

Highway: portable vector intrinsics

Jan Wassenberg
Software engineer

2024-09-19

Agenda

- 01 What, why, where, how
- 02 Porting code
- 03 Runtime dispatch
- 04 Design rationale
- 05 Users

01

What, why, where,
how

What is SIMD?

Single / same
Instruction / operation on
Multiple
Data (lanes)

16-32 fused multiply-add / cycle per core!



Why SIMD?

"SIMD feels like magic!"

[Googler who saw a 10x speedup in a day's work]

- Widely available
 - x86, ARM, RISC-V, MIPS, ...
- Minimal incidental complexity
 - Same toolchain, memory space, debugger
 - No PCI-e latency
- Vendor-independent



Where SIMD?

Anything involving kGCU and **arrays** of built-in types!

- Linear algebra (ML), image/video, audio samples
- Strings (strlen, [JSON](#), [CSV parsing](#), [UTF validation](#))
- Hashing, Cryptography
- Database (bit packing, filter, join, vector search)
- QuickSelect (“Fast Top-K in ScaM”), [QuickSort](#)
- Computational biology
- Computer graphics



Why Highway?

Same code, multiple platforms

Easy to port from existing intrinsics

Helps work around compiler bugs

Reliable and predictable performance

Also designed for variable vectors (SVE)



Why not autovec?

OpenMP 4.0, armclang, Intel compiler

Minimal code changes, scalable

Brittle (maintenance, compiler upgrade)

Risk of **poor codegen** ("SIMD" memcpy)

```
movzx    ecx, byte [rax+rdi*4+8]
movd     xmm1, ecx
pinsrb   xmm1, [rax+rdi*4+0 ], 1
pinsrb   xmm1, [rax+rdi*4+12], 2
pinsrb   xmm1, [rax+rdi*4+4 ], 3
```



Why not assembly?

Used in FFMPEG

Potentially more efficient

Error-prone: major penalty for

MOVAPS xmm0, xmm1 vs.

VMOVAPS xmm0, xmm1

Laborious (though macros help)

- Porting: FMLA vs. vfmadd132ps, ...
- Manual register allocation
- Beware ABI differences



Why not intrinsics?

Widely used, also on MSVC

Error-prone

Compiler bugs (see next slide)

Laborious

- Porting: `_mm512_mask_mov_ps`
vs. `_mm256_blendv_ps`
- Verbose: `_mm256_load_si256(
 reinterpret_cast<
 const __m256i*>(ptr))`



Compiler bugs

clang-6: incorrect codegen for partial vector writes. **Workaround:** memcpy instead of intrinsics

clang-6: incorrect ARMv7 codegen, read after write data hazard. **Workaround:** clobber memory

clang-6: suboptimal codegen for VBROADCASTI128. **Workaround:** inline assembly

clang-6: missing KORTTEST for AVX-512. **Workaround:** treat masks as integers

clang-6: incorrect msan codegen. **Abandoned:** require clang-7

clang-7: unaligned spills in asan. **User workaround:** shorter variable lifetime

clang-8: various "Do not know how to split". **Workaround:** find op, replace with other

clang-8: inconsistent inlining/attribute requirements. **Workaround:** use pragma

clang-8: pragma must be at global scope. **Workaround:** HWY_BEFORE_NAMESPACE

clang-9: crash due to vector class constructor. **Workaround:** aggregate init

gcc 9.2: incorrect intrinsics for signed compare. **Workaround:** vector extension

Wasm: **Workaround:** emulate missing/broken instructions



Highway library

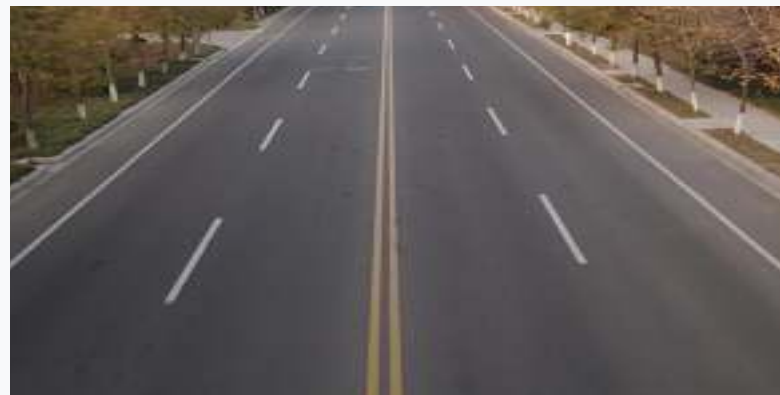
<https://github.com/google/highway>

[Example](#)

Developed since 2017, open sourced 2019

Advice:

Connor Fitzgerald, Daniel Lemire
Jyrki Alakuijala, Povilas Kanapickas
Rich Winterton



Under the hood

```
template <typename T>
HWY_API Vec256<T> IfVecThenElse(
    Vec256<T> mask, Vec256<T> yes, Vec256<T> no) {
  #if HWY_TARGET <= HWY_AVX3
    const DFromV<decltype(yes)> d;
    const RebindToUnsigned<decltype(d)> du;
    using VU = VFromD<decltype(du)>;
    return BitCast(d, VU{_mm256_ternarylogic_epi64(
        BitCast(du, mask).raw, BitCast(du, yes).raw,
        BitCast(du, no).raw, 0xCA)}));
  #else
    return IfThenElse(MaskFromVec(mask), yes, no);
  #endif
}
```



Your code

```
void Squared(const float* in, float* out, size_t num) {  
    ScalableTag<float> d; // asks for full vector  
  
    for (size_t i = 0; i < num; i += Lanes(d)) {  
        const auto vec = LoadU(d, in + i);  
        StoreU(Mul(vec, vec), d, out + i);  
    }  
    printf("F(x)->x^2, F(%.0f) = %.1f\n", in[2], out[2]);  
}
```



Example: RNG

```
class Xorshift128Plus {  
public:  
    // 8 independent generators  
    // (= single iteration for AVX-512)  
    enum { N = 8 };  
  
    HWY_INLINE HWY_MAYBE_UNUSED void  
    Fill(uint64_t* HWY_RESTRICT random_bits) {  
        // see next slide  
    }  
  
    // unsafe, requires class to be aligned  
    HWY_ALIGN uint64_t s0_[N];  
    HWY_ALIGN uint64_t s1_[N];  
};
```

Porting RNG

```
for (size_t i = 0; i < N; ++i) {  
    auto s1 = s0_[i];  
    const auto s0 = s1_[i];  
    const auto bits = s1 + s0;  
    s0_[i] = s0;  
    s1 ^= s1 << 23;  
    random_bits[i] = bits;  
    s1 ^= s0 ^ (s1 >> 18) ^  
        (s0 >> 5);  
    s1_[i] = s1;  
}
```

```
const ScalableTag<uint64_t> d; // assume <= 512bit  
for (size_t i = 0; i < N; i += Lanes(d)) {  
    auto s1 = Load(d, s0_ + i);  
    const auto s0 = Load(d, s1_ + i);  
    const auto bits = s1 + s0;  
    Store(s0, d, s0_ + i);  
    s1 ^= ShiftLeft<23>(s1);  
    Store(bits, d, random_bits + i);  
    s1 ^= s0 ^ ShiftRight<18>(s1) ^  
        ShiftRight<5>(s0);  
    Store(s1, d, s1_ + i);  
}
```


02

Porting code

Annotate

AVX2: Clang/GCC require -mavx2 (unsafe) or function attribute

Could annotate each function with HWY_ATTR

Easy to forget, causes errors on other compilers

Or: single #pragma attribute/target (HWY_BEFORE_NAMESPACE)

Convenient

Compiler-specific

Must be outside namespace (thanks Robert Obryk!)

```
HWY_BEFORE_NAMESPACE();
namespace myproject {
namespace HWY_NAMESPACE {
HWY_ATTR void MyFunc(float* HWY_RESTRICT out) {}
} // namespace HWY_NAMESPACE
} // namespace myproject
HWY_AFTER_NAMESPACE();
```

Create vectors

```
// Defined by Highway:
template <typename Lane, size_t kLanes>
struct Simd { // Empty tag type
    using T = Lane;
};
Type128 Zero(Simd<float, 4> /*tag*/);
Type256 Zero(Simd<float, 8> /*tag*/);

// Your code
const ScalableTag<float> d; // = Simd<float, ??>
const auto zero = Zero(d);
const auto one = Set(d, 1.0f);
```

Loops, memory

```
for (size_t x = 0; x < xsize;
-                                     ++x) {
+                                     x += Lanes(d)) {
-   const float xval = rowx[x];
-   const float yval = rowy[x];
+   const auto xval = Load(d, rowx + x);
+   const auto yval = Load(d, rowy + x);

-   const float scaler = s + (yw * (1.0f - s)) /
-                               (yw + yval * yval);
+   const auto scaler = s + (yw * (one - s)) /
+                               MulAdd(yval, yval, yw);

-   rownew[x] = scaler * xval;
+   Store(scaler * xval, d, rownew + x);
}
```

Alignment

```
std::vector<float> rowx(128);
for (size_t x = 0; x < xsize; x += Lanes(d)) {
    // CRASH - unaligned
    const auto xval = Load(d, rowx.data() + x);
    // ...
}

// less efficient
const auto xval = LoadU(d, rowx.data() + x);

// unsafe for member variables and large vectors
HWY_ALIGN float rowx[128];

// works for member variables and large vectors
hwy::AlignedFreeUniquePtr<float[]> rowx =
hwy::AllocateAligned<float>(128);
```

Data layout

```
struct Point {  
    float x;  
    float y;  
};  
hwy::AlignedFreeUniquePtr<Point[]> points =  
hwy::AllocateAligned<Point>(N);  
  
const ScalableTag<float> d;  
  
// mixes x and y in vector  
auto mixed = Load(d, &points.data().x);  
  
hwy::AlignedFreeUniquePtr<float[]> all_x_then_y =  
hwy::AllocateAligned<float>(N * 2);  
auto only_x = Load(d, all_x_then_y.data());  
auto only_y = Load(d, all_x_then_y.data() + N);
```

Branches

```
float RemoveRangeAroundZero(float w, float x) {  
    return      x > w ? x - w :  
                x < -w ? x + w : 0.0f;  
}  
  
template<class V>  
V RemoveRangeAroundZero(V w, V x) {  
    return IfThenElse(x > w, x - w,  
        IfThenElseZero(x < Neg(w), x + w));  
}  
  
bool AllPositiveIntegers(int v) {  
    return v >= 0;  
}  
  
template<class V>  
bool AllPositiveIntegers(V v) {  
    // avoids/hides 'zero'/'sign bit' constant  
    return AllTrue(Abs(v) == v);  
}
```

[Headers]

```
// Special include guard
#if defined(MYPROJECT_FILE_INL_H_) == \
    defined(HWY_TARGET_TOGGLE)
#ifdef MYPROJECT_FILE_INL_H_
#undef MYPROJECT_FILE_INL_H_
#else
#define MYPROJECT_FILE_INL_H_
#endif

// header contents, like normal SIMD module

#endif // include guard
```

"Toggles" include guard macro - prevents multiple inclusion within a particular target

Thanks to Lode Vandevenne for this clever idea!

03

Runtime dispatch

Multi-target

```
// At top of file, before other hwy includes
#undef HWY_TARGET_INCLUDE
#define HWY_TARGET_INCLUDE "path/filename.cc"
#include <hwy/foreach_target.h>

HWY_BEFORE_NAMESPACE();
// implementation - compiled once per target
HWY_AFTER_NAMESPACE();

#if HWY_ONCE
namespace myproject {
HWY_EXPORT(MyFunc); // defines function table

void Caller() {
    // dispatches to best available implementation
    HWY_DYNAMIC_DISPATCH(MyFunc)(args);
}
#endif
```

Definitions

Target = instruction set (e.g. AVX2)

Baseline = what compiler targets (= CPU requirement)
Determined by -mavx2 or HWY_BASELINE_TARGETS

Enabled = non-denylisted targets
Determined by known issues / HWY_DISABLED_TARGETS

Static target = best enabled baseline

Attainable = extra targets Highway can generate
Determined by compiler: all enabled on x86, or baseline

Superseded: baseline \ static target
If SSE4 baseline, skip scalar to reduce code size

[Dynamic] targets = configurable: { scalar | static | attainable |
attainable \ superseded }

Dispatching

```
// Direct call into baseline from normal code:  
// (Can make sense if baseline is sufficient -  
// avoid generating for all targets)  
HWY_STATIC_DISPATCH(MyFunc)(args);
```

SupportedTargets() // bitfield, depends on CPU

```
// Indirect call into best available SIMD:  
HWY_DYNAMIC_DISPATCH(MyFunc)(args);
```

```
// Call for each target from anywhere:  
hwy::RunTest(func, args);
```

gTest adapters also provided.

04

Design rationale

Simd<T, N>:: ?

SVE backend: as of 2021-05, vectors are sizeless types

Cannot be a class member

→ API based on overloaded functions

Can we have Load(V(), ptr)?

No, V is builtin on SVE, cannot indicate limit on #lanes

What about Load(V(), IntConst<N>, ptr)?

Error-prone - can break if not all call sites updated

→ still have tag argument called Simd<T, N>

Why auto?

Rarely need to know vector type, can deduce

Even for output params: `auto out = Undefined(d);`

Large number of vector types (50-70)

Types: {u, i} x {8, 16, 32, 64}, f32, f64

Lanes: 1, 2, 4, 8, 16, 32, 64

For portability, encourage size-agnostic code

But: auto everywhere hard to read / understand

➡ compromise: user-defined typedef:

`using V = decltype(Zero(d));` or

`using V = Vec<decltype(d)>;`

Why not stack?

SVE: max size 256 bytes. Wasteful on stack

RISC-V V: large upper bound: 64K elements

One implementation actually has 16 KiB vectors.

➡ use `hwy/aligned_allocator.h` and actual size: `Lanes(d)`

Why >1 header?

Including everything actually expensive (huge immintrin.h)

base.h for users / headers who just want

HWY_RESTRICT (function parameter) or

HWY_REP4 (define input for LoadDup128) etc.

targets.h useful for callers who want to know/influence

which target is active

highway.h only for implementers of SIMD modules, not their users

05

Users

Highway ecosystem

4.1K Github stars

41 contributors with >1 CL

>100 unique Git cloners per day

Available in [dozens of package managers](#)

Evaluation of C++ SIMD Libraries:

"Highway excelled with a strong performance across multiple SIMD extensions [...]. Thus, Highway may currently be the most suitable SIMD library for many software projects."



JPEG XL

Next-generation image compression. jpeg.org/jpegxl

Uses integer (random generation) and floating-point:

- DCT, filtering, color conversion, noise synthesis
- Quantization, function approximations, ...

Runtime dispatch, 1.4x speedup from AVX2 to AVX-512

Main dev team: 9 engineers, positive feedback on Highway API

[Shipping on iOS 17 and OS X 14](#)



Gemma.cpp

Lightweight, standalone C++ LLM inference on [github](#)

Supports Gemma, RecurrentGemma, PaliGemma (soon)

8-bit floating-point decompression, bf16 arithmetic

[Faster than llama.cpp](#) on SKX/Zen4 CPUs

Experimentation platform for inference R&D



Also related

VQSort: [fastest known](#) QuickSort

- Vectorized partitioning and 2D sorting network

HighwayHash: fast MAC/pseudorandom function

- Built around [SIMD multiply and permute](#)
- 64, 128, 256 bit result (1024 bit internal state)
- Difficult to create collisions ($> 2^{64}$ work)
- Similar SIMD via intrinsics + runtime dispatch

Randen: [Abseil's random generator](#)

- Cryptographic ([indistinguishable from random](#))
- Faster than some common insecure generators
- Enabled by SIMD AES (x86, ARM, POWER)
- Simpler SIMD wrapper over intrinsics



Questions/contact

janwas@google.com

Staff Software Engineer

highway-users@google.com

More info: go/hwy, [workshop slides](#)