}

default:
{
    //Step 1 of Viterbi Algorithm: Initialization
    for( i = 0; i < Pack8States; i++)
        {dist8[i] = sat_add(pi8[i],b8[i]);}

    //Step 2 of Viterbi Algorithm: Recursion
    //Loop for each observation or obsCount
    for(i = 1; i < obsLen; i++)
    {
        b8          = (Is16vec8 *)(bProb[obsVect[i]]);

        //EightStateLoop
        for(int j = 0; j < Pack8States; j++)
        {
            d8 = dist8[j];
            d8next = dist8[j+1];

            //Remove unaligned loads by combining 2 aligned loads together
            d8_1 = (Is16vec8)_mm_srli_si128((I128vec1)d8, 2);
            d8_1 |= (Is16vec8)_mm_slli_si128((I128vec1)d8next,14);
            d8_2 = (Is16vec8)_mm_srli_si128((I128vec1)d8 ,4);
            d8_2 |= (Is16vec8)_mm_slli_si128((I128vec1)d8next,12);

            sum0 = sat_add(d8, a8[j]);
            sum1 = sat_add(d8_1,a8_1[j]);
            sum2 = sat_add(d8_2,a8_2[j]);

            min0 = simd_min(sum2,simd_min(sum1,sum0));
            dist8[j] = sat_add(b8[j],min0);

        }
    }
}

```

```
//Step 3 of Viterbi Algorithm: Termination
// Get 8 Minimum Distances
min0 = dist8[0];
for(i = 1; i < Pack8States; i++)
{
    min0 = simd_min(min0, dist8[i]);
}
break;
}
}

// Get Minimum Distance
//min0 = m7 m6 m5 m4 m3 m2 m1 m0
//min1 = xx xx xx xx m7 m6 m5 m4
//min0 = xx xx xx xx m3 m2 m1 m0
min1 = (Is16vec8) _mm_shuffle_epi32(min0, 0xE);
min0 = simd_min(min0, min1);
//min0 = xx xx xx xx m3 m2 m1 m0
//min1 = xx xx xx xx xx xx m3 m2
//min0 = xx xx xx xx xx xx m1 m0
min1 = (Is16vec8)_mm_shufflelo_epi16(min0, 0xE);
min0 = simd_min(min0, min1);
//min0 = xx xx xx xx xx xx m1 m0
//min1 = xx xx xx xx xx xx xx m1
//min0 = xx xx xx xx xx xx xx min
min1 = (Is16vec8)_mm_shufflelo_epi16(min0, 0x1);
min0 = simd_min(min0, min1);
//minDist = min
minDist = _mm_extract_epi16(min0, 0);

return(minDist);
}
```

付録 A - パフォーマンス・データ

パフォーマンス・データの改訂履歴

改訂番号	改訂履歴	改訂時期
2.0	インテル® Pentium® 4 プロセッサ 1.20 GHz を使用した パフォーマンス・データによる更新	2000 年 7 月
1.0	初版	1999 年 9 月

表 1: HMM コードのパフォーマンス・データ

パフォーマンス・データ(マイクロ秒)		
コードの種類	Pentium III プロセッサ (733 MHz)	Pentium 4 プロセッサ (1.20 GHz)
SSE IVEC(一般的な最適化)		
8 個の状態	1.09	0.91
16 個の状態	2.80	2.30
24 個の状態	6.59	4.71
32 個の状態	10.31	8.02
SSE IVEC(固有の最適化)		
8 個の状態	0.63	0.52
16 個の状態	2.41	1.69
24 個の状態	5.99	4.45
32 個の状態	9.99	7.74
SSE2 IVEC(一般的な最適化)		
8 個の状態	-	0.68
16 個の状態	-	1.59
24 個の状態	-	2.61
32 個の状態	-	4.40
SSE2 IVEC(固有の最適化)		
8 個の状態	-	0.41
16 個の状態	-	1.16
24 個の状態	-	2.07
32 個の状態	-	4.41

表 2: 表 1 のパフォーマンス・データに基づく処理速度の比較

コードおよびプラットフォーム	処理速度
Pentium 4 プロセッサ上の最適な SSE IVEC と Pentium III プロセッサ上の最適な SSE IVEC	1.21 - 1.42 倍
Pentium 4 プロセッサ(最適な SSE2 IVEC と最適な SSE IVEC)	1.26 - 2.15 倍
Pentium 4 プロセッサ上の最適な SSE2 IVEC と Pentium III プロセッサ上の最適な SSE IVEC	1.53 - 2.80 倍

パフォーマンスは、Pentium III プロセッサ 733 MHz と Pentium 4 プロセッサ 1.20 GHz を使用して測定された。テスト・システムについての詳細は、A-3 ページの「テスト・システムの構成」の表 3 と表 4 を参照のこと。上記の表 1 と表 2 は、ウォーム・キャッシュ・パフォーマンスの測定値である。

このアプリケーション・ノートでは、Viterbi アルゴリズムの 4 種類のコード(および最適化されていない機能コード)について説明した。IVEC コードは、インテル C/C++ クラス・ライブラリ(SIMD 演算対応)の SIMD 整数ベクトル・クラス(IVEC)を使用する。PIVEC コードと PIVEC_GEN コードは、SSE を使用する。WIVEC コードと WIVEC_GEN コードは、SSE2 を使用する。この文書で説明したように、これらの 4 種類のコードはアライメントの合っていないロードの除去によって、ストア・フォワードのペナルティを回避する。PIVEC_GEN コードと WIVEC_GEN コードは、ストア・フォワードのペナルティを避ける最適化手法のみを使用する。PIVEC コードと WIVEC コードは、HMM 状態の数に依存する最適化手法も使用する。HMM 状態の数に合わせて最適化を行うコードは、いずれのプロセッサでも、最も良いパフォーマンスを示している。

表 2 は、表 1 の結果をまとめたものである。表 2 では、Pentium III プロセッサ上の最適な SSE IVEC コードに対する、Pentium 4 プロセッサ上の同じコードの処理速度は、1.21 ~ 1.42 倍にアップしている。Pentium 4 プロセッサ上では、SSE に対する SSE2 の処理速度は、HMM 状態の数に応じて、1.26 ~ 2.15 倍にアップする。このパフォーマンス向上は、SIMD 幅の拡張による。SSE2 を使用すれば、より少ない数の命令で、1 ループ当たり 2 倍の数の状態を処理できる。Pentium III プロセッサ上の最適な SSE IVEC コードに対する、Pentium 4 プロセッサ上の最適な SSE2 IVEC コードの処理速度は、1.53 ~ 2.80 倍に向上する。このパフォーマンス向上は、マイクロアーキテクチャの高速化と SSE2 による高速化の両方の影響を受けている。

結論として、SSE2 は、Viterbi デコーディング・アルゴリズムのパフォーマンスを大きく向上させる。SIMD 幅の拡張が、この高速化の主要な要因である。その他の最適化手法には、`pminsw` 命令を使用する、ストア・フォワードのペナルティを避ける、ベクトルのパディングによって内側ループの分岐を除去する、HMM 状態の数に合わせた最適化を行うなどがある。

テスト・システムの構成

表 3: Pentium III プロセッサのシステム構成

プロセッサ	Pentium III プロセッサ 733 MHz
システム	インテル® Desktop Board VC820
BIOS のバージョン	VC82010A.86A.0028.P10
2次キャッシュ	256 KB
メモリ・サイズ	128 MB RDRAM PC800-45
Ultra ATA ストレージ・ドライバ	製品版候補 6.00.012
ハードディスク	IBM DJNA-371800 ATA-66
ビデオ・コントローラ/ バス	Creative Labs 3D Blaster [†] Annihilator [†] Pro AGP nVidia GeForce256 [†] DDR –32MB
ビデオ・ドライバの リビジョン	NVidia リファレンス・ドライバ 5.22
オペレーティング・ システム	Windows [†] 2000 ビルド 2195

表 4: Pentium 4 プロセッサのシステム構成

プロセッサ	Pentium 4 プロセッサ 1.20 GHz
システム	インテル Desktop Board D850GB
BIOS のバージョン	GB85010A.86A.0014.D.0007201756
2次キャッシュ	256 KB
メモリ・サイズ	128 MB RDRAM PC800-45
Ultra ATA ストレージ・ ドライバ	製品版候補 6.00.012
ハードディスク	IBM DJNA-371800 ATA-66
ビデオ・コントローラ/ バス	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
ビデオ・ドライバの リビジョン	NVidia リファレンス・ドライバ 5.22
オペレーティング・ システム	Windows 2000 ビルド 2195