Transformation of Beamforming Using MMX Instructions

Bret Martin and Maimoon Nasim

May 14, 2004

1 Introduction

Beamforming is a important technique with applications in active and passive sonar, space division multiple access, E911 mobile position location, and cellular/PCS base station antenna design[4]. We look at a basic delay and add beamforming algorithm and use MMX instructions to optimize the algorithm. By optimizing the algorithm, we can more easily use higher frequency signals and in active sonar or similar applications we can get finer resolution of distance by examining shorter periods of the signal.

1.1 Beamforming

In delay and add beamforming, a delays are added to the signals from a linear array of microphones in order to simulate it facing different directions without needing to move it. To "look" in a certain direction, the delays between when the peak of a wave coming from that direction would hit each of the array elements can be found and used to index into a buffer of the data from the array. In this way, instead of looking scanning on only one

direction, all directions can be used at once by using different delays with the same data. By being able to scan each beam quickly, more beams can be looked at in the same period giving better resolution for the direction. Alternatively, one can scan the beams more often which would allow you to have better resolution on the time the wave hit the array; this would be useful for getting better distance measurements in active sonar.

1.2 MMX Instructions

MMX technology is designed to accelerate multimedia and communications applications by including new instructions and data types that allow certain applications to achieve higher performance. It exploits the parallelism inherent in many multimedia and communications algorithms, yet maintains full backward compatibility.

Most multimedia and communication applications consist of algorithms that perform localized, recurring operations on small chunks of data. For example, most audio data is represented in 16-bit quantities. Most of the multimedia and communication applications show the following common fundamental characteristics. Small integer data types (for example: 8-bit pixels, 16-bit audio samples), small, highly repetitive loops, frequent multiplies and accumulates, compute-intensive algorithms, highly parallel operations.

MMX Technology exploits a technique called Single Instruction, Multiple Data (SIMD) that can drastically improve the performance of these types of applications. In SIMD a single instruction operates on multiple pieces of data in parallel. This allows programmers to pack several small chunks of data into a 64-bit register and then use a single instruction to tell the CPU to perform a specific operation on each of those data elements. For example, with a single MMX instruction, up to eight numbers can be added together simultaneously. There are a total of 57 new instructions that, support parallel operations on byte, word, and double-word data elements, and allow you to

perform 64-bit integer operations (see Figure 1). MMX also introduced four new data types along with addition of 8, 64bit MMX registers. Three of these data types are packed data types and the other is a 64-bit data types.

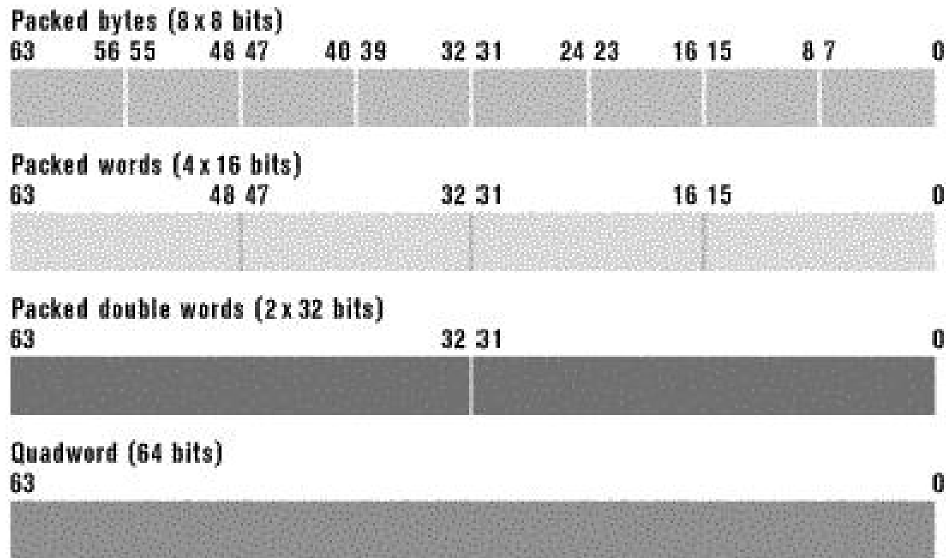

Figure 1: MMX Data Types

Because this technology allows several small data elements to be manipulated simultaneously, it provides a way to drastically improve the performance and quality of multimedia and communication applications. For example, suppose an application manipulates the pixels in a 256-color image, and each pixel needs to be represented by a single byte. It's easy to see that much better performance is offered by MMX Technology instead of traditional assembly language instructions because the processor can operate on eight pixels simultaneously exploiting the inherent parallelism.

Saturation Arithmetic is another very useful enhancement in MMX. If the sum of the two 8-bit numbers is too large to be represented with 8 bits, there are two options either to truncate the overflow bits, or to carry the extra bits over to another byte. This arithmetic technique is often called wraparound. Conversely, in saturation mode the result of an operation that would cause

an overflow or an underflow is set to the range limit for the data type instead of rolling over. For example, if you add two unsigned, single-byte numbers whose sum is greater than 255, the saturation result is simply 255.

Saturation arithmetic operations are of great importance in video and image applications. Imagine superimposing one image onto another by adding the colors from corresponding pixels in each image. Saturation mode allows completing this task without having to correct for wraparound, thus increasing the performance of the operation. Similar benefits occur in other data processing algorithms as well.

The MMX instructions cover several functional areas including basic arithmetic operations such as add, subtract, multiply, arithmetic shift and multiply-add, comparison operations. Conversion instructions are used to convert between the new data types i.e. to pack data together, and unpack from small to larger data types. Logical operations such as AND, OR, and XOR, shift operations. Data transfer instructions for MMX register-to-register transfers or 64-bit and 32-bit load/store to memory and state management instruction to handle MMX to floating point transitions.

2 Implementation

In order to optimize the beamforming algorithm with MMX instructions we needed to select a hardware model for the system the code would run on. We based our choices off of a simple beamforming sonar which has an array of four microphones and obtains sixteen bit samples from each of them at a frequency of 100 kHz. These samples are stored in a data buffer and periodically, a beamforming function is called to process the data in the buffer. The output from the function is an array of thirty two bit values, one for each beam we want to calculate. Code for the original generic beamforming function can be found in Appendix A and the assembly code generated by gcc is in Appendix

B.

Our first step to optimize this algorithm is to unfold the innermost loop. This produces code that does the four loads and adds all in one loop body which removes the overhead from the loop and its bounds checking and the multiple stores to and reads from memory for the location accumulating the sum.

Next, in order to make use of the MMX instructions, we unfold the second loop by a factor of four, so we will summing four rows of the window at once in order to make use of the four sample wide MMX registers and the word operations provided in MMX. We are able to use PADDSW to do the adds for all four rows at once.

Next to optimize are the four squarings that need to be done. Since we also need to sum the results of these squarings, we can use PMADDWD to do the multiplications and two of the adds needed to reduce the four results to one result. The final add is done by shuffling the values of our results register into another so that the two values line up to be added.

Finally, the original code keeps the temporary values of the results in memory which causes the value to be loaded, modified and stored again each iteration. To get rid of this problem, we store the temporary result in one of the general purpose registers and move the results from the MMX registers to the general purpose registers and do the add.

The second loop could be further unfolded; however, with hardware branch prediction and register renaming on the processor we were running our code on it would not give much if any benefit since it would be done for us in hardware.

3 Results & Conclusion

We observed an overall improvement of a factor of 2 in the performance. Loop unrolling as a means for instruction level parallelism provides an excellent speedup. It is worth observing that loop unrolling has an effect of hiding memory latencies as well.

Software techniques like software pipelining and software perfecting could have been used to further achieve performance gains. Software pipelining is a technique in which memory assesses and computations are overlapped from different iterations in a program loop. This technique can provide performance gains but requires large number of registers. In software perfecting the non-blocking prefect instructions bring the data into cache before its requested by the application. Perfecting is reported to have significant performance gains especially in newer processor where a cache miss can be very expensive.

Although good compilation support is not provided for MMX instructions, hand coding the computationally intensive part is worth the effort as the speedups achieved can be significantly high.

References

[1] G. Allen, What Is Beamforming?, http://www.ece.utexas.edu/~allen/Beamforming/.

[2] Curtis Technology Ltd., Principles of Sonar Beamforming, http://www.curtistech.co.uk/papers/beamform.pdf, 1998.

[3] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference,* 2002.

[4] M. G. Morrow, T. B. Welch, C. H. G. Wright, G. York, Teaching Real-time Beamforming with the C6211 DSK and MATLAB, *Proceedings of DSPS Fest,* 2000.

A Original Beamforming C Code

```
#define NUMCHAN 4
#define NUMBEAMS 21
#define WINDOWLENGTH 20
long beams[NUMBEAMS];
short data[NUMCHAN][1000];
void beamform(void) {
    int Delay = 10;
    int Offset = 400;
    int i, k, j;
    long data2 [WINDOWLENGTH];
    for (i=0; i<NUMBEAMS; i++) {
        beams[i] = 0;
        for (k=0; k<WINDOWLENGTH; k++) {
            data2[k] = 0;
            for (j=0; j<NUMCHAN; j++) {
                data2[k] += data[j][Offset + k - j * Delay];
            }
            data2[k] *= data2[k];
            beams[i] += data2[k];
        }
        Delay--;
    }
}
```

B Assembly Generated by GCC

Comments were added by hand afterwards to show how each part of the assembly corresponds to the original code.

```
        .file   "beamform.c"
        .text
.globl beamform
        .type   beamform,@function
beamform:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %esi
        pushl   %ebx
        subl    $112, %esp
        movl    $10, -12(%ebp)          ; Delay = 10;
        movl    $400, -16(%ebp)         ; Offset = 400;
        movl    $0, -20(%ebp)           ; i = 0;
.L2:
        cmpl    $20, -20(%ebp)          ; i < NUMBEAMS
        jle     .L5
        jmp     .L1
.L5:
        movl    -20(%ebp), %eax
        movl    $0, beams(,%eax,4)      ; beams[i] = 0;
        movl    $0, -24(%ebp)           ; k = 0;
.L6:
        cmpl    $19, -24(%ebp)          ; k < WINDOWLENGTH
        jle     .L9
        jmp     .L7
.L9:
```

```
        movl    -24(%ebp), %eax
        movl    $0, -120(%ebp,%eax,4)    ; data2[k] = 0;
        movl    $0, -28(%ebp)            ; j = 0;
.L10:
        cmpl    $3, -28(%ebp)            ; j < NUMCHAN;
        jle     .L13
        jmp     .L11
.L13:
        movl    -24(%ebp), %esi
        movl    -24(%ebp), %ebx
        movl    -28(%ebp), %ecx
        movl    -24(%ebp), %eax
        movl    -16(%ebp), %edx
        addl    %eax, %edx               ; Offset + k
        movl    -28(%ebp), %eax
        imull   -12(%ebp), %eax          ; j * Delay
        subl    %eax, %edx               ; Offset + k = j * Delay
        movl    %ecx, %eax
        imull   $1000, %eax, %eaxa       ; j * 1000
        addl    %edx, %eax
        movswl  data(%eax,%eax),%eax     ; data[j][Offset+k=j*Delay]
        addl    -120(%ebp,%ebx,4), %eax ; + data2[k]
        movl    %eax, -120(%ebp,%esi,4) ; data2[k] = "
        leal    -28(%ebp), %eax
        incl    (%eax)                   ; j++;
        jmp     .L10                     ; end j loop
.L11:
        movl    -24(%ebp), %ecx
        movl    -24(%ebp), %eax
        movl    -24(%ebp), %edx
```

```
        movl    -120(%ebp,%eax,4), %eax ; data2[k]
        imull   -120(%ebp,%edx,4), %eax ; data2[k] * data2[k]
        movl    %eax, -120(%ebp,%ecx,4) ; data2[k] = data2[k]^2
        movl    -20(%ebp), %ecx
        movl    -20(%ebp), %edx
        movl    -24(%ebp), %eax
        movl    -120(%ebp,%eax,4), %eax ; data2[k]
        addl    beams(,%edx,4), %eax    ; data2[k] + beams[i]
        movl    %eax, beams(,%ecx,4)    ; beams[i] = data2[k] + beams[i];
        leal    -24(%ebp), %eax
        incl    (%eax)                  ; k++;
        jmp     .L6                     ; end k loop
.L7:
        leal    -12(%ebp), %eax
        decl    (%eax)                  ; Delay--;
        leal    -20(%ebp), %eax
        incl    (%eax)                  ; i++;
        jmp     .L2                     ; i loop
.L1:
        addl    $112, %esp
        popl    %ebx
        popl    %esi
        popl    %ebp
        ret
.Lfe1:
        .size   beamform,.Lfe1-beamform
        .comm   beams,84,32
        .comm   data,8000,32
        .ident  "GCC: (GNU) 3.2.3"
```

C Modified Assembly Using MMX Instructions

```
        .file   "beamform.c"
        .text
.globl beamform
        .type   beamform,@function
beamform:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %esi
        pushl   %ebx
        subl    $112, %esp
        movl    $10, -12(%ebp)          ; Delay = 10;
        movl    $400, -16(%ebp)         ; Offset = 400;
        movl    $0, -20(%ebp)           ; i = 0;
.L2:
        cmpl    $20, -20(%ebp)          ; i < NUMBEAMS
        jle     .L5
        jmp     .L1
.L5:
        movl    -20(%ebp), %eax
        movl    $0, beams(,%eax,4)      ; beams[i] = 0;
        movl    $0, -24(%ebp)           ; k = 0;
        xor     $ecx, $ecx              ; accum = 0;
.L6:
        cmpl    $19, -24(%ebp)          ; k < WINDOWLENGTH
        jle     .L9
        jmp     .L7
.L9:
        movl    -24(%ebp), %eax
        movl    $0, -120(%ebp,%eax,4)   ; data2[k] = 0;
```

```
        movl    -24(%ebp), %esi
        movl    -24(%ebp), %eax
        movl    -16(%ebp), %edx
        addl    %eax, %edx              ; Offset + k

        movq    data(,%edx), %mm0
        addl    $1000, %edx
        movq    data(,%edx), %mm1
        addl    $1000, %edx
        movq    data(,%edx), %mm2
        addl    $1000, %edx
        movq    data(,%edx), %mm3
        paddsw  %mm1, %mm0
        paddsw  %mm3, %mm2
        paddsw  %mm2, %mm0
        pmaddwd %mm0, %mm0
        pshufw  $0x4E, %mm0, %mm1
        paddd   %mm1, %mm0
        movd    %mm0, %ebx
        add     %ebx, %ecx
        leal    -24(%ebp), %eax
        addl    $4, (%eax)              ; k += 4
        jmp     .L6                     ; end k loop
.L7:
        leal    -12(%ebp), %eax
        decl    (%eax)                  ; Delay--;
        movl    -20(%ebp), %eax
        movl    %ecx, beams(,%eax,4)    ; beams[i] = %ecx
        leal    -20(%ebp), %eax
        incl    (%eax)                  ; i++;
```

```
        jmp     .L2                     ; i loop
.L1:
        addl    $112, %esp
        popl    %ebx
        popl    %esi
        popl    %ebp
        ret
.Lfe1:
        .size   beamform,.Lfe1-beamform
        .comm   beams,84,32
        .comm   data,8000,32
        .ident  "GCC: (GNU) 3.2.3"
```