

SIMD and data structures for efficient reconstruction algorithms

Arthur Hennequin – CERN Ph.D. student (LIP6 and LPNHE, Sorbonne University)
arthur.hennequin@cern.ch



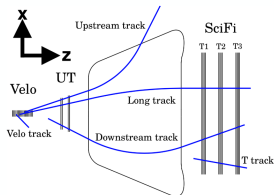
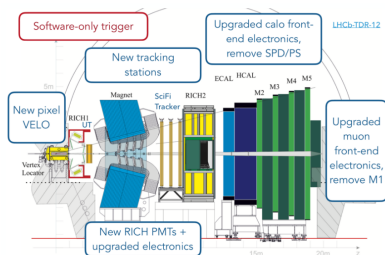
Introduction

The goal: HLT1 dataflow above 30 kHz per node

Best physics possible → no cuts

What is in this talk ?

- Some context...
- How to **understand** and use SIMD
- How to use SoA and PoD to speed up
- Some examples...
- Some results...



What is SIMD ?

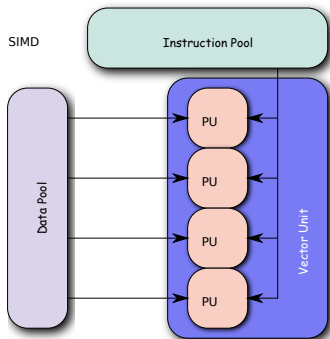
Single Instruction Multiple Data

Instruction set extensions:

- x86: SSE (128 bits), AVX2 (256 bits), AVX512 (512 bits)
- ARM: Neon (128 bits), SVE (128-2048 bits)
- PowerPC: AltiVec (128 bits)

Main types of instructions:

- Memory access (load, store, gather, scatter)
- Arithmetic and logic computation
- Data exchange (shuffle, permute, compress...)

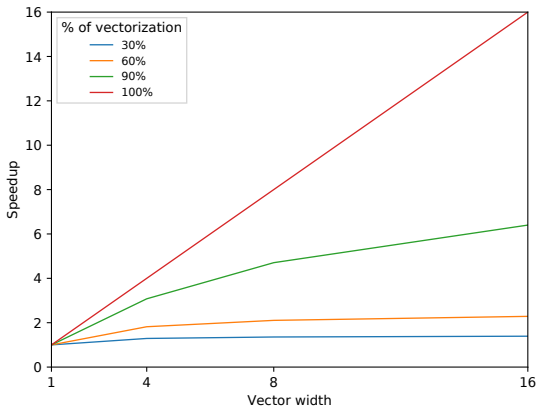


How much can we gain from SIMD ?

Amdahl's law: theoretical model, expressing the maximum speedup achieved by a parallel machine.

$$\text{speedup}(w) = \frac{1}{1 - p + \frac{p}{w}}$$

- p: percentage of SIMD code
- w: size of SIMD register



How much can we lose from SIMD ?

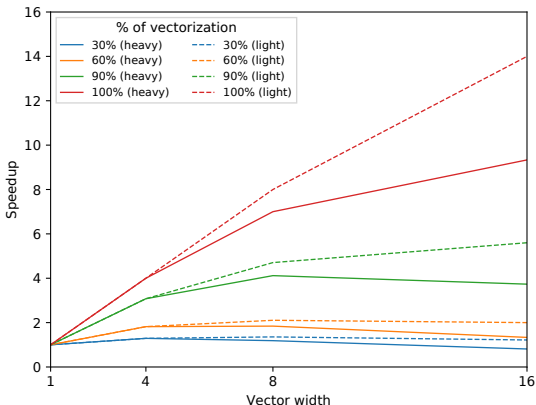
Amdahl's law with frequency scaling correction:

$$\text{speedup}(w) = \frac{1}{1-\rho + \frac{\rho}{w}} \times \frac{\text{freq}(w)}{\text{freq}(1)}$$

Intel Xeon Silver 4114

frequencies (GHz):

	Base	Turbo
Non-AVX	2.2	2.5
Heavy AVX2	1.8	2.2
Heavy AVX-512	1.1	1.4



Advices for SIMD

- Know your instruction set:
<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>
- Know (and do not trust) your compiler: <http://godbolt.org/>
- Know your algorithm: analyse data dependencies, function latency and throughput, arithmetic intensity (ratio memory access/computation)
- Always compare your throughput with previous build
- Avoid loading, storing and shuffling registers as much as you can
- Keep your algorithm design simple (this helps with other points)
- Identify and reuse optimized patterns
- Inline your functions as much as possible
- Avoid scatter / gather

→ Use SoA...

What is SoA ?

Memory layouts:

- AoS: Array of struct → What we used to have ie. `std::vector<Track>`
- SoA: Struct of array → What is needed for SIMDzation
- AoSoA: Array of struct of array → Hybride layout sometime used for SIMDzation

Conceptual Layout

Physical Memory Layout

`x y w`

...

`x y w`

...

`x y w`



Array-of-Structs

`x y w x y w x y w x y w x y w x y w x y w x y w x y w ... x y w`

Struct-of-Arrays

`x x x x x x x ... x y y y y y y ... y w w w w w w ... w`

Array-of-Structs-of-Arrays

`x x x x y y y y w w w w ... x x x x y y y y w w w w`

SIMD Width

Containers with SoA interface

Containers with internal SoA storage in a single buffer (important for limiting memory allocations)

Fields can be float or int or float[N]...[M] or int[N]...[M] if dimensions N to M are known at compile time.

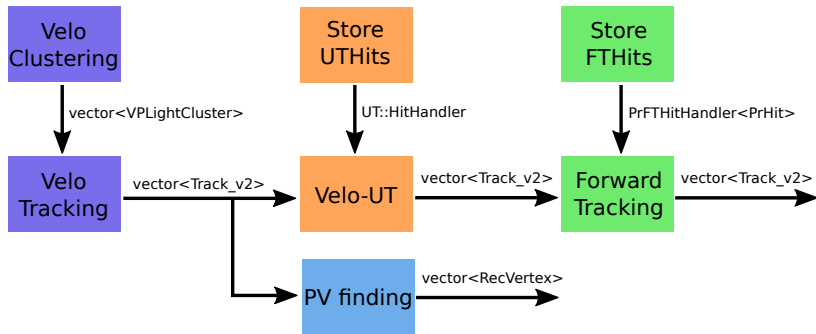
Each field has 5 templated accessors:

- T load(int key, ...) const;
- T gather(I key, ...) const;
- T masked_gather(I key, ..., M mask) const;
- T store(int key, ..., T value);
- T compress_store(int key, ..., M mask, T value);

... stands for other dimensions indexes.

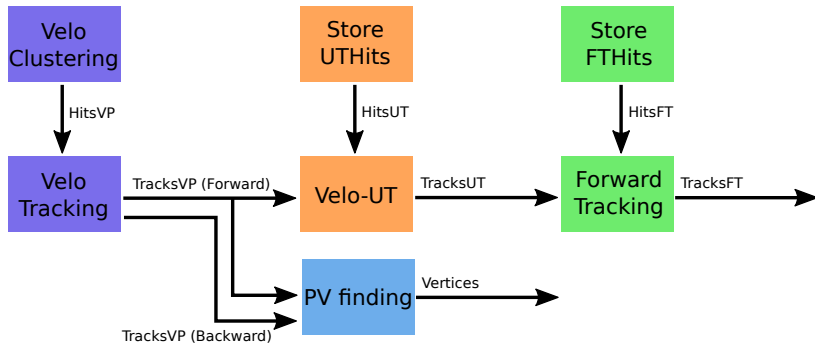
Upgrade HLT1 (Some weeks ago...)

Large subset of HLT1 algorithms (without fit, muon id...)



Upgrade HLT1 (Current status)

Some changes in data structures (AoS \rightarrow SoA):



Example: HitsVP (I)

```
class HitsVP {
    constexpr static int max_hits = align_size(10000);
public:
    HitsVP() { // Constructor
        const size_t size = max_hits * 4 * sizeof(int);
        m_data = static_cast<int*>(std::aligned_alloc(64, size));
    }

    HitsVP(const HitsVP&) = delete; // No Copy Constructor

    HitsVP(HitsVP&& other) { // Move Constructor
        m_data = other.m_data;
        other.m_data = nullptr;
        m_size = other.m_size;
    }

    inline int size() const { return m_size; }
    inline int& size() { return m_size; }
```

Example: HitsVP (II)

```
VEC3_SOA_ACCESSOR(pos,  
                  (float*)m_data,  
                  (float*)&m_data[max_hits],  
                  (float*)&m_data[2*max_hits])  
  
SOA_ACCESSOR(LHCbId, &m_data[3*max_hits])  
  
~HitsVP() {  
    std::free(m_data);  
}  
  
private:  
    alignas(64) int* m_data;  
    int m_size = 0;  
};
```

SIMD wrappers

Why do we need a custom SIMD Wrapper ?

- Need to be able to simply add missing instructions / concepts
- Need to be able to simply add new architectures (AVX256, AVX512, SVE)
- Need to be simple to understand (not a blackbox)

1 header file, < 600 lines of code → already 3 backends:

- scalar (size=1)
- avx2 (size=8)
- avx512 (size=16)

You can choose per function which wrapper you want to use, if not available it will fallback to the first available one (avx512 fallback to avx2 and avx2 fallback to scalar)

Simple filter example

```
// Defines which wrapper to use:
using simd = SIMDWrapper::avx512::types;
using F = simd::float_v;

// Loop over tracks simd::size at a time
for (int t=0 ; t<tracks.size() ; t+=simd::size) {
    // Get a loop mask, true when t+idx < tracks.size()
    auto loop_mask = simd::loop_mask(t, tracks.size());

    // Load fields, do some computation and test the result:
    auto pt = tracks.pt<F>(t) * GeV;
    auto mask = pt > 0.4f && loop_mask;

    // Helper to copy all fields and increment
    // the number of output tracks:
    out.copy_back<simd>(tracks, t, mask);
}
```

Maths on wrappers

Some common maths functions are defined on all wrapper types. Templated when possible to have only 1 implementation for all backend.

Vec3<T>:

- 3 components vector (x, y, z)
- +, -, *, cross, dot, mag2, perp2, mag, rho, theta, phi, eta
- works on any type
- custom SoA container accessors (5 of them)

Idea: implement as you go, need a feature ? just add it or ask for it.

Try to factorize code as much as possible. Don't create too complex proxy types to maximize re-usability.

IP filter example

```
for (int i=0 ; i<tracks.size() ; i+=simd::size) {
    auto loop_mask = simd::loop_mask( i, tracks.size() );

    Vec3<F> B = tracks.statePos<F>( i, CLOSEST_TO_BEAM );
    Vec3<F> u = tracks.stateDir<F>( i, CLOSEST_TO_BEAM );

    F min_d = 10e3;
    for (int j=0 ; j<(int) vertices.size() ; j++) {
        auto PV = vertices[j].position();
        Vec3<F> A = Vec3<F>( PV.x(), PV.y(), PV.z() );
        auto d = (B - A).cross(u).mag2();
        min_d = min( min_d, d );
    }

    auto d = sqrt( min_d / u.mag2() );
    auto mask = ip_cut_value < d;

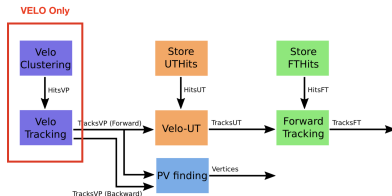
    tracks_out.copy_back<simd>( tracks, i, mask && loop_mask );
}
```


"Velo only" Results

Scenario: Velo Clustering + Velo Tracking

- Baseline \rightarrow 38 kHz ($\times 1$)
- With SoA \rightarrow 40 kHz ($\times 1.05$)
- With SIMD \rightarrow 47 kHz ($\times 1.23$)
- With SIMD and SoA/PoD \rightarrow 58 kHz ($\times 1.52$)

Non-Linear improvement by combining optimizations. Development of this algorithm benefited from some exchange of ideas with the Allen team.

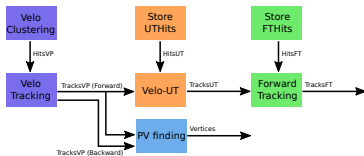


Overall HLT1 Results

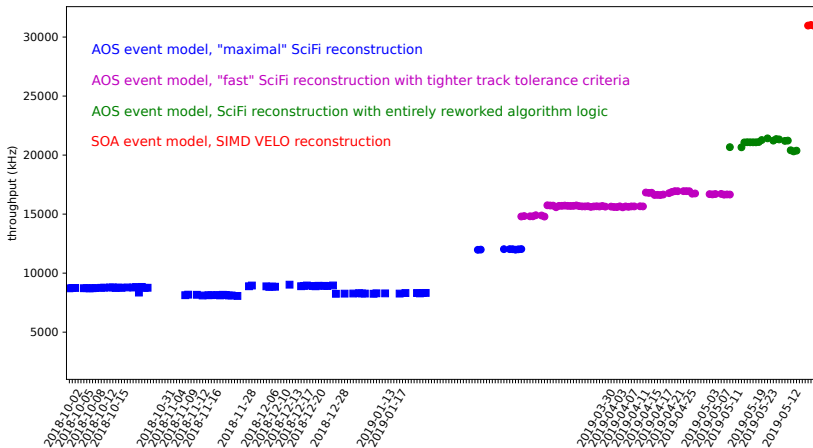
Scenario: VP + PV + UT + FT

- Baseline → 13.8 kHz ($\times 1$)
- With Velo SIMD and new FT → 24.7 kHz ($\times 1.78$)
- With Velo SIMD, new FT and SoA/PoD → 31 kHz ($\times 2.24$)

SciFiTrackForwarding: new FT algorithm from Christoph, Jiangqiao, Michel, Sascha
→ physics optimisation



A brief history of throughput



Takeaways

- Learn about instruction latency and throughput
- Verify that your compiler does what you intended
- Use SoA to avoid gathers and scatters
- Use SoA / PoD to avoid allocation overheads
- Always measure the throughput before/after a change
- Understand why your change makes things better (or not)
- Use simple designs, don't hide complexity behind layers of interfaces
- Avoid using too many aliases, keep your classes names short and descriptive

Contributions

Many thanks to everyone who contributed to the developments, provided some throughput results and participated to the discussions:

C. Hasse, S. Ponce, P. Seyfert, P. Kardos, O. Lupton, M. Szymanski, M. Clemencic, M. Cattaneo, R. Quagliani, S. Stahl, V. Coco, V. Gligorov