CABIOS

Vol. 13 no. 2 1997
Pages 145-150

Using video-oriented instructions to speed up sequence comparison

A.Wozniak

Abstract

Motivation: *This document presents an implementation of the well-known Smith–Waterman algorithm for comparison of proteic and nucleic sequences, using specialized video instructions. These instructions, SIMD-like in their design, make possible parallelization of the algorithm at the instruction level.*
Results: *Benchmarks on an ULTRA SPARC running at 167 MHz show a speed-up factor of two compared to the same algorithm implemented with integer instructions on the same machine. Performance reaches over 18 million matrix cells per second on a single processor, giving to our knowledge the fastest implementation of the Smith–Waterman algorithm on a workstation. The accelerated procedure was introduced in LASSAP—a LArge Scale Sequence compArison Package software developed at INRIA—which handles parallelism at higher level. On a SUN Enterprise 6000 server with 12 processors, a speed of nearly 200 million matrix cells per second has been obtained. A sequence of length 300 amino acids is scanned against SWISSPROT R33 (18 531 385 residues) in 29 s. This procedure is not restricted to databank scanning. It applies to all cases handled by LASSAP (intra- and inter-bank comparisons, Z-score computation, etc.).*
Availability: *This implementation is available through LASSAP package http://www-rocq.inria.fr/genome*
Contact: *E-mail: Andrzej.Wozniak@inria.fr*

Introduction

The performance of proteic and nucleic sequence comparison algorithms is one of the key issues for sequence data bank scanning. Using the regular Smith–Waterman (SW) algorithm, the actual levels of performance of processors used in workstations varies from 2 to 10 million comparisons per second.

There are some specialized hardware solutions, like BIOCCELERATOR (Ltd., 1996) or SAMBA (Lavenier, 1996), which can give performance of more than two orders of magnitude higher than workstations, but these solutions also have some drawbacks: high cost, difficult programming and debugging, hardware that cannot be re-used for other purposes. This makes the alternate, pure software implementation, possibly running in parallel on many workstations, an attractive solution. This is the case of LASSAP developed at INRIA (Glémet and Codani, 1996).

From the execution profile of LASSAP sequence comparison with the SW algorithm, we found that the program spends 99% of the time in the SW algorithm procedure. So every improvement in the speed of this procedure will give almost the same improvement at the application level.

In the following section, we describe our implementation of the algorithm; the next section presents benchmarks of real data bank scanning. The final section presents the conclusions.

Algorithm implementation

Figure 1 presents the basic affine-scoring SW (Smith and Waterman, 1981) algorithm expressed in C language.

Two protein sequences, seqA and seqB, of length respectively lena and lenb, are compared using the scoring matrix matrix and two parameters gap_open and gap_ext, representing the penalties for opening and extending a gap, respectively.

One can see this algorithm as a matrix of calculi of size lena × lenb, where one letter of the sequence seqB (outer loop) is compared against all letters of the sequence seqA (inner loop). For each comparison, three intermediary results are produced in each iteration: nogap, a_gap and b_gap, i.e. perfect alignment and penalties for opening and extending the gap on seqA and seqB sequences, respectively. The final value of score is computed as the maximum of all the values found in each iteration. Only two vectors of intermediate results nogap[lena] and b_gap[lena] need to be stored. The overall complexity of the SW algorithm is $O(\text{lena} \times \text{lenb})$.

Owing to growth of the size of databanks, scanning using the SW algorithm can take hours on a single workstation. Moreover, emerging large-scale analysis such as building families of sequences leads to inter- and intra-data bank comparisons. This process can take weeks of computation.

Execution speed-up can be obtained using multiprocessors or clusters of workstations because sequence comparisons are independent. In this article, we present a parallelization technique at the processor instruction level to speed-up execution of the SW algorithm further.

Two potential kinds of parallelization can be considered: for sequence-to-bank comparison and for sequence-to-sequence comparison. The first is based on lack of dependency

INRIA, BP 105, 78153 Le Chesnay Cedex, France

```
int nogap[MAX_A], b_gap[MAX_A];

int sw(char *seqA, int lena, char *seqB, int lenb,
       int gap_open, int gap_ext, SCORING_MATRIX_TYPE matrix)
{ int i, j;
  int last_nogap, prev_nogap;
  int score = 0;
  init_vect(lena, nogap, 0);
  init_vect(lena, b_gap, - gap_open);
  for (i=0; i<lenb; ++i) {
    int a_gap;
    last_nogap = prev_nogap = 0;
    a_gap = - gap_open;
    for (j=0; j<lena; ++j) {
      a_gap = MAX((last_nogap  -  gap_open - gap_ext), (a_gap - gap_ext));
      b_gap[j] = MAX((nogap[j] - gap_open - gap_ext), (b_gap[j] - gap_ext));
      last_nogap = MAX((prev_nogap + matrix[seqA[j]][seqB[i]]), 0);
      last_nogap = MAX(last_nogap, a_gap);
      last_nogap = MAX(last_nogap, b_gap[j]);
      prev_nogap = nogap[j];
      nogap[j] = last_nogap;
      score = MAX(score, last_nogap);  }}
  return score;
}
```

Fig. 1. Smith–Waterman affine-scoring algorithm.

between two consecutive comparisons. Thus, one sequence can be compared against two or more sequences at the instruction level using the same control structure. Of course, each compared sequence has its own set of intermediate variables. An implementation using similar principles is described in Alpern *et al.* (1995).

The above approach is limited to pure databank scanning. It is not possible, for example, to implement easily sequence shuffling procedures to compute *Z*-scores (Lipman *et al.*, 1984; Pearson and Lipman, 1988). Furthermore, it cannot take advantage of LASSAP's programming capabilities which allow combination 'on the fly' of pairwise comparison-based algorithms. As an example, LASSAP implements an algorithm called *blsw* which computes, for every pair of sequences, SW alignments depending on the results of *blast* (Altschul *et al.*, 1990).

Our approach is to parallelize at the single sequence comparison level. This preserves efficiency in every case. One can see from Figure 1 that results in the (i, j)th iteration depend on $(i-1, j)$, $(i-1, j-1)$ and $(i, j-1)$ values. This can be visualized as vertical, diagonal and horizontal dependencies (Figure 2).

If we shift the computation of the second row one place to the right, rows 1 and 2 can be executed in parallel, with only a small overhead of two steps: one on the beginning of the inner loop and the second on its end.

If the length of the sequence seqB is odd, on the last step only one row is computed. As the values of nogap[j] and b_gap[j] computed in the first row are used by the second one, we can transfer these values locally, dividing by two the memory band width for storing and loading these values.

To make the computation uniform, one can add one dummy symbol at the beginning and at the end of the sequence seqA, and a dummy symbol to sequence seqB in case its length is not even. This scheme can be extended to parallel computation of an arbitrary number of rows.

Figure 3 presents a four-row implementation of the algorithm in C pseudocode. Data type Vint4 is a vector containing four integers. VMAX and VSHIFT are vector operators. The first computes a vector of pairwise maxima of argument vectors, the second shifts left all components of its

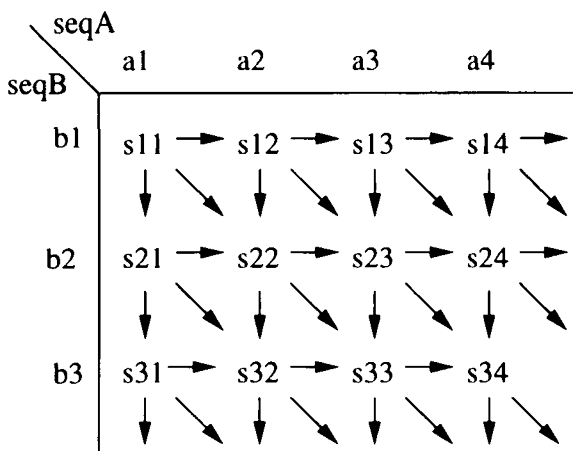

Fig. 2. Data flow diagram.

```
int nogap[MAX_A], b_gap[MAX_A];

int Vectsw(char *seqA, int lena, char *seqB, int lenb,
       int gap_open, int gap_ext, SCORING_MATRIX_TYPE matrix)
{
  int i, j, score;
  Vint4 Vscore = {0, 0, 0, 0};
  Vint4 Vzero = { 0, 0, 0, 0};
  Vint4 Vgap_ext = { gap_ext, gap_ext, gap_ext, gap_ext};
  Vint4 Vgap_open = { gap_open,  gap_open,  gap_open,  gap_open};
  init_vect(lena+6, nogap, 0);
  init_vect(lena+6, b_gap, - gap_open);
  for (i=0; i<lenb+3; i+=4) {
    Vint4 Vmatrix;
    Vint4 Va_gap = - Vgap_open;
    Vint4 Vb_gap = { b_gap[2], b_gap[1], b_gap[0], 0};
    Vint4 Vnogap = { nogap[2], nogap[1], nogap[0], 0};
    Vint4 Vlast_nogap = Vzero;
    Vint4 Vprev_nogap = Vzero;
    for (j=0; j<lena+3; ++j) {
      Vb_gap = VSHIFT(Vb_gap, b_gap[j+3]);
      Vnogap = VSHIFT(Vnogap, nogap[j+3]);
      Va_gap = VMAX((Vlast_nogap -  Vgap_open - Vgap_ext), (Va_gap - Vgap_ext));
      Vb_gap = VMAX((Vnogap - Vgap_open - Vgap_ext), (Vb_gap - Vgap_ext));
      Vmatrix = { matrix[seqA[j+3]][seqB[i]], matrix[seqA[j+2]][seqB[i+1]],
                  matrix[seqA[j+1]][seqB[i+2]], matrix[seqA[j]][seqB[i+3]] };
      Vlast_nogap = VMAX((Vprev_nogap + Vmatrix), Vzero);
      Vlast_nogap = VMAX(Vlast_nogap, Va_gap);
      Vlast_nogap = VMAX(Vlast_nogap, Vb_gap);
      Vprev_nogap = Vnogap;
      Vnogap = Vlast_nogap;
      b_gap[j] = Vb_gap.v[3];
      nogap[j] = Vnogap.v[3];
      Vscore = VMAX(Vscore, Vlast_nogap);  } }
  return max(Vscore.v[0], Vscore.v[1], Vscore.v[2], Vscore.v[3]);
}
```

Fig. 3. Parallelized Smith–Waterman affine-scoring algorithm.

vector argument and puts at the rightmost place its second integer argument. All '+' and '−' are vector add and subtract operators, respectively.

Although implementable in standard integer instruction set, this parallelization scheme gives little performance improvement over the basic implementation (see benchmarks on Figure 6).

Substantial speed-up can be achieved using special, video-oriented instructions like VIS (Visual Instruction Set) found in the SUN ULTRA SPARC processor (Sun Microelectronics, 1996a).

VIS instructions are executed in specially enhanced floating point unit (FPU) and use the 64-bit registers of the FPU. Instructions operate on two 32-bit, or four 16-bit integer data packed in a 64-bit double word. These instructions include arithmetic ADD, SUBTRACT, COMPARE and MULTIPLY, logic AND, OR, XOR, NOT, and data conversion and manipulation. One can add in one instruction two sets of four 16-bit integers and get four 16-bit results, or multiply them by four 8-bit integers. Details can be found in Sun Microelectronics (1996b).

We can use VIS instructions to execute in parallel four rows of the SW algorithm. The implementation of most operations from Figure 3 using VIS inline interface is straightforward. We replace type Vint4 by type vis_d64 which is actually a vector of four 16-bit integers packet in a 64-bit word. VMAX operation is replaced by VIS_MAX_INT from Figure 5. Add, subtract and VSHIFT are replaced, respectively, by vis_fpadd16, vis_fpsub16 and vis_faligndata instructions.

The ULTRA SPARC VIS compare instructions compute a 4-bit mask of result, which can be used by a selective store instruction (from four 16-bit words only these with corresponding status bits set to 1 are stored). Unfortunately, we need many comparisons with re-use of the intermediate results which implies reload of the stored values. This is time

```
#define MAX_INT_NO_JUMP(int_max, int_val)\
{\
  int mask;\
  int_max  -= int_val;\
  /* no overflow allowed here ! */   \
  mask      = (int_max>>(8*sizeof(int_max)-1));\
  /* -1 if negative, 0 otherwise */ \
  int_max & int_max + (int_val); ~mask; \
}
```

Fig. 4. Max operation with no branch instruction.

and memory band width consuming, and decreases effective execution speed.

Instead, we use maximum integer value calculation based on integer code presented on Figure 4. It computes the maximum of two signed integer values int_max and int_val, the result is stored in int_max. This code does not use conditional branches, so it avoids potential jump misprediction and flushing of pre-decoded, partially executed or speculatively executed instructions on some CPU.

The result is correct as long as there is no overflow in the computation of the difference (int_max - int_val) so the method can be used safely on a restricted range of arguments.

Figure 5 shows the implementation of the above method for packed 16-bit integers in VIS instructions. There is no arithmetical bit-shift instruction in VIS, so the mask is generated with the multiplication instruction vis_fmul8ulx16. As this multiplication is performed on a separate unit and the graphic processor is a double-issue engine, there is little if any performance degradation.

The ULTRA SPARC graphics unit is fully pipelined and has a latency of three clock cycles for multiplication. So the result can be used by another instruction only three clock cycles later without stalling the CPU.

In our implementation, we process eight rows at once, so another multiplication can be started in the following cycle, keeping the CPU busy for maximum performance. The large set of 32 64-bit registers in the graphics unit helps keep all the computations local.

The inner loop is executed (lena+7)/8 times. As we mentioned before, we add seven dummy symbols at the beginning and at the end of the sequence A, and the sequence B is extended to a multiple of eight.

```
#define VIS_MAX_INT(aa, bb) \
{\
  vis_d64 mm;\
  aa = vis_fpsub16(aa,bb);\
  mm = vis_fand(aa,fmask_0x8000800080008000);\
  mm = vis_fmul8ulx16(fmask_0x0002000200020002,mm);\
  aa = vis_fandnot(mm,aa);
  aa = vis_fpadd16(aa,bb);\
}
```

Fig. 5. Max operation using VIS.

Sixteen-bit signed precision is acceptable for the comparison of proteic and nucleic sequences. Indeed, it allows one to compute a score up to 32 767 which represents on average a perfect alignment of length ~5000 with BLOSUM62 matrix. This length can be greater if mismatches and gaps are present.

The implementation guarantees saturation on the score of 32 767 (see below for details of the implementation). In case of saturation, one can compute the exact score with a standard integer implementation of the algorithm. In practice, such alignments with very high scores can be detected with faster algorithms such as fasta (Pearson and Lipman, 1988) and need, anyway, deeper studies due to their lengths. We believe that our implementation covers most of the needs when using the SW algorithm.

The result (i.e. score value) is correct up to 32 767 and then saturates at this value. The proof of correctness is as follows. The only case of overflow in the subtraction of signed integer values is when the two operands are of opposite sign and their difference is greater in magnitude than the maximum representable value (32 767, −32 768 for 16-bit integers).

In our case, the most negative value of a_gap and b_gap is -(gap_open + gap_ext). The value of nogap is always positive so the error condition could arise only when one argument of the maximum value calculation approaches 32 767 and the other one is negative.

From the algorithm in Figure 1, one can see that at any point of the calculation the values of nogap, a_gap and b_gap can change only by limited steps based on scoring matrix values, gap_open and gap_ext. The absolute value of the difference between a_gap or b_gap and last_nogap arguments of the MAX function (1) is bound by the expression:

$$2 \times (\mathit{max}(\text{matrix}) + \text{gap\_open} + \text{gap\_ext}) - \mathit{min}(\text{matrix})$$

where *max*, *min*(matrix) are the extreme substitution costs of the scoring matrix. As this value is small (usually < 200) compared to 32 767, the values of nogap, a_gap and b_gap are strictly positive when the score is reaching the saturation value.

Benchmarks

We have implemented the SW algorithm from Figure 1 both in integer (32-bit) instructions and in VIS (16-bit) for an ULTRA SPARC machine. We have chosen four typical sequences of length 104, 319 and 5217, respectively. The execution speed results are presented in Figure 6.

The VIS implementation of the algorithm for a SPARC ULTRA machine is twice as fast as the integer code giving the figure of 18 million matrix cells per second. For comparison, we give the performance of the same integer code recompiled and executed on a DEC ALPHA 300 MHz (DEC Unix) and PENTIUM PRO 200 MHz (LINUX).

Machine \ Sequence	104 vs. 104	329 vs. 104	329 vs. 329	5217 vs. 5217	Mean	Comments
ULTRA SPARC 167 MHz integer	8.71	7.83	9.09	7.79	8.36	Solaris 2.5 cc ver 4.0
ULTRA SPARC 167 MHz integer 2 rows	8.19	7.36	8.54	8.43	8.13	
ULTRA SPARC 167 MHz VIS	17.11	18.14	18.16	18.18	17.89	cc ver 4.0 + VIS inlines
DEC ALPHA 300MHz integer	9.85	9.69	10.02	8.62	9.54	DEC Unix 3.2D-1 cc ver. 3.11
DEC ALPHA 300MHz integer 2 rows	12.68	12.65	12.68	11.97	12.50	
PENTIUM PRO 200MHz integer	5.76	5.15	6.23	7.20	6.08	LINUX 1.2.13 gcc ver 2.7.2
PENTIUM PRO 200MHz integer 2 rows	6.16	5.20	6.44	6.61	6.10	

Fig. 6. Execution speed in millions of matrix cells per second.

Next, the code was integrated into LASSAP's software package and the real application of scanning a sequence of length 300 amino acids against SWISSPROT R33 (18 531 385 residues) was executed on a SUN ULTRA SPARC based Enterprise 6000 server with 12 processors running at 167 MHz. Figure 7 presents the results of the overall real execution time of the application as a function of the number of processors used.

The global performance scales quite perfectly with the number of processors used, reaching at 12 processors 200 million matrix cells per second. We believe we can get 500 million matrix cells per second on existing SUN Enterprise 30 processors servers. Similar performance can be obtained using ULTRA SPARC workstations connected by a network.

This level of performance can be compared to specialized commercially available hardware solutions such as the BIOCCELERATOR (Ltd., 1996) machine. Based on the XILINX field programmable gate arrays (FPGA), the accelerator is composed of a rack containing up to 16 accelerator chips connected to a workstation through the SCSI interface. One chip computes 80 million matrix cells per second, giving 1280 million matrix cells per second in the maximum configuration.

N	Real Execution Time	Global Performance	Performance per Processor
1	5'10"	17.93	17.93
2	2'38"	35.19	17.59
4	1'19"	70.37	17.59
6	0'53"	104.89	17.48
8	0'40"	138.98	17.37
10	0'33"	168.46	16.84
12	0'29"	191.70	15.97

Fig. 7. Real execution time versus number of processors in millions of matrix cells per second.

Conclusion

Excellent performance has been obtained using VIS, albeit the VIS instructions were not intended initially for general purpose computation. We believe that the method is attractive and can be extended to other types of parallelizable algorithms giving more processing power to pure software implementations.

Although our solution offers less performance than specialized hardware, it offers more flexibility. Furthermore, the principle is applicable to other instruction sets such as MMX for INTEL PENTIUM and PENTIUM PRO processors which will be introduced soon (Gwennap, 1996), enabling the use of clusters of PCs as computing power.

We expect that other processor families like DEC ALPHA, HP PA-RISC and PowerPC will follow the trend, adding VIS-like instructions to their CPUs. Since the performance scales with the CPU clock speed, we expect proportionally higher performance on the announced ULTRA SPARC 250 MHz machines.

The parallelization scheme presented in this paper is not limited to VIS-like implementation On a 64-bit CPU, one can pack three 21-bit or four 16-bit integers in the machine word. Vector add and subtract can be computed using bit masks to avoid carry propagation between packed integers. The proof of lack of overflow can be used to simplify the VMAX operation. On some machines, one can expect performance gain compared to full integer implementation.

Acknowledgements

I would like to thank E.Brouart of Sun Microsystems, France, for providing us access to the Enterprise server for benchmarking and G.Pelissier of Sun

Microelectronics for helping me start with VIS. Many thanks to Genome Group members at INRIA: to J.P.Comet for the formal proof of the correctness of the saturation formula, and to E.Glémet and J.J.Codani for the effort in integration of my code in LASSAP.

References

Alpern,B., Carter,L. and Gatlin,K. (1995) Microparallelism and high-performance protein matching. In: *Proc. Supercomputing.*

Altschul,S., Gish, W., Miller,W., Myers,E. and Lipman,D. (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

Glémet,E. and Codani,J. (1996) Lassap: a large scale sequence comparisons package. *Comput. Appl. BioSci.*, **13**, 137–144.

Gwennap,L. (1996) Intel's mmx speeds multimedia *Microprocessor Rep.*, **10**, 6–10.

Lavenier,D. (1996) Samba: Systolic accelerators for molecular biological applications. Technical Report TR 988, IRISA, France.

Lipman,D., Wilbur,W., Smith,T. and Waterman,M. (1984) On the statistical significance of nucleic acid similarities. *Nucleic Acids Res.*, **12**, 215–226.

Ltd.,C. (1996) *BIOCCELERATOR Manual.* Petah-Tikva, Israel.

Sun Microelectronics. (1996a) *UltraSPARC-I User's Manual, Rev 1.4.*

Sun Microelectronics. (1996b) *Visual Instruction Set User's Guide.*

Pearson,W. and Lipman,D. (1988) Improve tools for biological sequence comparison. *Proc Natl Acad. Sci. USA*, **85**, 2444–2448.

Smith,T. and Waterman,M. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

Received on September 20, 1996; revised on November 13, 1996; accepted on November 21, 1996