

# Addressing Real-Time Requirements of Automatic Vehicle Guidance with MMX Technology <sup>\*</sup>

Massimo Bertozzi<sup>1</sup>, Alberto Broggi<sup>2</sup>, Alessandra Fascioli<sup>1</sup>, Stefano Tommesani<sup>1</sup>

<sup>1</sup> Dipartimento di Ingegneria dell'Informazione  
Università di Parma, I-43100 Parma, Italy  
bertozzi, fascioli, tommesa@ce.unipr.it

<sup>2</sup> Dipartimento di Informatica e Sistemistica  
Università di Pavia, I-27100 Pavia, Italy  
broggi@dis.unipv.it

**Abstract.** This paper describes a study concerning the impact of MMX technology in the field of automatic vehicle guidance. Due to the high speed a vehicle can reach, this application field requires a very precise real-time response. After a brief description of the ARGO autonomous vehicle, the paper focuses on the requirements of this kind of application: the use of only visual information, the use of low-cost hardware, and the need for real-time processing. The paper then presents the way these problems have been solved using MMX technology, discusses some optimization techniques that have been successfully employed, and compares the results with the ones of a traditional scalar code.

## 1 Introduction

During the last years, a large number of research institutes worldwide have been involved in national and international projects about *Automatic Vehicle Guidance*, namely the techniques aimed at automating –entirely or partially– one or more driving tasks. The automation of these tasks carries a large number of benefits, such as: a higher exploitation of the road network, lower fuel and energy consumption, and –of course– improved safety conditions compared to the current scenario.

As a result, a number of prototypes of vehicles embedding systems for Automatic Vehicle Guidance have been designed, implemented, and tested on the road [3]. For being sold on the market, these systems needs to fit three requirements.

1. Since it is expected that an increasing number of vehicles will be equipped with these facilities, the sensors for data acquisition must not interfere with the ones used on other vehicles. This suggests the use of passive sensors, that do not alter the environment; in this case, the *processing of visual information* plays a fundamental role.

---

<sup>\*</sup> The work described in this paper has been carried out under the financial support of the Italian *Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST)* in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project and the financial support of the *CNR Progetto Finalizzato Trasporti* under contracts 93.01813.PF74 and 93.04759.ST74.

2. The system must acquire data and produce results in *real-time*, since the vehicle maximum speed is determined by the response time of the system. Therefore a high performance computing engine is needed.
3. On the other hand, the complete system must have a sufficiently *low cost* to allow a widespread use in the car market.

These requirements have been used in developing a prototype system [3] installed on the ARGO autonomous vehicle (figure 1). The ARGO capabilities have been demonstrated to the international scientific community during the first week of June 1998, when ARGO drove itself autonomously for more than 2000 km along the Italian highway network. The percentage of automatic driving was about 95%, up to a maximum speed of about 120 km/h.

The ARGO's system is based on a low-cost computational engine, a standard PC, which processes data coming from a pair of videocameras in real-time.

The needs for both low-cost hardware and real-time image processing as well as the recent enhancements (MMX) of the x86 processors family dedicated to the efficient handling of audio and video data [4, 5] suggest the exploitation of MMX technology and the design of highly optimized algorithm implementations.

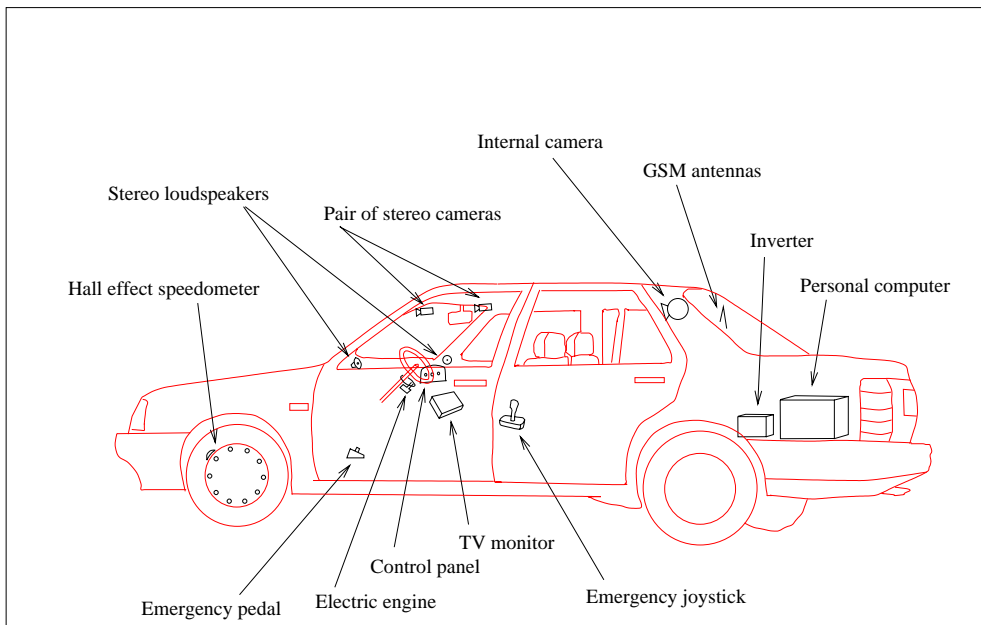
This work presents the preliminary results of a study about the use of the MMX technology on intelligent vehicles applications. Particularly, the low level portion of the processing aimed at the detection of obstacles [1] has been implemented exploiting the MMX features. The effectiveness of this implementation makes the whole system very responsive, requiring less than one millisecond to perform the low and middle level portion of obstacle detection.

This paper is organized as follows: the next paragraph introduces a brief description of the Obstacle Detection algorithm used on the ARGO vehicle giving emphasis to the low- and middle-level portions of the processing, since they gain the most benefit from the MMX use, section 3 sketches the MMX programming/optimizing issues, while section 4 presents the MMX implementation of the algorithm. Section 5 ends this work with some concluding remarks.

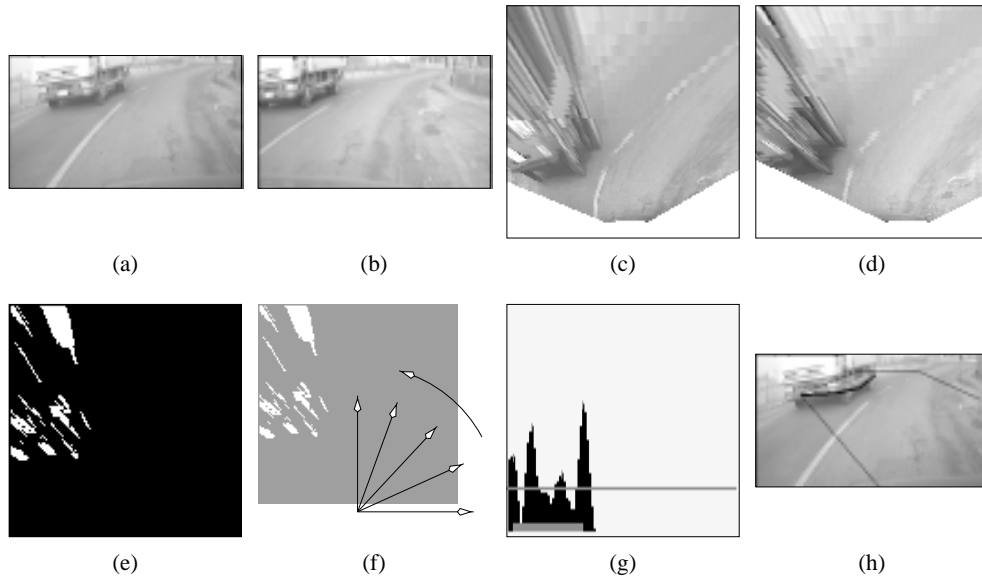
## 2 The Obstacle Detection Algorithm

The Obstacle Detection algorithm used on board the ARGO vehicle is based on the *Inverse Perspective Mapping*, IPM [2], geometrical transform widely used in this kind of applications [6–8]: once known all the acquisition intrinsic and extrinsic parameters, the IPM can be used to remap the acquired images in a new domain, called *road domain*, that represents a bird's eye view of the road surface.

The use of the IPM transform on pairs of stereo images (figs. 2.a and 2.b) allows to obtain two patches (*remapped images*) of the road texture (figs. 2.c and 2.d) which can be brought to correspondence exploiting the knowledge of the vision system setup. Any difference in the two remapped images represents a deviation from the starting hypothesis of flat road and thus identifies a potential obstacle, namely anything rising up from the road surface.



**Fig. 1.** The ARGO autonomous vehicle and its equipment



**Fig. 2.** Obstacle detection: (a) left and (b) right stereo images, (c) and (d) the remapped images, (e) the difference image, (f) the angles of view overlapped with the difference image, (g) the polar histogram, and (h) the result of obstacle detection using a black marker superimposed on the acquired left image; the thin black line highlights the road region visible from both cameras

An obstacle is detected when the **difference image**, obtained comparing the two remapped images, presents sufficiently large clusters of non-zero pixels having a specific shape. Thus, the low-level portion of the processing consists in the computation of the difference between the remapped images and to the binarization of the resulting image followed by a **noise filtering** to remove isolated pixels.

Due to the different angles of view of the stereo system, the vertical edges of an obstacle generate two triangles in the difference image. Unfortunately, due to their texture, irregular shape, and non-homogeneous brightness, real obstacles produce triangles that are not so clearly defined; nevertheless in the difference image some clusters of pixels having a quasi-triangular shape are anyway recognizable (see fig. 2.e). The obstacle detection process thus is reduced to the localization of pairs of these *triangles*.

A **polar histogram** is used for the detection of triangles: it is computed scanning the difference image with respect to a focus (fig. 2.f) and computing the number of overthreshold pixels for every straight line originating from the focus. The histogram is normalized and low-pass filtered (fig. 2.g). The position of a peak within the histogram determines the angle of view under which the obstacle edge is seen.

A further analysis of the difference image along the directions pointed out by the maxima of the polar histogram allows the determination of obstacle distance [1]. Figure 2.h show the result displayed with a black marker superimposed on a brighter version of the left image.

### 3 Implementation with MMX Technology

All low-level plus some middle level computations of the discussed algorithm have been implemented exploiting the MMX enhancements of the x86 processors family.

The main characteristics that have been considered in this new implementation are:

- the processing of multiple data according to the SIMD computational model (*data parallelism*), and the use of a specific instruction set dedicated to Image Processing;
- the careful exploitation of the *instruction level parallelism* of the code and the use of the two internal pipelines;
- the reduction of the amount of data in order to fit into the internal processor cache.

The complete obstacle detection process can now be performed in less than 20 ms (more precisely it takes 5 ms), which is the time required to acquire a single image field from a conventional camera. Hence, the whole processing is performed in real-time (at frame rate) during image acquisition, which becomes the system bottleneck.

The next paragraphs introduce the MMX optimizations topic; for a more detailed discussion see also [9].

#### 3.1 MMX Optimization Issues

Major performance gains can be attained with the fastest algorithms and the most efficient data structures, so every code optimization attempt should start at the highest level of abstraction, focusing on more streamlined procedures. Low level code optimization must be used when the algorithms are fully tested, and no more high level optimizations can be applied. Low level optimizations generally produce code that cannot be easily modified, so it is useless to optimize code that could change soon.

We will focus this discussion on a pair of strategies that can easily be implemented and that guarantee the highest performance payoff:

**Data alignment:** a misaligned access in the data cache or on the bus costs at least three extra clock cycles on the Pentium processor. Given that an aligned memory access might take only 1 cycle to execute (assuming a L1 cache hit), this is a serious performance penalty.

Data alignment is critical when writing memory-bounded MMX code: the execution speed can be boosted by more than 30% by using an efficient memory allocator.

**Instruction pairing:** the Pentium processor is an advanced superscalar processor: it is built around two general-purpose integer pipelines and a pipelined floating-point unit, allowing the processor to execute two integer instructions simultaneously. The Pentium processor can issue two instructions every clock cycle, one in each pipe. The first logical pipe is referred to as the *U-pipe*, and the second as the *V-pipe*. During decoding of any given instruction, the next two instructions are checked, and, if possible, they are issued such that the first one executes in the U-pipe and the second in the V-pipe. If it is not possible to issue two instructions, then the next instruction is issued to the U-pipe and no instruction is issued to the V-pipe.

Therefore the execution of two MMX instructions in a single clock cycle might double the performance of the code by keeping both pipes busy. Unfortunately,

this is not always possible: i.e. two shift instructions cannot be allocated on the same clock cycle, because the Pentium has only one shifter unit. In addition not all instructions can be routed to both pipes indifferently: table 1 shows the number of functional units and whether they are addressable from the U or the V-pipe.

Operation	Number of Functional Units	Latency	Throughput	Execution Pipes
ALU	2	1	1	U and V
Multiplexer	1	3	1	U or V
Shift/pack/unpack	1	1	1	U or V
Memory access	1	1	1	U only
Integer register access	1	1	1	U only

**Table 1.** number of functional units and their accessibility from the U and V-pipes

- Two MMX instructions which both use the MMX shifter unit (pack, unpack, and shift instructions) cannot pair since there is only one MMX shifter unit, although they may be issued in either the U-pipe or the V-pipe.
- Two MMX instructions which both use the MMX multiplier unit (pmull, pmulh, pmadd type instructions) cannot pair since there is only one MMX multiplier unit. Multiply operations may be issued in either the U-pipe or the V-pipe but not in both in the same clock cycle.
- MMX instructions which access either memory or the integer register file can be issued in the U-pipe only. Their scheduling in the V-pipe causes a wait and a swap to the U-pipe.

### 3.2 Tuning MMX code

This section presents the *Loop Interleaving* technique heavily exploited in this project. MMX code usually consists of a short loop iterated many times over an array of data. Often there is no relationship between two iterations of the same loop, so the output of the second iteration does not depend on the output of the first. In fact, they may be executed in any order (even at the same time). The first step is designing the algorithm so that it occupies the lowest number of MMX registers: there are only 8 of them, so they limit the number of iterations that can be executed at the same time. MMX registers that are shared between the iterations (for example, bit masks) should be counted only once. Having calculated the highest number of iterations that can be executed in parallel ( $N$ ), the loop can be unrolled by  $N$  and then instructions can be moved around to maximize the use of both pipelines:

- MMX instructions that access memory are always executed in the U-pipe, so it is better to spread them around and try to exploit the V-pipe with ALU or shift instructions.
- Two shift instructions cannot be issued in the same cycle, but all shifts of the different iterations will end up very near, because they belong to the same stage of the loop. This problem can be solved by introducing a slight delay between

the iterations, so that the group of MMX instructions that are physically near belong to different stages of their respective iterations. This is best suited to long loops, because in short loops the overhead given by the bunch of instructions belonging to the same iteration (which usually do not pair) situated at the top of the loop may offset the benefits of a greater V-pipe exploitation.

## 4 Implementing the Obstacle Detection Algorithm with MMX

This section describes the MMX implementation of each low and medium level routine for obstacle detection and its performance analysis.

### 4.1 Computation of the Difference Image

The idea is to compute the absolute difference between the left and the right image, set pixels below a given threshold to zero, then pack each pixel using only one bit per pixel.

The absolute difference can be easily implemented computing two saturated [4] differences and OR'ing them:

```
MOVQ    MM2, MM0 ; make a copy of MM0
PSUBUSB MM0, MM1 ; compute saturated difference one way
PSUBUSB MM1, MM2 ; compute saturated diff. other way
POR     MM0, MM1 ; OR them together
```

The threshold is loaded into register MM7 at startup. Pixels below the threshold are set to zero using a further saturated subtraction, then one bit is extracted from the result, which will be 1 only if its byte is above the threshold.

Since the image is  $128 \times 128$ , the use of one bit per pixel shrink the output data down to 2 kbyte allowing a more efficient use of the Pentium's L1 cache memory. Additional benefits of this choice will be shown in the following. Unfortunately this packing process requires quite a lot of instructions to be implemented, and an additional phase of bit swapping.

To minimize the performance hit, the *packing process* works on 32 bits word on each loop extracting a bit from each saturated byte (namely a single pixel) and then a *bit swapping* process rearranges two 32 bits words into a single double word (64 bits). The Loop Interleaving technique has been applied by running two iterations in parallel, and the resulting dual pipeline exploitation ratio is quite high.

The bit swapping code is simple: it applies a mask to isolate all the bits with index  $i$  then copies them into position  $7 - i$  (where  $i$  is between 0 and 7). The bit swapping can be highly optimized by using only three MMX registers and thanks to a round-robin policy: many flow dependencies can be avoided. Careful instruction scheduling is required due to the limitations on the use of shift instructions: only one shift operation can be issued in each clock cycle.

### 4.2 Noise Filtering

This routine removes isolated pixels from the difference image. A pixel  $\bullet$  is defined as isolated if all  $\circ$  pixels are zero as shown in the following diagram:

o		o
o	•	o
o		o

The highly packed image format described in the previous paragraph gives more benefits now: this small chunk of data (2 kbyte) fits comfortably into the Pentium MMX's 16 kbyte L1 cache, avoiding slow memory accesses on the system bus. Even more important, the high degree of parallelism allows to process 64 pixels each loop, since each pixel is represented by a bit. This powerful combination, plus some optimized coding, make this process extremely fast.

A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3

The target is to evaluate the pixels in the second row ( $A2 \rightarrow D2$ ). Applying the rule explained before B2 is isolated where A1, A2, A3 and C1, C2, C3 are all set to zero. A new row is computed by joining all three rows with the OR operator:

A1+A2+A3	B1+B2+B3	C1+C2+C3	D1+D2+D3
----------	----------	----------	----------

this new row is copied twice and shifted once left and once right by one position:

0	A1+A2+A3	B1+B2+B3	C1+C2+C3
A2	B2	C2	D2
B1+B2+B3	C1+C2+C3	D1+D2+D3	0

All that remains to do is joining the first and the third row with the OR operator, saturate the result and then AND'ing the second row with the resulting row. The B2 cell will contain the B2 value only if at least one pixel among A1, A2, A3 and C1, C2, C3 is not zero. This routine can also gain advantage from the Loop Interleaving optimization: the difference image has a width of 128 pixels, which can be packed into two MMX registers. Unrolling the loop by two, a single row in each iteration can be analyzed (actually these two loops are not completely independent, due to the pixels on the edges of the central column that must be exchanged between the loops).

This routine performs very well, with an high instruction pairing ratio and a very good memory access pattern. The difference image's size (2 kbyte) fits perfectly into the L1 cache, therefore most memory accesses do not use the system bus. Given these assumptions, each loop takes 17 CPU cycles to execute, and the whole routine process takes about 2200 cycles. Actually in execution benchmarks it took slightly more than 3000 cycles to execute: this is probably due to L2 cache memory access. According to these numbers, each pixel in the difference image is checked and eventually removed in only 0.19 cycles.

### 4.3 Computation of the Polar Histogram

This routine builds the polar histogram counting all non-zero pixels on each ray, ranging from 40 to 140 degrees with 1 degree increment and storing the results into an array. It is really hard to design a parallel algorithm that performs this task, and the simplest and fastest way to solve it is using a lookup table. We have not exploited the parallel capabilities of MMX while developing this routine, nonetheless smart coding speeds up the resulting program compared to non-MMX code: the additional 8 MMX registers help reduce register pressure, and MMX shifts are much faster than integer ones. The lookup table is pre-calculated and it is stored with the layout shown in table 2

Number of internal loops ( $N_1$ ) of first angle	Internal loop block (repeated $N_1$ times)	...	Number of internal loops ( $N_n$ ) of last angle	Internal loop block (repeated $N_n$ times)	End of loop (zero internal loops)
---	--	-----	--	--	-----------------------------------

**Table 2.** Layout of the lookup table for polar histogram computation

Address of first memory block	First bit to isolate	Address of second memory block	Second bit to isolate	Address of third memory block	Third bit to isolate
-------------------------------	----------------------	--------------------------------	-----------------------	-------------------------------	----------------------

**Table 3.** Internal loop block for polar histogram computation

	Average loop (clocks) %	Startup loop (clocks) %
Difference Image Computation	43177 32.6	83229 36.8
Noise Filtering	3054 2.3	4376 1.9
Polar Histogram Computation	84867 64.2	136645 60.4
Histogram Normalization	463 0.4	970 0.4
Histogram Average Computation	716 0.5	1041 0.5
Total	132277	226261
Execution time at 200 MHz (ms)	0.661	1.131

**Table 4.** execution times of the MMX routines

where the number of internal loops is stored as

Number of internal loops (4 bytes)	Padding (4 bytes set to zero)
------------------------------------	-------------------------------

and the internal loop block as shown in table 3.

In addition Loop Interleaving technique has been applied again to compute 3 pixels in each loop; an higher level of parallelism may give further performance gains, but we must consider that the number of pixels in each line will be an integer multiple of the unrolling factor, so when using large numbers the overhead due to padding pixels may offset the benefits of longer loops.

Experimental tests showed that both pipes are fully used most of the time. Some optimizations that may not appear obvious are discussed below:

- Memory accesses must be executed in the U-pipe. Each iteration reads 6 quad words: they are three groups of bit mask plus image map accesses. The bit mask quad word defines the block of the image map to read and the position of the desired bit into the block. This data layout wastes some memory, because only 5 bytes are needed to store this piece of information, but for alignment reasons this number has been rounded up to 8 bytes. We have tried to pack all 3 bit positions into a single double word, but the additional computations offset the reduced number of memory accesses. We can reasonably assume that the image map will be stored into the Pentium's L1 cache memory, being small (only 2 kbyte) and recently accessed (it was fully scanned). The position of the bit mask array into the memory hierarchy may vary: on systems with a large L2 cache memory it may still be there, as a remainder of the previous execution, but it could also be in the main memory. The location of the bit mask array is the most important factor when evaluating the performance of

this routine, as the worst case can bring a 70% speed penalty compared to the best case.

- Memory addressing: the address into the image map is extracted from a MMX register and put into an integer register to access memory. This process can cause additional penalties: at least one CPU cycle must pass between the writing of the address into the integer register and the reading from that address. This problem was solved by interleaving another memory access that uses a different integer register: by using two kinds of data (bit masks and image map blocks) we can interleave their memory accesses and avoid the penalty explained above.
- Integer register access: all instructions that access integer registers must be issued to the U-pipe. Two penalties are due to this limitation.
- Loop delay: the three iterations are delayed, so that their shift instructions (which cannot be paired) are located in different zones of the code. There are no penalties due to shift instructions or register dependencies.

In execution benchmarks this routine took slightly more than 100,000 cycles to run when the bit mask array was located into the L2 cache, and slightly less than 170,000 cycles when it was in the main memory. This difference is striking, and shows that memory optimizations with modern CPU's are much more important than they used to be, as they can offer a big payoff, even better than the ones that can be obtained with coding techniques such as instruction scheduling.

Since the performance bottleneck of the previous algorithm is reading the bit mask table from slow memory, a simple way to improve the running speed is shrinking the bit mask table: two adjacent pixels that belong to the same angle can be encoded in a single table entry, using only one shift index. The bit mask table is then split in two parts: the first one is a list of pixel duos, the latter a list of single pixels. In real world benchmarks, this algorithm shrinks the bit mask table to 68808 bytes down from 100466 bytes, and reduces the execution time by 19%, down to 84867 cycles.

Each value in the histogram is the number of non-zero pixels that belong to that angle. However, each angle spans a different area (see figure 2), so if we want the ratio between non-zero pixels and all pixels contained in that area, we will have to scale each value of the resulting histogram. (the scaling factors are computed at startup). In this case the running time is so short (463 cycles on average) that specific optimizations are not worth applying.

Each value contained in the histogram is averaged to smooth spikes. The following formula was implemented (the division by 2 or 4 can be coded as a shift right instruction):

$$\text{Average}(V_i) = \frac{\frac{V_{i-2} + V_{i+2}}{2} + V_{i-1} + V_i + V_{i+1}}{4} \quad (1)$$

Also this routine's running time is so short (slightly more than 700 cycles) that we cannot see the point in further optimizations.

## 5 Conclusions

This paper presented a porting of the obstacle detection algorithm used on board of the ARGO autonomous vehicle toward the MMX technology.

The discussed implementation fully satisfies real-time requirements, being able to complete all computations in less than one millisecond.

The major performance hit is given by an empty L2 cache, as table 4 shows: on the startup loop all data lies in the main memory, while on the average loop it is split between L1 and L2 cache memories. The startup condition is more likely in real world execution, because other routines used on the ARGO vehicle (i.e. lane detection and perspective removal) use a lot of memory therefore flushing the data out of cache memories.

No automatic code parallelization or optimization was made exploiting the MMX enhancements: the complete code was written in assembly language by hand and the optimization was done after a careful analysis of the specific instructions sequence. This hard and time-consuming task has been rewarded with an enormous speed-up of the processing: the new MMX-based code runs about 26 times faster than its scalar C version.

## References

1. M. Bertozzi and A. Broggi. GOLD: a Parallel Real-Time Stereo Vision System for Generic Obstacle and Lane Detection. *IEEE Trans. on Image Processing*, 7(1):62–81, Jan. 1998.
2. M. Bertozzi, A. Broggi, and A. Fascioli. Stereo Inverse Perspective Mapping: Theory and Applications. *Image and Vision Computing Journal*, 1998(16):585–590, 1998.
3. A. Broggi, M. Bertozzi, A. Fascioli, and G. Conte. *Automatic Vehicle Guidance: the Experience of the ARGO Vehicle*. World Scientific, 1999.
4. Intel Corporation. *Intel Architecture MMX Technology Developers' Manual*. Intel Corporation, 1997. Available at <http://www.intel.com>.
5. Intel Corporation. *MMX Technology Programmers Reference Manual*. Intel Corporation, 1997. Available at <http://www.intel.com>.
6. H. A. Mallot, H. H. Bülthoff, J. J. Little, and S. Bohrer. Inverse perspective mapping simplifies optical flow computation and obstacle detection. *Biological Cybernetics*, 64:177–185, 1991.
7. L. Matthies. Stereo vision for planetary rovers: stochastic modeling to near real-time implementation. *Intl. Journal of Computer Vision*, 8:71–91, 1992.
8. M. Ohzora, T. Ozaki, S. Sasaki, M. Yoshida, and Y. Hiratsuka. Video-rate image processing system for an autonomous personal vehicle system. In *Procs. IAPR Workshop on Machine Vision and Applications*, pages 389–392, Tokyo, 1990.
9. S. Tommesani. The Pixel64 Module. Technical report, Dipartimento di Ingegneria dell'Informazione, Università di Parma, May 1998. Available at <http://www.ce.unipr.it/~tommesa>.