

# Enhancing Screen Teleconferencing with Streaming SIMD Extensions

John G. Allen and Jesse S. Jin

Biomedical and Multimedia Information Technology (BMIT) Group,  
School of Information Technologies  
Madsen Building F09, University of Sydney, NSW 2006

jallen@cs.usyd.edu.au

## Abstract

It is often necessary to reduce storage and bandwidth requirements when recording or broadcasting a sequence of actions on a computer screen.

These applications most commonly fall into the category of thin client architectures, screen teleconferencing and the recording of demonstration sequences. This paper explores the use of MMX™ and Streaming SIMD Extension (SSE) instructions [11,12] to improve the performance of a hybrid Lempel-Ziv [15] encoding algorithm with temporally seeded history buffer and predictive quarter-static range entropy encoder [9, 14].

We introduce lossless compression via the packed MMX™ comparison instructions as well as a "lossy" technique that makes use of SSE extensions to facilitate partial pattern matching [12].

*Keywords:* screen compression, teleconferencing, video, multimedia, MMX, DirectX.

## 1 Introduction

### 1.1 Background

The dimension and colour depth of the computer screen is often much higher than typical video sequences and thus the compression mechanism must be very efficient to occur in real-time without noticeable impact on the overall system response.

Screen recordings typically have high temporal and spatial redundancy and in general will compress well with a variety of schemes including dictionary, run-length and block data matching techniques. For this application it is not sensible to use full-blown Discrete Cosine Transform (DCT) compression with motion estimation when the average difference between frames is very small, since we may in fact be wasting a large number of cycles to encode a simple shift in mouse position, for example.

Several commercial products exist that aim to provide a solution for desktop screen teleconferencing. Some products demonstrate slow or "jerky" compression at high resolutions and often have a poor screen capture mechanism that is often incapable of capturing video sequences due to dependence on performance settings [1].

Our system implements two distinct components; screen capture and video compression.

The screen capture component is implemented as a DirectShow™ source filter that enumerates as a standard video capture device and exposes a single capture pin, thus enabling any teleconferencing system with support for DirectShow™ to use the screen capture facility.

Our implementation uses hardware bitblt to capture the frame buffer and supports changing screen settings via an internal conversion in representation [4].

The compressor is implemented as a legacy Video for Windows (VFW) compression driver and will work with the standard DirectShow™ wrapper modules to provide compression facilities within a filter graph [8].

### 1.2 Architecture Notes

We implement a DirectShow source filter that enumerates as a video capture device with a single capture pin [5, 6, 7, 8].

This enables the device to be selected by any teleconferencing software that supports standard video capture devices.

We use a highly efficient bitblt technique that is insensitive to video performance settings and changes in screen dimension and can capture real-time video sequences with or without region selection [2, 4].

Our codec is implemented as a legacy Video for Windows (VFW) codec that can be used with existing software that supports VFW codecs such as Windows Media Player [8].

## 2 LZ77 Compression with SIMD Extensions

Intel® introduced a number of extension instructions to their MMX™ technology instruction set that are useful for implementing more efficient video compression codecs.

The packed average (pavg) and packed sum of differences instructions (psadbw) are useful in improving the performance of motion estimation and motion compensation algorithms [11, 12].

We will use the psadbw instruction to provide a mechanism for "lossy" screen compression and the pcmpeqb instruction to match 8-bytes of the LZ77 search and history buffer in parallel.

While this means that we have essentially increased the granularity of the string match length and offset, it also means that we can allow much longer match length and offset buffers in a comparable number of bits, where the

search mechanism is up to eight-times faster and always produces parameters that are reducible by a division of eight.

This method is generally not suitable for compressing fast-motion or high detail video sequences but is found to be sufficient for compressing screen images. In fact, the difference between successive frames is usually so small that the best optimisation we can provide is to locate the changing regions as rapidly as possible and compress them.

The low contrast nature of computer screens also means that good pattern matches will usually be found even with a pattern offset of 8-bytes where typical screens consist of relatively long blocks of unchanging pixels.

## 2.1 SIMD Temporal Seeding

First a single line is fetched from the frame buffer into main memory. We then search for the first and last (8-byte) positions that differ from the preceding frame.

```
int_cdecl Scan(
    BYTE *Input, BYTE *Frame, int Width)
{
    int result=0, change=1;

    _asm
    {
        mov     eax,     Input
        mov     ebx,     Frame
        mov     edi,     Width
        shr     edi,     3
    mainLoop:
        movq    mm0,     [eax]
        movq    mm1,     [ebx]
        pcmpeqb mm0,     mm1
        movq    mm1,     mm0
        psrlq   mm1,     32
        pand   mm0,     mm1
        movd   ecx,     mm0
        cmp    ecx,     0xffffffff
        jnz    doExit
        add    eax,     8
        add    ebx,     8
        dec    edi
        jnz   mainLoop
        dec    change
    doExit:
        mov    result,  edi
    }
    if (change)
        return (Width >> 3) - result;
    else return -1; /* no change */
}
```

**Figure 2.1: Implement fast forward scan for changing blocks using MMX**

If no change is detected we insert the entire scan-line into the history buffer. We then inspect the change flag and send a zero byte to the entropy encoder if it is set; otherwise we increment the zeros flag to indicate unsent zero bytes.

If a change has occurred, we check the change flag. If the change flag is raised we initialise the entropy encoder and send the number of zero bytes indicated by the zeros flag. We insert the first block of unchanged bytes into the LZ history buffer.

For SIMD type values, an 8-bit header containing the change bit and the offset of the change is sent to the entropy encoder. We then encode the number of bytes that have changed in the current scan-line using the LZ77 compression algorithm. The resulting symbols are then sent byte-wise to the entropy encoder with any remaining bytes at the end of the row being inserted into the history buffer.

The change detection mechanism can be improved upon via the SSE packed sum-of-differences instruction to permit faster matches (see Figure 2.2).

This instruction also reduces the core complexity of the pattern matching mechanism by about 3 instructions and allows partial matches to be accepted via a threshold value.

```
int_cdecl Scan(
    BYTE *Input, BYTE *Frame,
    int Width, int Threshold)
{
    int result=0, change=1;

    _asm
    {
        mov     eax,     Input
        mov     ebx,     Frame
        mov     edi,     Width
        shr     edi,     3
    mainLoop:
        movq    mm0,     [eax]
        movq    mm1,     [ebx]
        psadbw  mm0,     mm1
        movd   ecx,     mm0
        cmp    ecx,     Threshold
        jg     doExit1
        jnz    doExit
        add    eax,     8
        add    ebx,     8
        dec    edi
        jnz   mainLoop
        dec    change
    doExit:
        mov    result,  edi
    }
    if (change)
        return (Width >> 3) - result;
    else return -1; /* no change */
}
```

**Figure 2.2: Faster scanning with SSE instructions**

This allows the change detection core to accept partial matches as being unchanged from the temporal frame and push them directly into the LZ history buffer.

This can facilitate much faster compression because on average less data is sent to the LZ and entropy encoders.

An added benefit of this behaviour is that for typical screens, the initial history buffer contains some useful patterns as opposed to being empty in the non-temporal case.

Note that the compression will now be "lossy" unless we supply a threshold value of zero. A zero value allows the new detection routine to act as an efficient replacement for the lossless implementation with identical behaviour.

## 2.2 SIMD Block Pattern Matching

Pattern matching is implemented using the packed sum-of-difference instruction. Improved versions of the block-matching core may use accept partial matches and has also been written to guarantee maximum difference values for individual bytes.

```

BYTE *_cdecl BlockMatch(
    BYTE *pstr, int slen,
    BYTE *pbuf, int buflen,
    int max_match, int *result)
{
    auto int i=0, len, remain, max;
    auto unsigned char *match_ptr=NULL;
    auto int match_len=0;

    do {
        len = 0, remain = buflen - i;

        if (remain >= max_match)
            max = max_match;
        else max = remain;

        if(max > slen) max = slen;

        /* enough bytes left? */
        if(max < match_len) break;

        _asm {
            mov     eax,     pstr
            mov     ebx,     pbuf
            mov     edi,     max
            shr     edi,     3
            strloop:
            movq   mm0,     [eax]
            movq   mm1,     [ebx]
            pcmpeqb mm0,     mm1
            movq   mm1,     mm0
            psrlq  mm1,     32
            pand   mm0,     mm1
            movd   ecx,     mm0
            cmp    ecx,     0xffffffff
            jnz    doExit1 ; not equal.
            add   len,     8
            add   eax,     8
            add   ebx,     8
            dec   edi
            jnz   strloop
        doExit1:
        }
        if(len > 0)
        {
            if(len > match_len)
            {
                match_len = len;
                match_ptr = pbuf;
                if(len == max_match) break;
            }
            pbuf += len;
            i += len;
        }
        else
        {
            pbuf += 8;
            i += 8;
        }
    } while(i < buflen);

    *result = match_len;

    return(match_ptr);
}

```

Figure 2.3: Block pattern matching with MMX

The pattern-matching core can be reduced via the SoD operator. In fact, the operator can also be used to facilitate partial “fuzzy” pattern matches. For instance, a threshold of zero indicates that only exact patterns will match between the search and history buffer.

```

_asm
{
    mov     eax,     pstr
    mov     ebx,     pbuf
    mov     edi,     max
    shr     edi,     3
    strloop:
    movq   mm0,     [eax]
    movq   mm1,     [ebx]
    psadbw mm0,     mm1
    movd   ecx,     mm0
    cmp    ecx,     threshold
    jg     doExit1 ; SoD > threshold
    add   len,     8
    add   eax,     8
    add   ebx,     8
    dec   edi
    jnz   strloop
doExit1:
}

```

Figure 2.4: Partial block matching core with SSE

## 2.3 Formalisation for Error Propagation

If we wish to support partial matches via sum-of-differencing, we must also employ a mechanism to place an upper bound on error propagation.

Let  $X_m^n = X_m \dots X_{m+k} = X^1 \dots X^n$  be a sequence to be encoded such that;

$$\frac{1}{n} \sum_{i=m}^{m+n} d(x_i, \hat{x}_i) \leq D.$$

Our SIMD alphabet  $A$  may be considered a normal alphabet with  $|A| = 256$  and  $|x| = 8$ .

For lossy compression we define fidelity as the distance of the source vector from the reproduced vector  $\hat{X}_m^n$ . We will only consider single letter fidelity accumulated on blocks of size  $|x|$ . For our basic SIMD compression scheme we will have [13];

$$\begin{aligned} d(x^i, \hat{x}^i) &= \frac{1}{n} \sum_{i=m}^{m+n} d(x_i, \hat{x}_i) \\ &= \frac{1}{|x|} \sum_{i=1}^{|x|} |x_i - \hat{x}_i| \end{aligned}$$

Where  $|x|$  is the width in bytes of the SIMD register (usually eight). Furthermore, if we make use of the SSE packed min/max instructions, then we have a selection operator that enables us to guarantee single byte fidelity values using a max-difference constraint, such that;

$$d(x_i, \hat{x}_i) \leq \frac{D}{|x|}$$

The usefulness of this constraint is demonstrated by the reduction of artefacts in regions of sharp derivative (such

as edges) in Appendices III-IV. The implementation is slightly more expensive than the sum-of-differences operator however we can still maintain byte level parallelism with the overhead of only a few instructions.

For temporal video compression, we introduce an error vector of length  $|k|$  that accumulates on some  $x$ -byte block of the video sequence. If we employ key frames at interval  $k$ , then the maximum fidelity at time  $k$  for some particular byte  $B_i$  with any  $d^k(x^i, \hat{x}^i) \leq D$  will be;

$$\begin{aligned} d^k(x^i, \hat{x}^i) &= \sum_{t=1}^k \left| d^t(x^i, \hat{x}^i) \right|_{\max} \\ &= \frac{kD}{|x|} + \frac{iD}{|x|} \\ &= \frac{(k+i)D}{|x|} \end{aligned}$$

That is, the value at the decoder has the potential to become linearly further away from the real value with both the key-frame interval and the block position. This also means that error has an opportunity to propagate towards the end of the document. Notice that even if the average deviation is very small, this error very rapidly degrades our compressed image. This property is demonstrated in Appendix II.

## 2.4 Temporal and Document Error Limits

If we accept a partially matching block and send its length and offset to the decoder, we accumulate some error every that block is selected in the future. If we assume it is equally likely to accept a block that has been subject to replacement, then the amount of error accumulated after  $k$  block matches will be at most  $d^k(x^i, \hat{x}^i)$ , which represents the maximum amount of drift from the real block value at the decoder in the worst case (that is, when accepted blocks are on average favouring either a positive or negative deviation from the actual block value where  $d(\cdot)$  is close to  $D$ ).

Our first heuristic is use a zero threshold for the first  $H$  blocks of the image to ensure that each block in the initial history buffer has no error. To complete this heuristic we never allow a threshold when temporally seeding any part of the frame. Since our LZ compression scheme uses only blocks from the history buffer and not the temporal frame, and since our initial history buffer can be guaranteed to have zero error, we have a simple mechanism to prevent temporal error propagation that does not generally reduce the compression ratio for sequences of screen images. We can limit within document error propagation by modifying the encoder so that it does not remember successive blocks that resulted from a partial match.

This is achieved by replacing blocks in the history buffer when they have been subject to partial replacement. Since we send raw patterns when  $d(\cdot) > D$  for all blocks in the history buffer, we never reproduce any block with error greater than  $D$  because we now select blocks based

on the current value at the decoder rather than the previous frame of the video sequence. Although on average we will be sending a greater number of raw blocks to the decoder, we have placed a simple constraint on the error in the decoded values that is not dependant on the key-frame interval or the number of pattern replacements. In fact, the dictionary nature of the compression method means that frames that differ from the temporal frame will start receiving new blocks and eventually recover as the history buffer contains more relevant patterns.

In terms of implementation we achieved a good result because by default the decoder will only learn a new block if it is sent explicitly. We need only mimic this behaviour at the encoder to ensure we have the same value the decoder has. This is achieved by replacing the block in our temporal frame with the block for which it is matched so that historically the block looks the same as it does to the decoder.

Since we accept partially matching blocks that are inserted into our history buffer, it would seem possible for error to propagate towards the end of the document in a cumulative manner. However, partially matching blocks are always replaced with their approximate matches before being inserted into the history buffer and thus, no error is ever accumulated and the rule for selecting blocks with  $d(\cdot) \leq D$  is sufficient to prevent within document error propagation that exceeds  $D$ . However, since blocks in the temporal frame are replaced with their partially matching equivalents, any visible artefact that is generated will be propagated top-down towards the end of the document in a repeating manner.

## 2.5 Adaptive Threshold Estimation

It is a well-known fact of human perception that an observer is more likely to recognise an undesired distortion on a slowly varying background [4].

To implement variable thresholding we construct an adaptive threshold based on the approximate block pixel derivative.

We ignore contributions to the derivative from next or previous rows. We start with a weighted approximation to the average grey-level error;

$$\begin{aligned} AGLE &= \frac{1}{B^2} \sum_{i=1}^B w_i \circ d(x^i, \hat{x}^i) \quad B = \frac{N}{|x|} \\ w_i &\propto \left( \sum_{n=1}^{|x|} |x_{i+n} - x_{i+n+1}| \right)^{-1} \\ \therefore W_i &\propto w_i^{-1} \approx \sum_{n=1}^{|x|} |x_{i+n} - x_{i+n+1}| \end{aligned}$$

We derive our cost measure with the knowledge that weighting estimates for human perception are usually computed in proportion to the inverse of the pixel derivative.

We would also like to consider the average motion at  $i$  in our computation, so that fast moving sequences can be compressed;

$$m_i(t) = \sum_{n=1}^{|x|} |x_{t,i+n} - x_{t-1,i+n}|$$

$$M_i(t) = \begin{cases} \frac{1}{2}(m_i(t-1) + m_i(t)) & m_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

and,

$$W_i(t) \approx \frac{1}{2^\beta} [D_i(t) + M_i(t)]$$

finally,

$$W_i^*(t) = \begin{cases} W_{\min} & W_i(t) \leq W_{\min} \\ W_{\max} & W_i(t) \geq W_{\max} \\ W_i(t) & \text{otherwise} \end{cases}$$

It is also desirable to clip the result between a low and high value to enforce certain constraints.

Notice that we only consider motion contributions when there is consecutive non-zero motion vectors, as this helps to smooth out spikes from single screen transitions.

The following MMX routine is used to compute a value for the first block derivative of a 16-bit RGB sequence "1111111218111111" addressed by EAX:

	movq	mm1,	[eax]	:	u				
mm1	2	1	1	1	1	1	1	1	1
	movq	mm3,	[eax+8]	:	u				
mm3	1	1	1	1	1	1	1	1	8
	movq	mm2,	mm1	:	v				
mm2	2	1	1	1	1	1	1	1	1
	psrlq	mm2,	16	:	u				
mm2	0	0	2	1	1	1	1	1	1
	psllq	mm3,	48	:	u				
mm3	8	1	0	0	0	0	0	0	0
	por	mm2,	mm3	:	u				
mm2	8	1	2	1	1	1	1	1	1
	psadbw	mm2,	mm1	:	u				
mm2	0	0	0	0	0	0	0	0	7
	movd	eax,	mm2	:	u				
eax									7

Figure 2.5: Fast 16-bit block derivative with SSE

When compared with the C-implementation the SSE version gave us a twofold increase in performance. Notice that the U-V utilisation is rather poor in both this example and the code fragment in Figure 2.6.

We would normally interleave some of the processing into the different pipes to gain better pipe utilisation, although we have shown the simpler implementation for clarity.

The complete MMX code listing to compute our adaptive threshold for 16-bit colour is;

```

BYTE _cdecl Threshold_RGB16(
    BYTE *b0, BYTE *b1, int *motion)
{
    auto int wi, m1, m0 = motion[0];

    _asm
    {
        mov     eax,     b1
        mov     ebx,     b0
        movq    mm1,     [eax]
        movq    mm3,     [eax+8]
        movq    mm2,     mm1
        psrlq   mm2,     16
        psllq   mm3,     48
        por     mm2,     mm3
        psadbw  mm2,     mm1
        movd    wi,      mm2
        movq    mm2,     [ebx]
        psadbw  mm1,     mm2
        movd    m1,      mm1
    }

    if(m0 && m1) wi += (m0 + m1) >> 1;

    wi >>= BETA; /* div. by 2^beta */

    motion[0] = m1; /* return motion */

    /* optimize for cond. move */

    return (wi <= D_MIN) ? D_MIN :
           (wi >= D_MAX) ? D_MAX : wi;
}

```

Figure 2.6: 16-bit threshold estimation with SSE

In order to select parameters for the weighting function, it is useful to measure the average derivative and motion for some typical screen segments.

Region	$D_i$	$M_i$	$W_i$
Black Text on White	92	0	92
Detailed Window	109	0	109
Moving Window	145	34	179
Still Image	171	0	171
Video (Slow Motion)	209	109	318

Figure 2.7: Some adaptive threshold values

## 2.6 Lossless Compression Algorithm

The lossless compression process can be summarised by the following algorithm;

1. Load scan-line from frame buffer into main memory.
2. Find offset of start and end of changed region and send offset to entropy encoder.
3. Insert unchanged blocks at start of row into history buffer.
4. Compress changed blocks (if any) using LZ77 compression algorithm and send symbols to the entropy encoder, inserting into the history buffer as normal.

5. Insert remaining blocks at end of row into the history buffer.
6. If any more rows, move to next row.

## 2.7 Lossless Decompression Algorithm

The lossless decompression process can be summarised by the following algorithm;

1. Receive offset from entropy decoder. If row is unchanged, insert the entire row from the temporal frame into the history buffer.
2. If changed, insert unchanged blocks from the temporal frame into the history buffer.
3. Receive LZ symbols from the entropy decoder and decode them, inserting into the history buffer as normal.
4. Insert the remainder of the temporal row into the history buffer.
5. If any more rows, move to next row.

## 2.8 Lossy Compression Algorithm

The lossy algorithm is identical except that we amend the modifications for partial string matching, block replacement and adaptive threshold estimation.

It was also necessary to implement a maximum threshold comparison using the SSE instructions in which we made use of packed min-max and data manipulation instruction.

By increasing the average pattern length we spend less time wastefully searching the history buffer. Also, this aids in sending less data to the entropy encoder.

## 3 Results

Preliminary results for lossless compression mode show promising compression time and compression ratio.

We have achieved real-time compression of high-resolution screen data (1024x768) at arbitrary bit depths on a 550MHz Pentium III. Naturally the compression ratio for this type of data was very high, with unchanging screens requiring a single byte to encode. A shift in mouse position required about 250 bytes (where the mouse pointer size is about 4KB) giving a compression ratio of about 3000:1.

For screens with a large degree of change (such as key-frames or bringing another application to the foreground) we achieved a compression ratio of between 10:1 and 100:1.

More importantly is the compression speed, which is our primary objective. We dropped zero frames and achieved an average compression time of about 19ms per frame at a resolution of 800x600x16bpp. For unchanged frames the compression time was about 13ms. For frames with a high degree of change this figure was around 45ms.

These figures have satisfied our goal of producing an efficient screen compression codec that can operate at

resolutions that are generally unattainable for conventional codecs.

In comparison we achieved lossless compression with a compression ratio that was four times higher with no noticeable slowdown of the test machine.

Preliminary results for lossy screen compression with fixed threshold value indicate an overall compression ratio of up to 450:1 where only slight imperfections are seen in the compressed screen sequence.

## 4 References

- [1] TECHSMITH (2001): Camtasia Screen Capture SDK, <http://www.techsmith.com>
- [2] MICROSOFT (2001): HOWTO: Obtain A Handle To the Current Cursor, *Microsoft Product Support Services (Q230495)*.
- [3] MICROSOFT (2001): How to Write a Capture Application, *Microsoft Developer Network*.
- [4] MICROSOFT (2001): HOWTO: Capture and Print the Screen, a Form, or any Window, *Microsoft Product Support Services (Q161299)*.
- [5] MICROSOFT (2000-2001): Write a Video Capture Filter: Capture and Preview Pin Requirements, *Microsoft Developer Network*.
- [6] MICROSOFT (2001): Bouncing Ball Source Filter Sample, *Microsoft DirectX 8.0 SDK*.
- [7] MICROSOFT (2000): VidCap Source Filter Sample, *Microsoft DirectShow 6.0 SDK*.
- [8] MICROSOFT (2001): Microsoft DirectX 8.0 SDK Documentation.
- [9] M. SCHINDLER (2000), Adaptive Order 0 Range Encoder with Quasistatic Probability Model, <http://www.compressconsult.com>
- [10] B. RUDIACK-GOULD: HUFYUV Lossless Codec, <http://www.math.berkeley.edu/~benrg/huffyuv.html>
- [11] S. THAKKAR, T. HUFF (1999), The Internet Streaming SIMD Extensions, *Intel Technology Journal Q2, 1999*.
- [12] J. ABEL, K. BALASUBRAMANIAN, M. BARGERON, T. CRAVER AND M. PHILIPOT (1999): Applications Tuning for Streaming SIMD Extensions, *Intel Technology Journal Q2, 1999*.
- [13] M. ATALLAH, Y. GÉNIN, W. SZPANKOWSKI (1996): Pattern Matching Image Compression: Algorithmic and Empirical Results.
- [14] G. N. N. MARTIN (1979): Range encoding, an algorithm for removing redundancy from a digitised message, *Video and Data Recording Conference, Southampton, July 24-27, 1979*.
- [15] J. ZIV AND A. LEMPE (1978): L, Compression of individual sequences via variable rate coding, *IEEE Trans. Inform. Th., IT-24(1978), 530-536*.

## 5 Appendices



**Appendix I**



**Appendix II**



**Appendix III**



**Appendix IV**

Image	Description	Ratio <sup>+</sup>
I	Zero Threshold (Lossless Compression Mode)	0.8:1
II	Fixed Threshold, No Block Replacement at Encoder	2.4:1
III	Adaptive Threshold, Block Replacement at Encoder	2.2:1
IV	Constrained Adaptive Threshold with Fidelity Constraint, Block Replacement	2.1:1

<sup>+</sup> Compression ratio does not include contribution from the entropy encoder. When compressed with a typical LZH style compressor the compression ratio for this image was about 1.38:1. We can achieve a compression ration many times higher than this by adjusting the compression parameters and sending the LZ symbols to the entropy encoder.