A Single-Loop Approach to SIMD Parallelization of 2-D Wavelet Lifting

Rade Kutil
University of Salzburg
Department of Scientific Computing
Jakob Haringer Str. 2, 5020 Salzburg, Austria
rkutil@cosy.sbg.ac.at

Abstract

Widespread use of wavelet transforms as in JPEG2000 demands efficient implementations on general purpose computers as well as dedicated hardware. The increasing availability of SIMD technologies is a great challenge since efficient SIMD parallelizations are not trivial. This work presents a parallelized 2-D wavelet transform following a single-loop approach, i.e. a loop fusion of the lifting steps of horizontal filtering, and interleaving horizontal and vertical filtering for optimal temporal locality. In this way, each input value is read only once and each output value is written once without subsequent updates. Such an approach turns out to be a necessary basis for an efficient SIMD parallelization. Results are obtained on a general purpose processor with a 4-fold single-precision SIMD extension. Speedups of about 3.7 due to the use of SIMD, 2.55 due to the single-loop approach and up to 6 due to cache effects for pathologic data sizes are obtained, giving total speedups of up to 56.

1. Introduction

Today's most efficient wavelet transforms can be realized by the lifting scheme [6, 17, 18], a factorization of the involved filter pair. The most prominent filter of this kind is the biorthogonal 7/9 filter, which is used in the JPEG2000 standard [9]. For this reason, it is used in this work as an example for the presented parallelization approach.

The lifting scheme is a sequence of prediction and update passes of the form $x_{2n} \leftarrow x_{2n} + \alpha(x_{2n-1} + x_{2n+1})$ or $x_{2n+1} \leftarrow x_{2n+1} + \alpha(x_{2n} + x_{2n+2})$ where α is a predefined factor which is different for each pass. These factors determine the underlying wavelet completely. In the end, even samples x_{2n} represent low-pass filtered coefficients and odd samples x_{2n+1} represent high-pass coefficients. Often, these interleaved subbands are rearranged into separate low- and high-pass subbands. In this work the separation is done implicitly without an extra rearrangement step.

In the 2-D case, each line is filtered by this scheme followed by columns being processed in the same way, giving four subbands denoted by LL, LH, HL, HH. The LL subband is transformed further in a recursive way in the so-called pyramidal transform. We will only investigate a single filtering step in this work. However, the presented methods for subband filtering can be applied without modification at each transform level as a consequence of subband separation. Otherwise, methods for the transform as a whole would have to be developed [13].

Despite the speed of the algorithm (linear complexity) it is still demandable to investigate speedup techniques, since many applications have to satisfy realtime constraints and processed data sets are becoming larger. A significant amount of work has been done for MIMD parallelization [12, 20, 8] and old SIMD arrays [15, 1, 7]. The use of SIMD extensions of modern general purpose processors for wavelet transforms is investigated in [19, 2] together with extensive cache optimizations.

The SIMD parallelization of the vertical filtering is usually the easier task if samples are arranged in row major order. One simply has to perform the sequential algorithm while operating on packed words of several horizontally neighbouring samples, thus filtering several columns at once [19, 2]. The horizontal filtering is not so straight forward to parallelize because it involves an amount of shuffle operations [10] which have to be selected carefully so as to minimize the total number of SIMD operations. This work basically presents a successful new SIMD parallelization of the horizontal filtering and evaluates its performance in conjunction with existing techniques for SIMD parallelization and cache issues of the 2-D filtering.

The lifting scheme can be implemented in a single-loop manner by a loop fusion of the lifting steps, which is a well known state-of-the-art compiler technique. In this way, each input value is read from memory only once and each out-

put value is written to memory once without subsequent updates. While this is an improvement in itself, since it minimizes memory access, it turns out to be the only reasonable way to go for the SIMD parallelization. To see why, let us examine the number of operations in a single lifting pass $x_{2n} \leftarrow x_{2n} + \alpha(x_{2n-1} + x_{2n+1})$. There are 2 adds and 1 multiply for every second sample, which makes 1 add and $\frac{1}{2}$ multiply per sample. Now, assume that SIMD operates on packed words of 4 samples. We can implement this operation by

$$x_{(2n,\dots,2n+3)} = x_{(2n,\dots,2n+3)} + (\alpha, 0, \alpha, 0) \cdot (x_{(2n-1,\dots,2n+2)} + x_{(2n+1,\dots,2n+4)}) ,$$

where $x_{(a,\dots,a+3)}$ is the packed word consisting of the four samples x_a, x_{a+1}, x_{a+2}, x_{a+3}, and $+$ and $\cdot$ are (pointwise) packed add and multiply operations. Since $x_{(2n-1,\dots,2n+2)}$ and $x_{(2n+1,\dots,2n+4)}$ require shuffle operations, we need 2 shuffles, 2 adds and 1 multiply for every 4 samples, giving $\frac{1}{2}$ shuffle, $\frac{1}{2}$ add and $\frac{1}{4}$ multiply per sample or – taken together – 1.25 operations instead of 1.5 in the non-SIMD case. This is, obviously, not a satisfying speedup, given the theoretical maximum speedup of 4.

Therefore, this work concentrates on combining all passes into a single one leading to the single-loop implementation of the 1-D case. Performing all passes at once gives more freedom to the SIMD parallelization since more operations have to be performed consecutively which can be rearranged to a certain degree. By exploiting these degrees of freedom, the SIMD parallelization can be optimized.

If the single-loop approach is applied horizontally and vertically, whether with or without using SIMD, a double-loop algorithm results, with one loop for horizontal and one for vertical filtering. These two loops can be incorporated into a single loop if each value produced by an iteration of the horizontal part is immediately fed into an iteration of the vertical part, so the two iterations can be merged (or interleaved). This technique is also called pipeline computation [2, 3]. The resulting algorithm turns out to be optimal in terms of memory access and cache usage. It can be combined successfully with SIMD parallelization.

All results in this work have been conducted on an Intel Pentium 4 CPU with 2.80GHz and 1MB cache size using the SSE extension with packed words of 4 single precision numbers. All implementations use the same amount of code optimization, i.e. memory access through incremented pointers instead of indexed arrays, and compilation with gcc 3.3.5 with the -O3 option. SIMD operations are implemented using built-in functions for vector extensions and the -msse option. Note that in order to have full control over generated code, no automatic vectorization is applied.

2. Cache Issues

The first step in the optimization of the wavelet lifting algorithm has to be optimization of cache usage. There is a major problem for higher dimensional transforms on almost all common platforms. When accessing data in the second (or higher) dimension, steps (i.e. distances between vertically neighbouring samples in terms of memory addresses) that are a power of two are frequently observed. Caches are, on the other hand, divided into a number of cache sets. A certain amount of low-order bits of memory addresses are used to assign addresses to a cache set. In almost all cases, the number of these cache sets is also a power of two. As a consequence, whole columns are often assigned to a single cache set. This leads to excessive cache misses and dramatically reduced performance.

This issue has been addressed in [14, 11]. An easy method to overcome this problem is to insert gaps between rows of data (array padding). Contrary to what is said in [14], this does not mean that the data size has to be extended; the data in the gaps should simply not be accessed instead. The stride between consecutive rows (i.e. row width plus gap) should be chosen so that a sequence of memory accesses in a single column should reuse cache sets as late as possible. This is the case if the stride and the number of cache sets are relatively prime. [14] suggests that the stride be the nearest prime number greater than the row width. However, since the number of cache sets is a power of two, it is sufficient that the stride be odd, which guarantees relative primeness.

More sophisticated cache optimizations have been developed in [19, 2] by altering the memory layout of the data arrays. In this work a memory layout with separated subbands is chosen, which is denoted "mallat" in [19]. It has the advantage that subsequent filtering steps of the full transform can be applied analogously to the low-pass subband. Moreover, subband-based applications such as JPEG2000 prefer separated subbands. Other layouts, such as fully interleaved subbands [13], are more cache optimal and do not need extra memory due to in-place transformation and are suitable for coefficient-tree based applications [16]. These memory layouts will not be considered in this work, because the presented algorithms can themselves be viewed as an alternative way of memory access optimization.

Figure 1 shows a comparison of the standard (multi-loop) implementation with and without array padding. Note that execution time is divided by the number of pixels of the transformed image, so horizontal lines would indicate (the expected) linear complexity of the algorithm. One can see that starting at an image size of about 200,000 pixels the performance depends strongly on the image size. If the image size is a power of two or a sum of large powers of two, execution times tend to be 6 times that of the array padded

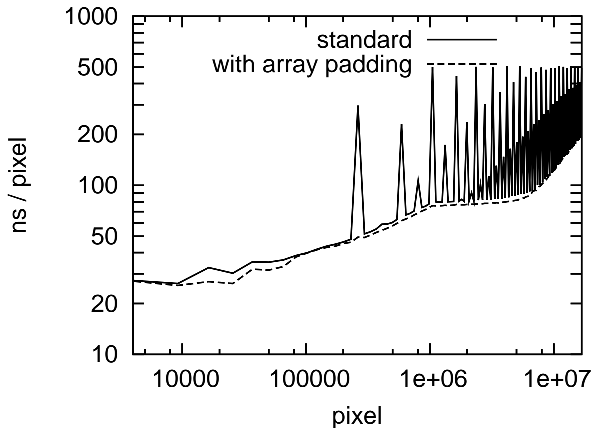

Figure 1. Cache misses for the vertical transform can be reduced by array padding, i.e. by inserting gaps between rows of data so that the stride between vertically neighbouring sample points is relatively prime to the number of cache sets. Execution time in nanoseconds per pixel is shown.

version.

In [14] another approach called aggregation is suggested. This approach is somehow similar to the single-loop approach presented in this work in that it processes several columns at once. If this idea is extended to the full width, one could incorporate the horizontal and the vertical filtering into a single loop, although this is not done in [14]. However, this shows that we can expect that the approaches presented in this work will be able to resolve the cache miss problem even without array padding.

3. Loop Fusion

To see how loops of the wavelet lifting scheme can be incorporated into a single one, see Figure 2. It depicts the data flow of the lifting algorithm for a 10 sample data set. There are four factors $\alpha, \beta, \gamma, \delta$ used in four passes as described in the introduction, followed by a scaling pass with factors $\frac{1}{e}$ for even and $-e$ for odd samples.

Now we apply a loop fusion technique by splitting the whole data flow graph into areas of width two as shown in Figure 2. The computations in each area can be executed from top to bottom if intermediate results from the preceding pass are accessible. These intermediate values are indicated by arrows that cross the borders of areas. To make these values accessible, they have to be passed from iteration to iteration. The areas (iterations) are executed from left to right. In each main iteration, two values are read from memory and two values are output, one low-pass and one

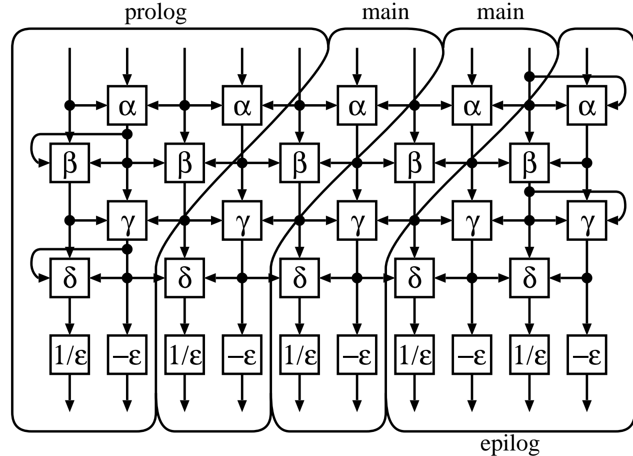

Figure 2. The lifting scheme. Boxes with three inputs (l left, r right, u upper) denote basic lifting operations $y = u + \alpha(l + r)$, boxes with one input denote simple multiplication. Loop iterations in a 1-D single-loop implementation are indicated by rounded frames. Arrows crossing the borders indicate values that have to be passed from iteration to iteration.

high-pass coefficient. The leftmost and the rightmost area labeled prolog and epilog in the figure (first and last iteration) differ from the main iterations and have to be implemented separately. They are responsible for correct border handling (with uneven mirroring).

This procedure can also be described in terms of signal processing, which is shown in Figure 3. Blocks with three inputs, labelled with $\alpha, \beta, \gamma, \delta$, have the same meaning as in Figure 2, i.e. they calculate $y = u + \alpha(l + r)$, where

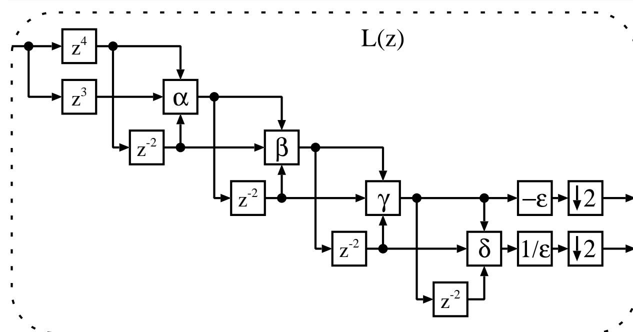

Figure 3. Signal processing model for lifting. This leads to the same single-loop procedure as shown in Figure 2.

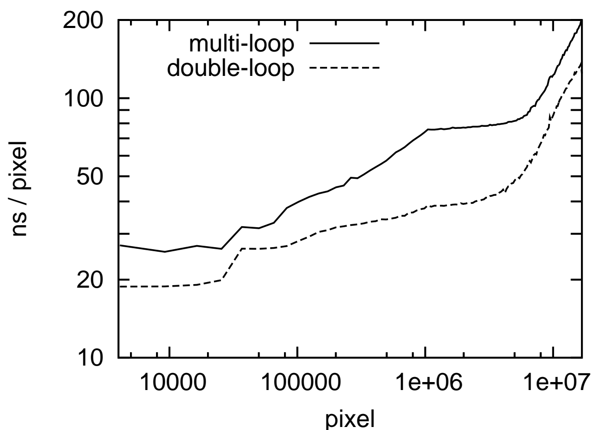

Figure 4. Execution times per pixel for loop-reduced 1-D lifting. Whereas multi-loop denotes the standard implementation, double-loop indicates that there is a single loop for each dimension.

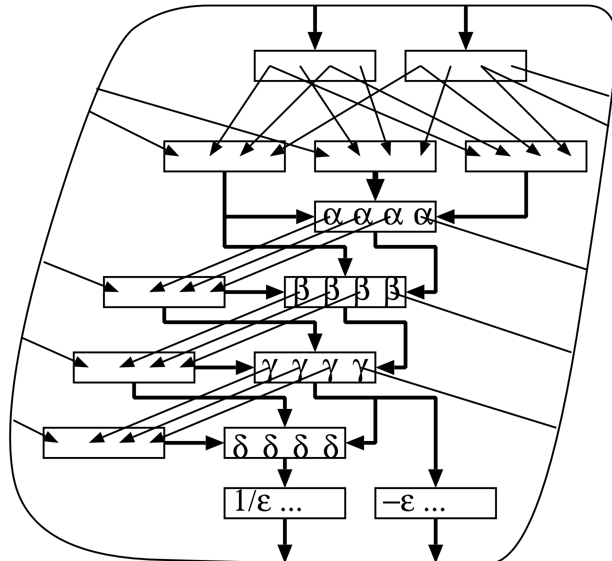

Figure 5. Lifting with SIMD operations. The same idea of splitting the algorithm into areas which can be executed by passing values from iteration to iteration applies here as well. A single iteration is shown. Boxes represent vectors (packed words) except for those labeled with $\alpha \ldots \epsilon$ which represent basic lifting operations (as in Figure 2). Thin arrows indicate shuffle operations. Arrows that leave the iteration to the right and come in to the left indicate the passing of values between iterations.

in this case u is the horizontal input and l, r are the vertical inputs. The passing of values between iterations is done by the blocks labelled z^{-2} which delay intermediate values by 2 samples, i.e. by the duration of one iteration.

It is this block $L(z)$ that is the basic building block for all following algorithms in this work. The obvious advantage is reduced memory access since intermediate values that have to be passed from iteration to iteration can be kept in CPU registers. This also reduces cache problems as a further side effect.

Although we now have a single loop for 1-D lifting, we need another loop for the second dimension in the 2-D transform. Therefore, we call this algorithm the *double-loop* algorithm. Figure 4 shows the execution times of the double-loop implementation compared to the standard implementation, which we will call *multi-loop* algorithm from now on. Both apply array padding so we can see the performance gain independent of simple cache problems. A reduction of execution time by a factor of up to 2.2 can be reached. Note also that the time per pixel increases with the number of pixels. This means that the total execution time does not grow linearly with the image size. The reason for this is that the rate of remaining cache misses grows with the image size.

4. SIMD Parallelization

4.1. Pure Horizontal Parallelization

After having introduced the single-loop approach for the 1-D transform we shall now come to the main achievement in this paper, i.e. an efficient SIMD parallelization of the horizontal filtering. Note that we assume a row-major memory layout, in which case the horizontal filtering is the more difficult part. Figure 5 contains a schematic representation of the operations involved. The main idea from the sequential single-loop approach applies here as well. The algorithm is split into iterations which can be executed by passing values from one iteration to the next. One such iteration is depicted in Figure 5. The difference is that now two whole packed words are read from memory per iteration and two packed words are output, one containing four low-pass coefficients and one containing four high-pass coefficients.

There is a number of shuffle operations involved, indicated by thin arrows in Figure 5. Depending on the instruction set of the SIMD processor, these shuffle operations can often not be implemented directly (they cannot in Intel SSE). Therefore, some intermediate words have to be produced. If these are chosen wisely, they can be reused in other shuffle operations. This is addressed in more detail in [10]. In the current implementation 11 shuffles are neces-

sary for every two input words. Again there are prolog and epilog iterations at the beginning and at the end of the loop which have to care for border handling.

4.2. Vertical Parallelization

As stated before, the vertical filtering can be implemented more easily by adopting the sequential algorithm to act on four columns at the same time. Instead of a single sample of a single column, four horizontally neighbouring samples are read from memory into a packed word. It is straight forward to implement this in the single-loop style introduced in Section 3. Together with the horizontal SIMD filtering we get a SIMD parallelized double-loop algorithm.

4.3. Verticalized Horizontal Parallelization by Local Transposition

The easier vertical approach to SIMD parallelization can also be applied in the horizontal direction by performing a transposition of data before and after the filtering. This, of course, means extra computations and memory accesses. It saves ourselves the shuffle operations of the pure horizontal approach, though. In [2] this is done on blocks of data to increase locality.

As this is not consistent with the intended single-loop approach, we choose to perform the transposition "on the fly", i.e. we read four packed words, transpose this 4×4-block and perform the 1-D single-loop algorithm on it. As the result has to be transposed again, we actually have to read two 4×4-blocks at a time, producing one low-pass block and one high-pass block, which are transposed and stored separately. As in [2] the transposition is done with SIMD shuffle instructions. One such transposition requires 8 instructions. This makes $4 \cdot 8 = 32$ instructions for every 8 input words, or 8 instructions for every two input words, which is less than the 11 instructions of the pure horizontal parallelization.

Figure 6 shows execution times of the SIMD parallelized double-loop algorithms compared to the SISD double-loop algorithm of Section 3. We see that there is another performance gain by a factor of about 2.8 over the whole range of image sizes. The transposition-based parallelization is slightly better than the pure horizontal approach (denoted line-SIMD), mainly due to the lower number of total shuffle operations.

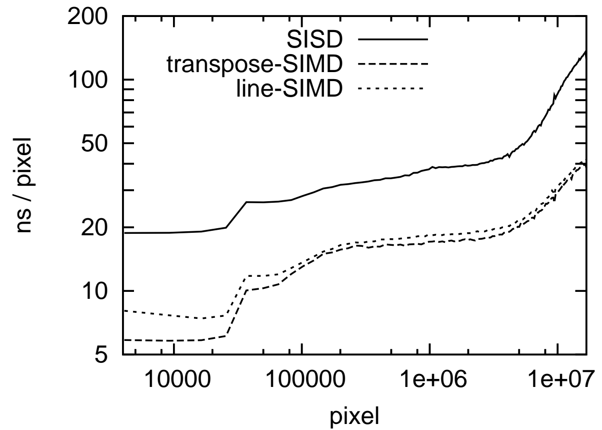

Figure 6. Execution times per pixel for the double-loop implementations with and without SIMD.

5. Single-Loop

As we have seen, the algorithms we encountered so far have not reduced the number of loops to one. So the promise of the title is still to be fulfilled. In the following, there will be two nested loops, an outer vertical and an inner horizontal loop. However, we shall consider this as a single loop over all pixels of the image. (The algorithm could actually be implemented this way although this would be impractical.)

In the 1-D case, we pass four values from one iteration to the other. To do a similar thing in the second dimension, we apply an approach that is known as pipeline or line-based computation [3]. If we imagine a whole row as a single value (as in the easy vertical SIMD algorithm, only with words of the size of a whole row), we must pass four such rows from one iteration to the other. This amounts to a buffer of four rows. In the 1-D case, we read two values from memory in a single iteration. In our row-wise approach this means that we need two new rows to start an iteration.

Since the source data for this row-wise vertical filtering is the output of the horizontal filtering, we try to use the output of the horizontal filtering in the vertical transform immediately after it is available. Thus, we have to perform two horizontal filterings (on two consecutive rows) at once. For each row we get a low-pass and a high-pass coefficient, which makes four values in total. The two low-pass values are fed into an iteration of the vertical type which produces an LL- and an LH-type coefficient, followed by the same operation on the two high-pass coefficients which produces an HL- and an HH-type coefficient. In each iteration the vertical part updates four values in the four-row buffer, which are reused when the next two rows are processed.

All this can be depicted as a simple signal-processing model, which is shown in Figure 7. The first module represents the horizontal filtering, producing a low-pass and a

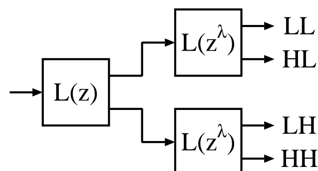

Figure 7. Signal processing model of the single-loop algorithm. $L(z)$ is defined in Figure 3. z^λ denotes the delay of a whole row (λ being the image width). In the latter case, downsampling has to be applied to rows, i.e. one row is passed to the output, the other is not.

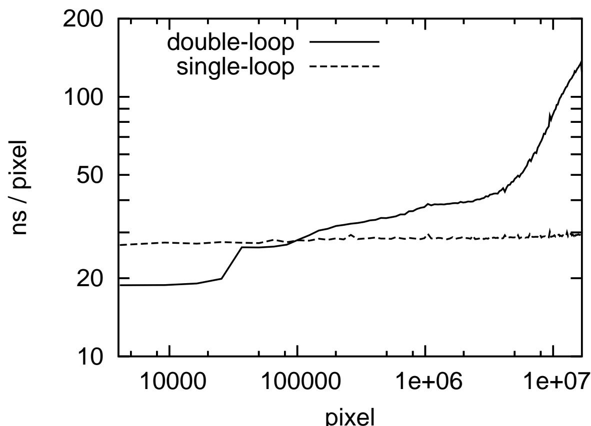

Figure 8. Execution times for the single-loop implementation compared to the double-loop implementation.

high-pass data stream. Each of these is processed vertically by $L(z^\lambda)$ where λ is the image width. In $L(z^\lambda)$ all occurrences of z^{-k} in $L(z)$ of Figure 3 are replaced by $z^{-\lambda k}$, representing a delay of k rows. Downsampling in $L(z^\lambda)$ has to be applied to rows, i.e. one row is passed to the output, the other is not.

Although this model seems to be very simple, the implementation is quite complicated. The main problem are the prolog and epilog phases (see Figure 2). Both the horizontal as well as the vertical part have three kinds of phases: prolog, main, and epilog. As in the 1-D case, these have to be implemented individually. However, now we have actually nine phases for all combinations of horizontal and vertical phases. This makes the coding arduous since there are no language features that support such an attempt of "code-time scheduling of concurrent tasks".

However, we finally have a *single-loop* algorithm that performs vertical and horizontal filtering in one pass. Each pixel is read only once from memory and each output coefficient is written once to memory and not updated anymore. To see the performance gain of this approach, look at Figure 8. It compares the single-loop algorithm to the double-loop algorithm. There is some speedup for image sizes over 100,000 pixel. The most impressive fact, though, is that the execution time per pixel is now constant for the single-loop algorithm, which means that the implementation scales linearly with the image size due to reduced dependence on cache performance. This leads to high speedups for large images.

Note that this approach is also useful for memory limited applications [4, 3, 5] such as hardware implementations for streaming image data. The buffer needed in this algorithm is only 4 rows for the passing of data between iterations and 1 row because we have to process two rows at once in the vertical part. The idea of the single loop can also be extended to the whole wavelet transform, although the implementation becomes even more complicated. In this case the buffer size is $5(1 + \frac{1}{2} + \frac{1}{4} + \ldots) \approx 10$ rows in size. See [13] for a hardware implementation of an equivalent scheme.

6. Single-Loop SIMD Parallelization

Finally, the single-loop algorithm can be SIMD parallelized. The same ideas apply as in the sequential case. For the horizontal and vertical parts of each iteration, the SIMD parallelized versions of the 1-D single-loop algorithm of Section 3 are used. All memory accesses are through packed words. The problem of the nine phases has to be coped with here as well.

Both, the pure line-SIMD and the transposition based approach can be used for the horizontal part. In the latter case, the vertical filtering has to be performed in chunks of 4 lines, which unfortunately increases the required buffer size to 8 rows of data for one filtering step, or 16 rows for a whole transform.

Figure 9 shows the execution time of the final SIMD parallelized single-loop algorithms compared to the non-parallelized version of the single-loop algorithm. Fortunately, the constancy of the execution time per pixel over the whole range of image sizes is still present. This time the transposition based algorithm is significantly worse than the pure line-SIMD approach. The reason for this is increased buffer size destroying data locality, and an increased number of concurrently processed intermediate words per iteration making register allocation more difficult. The line-SIMD algorithm, however, performs about 3.7 times faster than the non-parallelized, which is very close to the theoretical maximum of 4.

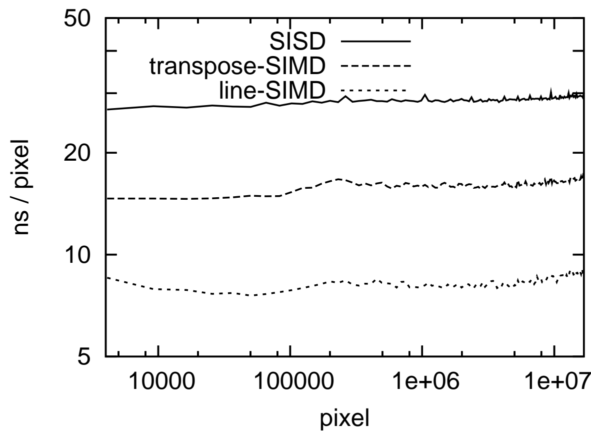

Figure 9. Execution times per pixel for the single-loop implementation with and without SIMD.

7. Conclusion

We have seen several performance boosts in this work. The first simple improvement was array padding, which avoided pathological cache misses for certain image sizes. The development of single-loop variants of the lifting algorithm yielded significant reduction of memory access for the 1-D case involved in the 2-D double-loop algorithm as well as for 2-D single-loop algorithm which incorporates horizontal and vertical filtering into a single loop over the whole image by a pipelining-technique. It was shown that the single-loop approach allows an efficient SIMD parallelization which can be combined to a 2-D single-loop SIMD algorithm which outperforms all other variants significantly.

Array padding does not only serve as a performance improvement, it also shows the cache dependence of the algorithms presented in this work. Figure 10 shows a comparison of the speedups array padding causes for each of the algorithms. The fact that there is no speedup for the single-loop variants shows that these have very low cache dependence. This also shows that hardware implementations for streaming wavelet lifting with low memory demand can be derived.

Figure 11 shows a final speedup comparison for all algorithms. These speedups are relative to the array-padded standard multi-loop algorithm. A speedup of 9.5 for image sizes of one megapixel and above is achieved, which consists of 2.55 due to the single-loop algorithm and 3.7 due to the use of SIMD. Because the reference algorithm also contains some speedups (see multi-loop in Figure 10) we can multiply these speedups, so we get speedups of up to 56 for pathological image sizes (powers of two and similar) which

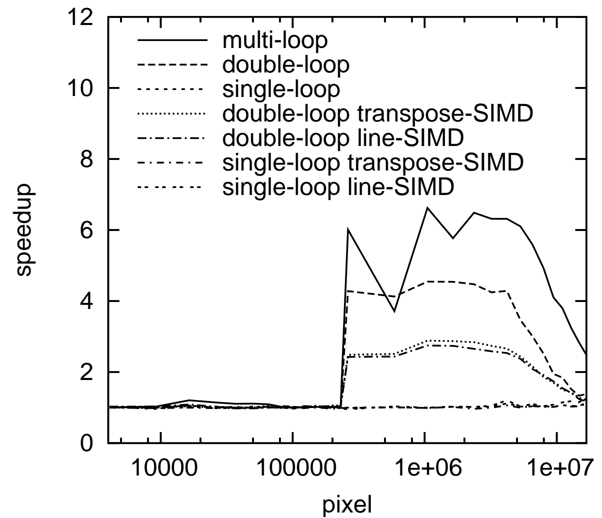

Figure 10. Effect of array padding on each algorithm. This graph shows only execution times for critical image sizes (compare Figure 1). One can see that the single-loop implementations are able to resolve all cache problems since there is no additional speedup for array padding.

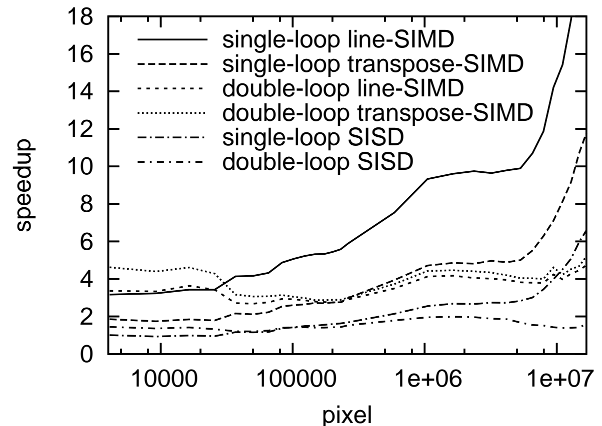

Figure 11. Speedups of each algorithm compared to the array-padded multi-loop implementation. These speedups can be multiplied by the speedups for multi-loop in Figure 10 to compare each algorithm to the completely non-optimized implementation.

are actually quite common. Although the speedups become even higher in Figure 11 for very large images, this is compensated by decreasing speedups in Figure 10, so the total speedup remains at about 56 for all image sizes above one megapixel.

References

[1] C. Chakrabarti and M. Vishvanath. Efficient realizations of the discrete and continuous wavelet transforms: From single chip implementations to mappings on SIMD array computers. *IEEE Transactions on Signal Processing*, 3(43):759–771, 1995.

[2] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. 2-D wavelet transform enhancement on general-purpose microprocessors: Memory hierarchy and SIMD parallelism exploitation. In *Proceedings of the 2000 International Conference on High Performance Computing*, Bangalore, India, Dec. 2002.

[3] C. Chrysafis and A. Ortega. Line based, reduced memory, wavelet image compression. *IEEE Transactions on Image Processing*, 9(3):378–389, Mar. 2000.

[4] C. Chrysafis and A. Ortega. Minimum memory implementations of the lifting scheme. In *Proceedings of SPIE, International Symposium on Optical Science and Technology*, San Diego, CA, USA, July 2000.

[5] P. C. Cosman and K. Zeger. Memory constrained wavelet-based image coding. *IEEE Signal Processing Letters*, 5(9):221—223, 1998.

[6] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis Applications*, 4(3):245–267, 1998.

[7] M. Feil and A. Uhl. Wavelet packet decomposition and best basis selection on massively parallel SIMD arrays. In *Proceedings of the International Conference "Wavelets and Multiscale Methods" (IWC'98), Tangier, 1998*. INRIA, Rocquencourt, Apr. 1998. 4 pages.

[8] J. Fridman and E. Manolakos. On the scalability of 2D discrete wavelet transform algorithms. *Multidimensional Systems and Signal Processing*, 8(1–2):185–217, 1997.

[9] ISO/IEC JPEG committee. JPEG 2000 image coding system — ISO/IEC 15444-1:2000, Dec. 2000.

[10] R. Kutil, P. Eder, and M. Watzl. SIMD parallelization of common wavelet filters. In *Parallel Numerics '05*, pages 141–149, Portorož, Slovenia, Apr. 2005.

[11] R. Kutil and A. Uhl. Hardware and software aspects for 3-D wavelet decomposition on shared memory MIMD computers. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *Parallel Computation. Proceedings of ACPC'99*, volume 1557 of *Lecture Notes on Computer Science*, pages 347–356. Springer-Verlag, 1999.

[12] R. Kutil and A. Uhl. Optimization of 3-d wavelet decomposition on multiprocessors. *Journal of Computing and Information Technology (Special Issue on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia)*, 8(1):31–40, 2000.

[13] G. Lafruit, B. Vanhoof, L. Nachtergaele, F. Catthoor, and J. Bormans. The local wavelet transform: a memory-efficient, high-speed architecture optimized to a region-oriented zero-tree coder. *Integrated Computer-Aided Engineering*, 7(2):89–103, Mar. 2000.

[14] P. Meerwald, R. Norcen, and A. Uhl. Cache issues with JPEG2000 wavelet lifting. In C.-C. J. Kuo, editor, *Visual Communications and Image Processing 2002 (VCIP'02)*, volume 4671 of *SPIE Proceedings*, pages 626–634, San Jose, CA, USA, January 2002. SPIE.

[15] M. Pic, H. Essafi, and D. Juvin. Wavelet transform on parallel SIMD architectures. In F. Huck and R. Juday, editors, *Visual Information Processing II*, volume 1961 of *SPIE Proceedings*, pages 316–323. SPIE, Aug. 1993.

[16] A. Said and W. A. Pearlman. A new, fast, and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3):243–249, June 1996.

[17] W. Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Appl. Comput. Harmon. Anal*, 3(2):186–200, 1996.

[18] W. Sweldens. The lifting scheme: A construction of second generation wavelets. *Siam J. Math. Anal.*, 29(2):511–546, 1997.

[19] C. Tenllado, D. Chaver, L. Piñuel, M. Prieto, and F. Tirado. Vectorization of the 2D wavelet lifting transform using SIMD extensions. In *Workshop on Parallel and Distributed Image Processing, Video Processing, and Multimedia, PDIVM '03*, Nice, France, Apr. 2003.

[20] M.-L. Woo. Parallel discrete wavelet transform on the Paragon MIMD machine. In R. Schreiber et al., editors, *Proceedings of the seventh SIAM conference on parallel processing for scientific computing*, pages 3–8, 1995.