VIR, an intermediate representation for SIMD text processing

Loup Lobet

Inria Lille & ULille
D-DAL team
Supervised by L. Gonnord (Grenoble INP-UGA) and C. Paperman 🐓 (ULille)

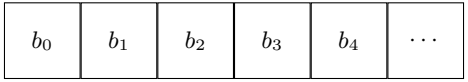
Sept 17th 2026
Verimag

Context

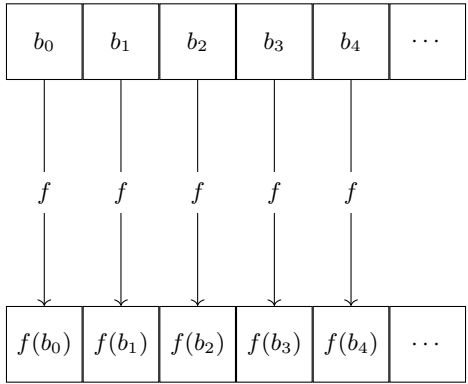
- First vector supercomputers in 1970s
- Computation speedup through specialized hardware
- In modern CPU as *Single Instruction Multiple Data* (SIMD) architecture

Figure: Cray-1 supercomputer

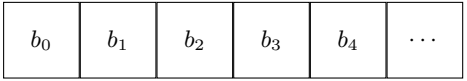
The idea behind SIMD



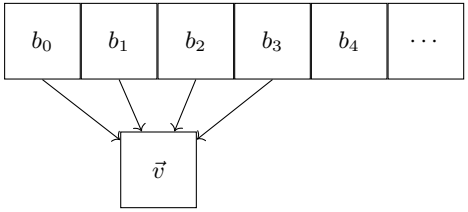
The idea behind SIMD



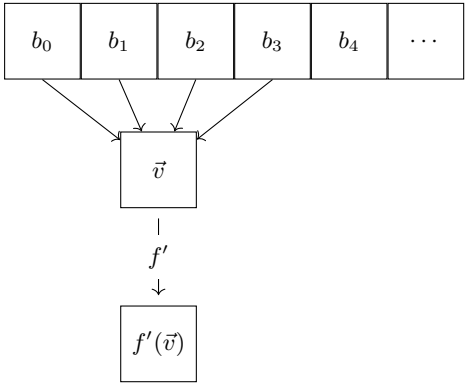
The idea behind SIMD



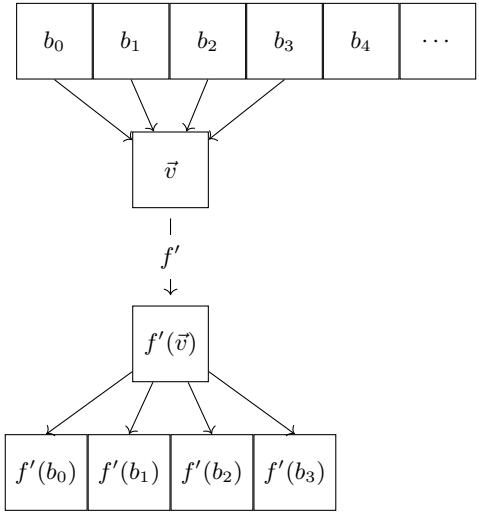
The idea behind SIMD



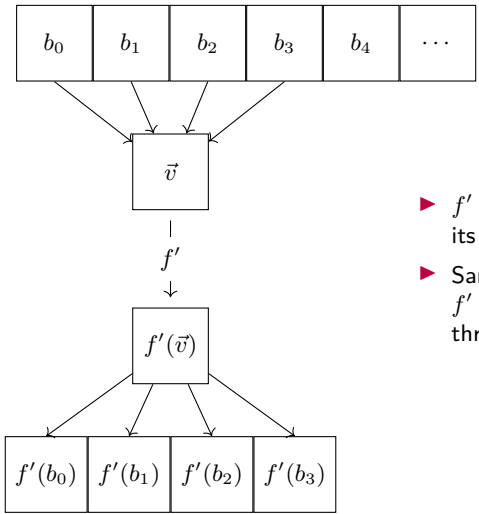
The idea behind SIMD



The idea behind SIMD



The idea behind SIMD



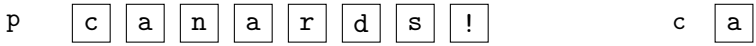
- f' is provided by the CPU as part of its SIMD instruction set
- Same latency between f and f', thus f' usage results in increased throughput

Quick example with memchr(3)

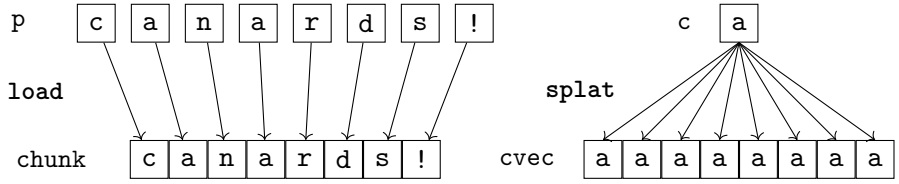
Find the first iteration of c as a char in s:

```
 0  void *
 1  memchr(const void *p, int c, size_t n)
 2  {
 3      if (n) {
 4          const unsigned char *p = s;
 5          do {
 6              if (*p++ == (unsigned char)c)
 7                  return ((void *)(p - 1));
 8          } while (--n);
 9      }
10      return NULL;
11  }
```

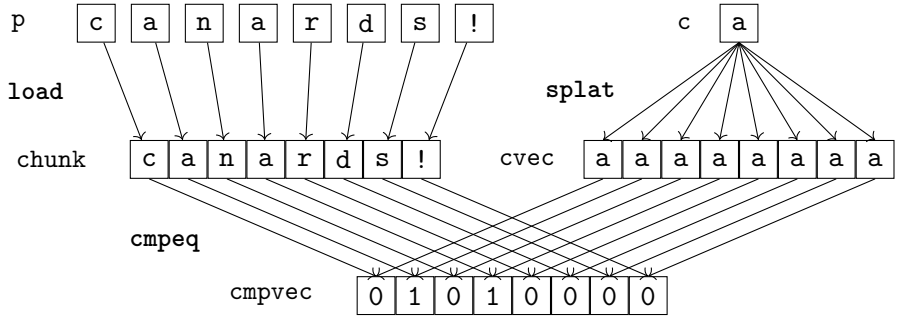
Quick example with `memchr(3)`



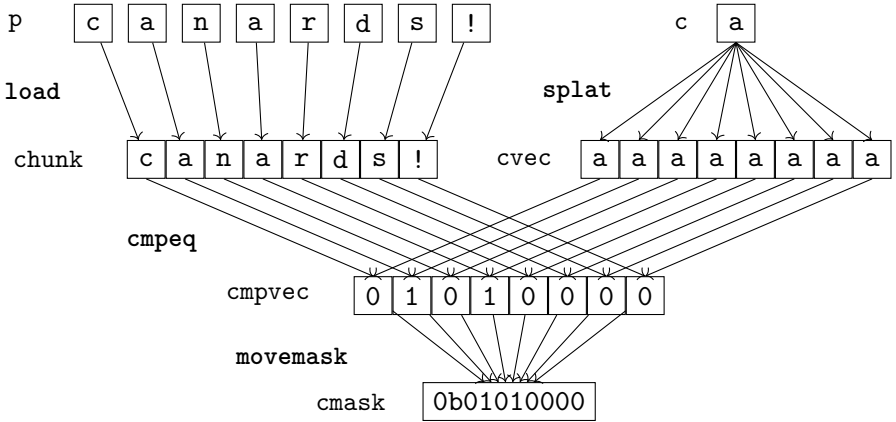
Quick example with `memchr(3)`



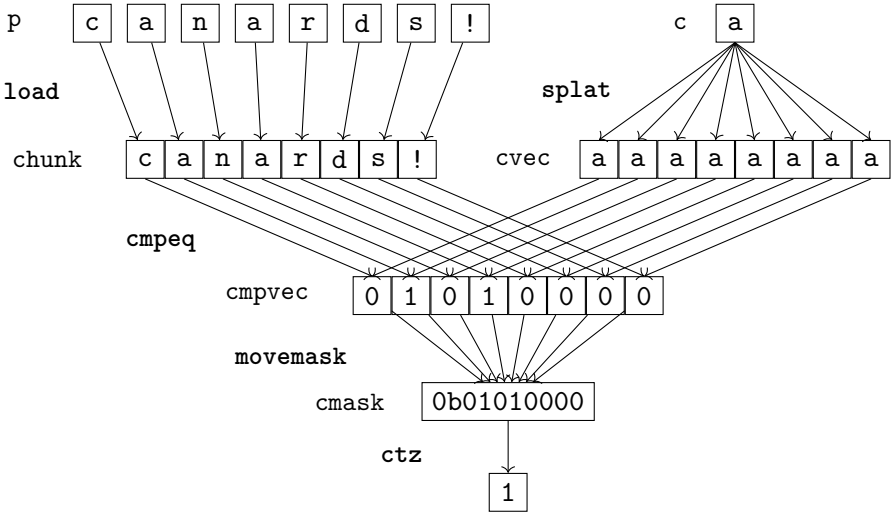
Quick example with `memchr(3)`



Quick example with `memchr(3)`

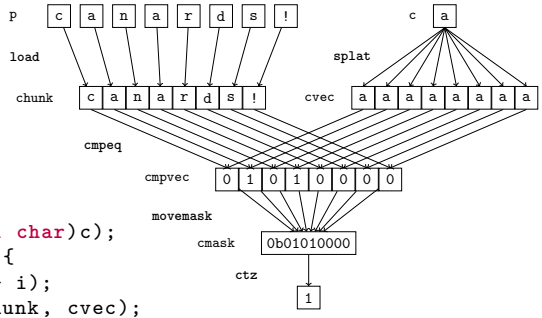


Quick example with `memchr(3)`



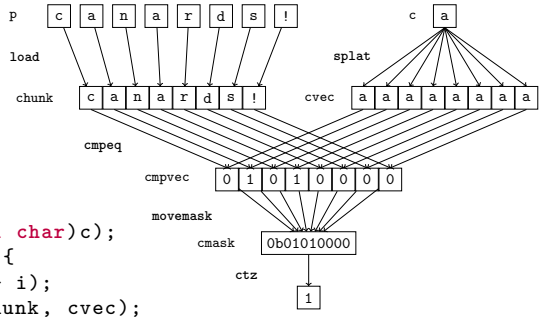
Quick example with memchr(3)

```
0  void *
1  vmemchr(void *p, int c, size_t n)
2  {
3      __m256i chunk, cmpvec, cvec;
4      int cmask, i;
5
6      cvec = _mm256_set1_epi8((unsigned char)c);
7      for (i = 0; i + 32 < n; i += 32) {
8          chunk = _mm256_load_si256(p + i);
9          cmpvec = _mm256_cmpeq_epi8(chunk, cvec);
10         cmask = _mm256_movemask_epi8(cmpvec);
11         if (cmask)
12             return p + i + ctz(cmask);
13     }
14     return memchr(p + i, c, n - i);
15 }
```



Quick example with `memchr(3)`

```
0  void *
1  vmemchr(void *p, int c, size_t n)
2  {
3      __m256i chunk, cmpvec, cvec;
4      int cmask, i;
5
6      cvec = _mm256_set1_epi8((unsigned char)c);
7      for (i = 0; i + 32 < n; i += 32) {
8          chunk = _mm256_load_si256(p + i);
9          cmpvec = _mm256_cmpeq_epi8(chunk, cvec);
10         cmask = _mm256_movemask_epi8(cmpvec);
11         if (cmask)
12             return p + i + ctz(cmask);
13     }
14     return memchr(p + i, c, n - i);
15 }
```



General purpose compilers cannot do such transformation!

- Array breakdown into chunks (scalar queue handling)
- Use of `movemask`
- Early return

Pros and cons

Pros and cons

Pros:

- Significant performance gains (2x-20x)
- Benefits from instruction pipelining
- Highly available in desktop CPU
- Good compilers are capable of auto-vectorization

Pros and cons

Pros:

- Significant performance gains (2x-20x)
- Benefits from instruction pipelining
- Highly available in desktop CPU
- Good compilers are capable of auto-vectorization

Cons:

- Moving data causes extra overhead
- Memory alignment and data layout matters
- SIMD programming is hard, and is hardware dependant
- Adapting algorithms isn't always trivial

The Shannon X Cray project, and the Vectoid language

Funded by ANR, led by C. Paperman (Lille), G. Radanne (Lyon), L. Gonnord (Grenoble), N. Fijalkow (Bordeaux)

The Shannon X Cray project, and the Vectoid language

Funded by ANR, led by C. Paperman (Lille), G. Radanne (Lyon), L. Gonnord (Grenoble), N. Fijalkow (Bordeaux)

- Auto-vectorization only works in a few cases, and handcrafted vectorization is hard

The Shannon X Cray project, and the Vectoid language

Funded by ANR, led by C. Paperman (Lille), G. Radanne (Lyon), L. Gonnord (Grenoble), N. Fijalkow (Bordeaux)

- Auto-vectorization only works in a few cases, and handcrafted vectorization is hard
- Stream processing is highly vectorizable

The Shannon X Cray project, and the Vectoid language

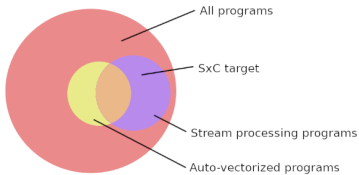
Funded by ANR, led by C. Paperman (Lille), G. Radanne (Lyon), L. Gonnord (Grenoble), N. Fijalkow (Bordeaux)

- Auto-vectorization only works in a few cases, and handcrafted vectorization is hard
- Stream processing is highly vectorizable
- Constraints on expressiveness and semantics for new vectorization oportunities

The Shannon X Cray project, and the Vectoid language

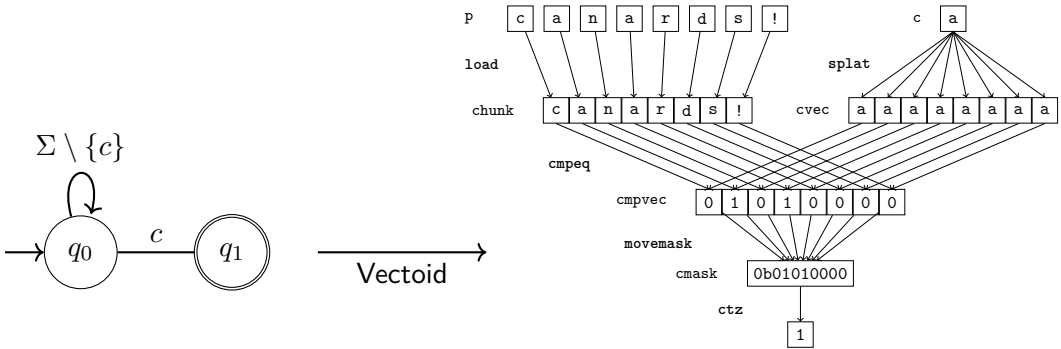
Funded by ANR, led by C. Paperman (Lille), G. Radanne (Lyon), L. Gonnord (Grenoble), N. Fijalkow (Bordeaux)

- Auto-vectorization only works in a few cases, and handcrafted vectorization is hard
- Stream processing is highly vectorizable
- Constraints on expressiveness and semantics for new vectorization oportunities



My thesis: Design an intermediate representation (VIR) for a new a DSL (*Vectoid*).

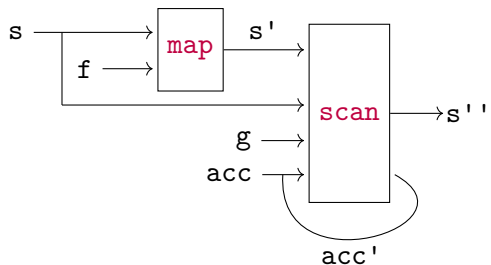
Presentation of Vectoid and VIR



Turn an automaton / transducer (expressed as mutually recursive functions) into bounded depth circuit(s), with a width equal to memory word / SIMD vector length (current *Etienne Parent*'s Thesis).

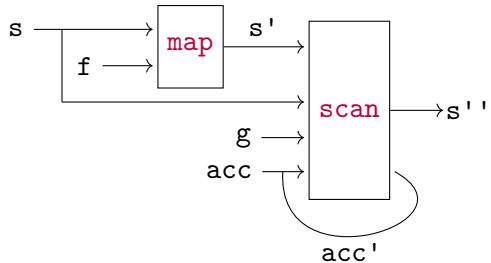
VIR semantics

VIR expresses an acyclic dataflow with functional combinators, where the circuits are used as SIMD "kernels". Dataflow has a **pulling semantics**, where each node requests it's next input to upstream producer (blocking).



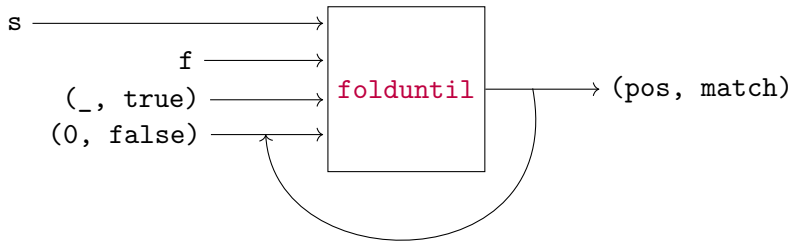
VIR semantics

VIR expresses an acyclic dataflow with functional combinators, where the circuits are used as SIMD "kernels". Dataflow has a **pulling semantics**, where each node requests it's next input to upstream producer (blocking).



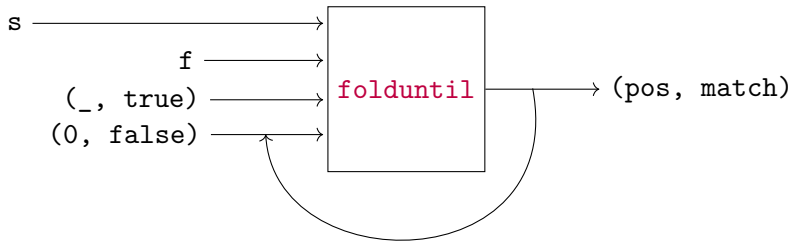
- deterministic computation with no RT constraints
- input stream(s) can be finite
- ouput dictates node execution rhythm and can be a stream or/and an atom
- semantics is equivalent to KPN
- possibly unbounded memory on sharing points

Dataflow of `memchr(3)`



- input stream `s: #[u8;32]`
- circuit `f: ([u8;32], (usize, bool)) -> (usize, bool)`
- break accumulator value: `(_, true)`
- start accumulator value: `(0, false)`

Dataflow of `memchr(3)`



- input stream `s: #[u8;32]`
- circuit `f: ([u8;32], (usize, bool)) -> (usize, bool)`
- break accumulator value: `(_, true)`
- start accumulator value: `(0, false)`

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3              // spalt c over a 32 bytes vector
4              let c_vec: [u8; 32] = splat(c, [u8; 32];
5              // vectorial comparison
6              let cmp_vec: [u8; 32] = cmp(v, c);
7              let mask: u32 = movemask(cmp_vec);
8              if mask > 0 {
9                  return (acc.0 + ctz(mask), true);
10             } else {
11                 return (acc.0 + 32, false);
12             }
13         },
14         chunks, (0, false), (_, true)
15     );
16     return cnt;
17 }
```

Define new dataflow function

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3              // spalt c over a 32 bytes vector
4              let c_vec: [u8; 32] = splat(c, [u8; 32];
5              // vectorial comparison
6              let cmp_vec: [u8; 32] = cmp(v, c);
7              let mask: u32 = movemask(cmp_vec);
8              if mask > 0 {
9                  return (acc.0 + ctz(mask), true);
10             } else {
11                 return (acc.0 + 32, false);
12             }
13         },
14         chunks, (0, false), (_, true)
15     );
16     return cnt;
17 }
```

Compute the position of the first c

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3              // spalt c over a 32 bytes vector
4              let c_vec: [u8; 32] = splat(c, [u8; 32]);
5              // vectorial comparison
6              let cmp_vec: [u8; 32] = cmp(v, c);
7              let mask: u32 = movemask(cmp_vec);
8              if mask > 0 {
9                  return (acc.0 + ctz(mask), true);
10             } else {
11                 return (acc.0 + 32, false);
12             }
13         },
14         chunks, (0, false), (_, true)
15     );
16     return cnt;
17 }
```

Define the `folduntil` atom-to-atom function

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3              // spalt c over a 32 bytes vector
4              let c_vec: [u8; 32] = splat(c, [u8; 32];
5              // vectorial comparison
6              let cmp_vec: [u8; 32] = cmp(v, c);
7              let mask: u32 = movemask(cmp_vec);
8              if mask > 0 {
9                  return (acc.0 + ctz(mask), true);
10             } else {
11                 return (acc.0 + 32, false);
12             }
13         },
14         chunks, (0, false), (_, true)
15     );
16     return cnt;
17 }
```

Splat c over a 32x8 bits vector

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3              // spalt c over a 32 bytes vector
4              let c_vec: [u8; 32] = splat(c, [u8; 32];
5              // vectorial comparison
6              let cmp_vec: [u8; 32] = cmp(v, c);
7              let mask: u32 = movemask(cmp_vec);
8              if mask > 0 {
9                  return (acc.0 + ctz(mask), true);
10             } else {
11                 return (acc.0 + 32, false);
12             }
13         },
14         chunks, (0, false), (_, true)
15     );
16     return cnt;
17 }
```

Performe vectorial comparison

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3              // spalt c over a 32 bytes vector
4              let c_vec: [u8; 32] = splat(c, [u8; 32];
5              // vectorial comparison
6              let cmp_vec: [u8; 32] = cmp(v, c);
7              let mask: u32 = movemask(cmp_vec);
8              if mask > 0 {
9                  return (acc.0 + ctz(mask), true);
10             } else {
11                 return (acc.0 + 32, false);
12             }
13         },
14         chunks, (0, false), (_, true)
15     );
16     return cnt;
17 }
```

Group each vector elment LSB in a u32

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3              // spalt c over a 32 bytes vector
4              let c_vec: [u8; 32] = splat(c, [u8; 32];
5              // vectorial comparison
6              let cmp_vec: [u8; 32] = cmp(v, c);
7              let mask: u32 = movemask(cmp_vec);
8              if mask > 0 {
9                  return (acc.0 + ctz(mask), true);
10             } else {
11                 return (acc.0 + 32, false);
12             }
13         },
14         chunks, (0, false), (_, true)
15     );
16     return cnt;
17 }
```

Match found return position and `true`

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3              // spalt c over a 32 bytes vector
4              let c_vec: [u8; 32] = splat(c, [u8; 32];
5              // vectorial comparison
6              let cmp_vec: [u8; 32] = cmp(v, c);
7              let mask: u32 = movemask(cmp_vec);
8              if mask > 0 {
9                  return (acc.0 + ctz(mask), true);
10             } else {
11                 return (acc.0 + 32, false);
12             }
13         },
14         chunks, (0, false), (_, true)
15     );
16     return cnt;
17 }
```

No match found, update position by 32 bytes and continue

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3              // spalt c over a 32 bytes vector
4              let c_vec: [u8; 32] = splat(c, [u8; 32]);
5              // vectorial comparison
6              let cmp_vec: [u8; 32] = cmp(v, c);
7              let mask: u32 = movemask(cmp_vec);
8              if mask > 0 {
9                  return (acc.0 + ctz(mask), true);
10             } else {
11                 return (acc.0 + 32, false);
12             }
13         },
14         chunks, (0, false), (_, true)
15     );
16     return cnt;
17 }
```

Start at position 0 and flag as `false`, stop when the flag becomes `true`

memchr(3) in VIR

```
0  dataflow memchr_32(chunks: #[u8;32], c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2              |v: [u8; 32], acc: (usize, bool)| -> (usize, bool) {
3                  let c_vec: [u8; 32] = splat(c, [u8; 32];
4                  let cmp_vec: [u8; 32] = cmp(v, c);
5                  let mask: u32 = movemask(cmp_vec);
6                  if mask > 0 {
7                      return (acc.0 + ctz(mask), true);
8                  } else {
9                      return (acc.0 + 32, false);
10                 }
11             },
12         chunks, (0, false), (_, true)
13     );
14     return cnt;
15 }
```

dataflow
circuit (imperative)

Packing and unpacking

On finite data, unalignment with SIMD vector always lead to the same kind of algorithms with two different state

- a steady-state where the stream is processed by SIMD instruction
- a transient state where the stream queue is processed sequentially

Packing and unpacking

On finite data, unalignment with SIMD vector always lead to the same kind of algorithms with two different state

- a steady-state where the stream is processed by SIMD instruction
- a transient state where the stream queue is processed sequentially

For stream chunking VIR used the `pack` and `unpack` operators:

`sizeof(T) < sizeof(K)`

`sizeof(K) > sizeof(T)`

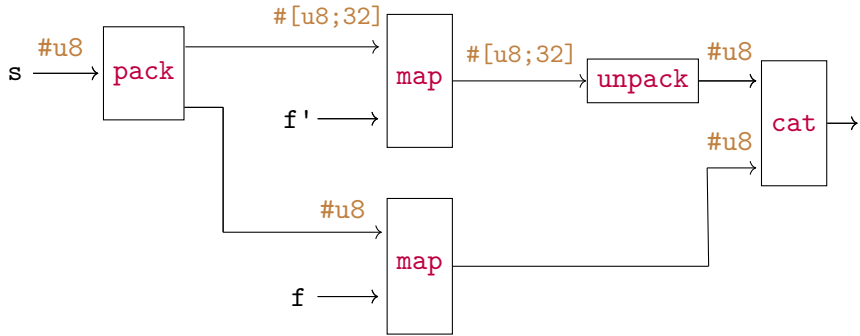
- `T` and `K` are memory aligned
- `pack` (resp. `unpack`) speeds (resp. slows) upstream nodes execution rate
- `pack` and `unpack` are in fact a buffers of size `sizeof(K)`

Packing and unpacking

E.g., we want to convert a stream of bytes over `[a-zA-Z]*` to uppercase:

Packing and unpacking

E.g., we want to convert a stream of bytes over `[a-zA-Z]*` to uppercase:



- `f = |x: u8| -> u8 { 0xdf & x }`
- `f' = |x: [u8;32]| -> [u8;32] { 0xdf & x }` (SIMD bitwise AND)
- Vectoid is able to give both `f` and `f'` (same circuit with different width)

memchr_32 sequential counterpart:

```
0  dataflow memchr(chunks: \#u8, c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: u8, acc: (usize, bool)| -> (usize, bool) {
3              if v == c {
4                  return (acc.0, true)
5              } else {
6                  return (acc + 1, false)
7              }
8          },
9          chunks, (0, false), (_, true)
10     );
11     return cnt;
12 }
```

`memchr_32` sequential counterpart:

```
0  dataflow memchr(chunks: \#u8, c: u8) -> (usize, bool) {
1      let cnt: (usize, bool) = folduntil(
2          |v: u8, acc: (usize, bool)| -> (usize, bool) {
3              if v == c {
4                  return (acc.0, true)
5              } else {
6                  return (acc + 1, false)
7              }
8          },
9          chunks, (0, false), (_, true)
10     );
11     return cnt;
12 }
```

- Same overall structure, byte equality replaces the "`cmp`, `movemask`, `ctz`" technique
- not always trivial, e.g. vectorized sparse byte removal / addition in a stream

Additional control flow with modes

Addition of mode automata [*Maraninchi*, *Rémond*] from reactive systems to VIR for enhanced control flow.

Additional control flow with modes

Addition of mode automata [*Maraninchi*, *Rémond*] from reactive systems to VIR for enhanced control flow.

- mutual recursion
- straightforward generation from Vectoid input automata/transducers
- control flow between steady states

Additional control flow with modes

Addition of mode automata [*Maraninchi*, *Rémond*] from reactive systems to VIR for enhanced control flow.

- mutual recursion
- straightforward generation from Vectoid input automata/transducers
- control flow between steady states

E.g. Let `s` be a raw text stream, and let `names` be a `csv` stream of names, both `'\0'` terminated. Replace every `'@'` in `s`, with the next name in `names`.

`s`	`"Hello I'm @, here is my friend @!"`
`names`	`"Rob,Ken,Dave,"`
`insert_names(s, names)`	`"Hello I'm Rob, here is my friend Ken!"`

Name insertion

```
0  automaton insert(s: #u8, names: #u8) -> #u8 {
```

Name insertion

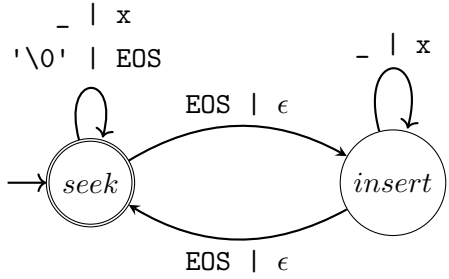
```
automaton insert(s: #u8, names: #u8) -> #u8 {
    entry seek() -> #u8 {
        let x = map(|c: u8| -> u8) {
            return c == '@' ? EOS : c;
        }, s);
    }{
        match x {
            EOS => insert(),
            '\0' => EOS, // accepting
            _ => x,
        }
    }
```

Name insertion

```
0  automaton insert(s: #u8, names: #u8) -> #u8 {
1      entry seek() -> #u8 {
2          let x = map(|c: u8| -> u8) {
3              return c == '@' ? EOS : c;
4          }, s);
5      }{
6          match x {
7              EOS => insert(),
8              '\0' => EOS, // accepting
9              _ => x,
10         }
11     }
12     mode insert() -> #u8 {
13         let x = map(|c: u8| -> u8 {
14             return c == ',' ? EOS : c;
15         }, names);
16     }{
17         match x {
18             EOS => seek(),
19             _ => x,
20         }
21     }
22 }
```

Name insertion

```
0  automaton insert(s: #u8, names: #u8) -> #u8 {
1      entry seek() -> #u8 {
2          let x = map(|c: u8| -> u8) {
3              return c == '@' ? EOS : c;
4          }, s);
5      }{
6          match x {
7              EOS => insert(),
8              '\0' => EOS, // accepting
9              _ => x,
10         }
11     }
12     mode insert() -> #u8 {
13         let x = map(|c: u8| -> u8 {
14             return c == ',' ? EOS : c;
15         }, names);
16     }{
17         match x {
18             EOS => seek(),
19             _ => x,
20         }
21     }
22 }
```



- an automaton produce an atom or stream of atoms
- accepting mode(s)
- bounce between modes until an atom can be produced
- **data-dependent reads on input streams** (asynchronous)

Conclusion

SIMD can offer significant performance gains, but programming can be difficult and text processing is only rarely considered (data dependency usually highly branching code).

VIR goal is to express text processing as successive SIMD steady-states sparated by control flow.

The goal is "as fast as you can" code, no RT / safety.

what's next ?

- I'm currently working on VIR evaluator called `circ8`
- explore more examples
- modes expressiveness
- compilation from VIR to LLVM and interface with Vectoid
- static code analysis, bounded memory, clock computation

Thank you