Author:

John E Lecky, Director of Machine Vision Software, Imaging Technology Inc.

He received the B.S. degree in mechanical and aerospace engineering from Princeton University in 1984 and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Vermont in 1987 and 1999. He started Lecky Engineering and Development Co. in 1985. In 1991, he founded Intelec Corporation which designed and developed complete robotic vision systems. Intelec was acquired by Imaging Technology Incorporated in 1995, where he was Director of Machine Vision Software Products until late '98. He is now President of Lecky Engineering, LLC, and a Member of the Technical Staff at Hill Associates, Inc.

In this technical feature, we explore in detail the process of optimizing a simple machine vision application for MMX. This exercise reveals many interesting properties, features, and computation idiosyncrasies common to machine vision algorithm optimization in general

Geovany A. Borges.

Turbocharged Algorithms

How to optimize a machine vision application for MMX

MMX technology for accelerating multimedia applications has become quite commonplace in recent months. Processors from Intel, Cyrix, and AMD are now available with the special hardware additions necessary to implement MMX. Many of the core requirements of multimedia processing overlap with industrial machine vision requirements, and so it's natural that the vision community benefit from this new computational capacity.

This article will explore, in detail, the process of optimizing a simple machine vision application for MMX. This exercise reveals many interesting properties, features, and computation idiosyncrasies common to machine vision algorithm optimization in general.

What is MMX Technology?

Intel's MMX Technology adds several new data types and some specialized machine language instructions to an MMX-compliant CPU. The new data types allow handling of 64-bit data. This is accomplished by reassigning 64 bits of each of the eight 80-bit floating-point registers as MMX registers. The 64-bit registers may be thought of as eight 8-bit bytes, four 16-bit words, two 32-bit double words, or one 64-bit quadword.

Since the MMX hardware and floating-point unit share registers, floating-point and MMX instructions cannot normally be intermixed without severe performance penalties. It presently takes around 50 clock cycles to toggle the floating-point register set between floating point use and MMX operation.

In addition to the new 64-bit integer register set, MMX defines some new CPU instructions that allow manipulation of these quantities in parallel. When an operation is performed on one byte in a 64-bit register, the same operation may be performed on the other 7 bytes simultaneously. In addition, a 4-wide parallel multiply-accumulator allows 4 16-bit quantities to be multiplied by 4 other 16-bit quantities and partially summed into two sums of two multiplies each in a single instruction. This

powerful operation speeds traditional image processing functions, such as convolution and morphology.

The list below shows the complete set of new instructions available in MMX for manipulating 64-bit data.

- ADDITION/SUBTRACTION. Add or subtract 8 bytes, 4 words, or 2 doublewords in parallel. Also includes saturation hardware to prevent overflow or underflow wraparound.
- COMPARE. Compare bytes, words, or doublewords to build a Boolean mask which can be used to selectively pick elements in subsequent operations.
- MULTIPLY. Multiply four 16-bit words in parallel, producing four 16-bit truncated products.
- MULTIPLY/ACCUMULATE. Multiply four pairs of 16-bit operands and sum the first two products and the last two products to give two 32-bit sums.
- SHIFT. Arithmetic and logical shifts and rotates by word, doubleword, or quadword.
- PACK/UNPACK. Useful for converting between 8-, 16-, and 32-bit data.
- LOGICAL. And, Or, Xor; up to 64 bits.
- MOVE. Move 32 or 64 bits between MMX register and memory or other MMX register, or move 32 bits between MMX registers and integer registers.

The best way to understand the operation of these instructions and data types is by example. The example algorithm to be developed which is briefly described is Statistical Variance.

The Variance Algorithm

The variance algorithm is commonly used in machine vision for presence/absence or flaw detection. Variance asks the question, "Do you see something?" It does this by measuring the width of the intensity distribution of the pixel values. If all values are more or less the same, the variance is small. If the values vary, with some dark and some light, the variance is large. Other favourable properties of variance include:

- Variance is a real machine vision algorithm, and so the properties illustrated in optimizing it have real-world application.
- The variance calculation is simple, but not trivial, allowing us to roll our sleeves up a bit, but not requiring that we get lost in 20-page code listings.
- Variance is not a typical multimedia algorithm, and so receives little treatment in the mainstream MMX literature. Therefore, it should reveal more about the machine vision twist of MMX.

A key difference between machine vision algorithms and multimedia or image processing algorithms is that machine vision algorithms frequently produce a reduction; that is, many pixels go in, and only a handful of results come out. In the case of variance, our input can be 1 million pixels, and the output is just a single double-precision floating-point number. This behaviour is actually sub-optimal for direct MMX implementation, and requires some different approaches.

- Another important feature of machine vision algorithm implementations is that they frequently work on odd-shaped data. Instead of processing entire images, machine vision algorithms are concerned only with small Regions-Of-Interest (ROIs), requiring the algorithms to behave properly when started on odd boundaries, or when working with line lengths that are not multiples of "nice" integers like 4, 8, or 16. Real machine vision algorithms always need a few extra pieces to deal with the "rough edges."

Testing Setup

The code fragments in this article were all developed and tested in Microsoft Visual C++ 5.0, using a 266 MHz Intel PentiumPro system with 128 MB of memory running Microsoft Windows NT 4.0. Information is provided on code compiled with no optimization, as well as code automatically optimized by the compiler for maximum speed. All timing information is for a 1023 x 1023 8-bit image, with the odd image size chosen to verify that the algorithms work properly on odd-sized inputs.

Direct Implementation

Listing 1 shows a standard implementation for variance on image data. It is assumed that the width (dx) and height (dy) of the image data are passed to the function, along with a row-address-table (rat). The row address table is a vector of dy elements, each pointing to the start of an individual row of pixel data. This is a good selection for implementing machine vision data where ROIs frequently have highly variable sizes.

Listing 1: Classic variance with double accumulators

```
double Variance0(BYTE** rat,int dx,int dy)
{
   double sum=0;
   double sumsq=0;
   for(int x=0; x<dx; x++)
         for(int y=0; y<dy; y++)
         {
               double pixel=rat[y][x];
               sum+=pixel;
               sumsq+=pixel*pixel;
         }

   double n = (double)dx*(double)dy;

   if(n>1) return (n*sumsq - sum*sum)/n/(n-1.);
   else return 0;
}
```

- Run Time with Compiler Optimizer 147.8 ms
- Run Time without Compiler Optimizer 221.8 ms

We'll develop seven more versions of this algorithm as we optimize it to get the final run time down to 14.7 ms, almost exactly 10 times the speed of this initial version.

Data Size Optimization

The first problem with the implementation in Listing 1 is the use of double precision floating point math inside the inner loop. Eliminating the inner loop floating point is important for a number of reasons:

- Floating point operations are slower than integer operations and standard CPUs have only one floating point pipeline, limiting the ability to do concurrent processing.
- MMX operations require the reassignment of the floating point registers, making simultaneous floating point and MMX operations inefficient.
- The MMX arithmetic unit only processes integers, so floating-point algorithms are difficult, if not impossible, to speed up using MMX technology.

The sum of the squares of the pixel values will always contain the larger value of the two accumulators. Since the largest square is 255 * 255 = 65,025, the worst case minimum number of pixels that can be accumulated into a 32-bit unsigned integer is 4,294,967,296 / 65,025 = 66,051 pixels. This is roughly a 256 x 256 ROI.

Today, it is desirable to be able to handle images of at least 1024 x 1024 without error. If overflow occurs, it will only occur on the sum of the squares; this would make the final variance calculation result in a negative number, which obviously makes no sense.

For this reason, we can use 32-bit accumulators in the optimized algorithm; only in the event of a negative final result will we have to recompute the sum of the squares using a larger accumulator.

The converted algorithm with 32-bit accumulation is shown in Listing 2. In addition, the final variance calculation has been split off as a separate function. In testing, this final calculation was found to run in about 0.002 ms, and is therefore insignificant in the timing analyses to follow.

Listing 2: Variance with unsigned accumulators

```
double Variance1(BYTE** rat,int dx,int dy)
{
   unsigned sum=0;
   unsigned sumsq=0;
   for(int x=0; x<dx; x++)
         for(int y=0; y<dy; y++)
         {
               unsigned pixel=rat[y][x];
               sum+=pixel;
               sumsq+=pixel*pixel;
         }
return FinalCalc(dx,dy,sum,sumsq);
}

double FinalCalc(int dx,int dy,unsigned sum,unsigned sumsq)
{
   double n = (double)dx*(double)dy;
   double dsum=(double)sum;
```

```
double dsumsq=(double)sumsq;

if(n>1) return (n*dsumsq - dsum*dsum)/n/(n-1.);
else return 0;
```

• Run Time with Compiler Optimzer 121.6 ms

• Run Time without Compiler Optimzer 184.5 ms

This is not a dramatic speed improvement, but it does set the stage for further improvements by eliminating the floating-point computation.

Using a Pointer Limit

The next optimization to consider is elimination of the inner for loop. œFor" loops are complex structures requiring counter variables; counter variables place an additional strain on the CPU register pool, which in the Intel architecture is uncomfortably small. Using a pointer increment with an end-of-line limit greatly increases speed since the incrementing pointer is also the œloop counter." This technique is shown in List 3.

Listing 3: Using pointer limits

```
double Variance2(BYTE** rat,int dx,int dy)
{
  unsigned sum=0;
  unsigned sumsq=0;
  for(int y=0; y<dy; y++)
  {
      BYTE* pp=rat[y];
      BYTE* ppFinal=pp+dx;
      while(pp<ppFinal)
      {
          unsigned pixel=(unsigned)*pp++;
          sum+=pixel;
          sumsq+=pixel*pixel;
      }
  }
  return FinalCalc(dx,dy,sum,sumsq);
}
```

• Run Time with Compiler Optimzer 26.6 ms

• Run Time without Compiler Optimzer 54.6 ms

This is a dramatic improvement over prior implementations, but we can still improve on this result by nearly 50%.

Loop Unrolling

The next way to reduce inner loop overhead is through loop unrolling. This technique reduces the number of inner loop iterations by repeating the inner loop operations several times. In a reduction algorithm such as variance, it is not acceptable to go beyond the end of the line and process extra pixels in the event that dx is not an exact multiple of the pipeline size. For this reason, we must do something else to make sure that the exact number of pixels is processed per line.

In this implementation, a four-at-a-time pipeline processes as many pixels as possible, then the final 0-3 pixels are processed one at a time. This concept will be important in the MMX implementation coming up, since the MMX registers will load 8 pixels at a time.

Listing 4: Loop Unrolling

```
double Variance3(BYTE** rat,int dx,int dy)
{
  if(dx<4) return Variance2(rat,dx,dy);

  unsigned sum=0;
  unsigned sumsq=0;
  unsigned pixel;
  for(int y=0; y<dy; y++)
  {
      BYTE* pp=rat[y];
      BYTE* ppFinal=pp+dx;

      // shorten up the limit and process 4 pixels per loop
      BYTE* ppUnwrap=ppFinal-3;
      while(pp<ppUnwrap)
      {
          pixel=(unsigned)*pp++;
          sum  += pixel;
          sumsq += pixel*pixel;
          pixel=(unsigned)*pp++;
          sum  += pixel;
          sumsq += pixel*pixel;
          pixel=(unsigned)*pp++;
          sum  += pixel;
          sumsq += pixel*pixel;
          pixel=(unsigned)*pp++;
          sum  += pixel;
          sumsq += pixel*pixel;
      }

      // get the 'rough' 0-3 pixels at the end of the line
      while(pp<ppFinal)
      {
          pixel = (unsigned)*pp++;
          sum  += pixel;
          sumsq += pixel*pixel;
      }
  }

  return FinalCalc(dx,dy,sum,sumsq);
}
```

• Run Time with Compiler Optimzer 22.3 ms

• Run Time without Compiler Optimzer 50.5 m

Unfortunately, this extra work does not add much speed to the algorithm. Obviously, the compiler optimizer was already doing a pretty good job of keeping loop overhead small to begin with. To make any significant progress, we'll have to drop down into assembly language.

First Assembly Version

The C version from Listing 3 (No unrolling) is translated to PentiumPro assembler in Listing 5 below. This is a mixed C/Assembler version that allows the assembly function to have a C header for easy inclusion in C code.

Listing 5: First Assembly Version

```
double Variance4(BYTE** rat,int dx,int dy)
{
  BYTE** pixels=rat;
  unsigned sum;
  unsigned sumsq;
  int w=dx;
  int h=dy;
  _asm
  {
      xor   ebx,ebx     ; row counter
      xor   ecx,ecx     ; sum
      xor   edx,edx     ; sumsq

outer_loop:
      mov   esi,pixels  ; set esi to pix[row]
      mov   esi,[esi+ebx*4]

      mov   edi,esi     ; end of line pointer
      add   edi,w

loop1:
      xor   eax,eax     ; get 1 pixel into LSB of eax
      mov   al,[esi]
      add   ecx,eax     ; add into sum
      mul   al          ; square (ax = al*al)
      add   edx,eax     ; add into squares

      inc   esi         ; next pixel
      cmp   esi,edi
      jl    loop1

      inc   ebx         ; next row
      cmp   ebx,h
      jl    outer_loop

      mov   sum,ecx     ; save accumulations
      mov   sumsq,edx
  }

  return FinalCalc(dx,dy,sum,sumsq);
}
```

• Run Time (Assembly, No Optimizer) 23.3 ms

This version actually runs more slowly tha the unrolled C! To go faster than optimize C, we'll have to optimize the assembly language. The point here is that since the compiler is already doing a good job of generating assembly language, we can onl achieve further improvements if we build assembly code more cleverly than the compiler can. The unrolled version is the next place to try.

Second Assembly Versio

Implementing the loop unrolling metho-dology of Listing 5 is shown in Listing 6.

Listing 6: Loop Unrolling in Assembler

```
double Variance5(BYTE** rat,int dx,int dy)
{
  if(dx<4) return Variance2(rat,dx,dy);

  BYTE** pixels=rat;
  unsigned sum;
  unsigned sumsq;
  int w=dx;
  int h=dy;
  _asm
  {
      xor   ebx,ebx     ; row counter
      xor   ecx,ecx     ; sum
      xor   edx,edx     ; sumsq

outer_loop:
      mov   esi,pixels  ; set esi to pix[row]
      mov   esi,[esi+ebx*4]

      mov   edi,esi     ; end of line pointer - 3
      add   edi,w
      sub   edi,3

loop2:
      xor   eax,eax     ; get pixel into al
      mov   al,[esi]
      add   ecx,eax     ; add into sum
```

```
        mul   al    ; square
        add   edx,eax    ; add into sumsq

        xor   eax,eax    ; repeat 3 times (unroll)
        mov   al,[esi+1]
        add   ecx,eax
        mul   al
        add   edx,eax

        xor   eax,eax
        mov   al,[esi+2]
        add   ecx,eax
        mul   al
        add   edx,eax

        xor   eax,eax
        mov   al,[esi+3]
        add   ecx,eax
        mul   al
        add   edx,eax

        add   esi,4  ; bump pointer by 4
        cmp   esi,edi
        jl    loop2

        add   edi,3
oop3:
        cmp   esi,edi     ;test for end-of-line
        jge   row_done
        xor   eax,eax     ; process one-at-a-time
        mov   al,[esi]    ; for last 0-3 pixels on line
        add   ecx,eax
        mul   al
        add   edx,eax
        inc   esi
        jmp   loop3

ow_done:
        inc   ebx  ; next row of pixels
        cmp   ebx,h
        jl    outer_loop

        mov   sum,ecx    ; save accumulations
        mov   sumsq,edx
    }

    return FinalCalc(dx,dy,sum,sumsq);
}
```

• Run Time (Assembly, No Optimizer) 19.2 ms

Now we've broken the 20 ms barrier, but we can still improve more by moving to MMX.

Version 7: Using MMX

While several more optimizations are possible in the assembly program of Listing 6, to achieve any significant reduction in execution time will require more aggressive approach.

The multiply/accumulator in the MMX hardware can be used to generate sums (by multiplying by 1) and sums of squares (by multiplying pixels by themselves). A direct MMX implementation must be done carefully, however.

Even though there is only one MMX instruction pipeline, the execution flow is still a pipeline; that is, instructions are executed in a series of steps, and can be dynamically paired to be in the pipe at the same time. Furthermore, the two integer execution pipelines on the CPU can often be simultaneously performing other non-MMX operations.

Listing 7 shows a good first try at an MMX implementation.

Listing 7: Initial MMX Version

```
double VarianceMMX0(BYTE** rat,int dx,int dy)
{
  int h = dy;
  int w = dx;
  BYTE** pixels = rat;
  int nThrown = 4 - (w&3);
  unsigned sum = 0;
  unsigned sumsq=0;
  __asm
{
  mov     ebx,w; inner loop count
  shr     ebx,2 ; nCols/4

  pxor    mm6,mm6
  movd    mm6,nThrown ; shifts for ragged pixels
  psllq   mm6,3

  pxor    mm5,mm5 ;generate constant 1 in ea
            word of mm5
  pcmpeqw mm4,mm4
  psubw   mm5,mm4

  pxor    mm4,mm4 ; sum
  pxor    mm3,mm3 ; sums
  mov     eax,0 ; eax=row# to process next

outer_loop:
  mov     esi,pixels   ; set esi to pix[row]
```

```
    mov     esi,[esi+eax*4]
    mov     ecx,ebx     ; loop count (nCols/4)

inner_loop:
    movq    mm0,[esi]  ; get 8 pixels
    punpcklbw mm1,mm0 ; unpack 4 words into mm1
    psrlw   mm1,8      ; move pixels into LSBs of
              words
    movq    mm2,mm1 ; copy for squaring

    pmaddwd  mm1,mm5 ; 2 sums of 2 pixels as
                dwords
    paddd           mm4,mm1          ; accumulate sums
    pmaddwd  mm2,mm2 ; these are pixel squares
    paddd           mm3,mm2          ; accumulate squares

    add     esi,4  ; we just processed 4 pixels
    loop  inner_loop        ; loop until row done

; redo inner loop operation with partial mask to get odd pixels

;  at end of row
    movq    mm0,[esi]
    psllq   mm0,mm6 ; trash out pixels past dy-1
    punpcklbw mm1,mm0
    psrlw   mm1,8
    movq    mm2,mm1

    pmaddwd  mm1,mm5 ; 2 sums of 2 pixels as
                dwords
    paddd           mm4,mm1          ; accumulate sums
    pmaddwd  mm2,mm2 ; these are pixel squares
    paddd           mm3,mm2

    inc     eax   ; next row
    cmp     eax,h ; loop until done all rows jl

outer_loop

    movq    mm7,mm4 ; sum is sitting in two
                pieces in mm4
    psrlq   mm7,32    ; slide upper piece to
                bottom of mm3
    paddd           mm4,mm7          ; now we've got the
                complete sum
    movd    sum,mm4

    movq    mm7,mm3 ; SAME FOR SUM SQ
    psrlq   mm7,32    ; slide upper piece to
                bottom of mm3
    paddd           mm3,mm7          ; now we've got the
                complete sumsq
    movd    sumsq,mm3
    emms                ; done MMX processing
    }

  return FinalCalc(dx,dy,sum,sumsq);
}
```

Run Time (Assembly- No Optimizer)
15.7 mS

Version 8: Scheduled MMX

To get the most speed out of MMX, we have to think in terms of instruction scheduling. There are many rules regarding MMX scheduling. The most critical involve output operand collision.

If one MMX instruction modifies and MMX register, and the next instruction uses that register as an input, the pipeline will stall for one or more clocks. Eliminating these stalls can significantly increase throughput.

In Listing 7, several such stalls are apparent. For example, the code sequence

```
                pmaddwd
mm1,mm5         ; 2 sums of 2 pixels as
dwords
                paddd
mm4,mm1         ; accumulate sums
```

stalls since the paddd instruction must wait for the pmaddwd instruction to complete updating mm1 before it can proceed. Listing 8 shows a much more aggressive MMX implementation in which all 8 pixels from the memory fetch are processed by splitting four into one register and four into another. Since the data stream has been partitioned into two pieces, instructions can be scheduled to work alternately on different pieces, and therefore keep the pipeline busier. In addition, the non-MMX operations, like pointer increments and comparisons, are interspersed into the MMX code where the can execute in parallel with MMX instructions.

Listing 8: MMX Scheduling

```
double VarianceMMX3(BYTE** rat,int dx,int dy)
{
  int h = dy;
  int w = dx;
  BYTE** pixels = rat;
  int nThrown = 8 - (w&7);
  unsigned sum = 0;
  unsigned sumsq=0;
  __asm
{
    mov     ebx,w; inner loop count
    shr     ebx,3 ; nCols/8

    pxor    mm6,mm6
    movd    mm6,nThrown ; shifts for ragged pixels
    psllq   mm6,3

    pxor    mm5,mm5 ; generate constant 1 in ea
              word of mm5
    pcmpeqw  mm4,mm4
    psubw   mm5,mm4

    pxor    mm4,mm4 ; sum
    pxor    mm3,mm3 ; sumsq
    mov     eax,0 ; eax=row# to process next

outer_loop:
    mov     esi,pixels   ; set esi to pix[row]
    mov     esi,[esi+eax*4]
    mov     ecx,ebx     ; loop count (nCols/8)

inner_loop:
    movq    mm0,[esi]  ; get 8 pixels
    add     esi,8  ; we just processed 8 pixels
    punpcklbw mm1,mm0 ; unpack 4 wds into mm1,
              next 4 in mm2
    punpckhbw       mm2,mm0
    psrlw   mm1,8      ; move pixels into LSBs of
              words
    psrlw   mm2,8
    movq    mm0,mm1 ; copy first 4 in mm0, last
              4 in mm7
    movq    mm7,mm2

    pmaddwd  mm1,mm5 ; compute sums by
                multiplying by 1
    pmaddwd  mm2,mm5
    pmaddwd  mm0,mm0 ; compute squares
    pmaddwd  mm7,mm7

    paddd           mm1,mm2          ; sum the sums
    paddd           mm0,mm7          ; sum the squares
    paddd           mm4,mm1          ; accumulate the sums
    paddd           mm3,mm0          ; accumulate the squares

    loop        inner_loop  ; loop until row done

; redo inner loop operation with partial mask to get odd pixels

;  at end of row
    movq    mm0,[esi]  ; get next 8 pixels; some to
              ignore
    inc     eax   ; prepare for next row
    psllq   mm0,mm6 ; trash out pixels past dy-1
    cmp     eax,h ; sets comp flag for jl instr
              way below
    punpcklbw mm1,mm0 ; unpack 8 pixels into 4
                and 4
    punpckhbw       mm2,mm0
    psrlw   mm1,8
    psrlw   mm2,8
    movq    mm0,mm1 ; copies for squaring
    movq    mm7,mm2

    pmaddwd  mm1,mm5 ; compute sums
    pmaddwd  mm2,mm5
    pmaddwd  mm0,mm0 ; compute squares
    pmaddwd  mm7,mm7

    paddd           mm1,mm2          ; sum the sums
    paddd           mm0,mm7          ; sum the squares
    paddd           mm4,mm1          ; accumulate the s
    paddd           mm3,mm0          ; accumulate the s

    jl          outer_loop

    movq    mm7,mm4 ; sum is sitting in two
                pieces in mm4
    movq    mm6,mm3 ; sumsq is sitting in two
                pieces in mm3
    psrlq   mm7,32    ; slide upper piece to
                bottom of mm7
    psrlq   mm6,32    ; slide upper piece to
                bottom of mm6

    paddd           mm4,mm7          ; now we've got th
                complete sum
    paddd           mm3,mm6          ; now we've got th
                complete sumsq
    movd    sum,mm4
    movd    sumsq,mm3
    emms                ; done MMX processing
    }

  return FinalCalc(dx,dy,sum,sumsq);
}
```

Run Time (Assembly- No Optimizer)
14.7 mS

Conclusion

There are still some more optimization could be made, but the most importan have been taken.

...to get the most speed o of MMX, w have to thin in terms o instruction scheduling