

THE VERY FAST CURVELET TRANSFORM

Brian Eriksson

bceriksson@wisc.edu

ECE 734 - VLSI Structures for Digital Signal Processing
Final Project Report

ABSTRACT

The Digital Curvelet Transform provides near-optimal reconstruction of twice-continuously differentiable (C^2) curves. Previous implementations of the algorithm have not exploited newer technologies in modern processors, including the MMX and SSE instruction sets. Various optimization techniques were used on a form of the Digital Curvelet Transform resulting in the fastest Curvelet Transform currently available.

1. INTRODUCTION

Current work in Computational Harmonic Analysis focuses on developing multiscale data representation techniques to represent objects in a sparse manner. The sparser the representation, the fewer the number of coefficients that are needed to be transmitted (allowing for the use of various compression schemes), and the better the denoising that will result from coefficient shrinkage ([2]). While Fourier analysis works well on periodic structures (such as textures), and wavelet analysis works well on singularities (such as corners), neither particularly can reconstruct edges in a sparse matter. Curvelets were originally introduced in [1] as a non-adaptive transform that achieves near optimal m -term approximation rates in L^2 for twice-continuously differentiable curves (C^2). While the performance rates for the Curvelet Transform are quite good, the computational complexity of the algorithm is less promising. Current state-of-the-art techniques allow for a lower bound computation time of between 8 and 10 times the speed of an FFT of the original image. For practical purposes, this computation time must be reduced.

2. CONTINUOUS CURVELET TRANSFORM

The Continuous Curvelet Transform has gone through two major revisions. The first Continuous Curvelet Transform [1] (commonly referred to as the "Curvelet '99" transform now) used a complex series of steps involving the ridgelet analysis of the radon transform of an image. Performance was exceedingly slow.

The algorithm was updated in 2003 in [3]. The use of the Ridgelet Transform was discarded, thus reducing the amount of redundancy in the transform and increasing the speed considerably. In this new method, an approach of curvelets as tight frames is taken. Using tight frames, an individual curvelet has frequency support in a parabolic-wedge area of the frequency domain (as seen in **figure 1**).

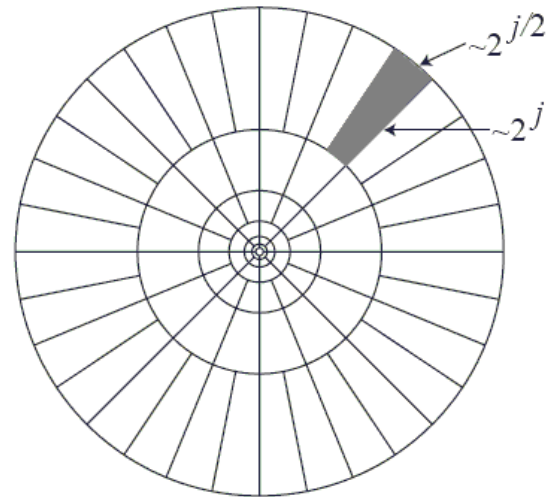


Figure 1. Continuous Curvelet support in the frequency domain

A sequence of curvelets $\{\gamma_{j,l,k}\}$ are tight frames if there exists some value for A such that :

$$A \|f\|_{L^2}^2 = \sum_{j,l,k} |\langle f, \gamma_{j,l,k} \rangle|^2 : \forall f \in L^2 \quad (1)$$

Where each curvelet in the space domain is defined as:

$$\gamma_{j,l,k} = 2^{\frac{2j}{3}} \gamma(D_j R_\theta x - k_\delta) \quad (2)$$

(With D_j = Parabolic Scaling matrix, R_θ = Rotation matrix, k_δ = translation parameter, γ = the "mother" curvelet)

Using the property of tight frames, the inverse of the curvelet transform is easily found as:

$$f = \sum_{j,l,k} \langle f, \gamma_{j,l,k} \rangle \gamma_{j,l,k} \quad (3)$$

In [3], a heuristic argument is made that all curvelets fall into one of three categories.

1. A curvelet whose length-wise support does not intersect a discontinuity. The curvelet coefficient magnitude will be zero. (Figure 2..)
2. A curvelet whose length-wise support intersects with a discontinuity, but not at its critical angle. The curvelet coefficient magnitude will be close to zero. (Figure 3.)
3. A curvelet whose length-wise support intersects with a discontinuity, and is tangent to that discontinuity. The curvelet coefficient magnitude will be much larger than zero. (Figure 4.)

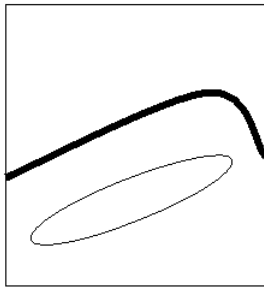


Figure 2. Curvelet Type A

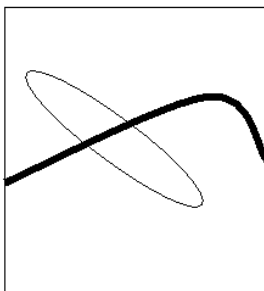


Figure 3. Curvelet Type B

3. WHY CURVELETS?

The fundamental questions should now be, "Why exactly should I use this Curvelet Algorithm?" This can be answered in one

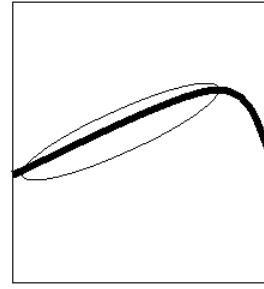


Figure 4. Curvelet Type C

word, **sparsity**. As can be seen in Figures 2-4, when performing the curvelet transform on a C^2 curve, very few curvelet coefficients will be above negligible magnitude values. In [3], it is declared that curvelets offer optimal sparseness for "curve-punctuated smooth" images, where the image is smooth with the exception of discontinuities along C^2 curves. Sparseness is measured by the rate of decay of the m-term approximation (reconstruction of the image using m number of coefficients) of the algorithm. Having a sparse representation, along with offering improved compression possibilities, also allows for improving denoising performance [2] as additional sparseness increases the amount of smooth areas in the image.

In [6] it was shown that orthogonal systems have optimal m-term approximations that decay in L^2 with rate $O(m^{-2})$ (as a lower bound). Currently, there does not exist a single computationally feasible transform that will obtain this lower bound. On images with C^2 boundaries, non-optimal systems have the rates:

Fourier Approximation:

$$\|f - f_m^F\|_{L_2}^2 \asymp O(m^{-\frac{1}{2}}) \quad (4)$$

Wavelet Approximation:

$$\|f - f_m^W\|_{L_2}^2 \asymp O(m^{-1}) \quad (5)$$

Curvelet Approximation:

$$\|f - f_m^C\|_{L_2}^2 \asymp O((\log m)^3 m^{-2}) \quad (6)$$

As seen from the m-term approximations, the Curvelet Transform offers the closest m-term approximation to the lower bound. Therefore, in images with a large number of C^2 curves (i.e. an image with a great number of long edges), it would be advantageous to use the Curvelet Algorithm.

4. DISCRETE CURVELET TRANSFORM - WRAPPING

Using the theoretical basis in [3] (where the continuous curvelet transform is created), two separate digital (or discrete) curvelet transform (DCT) algorithms are introduced in [4]. The first algorithm is the Unequispaced FFT Transform, where the curvelet coefficients are found by irregularly sampling the fourier coefficients of an image. The second algorithm is the the Wrapping transform, using a series of translations and a wrap-around technique. Both algorithms having the same output, but the Wrapping Algorithm gives both a more intuitive algorithm and faster computation time. Because of this, the Unequispaced FFT method will be ignored in this paper with focus solely on the Wrapping DCT method.

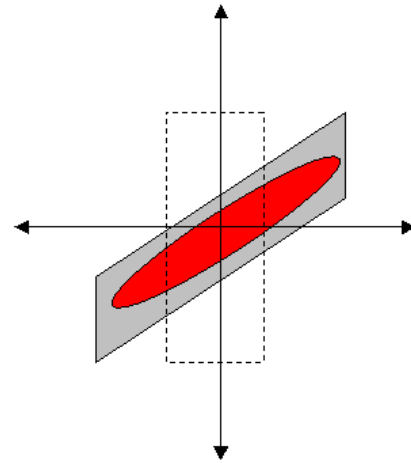


Figure 6. Support of Wedge before Wrapping

Wrapping DCT Algorithm (as described in [4]):

1. Take FFT of the image
2. Divide FFT into collection of Digital Corona Tiles (Figure 5.)
3. For each corona tile
 - (a) Translate the tile to the origin (Figure 6.)
 - (b) Wrap the parallelogram shaped support of the tile around a rectangle centered at the origin (Figure 7.).
 - (c) Take the Inverse FFT of the wrapped support
 - (d) Add the curvelet array to the collection of curvelet coefficients.

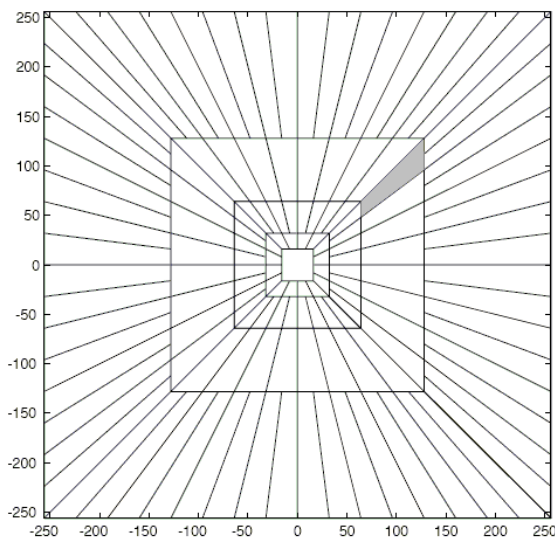


Figure 5. Digital Corona of the Frequency Domain

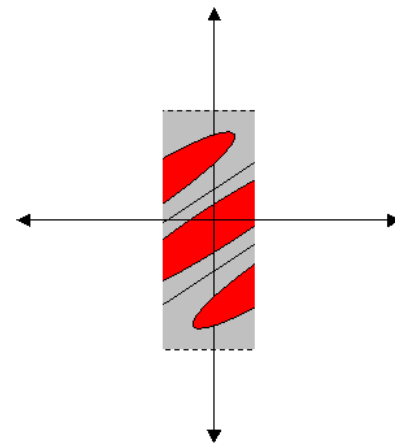


Figure 7. Support of Wedge after Wrapping

Inverse Wrapping DCT Algorithm (as described in [4]):

1. For each curvelet coefficient array
 - (a) Take the FFT of the array.
 - (b) Unwrap the rectangular support to the original orientation shape.
 - (c) Translate to the original position.
 - (d) Store the translated array.
2. Add all the translated curvelet arrays.
3. Take the inverse FFT to reconstruct the image.

5. PREVIOUS ALGORITHMS IMPLEMENTED

The current implementation of the Curvelet Transform is the CurveLab 2.0 software package [5]. The algorithm is implemented in both MATLAB and C code. The C code in CurveLab 2.0 contains no optimization using subword parallelism or any other standard algorithm speedup technique. Both the MATLAB and C code uses the FFTW software ([7]) to compute the Fast Fourier Transform.

6. OPTIMIZATIONS IMPLEMENTED

6.1. Double-Precision to Single-Precision

One of the first changes to the algorithm was the conversion of all floating point variables from double-precision to single-precision. The reason for this was the ability to pack four variables at single-precision into one 128-bit register, instead of only two variables at double-precision. The drawback to this approach was the possibility of quantization noise due to the loss of precision. This was a concern at first, but after several simulations the average mean-squared error (MSE) between double and single precision reconstructions was found to be on the order of 10^{-10} and thus the error was found to be negligible for image reconstruction quality.

6.2. Loop Unrolling

With the iteration of each loop, the iteration constraint must be checked to determine if the loop should continue to be taken. This constraint check adds to the number of computation cycles in the program. The theory of Loop Unrolling (or loop "unfolding") is to reduce the number of times the loop repeats to decrease the overhead of the loop constraint check calculation. Another advantage is that loop unrolling reveals "hidden concurrences" ([9]) in algorithms for compilers to find.

```
for(int j=0; j<N2; j++)
  for(int i=0; i<N1; i++)
    T(i,j) /= sqrtprod;
```

Figure 8. Loop Before Unrolling

Loop unrolling changes loops so that multiple data values are modified with each iteration. This leaves the algorithm prime for implementing packed subword parallel instructions.

```
for(int j=0; j<N2-1; j=j+2){
  for(int i=0; i<N1-1; i=i+2){
    T(i,j)      /= sqrtprod;
    T(i,j+1)    /= sqrtprod;
    T(i+1,j)    /= sqrtprod;
    T(i+1,j+1)  /= sqrtprod;
  }
}
```

Figure 9. Loop After Unrolling

6.3. Subword Parallel Instructions

6.3.1. MMX

Starting with the Pentium II processor, Intel set out to improve naive signal processing performance by adding support for Single-Instruction, Multiple-Data (SIMD) instructions. MMX added eight new 64-bit registers (shared with the existing floating-point registers, see figure 10). The new 64-bit registers offer the ability to pack eight 8-bit bytes types, four 16-bit short types, or two 32-bit int types (See figure 11). These new registers can then perform packed instructions that execute operations on each packed type individually in the register. These integer operations include:

1. Data transfer
2. Conversion
3. Comparison
4. Arithmetic
5. Shift
6. Etc.

List 1. - MMX Packed Integer Operations

The power of packed integer arithmetic can be seen in figure 12. Without MMX instructions, the four operations would take four separate operation instructions. With MMX, all four operations can be performed in one instruction.

What are the drawbacks to this approach? One of the fundamental problems with MMX is that the new 64-bit registers share space with the floating-point registers. After each segment of MMX code, the registers must be cleared before a floating-point operation can be performed. This adds to the overhead of using MMX.

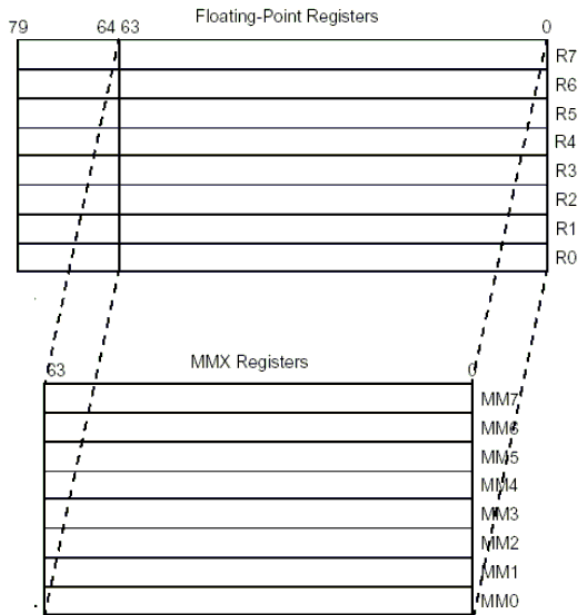


Figure 10. MMX Registers

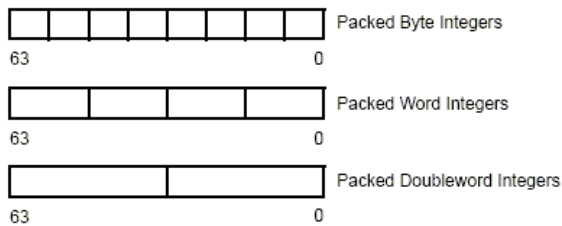


Figure 11. MMX Packed Types

6.3.2. MMX Implementation

For the wrapping function indices calculation in the Curvelet Transform, MMX subword integer instructions were used to optimize the algorithm. Instructions previously performed with slow modulo functions were changed to loop unrolled subword packed comparisons and additions (see **Appendix A** for examples). There are two different ways to implement MMX instructions, using Assembly language or by using MMX intrinsics. MMX intrinsics allow for C/C++ code to use function calls to load/store and perform operations on MMX registers. This allows the programmer to obtain the speed improvements of MMX while making optimization of the code a simpler and faster process to actually implement. In the Very Fast Curvelet Transform, optimization was done using the intrinsic function approach.

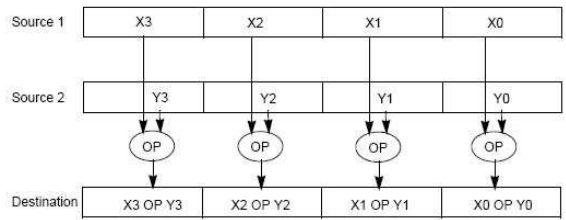


Figure 12. Example of Subword Parallel Operation

6.3.3. SSE

With the release of the Pentium III, Intel added a new SIMD instruction set called SSE (Streaming SIMD Extensions) to the instruction architecture. Like MMX, SSE was designed to improve naive signal processing performance on Intel microprocessors. Unlike MMX, SSE is given dedicated registers to implement packed instructions on. Also, SSE is designed to work on single-precision floating point numbers, not integers.

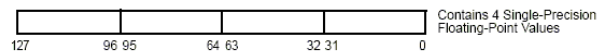


Figure 13. SSE Register Packed Types

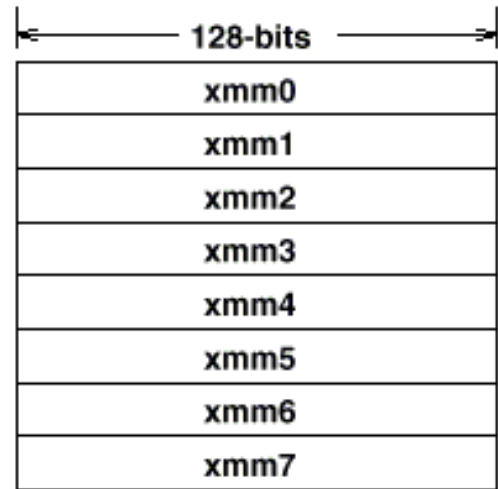


Figure 14. SSE Registers

6.3.4. SSE Implementation

SSE1 instructions were implemented in the Curvelet Transform for floating point subword parallel operations. Two of the main operations of the Discrete Curvelet Transform are the calculation of the filtered wedge of the fourier transform of the image, and the normalization of coefficients.

To filter each wedge out of the fourier transform of the image, the fourier domain is to be filtered using a windowing function. To perform this operation, a for-loop is used iterate over all the coefficients in the array and to multiply each fourier coefficient in the support of the wedge with a windowing filter stored in memory. To optimize this process, the loop was unrolled and the floating point multiplication operation was performed on a packed 128-bit SSE register containing four single-precision floating point fourier coefficients.

One of the main properties of the Curvelet Transform is that it is an isometry, in that the transform satisfies the constraint that:

$$\sum_{t_1, t_2} |f[t_1, t_2]|^2 = \sum_{j, l, k} |c^{D, N}(j, l, k)|^2 \quad (7)$$

(Where $c^{D, N}(j, l, k)$ is the properly normalized curvelet coefficient at scale = j , orientation = l , and location = k)

For the curvelet coefficients to be properly normalized, after each inverse fourier transform of the wrapped wedge array the new array must be normalized by $\frac{1}{\sqrt{L_1 L_2}}$. This requires a loop after each inverse FFT of a wedge section that iterates over all coefficients and divides by the constant values of the area of the wedge. This operation was optimized using loop unrolling and then subword parallel divisions on the coefficients (See **Appendix B** for details).

7. RESULTS

The three algorithms (Previously implemented MATLAB version, previously implemented C version, the new optimized MMX/SSE version of the Fast Curvelet Transform (the "Very Fast Curvelet Transform")) were tested on two separate machines.

	MATLAB	C Version	Optimized C
Forward	3.485	2.947	1.835
Inverse	4.066	2.744	1.366
Forward+Inverse	7.551	5.691	3.201

Table 1. Results of the Algorithms on an Athlon XP 2000+ machine (in seconds)

As can be seen in Table 1 and Table 2, the MMX/SSE optimized version performed significantly better than the previously implemented algorithms. Under an Athlon proces-

	MATLAB	C Version	Optimized C
Forward	2.06	1.46	1.12
Inverse	2.31	1.18	0.74
Forward+Inverse	4.37	2.64	1.86

Table 2. Results of the Algorithms on a Pentium 4 3.4 GHz machine (in seconds)

sor, the new algorithm runs a total (combining both forward and inverse algorithm times) 2.36x and 1.78x faster than the MATLAB version and the original C version (respectively). Under a Pentium 4 processor, the new algorithm runs 2.34x and 1.42x faster (again versus the MATLAB and C code respectively).

8. COSTS

All of these speed improvements have come with a cost. The original C implementation of the Fast Curvelet Transform contained 874 lines of code. The new MMX/SSE optimized implementation has increased the size to 2660 lines of code. Currently this is not thought of as a major concern, as the Fast Curvelet Transform is designed to run naively on PCs where the code size is not a concern. There currently have not been any attempts to implement this algorithm in an embedded system environment, where the size of the code would be large problem. In that situation, a standpoint of trading off speed to execute vs. size of the code would have to be taken.

9. FUTURE DIRECTIONS

Although the Curvelet algorithm performance was improved considerably, there still is room for improvement. The algorithm still uses the older FFTW 2.1 software package. The current version of that software package now takes advantage of SSE2 instructions and hyperthreading. Other possible updates to the software involve optimizing for multi-core processors by using multi-threading and implementing instructions from the SSE2 and SSE3 instruction set architecture.

10. CONCLUSIONS

The results of the Very Fast Curvelet Transform show the power of subword parallel optimization. The combination of loop unrolling and MMX/SSE instruction implementation gave a speed improvement of 2.36x and 1.78x faster than the previous MATLAB and C implementations respectively. This came at a trade-off of much larger code size, but for the current operating environments of the Fast Curvelet Transform, this is not a current concern. Further optimization of the algorithm is possible using the suggestions given in this paper.

11. REFERENCES

- [1] E.J. Candes, D.L. Donoho, "Curvelets - A surprisingly effective nonadaptive representation for objects with edges", *Curve and Surface Fitting*, Vanderbilt Univ. Press 1999.
- [2] D.L. Donoho, "De-noising by soft-thresholding", *IEEE Transactions on Information Theory*, 1995.
- [3] E.J. Candes, D.L. Donoho, "New Tight Frames of Curvelets and Optimal Representations of Objects with Smooth Singularities", *Technical Report, Stanford University*, 2002.
- [4] E.J. Candes, L. Demanet, D.L. Donoho, L. Ying, "Fast Discrete Curvelet Transforms" *Technical Report, Cal Tech*, 2005.
- [5] L. Ying, "CurveLab 2.0" California Institute of Technology, 2005.
- [6] B.S. Kashin, V.N. Temlyakov, "On best m-term approximations and the entropy of sets in the space L_1 " *Mathematical Notes* 56, 1137-1157, 1994.
- [7] M. Frigo, "FFTW Software Package", *Copyright 2006 Matteo Frigo and Massachusetts Institute of Technology*. <http://www.fftw.org>.
- [8] Intel "IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture" 2006.
- [9] K.K. Parhi "VLSI Digital Signal Processing Systems" Wiley Inter-Science, 1999.

Appendix A – MMX Optimization Example

```
float temp1;
for(ycur=yfm; ycur<=yto; ycur++) {
    temp1 = ycur%yn;
    wpdata(tmpx,temp1) = Xhgh(xcur,ycur);
}
```

Figure A-1 – Original Block of Code with no optimizations performed.

```
float temp1, temp2, temp3, temp4;

for(ycur=yfm; ycur<=yto-3; ycur=ycur+4) {

    temp1 = ycur%yn;
    temp2 = (ycur+1)%yn;
    temp3 = (ycur+2)%yn;
    temp4 = (ycur+3)%yn;

    wpdata(tmpx,temp1) = Xhgh(xcur,ycur);
    wpdata(tmpx,temp2) = Xhgh(xcur,ycur+1);
    wpdata(tmpx,temp3) = Xhgh(xcur,ycur+2);
    wpdata(tmpx,temp4) = Xhgh(xcur,ycur+3);

}
```

Figure A-2 – Block of code from Figure A-1 optimized using loop unrolling

```

for(ycur=yfm; ycur<=yto-3; ycur=ycur+4) {

    modvals = _mm_insert_pi16(modvals, (ycur%yn), 3 );
    modvals = _mm_insert_pi16(modvals, (ycur+1)%yn, 2 );
    modvals = _mm_insert_pi16(modvals, (ycur+2)%yn, 1 );
    modvals = _mm_insert_pi16(modvals, (ycur+3)%yn, 0 );

    p3rc2 = _mm_cvtpi16_ps(modvals);
    _mm_empty();
    _mm_store_ps(A,p3rc2);

    p3rc0 = _mm_set_ps1(0.0f);
    p3rc1 = _mm_set_ps(0.0f, 1.0f, 2.0f, 3.0f);
    p3rc5 = _mm_set_ps1((float) yn);

    p3rc3 = _mm_set_ps1((float) (-yh));
    p3rc5 = _mm_set_ps1((float) yn);
    p3rc4 = _mm_add_ps(p3rc5, p3rc3);

    _mm_store_ps(A,p3rc2);

    //<-yh => += yn
    p3rc6 = _mm_cmplt_ps(p3rc2, p3rc3);
    p3rc7 = _mm_and_ps(p3rc6, p3rc5);
    _mm_store_ps(A,p3rc7);

    p3rc2 = _mm_add_ps(p3rc2, p3rc7);

    _mm_store_ps(A,p3rc2);

    //>=-yh+yn => -= yn
    p3rc6 = _mm_cmpge_ps(p3rc2, p3rc4);
    p3rc7 = _mm_and_ps(p3rc6, p3rc5);
    p3rc2 = _mm_sub_ps(p3rc2, p3rc7);
    modvals = _mm_cvtps_pi16( p3rc2 );

    short temp1 = _mm_extract_pi16(modvals , 3);
    short temp2 = _mm_extract_pi16(modvals , 2);
    short temp3 = _mm_extract_pi16(modvals , 1);
    short temp4 = _mm_extract_pi16(modvals , 0);

    wpdata(tmpx,temp1) = Xhgh(xcur,ycur);
    wpdata(tmpx,temp2) = Xhgh(xcur,ycur+1);
    wpdata(tmpx,temp3) = Xhgh(xcur,ycur+2);
    wpdata(tmpx,temp4) = Xhgh(xcur,ycur+3);

}

    _mm_empty();

```

Figure A-3 – Code from Figure A-1 using MMX/Unrolling Optimization

Appendix B – SSE Optimization Example

```
for(int j=-XF2; j<-XF2+XS2; j++)
    for(int i=-XF1; i<-XF1+XS1; i++)
        X(i,j) *= lowpass(i,j);
```

Figure B-1 – Original Block of Code with no optimizations performed.

```
for(int j=-XF2; j<-XF2+XS2-1; j=j+2){
    for(int i=-XF1; i<-XF1+XS1-1; i=i+2){
        X(i,j)      *= lowpass(i,j);
        X(i,j+1)    *= lowpass(i,j+1);
        X(i+1,j)    *= lowpass(i+1,j);
        X(i+1,j+1)  *= lowpass(i+1,j+1);
    }
}
```

Figure B-2 – Block of code from Figure B-1 optimized using loop unrolling

```

for(j=-XF2; j<-XF2+XS2-1; j=j+2){
    for(i=-XF1; i<-XF1+XS1-1; i=i+2){

        pSrc1 = _mm_set_ps(lowpass(i,j), lowpass(i+1,j), lowpass(i,j+1), lowpass(i+1,j+1));
        pSrc2 = _mm_set_ps(X(i,j)._Val[0], X(i+1,j)._Val[0], X(i,j+1)._Val[0], X(i+1,j+1)._Val[0]);
        pSrc3 = _mm_mul_ps(pSrc1, pSrc2);
        _mm_store_ps(A, pSrc3);

        X(i,j)._Val[0] = A[3];
        X(i+1,j)._Val[0] = A[2];
        X(i,j+1)._Val[0] = A[1];
        X(i+1,j+1)._Val[0] = A[0];

        pSrc2 = _mm_set_ps(X(i,j)._Val[1], X(i+1,j)._Val[1], X(i,j+1)._Val[1], X(i+1,j+1)._Val[1]);
        pSrc3 = _mm_mul_ps(pSrc1, pSrc2);
        _mm_store_ps(A, pSrc3);

        X(i,j)._Val[1] = A[3];
        X(i+1,j)._Val[1] = A[2];
        X(i,j+1)._Val[1] = A[1];
        X(i+1,j+1)._Val[1] = A[0];

    }
}

```

Figure B-3 – Code from Figure B-1 using SSE/Unrolling Optimization