Bit vectors without compromises

Andrew Lelechenko
1@dxdy.ru

Barclays, London

Haskell Love, 31.07.2020

Bit vectors

Bit vector is an array of booleans.
It can be represented as unboxed `Vector Bool`.

Thank you!

Bit vectors as `Vector Bool` from `vector`

Pros:

- Allocates a continuous memory segment.
- Random access is $O(1)$.
- Has a mutable counterpart, so updates are $O(1)$.
- Slicing is $O(1)$.
- Loop fusion framework and rich API.

Cons:

- Stores only 1 value per byte.
  So requires $8\times$ more space than theoretically possible.
- Processes arrays bit by bit.
  So `map` and `zip` are $64\times$ slower than possible.

Bit vectors as `Array Bool` from `array`

Pros:

- Allocates a continuous memory segment.
- Random access is $O(1)$.
- Has a mutable counterpart, so updates are $O(1)$.
- Stores 64 values per `Word64`.

Cons:

- Processes arrays bit by bit.
  So `map` and `zip` are 64x slower than possible.
- Slicing is $O(n)$.
- No loop fusion framework and very limited API.

Bit vectors as `IntSet` from `containers`

Pros:

- The best representation for sparse bit vectors.
- Could store more than 8 values per `Word64`.
- Set operations are capable to process 64 elements at once.
- Rich API.

Cons:

- Employs a lot of pointers, not quite cache-friendly.
- Random access is $O(\log n)$.
- No mutable counterpart, so updates are $O(\log n)$.

Bit vectors as `Integer` from `bv`

Pros:

- Allocates a continuous memory segment.
- Random access is $O(1)$.
- Stores 64 values per `Word64`.
- Some operations are capable process 64 elements at once.

Cons:

- No mutable counterpart, so updates are $O(n)$.

No compromises

- Full-fledged `Vector` and `MVector` instances with expected asymptotic complexity.
- Handy `Bits` instance with vectorised blockwise operations.
- As compact as possible: store 64 bits per `Word64`.
- Allocate a continuous memory segment.

```haskell
newtype Bit = Bit { unBit :: Bool }
data BitVec = BitVec
  { offset :: Int, -- in bits
  , length :: Int, -- in bits
  , array  :: ByteArray
  }

index :: BitVec → Int → Bit
index (BitVec offset _ array) i =
  Bit (testBit (indexByteArray array q) r)
  where (q, r) = (i + offset) `quotRem` 64
```

Writing a mutable bit vector

```
data MBitVec s = MBitVec
  { offset :: Int, -- in bits
  , length :: Int, -- in bits
  , array  :: MutableByteArray s
  }

write :: MBitVec s → Int → Bit → ST s ()
write (MBitVec offset _ array) i (Bit b) = do
  let (q, r) = (i + offset) `quotRem` 64
  old ← readByteArray array q
  let new = (if b then setBit else clearBit) old r
  writeByteArray array q new
```

What could go wrong in a concurrent environment?

Thread-safe writes

Imagine having an atomic compare-and-swap (CAS):

```
casArray :: MutableArray s a → Int → a → a → ST s a
casArray array offset expected new = do
  actual ← readByteArray array offset
  when (actual == expected) $
    writeByteArray array offset new
  pure actual

write (MBitVec offset _ array) i (Bit b) =
  readByteArray array q >>= go
  where
    (q, r) = (i + offset) `quotRem` 64
    go expected = do
      let new = (if b then setBit else clearBit) expected r
      actual ← casArray array q expected new
      when (actual /= expected) (go actual)
```

Better thread-safe writes

GHC.Exts provides functions equivalent to

```
fetchAndIntArray, fetchOrIntArray, fetchXorIntArray
  :: MutableByteArray s → Int → Int → ST s Int
fetchAndIntArray array offset mask = ...

write :: MBitVec s → Int → Bit → ST s ()
write (MBitVec offset _ array) i (Bit b) = if b
  then fetchOrIntArray  array q              (bit r)
  else fetchAndIntArray array q (complement (bit r))
  where (q, r) = (i + offset) `quotRem` 64
```

Modifying a mutable bit vector

```
modify :: MVector s a → (a → a) → Int → ST s ()
modify vec func offset = ...
```

```
modify :: MVector s Bit → (Bit → Bit) → Int → ST s ()
```

There are only 4 functions `Bit → Bit`:

- `id`,
- `const True`,
- `const False`,
- `not`.

Flipping a bit

```haskell
flipBit :: MVector s Bit → Int → ST s ()
flipBit vec i = do
  Bit b ← read vec i
  write vec i (not b)
```

What could go wrong in a concurrent environment?

```haskell
flipBit :: MVector s Bit → Int → ST s ()
flipBit (MBitVec offset _ array) i =
  fetchXorIntArray array q (bit r)
  where (q, r) = (i + offset) `quotRem` 64
```

Killing two birds with one stone:

- Faster!
- Thread-safe!

Test your unboxed vectors

- `MVector` interface requires also defining `copy`, `move`, `set`, `grow`, all dealing correctly with (possibly, unaligned) offsets and lengths.
- Covering all cases with unit tests is unfeasible.
- `bitvec-0.1` was notoriously buggy.
- `Test.QuickCheck.Classes.muvectorLaws` provides ~ 30 properties for thorough testing of unboxed mutable vectors.
- Also available from `Hedgehog.Classes.muvectorLaws`.
- These properties have proved to be useful in test suites of `bitvec`, `arithmoi`, `mod`...

Blockwise map

```
map :: (a   → a)   → Vector a   → Vector a
map :: (Bit → Bit) → Vector Bit → Vector Bit
```

There are only 4 functions `Bit → Bit`:

- `id`,
- `const True`,
- `const False`,
- `not`.

```
invertBits :: Vector Bit → Vector Bit
```

- Invert 64 bits at once, applying `complement` to `Word64`.
- Take extra care for unaligned vectors in concurrent environment.
- Use `mpn_com` from GMP for ultra-fast vectorised processing, up to 1000× faster than `Vector Bool`.

Blockwise zip

```
zip :: (a → a → a) → Vector a → Vector a → Vector a
zip :: (Bit → Bit → Bit)
    → Vector Bit → Vector Bit → Vector Bit
```

There are only 16 functions `Bit` → `Bit` → `Bit`:

- `True`, `False`, x, $\bar{x}$, y, $\bar{y}$,
- $x \wedge y$, $\bar{x} \wedge y$, $x \wedge \bar{y}$, $\bar{x} \wedge \bar{y}$,
- $x \vee y$, $\bar{x} \vee y$, $x \vee \bar{y}$, $\bar{x} \vee \bar{y}$,
- $x + y$, $\bar{x} + y$, $x + \bar{y}$, $\bar{x} + \bar{y}$.

- Zip 64 bits at once, applying `complement`, `.&.`, `.|.` and `xor` on `Word64`.
- Aligning two vectors with different offsets is tough by itself, and becomes even worse in concurrent environment.
- Use rountines from GMP for ultra-fast vectorised processing, up to 1000× faster than `Vector Bool`.

Additional goodness in `bitvec`

- Blockwise population count and its reverse `nthBitIndex`.
- Operations for succinct data structures, backed by BMI2 instructions.
- Ultra-fast reversal, up to $O(1)$.
- Boolean polynomials for cryptographic applications.
- Conversions from/to `Vector Word` and `ByteString`.

- Full-fledged `Vector` and `MVector` instances with expected asymptotic complexity (but constant factor is up to 20% larger).
- Handy `Bits` instance with vectorised blockwise operations (usually 64$\times$ and up to 1000$\times$ faster).
- Allocate 8$\times$ less memory than `Vector Bool`.

Thank you!

1@dxdy.ru Bodigrim

github.com/Bodigrim/bitvec github.com/Bodigrim/my-talks