

# SIMD Implementation of the Discrete Wavelet Transform

Jake Adriaens

Electrical and Computer Engineering  
University of Wisconsin-Madison  
jtadriaens@wisc.edu

Diana Palsetia

Electrical and Computer Engineering  
University of Wisconsin-Madison  
palsetia@wisc.edu

## I. INTRODUCTION

Digital imaging is used in a wide variety of applications such as digital cameras, cellular technology and medical imaging systems. This has led to research in design of efficient image coding standards. JPEG2000 is the latest image compression standard that has been created to provide higher image compression than JPEG [3]. It is also shown that the compressed images using the JPEG2000 at very high compression rates do not suffer from the same degree of degradation as JPEG images. This is achieved by replacing the original Discrete Cosine Transform (DCT) in JPEG by the superior Discrete Wavelet Transform (DWT). The use of DWT incurs greater computational complexity both in terms of large number of operations and a large amount of memory. The latter is because it replaces the  $8 \times 8$  DCT, which requires only buffering of a single  $8 \times 8$  image tile at a time. For a 2-D DWT, decomposition could require the entire image to be in memory depending on the type of wavelet transform used and the tile size chosen.

Thus, in our project, we attempt to reduce this computational complexity of the DWT algorithm in JPEG2000 by creating a single instruction multiple data (SIMD) implementation. The transforms considered are a Lifting scheme and Daubechies DWT. The Lifting scheme is considered because it performs reasonably well for lossy as well as lossless compression over other filters [1]. We consider Daubechies because it is simple to implement. We provide comparison of the original algorithm implemented in C++ and our hand optimized algorithm using the SSE2 SIMD instruction set.

## II. MOTIVATION

In research many schemes have been studied to reduce computational complexity of the DWT algorithm. One way to reduce computational complexity is by using special purpose hardware. However, this would not be suitable to provide flexibility for choosing between different transformations with a choice of various transformation levels and algorithm parameters of the DWT algorithms. Another method is to hand code the algorithm using assembly language, or loop optimizations. This can achieve some performance gain over the original algorithm, but will be optimized only for memory access operations. The parts of the algorithm that perform numeric computation (multiplication and addition) may still not be

optimized. To achieve reduction in memory access and parallel computation, SIMD extensions such as MMX or SSE can be used along with hand-coded optimizations. In SIMD several small data types are packed into a larger register. Using SIMD instructions allows manipulation and processing of data simultaneously. The authors in [7] provide a speedup of 2.6 for Daubechies-4 DWT (real to real) with SSE extensions over a strict C implementation. Also, an architecture that implements a SIMD instruction set is easily available. For example, current generation processors such as Intel IA-32 and AMD-64 already incorporate SIMD extensions. In our project implementation, we chose to reduce the computation time of DWT algorithm by using hand-code optimization and SIMD implementation.

## III. BACKGROUND ON THE DWT

According to the JPEG2000 standard provided by the International Organization for Standardization (ISO), the JPEG2000 codec contains an encoder and decoder. The encoder codec is shown in figure 1.

The JPEG2000 encoder codec involves preprocessing of the input image in which an image is decomposed, tiled and mapped from RGB color space to YCrCb color space. The tiling is most important feature in this stage as it involves partitioning the input image into rectangular and non-overlapping tiles of equal size. The tiling of images is useful as each tile can be independently compressed as though it was entirely distinct image. Also images can be compressed in rectangular tiles of any size. Next, DWT is applied on each tile in the preprocessed image using 1-D (one dimension) 2-channel filter bank. After the DWT the image is quantized and encoded using bit-plane encoding. The inverse process is applied for the decoder codec [1].

In a DWT transform a signal is decomposed into two sub-bands, usually denoted as lower band and higher band. The lower band contains coarse portions of the signal (i.e. the signal is smoothed), which is performed by scaling function. The higher band contains finer details of the signal and is performed by the wavelet function. This transform can be easily extended to multiple dimensions by using separable filters, i.e. by applying separate 1-D transforms along each dimension. An image is considered a two dimensional signal, so in order to perform the transformation, a 1-D DWT is first

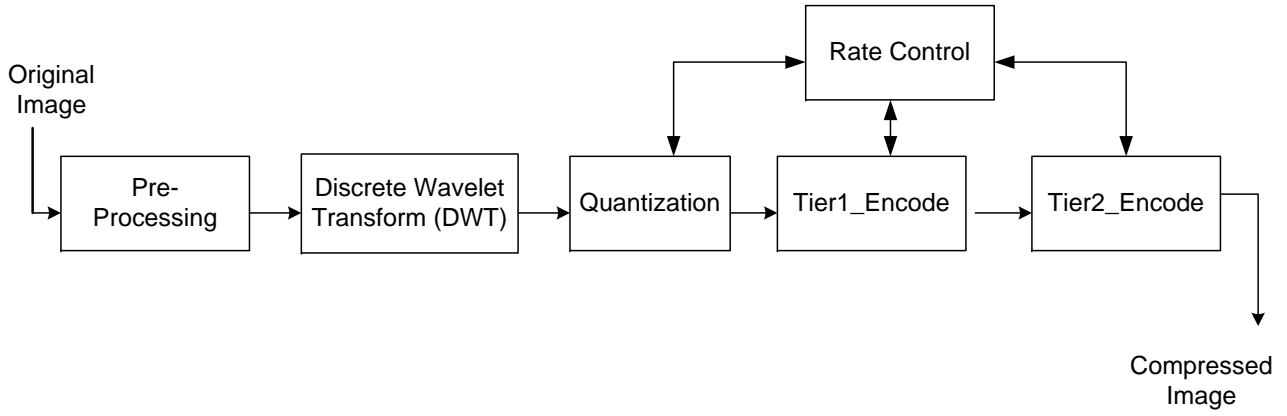


Fig. 1. JPEG2000 Encoder [1].

$$D_i(n) = D_i(n-1) - \sum P_n(k) * S_k(n-1)$$

$$S_i(n) = S_i(n-1) + \sum U_n(k) * D_k(n)$$

$$n \in [1, 2, \dots, N]$$

Fig. 3. Predict and update of the lifting scheme.

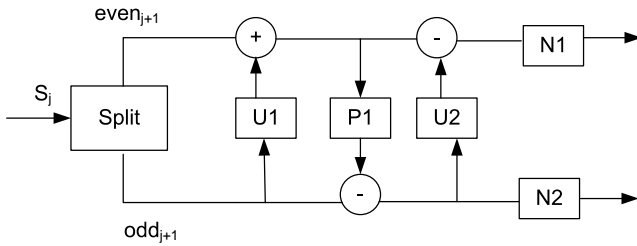


Fig. 4. The lifting scheme [4].

applied to the rows and then to the columns. The DWT we studied in our project are Daubechies 4 and Lifting scheme.

The Daubechies D4 transform uses correlation in data to remove the redundancy. It has four wavelet and scaling function coefficients. Each step of the wavelet transform applies the scaling function to the input data set. If the original data set has  $N$  values, the scaling function will be applied in the wavelet transform step to calculate  $N/2$  smoothed values. In the ordered wavelet transform the smoothed values are stored in the lower half of the  $N$  element input vector. The scaling and wavelet functions are provided in figure 2. In equations  $s$  is the input data and  $h$  and  $g$  are coefficients chosen appropriately for low and high band filtering [4].

Lifting scheme is used for DWT of JPEG-2000 codec. This scheme is a memory efficient scheme compared to other schemes such as plain Daubechies because it provides in-place computation of wavelet coefficients by overwriting the memory locations that contain the input sample values. In

```

//N = length of array
//vec = the array to split
start = 1;
end = N - 1;
while (start < end) {
  for (int i = start; i < end; i = i + 2) {
    tmp = vec[i];
    vec[i] = vec[i+1];
    vec[i+1] = tmp;
  }
  start = start + 1;
  end = end - 1;
}

```

Fig. 6. Original splitting scheme in the lifting DWT.

this scheme, the wavelet transform is divided into several steps. The general strategy is to first compute a trivial wavelet transform by splitting the original 1-D signal into odd and even indexed subsequences and then by modifying these values by alternating between prediction and updating steps. A general equation for the predict step and update step are provided in figure 2.

$D$  denotes the predict step that predicts each odd sample as a linear combination of even samples and subtracting it from odd samples to form a prediction error  $D_i$ .  $S$  denotes the update step that updates the even samples by adding to them a linear combination of already modified odd samples to form the update sequence. The variables  $P$  and  $U$  correspond to predict and update weights (filter coefficients) respectively and  $N$  is the number of times the prediction and update steps are iterated [6]. A general block diagram of the Lifting DWT scheme is shown in figure 4.

#### IV. IMPLEMENTATION

##### A. Code Profiling of the JPEG-2000 Codec

We have profiled a C implementation of the JPEG-2000 codec known as JasPer, in order to identify the most time consuming areas of the code. Table 1 below shows some of the profiled data run using the GNU Profiler [2]:

$$\text{Scaling : } a[i] = h[0] * s[2i] + h[1] * s[2i + 1] + h[2] * s[2i + 2] + h[3] * s[2i + 3]$$

$$\text{Wavelet : } c[i] = g[0] * s[2i] + g[1] * s[2i + 1] + g[2] * s[2i + 2] + g[3] * s[2i + 3]$$

Fig. 2. Scaling and wavelet functions.

% of Total	Cumulative(ms)	Executing(ms)	Total calls	Avg./call	Avg. Time	Function
26.94	11.23	11.23	30	0.37	0.65	jpc_ft_analyze
20.01	19.57	8.34	17391	0.00	0.00	jpc_qmfb1d_split

Fig. 5. Profiled JasPer code using GNU Profiler.

```

//N = length of array
//vec = the array to split
half = N >> 1;
for(i=0; i<half; i++) {
  vec[i] = vec[2*i];
  tmp[i] = vec[2*i+1];
}
for(i=0; i<half; i++) {
  vec[i+half] = tmp[i];
}

```

Fig. 7. Optimized splitting scheme in the lifting DWT.

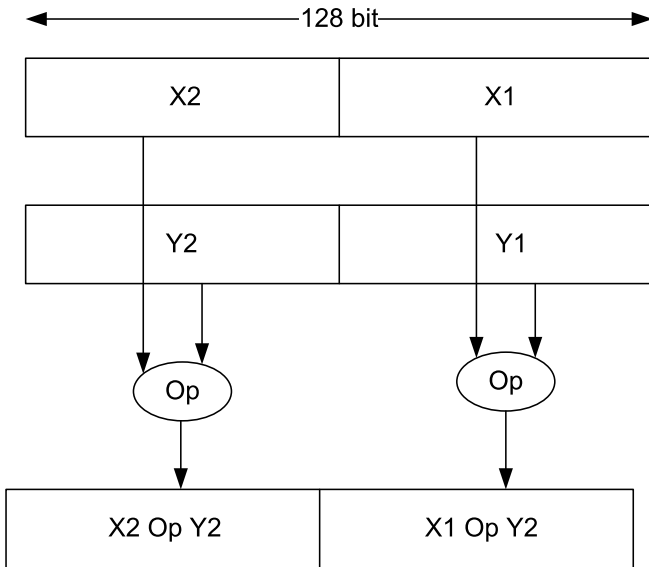


Fig. 8. Double Precision Floating Point SIMD Format.

The table shows the two functions that accumulate the largest amount of execution time in the program. These two functions are both related to the DWT. The split function is part of the lifting scheme and performs the splitting of the input array into lower and upper bands to prepare it for the DWT. The analyze function is a part of the forward DWT. We find that these two functions take up 40% of the total execution time for image processing. This indicates that just optimizing the forward lifting DWT portion could produce significant improvements in encoder run time. In our project,

we have chosen not to optimize the JasPer code directly due to the use of data structures that lend themselves poorly to SIMD operations. Before we could even attempt at using SIMD implementation we would have to inline a large quantity of function calls and redesign the data structures used to represent an image. The latter would lead to modifying almost the entire JasPer suite and hence we have chosen to concentrated on a separate implementation of the DWT, which could be used in JasPer if one where to modify its data structures.

### B. Lifting Scheme Implementation

The original lifting DWT algorithm (lift\_orig.cpp) is written in C++ and based on the 1D transform description provided in [4]. We have taken this implementation and inlined all the functions so we could perform loop transformations in order to prepare the code for conversion to SIMD instructions. After this we noticed the splitting portion of the DWT was implemented as an  $O(n^2)$  operation. This was done in order to reduce the memory requirements of the splitting phase, we re-implemented the splitting phase to be an  $O(n)$  operation. Our  $O(n)$  implementation is not in place and requires 1.5x more memory than the original splitting operation. We feel this is an acceptable increase when being used in the desktop computer environment, as typically the DWT for JPEG-2000 is not performed on large array sizes. Profiling results have shown this change in the splitting algorithm structure alone leads to a small decrease in performance for small array sizes (less than 32) and considerable performance increase for larger array sizes. This is because our new splitting algorithm performs  $3N/2$  move operations, where  $N$  is the size of the array, and the original performs  $3N(N+2)/8$  move operations, figures 6 and 7 depict the original and updated splitting functions.

After the initial preparations to the code we have combined the entire DWT into a doubly nested loop, and then unrolled the inner loop to prepare for performing parallel computations with the SSE2 instructions. When combining the code into the same for loop there where several RAW and WAR dependencies we had to deal with. The attached sources show the DWT after inlining the functions but before combining into the same doubly nested loop. In the first line of the predict section of code odd[0] depends on even[half-1], which is the last piece of data set in the updateOne loop, which makes it impossible to combine these two pieces of code within the

same loop without some transformations. To accomplish this we replace `even[half-1]` with `vec[n-2]+sqrt3*vec[n-1]`, which is how `even[half-1]` is calculated later in the loop. The second line of the update section shows `even[i]` depends on `odd[i+1]` which would not be calculated in time if the update loop was combined with the predict loop, in order to handle this we replace `even[i]` with `even[i-1]` in the loop so it now depends on `odd[i]` which is calculated in time, we also move iteration zero out of the main loop to handle the special case where we would have been calculating `even[-1]` and moved the last iteration outside of the loop as well so we calculate `even[half-1]`.

For the optimized with SSE2 version (`lift_sse.cpp`) we use Intel Streaming SIMD Extensions 2 (SSE2), which extends MMX technology [5]. In JPEG-2000, pixel values can range anywhere from 1 - 48 bits [1]. The pixel values can be integer or float. In our implementation the pixel size is 64-bit double. The reason for taking the 64-bit pixel size is to see how performance of SSE2 scales even for the larger pixel representations. Therefore, we use double-precision floating point SIMD instructions that allow for two floating-point operations to be simultaneously executed in SIMD format (shown in figure 8).

### C. Daubechies Implementation

We have also tested the Daubechies implementation of the DWT and created an SSE2 implementation of it. Figure 12 shows both the original C++ implementation of the Daubechies DWT and our SSE2 implementation. The Daubechies algorithm is considerably simpler to implement and apply SIMD techniques too, than the lifting scheme was. However, it does have the added draw back of requiring twice the memory that the lifting scheme does. We also tried an alternate implementation of the Daubechies scheme where the input is first multiplied with all the coefficients and then added and placed back in the array. This implementation requires five times the memory of the lifting scheme and shows considerable slowdown over the original daubechies implementation.

### D. Simualtion and Debugging

To aid in the debugging process when developing the new DWT implementations we created a program that first creates an array of random elements whose size is specified at runtime. After creating the array a copy is made, and then the DWT is applied to the original array. After performing the DWT, the inverse DWT is applied to the same array. When the inverse DWT is complete the copy of the array is compared against the array that under went the transformations, if they match within some small user defined threshold the runtime of the DWT is printed out. If the arrays do not match the user is warned.

We have also created a program that simulates performing a 2D DWT transform over a large image with variable tile sizes. The program randomly generates a 2048x2048 array of 64 bit pixels. After generating the pixels the program performs the

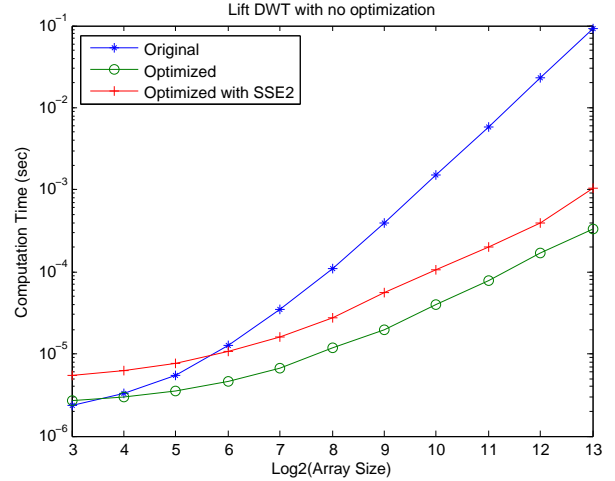


Fig. 9. Lifting DWT run times with no compiler optimizations.

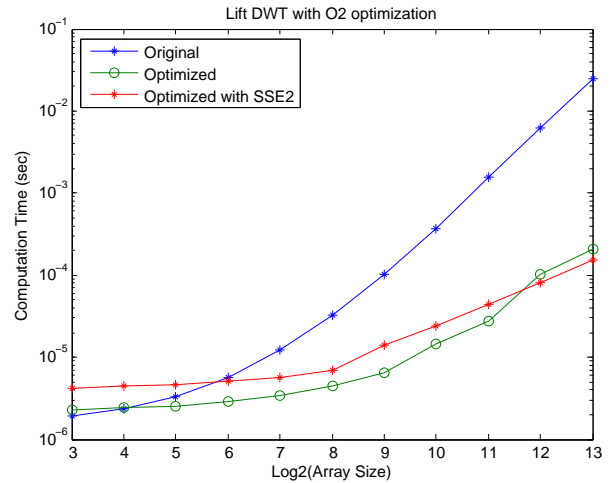


Fig. 10. Lifting DWT run times with -O2 enabled.

2D DWT on each tile, the size of which is specified at run time, by first doing the 1D transform on each row, and then on each column. This program allows us to compare which implementations are biased towards large numbers of small transforms, as you would see in small tile sizes such as 8x8, and which are biased towards low numbers of long transforms, as you would see in tile sizes nearing that of the image itself.

## V. RESULTS

In this section we compare the computation time of the optimized lifting DWT and the optimized with SSE2 implementation to that of the original version. The results are profiled over 1000 runs and performed for array sizes of 8 to 8192. We also provide these results for different compiler optimization flags. The C++ compiler optimization flags used are no optimization, -O2 flag and -O3 flag.

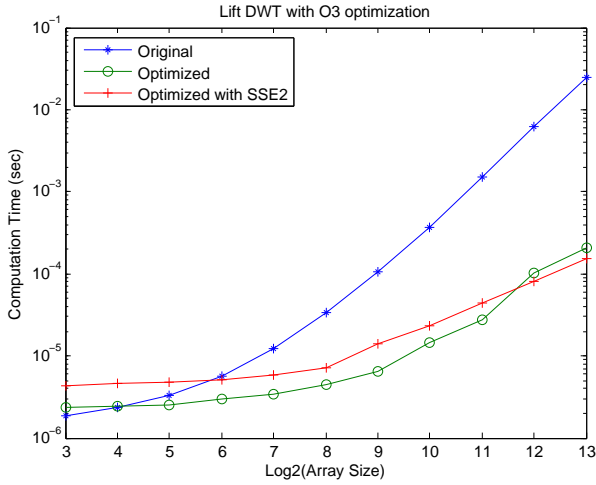


Fig. 11. Lifting DWT run times with -O3 enabled.

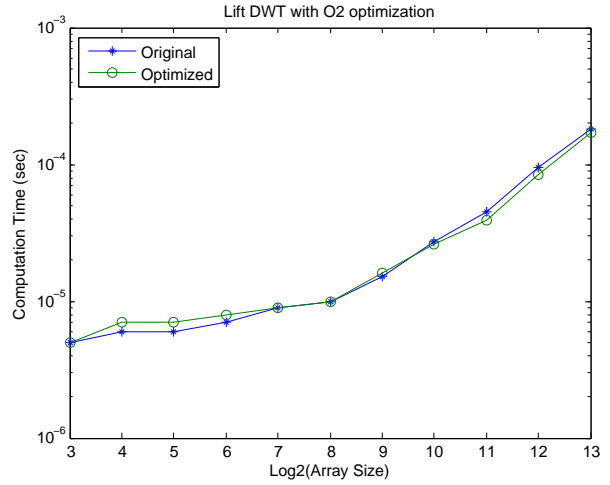


Fig. 13. Daubechies DWT run times with -O2 enabled.

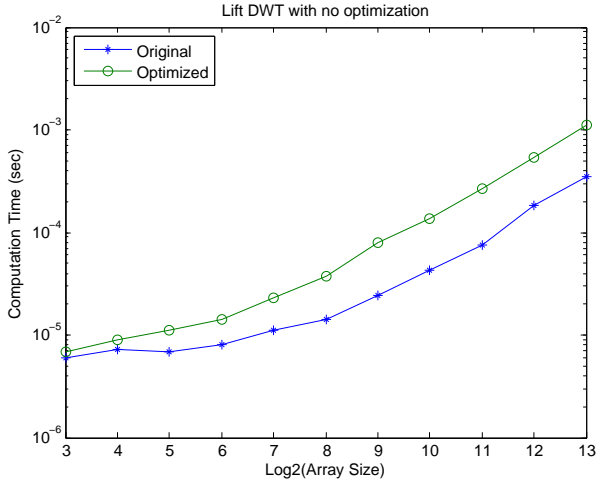


Fig. 12. Daubechies DWT run times with no compiler optimizations.

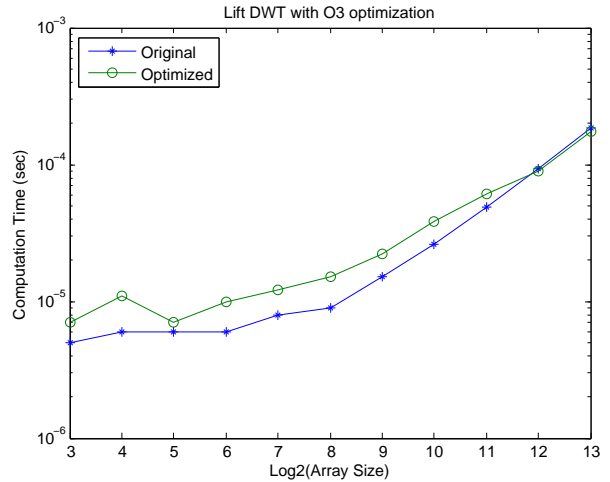


Fig. 14. Daubechies DWT run times with -O3 enabled.

### A. Comparison of Execution Times

The logarithmic plot shown in figures 9-11 show execution time measured in milliseconds for the lifting DWT scheme with the original, optimized and optimized with SSE2 implementations for each of the compiler optimization options. We find that our hand-optimized and optimized with SSE2 outperforms the original lift algorithm when the array size grows beyond 32. This result is consistent for all the three compiler optimization options. However, our optimized with SSE2 implementation does not outperform the optimized version until a very large array size of 212. Also, with no compiler optimization flags, the SSE2 version performs poorly compared to the optimized version for all array sizes. The speed up in the hand-optimized version is due to the use of function inlining and loop optimizations discussed in section 4.2. The speedup of the optimized with SSE2 is also contributed to by loop transformations and sub-word parallel operations. However, we believe that the SSE2 implementations

lack of speedup over the regular optimized implementation is because the compiler is much more architecture aware than the programmer. So when we put in our own hand assembly we really are limiting what instructions the compiler has to work with and what optimizations can be performed. Therefore, unless our implementation is really good, the compiler would have done better without any help from the programmer.

The Daubechies implementations are shown in figures 12-14. We observe there is little difference between the SSE2 implementation and the original. Figures 15-17 show the simulated 2D tiled DWT. These plots show both of our optimized implementations are considerably faster than the original for large tile sizes. It is also interesting to note the crossover point where the original and the optimized versions meet shows up much earlier in the 2D tiled transform. Here it occurs between 4 (4x4) and 8 (8x8) elements, while in the 1D transform it was occurring around 32 elements.

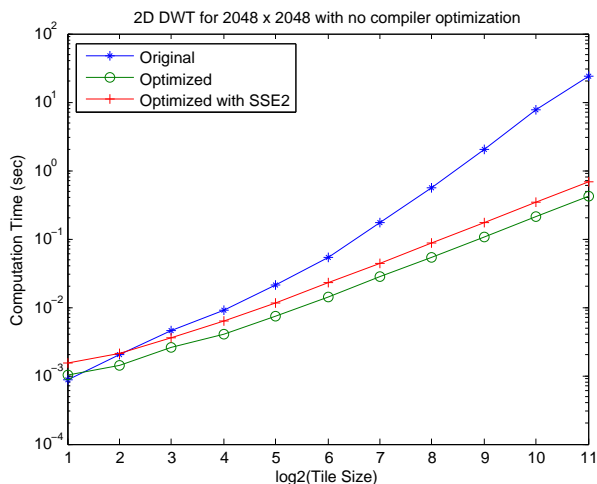


Fig. 15. 2D tiling DWT run times with no compiler optimizations.

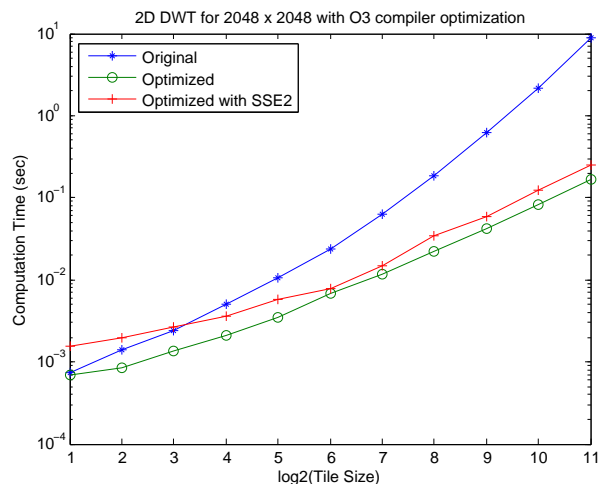


Fig. 17. 2D tiling DWT run times with -O3 enabled.

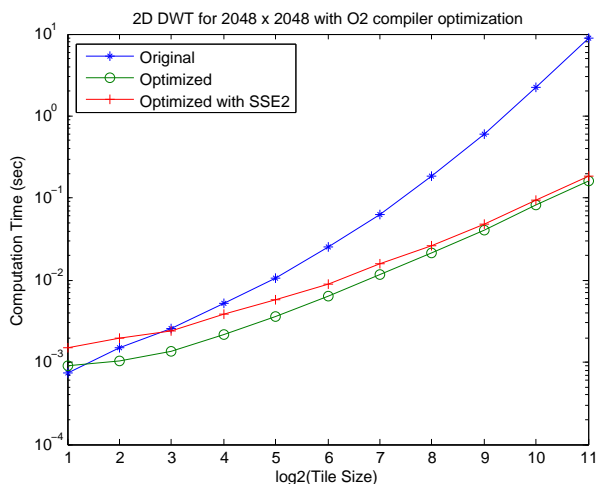


Fig. 16. 2D tiling DWT run times with -O2 enabled.

## VI. FUTURE WORK

The GNU compiler suite supports vector data types that allow for automatic mapping to SIMD instructions if they are available on the platform being compiled for. These vector data types allows for portable vector code to be written that will run on platforms without support for vectors, but still take advantage of any hardware vector support that may be available on other platforms. It is expected the compiler would be able to better optimize the built in vector data type code rather than the strict SSE2 instructions set we have chosen to use here. In the future it may be beneficial to investigate the performance gains presented on multiple platforms when the DWT is implemented with the GNU vector data types.

## VII. CONCLUSION

We have profiled an existing implementation of the JPEG-2000 codec. Our profiling results have show the DWT accounts for approximately 40% of the cumulative execution time when

encoding an image in the Jasper image suite. As a result of these studies we have attempted to create several optimized implementations of the DWT transform, namely a strictly C++ restructuring of the lifting scheme, an SSE2 implementation of the lifting scheme, and an SSE2 implementations of the Daubechies DWT. We have shown through simulation that the two implementations of the lifting scheme scale considerably better than the original with only an 1.5x increase in memory usage. We have also shown that compiler optimizations can often create significant performance increases and may be more important than careful hand coding of algorithms. We have also shown that our lifting scheme implementations scale well when applied to a 2D transform of an image with varying tile sizes.

## REFERENCES

- [1] Michael D. Adams and Faouzi Kossentini. Jasper: A software-based jpeg-2000 codec implementation. In *IEEE International Convergence on Image Processing*, volume 2, September 2000.
- [2] J. Fenlason and R. Stallman. Gnu profiler, April 2006. [http://www.cs.utah.edu/dept/old/texinfo/as/gprof\\_toc.html](http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html).
- [3] ISO/IEC FCD15444-1 ISO/IEC. Information technology - jpeg 2000 image coding system, March 2000. <http://www.jpeg.org>.
- [4] Ian Kaplan. Daubechies d4 wavelet transform, May 2006. <http://www.bearcave.com/software/java/wavelets/daubechies/index.html>.
- [5] Intel Processors. Sse2, May 2006. <http://www.intel.com/support/processors/sb/cs-001650.htm>.
- [6] M. Rabbani and R. Joshi. n overview of the jpeg 2000 still image compression standard. In *Signal Processing: Image Communication*, volume 17, pages 3–48. Elsevier, 2002.
- [7] Asadollah Shahbahrami, Ben Juurlink, and Stamatis Vassiliadis. Performance comparison of simd implementations of the discrete wavelet transform. In *ASAP '05: Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, pages 393–398, Washington, DC, USA, 2005. IEEE Computer Society.