Fast numerics, small changes

The minimal playbook

Marco Barbone
Flatiron Institute

October 17, 2025

Link

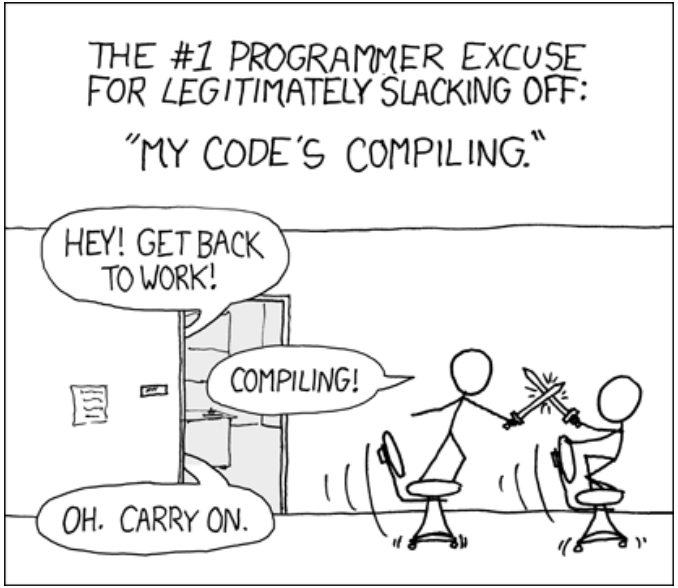
https://github.com/DiamonDinoia/fwam2025

~~How to shorten coffee breaks~~

The minimal playbook

Marco Barbone
Flatiron Institute

October 17, 2025


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."
HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

What is SIMD?

Single Instruction, Multiple Data: one instruction applies to multiple elements in parallel.

Scalar vs SIMD

Scalar

```
for i in 0 : 4:
    c_i = a_i + b_i
```

Vector

a_0	a_1	a_2	a_3
b_0	b_1	b_2	b_3
c_0	c_1	c_2	c_3

$+$

$=$

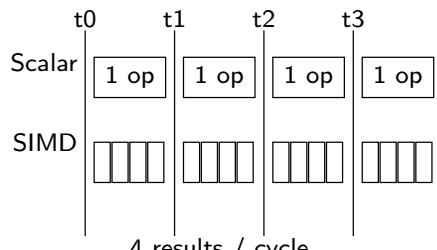
Vector width

$$128\text{b} = 2 \times 64$$

$$256\text{b} = 4 \times 64$$

$$512\text{b} = 8 \times 64$$

Throughput intuition



When SIMD can be used?

FLATIRON INSTITUTE

- **Independent operations** Each iteration or element must be computed without depending on previous results. *Example:* elementwise addition, scaling, normalization.
- **Uniform control flow** All lanes execute the same instruction sequence. Branching (`if/else`) should be minimal or identical across elements.
- **Aligned and contiguous memory access** Data should be stored in consecutive memory addresses for efficient loading/storing. Use arrays or structs-of-arrays layout.
- **Sufficient data parallelism** The problem size should be larger than or equal to the SIMD width (e.g., at least 4, 8, or 16 elements depending on 128b/256b/512b).
- **Identical operation type** All lanes perform the same arithmetic or logic operation (e.g., all do addition or multiplication).
- **No data hazards** No read-after-write (RAW), write-after-read (WAR), or write-after-write (WAW) conflicts within the vectorized section.

When SIMD is worth?

FLATIRON INSTITUTE

- **High arithmetic intensity** Computation per memory access is large enough to hide load/store overhead.
- **Large data sets** Loops with many iterations or large arrays where setup cost is amortized.
- **Regular memory access patterns** Contiguous or strided access that fits SIMD load/store instructions efficiently.
- **Tight inner loops** Performance-critical sections executed millions of times—SIMD pays off most here.
- **Uniform operations across data** Same operation repeated (e.g., vector addition, dot products, FFT kernels).
- **Hardware support available** Wider vector units (e.g., AVX2/AVX-512) and multiple execution ports increase benefit.

Goal: Transform scalar loops into SIMD code

We will use C++ and xsimd.

Step 0: Choose the right algorithm

FLATIRON INSTITUTE

Chebyshev polynomials of the first kind, $T_n(x)$

Recurrence relations for $T_n(x)$ – O(3N)

$$T_0(x) = 1, \qquad T_1(x) = x, \qquad T_{n+1}(x) = 2x\,T_n(x) - T_{n-1}(x)$$

Sequential dependency between iterations but lowest algorimic complexity

Barycentric interpolation – O(5N+1)

$$p(x) = \frac{\displaystyle\sum_{j=0}^{n} \frac{w_j f_j}{x - x_j}}{\displaystyle\sum_{j=0}^{n} \frac{w_j}{x - x_j}}, \quad x \neq x_j$$

Iterations are independent but higher arithmetic cost.

Step 1: define SIMD width

FLATIRON INSTITUTE

```
1 using batch = xsimd::batch<double>;
2 constexpr int64_t simd_width = batch::size; // lanes
```

Step 2: Split the loop

1. **Calculate the SIMD loop boundary**
   - Determine how many elements can be processed in full SIMD-width blocks.
   - Define **n_simd** is the largest multiple of the vector width $\leq$ 'N'.
2. **Split the loop body**
   - first loop for 'i $<$ n_simd'
     NOTE: Increment by simd_width
   - Scalar cleanup for remaining elements is basically the old loop with only the index changed from n_simd to N

Step 2: Split the loop

FLATIRON INSTITUTE

1. **Calculate the SIMD loop boundary**
   - Determine how many elements can be processed in full SIMD-width blocks.

```
const auto n_simd = (N / simd_width) * simd_width;
```

   - Ensures that ‘n_simd‘ is the largest multiple of the vector width $\leq$ N.
2. **Split the loop body**
   - Vectorized section for ‘i $<$ n_simd‘
   - Scalar cleanup for remaining elements is basically the old loop with only the index changed

```
// vectorized loop
for (int i = 0; i < n_simd; i += simd_width) {
}
// scalar remainder
for (int i = n_simd; i < N; ++i) {
    // old body unchanged
}
```

Step 2: Split the loop

FLATIRON INSTITUTE

```
const auto n_simd = N & (-simd_width); // round down to multiple of simd_width
std::size_t i = 0;
for (; i < n_simd; i += simd_width) {
// we will populate this
}

double num = 0;
double den = 0;

for (; i < N; ++i) {
    double diff = pt - x[i];
    double q = w[i] / diff;
    num += q * fvals[i];
    den += q;
}
return num / den;
```

Step 3: loading data in register

```
// it takes a pointer in input!
// it returns a simd register
auto bdata = xsimd::load_unaligned(data* ptr);

// alternatively xsimd::batch can be initialized with a variable:
batch bnum(0); // this sets all the elements of the register to 0

// HINT: in c++, std constainers usually have a data() function that returns a pointer
// i.e. x.data()
```

Step 3: loading data in register

FLATIRON INSTITUTE

```cpp
// it takes a pointer in input!
// it returns a simd register
auto bdata = xsimd::load_unaligned(data* ptr);

// alternatively xsimd::batch can be initialized with a variable:
batch bnum(0); // this sets all the elements of the register to 0

// HINT: in c++, std constainers usually have a data() function that returns a pointer
// i.e. x.data()
// load_unaligned(x[i]) will not work!
```

Step 3: loading data in register

FLATIRON INSTITUTE

```
1 const auto bx = batch::load_unaligned(x.data() + i);
2 const auto bw = batch::load_unaligned(w.data() + i);
3 const auto bf = batch::load_unaligned(fvals.data() + i);
```

Step 4: vectorizing the loop

FLATIRON INSTITUTE

`xsimd::batch` implements operator overloads, arithmetic operators +, -, *, and / and functions `xsimd::fma`, `xsimd::sqrt`, `xsimd::abs`, etc.) can be used directly on SIMD batches just like scalar values.
Now you have everything needed to vectorize the loop!

Step 5: Understanding partial SIMD results

FLATIRON INSTITUTE

```
const batch bpt(pt);
batch bnum(0);
batch bden(0);

std::size_t i = 0;
for (; i < n_simd; i += simd_width) {
    const auto bx = batch::load_unaligned(x.data() + i);
    const auto bw = batch::load_unaligned(w.data() + i);
    const auto bf = batch::load_unaligned(fvals.data() + i);

    const auto bdiff = bpt - bx;
    const auto bq = bw / bdiff;
    bnum += bq * bf;
    // bnum = xsimd::fma(bq, bf, bnum);
    bden += bq;
}
```

Step 5: Understanding partial SIMD results

FLATIRON INSTITUTE

By vectorizing the loop, each SIMD register now holds **partial reductions** over different strided subsets of the data. For example:

$$b_{\text{num}}[0] = \sum_{i=0,8,16,\ldots} q_i f_i, \quad b_{\text{num}}[1] = \sum_{i=1,9,17,\ldots} q_i f_i, \quad \text{and so on,}$$

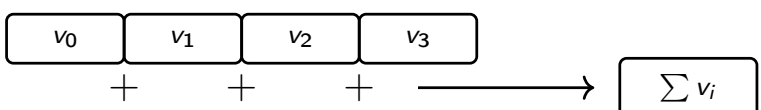
$$b_{\text{den}}[0] = \sum_{i=0,8,16,\ldots} q_i, \quad b_{\text{den}}[1] = \sum_{i=1,9,17,\ldots} q_i, \quad \text{and so on.}$$

Step 5: Horizontal sum

The **horizontal reduction** aggregates partial SIMD results into a scalar. Each lane in the batch contributes to the final sum.

$$\text{reduce\_add}([v_0, v_1, \ldots, v_{W-1}]) = \sum_{k=0}^{W-1} v_k$$

```
// Collapse SIMD register into a scalar value
const auto res = xsimd::reduce_add(data);
```



This step finalizes the SIMD computation by combining all per-lane partial results. Use other reductions like `reduce_max`, `reduce_min`, or `reduce_mul` for different aggregation operations.

Step 5: Horizontal sum

FLATIRON INSTITUTE

```
1 double num = xsimd::reduce_add(bnum);
2 double den = xsimd::reduce_add(bden);
```

Solution

FLATIRON INSTITUTE

```
double operator()(const double pt) const {
        for (int i = 0; i < N; ++i) {
            if (pt == x[i]) { return fvals[i]; }
        }
        // // shorthand for the xsimd type
        using batch = xsimd::batch<double>;
        // simd width since it is architecture/compile flags dependent
        constexpr std::int64_t simd_width = batch::size;

        const auto n_simd = N & (-simd_width); // round down to multiple of simd_width

        const batch bpt(pt);
        batch bnum(0);
        batch bden(0);
        ...
    }
```

Solution

FLATIRON INSTITUTE

```
double operator()(const double pt) const {
        ...
        std::size_t i = 0;
        for (; i < n_simd; i += simd_width) {
            const auto bx = batch::load_unaligned(x.data() + i);
            const auto bw = batch::load_unaligned(w.data() + i);
            const auto bf = batch::load_unaligned(fvals.data() + i);

            const auto bdiff = bpt - bx;
            const auto bq = bw / bdiff;
            bnum += bq * bf;
            // bnum = xsimd::fma(bq, bf, bnum);
            bden += bq;
        }

        double num = xsimd::reduce_add(bnum);
        double den = xsimd::reduce_add(bden);
        ...
    }
```

Solution

FLATIRON INSTITUTE

```
double operator()(const double pt) const {
        ...
        for (; i < N; ++i) {
            double diff = pt - x[i];
            double q = w[i] / diff;
            num += q * fvals[i];
            den += q;
        }
        return num / den;
    }
```