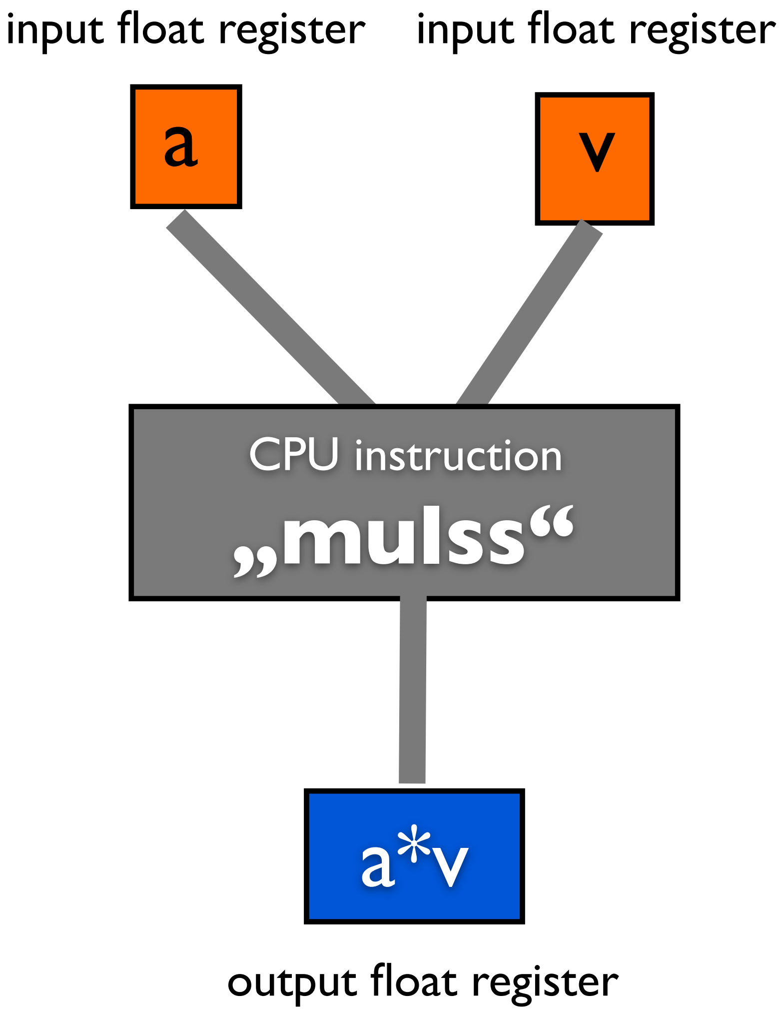
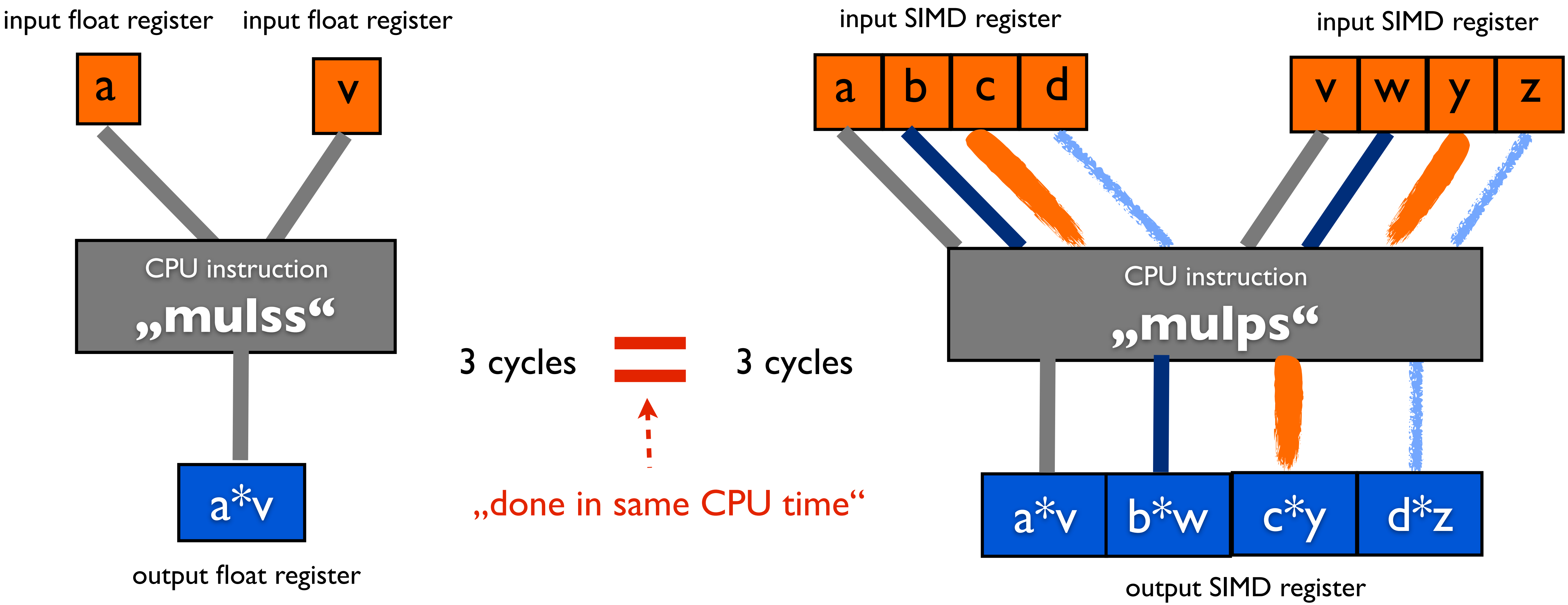
Intro to SIMD and explicit and portable SIMD vectorization techniques in C++

Conventional CPU operation



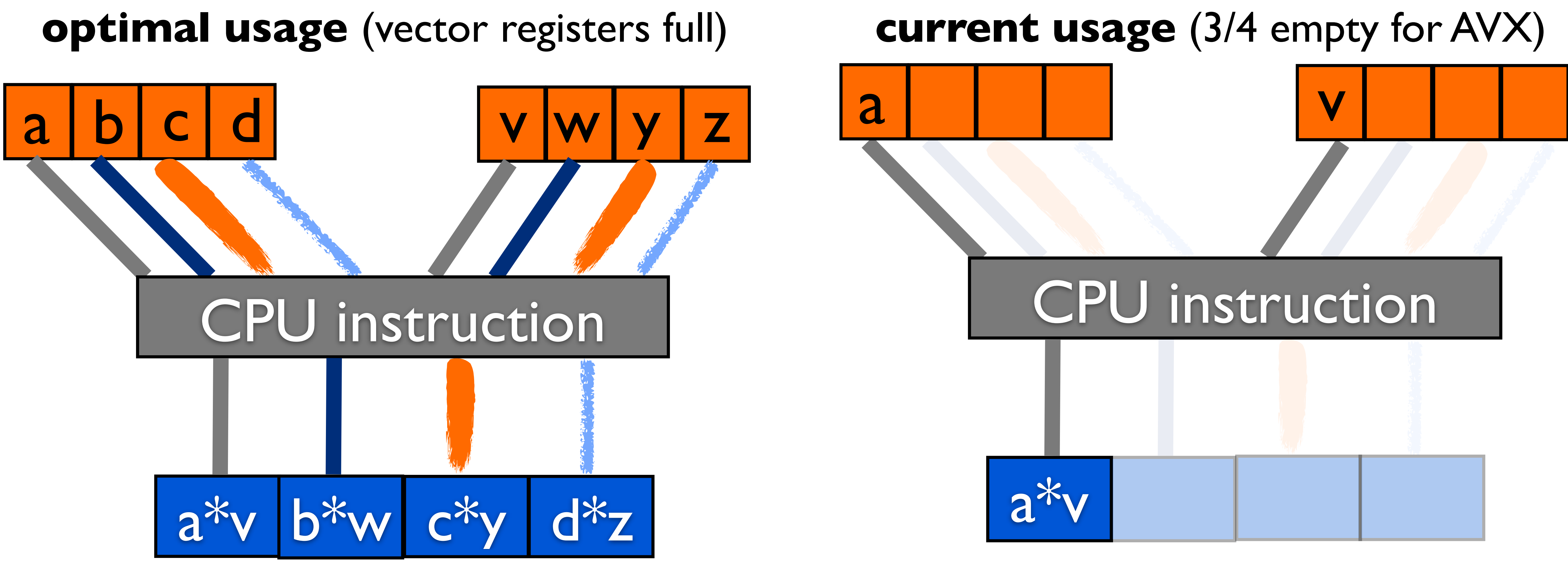
Vectored SIMD CPU operation



- CPUs can process vectored input data (a „vector“) in a single instruction = SIMD
- SIMD gives opportunity to accelerate data processing
- SIMD increasingly important: SSE (4 floats), AVX (8 floats), AVX512 (16 floats), ARM NEON
- next to multithreading, probably the most important performance dimension today ... as the pure speed of CPUs does not increase anymore

Optimal vs typical usage of CPU

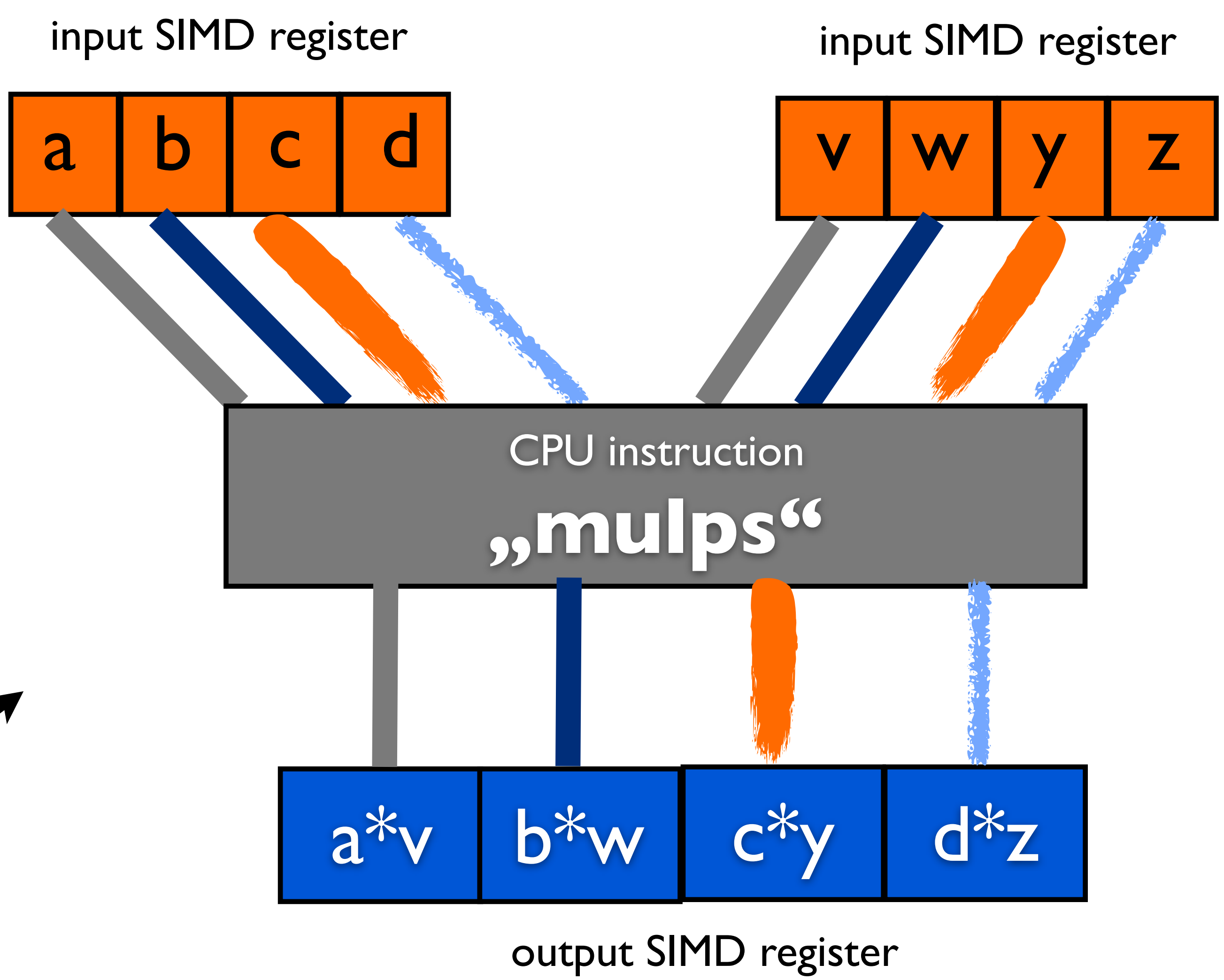
- Essentially all registers are **vector registers**
- We typically only use one slot in such a register (because we program in terms of scalar data flow)



We are not utilizing this performance dimension

Goal

„A fundamental requirement for vectorization is the availability of vectored data“

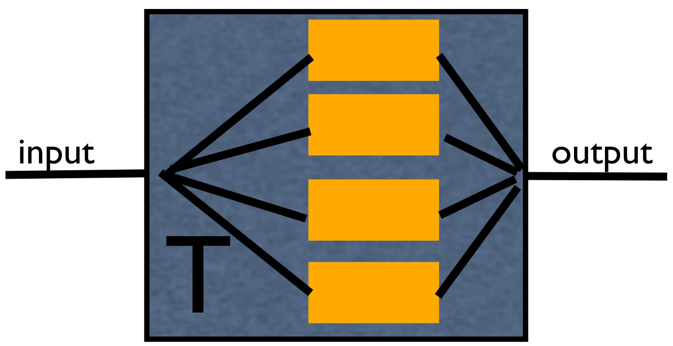


„If this is case, how can we **easily** manage to **reliably** use these instructions ?“

When can we aim for SIMD acceleration?

- Given an algorithm T ... SIMD acceleration can be applied whenever ...
  - we apply the same algorithm/ transformation to many primary input data elements (vectorize over data elements)
  - the algorithm itself consists of a repetition of similar operations (vectorize over algorithmic steps)

„easy“



„harder“

- T ...
  - color conversion for many pixels
  - gaussian convolution on many pixels
  - **smearing of many particles**
  - **3body decay for many X particles**

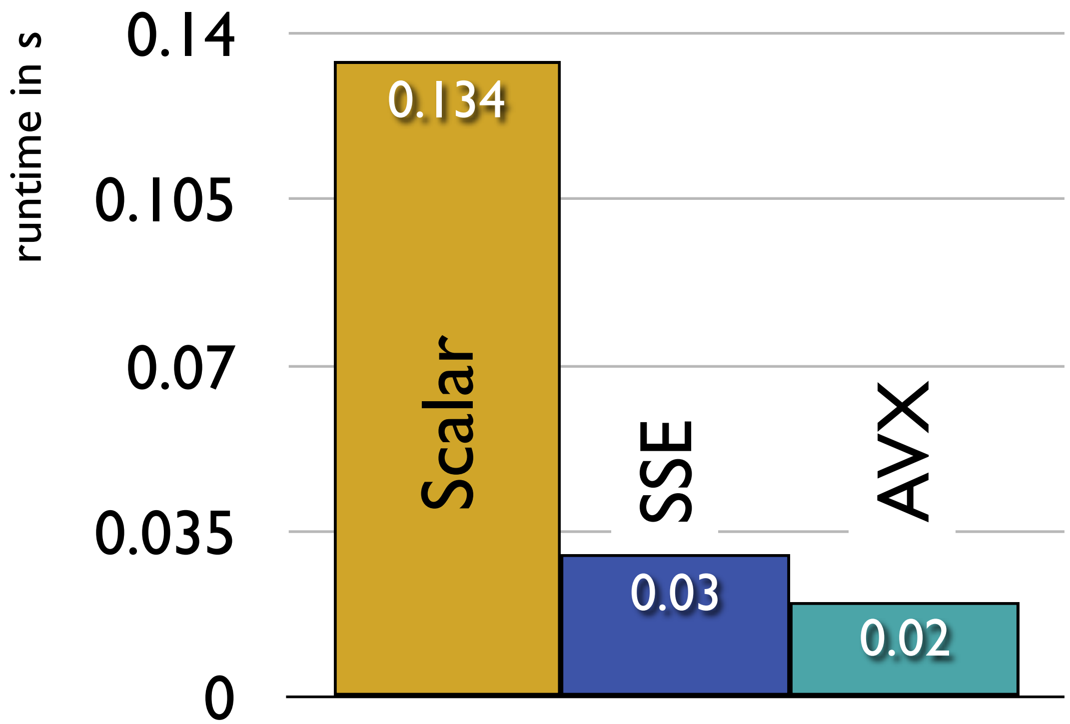
- T ...
  - some algorithm T having an inner loop
  - calculate 4-vector norm, …

- goal should be to accelerate biggest possible T of the code; should try to accelerate parts which will scale with architecture development

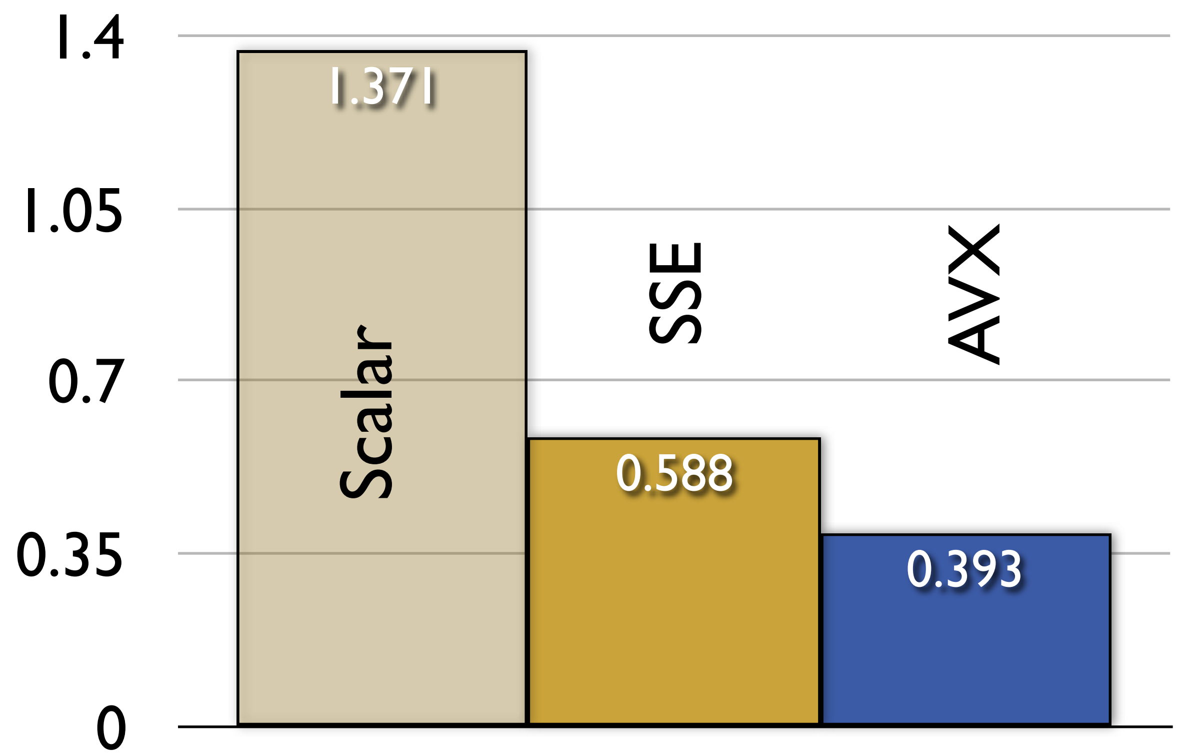
Is it relevant in practice? (image processing examples)

```
void foo(vector<float> const &inpixels, vector<float> &outpixels) {
  for (int i = 0; i < inpixels.size(); ++i) {
    outpixels[i] = T(inpixels[i]);
  }
}
```

T=exp



T=rgb2lab (color conversion)



The challenge

```
void foo(vector<float> const &inpixels, vector<float> &outpixels) {
  for (int i = 0; i < inpixels.size(); ++i) {
    outpixels[i] = T(inpixels[i]);
  }
}
```

- We have seen that is works ....
- Is it an automatic free lunch, provided by the compiler?
- **„Ideally“ yes** … but no guarantee
  - `gcc -msse4 -ftree-vectorize -O2 code.cpp` should do it
  - additional code annotations „`#pragma simd`“ may help

if the compiler is not doing it …

- rely on **algorithms** in **libraries**
  - e.g., Eigen (vectorized linear algebra operations on matrices and vectors);
  - nice but limited scope
- programming using assembly / intrinsics / compiler specific types
  - platform + compiler dependent; usually very ugly and unreadable code (many MACRO guards, compiler intrinsics)
  - hard to maintain
- high-level C++ libraries wrapping SIMD architecture
  - offering C++ types as an abstraction of the SIMD architecture
  - platform independent + compiler independent in user code
  - same or better performance than auto-vectorized code
  - the tool to develop fully SIMD vectorized yourself !

platform dependent code (possible code duplication !!)

```
#ifdef HAVE_SSE
inline float dot(const float *const buffer, const float *const kernel, int ksize) {
  int i = 0;
  float fsum = 0;

  __m128 sum = _mm_setzero_ps();
  for (; i < ksize - 3; i += 4) {
    sum = _mm_add_ps(sum, _mm_mul_ps(_mm_loadu_ps(buffer + i), _mm_loadu_ps(kernel +
i)));
  }

  sum = _mm_add_ps(sum, _mm_movehl_ps(sum, sum));
  sum = _mm_add_ss(sum, _mm_shuffle_ps(sum, sum, 0x55));
  _mm_store_ss(&fsum, sum);

  // do some tail treatment

  // and return
  return fsum;
}
#endif
```

„non comprehensible"

... to C++

```cpp
#include <Vc/Vc>
using float_v = Vc::float_v;
constexpr auto S = float_v::Size;
constexpr auto K = S - 1;

float dot_Vc(const float *const a, const float *const b, int ksize) {
  int i = 0;
  float fsum(0.f);
  float_v accum(0.f); // vector accumulator
  for (; i < ksize - K; i += S) {
    // interpret data as vector and do some operations
    accum += float_v(&a[i]) * float_v(&b[i]);
  }
  fsum = accum.sum();
  // tail correction part ...

  // return result
  return fsum;
}
```

platform independent (SSE, AVX, ARM, ...); (more) readable; fast

Intro Vc

https://github.com/VcDevel/Vc

- templated C++11 library wrapping low level intrinsics into C++ classes
  - PhD thesis Extending C++ for explicit data-parallel programming via SIMD vector types
  - main maintainer: Matthias Kretz (GSI, Darmstadt)
  - started ~2009; https://github.com/VcDevel/Vc; LGPL license
- Compilers: gcc, clang, icc, MSVC (release 0.7); Platforms: SSE, AVX, AVX512, ARM,...
- originated in High-Energy Physics (CERN, GSI, ALICE) ... now also in Industry (Nikon, Finance) ... growing community
- Vc (vector types) to be part of C++ language standard: https://en.cppreference.com/w/cpp/experimental/simd/simd

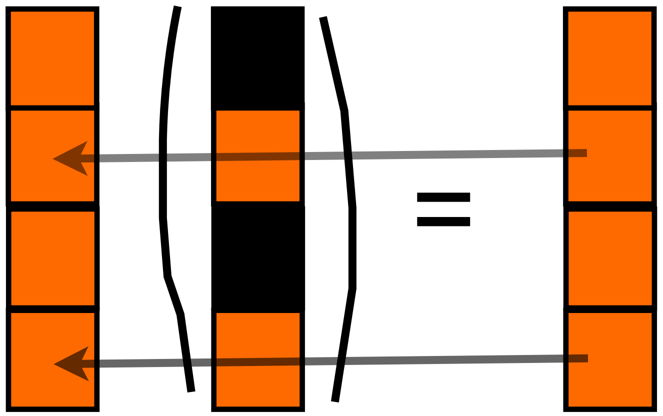
Main elements of Vc

- Vector classes
  - basic abstraction of a vector register
  - C++ operator abstractions for usual math operations
  - loads/stores to memory
  - accessors to vector components
- Mask classes
  - extension of a boolean to vector types
  - handle branching (if-statements) in vector code
- Math + Utilities
  - the usual math function operating on SIMD vectors; basically everything you find in <cmath>
- higher level containers and stl-like algorithms
  - Vc::SimdArray<T,N>, Vc::Simdize, ... not covered here

- Vector classes provide abstraction of a CPU vector (register)
  - `Vc::float_v (== Vc::Vector<float> )` ~ a | b | c | d
  - `Vc::double_v (==Vc::Vector<double>)`
  - These types automatically map to right register size depending on compiler flag
  - `Vc::float_v::Size` holds the number elements this type stores
- Loading, Storing
  - `Vc::float_v a(2.);` // from literal
  - `Vc::float_v a(&array[i]);` // construct from data array address
  - `a.store(&array[i]);` // store a to memory at address &array[i]
- Usual arithmetic operations
  - `Vc::float_v a,b,c; c = a OP b; ... ;` OP = { +,-,/,*, ...}
- Componentwise access
  - `Vc::float_v a; a[1] = 2.f; float f = a[2];`

The Mask class

- Mask class provides a vector version of „bool“ types
  - `Vc::float_v::Mask;` // a vector „boolean“ having the same number of entries as a Vc::float_v
- Comparison of vectors yields a mask
  - `Vc::float_v a,b; Vc::float_v::Mask m = a < b;`
- Vector operations can be done „masked“ (supported by CPU)
  - `Vc::float_v c,b; Vc::float_v::Mask m;`
  - `c(m) = b;` // assign b to c but only for components in which m is true
  - generalization of usual `if(m){ c=b; }`



- The usual boolean operations on masks
  - `Vc::float_v::Mask a,b,c; b = a OP b; OP = {&&, ||, !, etc.}`
- Accessors to components (like for vectors)
- horizontal queries: `if( Any(a) ){...}; if( Full(a) ){...}`

A complex example

```
void kernel1(float *a, float *b, float *c, float *res, int np) {
  for (int i = 0; i < np; ++i) {
    float d = (c[i] < 10.) ? c[i] : 2. * a[i];
    res[i] = a[i] * std::exp(d) + b[i];
  }
}
```

translating to Vc

```
using Vc::float_v;
void kernel2(float *a, float *b, float *c, float *res, int np) {
  for (int i = 0; i < np; i += float_v::Size) {
    float_v a_v(&a[i]);
    float_v c_v(&c[i]);
    auto d_v = c_v;
    auto cond = (c_v < 10.f);
    d_v(cond) = 2.f * a_v;
    auto r_v = a_v * std::exp(d_v) + float_v(&b[i]);
    r_v.store(&res[i]);
  }
}
```

`kernel1` and **`kernel2`** are doing same thing but `kernel2` is a lot faster

however, `kernel2` only correct for `np % float_v::Size == 0`

omitted consts for better readability -:)

Improved Complex Example

```
template <typename T>
T core_kernel(T a, T b, T c) {
  auto d = c;
  auto cond = c < T(10.f);
  d(cond) = T(2.f) * a;
  return a * std::exp(d) + b;
}
```

- put actual code into template kernel; then instantiate it in vector **as well as** scalar mode to fix the „tail problem"

```
using float_v = Vc::Vector<float>;                // represents an SIMD float type
using sfloat_v = Vc::Scalar::Vector<float>;       // represents a scalar float
auto constexpr S = float_v::Size;
auto constexpr K = S - 1;
void kernel3(float *a, float *b, float *c, float *res, int np) {
  int i = 0;
  // vectorizable part
  (; i < np - K; i += S) {
    core_kernel(float_v(&a[i]), float_v(&b[i]), float_v(&c[i])).store(&res[i]);
  }

  // tail part
  for (; i < np; ++i) {
    core_kernel(sfloat_v(&a[i]), sfloat_v(&b[i]), sfloat_v(&c[i])).store(&res[i]);
  }
}
```

Try to compile the last example (kernel1 and kernel2) and verify with valgrind that kernel2 is using SIMD instructions !

examples/Vc/complexKernel

example/Vc/complexKernel_googlebench

No free lunch

- Transforming your code to fully use SIMD will be hard
- Typically one arrives only at transforming certain parts of the code
  - but this could be your hotspot
- You likely need to reorganize your data flow and data layout
  - need to pass around vectors or containers of data (pass many events from one algorithm to the next likely better than passing single events)
  - need functions (T) working on vector input
  - organize data into columnar data format (SoA) better than AoS
- However, restructuring your code for SIMD is almost same as restructuring for GPU !
- Come to PowerWeek 2, where this will be done !