Algorithmic-level optimization and programming-level optimization are tightly coupled with each other. Many programmers can optimize the implementation of a specific algorithm using MMX™ technology. However, without algorithmic-level optimization, the speed-up of the optimization will be limited. On the other hand, many algorithm developers can optimize the DSP algorithm in terms of the numbers of operations (multiplications or additions). Nonetheless, without implementation details, the number of operations cannot be directly translated into the number of clock cycles spent in CPU. Moreover, many algorithms can accomplish the same task. For best performance of DSP/multimedia applications on personal computers, we should consider algorithm-MMX co-optimization.

One way to increase performance of digital signal processing is to execute several computations in parallel. MMX is one of the techniques that speed up software performance by performing the same operation on multiple data elements in parallel using a single instruction. However, MMX programming and designing DSP algorithms for MMX are full of twists and turns. Implementation of digital signal processing using MMX technology is a mix of a science and an art. **Matching the algorithms to MMX instruction capabilities is the key to extracting the best performance.** This chapter is covering algorithm design and algorithmic-level optimization for MMX. In this chapter, besides showing you how to optimize your code and algorithm---from a science view, we will show you how we go about optimizing ours---from an art perspective.
1. INTRODUCTION

1.1 Under Developing History

As digital signal processing finds broader areas of application, more processors are adapting to the need for DSP operations [35]. One way to boost the performance is to execute several computations in parallel, for example, using single-instruction multiple-data (SIMD) techniques. It used to be that computationally intensive, real-time multimedia application required application specific IC or digital signal processors. With the introduction of SIMD extensions to general-purpose processors, things have changed.

In 1996, Intel introduced MMX (MultiMedia eXtension) technology into Intel Pentium™ processors [37]. A set of 57 instructions were added to treat data in a SIMD fashion. They exploit the data parallelism that is often available in DSP and multimedia applications. These instructions can multiply, shift, saturating add/subtract, load/store, and do logical operations on 64 bits of data at a time. These 64 bits can be packed as two 32-bit (doublewords), four 16-bit (words), or eight 8-bit (bytes) groups of integers. The parallel operations can yield a speedup of 2 to 8 times over existing integer implementation [17,18,19,20].

Following a similar underline concept, in 1999, Intel introduced a new generation of the IA-32 microprocessor---Pentium III. This processor introduced 70 new instructions, which includes MMX technology enhancements, SIMD floating-point instructions, and cacheability instructions [25,40]. The introduction of Streaming SIMD Extensions by Intel has brought floating-point performance up to speed with 4 single-precision operations per cycle. In 2000, Intel's Pentium 4 processors further use 128-bit XMM registers for integer operations [11,21,22]. Since this is an under-developing history, this chapter uses the basic MMX as the example. Readers can extend the knowledge into advanced MMX technologies (even onto different microprocessors).

1.2 Motivation

Although MMX technology can offer much higher capability or speed, such parallel programming is easier said than done. First, different algorithms have different characteristics for parallel processing. Algorithm with repetitive, regular data movement and computation---as found in graphics, filtering, compression, decompression---typically can be improved a lot by using MMX. On the other hand, code that does not speed up typically lacks that repetitive (looping) behavior, such as decision trees [2].

Algorithm, software, and hardware must be design together for a good implementation. A multimedia application can be achieved by tons of different algorithms (some regular, while some irregular). Conventional software and hardware co-design may not be suitable for today's multimedia application. For opti-
mal implementation, we should bear the architectural style in mind when we design or choose the algorithm for a specific task. [9].

Choosing the right one depends on users’ needs. For example, let algorithm \( i = 1, \ldots, N \) be the solutions for a multimedia signal processing task and the execution time \( T_i = T_{is} + \frac{T_{ip}}{P} \), where \( T_{is} \) is the non-parallelizable execution time, \( T_{ip} \) is the parallelizable execution time, and \( P \) the number of parallel execution units. In a uniprocessor system where \( P = 1 \), algorithm \( i \) is better if \( T_{is} + T_{ip} \) is minimal. For a multiprocessor system, algorithm \( j \) is better if \( T_{is} + \frac{T_{ip}}{P} \) is minimal. As \( P \) become larger and larger in the near future, the design/choice of algorithms should be changed. Using motion estimation as a design example, we found that the full-search block-matching algorithm is more efficiently implemented in systolic array than the hierarchical-search block-matching algorithm although the full-search block-matching algorithm needs more operations.

Consequently, the implementation of digital signal processing using MMX technology should also consider the underline MMX architecture and re-design of algorithms. Thus, in this work, we will show you

1. Basic MMX instructions and programming methodology
2. Adaptation of algorithm design for MMX technology

1.3 Chapter Organization

Considering the underline MMX architecture for designing the best algorithm, we would like to show four basic principles of implementing DSP on MMX technology:

1. Put multiple identical operations in one instruction, i.e., vectorize discrete scalar operations into SIMD operations, and use the smallest possible data type to enable more parallelism
2. Convert conditional executions into logic operations to reduce branches and increase parallelism
3. Put data into the right format for parallel execution, and arrange the data structure for better memory access
4. Reduce shuffling and maximize pairing; do not over-optimize the algorithms in terms of the number of scalar operation.

We will show some of our work in MPEG video processing as examples of the principles. MPEG video decompression consists of (1) high memory bound elements, like motion compensation, (2) high-computational parts, like IDCT, and (3) highly irregular program flows in Huffman decoder. This mix is representative for many multimedia and digital signal processing algorithms.

This chapter is organized as the following. Section 2 first reveals some basic MMX programming environment (MMX registers, data types, and instruction sets) and guidelines of instruction-level optimization. We will also demonstrate how to convert scalar operations into SIMD operations in Section 2. In Section 3,
we will convert conditional executions into logic operations, using MPEG-4 pixel padding procedure as an example. Section 3 will also show that how to access memory efficiently is important in algorithm designs because of the one-dimensional nature of the computer memory. In Section 4, we will then show an example of algorithmic-level optimization of MPEG-4 Shape-Adaptive-DCT for MMX technology. We could use a whole book to cover the details of MMX programming. In this chapter, we focus more on guiding the algorithmic development of DSP on MMX. In order to help readers understand more about topics, we also list some examples. In Section 5, we will give pointers to additional resources for readers.

1. OVERVIEW OF THE MMX TECHNOLOGY AND BASIC OPTIMIZATION PRINCIPLES

The first rule of MMX optimization is to vectorize the operations, i.e., execute multiple identical operations in one instruction. Although the name MMX implies a specific set of applications, multimedia, the new instruction set is a general-purpose implementation of the SIMD concept, as shown in Figure 1. With MMX instructions, we can execute eight 8-bit integer operations per clock cycle. MMX benefits all applications that perform the same operation repetitively on contiguous blocks of data.

(a)  
\[
\begin{array}{cc}
A1 & B1 \\
A2 & B2 \\
A3 & B3 \\
A4 & B4 \\
\end{array}
\]

(b)  
\[
\begin{array}{cccc}
A1 & A2 & A3 & A4 \\
B1 & B2 & B3 & B4 \\
\end{array}
\]

Figure 1: (a) Conventional scalar operation. To add two vectors together, we have to add each pair of the components sequentially. (b) In processors with SIMD capability, we can add two vectors in parallel using one instruction.

Since the MMX can operate multiple operands in parallel, the fundamental principle of MMX optimization is to vectorize the operation: (1) arrange multiple operands to be execute in parallel; (2) use the smallest possible data type to enable
more parallelism with the use of a longer vector; and (3) avoid the use of conditionals. In this section, we will

1. Introduce MMX programming environment---MMX registers, data types, and instruction sets
2. Show general code optimization on Intel Architecture
3. List rules of MMX instruction-level optimization
4. Give a simple example of MMX to compute reduced-resolution image

Mainly, our goal is to let readers to have a high-level understanding of the MMX programming environment. Then, we demonstrate the first rule of MMX optimization by optimizing a procedure to compute reduced-resolution images.

2.1 Programming Environment

MMX technology provide the following new extensions to the Intel Architecture programming environment:

1. Eight MMX registers (MM0 through MM7).
2. Four MMX data types (packed bytes, packed words, packed double-words, and quadwords).
3. The MMX instruction set.

MMX technology introduces 8 new 64-bit MMX registers (MM0, MM1, ..., MM7). The MMX instructions operate in parallel on byte (8 bits), word (16 bits), doubleword (32 bits), and quadword (64 bits) data type packed in these 64-bit registers. As shown in Figure 2, you can interpret the 64-bit data format in an MMX register according to the instruction that you use. MMX defines some new 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.

The MMX instructions consist of arithmetic, comparison, conversion, logical, shift, and data transfer instructions. MMX instructions follow the format shown in Figure 3. In Table 1, you can find a list of the MMX instructions with a brief description of each. For a detailed description, please refer to [17,18]. 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.** Converting data between 8-, 16-, and 32-bit format.
- **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.

Figure 2: Data formats for MMX technology instructions.

Figure 3: The MMX instruction format.

MMX intersections PCMPGE, PCMPGT, and so on produce a redundant mask per data elements of all 1s or 0s, depending on the comparison's result. This approach allows programs to use the mask with subsequent logic operations (AND, ANDN, OR, XOR) to perform conditional moves. These instructions simplify data-dependent branching. (We will see how to convert conditional executions into logic operations in Section 3.)
In addition, a four-wide parallel multiply-accumulator allows four 16-bit quantities to be multiplied by four 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.

One of the best ways to understand the operation of these instructions and data types is by example. We will show some examples on the way.
2.2 Guidelines Code Optimization

2.2.1 Understand Where the Application Spends Most of Execution Time

Before changing a line of code, it is important to identify possible areas of MMX optimization first. The following are three basic steps [12,19,33]:

1. Understand where the application spends most of its execution time.
2. Understand which algorithm is the best for MMX technology in this application.
3. Understand where data values in the application can be converted to integer (fixed-point) while maintaining the required range and precision.

The benefit of optimizing computationally intensive parts is larger than that of optimizing less intensive parts. We should first start optimization from the most computationally intensive components. There are some tools can help in isolating the computationally-intensive sections of code. In this work, we use Intel's VTune™ Performance Analyzer to identify the bottleneck of the system [2,10,24].

2.2.2 General Instruction-Level Code Optimization

After we identify possible code segments for MMX optimization, we can start code optimization. First, we review some general code optimization techniques before we examine the MMX code optimization guidelines. General code optimization guidelines are

1. Use a current generation compiler that will produce an optimized application. This will help you generate good code from the start.
2. Maximize memory access performance:
   (1) Minimize memory references,
   (2) Maximize register usage,
   (3) Prefetch data,
   (4) Arrange code to minimize instruction cache misses and optimize prefetch,
   (5) Make sure all data are aligned,
   (6) Align frequently executed branch targets on 16-byte boundaries,
   (7) Avoid partial register stalls,
   (8) Load and store data to the same area of memory using the same data sizes and address alignments,
   (9) Use the "OP REG, MEM" format whenever possible. This format helps to free registers and reduce cycles without generating unnecessary loads.
3. Minimize branching penalties:
   (1) Minimize branch instructions, for instance, unroll small loops,
   (2) Arrange code to minimize the misprediction in the branch prediction algorithm, for example, forward conditional branches are usually not taken and backward conditional branches are usually taken.
4. Use software pipelining to schedule latencies and functional unit. Unroll small loops to schedule more instructions.

2.2.3 Matching the Algorithms to MMX Instruction Capabilities

Matching the algorithms to MMX instruction capabilities is the key to extracting the best performance. Once the computationally-intensive sections of code are identified, an evaluation should be done to determine whether the current algorithm or a modified one will give the best performance. In some cases, it is possible to improve performance by changing the types of operations in the algorithm. For a same DSP application, there could be many algorithms. Say, one of the algorithms uses $x$ multiplications and $y$ additions and another uses $x'$ multiplications and $y'$ additions. Usually, multiplications are more expensive in implementations and thus one algorithm, which uses less multiplications, is better than another. However, in MMX, paired multiplications are cheap and thus, one algorithm, which use more multiplications, can be better than another.

The MMX instructions offer the best support for 8-bit and 16-bit integer data types. While some DSP can be done in integer domain, some must be done in floating-point domain. MMX can provide significant speedup in certain DSP and multimedia applications, even over hand-optimized floating-point assembly code. MMX seems well-suited for image processing applications because of plenty of contiguous, 8-bit data available and rarely require precision beyond 8 bits. On the other hand, some signal processing applications seem to cause problems due to their higher-precision requirement [3].

Since the MMX can operate multiple operands in parallel, the fundamental principle of MMX optimization is to vectorize the operation: (1) arrange multiple operands to be executed in parallel; (2) use the smallest possible data type to enable more parallelism with the use of a longer vector; and (3) avoid the use of conditionals. Use the smallest possible data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible.

2.2.4 MMX Instructions-Level Code Optimization

The following is a list of rules for MMX instruction-level optimization:

1. Do not intermix MMX instructions and floating-point instructions. MMX instructions do not mix well with floating-point instructions. MMX registers and state are aliased onto the floating-point registers and state, so no new registers or states are introduced by MMX.

2. MMX code sections should end with "emms" instructions if floating-point operations are to be used later in the program.\(^1\)

---

\(^1\) In order to maintain operating system compatibility, MMX registers are aliases to the scale floating-point registers. As we read or write to an MMX register, we read and write to one of the floating-point registers and vice versa. Thus, we cannot reply on the contents of the
3. MMX shift/pack/unpack instructions do not mix well with each other. In general, two MMX instructions can be executed at the same clock. However, only one MMX shift/pack/unpack instruction can be executed at one clock because there is only one shifter unit.

4. MMX multiplication instructions "pmull/pmulh/pmadd" do not mix well with each other. Currently, there is only one multiplication unit.

5. MMX instructions, which reference memory or integer registers, do not mix well with integer instructions referencing same memory or registers.

6. It is important to arrange data in the best way for MMX processing, e.g., structure of array, array of structure, row-wise, or column-wise arrangements. Column-wise processing in general is better than sequential row-wise processing.

2.3 A Simple MMX Programming Example

The first rule of MMX optimization is to execute multiple identical operations in one instruction. Here, we show an example of using MMX to speed up the computation of the reduced-resolution image by 2.3 times. To compute a reduced-resolution image \( I' \) from a given image \( I \) is widely used in many applications, e.g., hierarchical motion estimation \cite{29}. In \cite{8,14}, we describe a spatial-domain watermarking detection scheme, which starts by computing a reduced resolution image. Based on that image, we compute a pseudo-random noise pattern. Then, we correlate the low-resolution image and the image-dependent pseudo-random signal to extract the watermark message.

Our first implementation of the watermark detection module is purely in C. From the VTune profiling of the first C implementation, we found that the most computationally intensive component of the detection system is the subroutine that calculates the reduced-resolution image. The subroutine takes 47% time of the watermark detection scheme, as shown in Figure 4(a). We exclude the MPEG decoding in the CPU time breakdown because our MPEG decoder is already optimized \cite{42}. Although this subroutine takes most of the CPU cycles, the operations of this subroutine are extremely simple as shown in the following:

```c
void get_dc_image_c(void) {
  int y, x, j, k, temp;
  for (y = 0; y < height_in_blocks; y++) {
    for (x = 0; x < width_in_blocks; x++)
```

Floating-point register after we execute an MMX piece of code, or vice versa. To mix MMX and floating-point code fragments in the same application is not recommended. To guarantee that no floating-point errors will occur when we switch from MMX to floating-point, we must use the new MMX instruction EMMS (Empty MMX Technology State), which marks all the floating-point registers as Empty.
The inner loops (loop j and loop k) are simple additions of 8-bit integers. Because of the extreme regularity, the inner loops of this subroutine can be implemented efficiently in MMX. We unrolled the inner loop so that the operation can be executed in parallel. The following is a high-level, conceptual code after unrolling. (We define \texttt{image_data[y, x, j, k]} as \texttt{image_data[(y * block\_size + j) * image\_width + x * block\_size + k]}.)

```c
for (k = 0; k < block\_size; k++)
    temp[k] = 0;

for (j = 0; j < block\_size; j++)
    { /* execute the following additions in parallel */
        temp[0] += image\_data[y, x, j, 0];
        temp[1] += image\_data[y, x, j, 1];
        temp[2] += image\_data[y, x, j, 2];
        ...
        temp[block\_size-1] += image\_data[y, x, j, block\_size-1];
    }

temp = 0;
for (k = 0; k < block\_size; k++)
    temp += temp[k];
```

If we choose the block size to be 8x8 pixels, then in theory we can use MMX instruction, PADDB (_mm_add_pi8), to add 2 sets of 8 pixels together. That is, instead of having 8 independent and sequential additions, we just need one MMX instruction. That is, the inner loop conceptually will be the following:

```c
for (j = 0; j < block\_size; j++)
    { temp[*] = _mm_add_pi8 (temp[*], image\_data[y, x, j, *]);
    }
```
Figure 4: Breakdowns of CPU time in our watermark detection scheme: (a) C implementation and (b) MMX-optimized implementation. We divide our watermark detection scheme into five major parts: (1) compute reduced-resolution images, (2) compute image-dependent pseudo-random noises, (3) extract spatial domain watermark, (4) extract frequency-domain watermark, and (5) function call overheads. In the C implementation, around half of the CPU time is on the module that calculates low-resolution images. After we optimize this module with MMX, this module is no longer the major bottleneck. We achieve 2.3x speed-up in this module, which makes the speed up our whole watermark detection by 1.9x.

Although, we can have 8x speed-up potentially, adding two 8-bit data together requires 9-bit precision. Adding eight 8-bit data together requires 11-bit precision. Instead of packed addition of eight 8-bit data, one practice of adding 8-bit data together is (1) to convert those 8-bit data into 16-bit words and (2) to add the data in the packed-word format. A sample code is shown below:

Input: MM0: unsigned source1 value (8-bit)
MM1: unsigned source2 value (8-bit)
MM7: a zero register (PXOR MM7, MM7)
Output: MM0: four 16-bit words from adding the four LOW end bytes
MM1: four 16-bit words from adding the four HIGH end bytes

MOVQ MM2, MM0
PUNPCKLBW MM0, MM7
PUNPCKHBW MM2, MM7 ; [mm2 mm0] ← mm0
MOVQ MM3, MM1
PUNPCKHBW MM1, MM7
PUNPCKLBW MM3, MM7 ; [mm1 mm3] ← mm1
PADDW MM0, MM3
PADDW MM1, MM2 ; [mm1 mm0] ← mm1 + mm0

After this MMX optimization, the subroutine that calculates the reduced-resolution image takes only 27.5% of the CPU time, as shown in Figure 4(b). Also, after we optimize this module with MMX, the execution time is distributed to other modules more evenly and thus this module is no longer the major bottleneck. We achieve 2.3x speed-up in this module.

2.4. What We Have Learnt
1. Since the MMX can operate multiple operands in parallel, the first rule of MMX optimization is to arrange multiple operands to be execute in parallel.
2. After vectorizing the operations, we can speed up the processing by 2x~8x potentially. In the above example, we find the speed up is around 2.3x.
3. Overhead of packing & unpacking slows down the maximal performance. In the above example, we have eight 8-bit data as input. However, we cannot use 8-bit as the internal representation because of potential arithmetic overflow. Thus, we have to convert those 8-bit data into 16-bit data. A number of unpacking instructions are used for this conversion and slow down the processing.
4. Choosing the right vector size and data arrangement is important. We choose a block size of 8x8 pixels instead of others, e.g., 9x9 pixels. MMX can process eight operands in one instruction. It is very natural to process 8x8 blocks (or 4x4 blocks). It would be much slower if we need to process 3x3 or 9x9 blocks.
5. Sometimes, however, we have to make trade-offs between the speed and the quality of the algorithm. In our watermarking scheme [14], the robustness and picture quality of the watermark depend on the block size. Our algorithm using 8x8-pixel is faster than using 2x2-pixels and 4x4-pixels, but using 8x8-pixels make the watermarks more visible. Similarly, there are many engineering trade-offs depending on the application requirements.
3. BASIC MMX PROGRAMMING—VECTOR AND LOGIC OPERATIONS

In this section, we will show two more principals of MMX optimization by using the MPEG-4 pixel padding procedure as example. The second rule of MMX optimization is to transform conditional executions into logic operations. The first rule of MMX optimization is to execute multiple identical operations in one instructions, as shown in Section 2.3. However, what about the conditional operations like the following:

```c
for (i = 0; i < block_size - 1; i++)
for (j = 0; j < block_size; j++)
{
    if (mask[i+1][j]==0)
        pixel[i+1][j]=pixel[i][j];
}
```

Even after we unroll the inner loop as the following:

```c
for (i = 0; i < block_size - 1; i++)
{
    if (mask[i+1][0]==0)
        pixel[i+1][0]=pixel[i][0];
    if (mask[i+1][1]==0)
        pixel[i+1][1]=pixel[i][1];
    ...
    if (mask[i+1][block_size-1]==0)
        pixel[i+1][block_size-1]=pixel[i][block_size-1];
}
```

Because we don’t know the relationship between `mask[i][j]` and `mask[i][j+1]`, we don’t when those `pixel[i+1][j]=pixel[i][j]` can be execute together. So far, MMX does not allow conditional execution in one instructions. In Section 3.2, we will demonstrate that the conditional operations can be expressed as logic operations so as to be executed in parallel.

The third rule of MMX optimization is to put the data into the right format for parallel execution. The example in Section 3.2 assumes that the data can be load into the MMX register easily in one instruction. What if the data array is in row major and we want to process the data in column major order like the following?

```c
for (i = 0; i < block_size; i++)
for (j = 0; j < block_size - 1; j++)
{
    if (mask[i][j+1]==0)
        pixel[i][j+1]=pixel[i][j];
}
```
MMX can load or store multiple operands if operands are placed in a row. However, multiple operands in a column are harder to access. For faster execution, we should arrange data to be processed in a row-major order or change the algorithm. On the other hand, for some 2D image/video processing operations, an algorithm needs to process data in both directions. In Section 3.4, a matrix-transpose procedure will be demonstrated to provide the flexibility of choosing row-major or column-major processing.

3.1 MPEG-4 Pixel Padding Procedure

In this section, we use the MPEG-4 pixel padding procedure as an example for MMX optimization. The new MPEG-4 video coding standard supports arbitrary shaped video objects in addition to the conventional frame-based functionality in MPEG-1 and MPEG-2 [28]. Thus, in MPEG-4, the video input is no longer considered as a rectangular region. On the other hand, similar to the MPEG-1 and MPEG-2, the MPEG-4 video coding scheme processes the successive images of a VOP (video-object-plane) sequence in a block-based manner (e.g., motion estimation, motion compensation, DTC, IDCT). Therefore, before motion estimation, motion compensation, and DCT, non-object pixels of contour macroblocks (which contain the shape edge of an object) are filled using the padding technique. The padding operation turned out to be a computationally complex and irregular operation [13]. Here, we have created a new MMX procedure to speed up the MPEG-4 padding process by 1.5x to 2.0x.

![Figure 5: Bounding box and macroblocks of an arbitrary shaped video object: (a) Outside the object, (b) inside the object, and (c) on boundary.](image)

First, for each arbitrary shaped object, a minimum sized rectangular bounding box is defined. The box is divided and expanded to an array of macroblocks with natural number of macroblocks in horizontal and vertical direction. Because of the arbitrary shape of the object, not all pixels inside this bounding box contain valid object pixel values. There are macroblocks that lie completely inside the object, macroblocks that lie completely outside, and macroblocks that cover the
border of the video object, as shown in Figure 5. Macroblocks that lie inside the object remain untouched. Macroblocks which cover the object boundary are filled using the repetitive padding algorithm.

Padding is accomplished by copying outwards the pixels that lay on the edge of the mask. First, the pixels are padded in the horizontal direction, with boundary pixels propagated both leftwards and rightwards. In a second pass, pixels are padded in the vertical direction. In both cases, if a pixel that lies outside of the mask is bounded by two masked pixels on opposite sides, the unmasked pixel should be assigned the average of both bounding pixels.

In the horizontal padding step, the macroblock is processed row by row, as shown in Figure 6(a). All non-object pixels inside a row of the macroblock are filled with the next border value of the object. As shown in Figure 7, if there are shape edges on both sides of a non-object pixel, the average of the border values is taken, e.g., Figure 8(b). Afterwards these filled pixel positions are marked as object pixel for the following padding step.

In the vertical padding step, the macroblock is processed column by column, as shown in Figure 6(b). All non-object pixels inside the column are filled with the next border value of the object or with the next pixel value filled in the horizontal-padding step. If there are shape edges or previously filled pixel on both sides of a non-object pixel, the average of these values is taken. If all columns of the macroblock are processed, the padding algorithm is done, e.g., Figure 8(c).

![Figure 6: Repetitive padding: (a) horizontal step and (b) vertical step.](image-url)
For (y=0; y<block_size; y++)
{
    If (mask[y][x] == 1)
        hv_pad[y][x] = hor_pad[y][x];
    Else
    {
        if (mask[y'][x] == 1 && mask[y''][x] == 1)
            hv_pad[y][x] = (hor_pad[y'][x] + hor_pad[y''][x])/2;
        else if (mask[y'][x] == 1)
            hv_pad[y][x] = hor_pad[y'][x];
        else if (mask[y''][x] == 1)
            hv_pad[y][x] = hor_pad[y''][x];
    }
}

Figure 7: The reference pseudo code of the vertical pixel padding algorithm, where y’ is the location of the nearest valid sample (mask[y'][x] == 1) above the current location y at the boundary of hv_pad, y” is the location of the nearest boundary sample below y. The input is a horizontally padded block.

Figure 8: An example of pixel padding. (a) shows the original 4x4 matrix. Pixels labeled with an asterix are outside of the pixel mask. (b) shows the matrix after the horizontal padding stage. Pixels in bold are the changed values. (c) shows the final matrix after the vertical padding stage.

3.2 They are All Logic Operations

So far, MMX does not allow conditional execution in one instruction. In this section, we demonstrate how to use logic operation to perform conditional operations. By assuming that the object is convex in shape, we do not need to average pixel values for padding. As shown in Figure 9, by assuming we are performing vertical padding, we simplified the pixel-padding procedure as the following:

for (i = 0; i < block_size-1; i++)  // vertical direction
for (j = 0; j < block_size; j++)  // horizontal direction
{
    if (mask[i+1][j]==0)
        pixel[i+1][j]=pixel[i][j];
}
Furthermore, we assume that it is always true that \( \text{mask}[i][j] = 0 \) or \( \text{mask}[i][j] = 1 \). In this case,

\[
\text{if (mask}[i+1][j] == 0) \\
\text{pixel}[i+1][j] = \text{pixel}[i][j];
\]

is equivalent to

\[
\text{pixel}[i+1][j] = \begin{cases} 
\text{pixel}[i][j] & \text{if mask}[i+1][j] = 0 \\
\text{pixel}[i+1][j] & \text{if mask}[i+1][j] = 1 
\end{cases}
\]

Then, it is clear that

\[
\text{pixel}[i+1][j] = \text{mask}[i+1][j] \times \text{pixel}[i+1][j] + (1 - \text{mask}[i+1][j]) \times \text{pixel}[i][j];
\]

carries the same functionality as

\[
\text{if (mask}[i+1][j] == 0) \\
\text{pixel}[i+1][j] = \text{pixel}[i][j];
\]

Thus, the above conditional procedure can be expressed as the following:

\[
\text{for (i = 0; i < block\_size - 1; i++)} \\
\text{for (j = 0; j < block\_size; j++)} \\
\{ \\
\text{pixel}[i+1][j] = \text{mask}[i+1][j] \times \text{pixel}[i+1][j] + (1 - \text{mask}[i+1][j]) \times \text{pixel}[i][j];
\}
\]

This can be done easily in MMX because the above algorithm can be executed without any knowledge of the pixel or mask values---there are no branch statements. This algorithm can be sped-up using MMX instructions by computing
all eight pixels in a row concurrently. Figure 10 shows our code for this simplified pixel padding procedure.

```
mov edi, pixelsptr
mov esi, maskptr
mov ecx, 7 ; loop counter
Simple_loop:
movq mm1, [esi+8] ; mm1 <- M2
movq mm0, [edi+8] ; mm0 <- P2
pandn mm1, [edi] ; mm1 <- P1 AND (NOT M2)
; do bitwise not of the mask of
; the current line, and it with
; the pixels of the last line
por mm0, mm1 ; mm0 <- P2 + [P1 AND (NOT M2)]
; and add them together. This
; "smears" the pixels in the
; down direction.
movq [edi+8], mm0
add esi, 8
add edi, 8
loop Simple_loop
```

Figure 10: The MMX procedure for simplified pixel padding.

### 3.3 Data Arrangement Optimization for MMX Technology

Efficient processing of a continuous stream of data requires both a faster CPU and fast memory throughput. Beside algorithmic and code optimization, there is another important issue in MMX optimization---data optimization. Multimedia applications are memory intensive in nature and exert a huge demand on the memory subsystem. While an x86 multiplication takes 10 cycles, MMX with four multiplies takes 1 cycle. Speeding up calculation makes memory access a bottleneck.

Depending on the nature of the data, certain access patterns are more efficient than others [10]. The performance can be improved by changing the data organization (if possible) or by changing the way data is processed. It is important to arrange data so that MMX instructions can perform operations in parallel with less load/store operations.

First, in this previous section, we assume that the pixel values and the mask values are both stored in 8-bit formats. Thus, we can perform 8 pixel-padding operations at each iteration. That is, in theory, we can have 8 times speed-up using MMX operations. If the data are stored in 16-bit values, then we can perform only 4 operations at one iteration (only 4x speed-up in theory).

It is clear that there are trade-offs between precision and speed in some applications. In some applications, precision is more important than the speed. Even for 9-bit values, it is better to store the data in 16-bit format. In this case, we can only process four 16-bit values in MMX operations. On the other hand, in some applications, speed is more critical than the precision. Thus, it may be better to
truncate the 9-bit values and store them into the 8-bit format for more parallelism in MMX operations.

Second, the data array for pixel values and the data array for mask values are stored separately instead of interleaving the pixel data with the mask data. Hence, one load instruction can move 8 pixel data into one register and another load instruction can move 8 mask data into another register. We can easily perform logic operations on them. Otherwise, if the data are interleaved, then we need additional operations to perform the logic operations. Clearly, it is important to design the data arrangement to match the capability of the MMX instructions.

Besides, data alignment is important for efficient data processing. Misaligned data incurs a 3-cycle penalty on quadword boundary or 6-9 cycles on cache-line boundary. When data for MMX operations are aligned with 8-byte boundary, the performance is better.

3.4 Matrix Transpose --- Relief to Column-Major Processing

MMX is very powerful in padding 8 pixels in a row in parallel. Previously, we assume we are performing vertical padding. In this case, multiple data in a row can be processed in parallel (column-wise processing) by reading the data row by row. Nonetheless, we need to pad the pixels horizontally as well. That is, multiple data in a column must be processed in parallel. In row major memory organization, pixel[i][j] and pixel[i][j+1] are adjacent to each other, but pixel[i][j] and pixel[i+1][j] are not. Although RAM stands for random access memory, to load pixel[i][j], pixel[i+1][j], pixel[i+2][j], ... into one MMX register takes huge efforts.

The solution to our difficulties in horizontal padding is to transpose the data matrix. After transposition, vertical padding, which processes multiple pixels in a row, is equivalent to the original horizontal padding. Figure 11 shows the flow chart of our implementation of the pixel-padding procedure; Figure 12 shows an example. First, we transpose the input macroblocks and perform a vertical padding processing. Then, we transpose the intermediate macroblocks back and perform another vertical padding processing. Although a matrix transposition takes significant amount of time, two matrix transpositions and one MMX vertical padding is still faster than a horizontal padding using scalar operations.

Since MMX registers store only one-dimensional array, the question now is how to transpose the matrix efficiently in MMX operations? In Section 2.1, we only briefly explain the unpacking instructions PUNPCKL and PUNPCKH. The unpack instructions shuffle half of the contents of one register with half of the contents of another register (or 64-bit memory location). The PUNPCKH instructions interleave the high-order data elements of the two operands and ignore the low-order data elements. Similarly, PUNPCKL instructions interleave the low-order data elements of both operands. PUNPCKHBW instruction interleaves the four high-order bytes from both operands. PUNPCKHWD interleaves the two high-order words from teach operand, as shown in Figure 13. If the second oper-
and contains all zeros, the result is a zero extension of the high-order elements of the destination operand. In this case, the instructions can convert data into a higher-precision representation.

Beside gathering data from different memory locations, interleaving planar, and duplicating data, the unpacking/packing instructions can transpose rows and columns of data. Figure 14 illustrates a method for performing 4x4 transpositions on 16-bit packed values [4]. Basic idea behind this method is the following. First, we collect the higher-order data into a set of registers. Then, collecting the higher-order data order data from the registers, which contains higher-order data, is equivalent to collecting the highest-order data from each row. That is, we have the data originally in a column now in a MMX register. To transpose 8x8 transpositions on 8-bit packed values is left as an exercise to readers.

Figure 11: Our pixel padding procedure contains four parts: (1) transpose of the block, (2) vertical padding, (3) transpose of the block, and (4) vertical padding.
Figure 12: An example of our pixel padding procedure: (a) original block, (b) transpose of the original block, (c) vertical padding, (d) transpose of the vertically-padded transposed block (which is equivalent to the horizontally-padded block), and (c) vertical padding.

![Diagram of pixel padding procedure](image)

Figure 13: Shuffling the most significant two words from two 64-bit operands into one 64-bit operand with PUNPCKHWD.

![Diagram of shuffling two 64-bit operands](image)

Figure 14: Performing a 4x4 transposition using PUNPCKH and PUNPCKL.

![Diagram of 4x4 transposition](image)

We measure the performance of the MMX optimized implementation. Our simulation results show that the new pixel padding routines to run between 1.5x and 2.0x faster than the original integer-only version.
3.5 Key Points We Learnt

To illustrate our points in MMX optimization, we optimize MPEG-4 pixel padding algorithm using MMX technology in this section. During the optimization, we learnt the following implications:

1. Algorithms should be designed to take advantage of the SIMD operations, i.e., to execute identical operations in parallel. Thus, it is important to avoid conditional operations if applicable; if not, use logic operations.
2. A deterministic algorithm is more preferable than a non-deterministic algorithm (similar to ASIC implementation). In MMX implementation, although logic operations can replace conditional operations, logic operations are overheads.
3. Algorithms should be designed to take advantage of the natural data arrangement (or arrange the data into the MMX processing order). In order to achieve MMX data optimization, we should determine the optimal data packing format for SIMD processing (array of structure, structure of array, or data layout).
4. We observed that in many implementations, too many of the instructions in the code are data format conversions, e.g., the implementations spend too much time packing and unpacking data. The format of input data to be processed by an algorithm may need to be rethought to make it optimal for MMX technology. In some cases the optimal format may be quite strange and not obvious.
5. We should design algorithms in the row-major processing order, i.e., can process multiple data in a row in parallel. While MMX is good in processing row-major data, MMX is less efficient in processing column-major data.
6. If column-major processing is necessary, we can use matrix transposition before process columns of data.
7. Two-dimensional or high-dimensional DSP algorithms are better designed into separable one-dimensional operations. Because the one-dimensional nature of MMX operations, inseparable high-dimensional operations are inherently harder to be processed in MMX.

4. ALGORITHM DESIGN FOR MMX IN SA-DCT AND SA-IDCT

The fourth rule of MMX optimization is to reduce shuffling and maximize pairing. In this section, we demonstrate this rule by optimizing SA-DCT (shape-adaptive discrete-cosine-transform) and SA-IDCT (shape-adaptive inverse-discrete-cosine-transform). 8x8 DCT and 8x8 IDCT have been widely used in image and video compression/decompression. Optimizing 8x8 DCT/IDCT or 8-point DCT/IDCT in single-processor/VLSI implementation has been studied for a
Moreover, the MMX optimization of 8-point DCT/IDCT has also been implemented as well [16,27,34,41].

Most of the fast DCT implementations reduce computational complexity by factoring out the discrete-cosine transformation matrix into butterfly and shuffle matrices. The butterfly and shuffle matrices can be computed with fast integer addition. For example, Chen’s algorithm [7] is one of the most popular algorithms of this kind. Conventionally, optimization usually focuses on reducing the number of DCT arithmetic operations, especially the number of multiplications.

Algorithmic optimization for MMX is different from the traditional optimizations, however. MMX instructions can perform multiple arithmetic operations in one instruction. MMX instructions work best when the operations are regular, and the data elements are adjacent. Furthermore, individual bytes and words within an MMX register are difficult to manipulate. Data shuffling is more expensive than actual arithmetic. In this case, "over-reducing" the number of arithmetic operations may not reduce the total computational time. Instead of re-examining the 8x8 DCT/IDCT, this section uses SA-DCT/IDCT, new tools introduced in MPEG-4, as examples of applying the fourth rules.

### 4.1 Shape-Adaptive DCT and Shape-Adaptive IDCT

One of the building blocks for Version 2 of MPEG-4 Visual Coding Standard is the shape-adaptive-DCT (SA-DCT) for arbitrary shape objects [28,30,36]. The new MPEG-4 video coding standard support arbitrary-shape video objects in addition to the conventional frame-based functionalities in MPEG-1 and MPEG-2. Thus, in an MPEG-4 image, there are some blocks called contour macroblocks, which contain the shape edge of an object, e.g., Figure 5(c). Instead of performing an 8x8 DCT after filling the non-object pixels, the new standard adaptively performs N-point DCT based on the shape. Compared to 8x8-DCT, SA-DCT provides a significantly better rate-distortion trade-off, especially at high bit rates [29].

Similar to standard DCT, forward and inverse SA-DCT convert \( f(x,y) \) to \( F(u,v) \) and vice versa. In contrast to standard 8x8-DCT, only the opaque pixels within the boundary blocks are really transformed and coded, controlled by shape parameters. As a consequence, SA-DCT does not require the padding technique and the number of achieved SA-DCT-coefficients is identical to the number of opaque pixels in the given boundary block.

Figure 15 outlines the concept of the SA-DCT baseline algorithm for coding an arbitrarily shaped image segment which is contained within an 8x8-block. The SA-DCT algorithm is based on predefined orthonormal sets of DCT basis functions. The forward 2D SA-DCT processes columns first, then rows; the inverse 2D SA-DCT applies the one-dimensional transform to rows first.

Figure 15(a) shows an example of an image block segmented into two regions, foreground (gray) and background (light). To perform the vertical transform of the foreground, the length (vector size \( N, 0 < N < 9 \) of each column \( j \) (0 < \( j \) < 9) of
the foreground segment is calculated, and the columns are shifted and aligned to the upper border of the 8x8 reference block, as shown in Figure 15(b).

Dependent on the vector size $N$ of each particular column of the segment, a one-dimensional N-point DCT (a transform kernel containing a set of $N$ basis vectors) is selected for each particular column and applied to the first $N$ pixels of the column. For example, in Figure 15(b), the right most column is transformed using 3-point DCT$^2$.

![Figure 15](image-url)

**Figure 15:** Successive steps involved for performing a SA-DCT forward transform on a block of arbitrary shape.

As shown Figure 15(d), before the horizontal DCT transformation, the rows are shifted to the left border of the 8x8 reference block. Figure 15(e) shows the final location of the resulting DCT coefficients within the 8x8-image block.

In this algorithm, the final number of DCT-coefficients is identical to the number of pixels contained in the image segment. Also, the coefficients are located in comparable positions as in a standard 8x8 block. The DC coefficient (●) is located in the upper left border of the reference block and, dependent on the actual shape of the segment, the remaining coefficients are concentrated around the DC coefficient.

Since the contour of the segment is transmitted to the receiver prior to transmitting the macroblock information the decoder can perform the shape-adapted inverse DCT as the reverse operation in both horizontal and vertical segment direction on basis of decoded shape data.

$^2$ One-dimensional N-point DCT is given by the following equation:

$$y_n = c_n \sum_{k=0}^{N-1} \cos\left(\frac{k(2k+1)}{2N}\pi\right)x_k$$

where $c_0 = \frac{1}{\sqrt{N}}$ and $c_n = \frac{2}{\sqrt{N}}$ for $n = 1, \ldots, N-1$. 

4.2 2-Point DCT

The computational complexity of the 2-point DCT can be simplified as follows:

\[
\begin{bmatrix}
  y_0 \\
  y_1
\end{bmatrix} = \begin{bmatrix}
  \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
  \cos(\frac{\pi}{4}) & \cos(\frac{3\pi}{4})
\end{bmatrix} \begin{bmatrix}
  x_0 \\
  x_1
\end{bmatrix} = \begin{bmatrix}
  \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
  \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}
\end{bmatrix} \begin{bmatrix}
  y_0 \\
  y_1
\end{bmatrix} = \begin{bmatrix}
  1 & 1 \\
  1 & -1
\end{bmatrix} \begin{bmatrix}
  x_0 \\
  x_1
\end{bmatrix}
\]

where \(x_i\) is the input data and \(y_i\) is the transformed data. In conventional algorithmic optimization, we minimize the number of additions and multiplications. Thus, we define

\[
\begin{align*}
  z_0 &= x_0 + x_1 \\
  z_1 &= x_0 - x_1
\end{align*}
\]

And, then

\[
\begin{align*}
  y_0 &= \frac{1}{\sqrt{2}} z_0 \\
  y_1 &= \frac{1}{\sqrt{2}} z_1
\end{align*}
\]

In this way, we need only two additions and two multiplications instead of two additions and four multiplications. The following is the C code for this algorithm.

```c
void fsadct2_float(float in[2], float out[2])
{
    static float f0 = 0.707107;

    out[0] = (in[0] + in[1]) * f0;
    out[1] = (in[0] - in[1]) * f0;
}
```

In MMX and Streaming SIMD Extensions, two additions and four multiplications can be done quickly with only one PMADDWD instruction. That is, out[0] = in[0] * f0 + in[1] * f0 and out[1] = in[0] * f0 + in[1] * (-f0) can be executed in one instruction. Although the following code express as four multiplications instead of two, it is faster than the above in MMX implementation.

```c
void fsadct2_float(float in[2], float out[2])
{
    static float f0 = 0.707107;
```
out[0] = in[0] * f0 + in[1] * f0;
out[1] = in[0] * f0 + in[1] * (-f0);
}

The following is the MMX code for the 2 point DCT.

```c
void fsadct2_mmx (short in[2], short out[2]) {
static __int64 xstatic1 = 0xA57E5A825A825A82;
// -f0 f0 f0 f0
static __int64 rounding = 0x0000400000004000;

__asm {
mov eax, in
mov ecx, out
movd mm0, [eax] // mm0 = xx, xx i1, i0
pshufw mm1, mm0, 01000100b // mm1 = i1, i0, i1, i0
pmaddwd mm1, xstatic1 // mm1 = i0*f0 - i1*f0,
// i0*f0 + i1*f0
paddd mm1, rounding // do proper rounding
psrad mm1, 15
packssdw mm1, mm7 // mm1 = x, x, o1, o0
movd [ecx], mm1
}
}
```

4.3 4-Point DCT

The computational complexity of the 4-point DCT can be simplified as the following:
The total number of operations is reduced from 16 multiplications to 12 multiplications in this case. Wide-referenced Chen's algorithm introduced in 1977 is an algorithm optimized for the number of multiplications [7]. Chen's algorithm is based on further factorizing the inner matrix from

\[
\begin{bmatrix}
\frac{1}{\sqrt{4}} & \frac{1}{\sqrt{4}} & \frac{1}{\sqrt{4}} & \frac{1}{\sqrt{4}} \\
\sqrt{\frac{2}{4}} \cos\left(\frac{\pi}{8}\right) & \sqrt{\frac{2}{4}} \cos\left(\frac{3\pi}{8}\right) & \sqrt{\frac{2}{4}} \cos\left(\frac{5\pi}{8}\right) & \sqrt{\frac{2}{4}} \cos\left(\frac{7\pi}{8}\right) \\
\sqrt{\frac{2}{4}} \cos\left(\frac{2\pi}{8}\right) & \sqrt{\frac{2}{4}} \cos\left(\frac{6\pi}{8}\right) & \sqrt{\frac{2}{4}} \cos\left(\frac{10\pi}{8}\right) & \sqrt{\frac{2}{4}} \cos\left(\frac{14\pi}{8}\right) \\
\sqrt{\frac{2}{4}} \cos\left(\frac{3\pi}{8}\right) & \sqrt{\frac{2}{4}} \cos\left(\frac{9\pi}{8}\right) & \sqrt{\frac{2}{4}} \cos\left(\frac{15\pi}{8}\right) & \sqrt{\frac{2}{4}} \cos\left(\frac{22\pi}{8}\right)
\end{bmatrix}
\]

to

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

The above equation can be expressed as the following:

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\
0 & \frac{\sqrt{2}}{2} & 0 & 0 \\
0 & 0 & \cos\left(\frac{3\pi}{8}\right) & \cos\left(\frac{\pi}{8}\right) \\
0 & 0 & -\cos\left(\frac{\pi}{8}\right) & \cos\left(\frac{3\pi}{8}\right)
\end{bmatrix} \begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3
\end{bmatrix}
\]

The total number of multiplications is reduced from 12 to 10.

Nevertheless, the above extra factorization does not reduce the clock cycle count of the implementation in MMX although it has fewer multiplications. Because the PMADDWD instruction always multiplies and adds, it is impossible to get four discrete 32-bit values from four sets of 16-bit multiplies. Multiplying one set of values consumes the same amount of processor time as multiplying four sets of values. Even if we further factor the lower right cosine block in the original matrix, leaving three total multiplies, it would still need two PMADDWD instructions.
operations, plus a substantial amount of additional instructions to shuffle and add the results. Thus, it is important to reduce shuffling and maximize pairing.

We choose to implement Eq. (1) and have the design like the following:

```c
void fsadct4_mmx (short in[4], short out[4])
{
    static __int64 xstatic1 = 0x4000C00040004000;
        // f0 - f0 f0 f0
    static __int64 xstatic2 = 0xDD5D539F22A3539F;
        // -f1 f2 f1 f2
    static __int64 rounding = 0x00004000000004000;

    __asm {
        mov   eax, in
        mov   ecx, out
        movq   mm0, [eax]     // i3 i2 i1 i0
        pshufw  mm1, mm0, 00011011b  // i0 i1 i2 i3
        movq   mm2, mm1
        paddw  mm2, mm0      // b0  b1  b1  b0
        psubsw  mm0, mm1      // -b3 -b2 b2 b3
        pmaddwd mm2, xstatic1   // o1 << 15, o0 << 15
        pmaddwd mm0, xstatic2   // o3 << 15, o2 << 15
        paddd  mm2, rounding    // proper rounding
        paddd  mm0, rounding    // proper rounding
        psrad  mm2, 15
        psrad  mm0, 15
        packssdw mm2, mm0      // o3 o1 o2 o0
        pshufw  mm3, mm2, 11011000b  // o3 o2 o1 o0
        movq   [ecx], mm3
    }
}
```

The rest of the N-point DCT and IDCT are left as exercises to the readers. Our final implementation boosts the SA-DCT/SA-IDCT process by 1.1 to 1.5 times in MPEG-4 video-object-plane (VOP) based coding scheme. The MMX versions of the N-point DCTs performed from 1.3 to 3.0 times faster than the fixed-point versions, as shown in Table 2.

Table 2: Performance comparison of N-point DCT and N-point IDCT implementations, using floating-point, integer, and MMX instructions.

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Floating-point</td>
<td>Integer</td>
</tr>
<tr>
<td>DCT</td>
<td>2</td>
<td>1260</td>
</tr>
</tbody>
</table>
We also compare the performance of our MMX SA-DCT/SA-IDCT implementation and the performance of an MMX 8x8-DCT/IDCT. SA-DCT is 1.5 times faster than the 8x8 DCT, even with the mask shifting overhead—the 8-point DCT is slower than the (lower N)-point DCTs (assuming that all N-point routines are called with equal probability).

### 4.4 Key Points We Learnt

1. It is important to maximize paired operations. Sometimes, over-reducing the computation in algorithms may give a worse performance.
2. Our MMX-optimized 4-point DCT runs faster than 3-point DCT; our optimized 8-point DCT runs faster than 7-point DCT. Several difficulties arise in the 3-point DCT. The 2-point DCT had groups of operations of two and four, each of which fit nicely into a single MMX operation. The 3-point DCT has a more irregular pattern, which necessitates additional operand shuffling. Although the two PMADDWD instructions are effectively performing eight multiplies and four additions, only six multiplies and three additions are actually needed. In fact, the code size for the 4-point DCT is actually shorter than that for the 3-point transform. For all the N-point DCT and IDCT transforms, the more regular transforms (8, 4, 6, 2) will benefit more than the irregular transforms (3, 5, 7) from MMX optimization.
3. Vector lengths, which are not a multiple of 4 words or 8 bytes, may be problematic for MMX technology. Code can be simplified by padding arrays of data with zero so that vector lengths become multiples of the basic datum.
4. There is significant overhead associated with shuffling the operands, as well as the rounding necessary to minimize inaccuracies due to fixed-point arithmetic.
5. Because using MMX instructions for any problem necessitates additional overhead for packing, unpacking, and shuffling data, the (larger N)-point transforms benefit more from MMX optimization.
6. Mask shifting is much harder in the overall SA-DCT/SA-IDCT process. Currently, we use the single integer instructions to do the shifting instead of using MMX instructions.

5. CONCLUSIONS

Although MMX can provide SIMD operations, there are many restrictions in coding MMX. Designing DSP algorithms for MMX technology is an engineer work in a constrained space. MMX code generation and optimization is a complex and time-consuming process. The process requires an understanding of the algorithms and the processor architecture. Thus, it is an art and a science.

MMX instruction-level optimization steps can be summarized as follows:
1. Unroll loops, reorder/pair instructions, and use logic operations to improve parallelism.
2. Determine minimal suitable precision for the processed values and the corresponding packed data type for SIMD processing.
3. Arrange data in the best way for SIMD processing (row-wise arrangements).
4. Use an optimization tool (such as Intel VTune) to fine-tune the code.

Moreover, in order to accomplish best performance, it is important to have MMX algorithmic-level optimization together with MMX instruction-level optimization. The key of MMX algorithmic-level optimization is to match the algorithms to MMX instruction capabilities.

This chapter covers the basic principles of DSP-MMX co-optimization and gives some examples. For more MMX programming information and examples, readers are encouraged to take a look at *The Complete Guide of MMX* by Bistry et al. [4]. Besides detailed information on MMX instructions & programming, it has sample codes of Viterbi sorting, finite impulse response (FIR) filter, a 3x3 linear separable image filter, the auto correlation step of linear predictive coding (LPC) analysis for speech processing, etc. Additionally, *DirectX®, RDX, RSX, and MMX™ Technology* by Coelho and Hawash [10] also provides some valuable information on Intel processors and performance optimization. It also demonstrates the use of Intel VTune performance optimization tool in detail.

Finally, Intel Corporation offers free implementation examples and optimized libraries to provide developers a jump start to build new applications using MMX technology. Further implementation examples, including fast implementation of DCT/IDCT, can be found from Intel Application Notes [16,25,26,27]. Free and MMX/SSE-optimized Performance Library Suite, which covers a variety of image processing, mathematical, and DSP functions, is available on-line [23].
REFERENCE:


