SIMD OPTIMIZATION OF EUCLIDEAN DISTANCE TRANSFORMS FOR PATTERN RECOGNITION

B. Martín[1], A.B. Moreno[2], A. Sánchez[2] and E. Frías-Martínez[3]
[1] Facultad de Informática, Universidad Politécnica de Madrid, 28660 Boadilla (Madrid), Spain
bermmn@terra.es
[2] ESCET, Universidad Rey Juan Carlos, 28933 Móstoles (Madrid), Spain
{a.b.moreno, an.sanchez}@escet.urjc.es
[3] Computer Science Department, New York University, 715 Broadway Room 719, NY, NY 10003, USA
frias@cs.nyu.edu

Abstract

This paper describes a SIMD optimization method for computing different Euclidean distance algorithms. Distance transforms have been widely applied to image analysis and pattern recognition problems. The proposed approach is based on the inherent fine and medium-grain parallelism of considered distance algorithms and has been implemented using Intel Streaming SIMD Extensions (SSE), intrinsics and VTune Analyzer. Experimental results show that optimized prefetched SIMD algorithms improve by four the number of execution cycles in comparison with the initial SISD solutions.

Keywords: Euclidean distance transforms, SIMD optimization, intrinsics, image analysis, pattern recognition.

1. INTRODUCTION

Numerous applications of distance transforms in image analysis and pattern recognition have been reported in the literature [1][2]. Euclidean distance transforms have been deeply studied and applied in research areas like medical image processing [3], computer graphics [4] and biometric pattern recognition [5]. In general, these domains are highly demanding both in terms of execution time and memory requirements. The volume of data to be processed is often large, and many times the results are useful only if they are available in real-time. Of course, an important part of these demands are handled by hardware, but the use of software optimization tools combined with the development of improved algorithms for the considered computational problems is also very critical for the final optimized solution. Distance transforms algorithms, as needed in many image and pattern recognition problems, require such efficiency improvements.

In general, these transforms are based on different distance functions. Euclidean distance transforms, which make use of the Euclidean metric to compute distances, are preferred in many applications because they are well known from classical geometry. The advantage of computing exact Euclidean distances is the fact that it is intuitively obvious. However, this distance transform is often regarded as a highly time-consuming operation due to square root and to its non-integer value. Consequently, other approximations to the exact Euclidean distance transforms have been proposed, such as square Euclidean, Manhattan, chessboard and Chamfer distances [1].

Euclidean distance transforms have a linear time complexity in the number of pixels of the image (i.e. $O(mn)$, for a m by n digital image). Many sequential algorithms [6] and theoretical parallel algorithms [7] for computing Euclidean distance transforms have been proposed. Furthermore, architectures that implement those algorithms have been developed [8].

In this paper, a parallelization technique for computing Euclidean distance transforms in pattern recognition applications, using the streaming SIMD extensions of Intel processors is presented. The proposed optimization approach also considers the characteristics of distance algorithms in real pattern recognition problems (i.e. high number of feature vectors with many data components) and tunes the solution to the target processor architecture. Rather than completely optimizing the distance algorithms, only the most critical parts (mainly nested loops) are vectorized to exploit the inherent fine-grain parallelism [9]. This has been done using VTune Performance Enhancement Environment [10]. Experimental results show significant cycle time improvements of our proposed parallelization approach when computing Euclidean distance transforms.

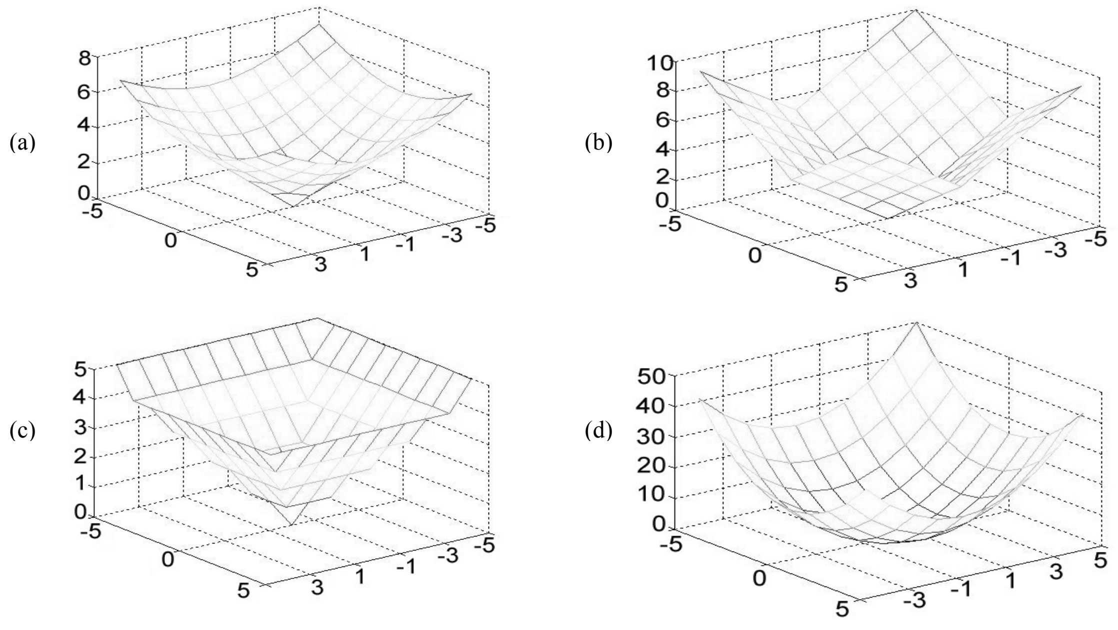


Figure 1. (a) Euclidean, (b) Manhattan, (c) chessboard and (d) square Euclidean distances.

2. EUCLIDEAN DISTANCE TRANSFORMS

In a 2D space, the *exact Euclidean distance* (D_E) between two points with co-ordinates (x_1, y_1) and (x_2, y_2) is:

$$D_E[(x_1,y_1),(x_2,y_2)] = \sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

To simplify the highly complex calculation due to square root, the *square Euclidean distance* (D_{SE}) can be computed as:

$$D_{SE}[(x_1,y_1),(x_2,y_2)] = (x_1-x_2)^2+(y_1-y_2)^2$$

The distance between two points can also be expressed as the minimum number of elementary steps to move from the starting point to the end point. Based on this idea other Euclidean distance transforms are usually defined. If only horizontal and vertical moves are allowed, the *Manhattan* or D_4 distance (also called *city-block*) is obtained:

$$D_4[(x_1,y_1),(x_2,y_2)] = |x_1-x_2|+|y_1-y_2|$$

If moves are also allowed in diagonal directions, *chessboard distance* or D_8 is formulated as:

$$D_8[(x_1,y_1),(x_2,y_2)] = \max(|x_1-x_2|,|y_1-y_2|)$$

A 3D graphical representation of these Euclidean distance transforms is shown in Figure 1. It can be seen that the surfaces defined by each distance transform are a sort of cones with sides that are radially symmetric. In many pattern recognition problems, the previous distance formulae are generalized for a *m*-dimensional space where *m* represents the pattern (feature) vector dimension.

3. PROPOSED OPTIMIZATION FRAMEWORK

This section outlines the issues taken into account to achieve the best performance for computing Euclidean distance algorithms using Intel streaming SIMD extensions. Figure 2 sketches the proposed fine and medium-grained parallelization environment.

The different distance transform algorithms are written in C in the form of nested loop structures and a preliminary high-level optimization is performed. We have used Intel C/C++ Compiler and Pentium III processor with SSE extensions to work with floating-point data. Source code analysis and tuning of the considered application has been done using VTune^{TM} Performance Analyzer (or VTune Analyzer) [10]. Next, we describe the elements of Figure 2.

3.1. Streaming SIMD Extensions

Pentium III processor from Intel is based on the P6 microarchitecture. This processor added new extensions to the AI-32 instruction set such as streaming Single Instruction Multiple Data (SIMD) Extensions (SSE) [11][12][13]. SIMD operations allow code developers to perform an identical operation on multiple pieces of data in parallel. SSE added 68 new instructions, including 45 new floating-point operations, 11 SIMD integer instructions, and 5 cache-management instructions [11]. The Pentium III pipeline is twelve

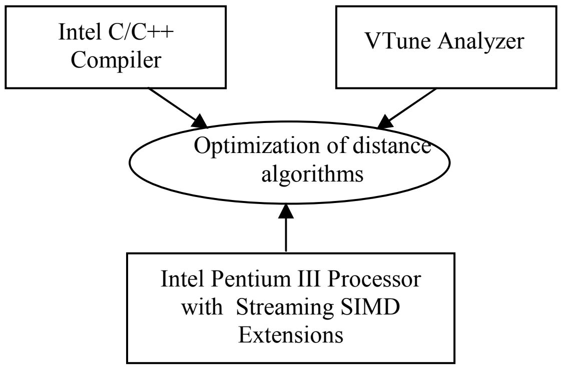

Intel C/C++ Compiler
VTune Analyzer
Optimization of distance algorithms
Intel Pentium III Processor with Streaming SIMD Extensions

Figure 2. Description of the optimization framework

stages long, and is a three way superscalar superpipeline. Out-of-order execution capability allows the concurrent execution of several instructions by considering the data dependencies and availability of resources. Data type available for SIMD-SSE is shown in Figure 3. It consists of a 128-bit packed floating point register which allows for four 32-bit floating point numbers to be packed into it. Other considered issues for optimization with the streaming SIMD Extensions are: SIMD and memory interactions, prefetch instructions, and optimizing memory use for array of data.

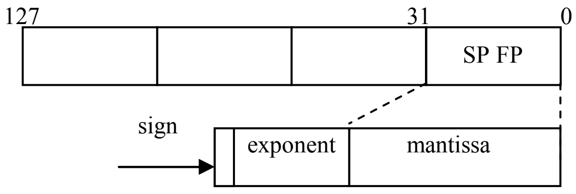


Figure 3. *SSE* data type.

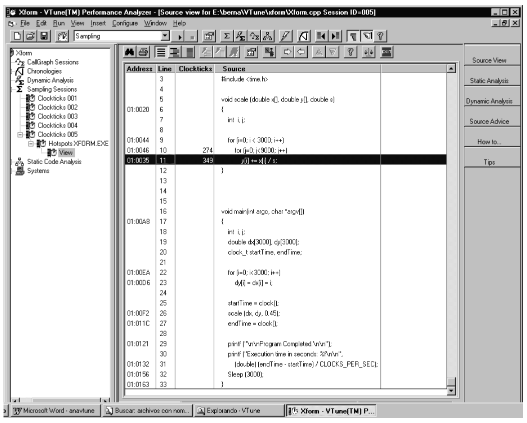
3.2. VTune Analyzer

VTune Analyzer [12] is a tool that gives a user a graphical view of the performance of an application, and it offers feedback on how to tune it. VTune Analyzer incorporates different code analysis techniques designed to optimize the source code: Sampling Analysis, Static Code Analysis, Dynamic Analysis, Code Coach, Call Graph Profiling and 'Chronologies'. Event-based sampling analysis is the most commonly used method for analyzing application performance. It allows to display the execution time of considered instructions to detect program bottlenecks. Figure 4 shows a view of VTune's Sampling Analyzer. The basic method of using Dynamic Analysis is to select a region of the code to be simulated.

Then, VTune can indicate the number of cache misses in a loop and can suggest the use of prefetching.

3.3. Intel C/C++ Compiler

Intel C/C++ Compiler [14] is a highly-optimized compiler. It offers several options for programmers to



Xform - VTune(TM) Performance Analyzer - [Source view for E:\berna\VTune\xform\Xform.cpp Session ID=005]

```
Address  Line  Clockticks  Source
         3                 #include <time.h>
         4
         5                 void scale (double x[], double y[], double s)
01:0020  6                 {
         7                    int  i, j;
         8
01:0044  9                    for (i=0; i < 3000; i++)
01:0046  10         274          for (j=0; j<9000; j++)
01:0035  11         349             y[i] += x[i] / s;
         12                }
         13
         14
         15
         16                void main(int argc, char *argv[])
01:00A8  17                {
         18                   int  i, j;
         19                   double dx[3000], dy[3000];
         20                   clock_t startTime, endTime;
         21
01:00EA  22                   for (i=0; i<3000; i++)
01:00D6  23                      dy[i] = dx[i] = i;
         24
         25                   startTime = clock();
01:00F2  26                   scale (dx, dy, 0.45);
01:011C  27                   endTime = clock();
         28
01:0121  29                   printf ("\n\nProgram Completed.\n\n");
         30                   printf ("Execution time in seconds: %f\n\n",
01:0132  31                      (double) (endTime - startTime) / CLOCKS_PER_SEC);
01:0156  32                   Sleep (3000);
01:0163  33                }
```

Figure 4. *VTune Analyzer*: source code

use Streaming SIMD Extensions: inlined assembly, intrinsics, vector class libraries, and vectorization. We have mostly used intrinsics to optimize Euclidean distance algorithms. *Intrinsics* are C-like function calls for which compiler generates inlined code. Each intrinsic map to a specific SSE (or MMX) instruction [10].

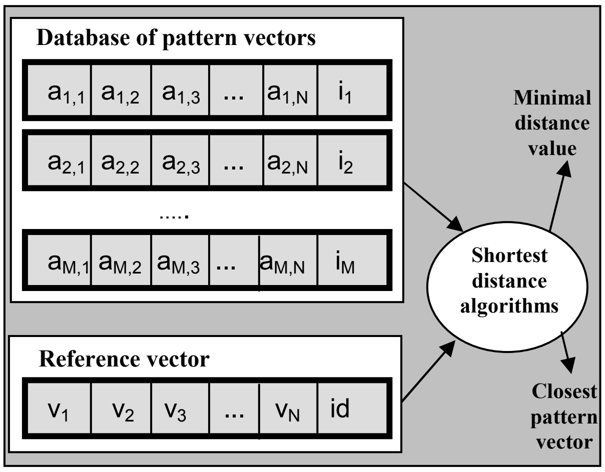
The use of intrinsics introduces a data definition according to Figure 3. The data type is __m128. The most common intrinsics are:

- __m128 mm_load_ps(float *mem),
  Loads a __m128 variable from memory.
- __m128 mm_add_ps(__m128 x, __m128 y),
  Adds two __m128 values
- __m128 mm_store_ps(float *mem, __m128 x),
  Stores a __m128 variable in memory.

4. RECOGNITION BASED ON EUCLIDEAN DISTANCE TRANSFORMS

As an example of pattern recognition application requiring real-time computations, we used for our experiments data extracted from 3D facial meshes captured with a 3D range digitizer. The digitizer provides a large 3D point density of scanned face surfaces which are illumination independent. After a preprocessing stage, most salient facial points are accurately computed (i.e. nasion, pronasale, etc.), and some local measures related to these 3D facial points are also obtained (i.e. median and Gaussian curvatures). With these data computed for each 3D facial surface, a database of normalized pattern (feature) vectors is defined [15].

The considered pattern recognition problem, as shown in Figure 5, consists in determining the most similar feature database vector with respect to a reference vector (computed in a similar way from an unknown captured 3D face mesh), and the corresponding distance value.



Database of pattern vectors
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ ... $a_{1,N}$ i_1
$a_{2,1}$ $a_{2,2}$ $a_{2,3}$ ... $a_{2,N}$ i_2
.....
$a_{M,1}$ $a_{M,2}$ $a_{M,3}$ ... $a_{M,N}$ i_M
Reference vector
v_1 v_2 v_3 ... v_N id
Shortest distance algorithms
Minimal distance value
Closest pattern vector

Figure 5. High level diagram of the considered recognition problem.

It should be pointed out that the proposed optimization method can be applied to any pattern recognition problem using Euclidean distances.

Let *M* be the number of feature vectors in the database and *N* the number of fixed features (characteristics) per vector. Each feature vector also contains an identification field, named i_k ($k \in [1..M]$), which not takes part in the distance computation algorithm but allows an easy identification of a given feature vector.

Similarity between two vectors is computed via the usual Euclidean distance transforms: exact Euclidean, square Euclidean, Manhattan and chessboard, respectively.

Two versions for each distance algorithms are implemented: *non-weighted distance algorithms* which means that all features are equally important, and *weighted distance algorithms* which assign different weights to features according to their discriminate power. Figure 6 sketches the initial C source code for solving the recognition problem using non-weighted Manhattan distances. Figure 7 shows the SIMD optimization of this previous algorithm which consists in computing four distance features in parallel using SSE extensions and intrinsics.

```
for(i=0;i<M1;i++)
  dist_cerc+=fabs(vec[i]-lista_vector[0][i]);
pos=1;
while (pos<N){
  dist=0.0;
  for(i=0;i<M1;i++)
    dist+=fabs(vec[i]-lista_vector[pos][i]);
  if (dist<dist_cerc){
    dist_cerc=dist;
    pos_enc=pos;
   }
  pos++;
}
```

Figure 6. *C* pseudocode for non-weighted Manhattan distance.

```
#include "xmmintrin.h"
....
cte[0]=0x7fffffff; cte[1]=0x7fffffff; cte[2]=0x7fffffff; cte[3]=0x7fffffff;
....
pos=1;
while (pos<N){
  *(__m128*)aux=_mm_and_ps(_mm_sub_ps(
    *(__m128*)&vec[0],*(__m128*)&lista_vector[pos][0]),*sse_cte);
  *(__m128*)aux2=_mm_and_ps(_mm_sub_ps(
    *(__m128*)&vec[4],*(__m128*)&lista_vector[pos][4]),*sse_cte);
  *(__m128*)aux3=_mm_and_ps(_mm_sub_ps(

*(__m128*)&vec[8],*(__m128*)&lista_vector[pos][8]),*sse_cte);
  *(__m128*)aux4=_mm_and_ps(_mm_sub_ps(

*(__m128*)&vec[12],*(__m128*)&lista_vector[pos][12]),*sse_cte);

*(__m128*)aux5=_mm_add_ps(*(__m128*)aux,*(__m128*)aux2);

*(__m128*)aux6=_mm_add_ps(*(__m128*)aux3,*(__m128*)aux4);

*(__m128*)aux7=_mm_add_ps(*(__m128*)aux5,*(__m128*)aux6);
  dist=aux7[0]+aux7[1]+aux7[2]+aux7[3];
  ....
  pos++;
}
```

Figure 7. Fine-grain parallelization on non-weighted Manhattan distance (I).

Once calculated the distances between each corresponding pair of features from any database vector and reference vector, it is necessary to determine the total distance value between these two vectors by adding all partial distance values. This is also done in parallel by means of the following intrinsic instruction:

```
*sse_aux3=_mm_add_ps(*sse_aux,*sse_aux2);
```

5. EXPERIMENTAL RESULTS

To test our proposed optimization approach for computing Euclidean distance transforms, we used different number of vectors and features. Let *M* be the number of feature vectors in the database and *N* the number of fixed features (characteristics) per vector.

Three different problem instances have been considered: problem *P1*, where $M = 100$ and $N = 16$; problem *P2*, where $M = 200$ and $N = 32$; and problem *P3*, where $M = 16$ and $N = 100$. Feature values are 32-bit floating-point values in the range [0.0..100.0] in millimetres. Figure 8 represents the total number of clock cycles for non-weighted distance algorithms for problem *P1*: (1) exact Euclidean, (2) Manhattan, (3) chessboard and (4) square Euclidean, respectively. For each distance algorithm, three execution times have been measured: SISD; SIMD and SIMD using L1 data cache (prefetch), respectively.

Figure 9 is similar to Figure 8, but now the corresponding weighted distance algorithms are considered.

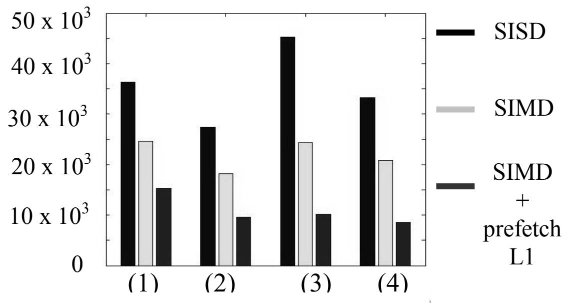


Figure 8. Clock cycles of non-weighted distance algorithms for problem *P1*.

From the two previous figures, we can deduce that best execution results are achieved using square Euclidean and Manhattan distance transforms. Another consequence of the experiments is that optimized SIMD with prefetch solution approximately reduces by four the number of clock cycles with respect to the corresponding sequential SISD solution.

To estimate the average error of different distance algorithms with respect to the exact Euclidean distance, the following error formula is used:

$$E_{MS}^{d} = \frac{1}{M} \cdot \sum_{j=1}^{M} \left(x_j^d - x_j^E \right)^2$$

where M represents the number of pattern vectors in the database, x_j^E is the exact Euclidean distance between vector j and reference vector, and x_j^d is a similar distance value but considering other distance transforms (square Euclidean, Manhattan and chessboard, respectively).

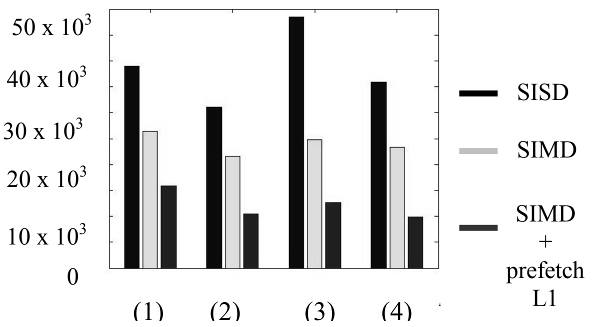


Figure 9. Clock cycles of weighted distance algorithms for problem *P1*.

Table 1 shows the number of clock cycles and estimated errors for each distance transform using the considered problems *P1*, *P2* and *P3*, respectively. A consequence of the results in Table 1 is that a better approximation distance to exact Euclidean distance requires a higher execution time. In the case when the number of features per vector increases (in problem *P3*), a trade-off solution is the use of chessboard distance transform algorithm which offers best ratio between execution time and average error with respect to exact Euclidean distance.

	Square Euclid.		Manhattan		Chessboard		Exact Euclid.
	Cycles	Error	Cycles	Error	Cycles	Error	Cycles
P1	8.6	$7 \cdot 10^8$	9.6	$1 \cdot 10^5$	10.1	8600	15.3
P2	32.5	$3 \cdot 10^9$	39.2	$7 \cdot 10^5$	43.4	24000	42.3
P3	8.7	$3 \cdot 10^{10}$	8.6	$8 \cdot 10^6$	7.5	10^5	9.5

Table 1. Clock cycles and distance errors for distance formulas and Problems P1, P2 and P3.

6. CONCLUSIONS

In this paper, we have presented a parallelization method based on Streaming SIMD Extensions for computing Euclidean distance transforms. Our approach provides a systematic methodology for optimizing many applications where exploitation of fine and medium-grain SIMD parallelism is needed. Euclidean distance transforms have been properly tuned to the considered final architecture using the proposed SIMD optimization environment. The number of cycles per instruction when computing distances has been approximately reduced by four in the parallel SIMD solution using data prefetch in L1 cache with respect to the corresponding sequential SISD algorithm. Future work includes the parallelization of Chamfer distance transform using the proposed approach. Migration to Pentium 4 architecture using SSE2 extensions for the considered distance algorithms is now in progress.

REFERENCES

[1] M. Sonka, V. Hlavak and R. Boyle, *Image Processing, Analysis, and Machine Vision* (Pacific Grove, CA: PWS Publishing, 1999).
[2] R.C. Gonzalez y R.E. Woods, *Digital Image Processing* (Reading, MA: Addison Wesley, 1993).
[3] O. Cuisenaire, *Distance Transforms: Fast Algorithms and Applications to Medical Image Processing*, PhD Thesis, Université Catholique de Louvain, Belgium, 1999.
[4] J. Foley, A. Van Dam, S. Feiner and J. Hughes, *Computer Graphics: Principles and Practice* (Reading, MA: Addison Wesley, 1990).
[5] V. Starovoitov, and D. Samal, A geometric approach to face recognition, *Proc. of the IEEE-EURASIP Workshop on Nonlinear Signal and Image (NSIP'99)*, v. *2*, 1999, 210-213.
[6] G. Borgefors, "Distance Transformations in Arbitrary Dimensions", *Computer Vision, Graphics and Image Processing*, v. *27*, 1984, 321-345.
[7] Y.-H. Lee and S.-J. Horng, "Fast Parallel Chessboard Distance Transform Algorithms", *Proc. 1996 Intl. Conf. on Parallel and Distributed Systems*, 1996, 488-493.

[8] D.W. Paglieroni, “A Unified Distance Transform Algorithm and Architecture”, *Machine Vision And Applications,* 1992, 47-55.
[9] K.K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation* (New York, NY: J. Wiley & Sons, 1999).
[10] Intel Corp., *Intel Architecture Optimization*, Intel Corporation, 1999.
[11] J. Keshava and V. Pentkovski, “Pentium III Processor Implementation Tradeoffs”, *Intel Technology Journal,* Intel Corporation, 1999.
[12] J.H. Wolf III, “Programming Methods for the Pentium III Processor’s Streaming SIMD Extensions using the VTune Performance Enhacement Environment”, *Intel Technology Journal,* Intel Corporation, 1999, pp. 1-11.
[13] J. Abel et al., “Applications Tuning for Streaming SIMD Extensions”, *Intel Technology Journal,* Intel Corporation, 1999.
[14] Intel Corp., *Intel C/C++ Compiler User’s Guide*, Intel Corporation, 1999.
[15] A.B.Moreno, A. Sánchez and J.F. Vélez, “Automatic Location of 3D Feature Points in Facial Meshes”, *Proc. IASTED Intl. Symp. on Applied Informatics*, Acta Press, 2001, 571-576.