An Evaluation of Massively Parallel Algorithms for DFA Minimization

GandALF 2024

Jan Martens[1,2] and Anton Wijs[1]

June 19, 2024

j.j.m.martens@liacs.leidenuniv.nl

[1]Eindhoven University of Technology

[2]Leiden Institute of Advanced Computer Science

Motivation

Graphics processing units GPUs are:

- incredibly powerful devices,
- made for regular problems
  (i.e. matrix multiplication),
- very parallel,
- hard to program irregular problems.

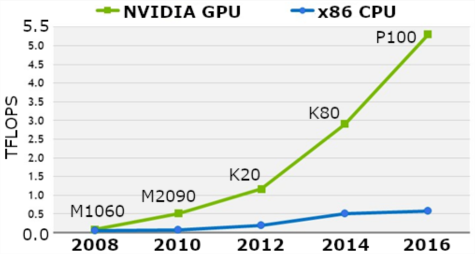


Fig. Computational power of GPUs.

Motivation

Graphics processing units GPUs are:

- incredibly powerful devices,
- made for regular problems (i.e. matrix multiplication),
- very parallel,
- hard to program irregular problems.

Goal:

Utilize the power of the GPU for generic computing tasks.

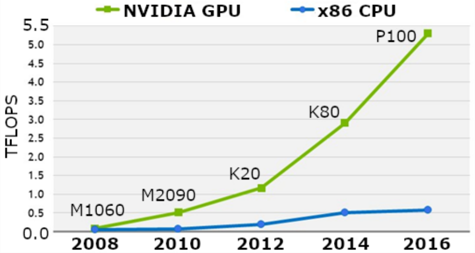


Fig. Computational power of GPUs.

Motivation

Graphics processing units GPUs are:

- incredibly powerful devices,
- made for regular problems (i.e. matrix multiplication),
- very parallel,
- hard to program irregular problems.

Goal:

Utilize the power of the GPU for generic computing tasks.

Fig. NVIDIA share price.

Outline

1. Motivation
2. Parallel complexity
3. *DFA minimization*
4. Three ways to compute minimal DFA
    - Partition refinement
    - Sorting
    - Transitive closure
    - **Bonus** PR with partial transitive closure.
5. Evaluation

Computational model – PRAM

The **Parallel Random Access Machine (PRAM)** is an extension on the RAM.

PRAM

- Unbounded collection of processors $P_0, P_1, P_2, \ldots$
- Unbounded collection of common memory cells the processors can access
- Each processor P_i has access to its index i
- Processors run the same program **synchronously**

A PRAM program contains a function $\mathcal{P} : \mathbb{N} \to \mathbb{N}$ defining how many processes are started, based on the size of the input.

Computational model – Complexity

A PRAM program comes with two complexity measures:

- Time – the number of sequential steps the PRAM takes,
- Work – the total work performed.

Work is equal to $P * T$ where P is the number of processors, and T the time.

Nick's class (NC)
Problems that can be solved in $\mathcal{O}(log^i n)$ time with $\mathcal{O}(n^c)$ processors for some $i, c \in \mathbb{N}$.

Open question: $P \stackrel{?}{=} NC$

Computational model – Complexity

A PRAM program comes with two complexity measures:

- Time – the number of sequential steps the PRAM takes,
- Work – the total work performed.

Work is equal to $P * T$ where P is the number of processors, and T the time.

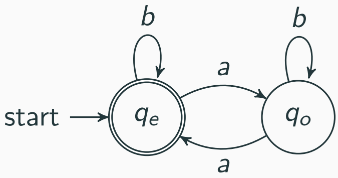
Nick's class (NC)
Problems that can be solved in $\mathcal{O}(log^i n)$ time with $\mathcal{O}(n^c)$ processors for some $i, c \in \mathbb{N}$.

Open question: $P \stackrel{?}{=} NC$

$P \neq NC \implies$ there are some inherently sequential *P-complete* problems.

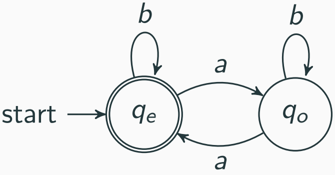
Deterministic Finite Automata

A DFA is a five tuple $\mathcal{A} = (Q, \Sigma, \delta, F, q_0)$



DFA minimization

Deterministic Finite Automata
A DFA is a five tuple $\mathcal{A} = (Q, \Sigma, \delta, F, q_0)$

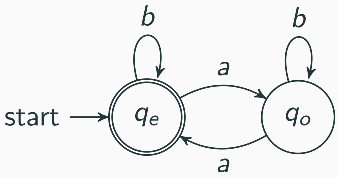


DFA minimization
Given DFAs $\mathcal{A}$ compute a minimal automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A})'$.

1. Remove not reachable states.
2. Merge equivalent states,

DFA minimization

Deterministic Finite Automata
A DFA is a five tuple $\mathcal{A} = (Q, \Sigma, \delta, F, q_0)$



DFA minimization
Given DFAs $\mathcal{A}$ compute a minimal automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A})'$.

1. Remove not reachable states.
2. **Merge equivalent states,** For DFAs equivalence corresponds to bisimilarity.

Complexity scenario

Computing bisimilarity quotient

- For non-deterministic systems (LTSs) P-complete.

Parallel setting
Parallel algorithms for these problem often also **partition refinement (PR)**.

Complexity scenario

Computing bisimilarity quotient

- For non-deterministic systems (LTSs) P-complete.
- For DFAs in NC. ✓

Sequential setting
For DFAs usually computed with [Hopcroft 1971] $\mathcal{O}(n \log n)$.

For LTSs a similar approach is used [Paige& Tarjan 1987].

Parallel setting
Parallel algorithms for these problem often also **partition refinement (PR)**.

Complexity scenario

Computing bisimilarity quotient

- For non-deterministic systems (LTSs) P-complete.
- For DFAs in NC. ✓

Sequential setting
For DFAs usually computed with [Hopcroft 1971] $\mathcal{O}(n \log n)$.

For LTSs a similar approach is used [Paige& Tarjan 1987].

Parallel setting
Parallel algorithms for these problem often also **partition refinement (PR)**.

Lower bound – parallel
Parallel partition refinement algorithms are $\Omega(n)$ [Groote, M. & De Vink 2023].

Complexity scenario

Computing bisimilarity quotient

- For non-deterministic systems (LTSs) P-complete.
- For DFAs in

Sequential setti
For DFAs usually

For LTSs a simila

Why use partition refinement for DFAs!?

Parallel setting
Parallel algorithms for these problem often also **partition refinement (PR)**.

Lower bound – parallel
Parallel partition refinement algorithms are $\Omega(n)$ [Groote, M. & De Vink 2023].

Partition refinement

Maintain a partition of states which are different $\pi = \{F, Q \setminus F\}$

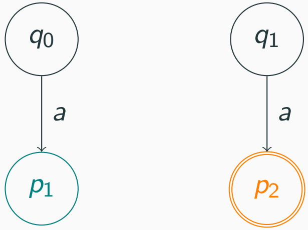
Base case:

Partition refinement

Maintain a partition of states which are different $\pi = \{F, Q \setminus F\}$

Base case:

Refinement: $\pi = \{\{q_0, q_1\}, \{p_1\}.\{p_2\}\} \mapsto \pi' = \{\{q_0\}, \{q_1\}, \{p_1\}.\{p_2\}\}$

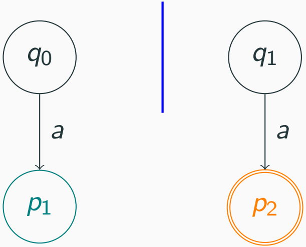


Partition refinement

Maintain a partition of states which are different $\pi = \{F, Q \setminus F\}$

Base case:

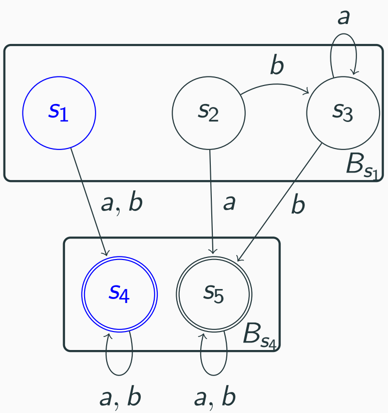
Refinement: $\pi = \{\{q_0, q_1\}, \{p_1\}.\{p_2\}\} \mapsto \pi' = \{\{q_0\}, \{q_1\}, \{p_1\}.\{p_2\}\}$



Algorithm I – Partition refinement `naivePR`[1]

Step 0: Initialize

$block$ :: Array$[n]$ of type Q
new_leader :: Array$[n]$ of type Q
Select initial leader states $q_f \in F$ and $q_n \in Q \setminus F$
do in parallel for $q \in Q$
    $block[q] := (q \in F\ ?\ q_f : q_n)$
$stable :=$ `false`



Step 0: Initialize leaders B_{s_4}, B_{s_1}

[1] Based on [M. , Groote, van der Haak, Hijma& Wijs 2021]

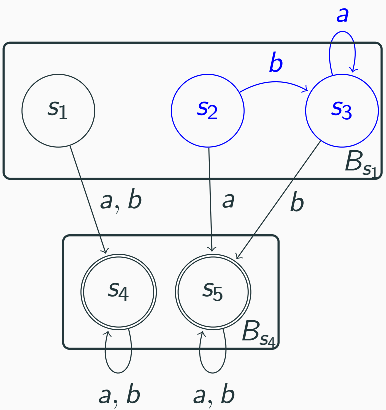
Algorithm I – Partition refinement `naivePR`[1]

Step 1: In parallel compare to leader

1: **do in parallel for** $q, a \in Q \times \Sigma$
2: **if** $block[\delta(q,a)] \neq block[\delta(block[q],a)]$ **then**
3: $new\_leader[block[q]] := q$

Step 2: Split states from leader

1: **do in parallel for** $q, a \in Q \times \Sigma$
2: **if** $block[\delta(q,a)] \neq block[\delta(block[q],a)]$ **then**
3: $block[q] := new\_leader[block[q]]$



Step 1: Compare to leader

[1] Based on [M. , Groote, van der Haak, Hijma& Wijs 2021]

Algorithm I – Partition refinement `naivePR`[1]

Step 1: In parallel compare to leader

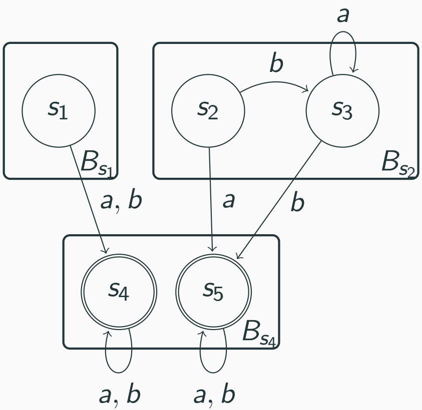
---

do in parallel for $q, a \in Q \times \Sigma$
  if $\mathit{block}[\delta(q,a)] \neq \mathit{block}[\delta(\mathit{block}[q],a)]$ **then**
    $\mathit{new\_leader}[\mathit{block}[q]] := q$

Step 2: Split states from leader

---

do in parallel for $q, a \in Q \times \Sigma$
  if $\mathit{block}[\delta(q,a)] \neq \mathit{block}[\delta(\mathit{block}[q],a)]$ **then**
    $\mathit{block}[q] := \mathit{new\_leader}[\mathit{block}[q]]$

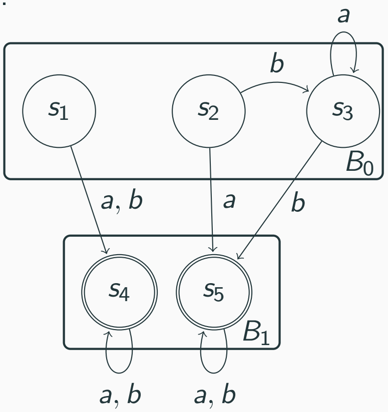


Step 2: Split into new blocks.

[1] Based on [M. , Groote, van der Haak, Hijma& Wijs 2021]

Algorithm II – Partition refinement SortPR[2]

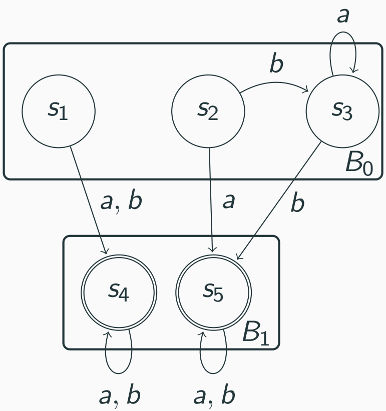
Idea: Compare all different target blocks (**signatures**).



[2]Based on [Ravikumar & Xiong 1996]

Algorithm II – Partition refinement SortPR[2]

Data structure	s_1	s_2	s_3	s_4	s_5
block	B_0	B_0	B_0	B_1	B_1
$block_a$	B_1	B_1	B_0	B_1	B_1
$block_b$	B_1	B_0	B_1	B_1	B_1

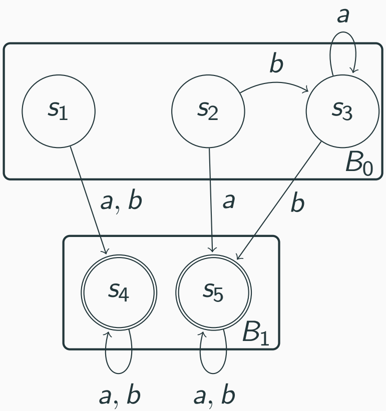


[2] Based on [Ravikumar & Xiong 1996]

Algorithm II – Partition refinement SortPR[2]

Data structure	s_1	s_2	s_3	s_4	s_5
block	B_0	B_0	B_0	B_1	B_1
$block_a$	B_1	B_1	B_0	B_1	B_1
$block_b$	B_1	B_0	B_1	B_1	B_1

Sorted Data	s_3	s_2	s_1	s_4	s_5
block	B_0	B_0	B_0	B_1	B_1
$block_a$	B_0	B_1	B_1	B_1	B_1
$block_b$	B_1	B_0	B_1	B_1	B_1

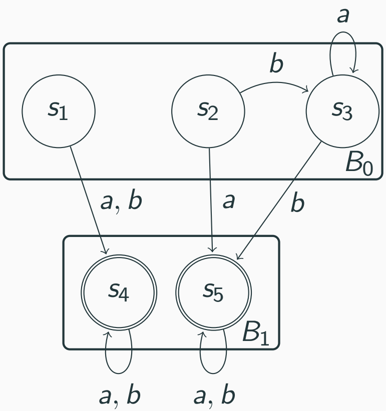


[2] Based on [Ravikumar & Xiong 1996]

Algorithm II – Partition refinement `SortPR`[2]

Data structure	s_1	s_2	s_3	s_4	s_5
block	B_0	B_0	B_0	B_1	B_1
block$_a$	B_1	B_1	B_0	B_1	B_1
block$_b$	B_1	B_0	B_1	B_1	B_1

Sorted Data	s_3	s_2	s_1	s_4	s_5
block	B_0	B_0	B_0	B_1	B_1
block$_a$	B_0	B_1	B_1	B_1	B_1
block$_b$	B_1	B_0	B_1	B_1	B_1
scan	0	1	1	1	0

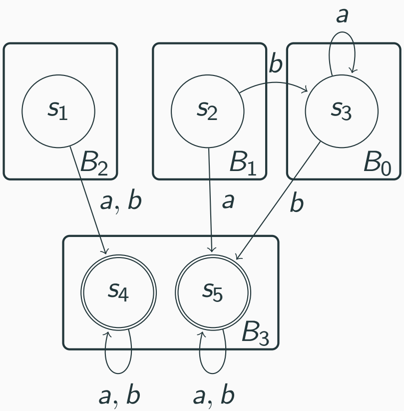


[2] Based on [Ravikumar & Xiong 1996]

Algorithm II – Partition refinement `SortPR`[2]

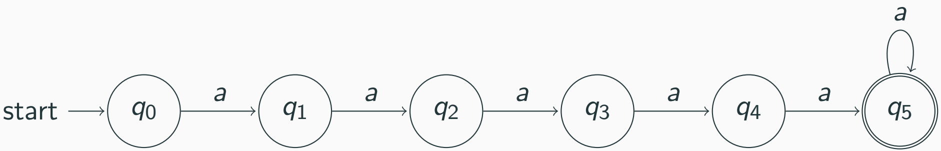
Data structure	s_1	s_2	s_3	s_4	s_5
block	B_0	B_0	B_0	B_1	B_1
$block_a$	B_1	B_1	B_0	B_1	B_1
$block_b$	B_1	B_0	B_1	B_1	B_1

Sorted Data	s_3	s_2	s_1	s_4	s_5
block	B_0	B_0	B_0	B_1	B_1
$block_a$	B_0	B_1	B_1	B_1	B_1
$block_b$	B_1	B_0	B_1	B_1	B_1
scan	0	1	1	1	0
prefixSum	0	1	2	3	3



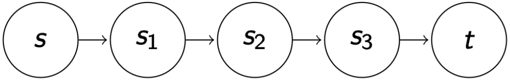
[2] Based on [Ravikumar & Xiong 1996]

Partition refinement – Parallel worst case



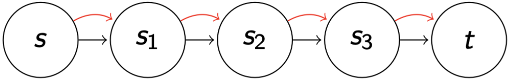
Parallel reachability

$Reach$ :: Array$[n][n]$ of type $\mathbb{B}$
do in parallel for $s, t \in V$
    if $(s, t) \in E$ **then**
        $Reach[s][t]$:= `true`
while $\neg stable$ **do**
    do in parallel for $s, t, u \in V$
        if $Reach[s][t] \& Reach[t][u]$ **then**
            $Reach[s][u]$:= `true`



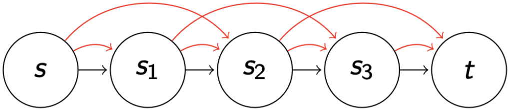
Parallel reachability

$Reach :: \text{Array}[n][n]$ of type $\mathbb{B}$
do in parallel for $s, t \in V$
  if $(s, t) \in E$ **then**
    $Reach[s][t] := \texttt{true}$
while $\neg stable$ **do**
  do in parallel for $s, t, u \in V$
    if $Reach[s][t] \& Reach[t][u]$ **then**
      $Reach[s][u] := \texttt{true}$



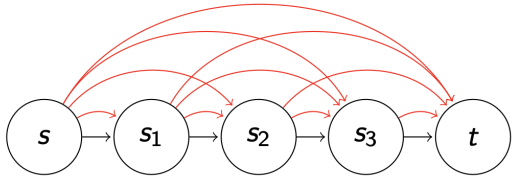
Parallel reachability

Reach :: Array$[n][n]$ of type $\mathbb{B}$
do in parallel for $s, t \in V$
  if $(s, t) \in E$ **then**
    $Reach[s][t] := \texttt{true}$
while $\neg stable$ **do**
  do in parallel for $s, t, u \in V$
    if $Reach[s][t] \& Reach[t][u]$ **then**
      $Reach[s][u] := \texttt{true}$



Parallel reachability

```
Reach :: Array[n][n] of type 𝔹
do in parallel for s, t ∈ V
    if (s, t) ∈ E then
        Reach[s][t] := true
while ¬stable do
    do in parallel for s, t, u ∈ V
        if Reach[s][t] & Reach[t][u] then
            Reach[s][u] := true
```



Complexity – overview

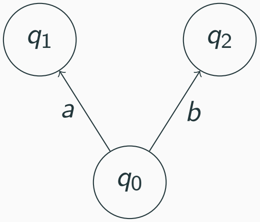
	deterministic (DFAs)	non-deterministic
class	NC	P-complete
Best parallel run-time	$\mathcal{O}(\log^2 n)$	$\Omega(n)$
parallel PR time	n	n
Sequential work	$\mathcal{O}(n \log n)$	$\mathcal{O}(m \log n)$

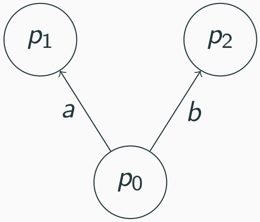
Algorithm III – Transitive Closure `trans`

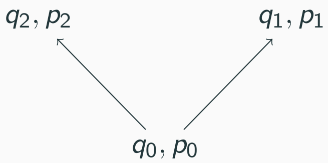
DFA in NC – [Cho & Huynh 1992 IPL]
DFA minimization is in NC by doing logarithmic transitive closure.

1. Construct a graph $V = Q \times Q$,
   and $E = \{((q,p),(q',p')) \mid q \to^a q' \text{ and } p \to^a p' \text{ for some } a \in \Sigma\}$.
2. Label all nodes $(q,q'),(q',q)$ with $\perp$ if
   $$q \in F \text{ and } q' \notin F$$
3. Compute parallel reachability,
4. Now $q \neq q' \iff (q,q') \to \perp$.

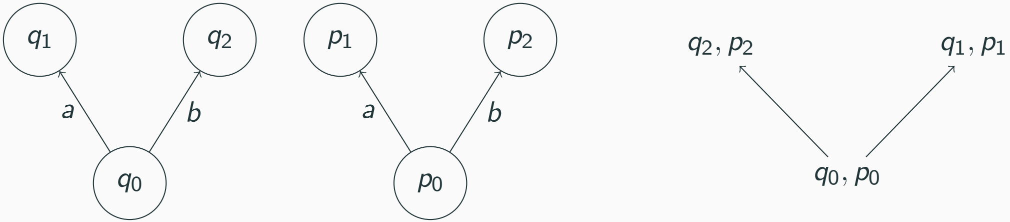
Algorithm III – `trans`







Algorithm III – `trans`



Note: The graph will have size n^2.

Overview

`naivePR`

- ✓ $\mathcal{O}(1)$ time iteration,
- ✓ Best time complexity,
- × Split block in two,
- × Not sub-linear.

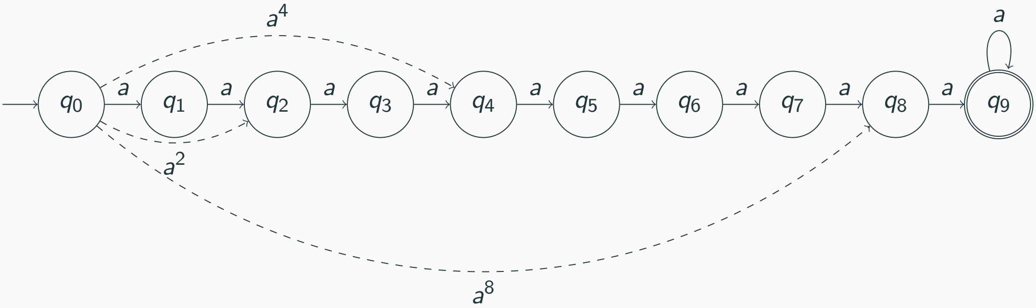
`sortPR`

- ✓ Split blocks in multiple,
- ✓ Native GPU operation,
- × Iteration time,
- × slow worse case.

`trans`

- ✓ Logarithmic iterations,
- × Amount of resources
  - Memory $O(n^4)$,
  - Processors $\mathcal{O}(n^5)$.

New algorithm – `transPR`
Idea: Build new DFA with more alphabet letters.



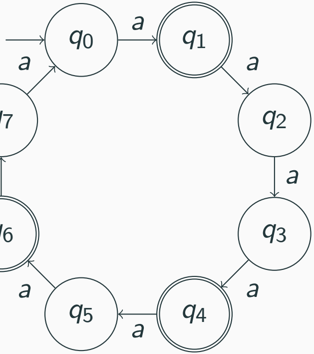
Algorithm IV – `transPR`

Pro's and cons

- ✓ Uses only $n \log n$ processors,
- ✓ Get (partial) transitive closure,
- × Only in very specific structures.

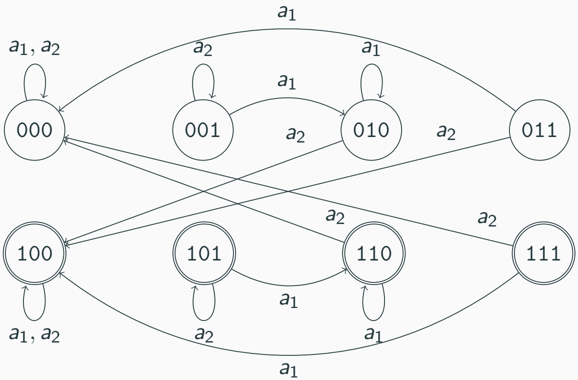
Evaluation
We evaluate on the following benchmarks

- Fibonacci automata $\texttt{Fib}_n$,
- the Bit splitter $\mathcal{B}_n$, and
- The very large transition system (VLTSs) benchmarks.



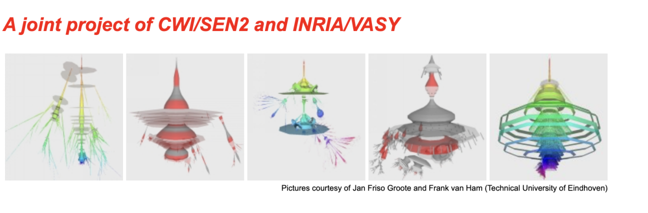
Evaluation
We evaluate on the following benchmarks

- Fibonacci automata $\texttt{Fib}_n$,
- the Bit splitter $\mathcal{B}_n$, and
- The very large transition system (VLTSs) benchmarks.



Evaluation
We evaluate on the following benchmarks

- Fibonacci automata $\texttt{Fib}_n$,
- the Bit splitter $\mathcal{B}_n$, and
- The very large transition system (VLTSs) benchmarks.



Evaluation I

Point 1/4: The logarithmic algorithm `trans` is not feasible (yet).

Name	N	Iterations	Time (ms)	Memory(Mb)	#threads
$\texttt{Fib}_4$	8	3	0.3	0	589,824
$\texttt{Fib}_5$	13	4	0.7	0	6,230,016
$\texttt{Fib}_6$	21	5	7.8	0	88,510,464
$\texttt{Fib}_7$	34	5	159.9	0	1,620,545,536
$\texttt{Fib}_8$	55	6	3,034.9	10	27,955,840,000
$\texttt{Fib}_9$	89	7	66,846.7	60	498,865,340,416
$\texttt{Fib}_{10}$	144	t/o	t/o	412	8,943,640,510,464

Evaluation II

Point 2/4: For the Fibonacci automata `transPR` works great.

	Benchmark metrics	Times (ms)		Iterations	
Name	N	`naivePR`	`transPR`	`naivePR`	`transPR`
$\texttt{Fib}_{20}$	17,711	308.8	1.7	17,710	14
$\texttt{Fib}_{21}$	28,657	494.2	2.4	28,656	25
$\texttt{Fib}_{22}$	46,368	778.7	4.1	46,367	61
$\texttt{Fib}_{23}$	75,025	1,241.3	8.0	75,024	101
$\texttt{Fib}_{24}$	121,393	2,006.7	12.5	121,392	104
$\texttt{Fib}_{25}$	196,418	3,251.3	18.3	196,417	138
$\texttt{Fib}_{26}$	317,811	5,277.8	49.8	317,810	102
$\texttt{Fib}_{27}$	514,229	8,607.7	96.1	514,228	268
$\texttt{Fib}_{28}$	832,040	22,723.0	178.4	832,039	299
$\texttt{Fib}_{29}$	1,346,269	59,510.8	726.9	1,346,268	755
$\texttt{Fib}_{30}$	2,178,309	141,601.0	1,109.3	2,178,308	914

Evaluation III

Point 3/4: In worst case scenarios `naivePR` works best.

	Benchmark metrics	Times (ms)		Iterations	
Name	N	`naivePR`	`sortPR`	`naivePR`	`sortPR`
$\texttt{Fib}_{24}$	121,393	2,006.7	34,793.1	121,392	121,392
$\texttt{Fib}_{25}$	196,418	3,251.3	64,411.7	196,417	196,417
$\texttt{Fib}_{26}$	317,811	5,277.8	178,367.4	317,810	317,810
$\texttt{Fib}_{27}$	514,229	8,607.7	t/o	514,228	t/o
$\texttt{Fib}_{28}$	832,040	22,723.0	t/o	832,039	t/o
$\texttt{Fib}_{29}$	1,346,269	59,510.8	t/o	1,346,268	t/o
$\texttt{Fib}_{30}$	2,178,309	141,601.0	t/o	2,178,308	t/o
$\mathcal{B}_{19}$	524,288	9.6	235.7	18	18
$\mathcal{B}_{20}$	1,048,576	19.3	520.2	19	19
$\mathcal{B}_{21}$	2,097,152	39.8	1,148.6	20	20
$\mathcal{B}_{22}$	4,194,304	82.6	2,538.5	21	21
$\mathcal{B}_{23}$	8,388,608	170.3	5,612.7	22	22

Evaluation IV

Point 4/4: In some (real world) examples in VLTS `sortPR` works best.

	Benchmark metrics	Times (ms)		Iterations	
Name	N	`naivePR`	`sortPR`	`naivePR`	`sortPR`
cwi_1_2	4,448	5.4	66.7	308	38
vasy_1112_5290	1,112,491	135.4	386.8	246	4
vasy_157_297	157,605	455.1	1,736.3	1,049	27
vasy_386_1171	355,790	36.9	489.4	58	8
vasy_574_13561	574,058	2,332.2	976.5	2,351	5
vasy_6120_11031	3,190,785	13,186.6	21,886.0	2,373	21
vasy_65_2621	65,538	2,591.8	38.3	36,575	4
vasy_66_1302	209,791	42,864.9	96.0	179,861	8
vasy_69_520	74,958	7,223.0	124.2	49,723	12
vasy_720_390	87,741	176.0	57.1	2,936	5
vasy_83_325	393,147	162,495.0	1,074.4	173,218	19

Conclusions

Recap
We studied massive parallel algorithms for DFA minimization on GPUs:

- Depending on the structure of the automaton either:
  - `naivePR` many iterations, few splits per iteration,
  - `sortPR` many new reasons to split blocks.
- Despite complexity bounds, we find partition refinement algorithms work best.

Conclusions

Recap
We studied massive parallel algorithms for DFA minimization on GPUs:

- Depending on the structure of the automaton either:
  - `naivePR` many iterations, few splits per iteration,
  - `sortPR` many new reasons to split blocks.
- Despite complexity bounds, we find partition refinement algorithms work best.

Future work
Close gap in work and time complexity.

- Heuristic or randomized algorithms, e.g.
  - Expand pre-processing,
  - Reachability.
- Work lowerbounds for logarithmic algorithms.

Thanks!