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 opm 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];
}
```

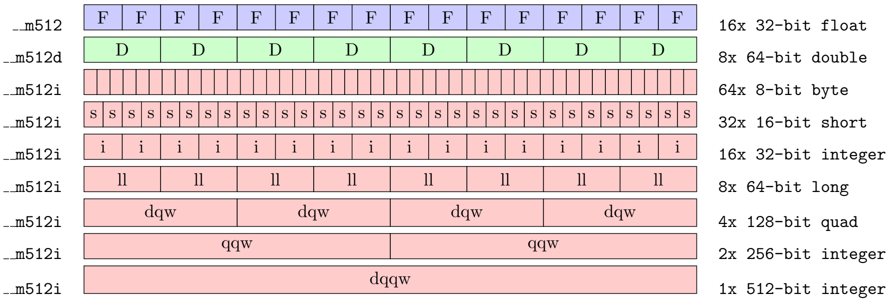
We have a data dependency, but only every second iteration: iteration i, where i is odd, depends on $i - 1$. Iteration i and $i + 1$ are completely independent, however. Hence, we can vectorize the loop with a vector length of 2. To specify this, we use the `safelen` clause like so:

```
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 `paraller 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 the in 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 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 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 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 throuphput 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 the 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:

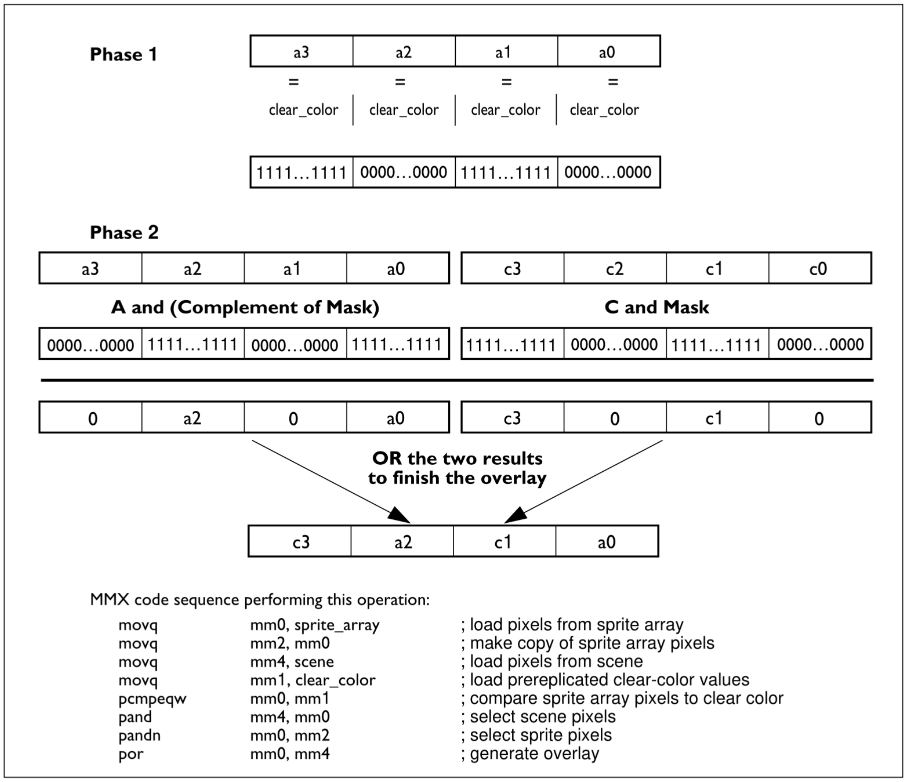
```
a[i] = expr_1 if cond else expr_2
```

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

```
a[i] = (expr_1 ∧ cond) ∨ (expr_2 ∧ ¬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:



MMX code sequence performing this operation:

```
    movq        mm0, sprite_array      ; load pixels from sprite array
    movq        mm2, mm0               ; make copy of sprite array pixels
    movq        mm4, scene             ; load pixels from scene
    movq        mm1, clear_color       ; load prereplicated clear-color values
    pcmpeqw     mm0, mm1               ; compare sprite array pixels to clear color
    pand        mm4, mm0               ; select scene pixels
    pandn       mm0, mm2               ; select sprite pixels
    por         mm0, mm4               ; generate overlay
```

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 vectorized. But I 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:

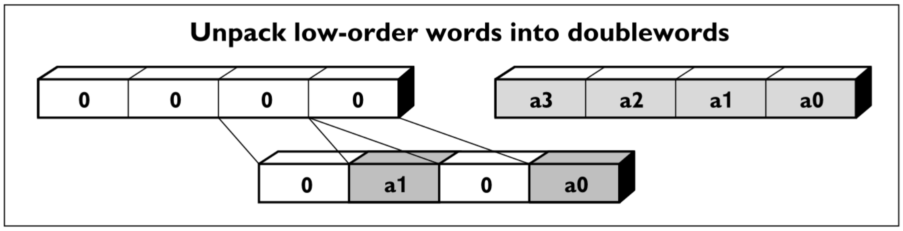
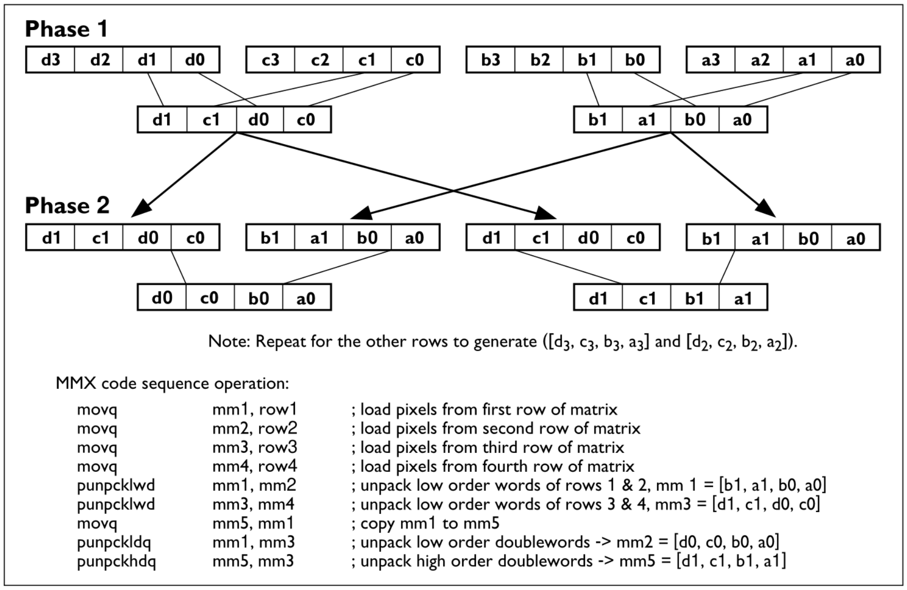


Figure 6. MMX technology Unpack high and low instructions on word data type

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:



Note: Repeat for the other rows to generate ($[d_3, c_3, b_3, a_3]$ and $[d_2, c_2, b_2, a_2]$).

MMX code sequence operation:

```
movq         mm1, row1     ; load pixels from first row of matrix
movq         mm2, row2     ; load pixels from second row of matrix
movq         mm3, row3     ; load pixels from third row of matrix
movq         mm4, row4     ; load pixels from fourth row of matrix
punpcklwd    mm1, mm2      ; unpack low order words of rows 1 & 2, mm 1 = [b1, a1, b0, a0]
punpcklwd    mm3, mm4      ; unpack low order words of rows 3 & 4, mm3 = [d1, c1, d0, c0]
movq         mm5, mm1      ; copy mm1 to mm5
punpckldq    mm1, mm3      ; unpack low order doublewords -> mm2 = [d0, c0, b0, a0]
punpckhdq    mm5, mm3      ; unpack high order doublewords -> mm5 = [d1, c1, b1, a1]
```

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.