
Efficient Implementation of Nearest Neighbor Classification^{*}

José R. Herrero and Juan J. Navarro

Computer Architecture Department, Universitat Politècnica de Catalunya,
Jordi Girona 1-3, Mòdul D6, E-08034 Barcelona, (Spain)
{josepr,juanjo}@ac.upc.es

Summary. An efficient approach to Nearest Neighbor classification is presented, which improves performance by exploiting the ability of superscalar processors to issue multiple instructions per cycle and by using the memory hierarchy adequately. This is accomplished by the use of floating-point arithmetic which outperforms integer arithmetic, and block (tiled) algorithms which exploit the data locality of programs allowing an efficient use of the data stored in the cache memory.

1 Introduction

The Nearest Neighbor (NN) classification procedure is a popular technique in pattern recognition, speech recognition, multitarget tracking, medical diagnosis tools, etc. A major concern in its implementation is the immense computational load required in practical problem environments. Other important issues are the amount of storage required and the data access time. In this paper, we address these issues by using techniques widely used in linear algebra codes. We show that a simple code can be very efficient on commodity processors and can sometimes outperform complex codes which can be more difficult to implement efficiently.

1.1 Nearest Neighbor Classification

The classification problem consists in assigning a class from ℓ classes C_1, C_2, \dots, C_ℓ to each of the D_{size} unclassified vectors $\mathbf{X}^j = [x_1^j, x_2^j, \dots, x_{V_{size}}^j]$ with length V_{size} , for $j = 1, \dots, D_{size}$. The NN classification uses a set of vectors $\mathbf{P}^k = [p_1^k, p_2^k, \dots, p_{V_{size}}^k]$, for $k = 1, \dots, P_{size}$, called a set of prototypes, whose class is known. Then, an unclassified vector \mathbf{X}^j is classified in the same class as \mathbf{P}^s if \mathbf{P}^s is the prototype with minimum distance to \mathbf{X}^j , that is

$$d(\mathbf{X}^j, \mathbf{P}^s) = \min_{k=1, \dots, P_{size}} d(\mathbf{X}^j, \mathbf{P}^k)$$

^{*}This work was supported by the Comissionat per a Universitats i Recerca of the Generalitat de Catalunya (BE94/Annex 3-642 and BE94-A1/110) and the Ministerio de Educación y Ciencia of Spain and the EU FEDER funds (TIN2004-07739-C02-01)

<http://people.ac.upc.edu/josepr/>

where, in our examples, the distance function is defined as the square of the Euclidean distance:

$$d(\mathbf{X}^j, \mathbf{P}^k) = \sum_{i=1}^{V_{size}} (x_i^j - p_i^k)^2$$

Hence, the distance between the vector \mathbf{X}^j , which is to be classified, and all the vectors \mathbf{P}^k , $k = 1, \dots, P_{size}$, in the prototype set must be computed. The time needed to classify a set of D_{size} vectors is, consequently, proportional to $(V_{size} \times D_{size} \times P_{size})$.

In the algorithms we use, which are shown below, the set of unclassified vectors is kept in matrix $D(V_{size}, D_{size})$ where $x_i^j = D(i, j)$. The set of prototypes is kept in matrix $P(V_{size}, P_{size})$, where $p_i^k = P(i, k)$. Vector $ClassP(P_{size})$ indicates the class the prototypes belong to, i.e. $ClassP(k) = r$ if \mathbf{P}^k belongs to class C_r . The result of the classification is stored in vector $ClassD(D_{size})$, where $ClassD(j) = r$ if \mathbf{X}^j is classified as belonging to class C_r .

Figure 1a shows the common brute force algorithm, which we label as *jki* form due to the loop ordering. In this algorithm, all of the V_{size} components of an unclassified vector \mathbf{X}^j , stored in a column of matrix D , are compared to the correspondent components of each vector in the prototype set, stored as columns of matrix P . Often, the *jki* code has been modified to exit the loop that computes the distance when the current distance exceeds the running minimum found, reducing the number of computations required for classification, while keeping the same accuracy as the brute force algorithm. Figure 1b shows the modified *jki* loop, henceforward called *jki_exit* algorithm.

```

MAX = 2 ** 30
DO J = 1, Dsize
  mindis = MAX
  DO K = 1, Psize
    distance = 0
    DO I = 1, Vsize
      sub = D(I,J) - P(I,K)
      distance = distance + sub*sub
    ENDDO
    IF (distance.LT.mindis) THEN
      mindis = distance
      mincla = ClassP(K)
    ENDIF
  ENDDO
  ClassD(J) = mincla
ENDDO
a)

MAX = 2 ** 30
DO J = 1, Dsize
  mindis = MAX
  DO 2 K = 1, Psize
    distance = 0
    DO 1 I = 1, Vsize
      sub = D(I,J) - P(I,K)
      distance = distance + sub*sub
      IF (distance.GT.mindis) GO TO 2
    1 CONTINUE
    mindis = distance
    mincla = ClassP(K)
  2 CONTINUE
  ClassD(J) = mincla
ENDDO
b)

```

Fig. 1. Codes for the **a)** *jki* and **b)** *jki_exit* forms

1.2 Related Work

A major concern in the implementation of the NN technique is the immense computational load associated with it and the large amount of computer memory required when large prototype and data sets exist. These problems have been addressed at length and many alternatives proposed: special-purpose hardware, such as systolic arrays [9] and several approaches have shown to be computationally advantageous over the brute force method:

- Modified metrics as alternative distance measures to the Euclidean distance used in classical NN classifiers [3, 5, 10, 12, 15, 18, 19, 21].
- Selection of a design subset of prototypes from a given set of prototype vectors [5, 8, 11, 18, 21] and generation of prototype reference vectors [7].
- Use of fuzzy logic and Self Organizing Maps [4, 14].

A paper [2] compared RISC-based systems to special purpose architectures for Image Processing and Pattern Recognition (IPPR). They concluded that although a lot of progress had been achieved in RISC technology, low advantages could be obtained for IPPR due to the difficulty of producing efficient code for such machines. In this paper, however, we study ways of improving the efficiency of Nearest Neighbor classification on general purpose RISC-based High Performance Workstations since their price can make them cost-effective. Our approach aims to maximize speed maintaining the accuracy of the brute force method by means of an efficient codification of the algorithm using floating-point arithmetic, which increases speed, and a block algorithm, which reduces the number of misses in the cache memory. Such techniques have often been used in numerical applications [1, 20] but never, to our knowledge, to NN classification.

1.3 Processor Overview

Our tests have been carried out on two high performance workstations, which incorporate superscalar processors: an HP PA-7150 [13] and a DEC Alpha AXP-21064 processor [6] respectively. Both implement Integer/Floating-Point two-way superscalar operation, i.e. one integer and one floating-point instruction can be issued each cycle. Loads and stores of floating-point registers are treated as integer operations. The CPU can read two consecutive data words (a total of 8 bytes) every cycle from the external data cache.

The PA-7150 cache has 256 Kbytes, with a line size of 32 bytes. The number of elements in a line (L) is therefore 32 for byte, 8 for simple and 4 for double precision floating-point data types. According to our experiments, a cache miss produces a penalty of 35 cycles. Consequently, the number of Cycles Per Miss (CPM) is 35. The AXP-21064 incorporates separate 8 Kbyte on-chip instruction and data caches, and a 1 Mbyte off-chip unified cache. All of them have line size equal to 32 bytes. A first level cache hit has a 3 cycle latency while a miss which hits in the second level cache is available in 11 cycles for the first word, and 18 for the following one.

1.4 Performance Metrics

In order to compare different codes that solve the same problem, CPU_{time} is a clear candidate to be used as a metric. However, when problem size is changed from execution to execution it is advisable to use a metric normalized to the size of the problem. In this paper we introduce Normalized Cycles (NC), which for our classification problem is computed by:

$$NC = \frac{CPU_time_in_cycles}{V_{size} \cdot P_{size} \cdot D_{size}} \quad (1)$$

We model the NC with the following expression:

$$NC = NC(cpu) + NC(mem) \quad (2)$$

where $NC(cpu)$ is the component obtained considering no misses in the memory hierarchy (caches, TLBs, page faults) and $NC(mem)$ represents the penalty cycles due to the misses in the memory system. In the analytical models we develop in this paper, we do not consider the misses produced by instruction fetches since a separate instruction cache exists and the programs we evaluate are sufficiently small so that no instruction misses occur. We include only misses produced by load accesses to matrices since they constitute almost all the data accesses. Experimental results of the NC for different codes are reported in sections 2 and 3. All our programs are written in Fortran.

2 Algorithm Analysis

In this section we present the results obtained from the execution of several codes using distinct data representations. For certain applications floating-point arithmetic is required. In other cases, however, vector elements can be coded as bytes. We have implemented the NN codes using three different data types for the vector elements: byte, simple and double precision floating-point numbers, which require 1, 4 and 8 bytes of storage space respectively.

For any of the data types used, results depend on problem size. For small problems, the sizes of both the prototype and data to be classified have been defined to be small enough to fit into the cache simultaneously. Data are brought into the cache the first time they are referenced. Subsequent references will hit in cache, since all data are kept in it. Executing a code many times and dividing the execution time by the number of times it is performed hides the misses from the first execution. Therefore, $NC(mem) \approx 0$ for a small problem executed many times and NC is approximately $NC(cpu)$:

$$NC_{SmallProblem} \approx NC(cpu) \quad (3)$$

When the problem size is big enough so that all the data do not fit in the cache at the same time, cache misses arise and data are flushed from the cache between uses. Locality is not well exploited resulting in a poor cache utilization. The $NC(mem)$ component in large problems can be easily estimated by:

$$NC(mem) \approx NC_{LargeProblem} - NC_{SmallProblem} \quad (4)$$

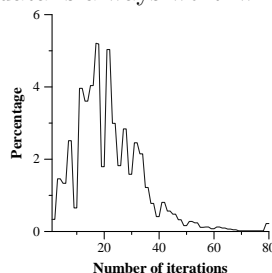
since $NC(cpu)$ are the same for both large and small problems.

Table 1. NC obtained for different problem sizes

Data type	PA-7150				AXP-21064			
	Small Problem		Large Problem		Small Problem		Large Problem	
	jki_exit	jki	jki_exit	jki	jki_exit	jki	jki_exit	jki
byte	6.1	15.0	7.1	16.3	24.0	24.0	25.1	25.1
simple float	4.5	3.6	6.0	8.4	22.1	11.9	23.9	19.7
double float	4.5	3.6	7.0	13.3	23.0	20.6	29.2	21.0

To analyze the consequences data size has on performance, two different problem sizes have been tested. Considering the PA-7150's 256 Kbytes data cache, as a small problem we used a database of 200 vectors — 100 for the prototype set and 100 used as data for classification — where each vector has 80 elements. Experiments on a large problem were carried out on a database of 20852 vectors — 10426 for the prototype set and 10426 used as data for classification — each also having 80 components. Table 1 shows the NC measured for a small and a large problem for the different algorithms and data types. These results show that the use of floating-point data is always worthwhile.

Data initialization is important since it impacts on the performance of the *jki_exit* algorithm. A distribution obtained from a real application has been used. The figure on the right shows the probability distribution of the number of iterations of the inner loop computed before it is exited. The mean value of the number of iterations is $\bar{x} = 22$.



2.1 The $NC(cpu)$ Component

Despite considerably increasing the memory requirements, using simple (4 bytes) or double (8 bytes) floating-point data is better than a simple byte (assuming a datum can be represented in a single byte). This is so because the PA-RISC 7150 processor can issue one load of a floating-point value together with one floating-point multiplication and one floating-point addition (or subtraction) per cycle. On the other hand, when integer arithmetic (byte or integer data type) is used, just one instruction - a load, a multiplication, an addition or a subtraction - can be issued each cycle. Moreover, several data conversions are performed, since all the arithmetic is performed on 32 bit data. Furthermore, the multiplication is computed on the floating-point unit which requires data movements from a general purpose register to a floating-point register, and vice-versa, through memory. From these results we infer that the use of floating-point arithmetic is always beneficial.

In the *jki* code, the compiler applies software pipelining [16] producing an instruction scheduling which circumvents the problem introduced by data dependencies. When a conditional branch is present in the loop body, as in the *jki_exit* code, no software pipelining is applied. Consequently, due to the dependencies between instructions plus the extra instructions implementing the "if" statement, the $NC(cpu)$ of the *jki_exit* becomes larger than that of the *jki* even for the reduced number of iterations of the *jki_exit* inner loop; e.g. an average of 22 for *jki_exit* in our experiments as opposed to 80 for *jki*.

The same is basically true for the AXP-21064. The overhead of the *jki_exit* code is so large that this algorithm is outperformed by the simpler *jki* code. However, since the compiler we had available was not able to perform software pipelining, the instruction scheduling obtained was not as effective as that of

the PA-7150. We will thus center our attention on the latter although results of a hand coded software pipelined version developed for the former processor will be shown at the end of the paper.

2.2 The $NC(mem)$ Component

For a small problem the $NC(mem)$ is negligible. As the problem size grows, the $NC(mem)$ component of the NC increases while the $NC(cpu)$ remains constant (equations (3) and (4)). In order to predict the number of cache misses a code produces, it is important to take several aspects into consideration. We analyze the jki and jki_exit codes and make comments upon the relevant points.

Spatial Locality

For each inner loop iteration one data element and one prototype element are referenced. It is important to note that for both data structures, accesses to consecutive addresses (column accesses for both D and P) are performed in consecutive iterations of the innermost loop I , exploiting the spatial locality. When the first element in a line which is not present in cache is referenced, a cache miss is produced with a penalty of CPM cycles. However, since the whole line, containing L elements, is brought into the cache, the subsequent $L - 1$ accesses to elements in that line are cache hits introducing no extra penalty cycles.

Temporal Locality

The elements of P as well as those of D are reused through the algorithm. Each element of D is reused once for each iteration of the middle loop K , while each element of P is reused for each iteration of the outer loop J .

For each iteration of the middle loop K a new column of P is referenced. However, a fixed column of D is reused for each iteration of loop K and will only be evicted from the cache, due to conflicts with elements of P , every $\frac{C}{E_{size} \cdot V_{size}}$ iterations of loop K , where C is the cache size in bytes and E_{size} is the element size: 1, 4 or 8 for byte, simple or double precision floating-point data respectively. Therefore, we can be reasonably certain that the elements of D are rarely involved in cache misses.

For each iteration of the outermost loop J , all the elements in P are referenced. However, for large matrices, when a new line of P is referenced in iteration $J = j$ a cache miss occurs. Despite having been accessed in iteration $J = j - 1$, the line has already been evicted from cache because accesses to the whole matrix P have been performed and conflicts appeared among its elements due to its large size.

An Analytical Model for $NC(mem)$

Taking into consideration the spatial and temporal locality of the algorithms presented above, we conclude that, for the jki algorithm, a total of $D_{size} \cdot P_{size} \cdot \frac{V_{size}}{L}$ misses occur for the prototype set P . Thus, from equation (1) we obtain:

$$NC(mem) \approx \frac{CPM}{L} \quad (5)$$

All the statements asserted above are valid for the *jki_exit* code with the only difference that matrix P is not completely referenced within an iteration of loop J . Given a mean number of iterations \bar{x} before the inner loop is exited, approximately $P_{size} \cdot V_{size} \cdot \frac{\bar{x}}{V_{size}}$ elements of P will be used. Assuming these data are still too many to fit in cache, $D_{size} \cdot P_{size} \cdot \frac{V_{size}}{L} \cdot \frac{\bar{x}}{V_{size}}$ misses occur. Therefore, for the *jki_exit* algorithm

$$NC(mem) \approx \frac{\bar{x}}{V_{size}} \cdot \frac{CPM}{L} \quad (6)$$

The two left columns in table 2 show the NC s obtained using our theoretical model. For these data, an estimation of the $NC(cpu)$ is obtained from the NC of the small problem (equation (3)) shown in table 1. Then, applying the results in equations (2), (3) (5) and (6) we obtain an estimation of the NC s for a large problem which are very close to the empirical results shown in table 1. It is important to note that for each data type used (byte, simple and double precision floating-point) the $NC(mem)$ differs since L changes (32, 8, 4) — see equations (5) and (6). For this reason the usage of simple floating-point data produces better NC s than the use of double floating-point data since both have the same $NC(cpu)$. Despite producing a lower $NC(mem)$, the use of byte data results in worse NC s since its $NC(cpu)$ component is too large to make it competitive.

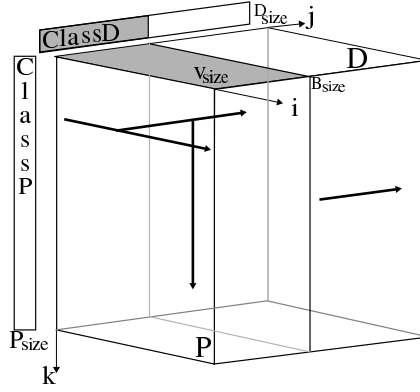
3 Block Algorithm

In this section we present a new code which exploits the locality in the algorithm considerably better, also producing an $NC(mem) \approx 0$ for large problems. Figure 2a shows the code of a block algorithm we propose for substituting the *jki* algorithm. The same idea can be applied to the *jki_exit* algorithm. In this code the number of loops has increased, but the arithmetic operations are the same. Consequently, the accuracy is exactly the same as that of the non-blocked algorithms. Figure 2b shows the Data and Computation Diagram [20] for the *block* algorithm. The shaded area shows cached data that can be reused. In the *block* code, the same column of P is referenced for each iteration of loop J , while a new column of D is accessed. The probability of cache hits for the data in D is very high, and we will assume no misses appear. For each iteration of loop K all the elements in a block of $V_{size} \times B_{size}$ elements of D are referenced. The block size B_{size} will be dimensioned so that the block fits into the cache: $B_{size} \times V_{size} < \frac{C}{E_{size}}$. If a large B_{size} is chosen, the number of intrinsic misses of P will decrease but the number of conflicts will grow. The optimal block size is considered to be approximately half the cache size C [17]. Consequently, the data in the block remain in the data cache during all the iterations of loop K . Some conflicts will arise, but their number and influence is so low to be considered null. Consequently, the $NC(mem)$ of a block algorithm is very low and can be considered insignificant.

```

MAX = 2 ** 30
DO JJ = 1, Dsize, Bsize
  DO ind = 1, Bsize
    Vaux(ind)=MAX
  ENDDO
  DO K = 1, Psize
    ind = 0
    DO J = JJ, MIN(JJ+Bsize-1,Dsize)
      ind = ind+1
      distance = 0
      DO I = 1, Vsize
        sub = D(I,J) - P(I,K)
        distance = distance + sub*sub
      ENDDO
      IF (distance.LT.Vaux(ind)) THEN
        Vaux(ind) = distance
        ClassD(J) = ClassP(K)
      ENDDO
    ENDDO
  ENDDO
ENDDO

```



a)

b)

Fig. 2. a) Code for the *block* form. b) Data and Computation Diagram for the *block* form.

The last two columns in table 2 show the experimental measures obtained for a large problem using the block algorithm proposed above for a block size $B_{size} = \frac{1}{2} \cdot \frac{256 \cdot K}{V_{size} \cdot E_{size}}$. It should be noted that the *NCs* obtained are almost identical to those shown in table 1 corresponding to a small problem.

Table 2. *NC* on the PA-7150 for large problems without and with block algorithms

Data type	<i>jki_exit</i>	<i>jki</i>	<i>block_exit</i>	<i>block</i>
byte	6.4	16.1	6.6	15.2
simple float	5.7	8.0	4.8	3.7
double float	7.2	12.3	4.9	3.9

When the byte data type is used, the improvement obtained by the use of blocks is minor, because there is a large *NC(cpu)* which was already high for the code without blocks which cannot be lowered by blocking. Moreover, there exists high spatial locality due to the fact that the number of elements in a cache line is big ($L = 32$). Simple floating-point data produced slightly better results than double floating-point data (see figure 3) due to more efficient use of the cache line (higher spatial locality).

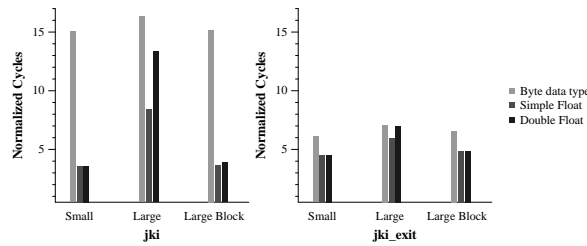


Fig. 3. Comparison of *NC* for different problem sizes (grouped by data types)

Finally in table 3 we show the results obtained when we hand-optimized the FORTRAN code on the Alpha AXP-21064 by the application of software pipelining techniques to improve the instruction scheduling for a better instruction level parallelism, and tiling to improve the use of data locality. We show only the results obtained for the single precision float data type since that was the one producing the best performance. SP means Software Pipelining, *num* Bl, means a *num* number of square blocks was applied, Pc implies that data precopies were done; BRL stands for Blocking at the Register Level meaning that tiling was also applied in order to improve the reuse of registers in the inner loop. The data in the table shows that the combination of well-known techniques applied to the Nearest Neighbor Algorithm produces significant improvements in performance.

Table 3. *NC* obtained with hand optimized code on the AXP-21064

Problem Size	No SP			With SP		
	<i>1 Bl</i>	<i>2 Bl</i>	<i>2 Bl + Pc</i>	<i>1 Bl</i>	<i>1 Bl + Pc + BRL</i>	<i>2 Bl + Pc + BRL</i>
small	13.7	15.0	11.0	8.1	4.8	4.6
large	14.7	15.1	11.2	9.4	5.6	4.6

4 Conclusions

NN classification has the significant drawback of requiring a large number of computations and data accesses which make it slow if the advantages that current computer architectures offer are not used to full advantage. Frequently, the byte data type has been used in an attempt to reduce the memory usage. In order to decrease the number of computations, an IF statement has often been added to the inner loop to test whether a better solution has already been found. In our experiments, this improved performance by a factor larger than 2. However, this is not the best solution available. Due to processor characteristics, the usage of floating-point arithmetic outperforms the use of integer arithmetic. The resulting machine code can run faster because the instruction level parallelism is higher and no data conversions are needed.

The disadvantage introduced by the use of floating-point data is the larger amount of memory used. This issue can be overcome easily by means of block algorithms. When these kinds of algorithms are used, the temporal locality of programs is better exploited resulting in low number of cache misses, allowing the computations to proceed at full speed. The use of simple floating-point data produces fewer misses than the use of double precision floating-point data due to better usage of spatial locality. However, the difference from the latter is almost negligible because of the reduced number of cache misses incurred when a block algorithm is used (see figure 3). In our experiments, the results obtained when a block algorithm and simple precision floating-point data are used are between 2 and 4 times faster than the algorithms which use integer arithmetic although they require 4 or 8 times more data storage. These results can be generalized for other superscalar architectures.

References

1. E. Anderson, J. Dongarra, LAPACK User's Guide, SIAM, Philadelphia, 1992.
2. P. Baglietto, M. Maresca, M. Migliardi, Image Processing on High-Performance RISC Systems, Proceedings of the IEEE, 84(7):917–930, July 1996.
3. S. Bandyopadhyay and U. Maulik Efficient prototype reordering in nearest neighbor classification, Pattern Recognition 35(12):2791–2799, Dec. 2002.
4. Z. Chi, J. Wu and H. Yan, Handwritten numeral recognition using self-organizing maps and fuzzy rules, Pattern Recognition, 28(1):59–66, 1995.
5. B.V. Dasarathy, Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques, IEEE Computer Society Press, 1991.
6. Digital Equip. Corp., DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors - Hardware Ref. Manual, 1994.
7. C. Decaestecker, Finding Prototypes for Nearest Neighbor Classification by Means of Gradient Descent and Deterministic Annealing, Pattern Recognition, 30(2):281–288, 1997.
8. A. Djouadi, E. Bouktache, A Fast Algorithm for the Nearest-Neighbor Classifier, IEEE Trans. on Pattern Analysis and Machine Intelligence, 19(3):277–282, 1997.
9. J. Fu, T.S. Huang, VLSI for Pattern Recognition and Image Processing, Springer-Verlag, Berlin, 1984.
10. P.J. Grother, G.T. Candela and J.L. Blue, Fast Implementation of Nearest Neighbor Classifiers, Pattern Recognition, 30(3):459–465, 1997.
11. Y. Harnamoto, S. Uchimura and S. Tornita, A Bootstrap Technique for Nearest Neighbor Classifier Design, IEEE Trans. on Pattern Analysis and Machine Intelligence, 19(1):73–79, 1997.
12. R. Van Der Heiden and F.C.A. Groen, The Box-Cox Metric for Nearest Neighbor Classification Improvement, Pattern Recognition, 30(2):273–279, 1997.
13. Hewlett Packard, PA-RISC 1.1 Architecture and Instruction Set Reference Manual, 1994.
14. T. Kohonen, The self-organizing map, Proc. of the IEEE 78(9):1464–1480, 1990.
15. M. Kudo, N. Masuyamaa, J. Toyamaa and M. Shimbob, Simple termination conditions for k-nearest neighbor method, Pattern Recognition Letters 24(9-10):1203–1213, June 2003.
16. M. Lam, Software Pipelining: An Effective Technique for VLIW Machines, Proc. of the SIGPLAN'88, pp 318–328.
17. M.S. Lam, E.E. Rothberg and M.E. Wolf, The Cache Performance and Optimizations of Blocked Algorithms, ASPLOS 1991, pp. 67–74.
18. E.W. Lee and S.I. Chae, Fast Design of Reduced-Complexity Nearest-Neighbor Classifiers Using Triangular Inequality, IEEE Trans. on Pattern Analysis and Machine Intelligence, 20(5):562–566, 1998.
19. C.-L. Liu, H. Sako, H. Fujisawa, Performance evaluation of pattern classifiers for handwritten character recognition, Int. J. on Document Analysis and Recognition 4:191–204, 2002.
20. J.J. Navarro, A. Juan and T. Lang, MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Computations, ACM Int. Conf. Supercomputing, 1994, pp. 354–363.
21. F. Ricci and P. Avesani, Data Compression and Local Metrics for Nearest Neighbor Classification, IEEE Trans. on Pattern Analysis and Machine Intelligence, 21(4):380–384, 1999.