1 Auto Vectorization

1.1 Characteristics of SSE, AVX(2) and AVX-512

SSE (Streaming SIMD Extensions):
Introduced in 1999 by Intel in its Pentium III series[1], SSE supports 128-bit vector registers, enabling SIMD operations on 4 single-precision floating-point or 2 double-precision values in parallel. Limited in width and functionality compared to later extensions.

AVX/AVX2 (Advanced Vector Extensions):
AVX were proposed by Intel in March 2008 and they extended the SIMD width from 128 to 256 bits, doubling parallelism compared to SSE[2]. AVX employs 16 vector registers and supports various additional instructions, like for instance so called *three-operand SIMD instructions*, which store the result of the operation in a third independent register. AVX2 expand most integer instructions to 256 bits and introduced yet again new features like gathering vector elements from non-contiguous memory locations.

AVX-512:
Widens SIMD to 512 bits, allowing operations on 16 single-precision floats or 8 double-precision values simultaneously. It increases the number of registers to 32, and adds 8 new mask registers for conditional operations[3]. New instructions and features were added, like including 4 operand operations, and introducing explicit rounding control, etc.

1.2 Impact of Memory Aliasing on Performance

Memory aliasing occurs when two or more pointers reference overlapping memory regions, making it difficult for the compiler or CPU to optimize memory accesses. To illustrate this, say we have the following code:

```
void add(float *a, float *b, int n) {
    for (int i = 0; i < n; i++) {
        a[i] = a[i] + b[i];
    }
}
```

[1]Source: https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
[2]Source: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions
[3]Source: https://en.wikipedia.org/wiki/AVX-512

At first glance, it looks well written such that the compiler has no difficulty vectorizing it. But this is a fallacy, as the programmer has made the hidden assumption that `a` and `b` don't overlap. For the argument's sake, let's assume that `a = b + 1`. This would mean that when updating `a[i]`, we also update `b[i+1]`, which is accessed in the following iteration. Thus, we have a dependency of every iteration to the previous one, and hence vectorizing it would change the the programs behavior and hence the compiler won't do it. The solution of course is to mark `a` and `b` as `__restrict__` if they indeed point to not-overlapping memory regions.

1.3 Advantages of Unit Stride Memory Access

Unit stride (stride-1) memory access sequentially accesses adjacent memory locations, maximizing cache utilization and resulting in higher bandwidth utilization. Furthermore, unit stride access patterns facilitate the utilization of efficient vector instructions employed by the compiler. Larger strides on the other hand, like stride-8, result in less efficient cache usage and increased memory latency. Vectorization is also harder and less efficient. Thus, it is generally a good advise trying to employ unit strides when accessing the memory whenever possible.

1.4 When to Prefer Structure of Arrays

Structure of Arrays (SoA) is preferred when working with SIMD/vectorized operations or when the workload requires processing individual fields of a dataset independently. This is simply due to the fact of strided and homogenous memory access like discussed previously. Especially for big data with these mentioned operations one should consider SoA over AoS (Array of Structures), which, on the other hand, may be used in less compute intensive tasks to improve readability and maintainability of the code, or in cases where the access patterns would align with such a layout.