

1 Guided Vectorization and Data Types for Vector Intrinsics

1.1 Vectorization Clauses

aligned

This clause tells the compiler that every element of an array a is n -bit aligned if one uses `#pragma omp simd aligned(a, n)`. The compiler employs this information to effectively vectorize the code section.

safelen

Imagine we had the following code:

```
a[0] = b[0];  
for (int i = 1; i < n; i++) {  
    a[i] = a[i-1] * (i % 2) + b[i];  
}
```

We have a data dependency, but only every second iteration: iteration i , where i is odd, depends on $i - 1$. Iteration i and $i + 1$ are completely independent, however. Hence, we can vectorize the loop with a vector length of 2. To specify this, we use the **safelen** clause like so:

```
a[0] = b[0];  
#pragma omp simd safelen(2)  
for (int i = 1; i < n; i++) {  
    a[i] = a[i-1] * (i % 2) + b[i];  
}
```

reduction

This clause has the same functionality when paired with **simd** as when paired with **parallel for**: It will reduce a variable by an associative operation. But instead of reducing it over multiple threads, we use vectorization for reducing. I imagine that the reduced variable itself will be vectorized, so that in the loop we can employ vector operations, and at the end the elements in the resulting vector will get reduced.

1.1.1 AVX512 data types

..m512	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	16x 32-bit float
..m512d	D		D		D		D		D		D		D		D		8x 64-bit double	
..m512i																	64x 8-bit byte	
..m512i	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	32x 16-bit short
..m512i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	16x 32-bit integer
..m512i	ll		ll		ll		ll		ll		ll		ll		ll		8x 64-bit long	
..m512i	dqw				dqw				dqw				dqw				4x 128-bit quad	
..m512i	qqw								qqw								2x 256-bit integer	
..m512i	dqqw																1x 512-bit integer	

1.1.2 Intel MMX

In the paper [Intel MMX for Multimedia PCs](#) the authors introduced the new MMX technology, which they integrated into their new cpus. It allows the cpu to employ vector instructions to increase throughput and hence optimize running time of common computing tasks, such as rendering pixels for example. They illustrated the use cases in several examples, two of them being:

Vectorizing Branches Say we want overlay sprite onto a scene, i.e. we have a pixel array representing the sprite, and another one representing the scene (so far). The sprite array consists of color values. However, there is one constant value which represents a *clear color*, which represents the sprite not being visible at that point, and hence it should not get rendered onto the scene. The overlaying operation can be performed by:

```
for i = 1 to Sprite_Line_size
if A[i] = clear_color then
    Out_frame[i] = C[i] else
    Out_frame[i] = A[i]
```

This is a typical example of branching inside a loop, where assignments have the following form:

```
a[i] = expr_1 if cond else expr_2
```

The idea is that we can vectorize this using vector instructions by:

$$a[i] = (\text{expr}_1 \wedge \text{cond}) \vee (\text{expr}_2 \wedge \neg \text{cond})$$

Where `cond` is evaluated to `111...111` if true, and to `000...000` if false. The expressions should be easily vectorizable themselves, such that ultimately the entire statement can be vectorized efficiently.

To see this in action, the authors provided a graphic for the example discussed earlier:

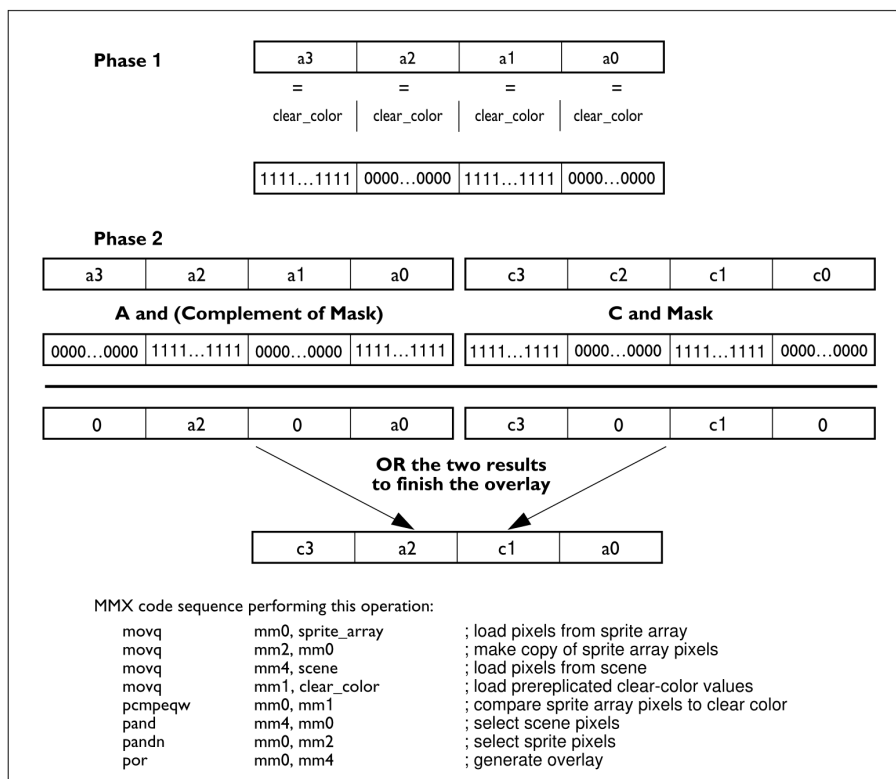


Figure 5. Overlay operation using packed compare

In *Phase 1*, the mask is computed, i.e. the condition evaluated (the condition is that the color value equals the `clear_color` constant).

Next, in *Phase 2*, we evaluate the expressions (in this case it is omitted), and mask the results with the appropriate mask (`mask`, and `¬mask`). Finally, the two results get reduced by a logical or, yielding the final output.

At the bottom of the graphic is a little assembly code section, emphasizing the fact that the operation can entirely be done with vector instructions.

Matrix Transposition Vector instructions can be fully utilized if the data is stored sequentially in memory. Hence, accessing rows of matrices can be vectorized. But if we want to work on the columns of the matrix, it might be beneficial to transpose the matrix. This transposing can be vectorized as well by employing the so called *unpack* operation, which interleaves two operands word-wise, as can be seen in the picture:

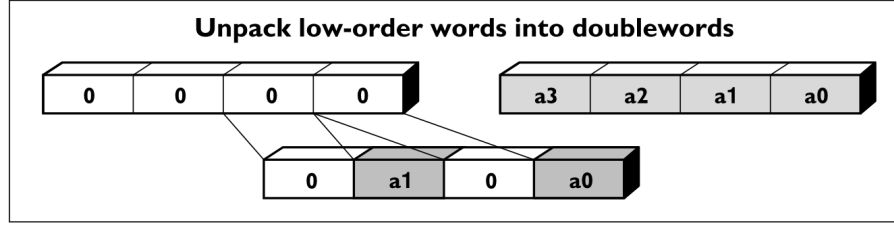


Figure 6. MMX technology Unpack high and low instructions on word data type

To now transpose an entire matrix we go on like this: Imagine for the sake of simplicity that we have the following matrix M :

$$M = \begin{pmatrix} d3 & d2 & d1 & d0 \\ c3 & c2 & c1 & c0 \\ b3 & b2 & b1 & b0 \\ a3 & a2 & a1 & a0 \end{pmatrix}$$

We now compute the last two rows of the matrix by firstly unpacking the last two entries of two adjacent rows, in this case d and c , and b and a , like shown in *Phase 1* in the following graphic:

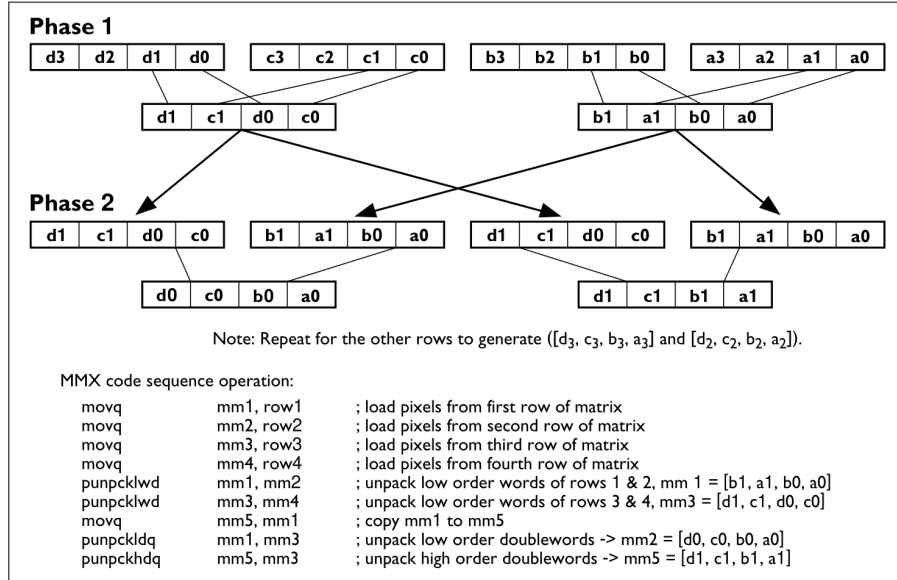


Figure 7. Matrix transposition using Unpack instructions

We now created two *column slices* of length two in every result of the unpack operations. Thus, we now need to unpack those among themselves, to get longer column slices, or in this case entire columns, as demonstrated in *Phase 2*. Once again, all the operations can be implemented with vector instructions, as hinted by the small assembly program at the bottom.