Coding for SingleInstructionMultipleData processing

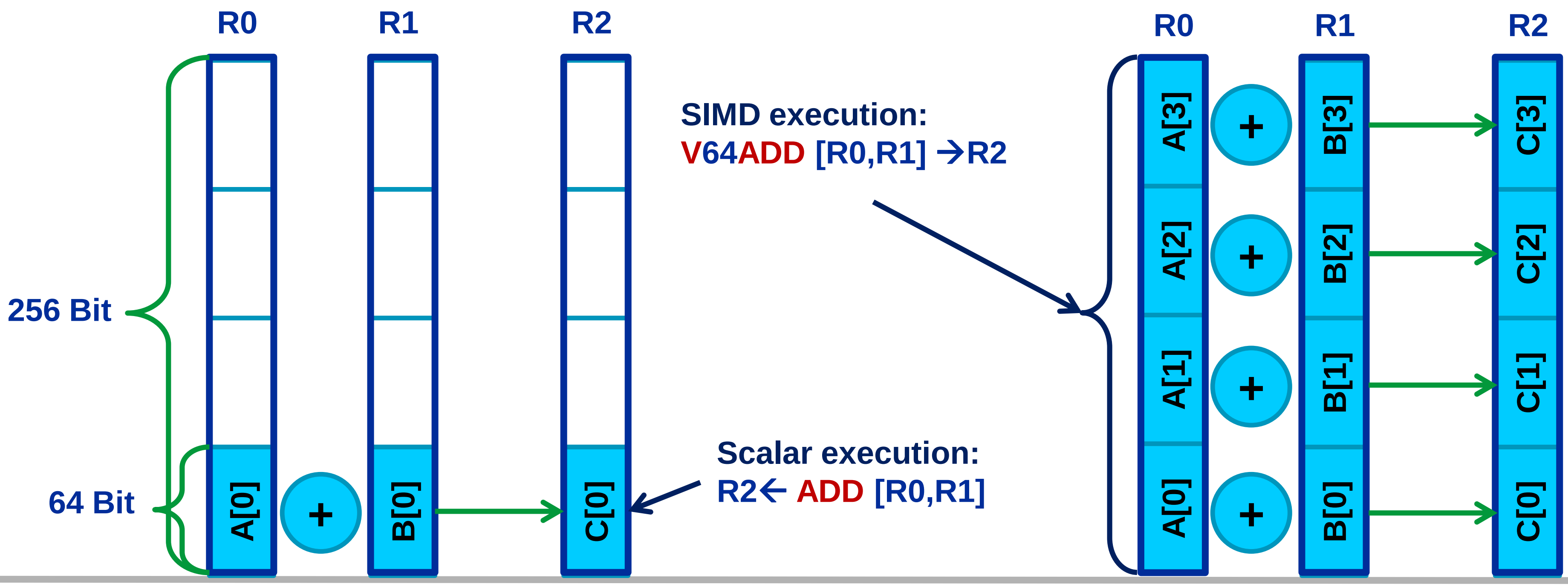
Basics

Compiler options

Reading x86 assembly language

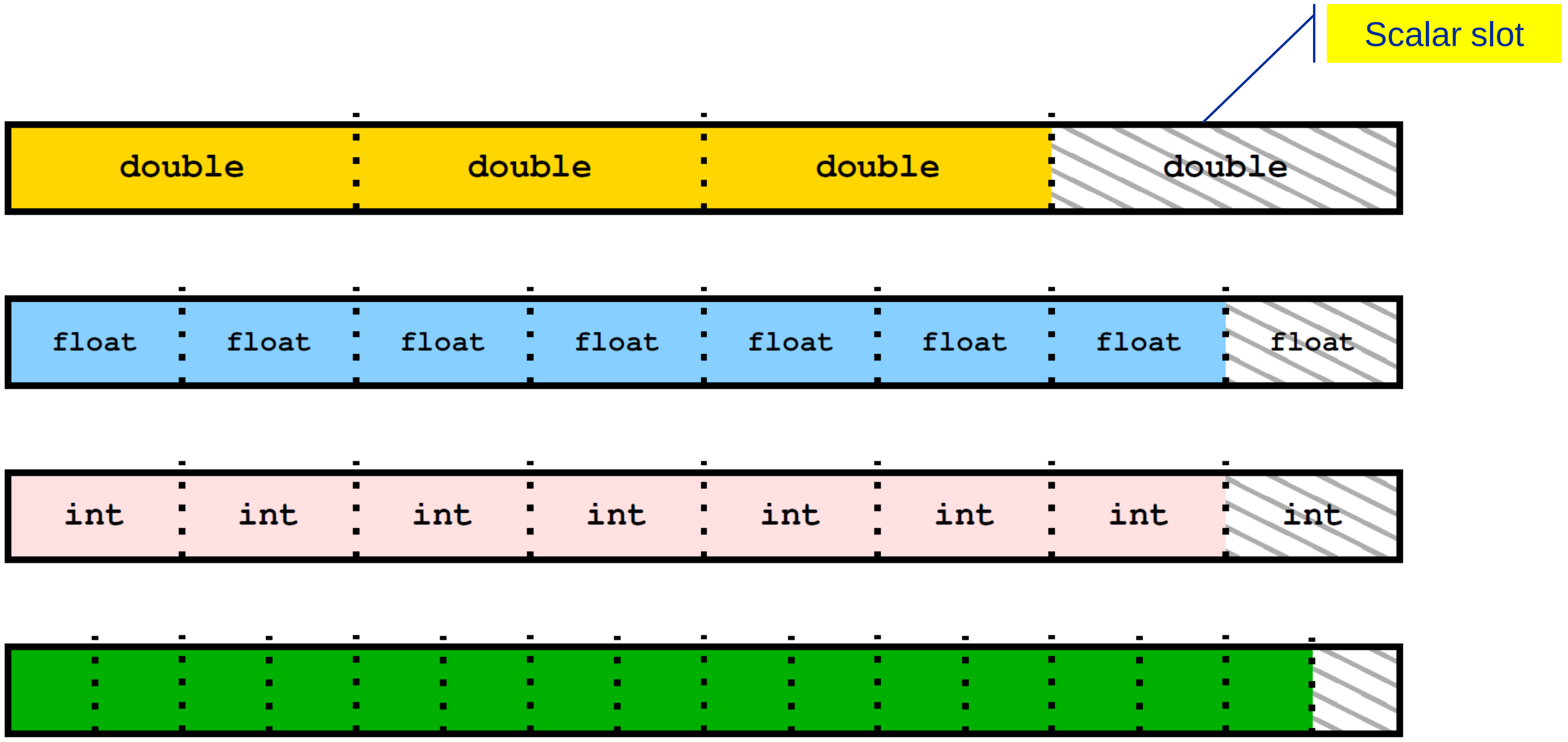
SIMD processing – Basics

- Single Instruction Multiple Data (SIMD) operations allow the concurrent execution of the same operation on “wide” registers.
- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands
- Adding two registers holding double precision floating point operands

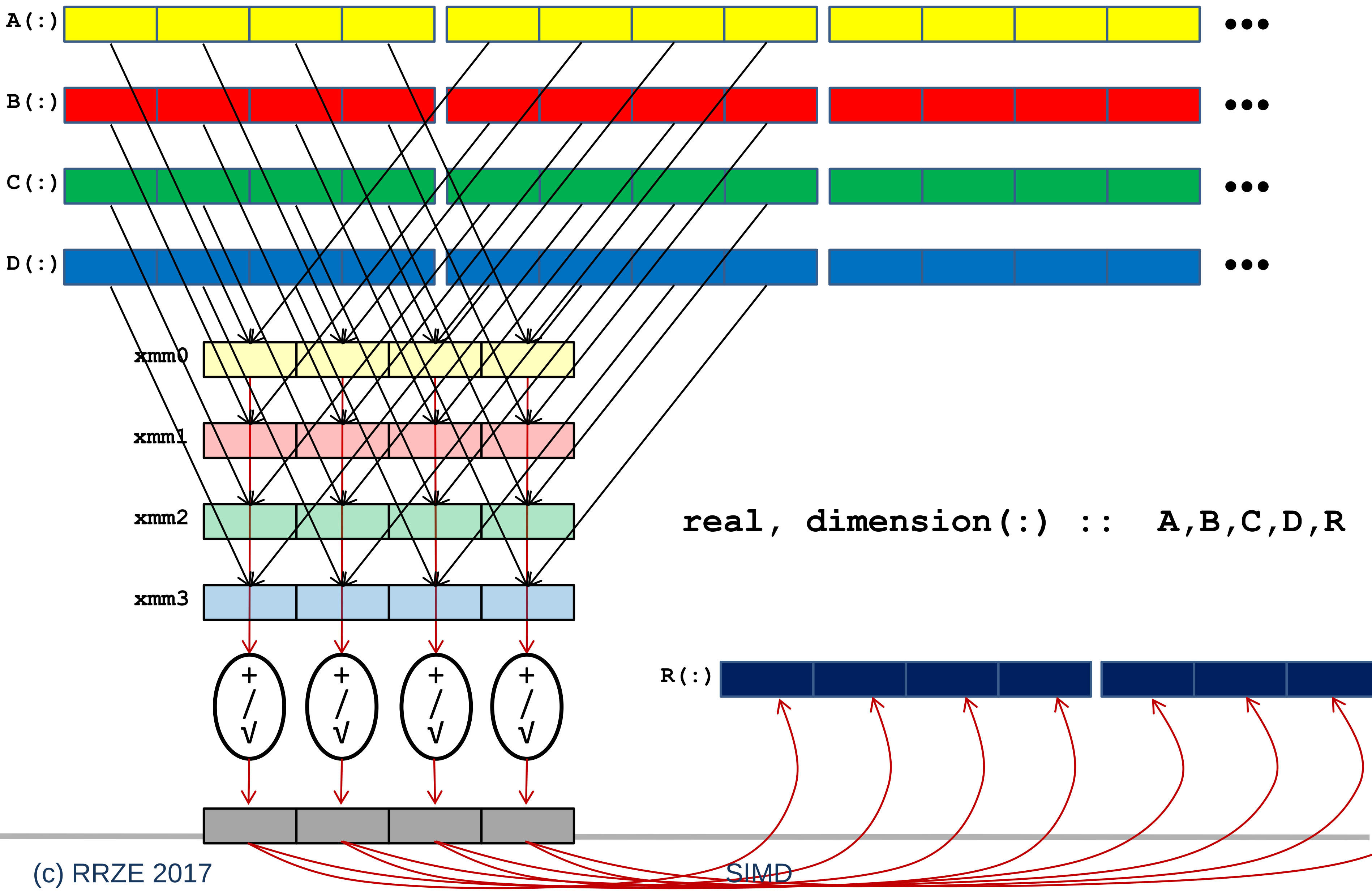


Data types in 32-byte SIMD registers

Supported data types depend on actual SIMD instruction set

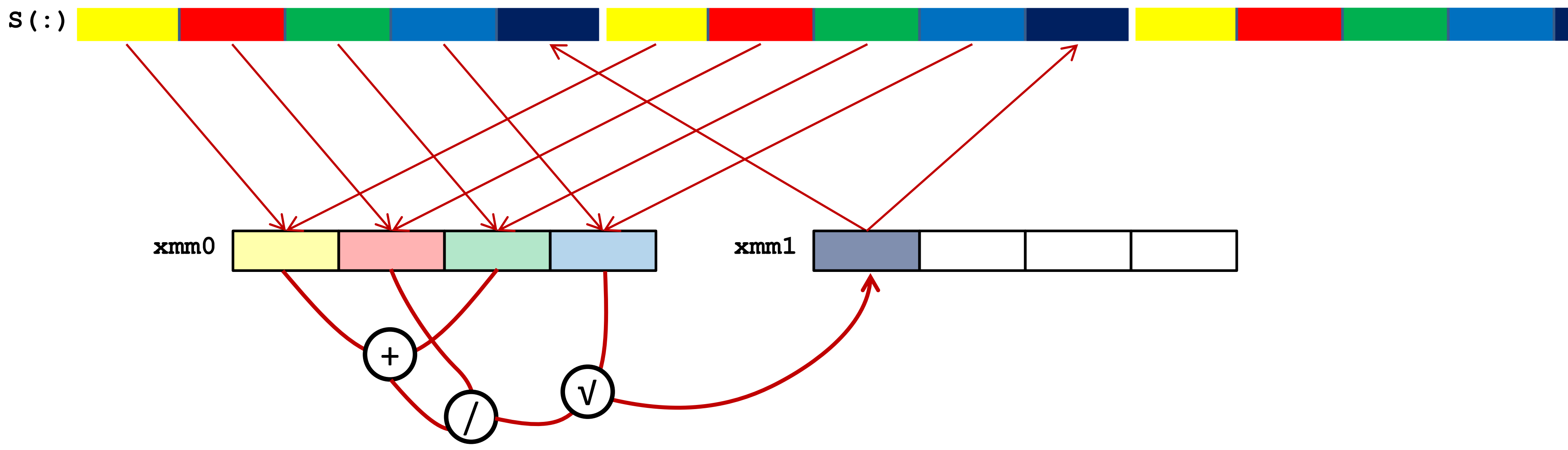


SIMD style structure of arrays processing



```
real, dimension(:) ::  A,B,C,D,R
```

SIMD style of array of structures processing



```
type struct
  real*4  A,B,C,D,R
end type struct

type(struct), dimension(:) :: S
```

SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n;i++)
        C[i]=A[i]+B[i];
```

“Loop unrolling”

```
for(int i=0; i<n;i+=4){
        C[i]  =A[i]  +B[i];
        C[i+1]=A[i+1]+B[i+1];
        C[i+2]=A[i+2]+B[i+2];
        C[i+3]=A[i+3]+B[i+3];}
//remainder loop handling
```

Don’t unroll for SIMD yourself – leave it to the compiler!

Load 256 Bits starting from address of `A[i]` to register `R0`

Add the corresponding 64 Bit entries in `R0` and `R1` and store the 4 results to `R2`

Store `R2` (256 Bit) to address starting at `C[i]`

```
LABEL1:
        VLOAD R0 ← A[i]
        VLOAD R1 ← B[i]
        V64ADD[R0,R1] → R2
        VSTORE R2 → C[i]
        i←i+4
        i<(n-4)? JMP LABEL1
//remainder loop handling
```

SIMD processing – Basics

- No SIMD vectorization for loops with data dependencies:

```
for(int i=0; i<n;i++)
       A[i]=A[i-1]*s;
```

- "Pointer aliasing" may prevent SIMDfication

```
void scale_shift(double *A, double *B, double *C, int n) {
       for(int i=0; i<n; ++i)
          C[i] = A[i] + B[i];
}
```

  - C/C++ allows that `A` → `&C[-1]` and `B` → `&C[-2]`
    → `C[i] = C[i-1] + C[i-2]`: **dependency → No SIMD**

  - **If "pointer aliasing" is not used, tell it to the compiler:**
    - `-fno-alias` (Intel), `-Msafeptr` (PGI), `-fargument-noalias` (gcc)
    - `restrict` keyword (C only!):
      `void f(double restrict *a, double restrict *b) {…}`

Vectorization compiler options (Intel)

optional

- **The compiler will vectorize starting with –`O2`.**
- **To enable specific SIMD extensions use the –x option:**
  - **`-xSSE2`** vectorize for SSE2 capable machines

  Available SIMD extensions:

  `SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, ...`

`-xAVX`	on Sandy/Ivy Bridge processors
`-xCORE-AVX2`	on Haswell/Broadwell
`-xCORE-AVX512`	on Skylake (certain models)
`-xMIC-AVX512`	on Xeon Phi Knights Landing

Recommended option:

- **`-xHost`** will optimize for the architecture you compile on
  (Caveat: do not use on standalone KNL, use MIC-AVX512)

Vectorization compiler options

optional

- **Controlling non-temporal stores (part of the SIMD extensions)**
  - `-opt-streaming-stores always|auto|never`

`always`	use NT stores, assume application is memory bound (use with caution!)
`auto`	compiler decides when to use NT stores
`never`	do not use NT stores unless activated by source code directive

  - NT stores are always vectorized and need SIMD-aligned memory addresses

Vectorization source code directives

optional

- **Fine-grained control of loop vectorization**
- **Use `!DEC$` (Fortran) or `#pragma` (C/C++) sentinel to start a compiler directive**

- **`#pragma vector always`**
  vectorize even if it seems inefficient (hint!)
- **`#pragma novector`**
  do not vectorize even if possible (use with caution)
- **`#pragma vector nontemporal`**
  use NT stores if possible (i.e., SIMD and lignment conditions are met)
- **`#pragma vector aligned`**
  - specifies that all array accesses are aligned to SIMD address boundaries (DANGEROUS! You must not lie about this!)
  - Evil interactions with OpenMP possible
- **`#pragma ivdep`**
  Ignore loop-carried dependencies; not really a SIMD option

User mandated vectorization

- **Since Intel Compiler 12.0 the `simd` pragma is available**
- **`#pragma simd` enforces vectorization where the other pragmas fail**
- **Prerequesites:**
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- **There are additional clauses: `reduction, vectorlength, private`**
- **Refer to the compiler manual for further details**

```
#pragma simd reduction(+:x)
  for (int i=0; i<n; i++) {
     x = x + A[i];
  }
```

- **NOTE: Using the #pragma simd the compiler may generate incorrect code if the loop violates the vectorization rules!**

x86 Architecture:
SIMD and Alignment

- **Alignment issues**
  - Alignment of arrays should optimally be on SIMD-width address boundaries to allow packed aligned loads (and NT stores on x86)
  - Otherwise the compiler will revert to unaligned loads/stores
  - Modern x86 CPUs have less (not zero) impact for misaligned LOAD/STORE, but **Xeon Phi KNC relies heavily on it!**
  - How is manual alignment accomplished?

- **Stack variables: `alignas` keyword (C++11/C11)**
- **Dynamic allocation of aligned memory (`align` = alignment boundary)**
  - C before C11 and C++ before C++17:
    `posix_memalign(void **ptr, size_t align, size_t size);`
  - C11 and C++17:
    `aligned_alloc(size_t align, size_t size);`

How to leverage SIMD: your options

Alternatives:

- The **compiler** does it for you (but: aliasing, alignment, language)
- Compiler directives (**pragmas**)
- Alternative **programming models** for compute kernels (OpenCL, ispc)
- **Intrinsics** (restricted to C/C++)
- Implement directly in **assembler**

To use intrinsics the following headers are available:

- `xmmintrin.h` (SSE)
- `pmmintrin.h` (SSE2)
- `immintrin.h` (AVX)
- `x86intrin.h` (all extensions)

```
for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
```

Rules for vectorizable loops

1. Inner loop
2. Countable (loop length can be determined at loop entry)
3. Single entry and single exit
4. Straight line code (no conditionals)
5. No (unresolvable) read-after-write data dependencies
6. No function calls (exception intrinsic math functions)

Better performance with:

1. Simple inner loops with unit stride (contiguous data access)
2. Minimize indirect addressing
3. Align data structures to SIMD width boundary
4. In C use the **`restrict`** keyword and/or **`const`** qualifiers and/or compiler options to rule out array/pointer aliasing