Using MMX Technology in Digital Image Processing
(Technical Report and Coding Examples)
TR-98-13

Vladimir Kravtchenko
Department of Computer Science
The University of British Columbia
201-2366 Main Mall, Vancouver
Canada, V6T 1Z4
vk@cs.ubc.ca

1. INTRODUCTION

1.1 Overview

MMX technology is designed to accelerate multimedia and communications applications. The technology includes new processor instructions and data types and exploits the parallelism in processing multiple data.

In this work we demonstrate how to use MMX technology in digital image processing applications and discuss important aspects of practical implementation with the GCC compiler. We also focus on the experimental results of common image processing operations and provide a comparative performance analysis.

This work is a result of collaboration between The University of British Columbia and International Submarine Engineering (ISE) Vancouver, Canada.

Note: In all programming examples we used the GNU GCC compiler (v2.7) and the following assembler convention: Source is on the Left, Destination is on the Right. All programs have been tested in Linux and UNIX (Solaris) operating systems. Details on how to include the assembly code into your GCC C code may be found in [13]. The information on the Intel processor instruction set may be found in [3].

1.2 History

Mar. 5, 1996 - Intel Corporation released details to the software community of its MMX technology. At the time it was the most significant enhancement to the Intel Architecture since the i386 processor. MMX has been the first enhancement to the x86 instruction set in more than ten years.

Shortly after, AMD cross-licensed MMX to incorporate it in its AMD-K6 processor and Cyrix licensed MMX for its 6x86MX processor. Both processors have been reported to support a standard set of 57 MMX instructions.

Jan. 8, 1997 - Intel introduced its first Pentium processor with MMX technology available at 166 and 200 MHz. Starting with Pentium II the MMX technology is integrated into every P6 family processor.

In early 1997 the leading compiler vendors, as Microsoft and Pwersoft/Watcom, delivered their first versions of compilers supporting MMX. Intel provided support for the MMX instruction set in its VTune performance tuning software.

Note: Intel Corporation claims MMX to be its brand name and not an acronym for Multi-Media-eXtension.

2. INTRODUCTION TO MMX TECHNOLOGY

MMX technology is designed to accelerate multimedia and communications applications. The technology includes new instructions and data types that allow applications to achieve a new level of performance. It exploits the parallelism inherent in many multimedia and communications algorithms, yet maintains full compatibility with existing operating systems and applications [1,5,6].

The highlights of the technology are:

- Single Instruction, Multiple Data (SIMD) technique
- 57 new instructions
- Eight 64-bit wide MMX technology registers
- Four new data types

Single Instruction, Multiple Data (SIMD) Technique

Today's multimedia and communication applications often use repetitive loops that, while occupying 10 percent or less of the overall application code, can account for up to 90 percent of the execution time. A process called Single Instruction Multiple Data (SIMD) enables one instruction to perform the same function on multiple pieces of data, similar to a drill sergeant telling an entire platoon, "About face," rather than addressing each soldier one at a time. SIMD allows the chip to reduce compute-intensive loops common with video, audio and graphics.

57 new instructions

MMX introduces 57 new instructions specifically designed to manipulate and process video, audio and graphical data more efficiently. These instructions are oriented to the highly parallel, repetitive sequences often found in multimedia operations.

Eight 64-bit wide MMX technology registers

MMX technology provides eight 64-bit general purpose registers. These registers are aliased on the floating point (FP) registers. This means that physically MMX registers and FP mantissas are the same but the content is interpreted in a different way depending on the MMX/FP mode. The MMX registers are accessed directly using the register names MM0 to MM7.

Four new data types

The principal data type of MMX technology is the packed fixed-point integer. The four new data types are: packed byte, packed word, packed double word and quad word all packed into one 64-bit quantity in quantities of 8, 4, 2 and 1 respectively.

An interactive on-line tutorial on MMX technology basics is available at [8].

3. MMX TECHNOLOGY FEATURES AND APPLICATION

3.1 New Features

MMX technology provides several new features to the existing processor architecture:

1. It supports a new arithmetic capability known as saturation arithmetic. Saturation provides a useful feature of avoiding wraparound artifacts.
2. The new 57 MMX instructions allow parallel processing of multiple data. These instructions may be grouped into 7 instruction categories: Arithmetic, Comparison, Conversion, Logical, Shift, Data Transfer and Empty MMX state. All MMX instructions start with a 0xFH byte and work on different data types.
3. The new 8 MMX registers are physically aliased with the FP registers. Switching to MMX mode is done automatically when MMX instruction is executed. Switching back to FP mode requires an execution of the EMMS instruction. Switching between the two modes is expensive and it is recommended not to intermix MMX and FP instructions in the same piece of code. Frequent transitions may result in significant performance degradation. It is recommended to stay in the MMX mode as long as possible during the application run. Before using the FP arithmetic the FP/MMX mode must be switched to FP, by executing EMMS (MMX mode exit instruction), to avoid FP errors. It should also be mentioned that MMX registers can not be used to address memory.
4. With regards to the operating systems, one should keep in mind that cooperative multi-tasking operating systems (e.g. MS Windows 3.1) do not save FP/MMX state when performing a context switch. Therefore application itself needs to save the relevant state before relinquishing direct or indirect control to the operating system. Preemptive multi-tasking systems (e.g. MS Windows 95) are responsible for saving and restoring the FP/MMX state when performing a context switch. Therefore, in the later case the application does not have to save or store the FP/MMX state.

In [1] the reader may find a detailed description of MMX features, data types and all MMX instructions.

To detect if MMX technology is present in the hardware one may use a CPUID instruction. [9] provides details on the information returned by the CPUID. We provide a small program in C and Assembler [15] that detects the presence of MMX in computer.

3.2 Writing MMX Code

The best way to design an efficient MMX code is to write blocks in assembly language that later can be included into your C code. When writing the code using MMX instructions one should follow general guidelines considering the processor architecture, pipelining and dynamic execution features. In [2,4] the reader may find important information on the following aspects:

- The rules of paring the instructions in U and V processor pipelines
- Optimal memory and register access sequencing
- Memory caching and read/write operations
- Branch prediction and dynamic execution
- Alignment of data and stack

The last item is very important for achieving high performance and we will focus on it separately.
When data access to a cacheable address misses the data cache, the entire line is brought into the cache from external memory. This is called a line fill. On Pentium and dynamic execution (P6-family) processors data arrives in a burst composed of four 8-byte sections to match the cache line size of 32 bytes. On the P6 when write occurs and the write misses the cache, the entire 32-byte cache line is fetched. On the Pentium processor, when the same write miss occurs, the write is simply sent out to memory. Therefore, by aligning data on a 32-byte boundary in memory, one may take advantage of matching the cache line size and

avoiding multiple memory-cache transfers. The delay for a cache miss on the Pentium with MMX is 8 internal clock cycles and on P6 the minimum delay is 10 internal clock cycles.

In order to align data on a 32-byte boundary, padding of data may be required. Many compilers allow the user to specify the alignment. However, one may use the following C code to guarantee the alignment:

Example 1:

```
unsigned char *pad, *data;                                    // --- padding the data ---

pad  = (unsigned char*) malloc(SIZE+31);                      // the pad pointer
data = (unsigned char*) ((((unsigned int) pad) +31) & (~31)); // the 32-byte aligned pointer

... your program ...

free(pad);
```

As a matter of convention, compilers allocate anything that is not static on the stack and it may be convenient to make use of 64-bit data quantities that are stored on the stack. Another reason for aligning the stack may be the following, if inside the MMX routine an argument passed to routine is accessed several times it is better be on an aligned boundary to avoid cache misses, if occur. The following code in the program prologue and epilogue will make sure the stack before calling the MMX routine is aligned:

Example 2:

```
  asm volatile
      (                                  // --- stack alignment ---

        "mov       %%esp, %%ebx \n\t"    // load ESP into EBX
        "sub           $4, %%ebx \n\t"   // reserve space on stack for the old value of ESP
        "and         $-32, %%ebx \n\t"   // align EBX along a 32 byte boundary
        "mov     %%esp, (%%ebx) \n\t"    // save old value of ESP in stack, behind the boundary
        "mov       %%ebx, %%esp \n\t"    // align ESP along a 32 byte boundary
        ::
      );

... your program with calls to MMX routines ...

  asm volatile
      (                                   // --- restoring old stack ---

        "mov     (%%esp), %%ebx \n\t"     // load the old value of ESP
        "mov       %%ebx, %%esp \n\t"     // restore the old value of ESP
        ::
      );
```

3.3 Measuring MMX Execution Times

An interesting problem is how to measure the execution time of the MMX routine. This could be done using the information from the RDTSC (read-time stamp counter) which contains the cycle counter. If we obtain the value of the counter at the start and finish of the routine and given a clock frequency of the processor we can calculate an approximate time it takes the routine to execute. The reader may find a detailed description of the RDTSC counter in [10] and a small program in C that performs the measurement in [15].

The measured timing is approximate and depends on many factors as OS overheads, number of processes running, cash situation if MMX code contains memory read/write operations, etc. It is recommended to close all unnecessary applications before measuring the time. Various conditions, as cache warming - prior reading and/or writing from/to the same memory blocks, a particular write strategy implemented in the processor and L2 cache, most significantly affect the performance. The basic information on L1 and L2 caches may be found in [12] and description of the write allocation effects in [2]. The runs in real conditions may be 2-3 times slower then in synthetic ones (e.g. with junk, non-initialized Source/Destinationt images, untouched cache). See the Notes section in the Digital Image Processing Test Results below.

The user must be careful considering the timing results associated with the marketing information. The results may look superior if application is run on a high-end machine with plenty L2 cache, fast RAM and with synthetic data. To make any judgements, see first how it runs on your machine, with your data and after you personally compiled the code. The last is very important. 'Add two images with saturation' example contains a very simple procedure that can give one a clue on the overall MMX software performance.

When application requires a memory transfer of big chunks of memory the MMX registers may be utilized. Although, it works a little slower compared to using 32-bit double words, in many cases timing is very satisfactory and such transfer may be even preferable. Various memory transfer techniques are well described in [11].

3.4 Hints for developing MMX code

Some hints on packing/unpacking techniques, absolute differences, clipping and generating constants in MMX registers is given in [2]. Below we provide additional hints that will help you to design even more efficient code.

a. To design the efficiently running loops the following guidelines are recommended, see Example 3:

- implement the loop counter in one of the general purpose registers, say ECX and decrement it to zero
- calculate the SOURCE and DESTINATION addresses in the general purpose registers, EAX and EBX for the source and EDI for the destination in our example
- load 8 bytes at a time directly into MMX registers or read 8 bytes from memory if direct loading may be avoided
- align the loop entry at a 16 byte boundary to use the advantage of branch prediction
- unroll double loops into a single loop. This may enlarge your code but will speed-up the execution

Example 3 (add two images with saturation):

```
int ImgAdd(unsigned char *Src1, unsigned char *Src2, unsigned char *Dest, int length)
{
  if (length < 8) return 0;              // image size must be at least 8 bytes

  asm volatile
      (
        "mov          %2, %%eax \n\t"    // load Src1 address into EAX
        "mov          %1, %%ebx \n\t"    // load Src2 address into EBX
        "mov          %0, %%edi \n\t"    // load Dest address into EDI

        "mov          %3, %%ecx \n\t"    // load loop counter (SIZE) into ECX
        "shr          $3, %%ecx \n\t"    // counter/8 (MMX loads 8 bytes at a time)

        ".align 16              \n\t"    // 16 byte alignment of the loop entry
        ".L1010:                \n\t"

        "movq    (%%eax), %%mm1 \n\t"    // load 8 bytes from Src1 into MM1
        "paddusb (%%ebx), %%mm1 \n\t"    // MM1=Src1+Src2 (add 8 bytes with saturation)
        "movq    %%mm1, (%%edi) \n\t"    // store the result in Dest

        "add          $8, %%eax \n\t"    // increase Src1, Src2 and Dest
        "add          $8, %%ebx \n\t"    // register pointers by 8
        "add          $8, %%edi \n\t"

        "dec              %%ecx \n\t"    // decrease the value of the loop counter
        "jnz             .L1010 \n\t"    // check loop termination, proceed if necessary

        "emms                   \n\t"    // exit MMX state
        : "=m"  (Dest)                    // %0
        :  "m"  (Src2),                   // %1
           "m"  (Src1),                   // %2
           "m"  (length)                  // %3
      );

  return 1;
}
```

b. Use PXOR to zero the MMX register and PCMPEQB to generate all ones in the MMX register, e.g.

```
       "pxor      %%mm0, %%mm0 \n\t"    // zero mm0 register
       "pcmpeqb   %%mm1, %%mm1 \n\t"    // generate all 1's in mm1
```

c. To perform a byte shift use a word shift and then apply a mask, avoid the pack/unpack operations, e.g.

```
unsigned char Mask[8] = { 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F };
```

load Mask into MM0

```
"psrlw        $1, %%mm1 \n\t"    // shift 4 WORDS of mm1 1 bit to the right
"pand      %%mm0, %%mm1 \n\t"    // apply Mask to 8 BYTES of mm1
```

Note: "pand" didn't work in the version of the GNU compiler we used so, the instruction was replaced with its byte representation ".byte   0x0f, 0xdb, 0xc8 \n\t". The description of byte codes may be found in Appendix B of [1].

d. Implement division/multiplication by the value, which is a power of 2, as left/right shifts.

e. There's no division instruction in MMX and if division by an arbitrary integer is necessary there's nothing to do but forget about MMX and use ordinary div, idivw and similar instructions.

f. When one wants to apply (add, subtract etc.) a byte constant to all elements of the data set multiply the constant in the MMX register and then operate with the resulting register, e.g

```
        "mov           %2, %%al \n\t"    // load a byte Constant into AL
        "xor         %%ah, %%ah \n\t"    // zero AH
        "mov         %%ax, %%bx \n\t"    // copy AX into BX
        "shl         $16, %%eax \n\t"    // shift 2 bytes of EAX left
        "mov         %%bx, %%ax \n\t"    // copy BX into AX
        "movd      %%eax, %%mm1 \n\t"    // fill the lower half of MM1 with the Constant
        "movd      %%eax, %%mm2 \n\t"    // copy EAX into MM2
        "punpckldq %%mm2, %%mm1 \n\t"    // fill the higher half of MM1 with the Constant

        "paddusb   %%mm1, %%mm2 \n\t"    // MM2=MM1+MM2
```

g. Be careful when you multiply positive bytes using the unpack into words and multiply words operations. If numbers are big you can get word results that may be interpreted as negative when packed back into bytes and saturation to a wrong value may occur. Take the abs value of words before packing them back into bytes using, for example, the following technique:

```
                                          // ** Take abs value (signed words) **

        "movq      %%mm3, %%mm5 \n\t"    // copy mm3 into mm5
        "psraw       $15, %%mm5 \n\t"    // fill mm5 words with word sign bit
        "pxor      %%mm5, %%mm3 \n\t"    // take 1's compliment of only neg. words
        "psubsw    %%mm5, %%mm3 \n\t"    // add 1 to only neg. words, W-(-1) or W-0
```

h. Use all general purpose registers: EAX, EBX, ECX, EDX, ESI, EDI and their integral parts and all MMX registers MM0 to MM7 to the full capacity to reduce the number of memory access operations.

i. <u>Important !</u> If you experience problems running MMX subroutines, e.g. computer hangs up, or some variables unexpectedly change their values after the call to the subroutine, be sure the compiler pushes the general purpose registers, used in the subroutine, on stack and pops them out afterwards. We experienced the problem when called the MMX subroutine from inside the for-loop. To insure the registers are saved and restored properly you may use the following code in the prologue and epilogue of your MMX code:

```
“push %%eax \n\t”  // or whatever registered are used in your MMX subroutine
“push %%ebx \n\t”

… the mmx code …

“pop %%ebx \n\t”
“pop %%eax \n\t”
```

A “pushad” instruction may be used to push all EAX, EBX, ECX, EDX, ESP, EBP, ESI and EDI registers on to stack and “popad” to pop them out.

4. DIGITAL IMAGE PROCESSING TEST RESULTS

Below you may find the test results of the MMX routines implementing common image processing operations. The routines themselves, the main program and the test images may be found at [15].

<u>Test conditions:</u>

Hardware:	Pentium II, 233 Mhz, 512 Kb L2 cache
Hardware:	dual Pentium II, 400 Mhz, 1024 Kb L2 cache
Software:	Linux OS
Images:	gray scale 256x256x8 bit and 512x512x8 bit
Results:	in milliseconds, 1 ms = 1/1000 sec
Measurements:	CPU cycles divided by CPU speed, first five run times are averaged
Conditions:	realistic, including the OS overheads and memory transfer, near to worst case (the source image memory blocks are filled with images from disk, the destination image is not in the write buffer/cache)
Comparison:	The results for the same or similar operations are given under each particular table and are taken from the commercial products information sources [7, 14].

Point-to-Point operations:

Processor		PII	PII	PII	PII
		233 Mhz	233 Mhz	400 Mhz	400 Mhz
Image		256x256	512x512	256x256	512x512
ImgAdd	D = saturation255(S1 + S2)	0.8	5.2	1.0	3.6
ImgMean	D = S1/2 + S2/2	0.9	5.7	1.1	3.7
ImgSub	D = saturation0(S1 - S2)	0.8	5.2	1.0	3.6
ImgAbsDiff	D = I S1 - S2 I	0.9	5.7	1.0	3.7
ImgMult	D = saturation255(S1 * S2)	1.1	6.8	1.2	4.3
ImgMultNor	D = saturation255(S1 * S2) non MMX	2.0	10.3	1.7	6.5
ImgMultDivby2	D = saturation255(S1/2 * S2)	0.9	6.0	1.1	4.0
ImgMultDivby4	D = saturation255(S1/2 * S2/2)	0.9	6.2	1.1	4.0
ImgBitAnd	D = S1 & S2	0.8	5.2	1.0	3.6
ImgDiv	D = S1 / S2 non MMX	4.1	18.4	2.9	11.3
Average	Excluding non-MMX	0.9	5.8	1.1	3.8

Sharp GPB-II board: average time excluding memory transfer:

256x256x8bit	2.62 ms
512x512x8bit	10.48 ms

Matrox MIL, PII 400 Mhz:

Add two 512x512x8 bit images with saturation	2.4 ms

Point unary operations:

Processor		PII 233 Mhz	PII 233 Mhz	PII 400 Mhz	PII 400 Mhz
Image		256x256	512x512	256x256	512x512
BitNegation	S = !S	0.3	2.0	0.3	1.2
AddByte	S = saturation255(S + C)	0.3	2.0	0.3	1.2
AddByteToHalf	S = saturation255(S/2 + C)	0.3	2.1	0.3	1.2
SubByte	S = saturation0(S - C)	0.3	2.0	0.3	1.2
ShiftRight	S = saturation0(S >> N)	0.3	2.1	0.3	1.2
MultByByte	S = saturation255(S * C)	0.6	3.7	0.7	2.4
ShiftRightAndMultByByte	S = saturation255((S >> N) * C)	0.5	3.6	0.6	2.2
ShiftLeftByte	S = (S << N) (more efficient !)	0.3	2.0	0.4	1.2
ShiftLeft	S = saturation255(S << N)	0.6	3.6	0.5	2.1
BinarizeUsingThreshold	S = S >= T? 255:0	0.3	2.1	0.3	1.2
ClipToRange	S = (S >= Tmin) & (S <= Tmax) 255:0	0.3	2.1	0.4	1.3
NormalizeLinear	S = saturation255((Nmax – Nmin)/ (Cmax - Cmin)*(S - Cmin) + Nmin)	0.7	4.1	1.0	2.8
Average		0.4	2.6	0.5	1.6

Sharp GPB-II board: average time excluding memory transfer:

256x256x8bit	2.62 ms
512x512x8bit	10.48 ms

Small Convolution Kernel Operations (Division takes approx. 50 % of the execution time)

Processor		PII 233 Mhz	PII 233 Mhz	PII 400 Mhz	PII 400 Mhz
Image		256x256	512x512	256x256	512x512
ConvolveKernel3x3Divide	Dij = saturation0and255(...)	17.4	72.0	10.5	42.0
ConvolveKernel5x5Divide	Dij = saturation0and255(...)	25.7	106.5	15.2	62.0
ConvolveKernel7x7Divide	Dij = saturation0and255(...)	30.0	124.1	17.7	72.3
ConvolveKernel9x9Divide	Dij = saturation0and255(...)	47.9	201.8	28.0	117.5
ConvolveKernel3x3ShiftRight	Dij = saturation0and255(...)	7.6	39.4	4.8	22.9
ConvolveKernel5x5ShiftRight	Dij = saturation0and255(...)	17.7	74.3	10.7	43.2
ConvolveKernel7x7ShiftRight	Dij = saturation0and255(...)	22.8	96.3	13.7	56.0
ConvolveKernel9x9ShiftRight	Dij = saturation0and255(...)	43.3	182.3	25.4	106.2

Matrox MIL, PII 400 Mhz: 512x512x8 bit image convolutions without division

ConvolveKernel3x3	8.0 ms
ConvolveKernel5x5	32.4 ms

Edge operators:

Processor		PII 233 Mhz	PII 233 Mhz	PII 400 Mhz	PII 400 Mhz
Image		256x256	512x512	256x256	512x512
SobelX	Dij = saturation255(...)	2.4	11.2	1.7	6.8
SobelXShiftRight	Dij = saturation255(...)	2.6	12.2	1.9	7.3

Matrox MIL, PII 400 Mhz: 512x512x8 bit image

Edge detection (sobel)	9.6 ms

Tested in Unix (SUN Solaris) environment, the MMX routines did not perform as well as in Linux on exactly the same hardware. The running time slightly increased (1.2-1.5 times) and results were not as repetitive.

Our basic image processing routines proved to be comparable in performance to the commercial MMX library [14] and better than the specialized board [7] performance. With regards to the GPB-II board, it should be noted that our results include memory transfer, and even with such inclusion the execution time for the basic operations is at least three times less. Considering the fact that memory transfer is a very expensive operation a much greater factor can be expected.

Notes:

To demonstrate the effect of cache on the overall performance the following synthetic conditions were introduced to test the ‘add two images with saturation’ procedure - ImgAdd on PII 400 MHz and 512x512x8 bit gray scale image:

Conditions:	Performance (in ms):
Real	3.6
Synthetic	
- With memory read/write operations (movq) removed	0.3
- Reads from cache pre-read memory blocks (junk), writes to cache pre-write memory blocks (junk)	1.0
- Reads and writes from the uncached memory (junk)	2.0
- Reads from cache pre-write memory blocks, writes to cache pre-read memory blocks (worst case)	3.7

We can see that memory transfer takes approximately 70 % of the execution time, and depending on the cache situation, exactly the same code processing a 512x512 image, may run from 1.0 to 3.7 ms.

An interesting aspect, tested on a slower machine with a Pentium II 233 MHz, the performance of some compute non-intensive routines turned out to be slightly better for smaller images, e.g. ImgAdd showed 0.8 ms versus 1.0 ms on Pentium II 400 MHz for 256 x 256 images. This could be explained with the difference in the chipset, cache implementation and RAM access speed. Where level of computation is low, images are small and mostly memory transfer capabilities compete, the computer with faster memory but slower processor may show very good performance.

Another interesting aspect is that execution time is not always linearly proportional to the size of the image. This may be explained considering the cache size. With 512 Kb L2 cache, all 256x256 Src/Dest images fit into cache, there's even some space left. With 512x512 images only 2/3 of each image fit into cache and reads/writes of the rest parts turn out to be very expensive because they involve cache/memory swaps. The expectation is that on large images computer with larger L2 cache will outperform the computer with a smaller cache.

To conclude the above speculations, if the user works with large images the size of the L2 (and L1) cache and its implementation becomes critical. If MMX routines are compute intensive the clock speed of the CPU will play its significant role. If MMX routines have a big amount of memory transfer the RAM speed and the bus characteristics should be adequate to the expected level of performance.

5. CONCLUDING REMARKS

Using MMX technology proved to improve the performance of common digital image processing operations. If parallel processing of multiple data is applicable, on the average, the MMX application exhibits two to four times better performance than its ordinary counterpart.

Experiments with different hardware and operating systems showed that execution time is dependent on many factors, of which memory transfer characteristics being most important, and the exact time can not be provided until the user performs his own test under his own conditions.

MMX technology is best suited to work with signed words rather than any other data type. For this reason, coding in some cases is not straightforward and various tricks have to be utilized to avoid awkward packing/unpacking operations. It would be desirable to have an additional set of byte-oriented instructions as well as some solution for the division operator.

To boost the performance of MMX routines various optimization techniques have to be utilized.

By large, considering various factors, the developer will benefit from using MMX technology in speeding up compute intensive loops and exploiting the parallelism in processing multiple data. The user will benefit from avoiding additional expenses for the specialized image processing hardware, because MMX software can provide comparable or even better performance.

6. REFERENCES

[1] Intel Architecture MMX Technology
Programmer's Reference Manual,
Intel, Order No. 243007-002, March 1996

[2] Intel Architecture MMX Technology
Developer's Manual,
Intel, Order No. 243006-001, March 1996

[3] Intel Architecture Software Developer's Manual,
Volume 2: Instruction Set Reference,
Intel, Order No. 243191, May 1997

[4] Intel Architecture Optimization Manual
Intel, Order No. 242816-003, 1997

[5] The Complete Guide to MMX Technology,
David Bistry, McGraw Hill, July 1997

[6] Direct, Rdx, Rsx, and Mmx Technology :
A Jumpstart Guide to High Performance Apis,
Rohan Coelho, Maher Hawash,
Addison-Wesley, December 1997

[7] Sharp GPB-2 Image Processing Board User's Manual,
Sharp Digital Information Products, January 1995

On-line resources:

[8] Introduction to MMX Technology, on-line tutorial, Intel
http://www.intel.com/design/perftool/cbts/mmxintro/

[9] A Quick and Easy Way to Use CPUID at Run-time, Intel
http://developer.intel.com/drg/pentiumII/appnotes/877.htm

[10] Using the RDTSC Instruction for Performance Monitoring, Intel
http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM

[11] Memory Transfer Timing Utility, Intel
http://developer.intel.com/drg/pentiumII/appnotes/813/813.htm

[12] CACHECHK, Ray Van Tassle
http://www.simtel.net/pub/simtelnet/msdos/sysinfo/memspd1.zip

[13] GCC info Index, SUN Microsystems
http://src.doc.ic.ac.uk/info/gcc,Index

[14] MIL Benchmarks, Matrox Electronic Systems
http://www.matrox.com/imgweb/products/mil/benchmarks.htm

[15] Using MMX Technology in Digital Image Processing,
Technical Report and Coding Examples, Vladimir Kravtchenko
http://www.cs.ubc.ca/spider/vk/mmx.html