Long vectors for WebAssembly

Petr Penzin

Intel Corporation

April 8, 2020

Background

- A number of discussions in the context of Wasm SIMD proposal regarding operations longer than 128-bit[1][2]
- Existing runtime solutions
  - Highway
  - System.Numerics.Vector in .NET

[1] Github issue #29, and a version of this deck has been presented on November 6, 2019
[2] #210, #212

Scope

This proposal is about:

- Length-agnostic vector operations which bridge the gap between hardware SIMD implementations

Out of scope:

- Supporting vector instruction sets, such as SVE (though it can in the future)
- Changing current Wasm SIMD proposal

Design Goals

- Same Wasm binary to run all platforms
- Unambiguous instruction selection
- Easy transition from Wasm SIMD instruction set

Alternatives

Longer fixed-width SIMD WebAssembly ISA

- Not universally supported in hardware
- Goes against WebAssembly's design goal of representing the common set of operations between hardware platforms
- Cross platform code generation is challenging

Proposal

We propose length-agnostic variants of operations already present in Wasm *simd128* proposal

- Loads and stores work with consecutive memory locations, like *simd128* loads and stores
- Maximum vector length is set to match the hardware by runtime at startup

Types and instructions

New types and instructions

- $vec. < type >$ – separate vector types for different lane types, size defaults to maximum supported by hardware
    - $i8$, $i16$, $i32$, $i64$ – integer
    - $f32$, $f64$ – floating point
- $vec. < type > .length$ – get number of elements in corresponding vector type

Types and instructions

Instructions extending existing operations in WebAssembly SIMD proposal

- $vec. < type > . < op >$ – same lane-wise operation as in *simd128* $< op >$, applied to vector of $vec. < type > .length$

  For example, $vec.f32.mul$ is identical to $f32x4.mul$ on a 4-lane vector, $vec.i32.add$ to $i32x4.add$, and so on

Example

Vector addition, $c = a + b$, sz is the size

```
(block $loop
  (block $loop_top
    (br_if $loop (i32.lt (get_local $sz) (vec.f32.length)))
    vec.f32.load (get_local $a)
    vec.f32.load (get_local $b)
    vec.f32.add
    vec.f32.store (get_local $c)
    ;; Decrement $sz and increment $a, $b, $c
    (br $loop_top)
  )
)
(block $scalar_loop ;; Finish the remaining elements
```

Code generation

- Identical to *simd128* for platforms that support only 128 bit SIMD
- Straight-forward extension to longer vectors on supporting platforms

Comparison against current SIMD proposal

- At 128-bit vector width operations are identical to current Wasm SIMD operations with sole exception of lane shuffle
- Transparent to developer and toolchain

Poll

Support phase 0 proposal for long vectors?

Thank you

Backup: Pure vectors

Proposal can be extended to support pure vectors - with user-visible length, but that would be challenging to execute on existing hardware.
It can be done by adding the following instruction:

- $vec. < type > .set\_length$ – set number of elements in corresponding vector type

  Takes an unsigned argument, allowed use smaller number per runtime's view of the hardware

Example

Vector addition, $c = a + b$, sz is the size

```
local $len i32
(block $loop
  (block $loop_top
    (br_if $loop (i32.eq (get_local $sz) (i32.const 0)))
    (set_local $len (vec.f32.set_length (get_local $sz)))
    vec.f32.load (get_local $a)
    vec.f32.load (get_local $b)
    vec.f32.add
    vec.f32.store (get_local $c)
    ;; Decrement $sz by $len; increment $a, $b, and $c by $len
    (br $loop_top)
  )
)
```

Code generation

Advantages:

- Reduced Wasm instruction count
- Some alignment with SIMD instruction sets supporting masking

Disadvantages:

- High cost for SIMD instruction sets without masking
- Managing global state

This can be seen as a future or experimental option, but it is not ready to be prototyped on widely available hardware.