RFC: Masks challenges for WASM flexible vectors

Florian Lemaitre

LIP6, Sorbonne University, CNRS, France

21/05/2021

Who I am

Florian Lemaitre

Career:

- Post-doctoral researcher at Sorbonne University (Paris, France)
  - Low-power motion detection (Cortex-M7)
  - Connected Component Labeling and Analysis (CPU & GPU)
- PhD in Computer Science (architecture) at CERN (Geneva)
  - High Throughput Computing (CPU)
  - Cholesky factorization and Kalman filter in low dimension

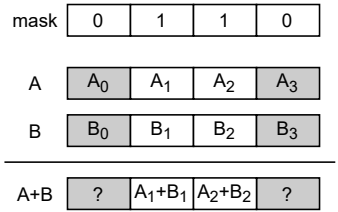
Fields of expertise:

- "SIMD"
  - CPU: SSE, AVX, AVX512, Neon, SVE, Altivec (VMX, VSX)
  - GPU: CUDA
- Computer architecture (software side)
  - x86 (Intel, AMD), ARM, PowerPC, Nvidia GPU
- Parallel programming
  - OpenMP, Atomics (Lock-free), Numa, Posix threads

What is a mask, already?

A mask is an object that indicates which elements of a vector are active

- An operation on a vector is applied only on active elements
- Mandatory to convert branchy scalar code
- Useful for loop prolog/epilog



There are several policies for inactive elements

- Keep destination elements
- Copy input elements
- Zero
- “Don’t care”

There are several ways to represent masks in hardware:

- 1 bit per element: AVX512, Risc-V V?
- 1 bit per byte: SVE
- 1 bit per bit: SSE, AVX, Neon, Altivec...

Example: sum of positive

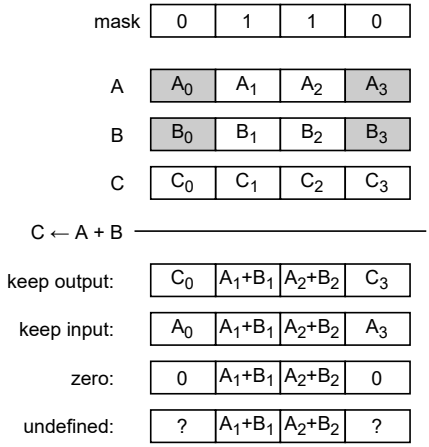
```scalar
// Scalar
int foo(const int* A, int n) {
  int s = 0;
  for (int i = 0; i < n; ++i) {
    int v = A[i];
    if (v > 0) {
      s += v;
    }
  }
  return s;
}
```

```avx512
// AVX512
void foo(const int* A, int* B, int n) {
  const __m512i zero = _mm512_setzero_si512();
  __m512i s = _mm512_setzero_si512();
  int i;

  // loop body
  for (i = 0; i < n-16; i += 16) {
    __m512i v = _mm512_load_epi32(&A[i]);
    __mmask16 m = _mm512_cmpgt_epi32_mask(v, zero);
    s = _mm512_mask_add_epi32(s, m, s, v);
  }

  // loop epilog
  if (i < n) {
    __mmask16 m = (1 << (n - i)) - 1;
    __m512i v = _mm512_maskz_load_epi32(m, &A[i]);
    m = _mm512_mask_cmpgt_epi32_mask(m, v, zero);
    s = _mm512_mask_add_epi32(s, m, s, v);
  }

  // reduction
  return _mm512_reduce_add_epi32(s);
}
```

Inactive element policy

- Keep destination: does not alter the inactive elements of the destination
- Copy input: inactive elements are copied from one of the input
- Zero: inactive elements are set to zero
- “Don’t care” (undefined): the compiler/machine is allowed to put anything in inactive elements

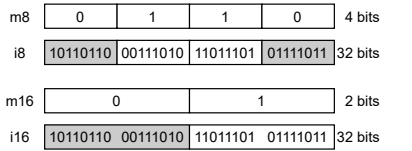
- For memory loads, usually keep destination or zero
- For memory stores, always keep destination
- For register-to-register operations, not a clear distinction between keep destination and copy input
- Keep destination and copy input are usually referred to as “merge”



Mask hardware representation

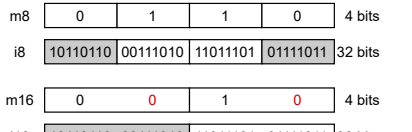
1 bit per element (vector size agnostic):

- Used by AVX512 and Risc-V V?
- Simple to mix vectors with same cardinal (but different element size)
- The most compact



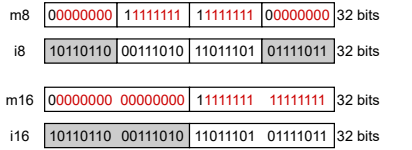
1 bit per byte (element size agnostic):

- Used by SVE
- For larger types, extra bits are ignored (and set to 0 on write)



1 bit per bit (bit mask, element size agnostic):

- Used by SSE, AVX, Neon, Altivec...
- Simple to mix vectors with same vector size (but different element size)
- Most of the bits are redundant, and might not be ignored
- Require only bitwise instructions for most operations



Abstract representation

WASM must be able to target platforms with different mask representation.

⇒ One mask type per element size

- WASM engine could choose native representation for target architecture
- Specific instructions to convert masks between different element sizes
  - Depending on the ISA, some conversions might be NOOPs
- Specific instructions to load/store masks
  - Which format to use in memory? Specific format? Unspecified? Left to the user?

Efficient design for legacy architecture

On legacy architectures, masked operations might be slower. It would be nice if there was a way to make full masks as fast as no mask.

- This could be done with *constant folding* and *monomorphization* at the engine level
- Every function taking a mask is compiled twice by the engine
  - Once with mask known to be full
  - Once with unknown and arbitrary mask
- When the engine detects a mask is full, it will call functions specialized for full masks
- Requires to duplicate the loop body for the engine to know that mask is full for all the iteration but the last that will be handled in the prolog.

How to handle masked memory accesses when no specialized exist (or too slow)?

- Scalarization
- Signal handler (for unaligned accesses): ignore the fault if element is masked out
- Compare-And-Swap loop (for stores)

WASM stack

When writing masked code, most operations use the same mask. How to deal with the stack?

- No special handling of masks
  - needs to duplicate the mask for every single masked operations
- Masked operations push the mask back on the stack
  - All operations have 2 outputs
- Section that defines the current mask for masked operations
  - Does not go through calls
- Separate stack for masks
  - Masked operations do no pop the mask stack
  - Much more complex WASM virtual machine
- Hybrid (whatever combination of the above)

Conclusion

Masks are an essential part of SIMD, and makes epilog handling natural, but comes with many challenges.

- Hardware implementations vary vastly: we need a way to abstract those differences
- Designing efficient abstraction for legacy architectures is hard
- As-is, the stack is not suited for masked operations

Questions?