Cell プロセッサを用いた編集距離計算の高速化

桜庭 俊† 吉田 悠一‡

† 東京大学大学院新領域創成科学研究科
‡ 京都大学大学院情報学研究科

Cell Challenge 2009 [4] の規定課題において、Cell Broadband Engine(以下 Cell) を用いて文字列の編集距離を計算するプログラムを作成するという課題が出題された。本稿では我々が規定課題において考案した最適化技法について報告する。基本となるアルゴリズムはビット並列化法と呼ばれるものであり、実行委員会によるサンプルプログラムと比べて約 315 倍の高速化を達成した。

1 はじめに

まず Cell Challenge 2009 の規定課題について述べる。二つの文字列 A と B の編集距離 $d(A,B)$(又は Levenshtein 距離 [3]) とは、文字の挿入、削除、置換の三操作を用いて A を B を変形するのに必要な最短手数のことである。Cell Challenge 2009 の規定課題では Cell を用いて編集距離を出来るだけ高速に計算するプログラムを作成するという課題が出題された。但し二つの入力文字列 A,B に対して、$|A|,|B|$ は 128 で割り切れ、$\max(|A|,|B|) \leq 2^{20}-128$, $|A||B| \leq 2^{34}$ という制約が課されている。

Cell には PPE と呼ばれる汎用プロセッサが一個と SPE と呼ばれるベクトル演算を行えるプロセッサが七個搭載されている。我々は、ビット並列化法、SPE を 7 個利用した並列化、メモリ転送の効率化、自明な入力に対する枝刈りなどの技法を用いて高速化を行った。その結果、一般の入力に対して実行委員会によるサンプルプログラムに対して約 315 倍の高速化を達成した。また特別な入力に対してはさらなる高速化を達成した。本稿では我々が考案した最適化技法や開発過程について報告する。

構成 2 節で編集距離を計算する基本的な手法である動的計画法と、レジスタ長が大きい時にこの動的計画法を高速に計算するビット並列化法について説明する。次に 3 節で Cell に特化した高速化手法を述べる。4 節ではその他の最適化手法として自明に編集距離が計算出来るような入力への対処法を述べる。本規定課題のように例外処理が多数発生するようなプログラムを作成する際にはプログラムのテストが欠かせない。5 節では今回行ったテストの手法について述べる。6 節で本選課題における我々のプログラムの結果を述べる。7 節ではまとめと今後の課題を述べる。

2 動的計画法とビット並列化法

本節では編集距離を計算する際の基本アルゴリズムである動的計画法とそれをビット並列化したものについて述べる。

2.1 動的計画法

二つの文字列 A,B 間の編集距離 $d(A,B)$ は次のような動的計画法により計算出来ることが知られている。最初に用語の定義を行う。$A[i](1 \leq i \leq |A|)$ を A の i 番目の文字とする。$A[i \cdots j]$ を A の i 番目から j 番目までを取り出して出来る文字列とする。

二次元の表 $T[i,j](0 \leq i \leq |A|, 0 \leq j \leq |B|)$ を用意する。ここで $T[i,j] = d(A[1 \cdots i], B[1 \cdots j])$ となることを意図している。初期化作業として $T[i,0] = i, T[0,j] = j$ とする。次に $1 \leq i \leq |A|, 1 \leq j \leq |B|$ について

$$T[i,j] = \begin{cases} T[i-1,j-1] & A[i] = B[i] \text{ の時} \\ \min(\begin{array}{l} T[i-1,j-1], \\ T[i-1,j], \\ T[i,j-1]) \end{array} & \text{それ以外} \end{cases}$$

として、表 $T[i,j]$ を埋めていく。最終的に $d(A,B) = T[|A|,|B|]$ として出力する。この方法により $O(|A||B|)$ 時間で編集距離を計算出来る。

2.2 ビット並列化法

SPE のレジスタ長が大きいことを利用してビット並列化 [1, 2] と呼ばれる手法を採用した。ビット並列化法では動的計画法で用いる表 T の複数個の要素を同時に計算する。ここで表 T において隣り合った要素の値は高々1 しか変わらないことを利用する。まず、各 $1 \leq j \leq |B|$ につい

て、ベクトル $\mathrm{PM}_j, \mathrm{D0}_j, \mathrm{HP}_j, \mathrm{HN}_j, \mathrm{VP}_j, \mathrm{VN}_j \in \{0,1\}^{|A|}$ を以下の様に定義する。

$$
\begin{aligned}
\mathrm{PM}_j[i] &= 1 \text{ iff } A[i] = B[j] \\
\mathrm{D0}_j[i] &= 1 \text{ iff } T[i,j] = T[i-1,j-1] \\
\mathrm{VP}_j[i] &= 1 \text{ iff } T[i,j] - T[i-1,j] = 1 \\
\mathrm{VN}_j[i] &= 1 \text{ iff } T[i,j] - T[i-1,j] = -1 \\
\mathrm{HP}_j[i] &= 1 \text{ iff } T[i,j] - T[i,j-1] = 1 \\
\mathrm{HN}_j[i] &= 1 \text{ iff } T[i,j] - T[i,j-1] = -1
\end{aligned}
$$

ここから $T[|A|,|B|] = |A| + \sum_{j=1}^{|B|} \mathrm{HP}_j[i] - \sum_{j=1} \mathrm{HN}_j[i]$ であるので、編集距離 $d(A,B)$ が計算出来る。j に対する各ベクトルの計算は $\mathrm{PM}_j, \mathrm{VP}_{j-1}, \mathrm{VN}_{j-1}$ を用いて行うことが出来る。以下の全ての演算はビット単位で行うものとする。

$$
\begin{aligned}
\mathrm{D0}_j &= (((\mathrm{PM}_j \& \mathrm{VP}_{j-1}) + \mathrm{VP}_{j-1}) \& \mathrm{VP}_{j-1}) | \\
&\quad \mathrm{PM}_j | \mathrm{VN}_{j-1} \qquad (1) \\
\mathrm{HP}_j &= \mathrm{VN}_{j-1} | \ (\mathrm{D0}_j | \mathrm{VP}_{j-1}) \\
\mathrm{HN}_j &= \mathrm{D0}_j \& \mathrm{VP}_{j-1} \\
\mathrm{VP}_j &= (\mathrm{HN}_j << 1) | \ (\mathrm{D0}_j | (\mathrm{HP}_j << 1) | 1)) \\
\mathrm{VN}_j &= \mathrm{D0}_j \& ((\mathrm{HP}_j << 1) | 1)
\end{aligned}
$$

編集距離に対する動的計画法は依存関係が多いので本来的には並列化しにくいが、ビット並列化ではその依存関係を加算の繰り上がりという形で表現している。加算はハードウェアで実行されるのでこれはソフトウェア上では既に依存関係では無い。従って、加算の繰り上がりとシフトの繰り上がりを除いた部分は並列化が可能である。

w ビットの値どうしの演算が $O(1)$ 時間で可能な場合、ビット並列化を用いることで編集距離の計算を $O(\lceil |A|/w \rceil |B|)$ 時間で行うことが出来る。SPE のレジスタ長は 128 ビットであるので $w = 128$ となり、単純な動的計画法と比べて 128 倍の高速化が期待できる。

3 Cellのアーキテクチャを利用した最適化

本節では Cell のアーキテクチャに特化した並列化法について述べる。

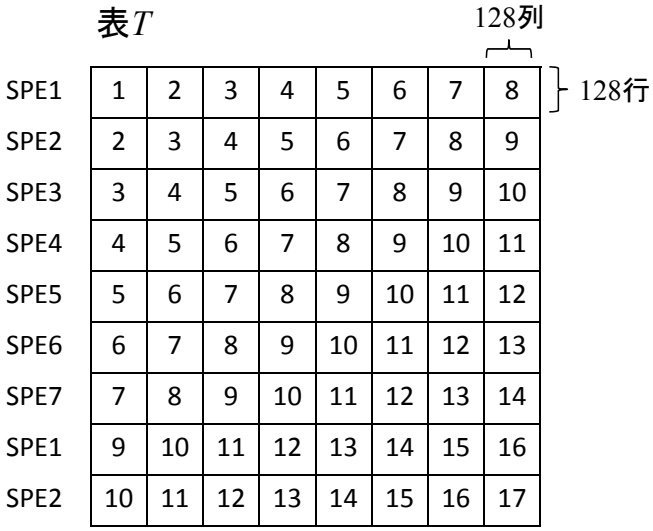

図 1: SPE7 個を用いた並列化の概要: 各 SPE が表 T の 128 行を担当する。表 T 中の値はそのブロックの計算が終わる時間である。一ブロック処理するのに 1 の時間がかかるものとしている。

3.1 SPE を複数個用いた並列化

Cell プロセッサは汎用処理を行う 1 個の PPE と、計算処理に特化した 7 個の SPE が存在する。ここでは Cell の 7 個の SPE を用いてビット並列化法をさらに並列化する方法を説明する。基本的な考え方を図 1 に示す。最初に表 T を 128×128 のブロックに分割する。次に一つの SPE に 128 行を担当させ、T の中の値を計算させる。各 SPE は 128 列ごとに直後の 128 行を担当している SPE に D0 の桁上がり、HP, HN のシフトあふれを渡す。こうすることで、直後の SPE は計算を進めることが出来る。この際、桁上がりは 128 ビットごとに纏め、16 バイトの数値として通信し、通信時間を削減する。

但し実際には効率の為に行の分割を次のように行った。まず 128 行単位で分けるのではなくて、128 の倍数行単位で分ける。これは一つの SPE 内で閉じる計算を増やすことによって、SPE 間の値の受け渡しによる遅延を減らす為である。実際の分割の単位は 128 から 128×16 行とした。

この分割の仕方では $|A|$ が $128 \times 16 \times 7$ の倍数で無い時に問題となる。何故なら、各 SPE が 128×16 行単位で行を処理していくとすると、終わり付近の行を処理している時に、全く計算を行っていない SPE が存在する可能性があるからである。そこで次の補題を利用して各 SPE に均等に行を割り当てる。

補題 3.1. x, n を任意の正整数とする。$1 \leq i \leq n$ に対して $x_i = \lfloor \frac{x+i-1}{n} \rfloor$ とおく。この時 $|x_i - \frac{x}{n}| < 1$ かつ $\sum_{i=1}^{n} x_i = x$ である。 □

ここで $|A| = 128x$ と置き、$n = \lceil \frac{x}{16\times 7} \rceil \times 7$ として、補題 3.1 を用いる。こうして得られた x_i を用いて $128x_i$ 行を SPE に割り当てる。ここで $x_i \leq 16$ となるので、分割の単位は 128×16 行で抑えられる。また n が 7 の倍数であるので、最後まで 7 個の SPE が並列に動作する。さらに各 x_i の差は 1 より小さいので処理量はほぼ均等である。以上の工夫により全体の計算時間は減少する。

3.2 SPE 内の処理の高速化

命令間の依存関係を減らす目的でループアンローリングを行った。3.1 節で一つの SPE が担当する行数を 128×16 としたのは、主にループアンローリングの効果を向上させるのが目的である。

SPE は 128 ビットのレジスタを 128 個持っている。128×16 行分の要素を計算するのに必要なベクトルは全てレジスタに保持することが出来るので、高速な計算が行える。

SPE には Even パイプラインと Odd パイプラインの二つのパイプラインが存在しており、Even パイプライン用の命令と Odd パイプライン用の命令は同時に実行出来る。Even パイプライン用の命令に四則演算やビット演算などが含まれているので、基本的には Even パイプラインを中心に使うことになるが、出来るだけ Odd パイプラインの命令に置き換えることで、高速化を行った。

3.3 PPE と SPE 間のデータ転送の高速化

SPE からまだページングされていないメモリへのアクセスを行うと、PPE からアクセスした場合と比較してページング処理に長い時間がかかるため、事前に PPE 側でメモリのページングを強制的に行った。

Cell は DMA と呼ばれる非同期の PPE-SPE 間メモリ転送機構を持つ。DMA は同時に三命令までのメモリ転送を実行できるが、7 個の SPE から同時に DMA 命令が発行された場合、スケジューリングが完全には公平でないため、いつまで経っても命令が実行されない場合がある。特に実行委員会のサンプルプログラムは DMA によりメインメモリにアクセスしスピンロックを行っていたため、この問題が顕著なものとなっていた。これを防ぐため、DMA 命令の削減とシグナルを用いた通信の整理を行った。

4 自明な入力への対応

入力によっては線形時間で編集距離を計算出来ることが有る。また最終的に動的計画法を用いるにしても、その作業量を減らすことが可能なことがある。本プログラムでは以下のような入力に対する枝刈りを行った。

1. 文字列 A が文字列 B を部分列として含む場合:
   もし文字列 A が文字列 B を部分列として含む、即ち或る $1 \leq p_1 < p_2 < \cdots < p_{|B|} \leq |A|$ が存在し、任意の $1 \leq i \leq |B|$ について $B[i] = A[p_i]$ であったとする。この時明らかに $d(A, B) = |A| - |B|$ である。A が B を部分列として含むかの計算は $O(|A|+|B|)$ 時間で行えるので、ビット並列化を施した動的計画法よりも高速に編集距離を求めることが出来る。

2. 文字列 A と B が共通接頭辞及び接尾辞を持つ場合:
   $A = B$ の場合は (1) で考慮されているので、ここでは $A \neq B$ であるとする。A と B の共通接頭辞の長さを l, 共通接尾辞の長さを l' とする。$A \neq B$ であるので $l + l' < min(|A|, |B|)$ である。この時、明らかに $d(A, B) = d(A[l+1 \cdots |A| - l'], B[l+1 \cdots |B| - l'])$ である。

   この処理に必要な時間は $O(|A|+|B|)$ である。ここから即座に $d(A, B)$ を計算することは出来ないが、この前処理によって文字列の長さを減らすことが出来るので、編集距離の計算にかかる時間を短縮出来る。

3. 文字列 A と B で使用する文字が全く異なる場合:
   この場合、文字列 A を文字列 B に変形するには、$|B|$ 文字置換を行った後に $|A| - |B|$ 文字削除するしかない。よって $d(A, B) = |A|$ となる。元々の A と B は同じ文字を使用しているが、前処理 (2) によって A と B で使用する文字が被らなくなることが有りうるので、(2) と同時に利用するのが良い。

また、枝刈りにより文字列長が著しく短くなった場合は、使用する SPE を 1 個に押さえ、SPE でのプロセス起動時間を節約するなどの細かな高速化も行った。

5 テスト

本プログラムでは 4 節で述べた自明な入力への対応などの様々なヒューリスティクスを用いているので、それらが全て正しく動作しているかを確認することは難しい。

表 1: 決勝ラウンド結果

問題	$\|A\|$	$\|B\|$	性質	時間 (s)	順位
1	94216	92416		0.07910	2
2	8064	1048320	(1)	0.00250	1
3	130816	130816	(2)	0.00115	1
4	130944	130944		0.13940	3
5	130816	130816	(3)	0.00129	1
6	131072	131072		0.13951	3
7	131072	131072		0.13951	3
8	92672	185344	(1)	0.00092	1
9	16384	1048320	(1)	0.00258	1
10	130816	130816	(2)	0.06736	3

その為、機械的且つ網羅的なテストが不可欠である。そこで本プログラムを作成する際には、様々な状況を考慮した入力例を大量に生成し、それらに対して全て正しい答えを出すかを調べるテストを行った。このようなテストをプログラムの改変を行う度に行うことで、バグの混入を避けることが出来る。

6 決勝ラウンド結果

表 1 は決勝ラウンドでの各問題に対する結果である。$|A|, |B|$ は入力文字列長を表す。性質は 4 節で取り上げた枝刈りのうち適用出来るものを意味している。実行委員会によるサンプルプログラムは問題 1 と 2 で約 22 秒、問題 3 から 10 で約 44 秒の時間を要した。枝刈りが利用出来ない問題で処理時間を比較すると本プログラムは約 315 倍の高速化が達成出来ていることが分かる。また枝刈り (1) 又は (3) が利用できる入力は、$O(|A| + |B|)$ 時間で編集距離を計算出来るので、殆ど時間を要しない。これらの入力については全て一位となっている。

7 まとめと今後の課題

様々な高速化を施すことによって実行委員会によるサンプルプログラムに対して 315 倍の高速化を達成することが出来た。

コンテスト終了の現時点ではさらなる高速化の余地があることが分かっている。例えば現在の実装では SPE の 128 ビットのレジスタを利用する為にビット並列化法で用いられるベクトルを 128 行 1 列単位で検索している。その為、式 (1) の計算に 128 ビットの加算が必要になるが、

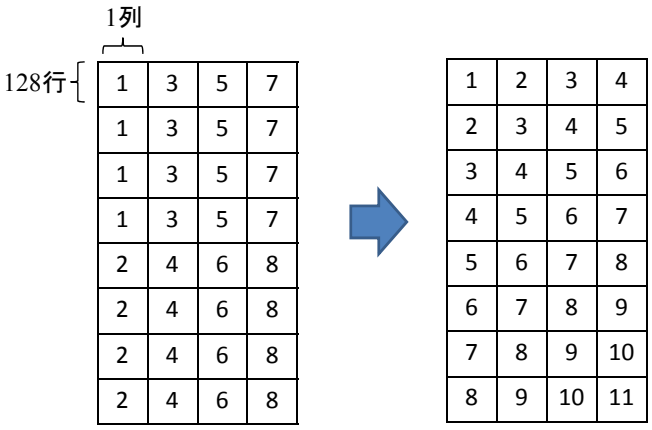

図 2: 左: 現在のプログラムの D0 の持ち方。右: 効率化された D0 の持ち方。同じ数字のブロック (合計 128 ビット以下) が一つのレジスタに格納され、数字の順番で計算される。

SPE には 128 ビットの加算が一命令では行えない為、32 ビットの加算命令を 4 回繰り返し行うことでこれを模倣している。しかし、これはビット並列化法で用いられるベクトルの持ち方を図 2 のように変えると解決する。何故ならば SPE には 32 ビットの加算を 4 つ並列で行う命令があるからである。これによって 128 ビットの加算を計算する時に発生していた繰り上がりによる依存関係も減少する為高速化が見込める。このような Cell に特化した高速化方法を考えることで、さらなる高速化を達成することが今後課題となるであろう。

参考文献

[1] Heikki Hyyrö. A bit-vector algorithm for computing levenshtein and damerau edit distances. *Nordic J. of Computing*, 10(1):29–39, 2003.

[2] Heikki Hyyrö and Gonzalo Navarro. Faster bit-parallel approximate string matching. In *CPM '02: Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, pages 203–224, 2002.

[3] Vladimir Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones (original in Russian). *Russian Problemy Peredachi Informatsii 1*, pages 12–25, 1965.

[4] Cell Challenge 2009 実行委員会. Cell spped challenge 2009. http://www.hpcc.jp/sacsis/2009/cell/.