High-speed image processing algorithms using MMX hardware

J. W. V. Miller and J. Wood
The University of Michigan-Dearborn

ABSTRACT

Low-cost PC-based machine vision systems have become more common due to faster processing capabilities and the availability of compatible high-speed image acquisition and processing hardware. One development, which is likely to have a very favorable impact on this trend, is enhanced multimedia capabilities present in new processor chips such as Intel MMX and Cyrix M2 processors. Special instructions are provided with this type of hardware which, combined with a SIMD parallel processing architecture, provides a substantial speed improvement over more traditional processors. Eight simultaneous byte or four double-byte operations are possible. The new instructions are similar to those provided by DSP chips such as multiply and accumulate and are quite useful for linear processing operations like convolution. However, only four pixels may be processed simultaneously because of the limited dynamic range of byte data. Given the inherent limitations with respect to looping in SIMD hardware, nonlinear operations such as erosion and dilation would seem to be difficult to implement. However, special instructions are available for required operations. Benchmarks for a number of image-processing operations are provided in the paper to illustrate the advantages of the new multimedia extensions for vision applications.

Keywords: machine vision, image processing algorithms, multimedia extensions, single instruction multiple data, performance gains

1. INTRODUCTION

Along with other recent advances in microprocessor developments for personal computers, the advent of a special instruction set for multimedia applications significantly decreases processing time for many machine vision algorithms. Originally specified and developed by Intel which they designated as MMX technology.1 Such processors are fully compatible with all earlier X86 software and are only modestly more expensive than processors without the new capabilities. At least two other companies, AMD and Cyrix, have developed compatible hardware with similar performance. Applications targeted for the new hardware are motion video, combined graphics and video, image processing, audio synthesis, speech synthesis and compression, among others. These applications are ideal for MMX because they all share the same fundamental characteristics, namely:

- Small integer data types (for example: 8-bit graphics pixels)
- Small, highly recurring loops
- Compute-intensive algorithms
- Highly paralleled operations
- Algorithms are compatible with Single Instruction Multiple Data (SIMD) architecture

2. OVERVIEW OF THE NEW MULTIMEDIA ARCHITECTURE

An SIMD architecture is used to improve performance by processing multiple data elements in parallel. This feature along with packed data types and special instructions provides significantly faster processing.

2.1 New data types and registers

The principal data type of the MMX instruction set is the packed fixed-point integer, where multiple integer quantities are grouped into a single 64-bit entity. The MMX data types are:

- Packed byte: Eight bytes packed into a single 64-bit quantity
- Packed word: Four 16-bit words packed into a single 64-bit quantity
• Packed doubleword Two 32-bit double words packed into a 64-bit quantity
• Quadword One 64-bit quantity

The new architecture provides eight 64-bit general-purpose registers. These registers are aliases for the floating-point registers and are directly accessed using the register names mm0 to mm7.

2.2 New Instructions

The MMX instruction set operates on all data types packed into a single 64-bit quantity to complete the following functions in parallel:

- Data transfer (move instruction) for MMX register-to-register transfers, etc.
- Basic arithmetic operations (add, subtract, multiply, divide)
- Conversions between the data types – pack data together, and unpack from small to large data types.
- Logical operations (and, or, and not, or, xor)
- Shift operations
- Comparison operations

The combinations of different packed integer data types with each of the functions above create the 57 new instruction op codes.

3. TRADITIONAL CODING FOR MACHINE VISION APPLICATIONS

Typical machine vision algorithms provide different types of image processing such as median filtering, morphology, and edge detection. Filtering applications process one or more input images and then produce one output image accordingly. Generally, the complete input image is stored in image buffers as either one byte per pixel (values 0-255) or two bytes per pixel (values 0-65535) depending upon the dynamic range required. Because every pixel (except in some cases border pixels) must be processed in sequence, image processing operations often require substantial amounts of time for completion. This often imposes substantial limitations on the amount of processing possible for real-time applications. If a given application requires additional processing, the classical approach would utilize expensive special-purpose hardware.

An implementation for a simple one-dimensional algorithm for edge detection using a digital derivative can be coded in C as follows:

```c
for (i = 0; i < nRow; i++)
    for (j = 0; j < nPixel; j++)
        outbuf[i][j] = inbuf[i][j] - inbuf[i][j+1];
```

The outer loop steps through the image buffer row by row, and the innermost loop steps through the image buffer pixel by pixel. This filter simply subtracts an adjacent pixel from the current pixel and stores it in the output image buffer. Note that the result may be less than 0 or greater than 255. When this happens the value may be clipped to either 0 if the result is less than 0, or 255 if the result is greater than 255 assuming that the image buffers are of type unsigned char. The additional coding required to accomplish this is not shown but will increase processing time.

Morphological filters can be implemented using a similar approach but in place of the subtraction operation, maximum and minimum operations are used instead. These filters perform the standard dilate, erode, open and close operations for a variety of structuring elements. Other operations include image sums and differences, logical operators, general convolution, and median filtering. All of the processing techniques are candidates for the new multimedia architecture.

4. SPEED IMPROVEMENTS WITH THE NEW ARCHITECTURE

Machine Vision applications possess all the characteristics required to exploit the advantages of the new multimedia architecture. Substantial performance gains, if realized, will greatly increase the amount of processing that can be accomplished for real-time applications without the need for expensive special-purpose hardware.
4.1 Potential performance gains

From a casual review of the architecture and instruction set, dramatic performance gains would be expected. However, other factors besides the raw processing power affect the actual improvements that are achieved as will be discussed later. As a starting point, however, the following are not unreasonable expectations:

- 8-bits/pixel images can be processed up to eight times faster than the original code by processing eight pixels in parallel.
- 16-bits/pixel images can see performance gains as much as four times the original code by processing four pixels in parallel.

4.2 Image processing with MMX hardware

As previously noted with traditional coding, one pixel is processed at a time. For example, if a gray-scale image has a resolution of 640 × 480 pixels, then 307,200 iterations will need to take place to process the entire image. With MMX hardware, however, only 38,400 iterations will be needed since 8 pixels can be processed at a time, which should be considerably faster. In order to achieve this, however, the original filter code must be revised to use the MMX instruction set. Performance gains for specific filters such as morphology, noise reduction and edge detection filters implementing MMX technology are shown near the end of this research paper.

Only hardware and software that supports MMX technology could be used to develop and test the new code. The hardware used for evaluation was a computer with 16 Mbytes of main memory, a 512K pipelined secondary cache and an AMD K6 166 MHz processor. The original routines (no MMX instructions for comparison purposes) were written in C code and compiled using the DOS port of GCC (DJGPP). Since MMX hardware is new, most high-level language compilers do not support it. This problem was overcome by writing the majority of the filter code in C and writing the portions that benefit from MMX (the innermost loop) in assembly code. The software currently being used is Microsoft Visual C/C++ V5.0 for compiling the C code and NASM V0.94 for compiling the MMX assembly code. Intel created a C/C++ Plug-In for use with Microsoft Visual C/C++ that includes special MMX “C type” syntax; however, this software was not used because it did not add much of a benefit over standard assembly code.

The following approach was used to implement various functions:

- Code the MMX assembly routines as external functions in C.
- Remove the innermost pixel-by-pixel iteration loop and replace it with the equivalent MMX code.
- Optimize the MMX code to optimize processing speed.

4.3 Accessing assembly code functions

Only segments of the filters that benefit from MMX will use assembly code, the rest will remain in C. The best way to combine C with assembly code is to call the MMX assembly code from an external function.2 External assembly code functions act exactly the same as standard ANSI C functions except the “internal” code is in assembly instead of C. The assembly code is compiled with NASM V0.94 and then linked with the C code to form the final executable.

4.4 Checking for the presence of MMX hardware

A check to verify the presence of MMX hardware was implemented to prevent aberrant operation if the hardware was not present. To prevent this, a special function was written to detect MMX hardware and run either straight C code or code that will use the MMX hardware.3

4.5 Using MMX instructions in the innermost loop

The segments that benefit the most from MMX are the innermost filter loops. The original C algorithms process each pixel one at a time; however, MMX can process up to eight pixels at a time by eliminating the innermost loop and replacing it with MMX code. Note that the outer loop could also have implemented MMX. However, this approach was not taken for two reasons: first, it would not provide a significant performance increase, and second, the MMX functions become much more versatile since row pointers can be passed from C instead of setting up specific row pointers inside the assembly code. Here is a code fragment for a morphology filter that finds the maximum of three pixel values and places the result in the output buffer. Both the original C code and the equivalent MMX code are show here:
Original C code algorithm:

```c
#define max(a,b)     (((a) > (b)) ? (a) : (b))

for (i = 1; i < nl[infile] - 1; i++)         /* imbuf[1] is the output image buffer and imbuf[0] is the input image buffer */
    for (j = 1; j < nb[infile] - 1; j++)
        imbuf[1][i][j] = max(imbuf[0][i-1][j-1], max(imbuf[0][i][j], imbuf[0][i+1][j+1]));
```

Equivalent C code implementing MMX technology:

```c
int nColSets = nb[infile] / 8;

for (i = 1; i < nl[infile] - 1; i++)
{
    unsigned char *InPtr0 = imbuf[0][i-1], *InPtr1 = imbuf[0][i] + 1;
    unsigned char *InPtr2 = imbuf[0][i+1] + 2, *OutPtr = imbuf[1][i] + 1;
    dilate_mmx(InPtr0, InPtr1, InPtr2, OutPtr, nColSets);
}
```

Notice that only the outer “for” loop is required for the MMX implementation. This loop accesses each line like the original code; however, its purpose is to set up line pointers for the MMX assembly code function (here called dilate_mmx). Since the input image buffer is of data type unsigned char the MMX code can process eight pixels at a time. Next the line pointers are setup to point at the line and the first pixel to be processed in the MMX assembly code. What the MMX assembly code does is load eight pixels starting from this first “pointer pixel.” Each of the eight pixels is processed independently of the other pixels. This assembly code function takes these three sets of eight pixel values and calculates the maximum value of the three sets. It then stores the maximum values at the output buffer pointer (OutPtr). The variable nb[infile] represents the amount of pixels/line in the input image. Since eight pixels are processed at a time with MMX technology, the number of bytes to process per loop iteration is reduced by eight. This is accomplished by dividing the nb[infile] variable by eight which determines the total number of pixel groups that the assembly code should process.

Here is the actual code for the function dilate_mmx:

```assembly
_dilate_mmx:
    push ebp
    mov ebp, esp

    mov esi, [ebp + 24] ; esi contains the value of nColSet
    mov ebx, [ebp + 8]  ; ebx contains InPtr0 pointer
    mov ecx, [ebp + 12] ; ecx contains InPtr1 pointer
    mov edx, [ebp + 16] ; edx contains InPtr2 pointer
    mov edi, [ebp + 20] ; edi contains OutPtr pointer
    movq mm6, [signOffset] ; mm6 contains the packed value 128 for converting from unsigned to “signed” values

    dilate_loop: ; the innermost loop begins here to process all interior pixels
        movq mm0, [ebx] ; move current eight pixels from pointer InPtr0 into mm0
        add ebx, 8 ; advance pointer by 8—to point at next eight pixels
        ; additional code...
    }
```
movq mm1, [ecx] ; move InPtr1 eight pixels into mm1
add ecx, 8 ; advance pointer to point at next eight pixels

psubb mm0, mm6 ; subtract 128 from each pixel to convert to “signed” values for PCMPGTB op-code
psubb mm1, mm6

movq mm4, mm0 ; copy mm0 to mm4

pcmpgtb mm4, mm1 ; store Greater Than Mask of InPtr0 and InPtr1 in mm4
pand mm0, mm4 ; keep only the greater values in mm0

pandn mm4, mm1 ; keep only the greater values in mm4
por mm0, mm4 ; logic OR mm0 with mm4 to store the great values of both mm0 and mm4

movq mm2, [edx] ; move InPtr2 eight pixels into mm2
add edx, 8

movq mm4, mm0
psubb mm2, mm6

pcmpgtb mm4, mm2

pand mm0, mm4 ; keep only the greater values in mm0
pandn mm4, mm2 ; keep only the greater values in mm4

por mm0, mm4 ; logic OR mm0 with mm4 to store the great values of both mm0 and mm4
paddb mm0, mm6 ; add 128 to restore original pixel size

mov [edi], mm0 ; move the greater values into the output buffer pointer OutPtr
add edi, 8 ; advance output pointer by eight to point at the next eight pixels

dec esi ; decrement the nColSet variable. When nColSet reaches zero—the loop ends.
jg dilate_loop

mov esp, ebp
pop ebp
ret

... signOffset dd 80808080h
dd 80808080h

The variable signOffset is required since the MMX instruction code PCMPGTB (Packed Compare Greater Than- Byte) only works with signed values, and, since the pixel values are unsigned, they must be temporarily converted to signed values. This is done by subtracting 80h (128 in decimal) from each packed value, thus creating a “signed” value, and then once all the calculations are complete, 80h is added to restore the original values without the loss of any data.

4.6 Optimizing MMX assembly code

High-level languages such as C automatically optimize their code to obtain maximum performance; however, assembly needs to be optimized by the programmer to achieve the same results. Pentium-type processors allow two instructions to be paired for each clock cycle. This means for each clock cycle, two instructions can be executed simultaneously. The two execution pipelines in the processor are the U and V pipelines. The U pipeline can execute any instruction, while only basic instructions may be executed in the V pipeline. To achieve the desired performance gains, much consideration should be
taken to pair up instructions in the U and V pipelines. Here are the general guidelines for pairing MMX instructions in the U and V pipelines:

- MMX instructions which access either memory or the integer register file can be issued in the U-pipe only.
- The MMX destination register of the U-pipe instruction should not match the source or destination register of the V-pipe instruction (dependency check).
- 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.
- Two MMX instructions in which both use the MMX shifter unit (pack, unpack, and shift instructions) cannot pair since there is only one MMX shifter unit. Shift operations may be issued in either the U-pipe or the V-pipe but not in both in the same clock cycle.

4.7 Memory architecture considerations

It would appear that MMX hardware imposes special demands on computer memory. With classical implementations of image processing algorithms, one pixel at a time is processed compared to eight with the new hardware. This means that much more data must be read out of memory in approximately the same time interval, and if the bandwidth is not available, stalls will occur which lessen the effectiveness of the MMX hardware.

The amount of cache memory available has a very significant effect on memory bandwidth. The main memory, which uses DRAM, operates at a much slower rate than the CPU. Without cache memory, performance would be greatly degraded for even processing a single pixel at a time. Two caches are provided to improve performance. A large secondary cache is used to buffer the main memory. However, this cache only operates at around 60 MHz and is 2-3 times slower than the processor clock. With no additional hardware, this would almost completely negate the advantage of clock-multiplied CPU chips. By providing a small but very fast primary cache, good performance would be expected even with demands from MMX hardware as long as there are not too many primary cache misses. However, when processing large images that are too big for the primary cache, image data will have to be obtained from the secondary cache with the potential for a significant performance loss.

5. PERFORMANCE RESULTS TO DATE

The available test results provide a number of surprises and inconsistencies. A variety of routines were evaluated using a 640 X 480 test image. A small 160 X 10 window of this image was also used to evaluate performance so that only the primary cache would be used to evaluate the significance of primary cache misses with large images. The results were obtained under the following conditions:

1. Performance results were obtained on an AMD K6 166MHz microprocessor computer with 16 Mbytes of memory and a 512K pipelined secondary cache. The primary cache instruction and data caches were both 32Kbytes.
2. The operating system was Windows 95.
3. The computer was lightly loaded (no other significant tasks were running concurrently) during testing.
4. C routines were compiled using Microsoft Visual C/C++ V5.0 with compiler options set to produce Pentium code and optimization set to maximum speed.
5. MMX technology routines were assembled using DJGPP with NASM V0.94.

5.1 Performance results with large images

Six different image processing routines were evaluated using both standard and MMX coding, digital derivative, two different dilation functions, logical operations between two images, sums of 8 and 16 bit/pixel images and difference of two 8 bit/pixel images. Processing rates are impressive, even without using the MMX instructions. However, significant gains are apparent when the multimedia hardware is used as given in Table 1. Improvement factors ranged from 1.55 to 4.83 and were generally best when using 8 bit/pixel data as expected. The most significant improvements can be noted in routines that manipulate data in more complex fashions.

<table>
<thead>
<tr>
<th>Filter description</th>
<th>Conditions</th>
<th>Conventional</th>
<th>MMX</th>
<th>Performance</th>
</tr>
</thead>
</table>


5.2 Performance results with small images

By reducing image size sufficiently so that they can be stored in the primary cache, processing speed increases should be expected. In addition, greater improvements should be expected for MMX processing because of its higher memory bandwidth requirements. These expectations are generally verified as shown in table 2. Even the conventionally coded routines are faster (on a per-pixel basis) for the small image sizes but the MMX routines benefit to a substantially greater degree.

<table>
<thead>
<tr>
<th>Filter description</th>
<th>Conditions</th>
<th>Conventional frames/sec.</th>
<th>MMX frames/sec.</th>
<th>Performance Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>digital derivative (edges)</td>
<td>300,000 passes</td>
<td>5618</td>
<td>35460</td>
<td>6.33X</td>
</tr>
<tr>
<td>rectangles (morphology)</td>
<td>100,000 passes</td>
<td>2424</td>
<td>10650</td>
<td>4.39X</td>
</tr>
<tr>
<td>diagonals (morphology)</td>
<td>100,000 passes</td>
<td>2278</td>
<td>10395</td>
<td>4.56X</td>
</tr>
<tr>
<td>logical (AND, OR, XOR)</td>
<td>300,000 passes</td>
<td>8285</td>
<td>29674</td>
<td>3.58X</td>
</tr>
<tr>
<td>Sum of two 8-bit images</td>
<td>300,000 passes</td>
<td>6242</td>
<td>14793</td>
<td>2.37X</td>
</tr>
<tr>
<td>Sum of two 16-bit images</td>
<td>300,000 passes</td>
<td>8696</td>
<td>15552</td>
<td>1.79X</td>
</tr>
<tr>
<td>Difference of two 8-bit images</td>
<td>300,000 passes</td>
<td>4277</td>
<td>35460</td>
<td>8.29X</td>
</tr>
</tbody>
</table>

Table 2 – Using 160 x 10 resolution input image(s).

5.3 A simple coding strategy for a small primary cache

While using small images provides a significant performance boost, this is clearly not a valid approach for most applications since image sizes are generally much larger. The best approach is to improve the memory bandwidth using an approach such as a larger primary cache. As noted previously, as the image data from a large image is processed it will be continually flushed out of the primary cache with a corresponding loss of processing speed.
However, there is at least one approach that can be employed for some algorithms to alleviate this problem. It will, however, cause an increase in coding complexity. The basic idea here is to process image data on a line by line basis in which several different algorithms process a given line of pixels consecutively. Since relatively few pixels are being processed at one time, they will remain in the primary cache for rapid accessing.

5.4 The Intel Image Processing Library

Intel has created an image processing library that contains functions similar to those described in this paper including arithmetic, logical and morphological operations.\(^6\) The library is designed to take full advantage of multimedia hardware when available. Intel has released performance gain results which are similar to the results included in the paper with and without MMX hardware.\(^7\) Additional performance comparisons are planned in the future to verify the performance gain results obtained in the paper with the Intel image processing library results.

6. CONCLUSIONS

Clearly the new multimedia hardware provides dramatic image processing capabilities for machine vision applications without the need for expensive custom hardware. Many additional real-time vision applications can be addressed with the MMX architecture in a very inexpensive manner. While speed increases have not been as great as anticipated, they are still very significant and offer the vision engineer the ability to address many applications that previously were too constrained by cost.

ACKNOWLEDGEMENTS

This project was supported in part by a National Science Foundation Research Experiences for Undergraduates grant and funding from the Center for Engineering Education and Practice at the University of Michigan-Dearborn.

REFERENCES

1. Error! Reference source not found.
2. Error! Reference source not found. Section 11.5.9
3. Error! Reference source not found. Section 4.5
4. Error! Reference source not found.
6. Error! Reference source not found.
7. Error! Reference source not found.