C++ summit

Sven Johannsen
sven@sven-johannsen.de
www.sven-johannsen.de

C++ User Group Aachen, 13.07.2022

Content

- Motivation
- Simple math with SSE / AVX instructions
- History
- Vectorization (SIMD)
- Re-visit first example

Motivation

Post-performance discussion about not portable code based on SSE intrinsics:

> "Readable" C++ code runs faster then high optimized code based on intrinsics.

Based on an introduction of SIMD for Intel/AMD CPU with simple addition instructions: (sum of floation point numbers)

Normalize 3D vector (x,y,z)

Calculate the unit vector (direction only, without lenght)

```
void normalizeVec3d(Vec3d<double>& val)
{
    const double norm = std::sqrt(val.x * val.x + val.y * val.y + val
    const double inv_norm = 1. / norm;
    val.x *= inv_norm;
    val.y *= inv_norm;
    val.z *= inv_norm;
}
```

"inplace" normalization of a 3d vector (x,y,z).

Modified example:

> Only for measurements, not ready for production! (e.g. no division with zero check)

Optimized, but slower(?) version

```
void normalizeSSE2(Vec3d<double>& val)
{
    __m128d x = { val.x };
    __m128d y = { val.y };
    __m128d z = { val.z };

    // hypot = x^2+y^2+z^2
    __m128d x2 = _mm_mul_sd(x, x);
    __m128d y2 = _mm_mul_sd(y, y);
    __m128d z2 = _mm_mul_sd(z, z);
    __m128d hypot = _mm_add_sd(_mm_add_sd(x2, y2), z2);

    // norm = sqrt(hypot)
    __m128d norm = _mm_sqrt_sd(hypot, hypot);
    // inv_norm = 1.0 / norm
    __m128d inv_norm = _mm_div_sd(_mm_set_sd(1.), norm);

    val.x = _mm_mul_sd(x, inv_norm)[0];
    val.y = _mm_mul_sd(y, inv_norm)[0];
    val.z = _mm_mul_sd(z, inv_norm)[0];
}
```

Optimized, but slower(?) version

Again modified version:

- One to one replacment for C++ code.
- Parameter as `struct Vec3d { double x,y,z };`
- One SSE register per parameter, no "shuffle"
- Not optimized for register usage. (SSE problem)

More portable code

use portable C++ standard function to calculate 3d norm (`std::hypot`)

```
void normalizeStdHypot(Vec3d<double>& val)
{
    // use C++ standard function (hypot) to calculate norm
    const double norm = std::hypot(val.x, val.y, val.z);
    const double inv_norm = 1. / norm;
    val.x *= inv_norm;
    val.y *= inv_norm;
    val.z *= inv_norm;
}
```

Code have same restrictions as the code before.

Benchmark results

(Linux 64bit, AMD Ryzen 7 PRO 5850U)

Benchmark	CPU (GCC)	CPU (Clang)
NormalizeVec3d (GCC)	9.89 ns	10.9 ns
NormalizeSSE2 (GCC)	11.0 ns	10.5 ns
NormalizeStdHypot (GCC)	14.0 ns	16.4 ns

Simple math with SSE / AVX instructions

based on adding floating-point numbers

```
float b = 3.14f;
float c = 2.71f

float a = b + c;
```

Assembler example

addnasm.asm

```
global add_f_asm

section .text

add_f_asm:
    addss XMM0, XMM1
    ret
```

C++ callee

```
extern "C" float add_f_asm(float, float);

float result = add_f_asm(3.14f, 2.71f);
```

Assembler example

How can this work?

- Where is my stack?
  - Parameters?
  - return value?

X86-64 ABIs

Traditional ABIs stores parameter on the stack for functions calls. (Windows 32 bit, Linux 1.x)

Modern ABIs (application binary interfaces) prefer registers.

The first integer and pointer parameters are stored in CPU registers:

- Windows: 4 Reg., RCX, RDX, R8, R9
- System V: 6 Reg., RDI, RSI, RDX, RCX, R8, R9

The first floating point arguments (float & double) are stored in the SSE registers

- Windows: 4 Reg.; **XMM0**, **XMM1**, XMM2, XMM3
- System V: 8 Reg., **XMM0**, **XMM1**, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7

The following parameters are stored on the stack.

Function result in RAX or **XMM0**.

System V amd64 ABI: Linux, MacOS, Solaris, ...

(Common speech: Itanium ABI)

Assembler example

```
extern "C" float add_f_asm(float left, float right);
```

```
;// first param (left) : XMM0
;// second param (right): XMM1
add_f_asm:

    ;// XMM0 = XMM0 + XMM1
    addss XMM0, XMM1

    ret ;// return XMM0
```

First parameter of `addss` is result and first input

Intrinsics example

replacment for inline assembler

```
#include <immintrin.h>

double add_f_sse(float left, float right)
{
    const __m128 left_r{left, 0.f, 0.f, 0.f};
    const __m128 right_r{right, 0.f, 0.f, 0.f};
    const __m128 result = _mm_add_ss(left_r, right_r);

    return result[0];
}

float result = add_f_sse(3.14f, 2.71f);
```

Intrinsics

Functions build in to the compiler, not from a library.

(GCC: built-in functions)

Examples (MSVC):

`__debugbreak`, `__nop`, `__movsb` and SSE, AVX, ... - equivalents.

References:

- https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (SSE, AVX, AVX512, ...)
- https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=msvc-170
- https://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html#x86-Built-in-Functions

Intrinsics example

(non verbose)

```
#include <xmmintrin.h>

double add_f_sse(float left, float right)
{
    return _mm_add_ss(__m128{left}, __m128{right}).m128_f32[0];
}

float result = add_f_sse(3.14f, 2.71f);
```

- __m128x: data type, compiler equivalent to the SSE register
  - __m128 (1 or 4 floats)
  - __m128d (1 or 2 doubles)
  - __m128i (integer, independent of the size)
- `_mm_add_ss`: intrinsic to force the compiler to generate the `addss` instruction
  - `addss` is a two parameter instruction. (result and first parameter in same register)
  - `_mm_add_ss`: two parameter plus result. Additional move instructions may needed.

Compiler generated SSE instructions

on AMD64 (X86-64, x64) platforms the compiler will generate SSE instructions.

```
float add(float left, float right)
{
    return left + right;
}
```

Generated output: (GCC with `-O2 -msse -std=c++17 -ftree-vectorize -mveclibabi=svml -funsafe-math-optimizations`)

```
add(float, float):
    addss xmm0, xmm1
    ret
```

see: https://godbolt.org/z/P1Y5GM8vf

No need to use intrinsics for simple math

(on x64 or similar platforms)

Simple math: `+-*/` with floats and doubles.

For float and double operations the C++ Compiler generate SSE SISD code and ignores the floating point unit (x87).

(System V amd64 ABI: long double still uses x87 instructions)

But

(today) is still potential to use vectorization / (packed) SIMD operations.

History (Vectorcomputers / SIMD units)

- CDC 8600 / Cray-1 (1968/1975)
- FPU / x87 (8087, 80187, 80287, 80387. 486 & Pentium CPU incl. FPU)
- MMX: 64 bit (1997)
- SSE: 128 bit (1999 - Pentium III)
- SSE2, SSE3, SSE4.1, SSE4.2
- AVX: 256bit (2011 - Sandy Bridge)
- AVX2: (2013 - Haswell)
- AVX 512: 512 bit (2015 Xeon Phi / Knights Landing)

Vectorcomputers / Cray

- 1968: Entwicklung CDC 8600 (cancelled in 1974). First Computer with SIMD / vectorization, 64 bit Registers, 125 MHz, 16 general purpose Register instead A and B Register
- 1974: CDC Star-100: memory-to-memory vector instructions (successor: CDC 8600)
- 1975: Cray-1: register-based vector instructions: 80MHz, 8 64-bit scalar, 8 24-bit address und 8 64 x 64-bit (4096 bit) vector registers.

Use case: Simulations: Weather, Crash, Bombs, Oil (Seismic interpretation)

First custumer: NSA (hidden instructions: count 1-bit, count trailing 0-bits)

FPU / x87

FPU 8087 (1980)...80387, 486 & Pentium CPU incl. FPU

- Floating point calculation in seperate unit,
- single instruction, single data
- 10 Byte / 80 Bit floats in a "floating point stack": st(0)-st(7)
- floating-point format of 8087 is the blue print for IEEE 754 (1985)
- 80387 (1987) full compatible to IEEE 754-1985

MMX (1997)

64bit:

- 8 new 64 bit registers: MM0-MM7: packed integer data in floating point registers.
- Physical same location as st(0)-st(7)
- Single Instruction, Multiple Data for integer.
- Context switch between Floating Point operations ('87 mode) and MMX mode: 50 CPU cycles.
- 3DNow!: enhanced MMX (2 single 32 bit floats in one 64 MMX register)

128 bit

SSE (1999 - Pentium III)

- 8 new 128 bit registers (XMM0-XMM7); single instruction, multiple data for floating point. (**4 floats**, no doubles)
- Don't overlap with st(0-7) or MMX registers.
- AMD64: 8 additionlly registers (XMM8-XMM15)

SSE2: (2001 - Pentium 4):

- 16 128 bit registers (XMM0-XMM15)
- Add double precision (replace FPU) (2x double 64bit)
- Adds integer operations to SSE (replace MMX) (16x8bit, 8x16bit, 4x32bit , 2x64-bit)

128 bit

SSE3: (2004 - Pentium 4 Prescott)

Horizontal operations, more integer operations:

- fisttp zur Wandelung von Gleitkommazahlen in ganze Zahlen
- addsubps, addsubpd, movsldup, movshdup, movddup für komplexe Arithmetik
- lddqu zur Video-Kodierung
- haddps, hsubps, haddpd, hsubpd zur Unterstützung der Grafik-Aufbereitung

SSE4: (2007 - Nehalem)

- Round, mask copy, integer operations (sign, min, max)
- Compare packed strings, CRC32

256 bit

AVX (2011 - Sandy Bridge)

- 16 new 256 bit registers (YMM0-YMM15)
- same physical location as XMM0-XMM15
- single instruction, multiple data for floating point. (8 floats or 4 double)

AVX2 (2013 - Haswell)

- Adds integer operations to AVX (8,16,32, 64-bit)
- Horizontal operations
- integer operations (sign, min, max)

512 bit

AVX 512 (2015 Knights Landing, Skylake-X)

- 32 new 512 bit registers (ZMM0-ZMM31): same physical location as YMM0-YMM15
- Set of multiple extensions. No all processors implements all extensions.
- AVX-512F (foundation) SSE and AVX instructions for 512bit types, part of all extensions.
- compatibility mode allow access to YMM16-YMM31 and XMM16-XMM31 to aviod context switch (significant number of cycles)
- potential thermal issues in current implementations

ARM

NEON (arm-v8 / 32bit)

- 128 bit registers
- float and integer types

NEON (arm-v9 / 64bit)

- 128 bit registers
- float, **double** and integer types

SVE & SVE2

- Vector Length Agnostic (VLA)
- Implementer can define the length of the registers

Operating System support

We need OS (Operating System) support to use the Streaming Extensions (SSE, AVX and AVX512).

On thread switch, the OS needs to save and restore the registers. This includes the SSE/AVX registers (XMMx, YMMx, ZMMx)

Windows SSE versus SSE2

The SSE instruction set provides support only for single-precision floating-point vectors. DirectXMath must make use of the SSE2 instruction set to provide integer vector support. SSE2 is supported by all Intel processors since the introduction of the Pentium 4, all AMD K8 and later processors, and all x64-capable processors.

Note

> Windows 8 for x86 or later requires support for SSE2. All versions of Windows x64 require support for SSE2. Windows on ARM / ARM64 requires ARM_NEON.

From: https://docs.microsoft.com/en-us/windows/win32/dxmath/pg-xnamath-internals#windows-sse-versus-sse2

Conclution

Window 7 32-bit is the last Windows version running x86 without SSE2 support.

Vectorization / SIMD

Run a Single Instruction for Multiple Data.

Vector Computer from the early 70s to the 90s:

- Operate efficiently and effectively on large one-dimensional arrays of data called **vectors**.
- Special instruction can be applied on an array of data (vector).

Modern CPU operates on registers rather then memory. Smaller chunks of data is loaded in registers of specific size (**packed data**). SIMD instructions work on these registers and result can to stored back as one pack.

- load memory into (packed) register
- SIMD instructions work with registers
- store register into memory

To optimize this workflow the data needs to be aligned.

Using SSE2 (floating point add instruction)

The "Streaming Extensions" Unit of a x86 processor support basic floating point operations in 4 variants.

SSE:

data type / size	SISD (s)	SIMD / packed (p)
float / single (s)	ADDSS (1 float)	ADDPS (4 floats)
double (d)	ADDSD (1 double)	ADDPD (2 double)

AVX:

data type / size	SISD (s)	SIMD / packed (p)
float / single (s)	VADDSS (1 float)	VADDPS (8 float)
double (d)	VADDSD (1 double)	VADDPD (4 double)

Compiler generate packed vector instructions

```
constexpr int SIZE=256;
using VEC = std::array< float, SIZE>;

void add_vec_size(const VEC& left, const VEC& right, VEC& result)
{
    for (int i = 0; i < SIZE; ++i) {
        result[i] = left[i] + right[i];
    }
}
```

(GCC: -O2 -msse4 -std=c++17 -ftree-vectorize)

https://godbolt.org/z/TxGc3Tc7n

Easy to break

```
constexpr int SIZE=256;
using VEC = std::array< float, SIZE>;

void add_vec_size(const VEC& left, const VEC& right, VEC& result)
{
    for (int i = 0; i < SIZE; ++i) {
        if (right[i] > 0.f)
            result[i] = left[i] + right[i];
        else
            result[i] = left[i];
    }
}
```

(GCC: -O2 -msse4 -std=c++17 -ftree-vectorize)

https://godbolt.org/z/bjKbY5YaG

Give the compiler a hint

change code to support the compiler: use ?-operator

```
constexpr int SIZE=256;
using VEC = std::array< float, SIZE>;

void add_vec_size(const VEC& left, const VEC& right, VEC& result)
{
    for (int i = 0; i < SIZE; ++i) {
        result[i]=left[i] + (right[i] > 0.f ? right[i] : 0.f);
    }
}
```

(GCC: -O2 -msse4 -std=c++17 -ftree-vectorize)

https://godbolt.org/z/EKW9eeanq

if as intrinsics

```
void add_vec_simd(const VEC& left, const VEC& right, VEC& result)
{
    constexpr __m128 zero = { 0.f, 0.f, 0.f, 0.f };

    for (int i = 0; i < SIZE; i +=4) {
        __m128 left_pack=_mm_load_ps(&left[i]);
        __m128 right_pack=_mm_load_ps(&right[i]);

        // (right > 0.f) ? right : 0.f
        __m128 bit_mask = _mm_cmpgt_ps(right_pack, zero); // bitmask
        __m128 float_masked = _mm_and_ps(right_pack, bit_mask); // r

        // result = left + float_masked
        __m128 result_pack = _mm_add_ps(left_pack, float_masked);

        _mm_store_ps(&result[i], result_pack);
    }
}
```

https://godbolt.org/z/PnG5YfePn

`if` as intrinsics

Replace literals with 'pre-loaded' register variables.

```
    constexpr __m128 zero = { 0.f, 0.f, 0.f, 0.f };
```

`if` as intrinsics

adjust loop step

```
    for (int i = 0; i < SIZE; i +=4) {
        ...
    }
```

Ensure loop is a multiply of register size:

- **SSE** floats: 4, double: 2
- **AVX** floats: 8, double: 4

Reminders needs extra SISD code

`if` as intrinsics

load memory in register variables

```
    __m128 left_pack=_mm_load_ps(&left[i]);
    __m128 right_pack=_mm_load_ps(&right[i]);
```

check for correct alignment (see later)

`if` as intrinsics

Compare intrinsics result in bit masks in float/double variables (all bits zero (0) or one (1))

```
    // (right > 0.f) ? right : 0.f
    __m128 bit_mask = _mm_cmpgt_ps(right_pack, zero); // bitmask sto
    __m128 float_masked = _mm_and_ps(right_pack, bit_mask); // right
```

logical operations (`_mm_and_ps`, `_mm_andnot_ps`, `_mm_or_ps`, `_mm_xor_ps`) runs on bit mask and one value

(Sometimes more complex operations needed)

`if` as intrinsics

do the work

```
    // result = left + float_masked
    __m128 result_pack = _mm_add_ps(left_pack, float_masked);
```

generate ADDPS machine instruction.

`if` as intrinsics

store register variables back in memory

```
_mm_store_ps(&result[i], result_pack);
```

check for correct alignment (again)

Function calls

```
constexpr int SIZE=256;
using VEC = std::array< float, SIZE>;

void add_vec_size(const VEC& left, const VEC& right, VEC& result)
{
    for (int i = 0; i < SIZE; ++i) {
        result[i]=left[i] + std::sin(right[i]);
    }
}
```

(GCC: -O2 -msse4 -std=c++17 -ftree-vectorize)

All call pathes needs to vectorizible. There is no vectorizable version of `sin()` in the C++ Standard.

see: https://godbolt.org/z/v33T3v548

SVML

special case optimized math function (Intel SVML). e.g. `sin()`

GCC: `-mveclibabi=svml -ftree-vectorize -funsafe-math-optimizations`

see: https://godbolt.org/z/jGrv1ors3

If a vector lib is specified the optimizer may replace a set of functions with vectorized counterparts.

see:

- https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html

Use SIMD class

```cpp
#include <vectorclass.h>
#include <vectormath_trig.h>

constexpr int SIZE=256;
using VEC = std::array< float, SIZE>;

void add_vec_size(const VEC& left, const VEC& right, VEC& result)
{
    for (int i = 0; i < SIZE; i +=Vec4f::size()) {
        const Vec4f vleft=Vec4f{}.load_a(&left[i]);
        const Vec4f vright=Vec4f{}.load(&right[i]);

        const Vec4f vresult=vleft + sin(vright);
        vresult.store(result.data() + i);
    }
}
```

`vectorclass` (https://github.com/vectorclass/version2) from Agner Fog.

- Vc (https://github.com/VcDevel/Vc)
- Eve (https://github.com/jfalcou/eve, Joël Falcou)

AOS vs SOA

Working with a list of non trivial data (e.g. struct of x,y,z),

Generic way (AOS - array of structs)

```
struct Pos3d { double x, y, z; };

vector< Pos3d > dataAOS;
```

SIMD friendly layout (SOA - struct of arrays)

```
struct DataSOA {
    vector< double > x,y,z;
};
DataSOA dataSOA;
```

AOS vs SOA (Layout)

Type	0	1	2	3	4	5	6	7	8	9	...	...
AOS	X	Y	Z	X	Y	Z	X	Y	Z	...		
SOA	X	X	X	...	Y	Y	Y	...	Z	Z	Z	...

Benefits of SOA

Most efficient way to load data from memory

```
__m128 _mm_load_ps(float * p )
```

> Loads four singles/floats values from memory. The address must be 16-byte-aligned.

SOA as basic technic to write portable SIMD code.

- `__m128 _mm_load_ps(float * p )` loads 4 floats
- `__m256 _mm256_load_ps(float * p )` loads 8 floats
- `__m512 _mm512_load_ps(float * p )` loads 16 floats

Implement SOA

and compare with SOA vs AOS.

```
template< typename FLOAT>
struct Vec3d
{
    FLOAT x, y, z;
};

using VectorAOS = std::vector< Vec3d< double>>;
using VectorSOA = Vec3d< std::vector< double>>;
```

SOA vs AOS (C++)

```
void normalizeVectorAOS(VectorAOS& val)
{
    for (size_t i = 0; i < val.size(); ++i) {
        const double norm = std::sqrt(val[i].x * val[i].x + val[i].y
        const double inv_norm = 1. / norm;

        val[i].x *= inv_norm; val[i].y *= inv_norm; val[i].z *= inv_n
    }
}

void normalizeVectorSOA(VectorSOA& val)
{
    for (size_t i = 0; i < val.x.size(); ++i) {
        const double norm = std::sqrt(val.x[i] * val.x[i] + val.y[i]
        const double inv_norm = 1. / norm;

        val.x[i] *= inv_norm; val.y[i] *= inv_norm; val.z[i] *= inv_n
    };
}
```

SISD example

```
void normalizeSSE2(Vec3d<double>& val)
{
    // load
    __m128d x = { val.x };
    __m128d y = { val.y };
    __m128d z = { val.z };

    // hypot = x^2+y^2+z^2
    __m128d x2 = _mm_mul_sd(x, x);
    __m128d y2 = _mm_mul_sd(y, y);
    __m128d z2 = _mm_mul_sd(z, z);
    __m128d hypot = _mm_add_sd(_mm_add_sd(x2, y2), z2);

    __m128d norm = _mm_sqrt_sd(hypot, hypot); // norm = sqrt(hypot)
    __m128d inv_norm = _mm_div_sd(_mm_set_sd(1.), norm); // inv_norm

    val.x = _mm_mul_sd(x, inv_norm)[0];
    val.y = _mm_mul_sd(y, inv_norm)[0];
    val.z = _mm_mul_sd(z, inv_norm)[0];
}
```

SOA and SSE

```
void normalizeSSE(VectorSOA& val, size_t index)
{
    static const __m128d one = { 1., 1. };
    __m128d x = _mm_load_pd(&val.x[index]);
    __m128d y = _mm_load_pd(&val.y[index]);
    __m128d z = _mm_load_pd(&val.z[index]);

    // hypot = x^2+y^2+z^2
    __m128d x2 = _mm_mul_pd(x, x);
    __m128d y2 = _mm_mul_pd(y, y);
    __m128d z2 = _mm_mul_pd(z, z);
    __m128d hypot = _mm_add_pd(_mm_add_pd(x2, y2), z2);

    __m128d norm = _mm_sqrt_pd(hypot); // norm = sqrt(hypot)
    __m128d inv_norm = _mm_div_pd(one, norm); // inv_norm = 1.0 / nor

    _mm_store_pd(&val.x[index], _mm_mul_pd(x, inv_norm));
    _mm_store_pd(&val.y[index], _mm_mul_pd(y, inv_norm));
    _mm_store_pd(&val.z[index], _mm_mul_pd(z, inv_norm));
}
```

SOA and AVX

```
void normalizeAVX(VectorSOA& val, size_t index)
{
    static const __m256d one = { 1., 1., 1., 1. };
    __m256d x = _mm256_load_pd(&val.x[index]);
    __m256d y = _mm256_load_pd(&val.y[index]);
    __m256d z = _mm256_load_pd(&val.z[index]);

    // hypot = x^2+y^2+z^2
    __m256d x2 = _mm256_mul_pd(x, x);
    __m256d y2 = _mm256_mul_pd(y, y);
    __m256d z2 = _mm256_mul_pd(z, z);
    __m256d hypot = _mm256_add_pd(_mm256_add_pd(x2, y2), z2);

    __m256d norm = _mm256_sqrt_pd(hypot); // norm = sqrt(hypot)
    __m256d inv_norm = _mm256_div_pd(one, norm); // inv_norm = 1.0 /

    _mm256_store_pd(&val.x[index], _mm256_mul_pd(x, inv_norm));
    _mm256_store_pd(&val.y[index], _mm256_mul_pd(y, inv_norm));
    _mm256_store_pd(&val.z[index], _mm256_mul_pd(z, inv_norm));
}
```

Missing loop

```
void normalizeVectorSOA_SSE(VectorSOA& val)
{
    for (size_t i = 0; i < val.x.size(); i += 2)
    {
        normalizeSSE(val, i);
    };
}

void normalizeVectorSOA_AVX(VectorSOA& val)
{
    for (size_t i = 0; i < val.x.size(); i += 4) {
        normalizeAVX(val, i);
    };
}
```

SOA vs AOS results

(Linux 64bit, AMD Ryzen 7 PRO 5850U)

Benchmark	cpu time
AOS C++	3215 ns
SOA C++	3246 ns
SOA SSE2	1669 ns
SOA AVX2	816 ns

see: https://www.quick-bench.com/q/oYg0vyr8dq9Y1xvfU2iE9bETGA0 (without AVX)

(different results on different hardware. e.g. No improvements for AVX on Intel i5 xxxx)

Left overs

Compiler hints

- Alignment
- Pointer aliasing
- assume

Alignment

Linux and Window x64 uses the SSE model. This includes a 128bit / 16byte alignment.

```
static_assert(__STDCPP_DEFAULT_NEW_ALIGNMENT__ == 16);
static_assert(__STDCPP_DEFAULT_NEW_ALIGNMENT__ * 8 == 128);
```

see: https://godbolt.org/z/Gxdh7xvf8

AVX needs 256bit / 32 byte alignment

Alignment

- alignas (C++11)
  - alignas don't work for data on heap (new and malloc)
- `_mm_malloc` Intel (Linux)
- `aligned_alloc` (C11 / C++17)
- `_aligned_malloc` (Windows)

or use an Alignment allocator (e.g. boost)

```
#include < boost/align/aligned_allocator.hpp>

std::vector< double, boost::alignment::aligned_allocator< double, 32>
```

(only needed for AVX, AVX512, ... SSE supported on x64 Linux/Window)

std::vector

as non-aligned, un-restricted pointer

std::vector is defined as 3 pointers. Not so easy (for the compiler) to detect the non-aligned, non-aliasing pointers.

Pointer aliasing

Vectorization or the use of SIMD intrinsics require independent memory regions

Example: overlapping memory

```
float vec = new float[100];

add_vec(vec, vec+1, vec+2 , 100-2);
```

Different result different if the algorithmus runs from left to right or right to left, or if packed operations (SIMD) are used.

The results will be different, if the calculation runs left to right or otherwise, or if the packed instructions are used.

Restrict pointers

The compiler will assume function might called with aligned pointers.

Except the `__restrict` keyword is use.

```
// adds num elements from left and right and store the sums in result
void add_vec(__restrict const float* left,
             __restrict const float* right,
             __restrict float* result, size_t num);
```

The caller needs to ensure, the function is called only with independent pointers / memory regions.

assume

CompilerExplorer example (give GCC some hints to generate nice code):
https://godbolt.org/z/G3rnGGEno (accumulate like)

Missing topics

- OpenMP and VectorABI: https://sourceware.org/glibc/wiki/libmvec?action=AttachFile&do=view&target=VectorABI.txt
- compare SIMD code with GPU code
- more details on branch-less programming (for SIMD)
- different compiler support (GCC vs MSCV) ...

Summery

- intrinsics allow to write hardware dependend code in C++
  - SIMD, but also AES, CRC, ...
- vector libraries allow to write SIMD code in less hardware dependend code
- both are closly releated, but not the same.
- it's only indirect connected to performance: enable CPU features from C++ (and C, Fortran, ...)
  - (these features are usally in the CPU for better performance)