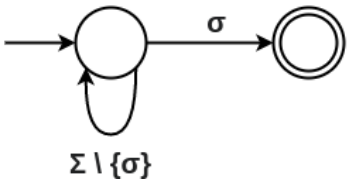
Vectorizing automata

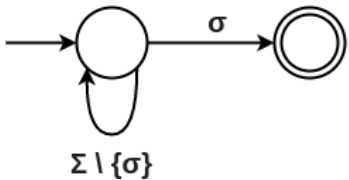
Claire Soyez-Martin,
joint work with Gilles Grimaud, Michaël Hauspie, Charles Paperman and Sylvain Salvati

June 16, 2021

The most simple example

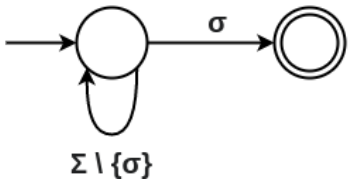


The most simple example



- A naive implementation:

The most simple example



- A naive implementation:

```
1 for u in w:
2     if u == σ:
3         return True
4 return False
```

How hard is it to actually implement

How hard is it to actually implement

It is a widely used function from glibc called memchr:

How hard is it to actually implement

It is a widely used function from glibc called memchr:

- A naive *C*-implementation: $\sim$ 4GB/s

How hard is it to actually implement

It is a widely used function from glibc called memchr:

- A naive *C*-implementation: $\sim$ 4GB/s
- The optimized glibc variant: $\sim$ 18GB/s

How hard is it to actually implement

It is a widely used function from glibc called memchr:

- A naive *C*-implementation: $\sim$ 4GB/s
- The optimized glibc variant: $\sim$ 18GB/s

How can such a *simple* function be optimized that much ?

How hard is it to actually implement

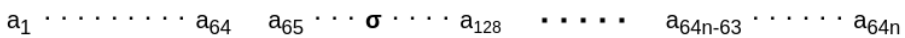
It is a widely used function from glibc called memchr:

- A naive *C*-implementation: $\sim$ 4GB/s
- The optimized glibc variant: $\sim$ 18GB/s

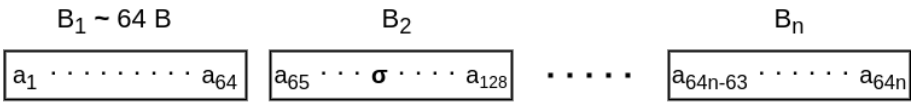
How can such a *simple* function be optimized that much ?

With vectorization!

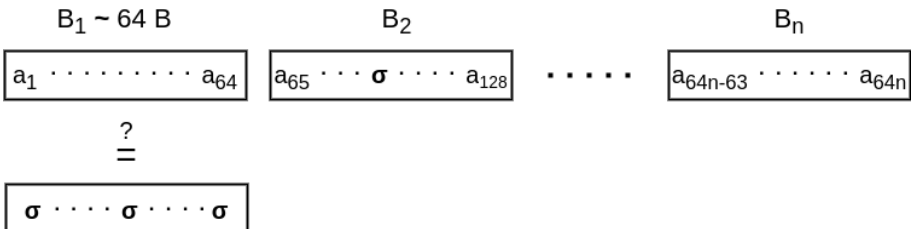
Memchr vectorized



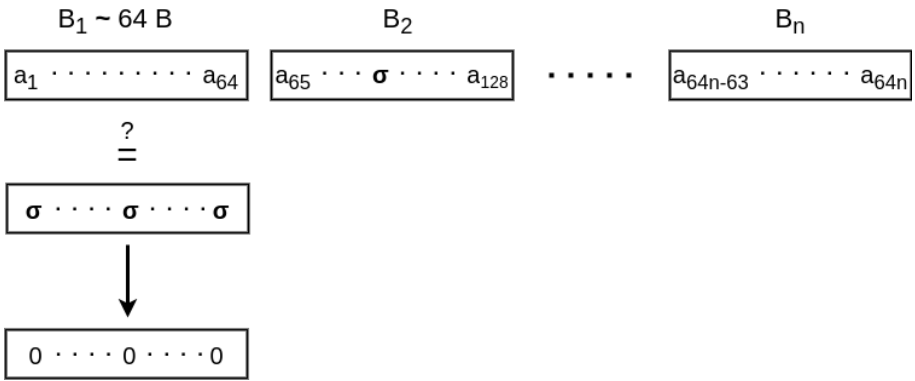
Memchr vectorized



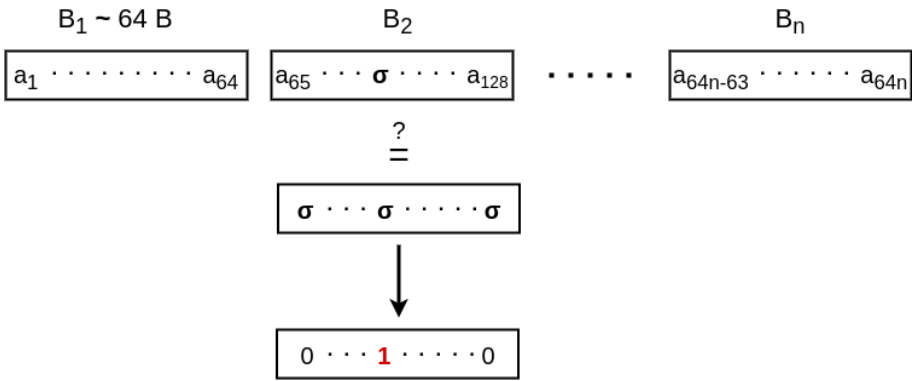
Memchr vectorized



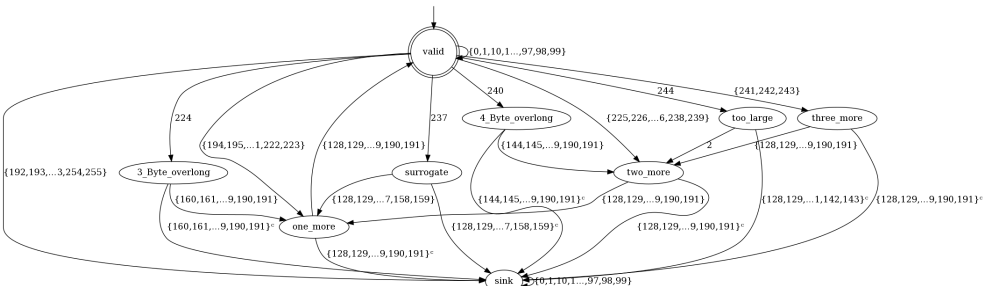
Memchr vectorized



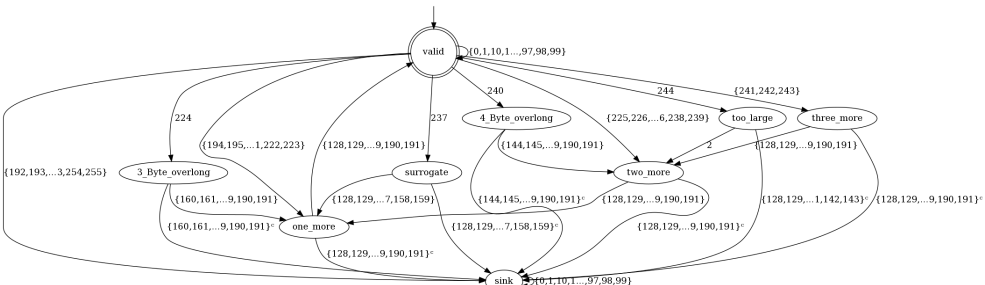
Memchr vectorized



A more complicated example: Validating UTF-8 [L20]

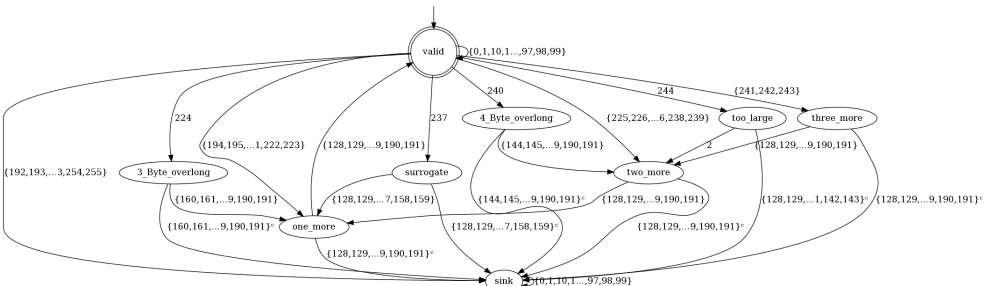


A more complicated example: Validating UTF-8 [L20]



Standard implementation: 2 to 4 GB/s depending on the input

A more complicated example: Validating UTF-8 [L20]



Standard implementation: 2 to 4 GB/s depending on the input

Results of Keiser and Lemire (2020): $\geq$ 12 GB/s

Summary about vectorization

- Based *SIMD* instructions: **S**ingle **I**nstruction **M**ultiple **D**ata
- It can do computations component-wise over blocks of inputs
- In between streaming and parallel computation

How to vectorize automata execution?

Monoids!

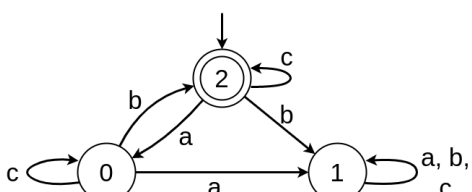
- A finite set of items with an **associative binary operation** and an **neutral element**

Monoids!

- A finite set of items with an **associative binary operation** and an **neutral element**
- It is possible to associate a monoid with an automaton. For example, by constructing its transition monoid

Monoids!

- A finite set of items with an **associative binary operation** and an **neutral element**
- It is possible to associate a monoid with an automaton. For example, by constructing its transition monoid

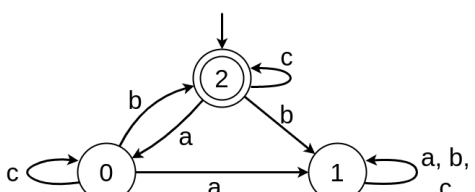


Monoids!

- A finite set of items with an **associative binary operation** and an **neutral element**
- It is possible to associate a monoid with an automaton. For example, by constructing its transition monoid

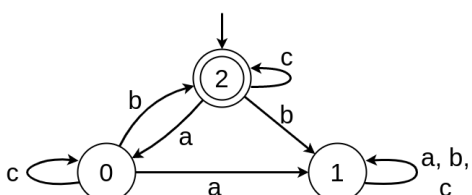
a: $0 \to 1,\ 1 \to 1,\ 2 \to 0$

b: $0 \to 2,\ 1 \to 1,\ 2 \to 2$



Monoids!

- A finite set of items with an **associative binary operation** and an **neutral element**
- It is possible to associate a monoid with an automaton. For example, by constructing its transition monoid



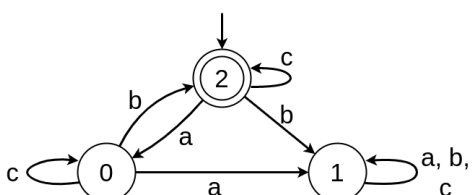
a: $0 \to 1,\ 1 \to 1,\ 2 \to 0$

b: $0 \to 2,\ 1 \to 1,\ 2 \to 1$

ab: $0 \to 1,\ 1 \to 1,\ 2 \to 2$

Monoids!

- A finite set of items with an **associative binary operation** and an **neutral element**
- It is possible to associate a monoid with an automaton. For example, by constructing its transition monoid



a: $0 \to 1,\ 1 \to 1,\ 2 \to 0$

b: $0 \to 2,\ 1 \to 1,\ 2 \to 1$

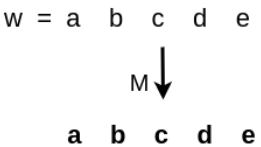
ab: $0 \to 1,\ 1 \to 1,\ 2 \to 2$

The mapping $\Sigma^* \to M$ is called the *transition morphism*.

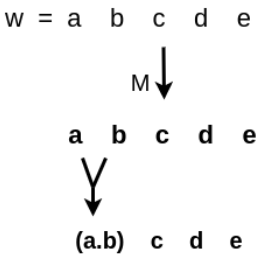
Monoids in parallel computation

w = a b c d e

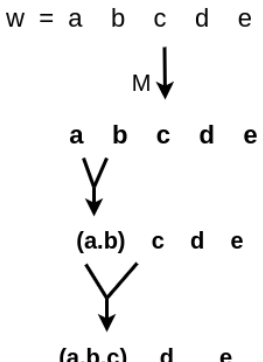
Monoids in parallel computation



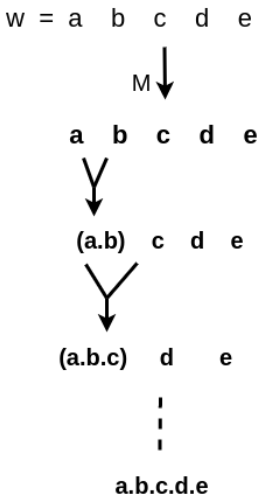
Monoids in parallel computation



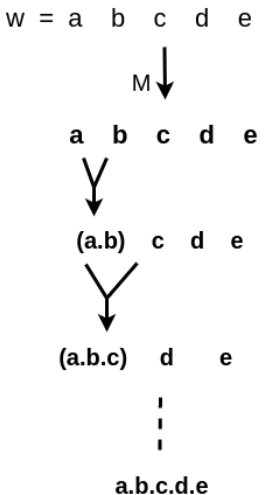
Monoids in parallel computation

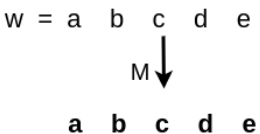


Monoids in parallel computation

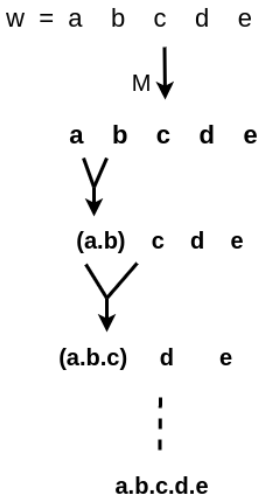


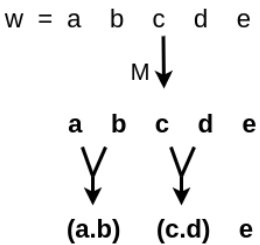
Monoids in parallel computation



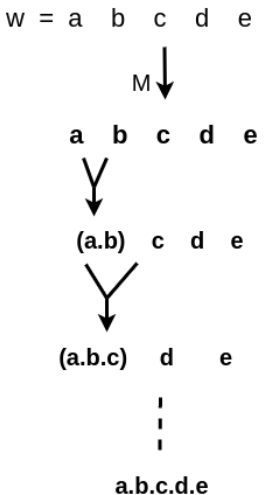


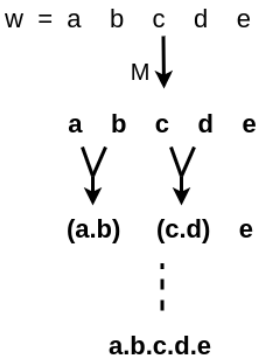
Monoids in parallel computation





Monoids in parallel computation





Are monoids computation vectorizable?

- How to execute the transition morphism on each cell of a block?

Are monoids computation vectorizable?

- How to execute the transition morphism on each cell of a block?
- How to aggregate horizontally the monoid values?

Are monoids computation vectorizable?

- How to execute the transition morphism on each cell of a block?
- How to aggregate horizontally the monoid values?

The general case seems hard, let's assume the monoids have some extra nice properties!

DA

A *(pseudo-)variety* defined by the equation: $\forall x, y, z \in M$,

$$(xyz)^\omega y (xyz)^\omega = (xyz)^\omega$$

where for any $s \in M$, ω is the least integer so that s^ω is idempotent

DA

A *(pseudo-)variety* defined by the equation: $\forall x, y, z \in M$,

$$(xyz)^\omega y (xyz)^\omega = (xyz)^\omega$$

where for any $s \in M$, ω is the least integer so that s^ω is idempotent

An automaton is said to be in **DA** its transition monoid is in **DA**.

DA

A *(pseudo-)variety* defined by the equation: $\forall x, y, z \in M$,

$$(xyz)^\omega y (xyz)^\omega = (xyz)^\omega$$

where for any $s \in M$, ω is the least integer so that s^ω is idempotent

An automaton is said to be in **DA** its transition monoid is in **DA**.

An automaton in **DA** can be recognized by a **turtle program**.

Turtle programs

- A generalization of memchr
- Base instruction: go to the next/previous letter until a letter from a set B is found

Turtle programs

- A generalization of memchr
- Base instruction: go to the next/previous letter until a letter from a set B is found

  The first a has a b somewhere before it

$$w = dcbcbdaea$$

Turtle programs

- A generalization of memchr
- Base instruction: go to the next/previous letter until a letter from a set B is found

  The first a has a b somewhere before it

$$w = dcbcbdaea$$

Turtle programs

- A generalization of memchr
- Base instruction: go to the next/previous letter until a letter from a set B is found

  The first a has a b somewhere before it

$$w = dcbcbdaea$$

Turtle programs

- A generalization of memchr
- Base instruction: go to the next/previous letter until a letter from a set B is found

  The first a has a b somewhere before it

  $$w = dcbcbdaea$$

- The program is **factorized**: we factorize some sub-turtle programs which have the same instructions

Our result

Recall that an automaton is in **DA** if its transition monoid is in **DA**.

Theorem

Let $\mathcal{A}$ be an automaton in **DA**. Let M be the transition monoid of $\mathcal{A}$. There exists a turtle program of size *linear* in $|M|$ that is *equivalent* to $\mathcal{A}$.

Our result

Recall that an automaton is in **DA** if its transition monoid is in **DA**.

Theorem

Let $\mathcal{A}$ be an automaton in **DA**. Let M be the transition monoid of $\mathcal{A}$. There exists a turtle program of size *linear* in $|M|$ that is *equivalent* to $\mathcal{A}$.

This theorem improves the state-of-the-art results (see [Kuf09]): previous proofs construct the turtle programs by enumerating formulas until some size depending on the size of the monoid.

Our result

Recall that an automaton is in **DA** if its transition monoid is in **DA**.

Theorem

Let $\mathcal{A}$ be an automaton in **DA**. Let M be the transition monoid of $\mathcal{A}$. There exists a turtle program of size *linear* in $|M|$ that is *equivalent* to $\mathcal{A}$.

Turtle programs are easy to execute over each block!

Conclusion

Our ongoing work:

- We are trying to vectorize efficiently automata which monoids are in **DA**

Conclusion

Our ongoing work:

- We are trying to vectorize efficiently automata which monoids are in **DA**
- We aim to extend our result to a broader class of automata: the automata that recognize starfree languages (also seen as the variety of aperiodic monoids)

Conclusion

Our ongoing work:

- We are trying to vectorize efficiently automata which monoids are in **DA**
- We aim to extend our result to a broader class of automata: the automata that recognize starfree languages (also seen as the variety of aperiodic monoids)
- This work is strongly tied to studies on circuit complexity (see Straubing's book)

Bibliography

[L20]: J. Keiser and D. Lemire, Validating UTF-8 In Less Than One Instruction Per Byte, 2020, arxiv

[Kuf09]: Manfred Kufleitner, Pascal Weil. On FO2 quantifier alternation over words. Mathematical Foundations of Computer Science 2009, Aug 2009, Slovakia. pp.513-524.