Accelerated Verification - part 2

Anton Wijs

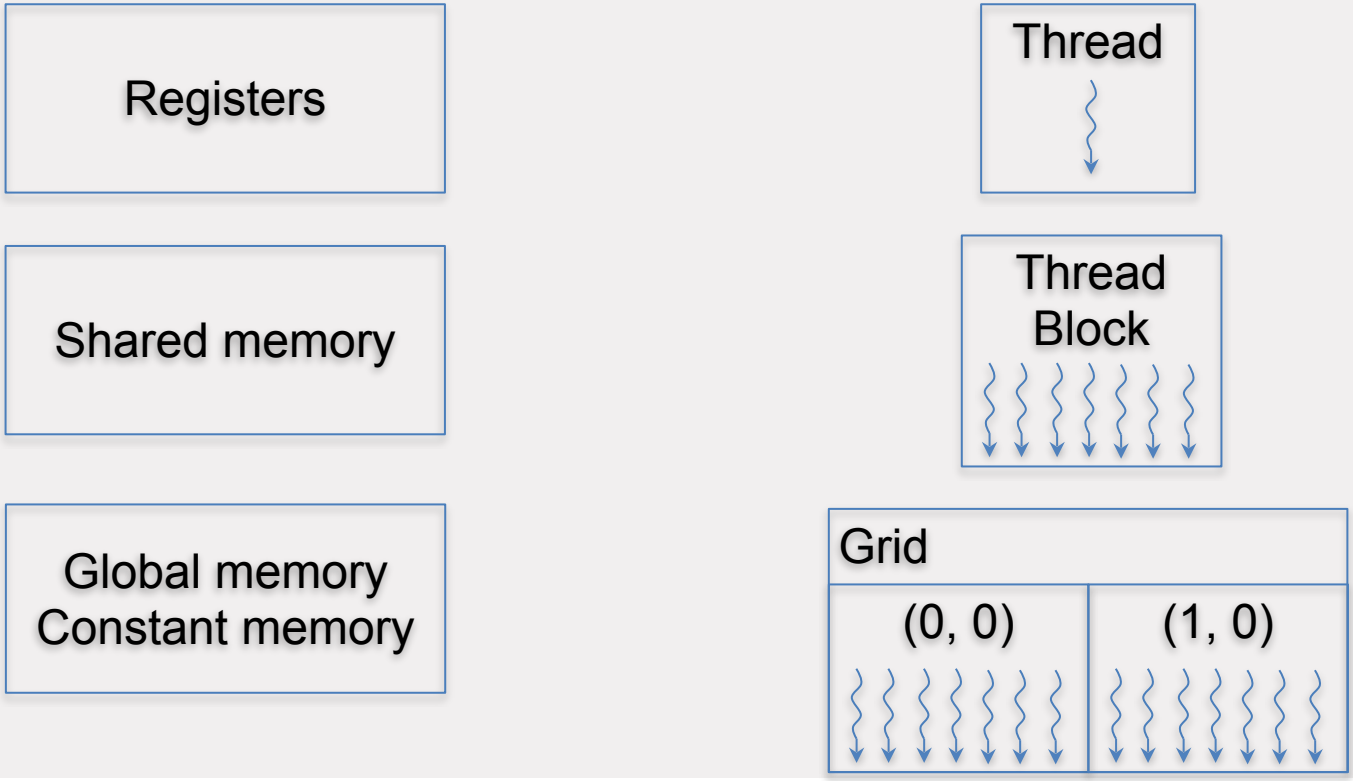
VTSA Summer school 2024 / 8 & 9 July

Schedule 9 July 2024

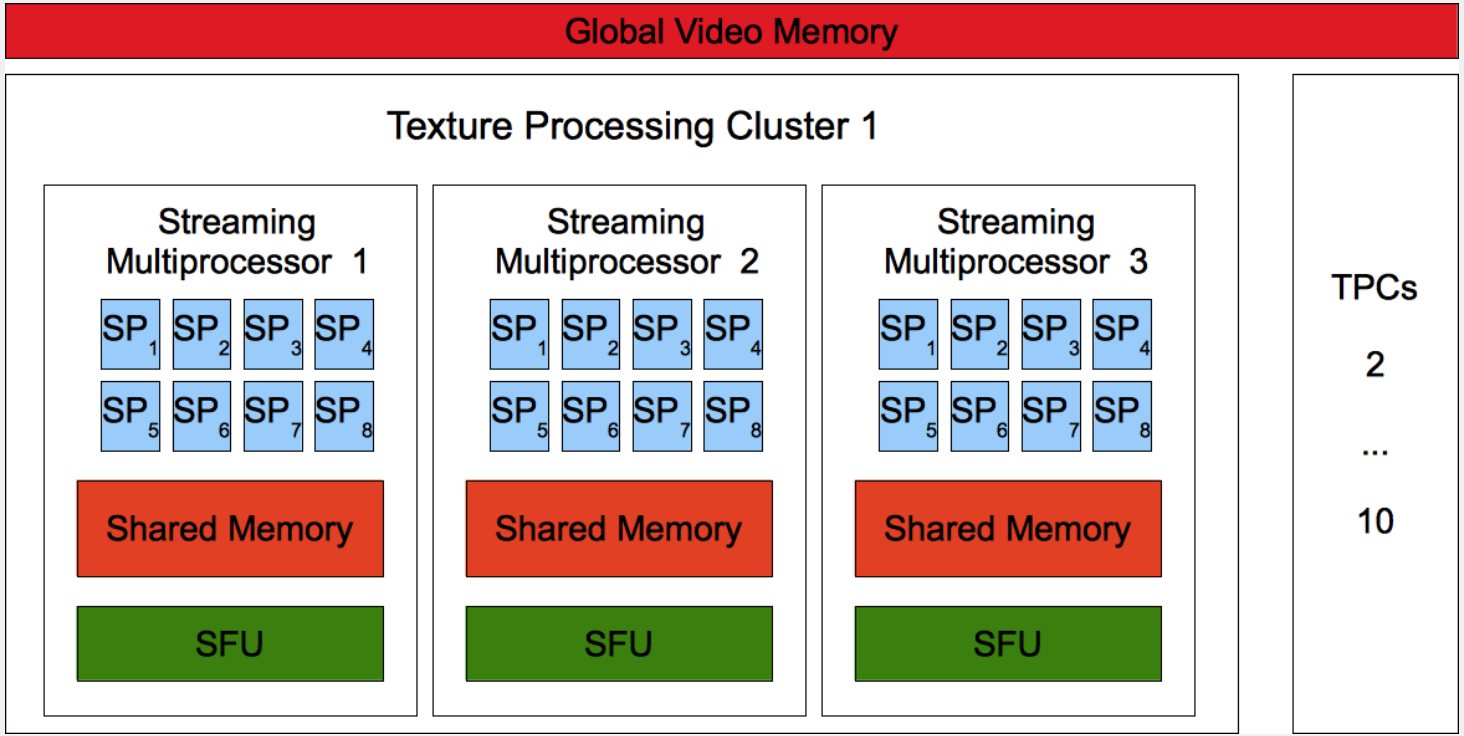
- **09:00 – 09:15** CUDA Programming model Part 3
- **09:15 – 10:00** 4th Hands-on Session + Solution
- **10:00 – 11:00** *GPUexplore*: explicit-state model checking with GPUs
- **11:00 – 11:30** Beyond the CUDA Programming model
- **11:30 – 12:00** 5th Hands-on Session + Solution
- **12:00 – 12:25** Warp-centric programming and atomics in *GPUexplore*
- **12:25 – 12:30** Wrap-up

CUDA Programming model
Part 3

CUDA memory hierarchy



Hardware overview



Memory space: Registers

- Example:

```
__global__ void matmul_kernel(float *C, float *A, float *B) {
    int tx = threadIdx.x;      //local variable in registers
    float local_sum[4];        //small compile-time sized array in registers
```

- **Registers**
  - Thread-local scalars or small constant size arrays are stored as registers
  - Implicit in the programming model
  - Behaviour is very similar to normal local variables
  - Not persistent, after the kernel has finished, values in registers are lost

Memory space: Global

- Example:

```
__global__ void matmul_kernel( float *C,   //C points to global memory
                               float *A,   //A points to global memory
                               float *B)   //B points to global memory
{
```

- **Global memory**
  - Allocated by the host program using `cudaMalloc()`
  - Initialized by the host program using `cudaMemcpy()` or previous kernels
  - Persistent, the values in global memory remain across kernel invocations
  - Not coherent, writes by other threads will not be visible until kernel has finished

Memory space: Constant

```
__constant__ float filter[filter_width * filter_height]; //initialized by a host function
__global__ void convolution_kernel(float *output, float *input) {
   ...
   for (j = 0; j < filter_height; j++) {
      for (i = 0; i < filter_width; i++) {
         sum += input[y + j][x + i] *
                filter[j * filter_width + i]; //index j and i do not depend on threadIdx (x and y)
      }
   }
```

- **Constant memory**
  - Statically defined by the host program using `__constant__` qualifier
  - Defined as a global variable
  - Initialized by the host program using `cudaMemcpyToSymbol()`
  - Read-only to the GPU, cannot be accessed directly by the host
  - Values are cached in a special cache optimized for broadcast access by multiple threads simultaneously, access should not depend on `threadIdx`

Memory space: Shared

```
__global__ void histogram(int *output, int *values, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    __shared__ int sh_output[NUM_BINS];                    //declare shared memory array
    if(i < n) {
        int bin = values[i];
        atomicAdd(&sh_output[bin], 1);                         //increment bin in shared memory
        __syncthreads();                                   //wait for all threads
    ...
```

- **Shared memory**
  - Variables have to be declared using `__shared__` qualifier, size known at compile time
  - In the scope of thread block, all threads in a thread block see the same piece of memory
  - Not initialised, threads have to fill shared memory with meaningful values
  - Not persistent, after the kernel has finished, values in shared memory are lost
  - Not coherent, `__syncthreads()` is required to make writes visible to other threads within the thread block

Shared memory: Example

```
__global__ void transpose(int h, int w, float* output, float* input) {
    int i = threadIdx.y + blockIdx.y * block_size_y;
    int j = threadIdx.x + blockIdx.x * block_size_x;

    __shared__ float sh_mem[block_size_y][block_size_x];      //declare shared memory array

    if (j < w && i < h) {
            sh_mem[threadIdx.y][threadIdx.x] = input[i*w+j];   //fill shared with values from global
    }
    __syncthreads();                                          //wait for all thread in block

    i = threadIdx.x + blockIdx.y * block_size_y;
    j = threadIdx.y + blockIdx.x * block_size_x;
    if (j < w && i < h) {
            output[j*h+i] = sh_mem[threadIdx.x][threadIdx.y];  //store to global using shared memory
    }
}
```

TU/e

Fourth hands-on session

- Go to folder **`4-reduction_fast`**, look at the source files
- Task:
  - Implement the kernel for reduction again, this time in such a way that shared memory is used to sum the per-thread partial sums into a single per-thread block partial sum
- Hints:
  - The number of thread blocks does not depend on n. All threads from all blocks first iterate (collectively) over the problem size (n) to obtain a per-thread partial sum
  - Within the thread block the per-thread partial sums are to be combined to a per-thread block partial sum
  - Each thread block stores its partial sum to `out_array[blockIdx.x]`
  - The kernel is called twice, the second kernel is executed with only one thread block to combine all per-block partial sums to a single sum

Solution

```
...
__shared__ float sh_mem[block_size_x];              //declare shared memory array
sh_mem[ti] = sum;                                   //store thread-local partial sum
__syncthreads();                                    //wait for all threads in the block

#pragma unroll
for (unsigned int s=block_size_x/2; s>0; s/=2) {    //iterate with s: 128, 64, 32, ..., 1
    if (ti < s) {                                   //threads with id < s
        sh_mem[ti] += sh_mem[ti + s];               //add partial sum of thread 's' away from ti
    }
    _syncthreads();                                 //wait for all threads in the block
}

if (ti == 0) {
    out_array[blockIdx.x] = sh_mem[0];              //store the per-block partial sum in global
}
```

TU/e

Revisiting SCC detection (BFS)

Algorithm 4 GPU-FWDBFS with local caching

Require: number of iterations *NrIters*
Ensure: *NrIters* local BFS iterations from the given search frontier have been performed

```
    extern volatile __shared__ unsigned int cache []
 2: <initialise cache>
    for (i ← Global-ThreadId; i < |V|; i ← i + NrOfThreads) do
 4:     srcinfo ← offsets[i]
        if inFrontier(srcinfo) then
 6:         offsets[i] ← movetoExplored(srcinfo)
            explore(srcinfo)
 8: for (iter ← 1; iter < NrIters; iter ++) do
        for j ← ThreadId; j < cachesize; j ← j + BlockSize do
10:         i ← cache[j]
            if i ≠ empty then
12:             cache[i] ← empty
                srcinfo ← offsets[i]
14:             if inFrontier(srcinfo) then
                    offsets[i] ← movetoExplored(srcinfo)
16:                 explore(srcinfo)
```

EXPLORE involves storing state ID in shared memory cache

TU/e

GPUexplore: explicit-state model checking with GPUs

Joint work with Nathan Cassee, Dragan Bošnački, Jan Heemstra, Muhammad Osama, Thomas Neele, Rik van Spreuwel, Jaco van de Pol
(TACAS 2014, 2023, 2024, CAV 2016, FM 2016, ATVA 2016, GaM 2017, SPIN 2023)

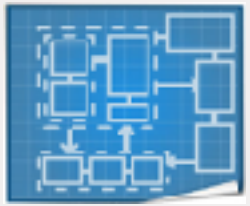
TU/e

Correctness of Concurrent Systems

- Distributed, concurrent systems common-place, but very difficult to develop
  - network applications, communication protocols, multi-threaded applications
- Systems may contain bugs such as *deadlocks* and *livelocks*
  - *Deadlock:* computation not finished, but system cannot progress
  - *Livelock:* system repeats computation steps without progressing

- Given a model of a concurrent system, these, and other functional properties can be checked using **model checking**
  - All states in which the system (design) can end up are inspected
  - It is **automatic**
  - Provides useful feedback (**counter-examples**)

Model checking



Model of a (concurrent) system

Finite automata

φ

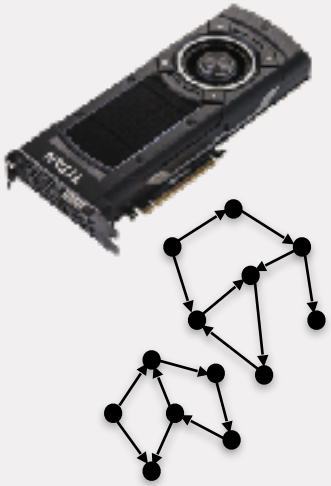
Functional requirement

Temporal logic formula

Model checker

+ counter-example

GPU accelerated explicit-state model checking



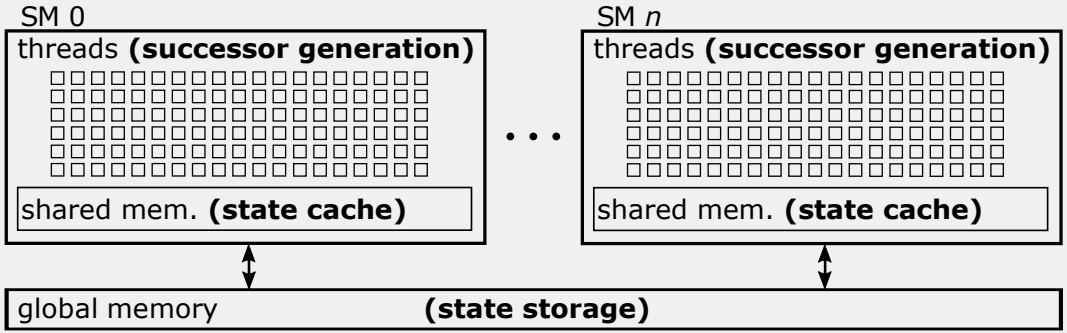
- **GPUexplore version 1.0: TACAS 2014 (networks of LTSs)**
  - Verification of deadlock freedom and safety properties: **2015** (**STTT**)
- **GPUexplore version 2.0: FM 2016**
  - + Partial-Order Reduction: **ATVA 2016**
  - + Verification of suffix-bounded LTL formulae: **CAV 2016**
- **GPUexplore version 3.0: TACAS & SPIN 2023 (Frontiers in HPC, 2024)**
  - Accepts Finite Automata with data variables & arrays (EFAs)
  - Uses GPU Tree Database for compact state storage
  - Support for Linear-Time Temporal Logic (LTL): **TACAS 2024**

2016: 36 seconds instead of 1.5 hours!
(TACAS, CAV, ATVA, FM, Software Tools for Technology Transfer)

GPU accelerated explicit-state model checking

- **Contributions:**
  - First implementation and evaluation of a tree database for GPUs in the context of GPU explicit-state model checking
  - Comparison of using various hashing techniques
    - First to implement Cleary compression for GPUs
    - Novel combination of Cleary compression and Cuckoo hashing: **Cleary-Cuckoo**

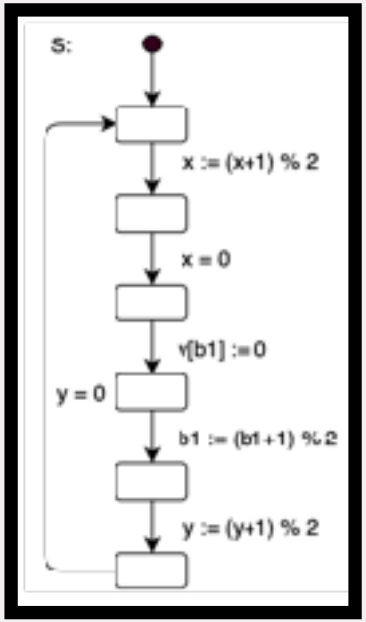
GPUexplore overview



while there are unexplored states:
- select set of unexplored states *S* (mark them explored)
- **for all** successors *s'* of all *s* ∈ *S:*
  - store *s'* in the (local) state cache
- sync. cache with global memory

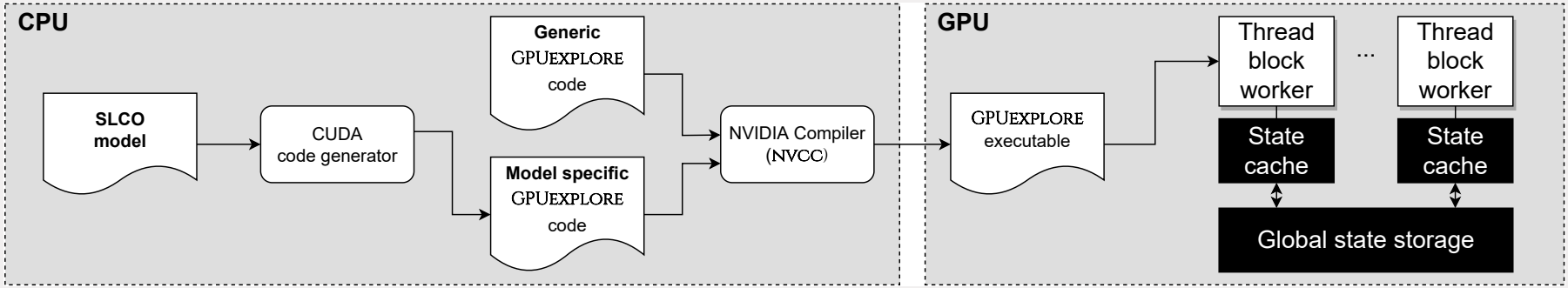
- **Global memory hash table**
  - *Open / Closed*
- Each Streaming Multiprocessor (SM) runs multiple **thread blocks** (512 threads)
- Each thread block has a **(shared memory) cache:**
  - temporarily store succs.
  - local dupl. detection
- **Fine-grained parallelism**
  - Groups of *n* threads are assigned to state vectors of size *n* (thread ⟷ process)

GPUexplore 3.0 workflow



- **SLCO** (Simple Language of Communicating Objects): concurrent state machines with data
- Given a model, generate code to **execute** the state machines

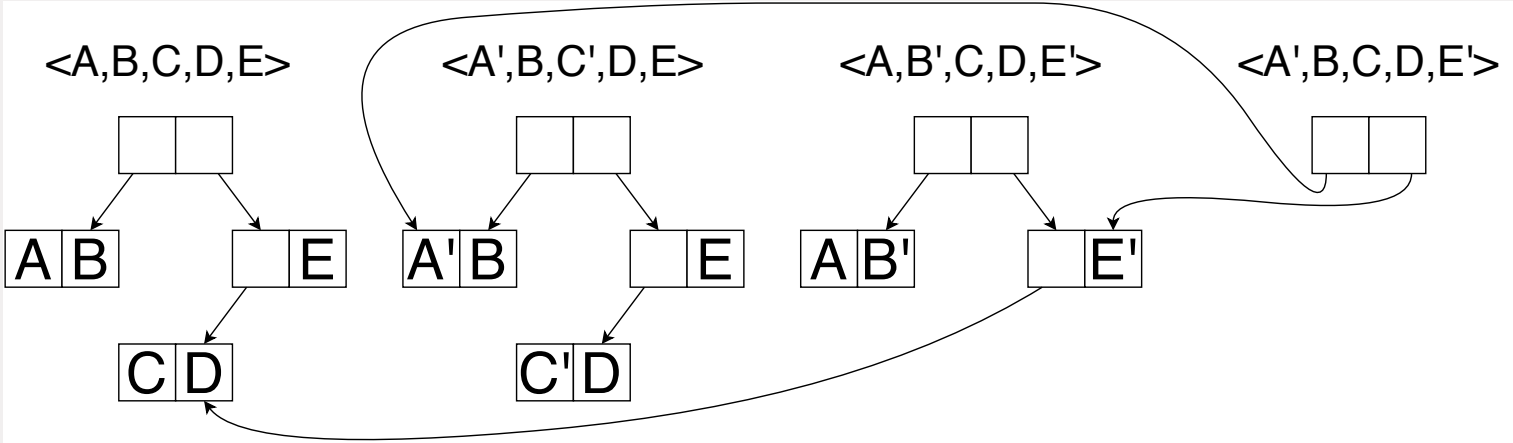
- **One search iteration (of a thread block):**
  - Fetching unexplored states (place them in shared memory work tile)
  - Explore the states, store successors in cache
  - Storing successors in global hash table



TU/e

Tree database: store system states as binary trees

- **With data, system states of SLCO models can be large**

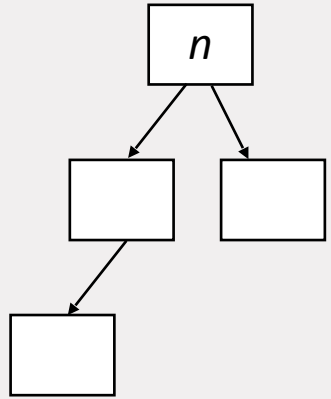


- Laarman, Van de Pol, Weber, *Parallel Recursive State Compression for Free*, SPIN (2011)
- Laarman, *Optimal compression of combinatorial state spaces*, Innovations in Systems and Software Engineering (2019)

TU/e

Unfolding recursion

- Recursion should always be avoided in a GPU program
  - Requires a stack, but thread stacks are stored in (slow) global memory
- And yet, storing trees is often implemented using recursion



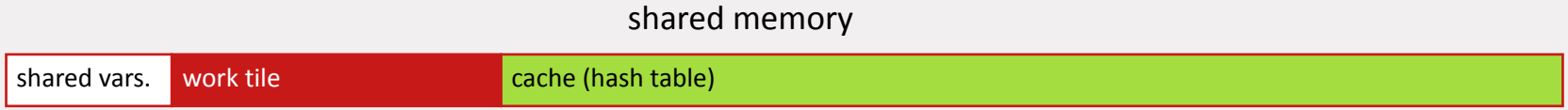

```
store(n) {
        addr1 = store(n->left);
        addr2 = store(n->right);
        set_addrs(n, addr1, addr2);
        write(n);
}
```

- **Solution:** write out recursion, possible because for a given model, the tree structure is fixed

```
store(n) {
        addr1 = write(n->left->left);
        set_addrs(n->left, addr1, null);
        addr1 = write(n->left);
        addr2 = write(n->right);
        set_addrs(n, addr1, addr2);
        write(n);
}
```


TU/e

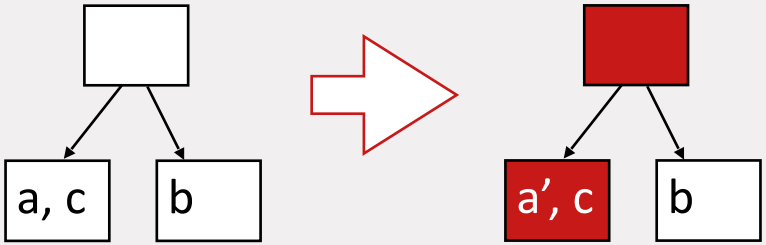
A thread block in GPUexplore 3.0



1. Fetch **root nodes** to be explored from global memory, place them in the *work tile*
    1. Some may have been claimed in previous search iteration
2. Distribute work over threads
3. **Fetch the trees of the roots, store them in the cache**
4. Threads produce successors and store them in the *cache*
5. When all threads are done, the cache is synchronised with the global hash table
    1. Any new states may be claimed for the next iteration

When fetching trees, only store the leafs? No!

Cache complete trees for efficient global memory storage of successors

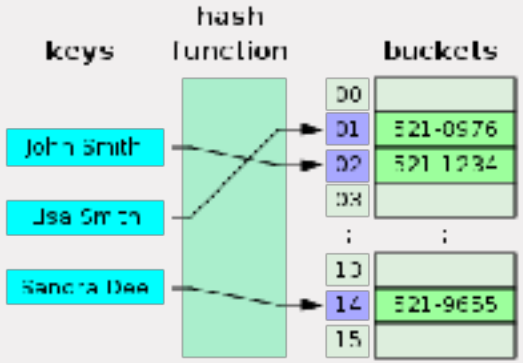


Tree compression becomes very time-efficient if storage of new trees can be **restricted to new nodes**

- Store **all** nodes of the trees to be explored in the cache
- Each node requires 64 bits for storage **plus 32 bits for cache pointers**
- **Additional benefit:** sharing (compression) of nodes in the cache

Hash table

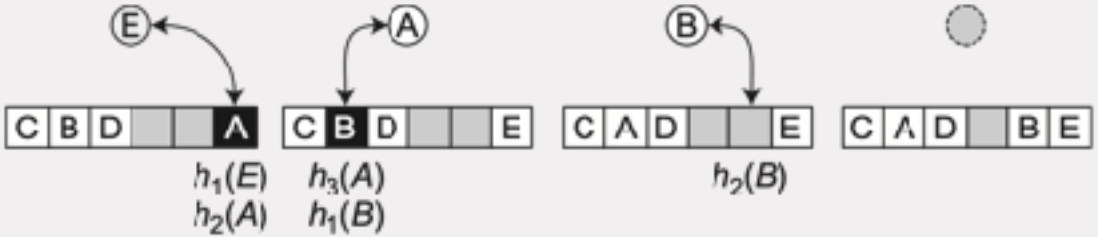
- Data structure to store elements
  - Fast insertion and lookup
- Ideal when the domain is too large to store all elements in memory, and it is expected that not all elements need to be stored
- Hash function *h* maps element *e* to position *h(e)*
- Can also be used to store (key, value) pairs
  - Store values either separately, or physically together with key
  - Involving values does not make hashing more complicated, so I ignore them
- **Collisions: *h* may hash keys *e1, e2* to the same address**
  - ***h(e1) = h(e2)***



Source: Wikipedia

Cuckoo hashing

- **Idea:** whenever there is a collision, evict the old element, store the new element, and move the old element using another hash function
- **Example:** E collides with A, A is moved, collides with B, B is moved
- **Benefit: constant time lookups**
- **Drawbacks:**
  - **no constant time insertion, can lead to eviction chains**
    - Restrict chains in size, four hash functions are often enough to reach load factor of 90%
- **GPUs:** first hash function to be implemented (Alcantara et al., 2011)



Alcantara *et al.*: Building an Efficient Hash Table on the GPU (2012)

State compression

- Can we use less space to store nodes?
- J.G. Cleary. *Compact Hash Tables Using Bidirectional Linear Probing*, IEEE Transactions on Computers (1984)
- Idea: consider a node *n* requiring *m* bits to store
  - Hash (bit scramble) *n* to *h(n)* (*m* bits)
  - Split *h(n)* into $h(n)_0$ and $h(n)_1$
  - Store $h(n)_1$ at address $h(n)_0$
- To reconstruct stored nodes, *h* must be **reversible:**
  - Get $h(n)_1$ from $h(n)_0$
  - Combine $h(n)_0$ and $h(n)_1$
  - $n = h^{-1}(h(n))$

Cleary compression to compress roots

- But how to handle collisions?
- Cleary defined an elaborate scheme with **linear probing**, which forms *clusters*
  - As nodes are moved, only roots of state trees can be compressed: use a **root** table and a **non-root (internal)** table

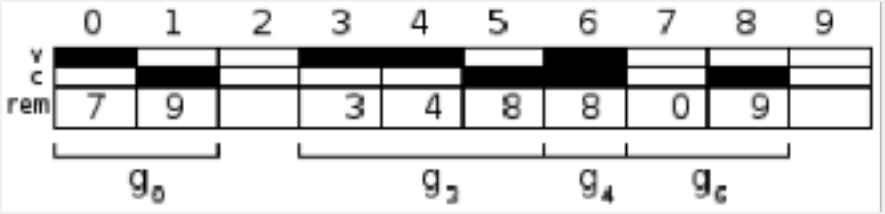


Image from Van der Vegt and Laarman, *A Parallel Compact Hash Table*, MEMICS (2011)

- **Group:** consecutive list of remainders, associated with one home location
- **Cluster:** consecutive groups, enclosed by empty positions
- **v bit:** corresponding address is a home location
- **c bit:** last address of a group, followed by another group, or an empty address
- Example stores 7, 9, 33, 34, 38, 48, 60, 69
- **Algorithm to restore elements:** move to the left from the stored location,
  count the number of set c bits, reach the left of the cluster, and count back the same number of set v bits

Cleary compression to compress roots

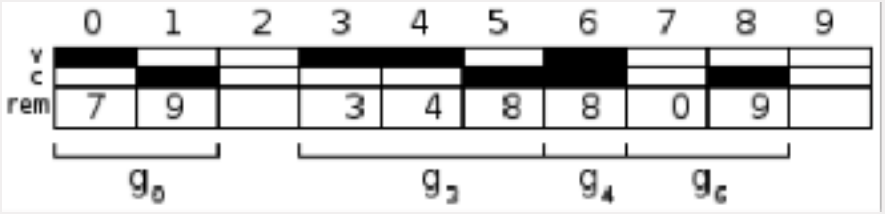


Image from Van der Vegt and Laarman, *A Parallel Compact Hash Table*, MEMICS (2011)

- Van der Vegt and Laarman: this scheme still works quite well for multi-threaded use
  - Each cluster requires a lock, clusters must sometimes be relocated
- For a GPU, this is **bad news:**
  - many global memory accesses, may not be aligned with (fixed-size) buckets
  - locks
- **Can we devise an alternative approach?**

Contribution: combine Cleary compression with Cuckoo hashing

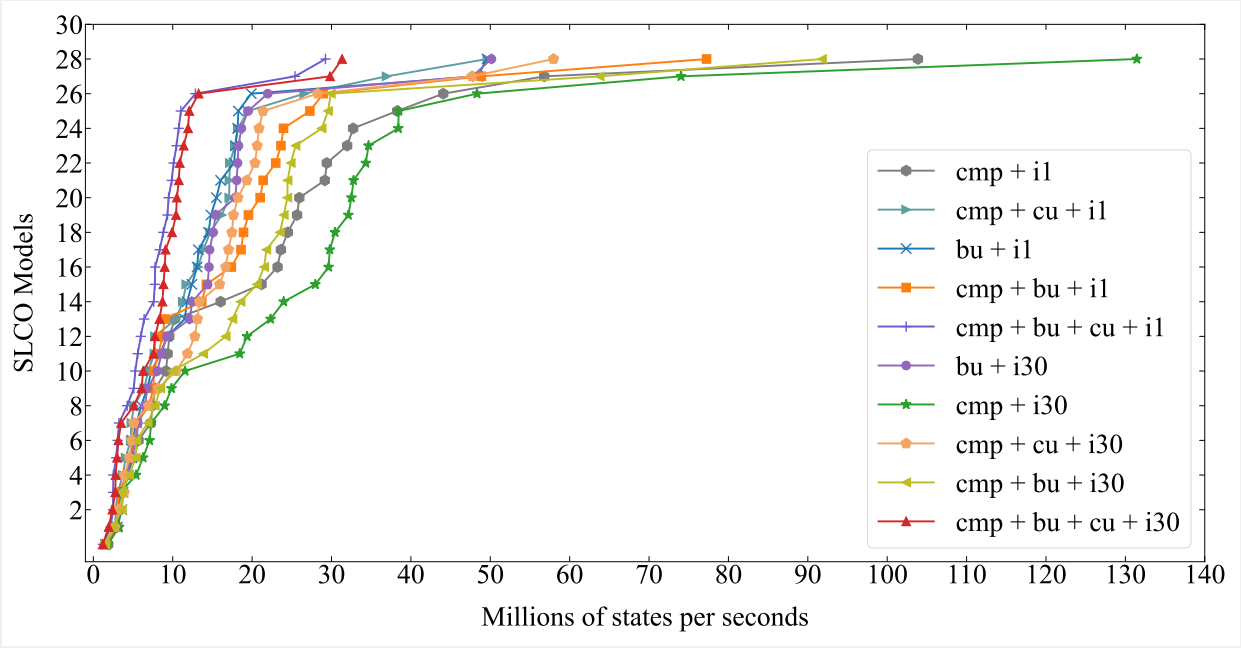
- Collision handling in Cuckoo hashing: when a new node *n* must be stored at the address where node *m* is stored
  - Evict *m*, store *n*
  - Rehash *m* with another hash function, and store *m* at new position
  - When the maximum length of an eviction chain is reached, conclude that the table is full

Contribution: combine Cleary compression with Cuckoo hashing

- **Combination:**
  - Hash *n* using function h_0: $h_0(n)$
  - Split $h_0(n)$ into $h_0(n)_0$ and $h_0(n)_1$
  - Try to store $h_0(n)_1$ **plus hash function ID 0** at $h_0(n)_0$
  - If there is a collision with an $h_i(m)_1$
    - Evict $h_i(m)_1$ and store $h_0(n)_1$
    - Combine $h_i(m)_0$ and $h_i(m)_1$
    - Retrieve $m = h_i^{-1}(h_i(m))$
    - Hash *m* using function h_{i+1}: $h_{i+1}(m)$
    - …

Experimental results

- BEEM benchmarks (Brno), translated to SLCO
- Bu: buckets, Cmp: Cleary compression, cu: Cuckoo, i<n>: # iterations per kernel (function) launch
- No Cuckoo: 32 (reversible) hash functions; when collision occurs, rehash current element



Experimental results

Table 1: Millions of states per second for various reachability tools and configurations. Pink cells: out of memory. Yellow cells: timeout. Green cell: best average. O.M.: out of memory at initialisation. SU: speedup of (CMP + i30) vs. (LM-1).

Input		CPU tools				GPUEXPLORE + SLCO Configurations								
Model	States	SP-1	SP-4	LM-1	LM-4	BITS	CR	BU + i1	CMP + i1	CMP + BU + i1	CMP + CU + i1	CMP + i30	CMP + CU + i30	SU
adding.20+	84,709,120	1.128	3.223	1.211	3.938	100	1.96	49.597	56.793	48.879	36.934	**74.026**	47.694	**61x**
adding.50+	529,767,730	0.856	O.M.	1.354	5.356	100	1.96	48.403	103.872	77.243	49.625	**131.444**	57.968	**97x**
anderson.6	18,206,917	0.623	1.362	0.516	1.309	122	1.82	14.814	16.035	13.647	11.265	**34.111**	17.649	**62x**
anderson.7	538,699,029	0.599	O.M.	0.448	1.583	141	2.75	9.309	21.192	14.244	10.426	**22.326**	10.435	**41x**
at.5	31,999,440	0.646	1.495	0.653	1.880	85	1.86	19.894	29.158	23.633	18.204	**38.457**	21.375	**59x**
at.6	160,589,600	0.454	0.869	0.695	2.387	85	1.90	17.901	38.275	27.275	19.498	**38.418**	20.359	**55x**
at.7	819,243,816	0.527	O.M.	0.666	2.372	97	1.98	12.415	**23.629**	17.381	13.194	22.329	13.378	**34x**
at.8+	3,739,953,204	0.534	O.M.	0.555	1.817	97	1.97	5.452	7.246	7.593	11.698	7.287	**11.854**	**13x**
bakery.5	7,866,401	1.400	2.570	0.410	0.904	140	2.51	11.504	7.838	7.585	6.407	**19.362**	12.782	**47x**
bakery.7	29,047,471	1.228	2.592	0.580	1.618	140	2.49	13.236	9.361	9.021	7.698	**29.783**	17.456	**51x**
bakery.8	841,696,300	0.760	1.269	0.690	2.436	140	2.40	3.745	29.410	23.957	17.116	**32.778**	18.215	**48x**
elevator2.3	7,667,712	0.554	1.099	0.463	0.985	189	3.96	4.890	3.259	3.185	2.817	**6.261**	4.827	**14x**
elevator2.4	91,226,112	0.263	0.561	0.623	1.945	213	3.97	3.025	**3.746**	2.907	3.087	3.267	2.703	**5x**
elevator2.5+	1,016,070,144	0.189	O.M.	0.473	1.630	317	5.95	1.540	**1.871**	1.545	1.520	1.839	1.491	**4x**
frogs.4	17,443,219	1.044	2.228	0.553	1.423	219	3.49	8.423	10.253	8.686	7.767	**11.549**	8.168	**21x**
frogs.5	182,772,126	0.531	1.048	0.751	2.630	251	3.84	6.766	9.573	8.214	6.898	**9.846**	6.943	**13x**
lamport.6	8,717,688	1.277	1.375	0.490	1.096	96	1.91	11.813	5.126	5.225	4.697	**27.966**	19.335	**57x**
lamport.7	38,717,846	1.001	1.822	0.672	1.979	116	1.98	18.176	23.205	18.915	16.170	**34.321**	20.641	**51x**
lamport.8	62,669,317	0.917	1.776	0.698	2.194	116	1.98	17.717	25.947	21.015	17.132	**35.387**	20.864	**50x**
loyd.2	362,880	1.278	0.758	0.255	0.497	90	1.05	**7.339**	4.204	4.220	3.723	3.243	3.930	**13x**
loyd.3	239,500,800	0.633	O.M.	0.650	2.338	114	1.96	18.268	44.073	28.970	26.556	**48.328**	28.248	**74x**
mcs.5	60,556,519	0.706	0.615	0.453	1.489	148	2.97	14.504	24.498	19.537	14.710	**29.635**	15.912	**65x**
mcs.6	332,544	1.240	0.244	0.181	0.331	156	2.75	**6.037**	3.003	3.097	2.751	3.446	3.131	**19x**
peterson.5	131,064,750	0.711	1.617	0.727	2.435	140	2.98	16.034	31.975	21.394	17.813	**32.331**	16.681	**42x**
peterson.6	174,495,861	0.852	0.756	0.720	2.451	140	2.98	15.503	32.725	22.975	17.198	**34.902**	17.030	**45x**
peterson.7	142,471,098	0.683	1.496	0.652	2.269	175	2.63	13.077	25.667	18.603	13.868	**26.183**	13.120	**37x**
phils.6	14,348,906	0.208	0.422	0.240	0.670	150	1.49	4.410	**7.458**	5.528	4.789	7.084	4.543	**30x**
phils.7	71,934,773	0.179	0.297	0.246	0.764	151	1.49	3.585	**5.702**	4.762	4.064	5.382	3.885	**22x**
phils.8	43,046,720	0.160	0.361	0.243	0.788	160	1.49	4.842	**9.151**	6.987	5.119	8.973	5.089	**37x**
szymanski.5	79,518,740	0.665	1.571	0.535	1.815	180	2.91	11.944	17.803	14.416	11.653	**18.357**	11.674	**33x**
Average		0.728	1.309	0.58	1.844	n/a		13.139	21.068	16.355	12.813	**26.621**	15.246	**40x**

- BEEM benchmarks (DVE -> SLCO)
- Comparison with LTSmin and SPIN (1- and 4-cores)
- Cleary compression not only more compact, also faster! (Bu+i1 vs. Cmp+i1)
- Cuckoo slows down!

TU/e

Conclusions

- GPU state space exploration with support for state machines with variables
- Code generation to execute model transitions
- For memory efficiency, store state vectors as binary trees
- Various optimisations applied
- Novel combination of Cleary compression and Cuckoo hashing
- Speedups up to 100x compared to state-of-the-art

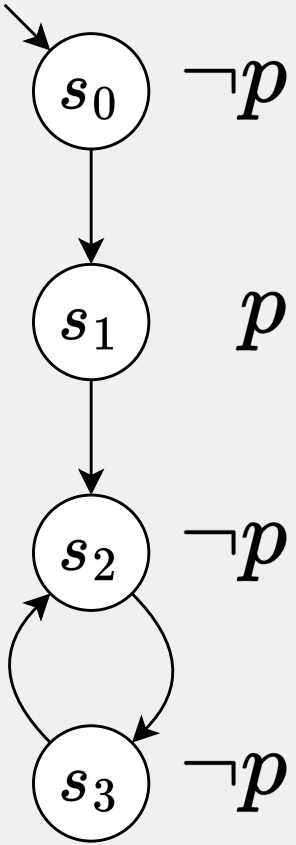
- **Future work:**
  - Hashing: search for ideal speed / load factor balance
  - Support for property verification (Linear Temporal Logic)
  - Multi-GPU support
  - Verification of *probabilistic* systems

amazon research awards

NWO

Support for Linear-Time Temporal Logic

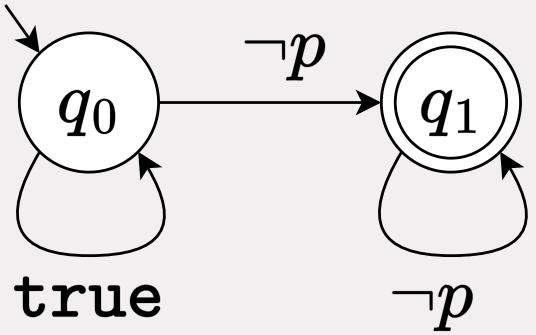
- Kripke structure to represent system behaviour
  - Finite set of states $\mathcal{S}$ (including an initial state)
  - Left-total transition relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$
  - $\lambda : \mathcal{S} \rightarrow 2^{AP}$ is a *labelling* function
    - Labels address *state-local* atomic propositions
      - E.g.: *x = 0*



TU/e

Automata-based LTL model checking

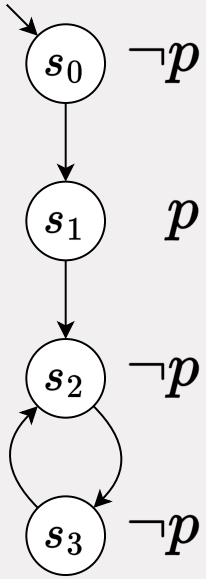
- Transform the *negation* of an LTL formula φ into a *Nondeterministic Büchi Automaton (NBA)* $B_{\neg\varphi}$
- An NBA is an automaton with a finite set of *accepting* states $\mathcal{Q}_F$ (visualised with double borders)
- A path through an NBA visits accepting states infinitely often



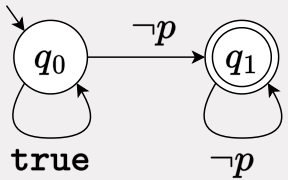
Negation of GF p

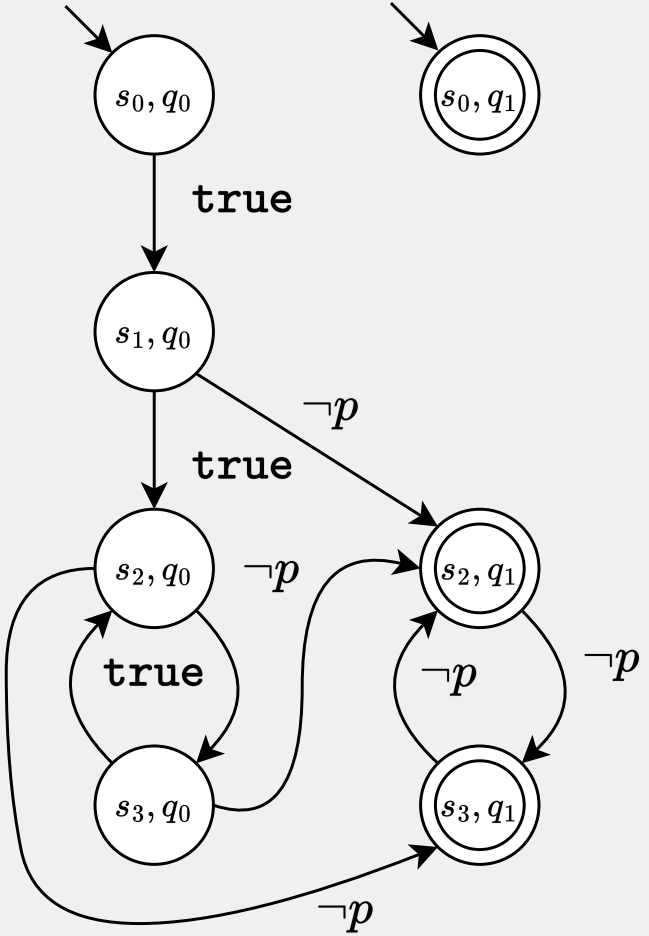
GF p: globally, eventually p holds

Automata-based LTL model checking

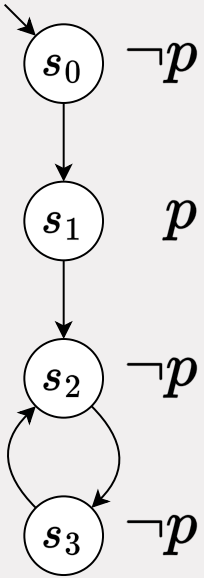


Compute the *product* of Kripke structure K and $B_{\neg\varphi}$

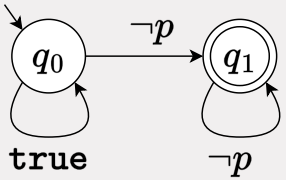


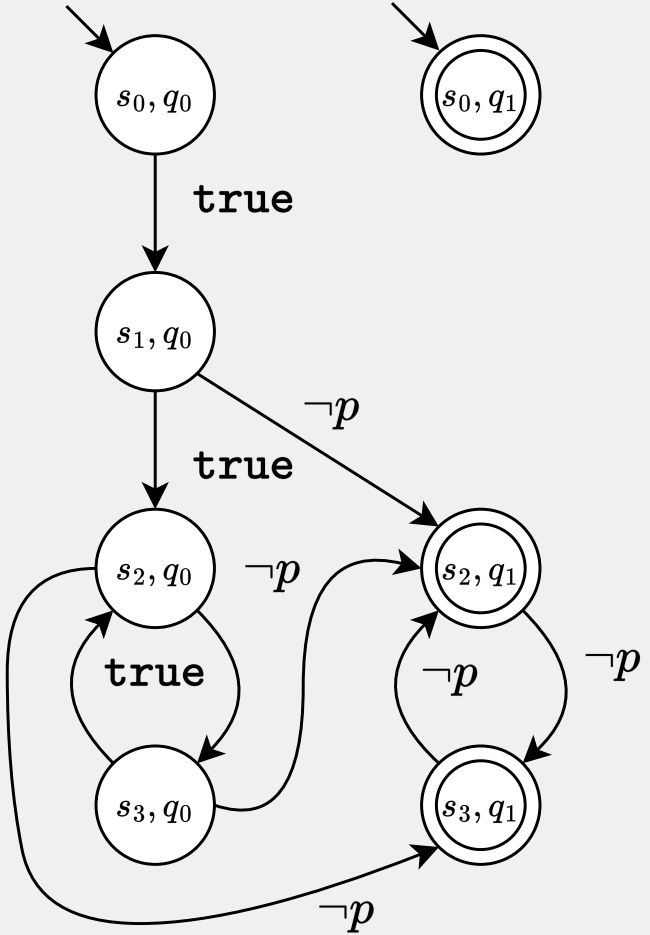


Automata-based LTL model checking



Compute the *product* of Kripke structure K and $B_{\neg\varphi}$



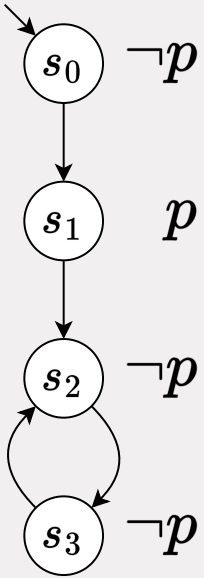


Combine states of K and $B_{\neg\varphi}$:

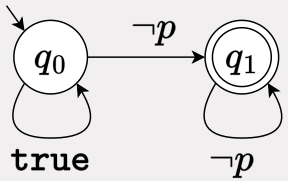
$$\mathcal{Q}_{\otimes} = \{\langle s, q\rangle \mid s \in \mathcal{S} \wedge q \in \mathcal{Q}\}$$

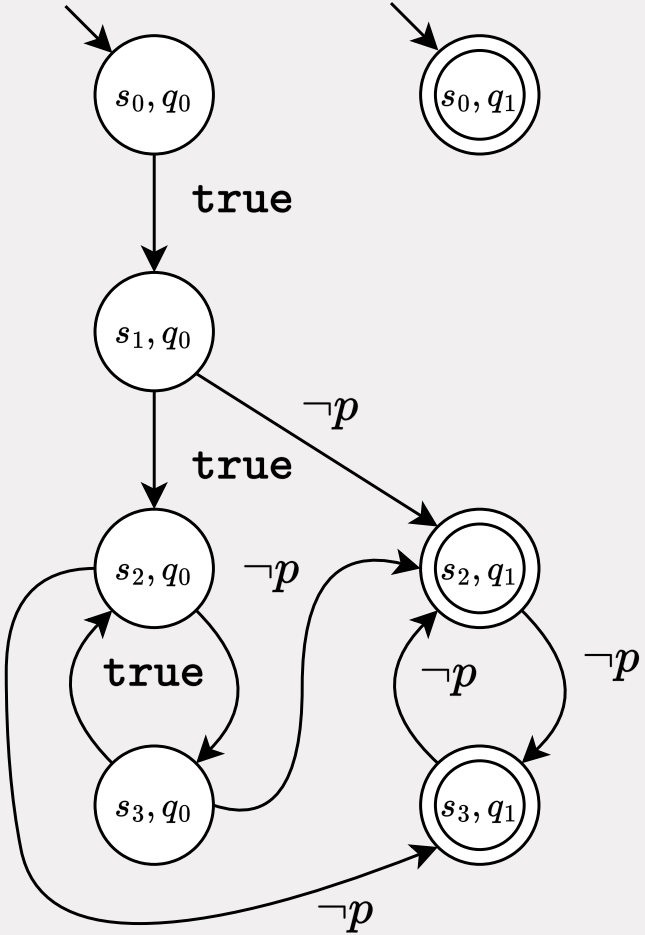
TU/e

Automata-based LTL model checking



Compute the *product* of Kripke structure K and $B_{\neg\varphi}$



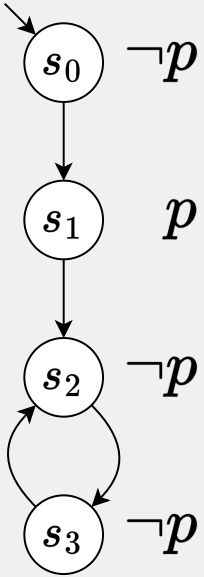


$s \to s', q \xrightarrow{\varphi} q'$ and $K, s' \vDash \varphi$ implies that $\langle s, q\rangle \xrightarrow{\varphi}_{\otimes} \langle s', q'\rangle$

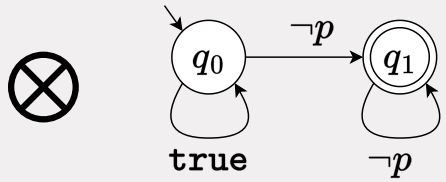
($K, s' \vDash \varphi$ means $\lambda(s')$ is a subset of the APs satisfying φ)

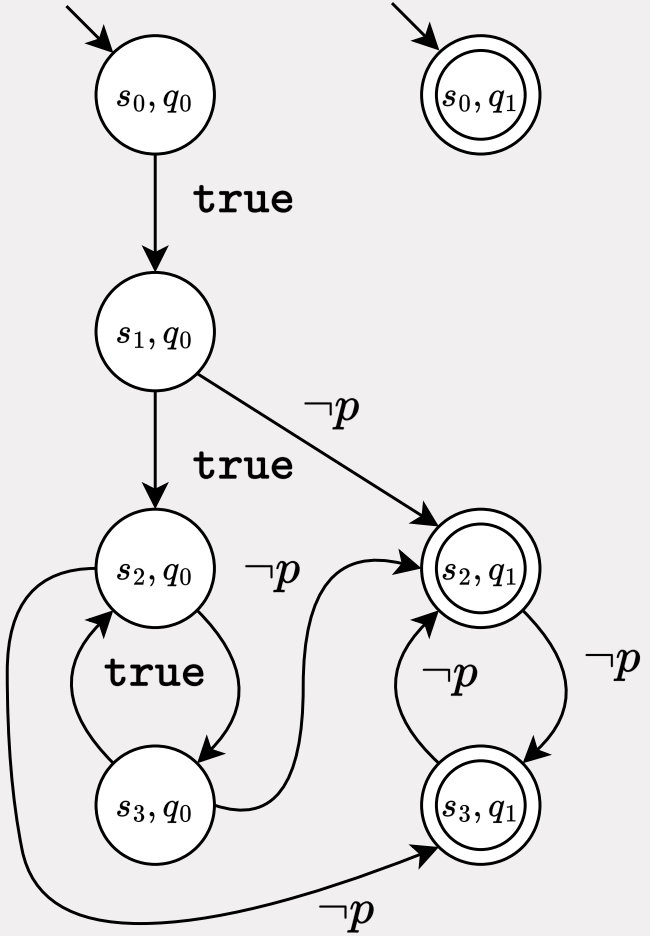
TU/e

Automata-based LTL model checking



Compute the *product* of Kripke structure K and $B_{\neg\varphi}$

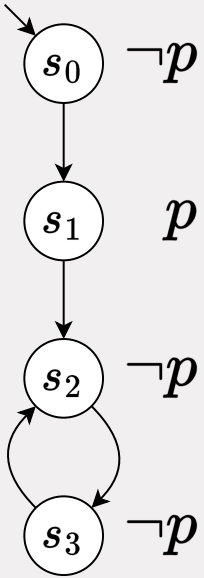




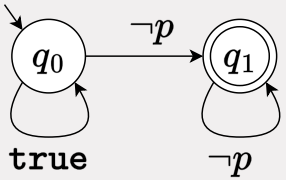
Initial states: combination of the initial state of K, and the initial state(s) of $B_{\neg\varphi}$ plus the states reachable in $B_{\neg\varphi}$ via transitions with a label such that $K, s \models \varphi$

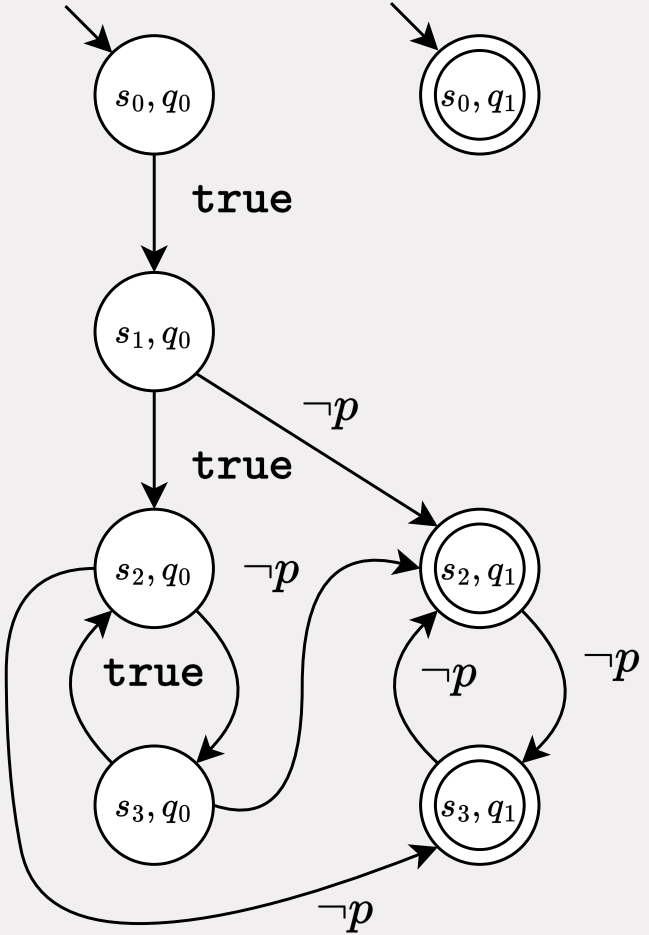
TU/e

Automata-based LTL model checking



Compute the *product* of Kripke structure K and $B_{\neg\varphi}$



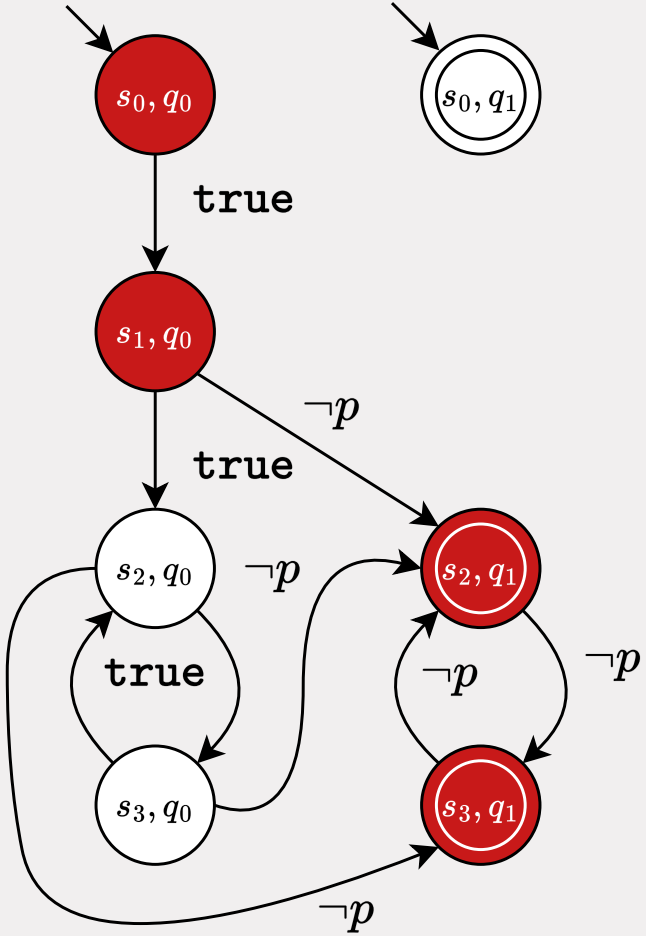


$\langle s, q \rangle$ is accepting iff q is accepting

TU/e

Automata-based LTL model checking

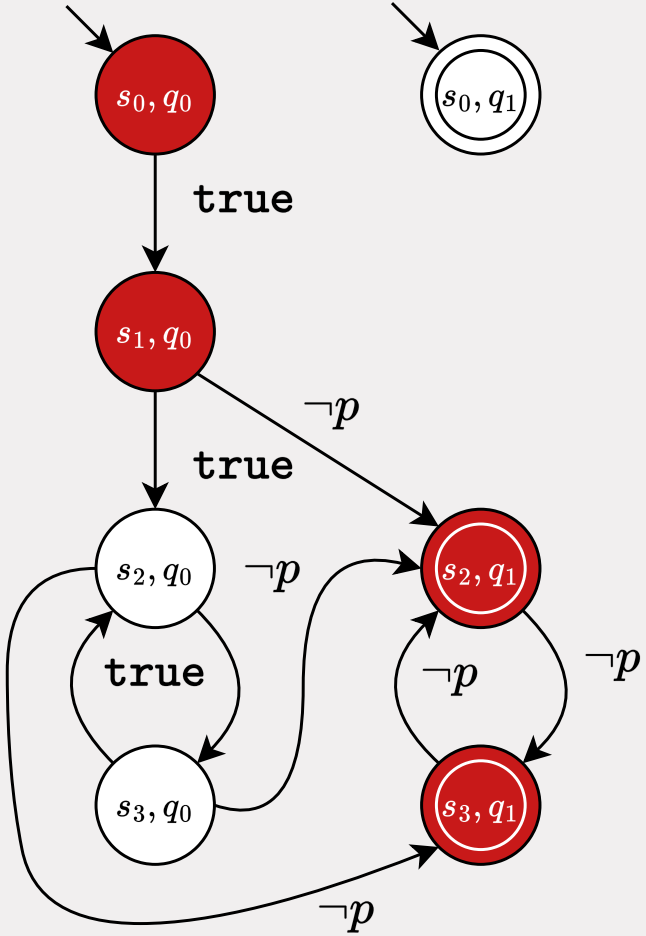
- LTL model checking can be performed by solving the *emptiness problem*
  - A Kripke structure K satisfies LTL formula φ iff $K \otimes B_{\neg\varphi}$ is empty, i.e., has no (accepting) paths
  - As paths are infinite, in a finite-state system, **an accepting path is a *lasso*** consisting of:
    - A finite sequence of transitions from an initial state to an accepting state $\langle s, q \rangle$
    - A cycle including $\langle s, q \rangle$



TU/e

Automata-based LTL model checking

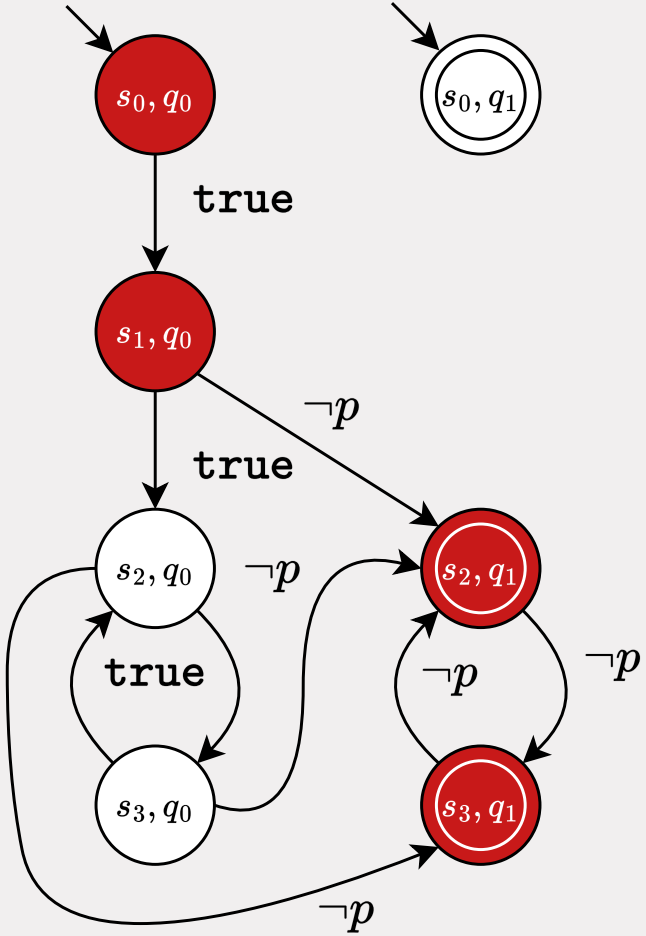
- State-of-the-art sequential algorithms are *Depth-First Search* (DFS) based
- **Nested Depth-First Search (NDFS) [Courcoubetis *et al.*, 1992]:**
  - When backtracking over an accepting state in the first DFS, start a second (nested) DFS
  - When the second DFS reaches the stack of the first DFS, a counter-example is found
- **Algorithms using Strongly Connected Component detection** (for instance, with Tarjan's algorithm, 1972)
- **Multi-core NDFS [Laarman *et al.*, 2011] (with AW):**
  - Multiple threads perform DFS-like searches (randomised)



TU/e

Automata-based LTL model checking

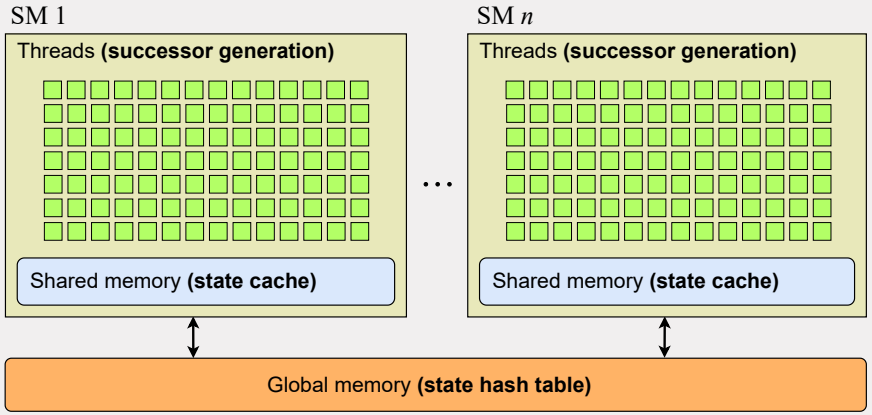
- State-of-the-art sequential algorithms are *Depth-First Search* (DFS) based
- **Nested Depth-First Search (NDFS) [Courcoubetis *et al.*, 1992]:**
  - When backtracking over an accepting state in the first DFS, start a second (nested) DFS
  - When the second DFS reaches the stack of the first DFS, a counter-example is found
- **Algorithms using Strongly Connected Component detection** (for instance, with Tarjan's algorithm, 1972)
- **Multi-core NDFS [Laarman *et al.*, 2011] (with AW):**
  - Multiple threads perform DFS-like searches (randomised)



However …

TU/e

DFS is *not* suitable for GPUs — see GPUexplore:



while there are unexplored states **do**
- fetch a set of unexplored states S to *state cache* (mark them explored)
- **for all** successors s' of all $s \in S$ **do**
  - store s' in state cache
- synchronize state cache with the global hash table

- **Global memory hash table**
  - *Open / Closed*
- Each Streaming Multiprocessor (SM) runs multiple **thread blocks** (512 threads)
- Each thread block has a **(shared memory) cache:**
  - temporarily store succs.
  - local dupl. detection
- **Fine-grained parallelism**
  - Groups of n threads are assigned to state vectors of size n (thread $\longleftrightarrow$ process)

So what about Breadth-First Search based LTL model checking?

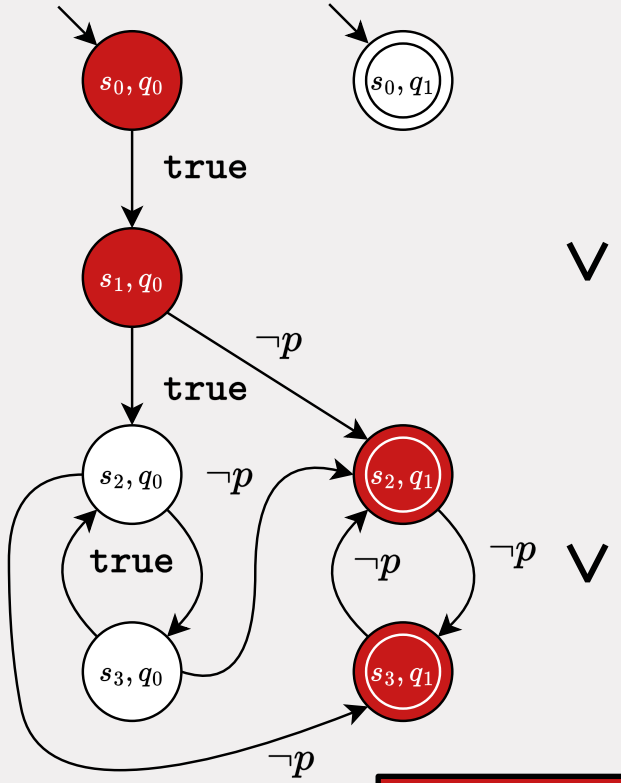
- Algorithms initially developed for *distributed* model checking
  - **Back-Level Edges [Barnat *et al.*, 2003]** searches for transitions that close a cycle
  - **Maximal Accepting Predecessors (MAP) [Brim *et al.*, 2004]** is shown to perform better
  - **One-Way-Catch-Them-Young (OWCTY) [Černá & Pelánek, 2003]** uses topological sorting, is *not* on-the-fly, i.e., cannot detect counter-example while constructing the state space
    - A heuristic, incomplete version of MAP is added as first phase to OWCTY **[Barnat *et al.*, 2009]**, so it *may* detect counter-examples early, if they exist
  - Heuristic, incomplete version of MAP is also implemented in SPIN as the **Piggyback algorithm [Holzmann, 2012, Filippidis & Holzmann, 2014]**
  - Post-exploration versions of MAP and OWCTY have been implemented for the GPU **[Barnat *et al.*, 2009, 2012]**

So what about Breadth-First Search based LTL model checking?

- Algorithms initially developed for *distributed* model checking
  - **Back-Level Edges [Barnat *et al.*, 2003]** searches for transitions that close a cycle
  - **Maximal Accepting Predecessors (MAP) [Brim *et al.*, 2004]** is shown to perform better
  - **One-Way-Catch-Them-Young (OWCTY) [Černá & Pelánek, 2003]** uses topological sorting, is *not* on-the-fly, i.e., cannot detect counter-example while constructing the state space
    - A heuristic, incomplete version of MAP is added as first phase to OWCTY **[Barnat *et al.*, 2009]**, so it *may* detect counter-examples early, if they exist
  - Heuristic, incomplete version of MAP is also implemented in SPIN as the **Piggyback algorithm [Holzmann, 2012, Filippidis & Holzmann, 2014]**
  - Post-exploration versions of MAP and OWCTY have been implemented for the GPU **[Barnat *et al.*, 2009, 2012]**

Let us consider MAP, as it is both complete and on-the-fly

MAP algorithm

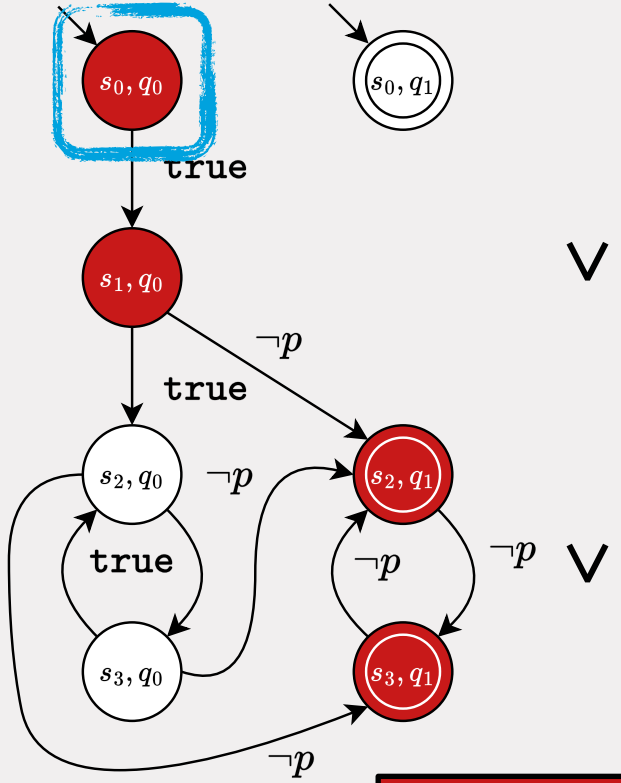


- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

TU/e

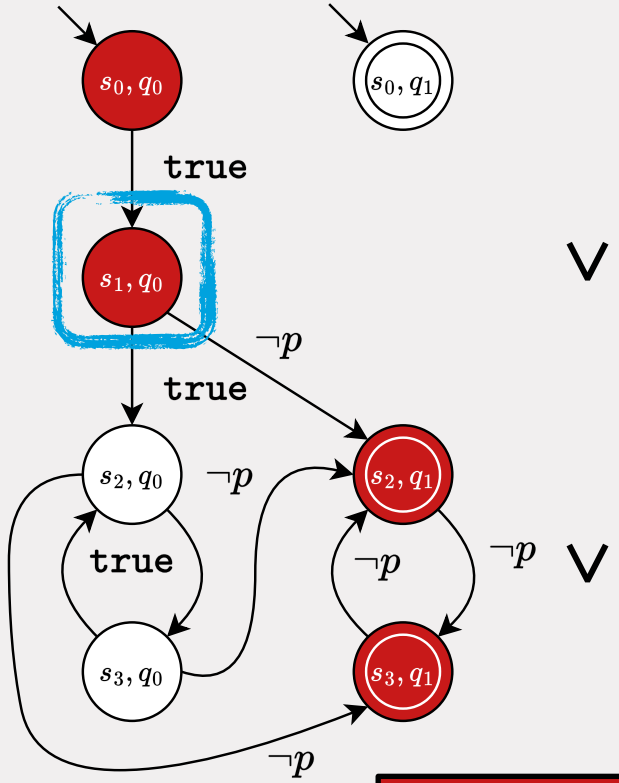
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

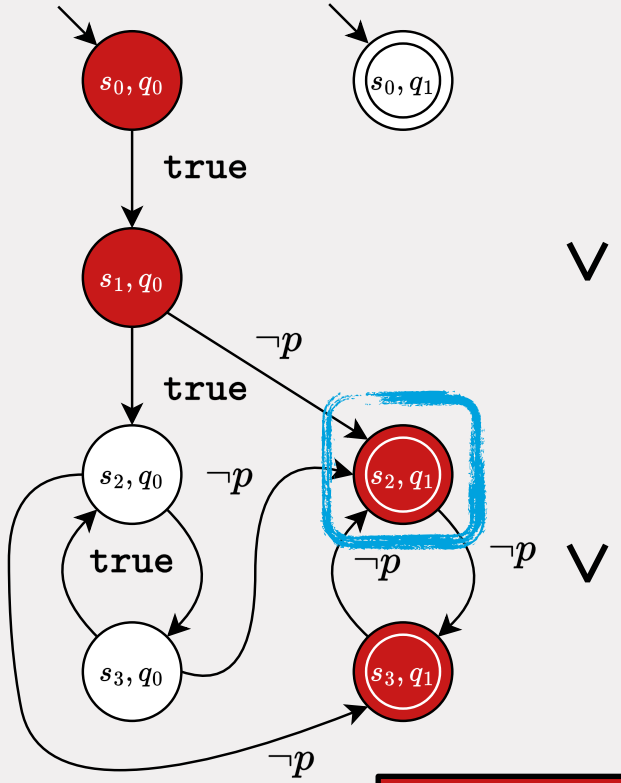
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

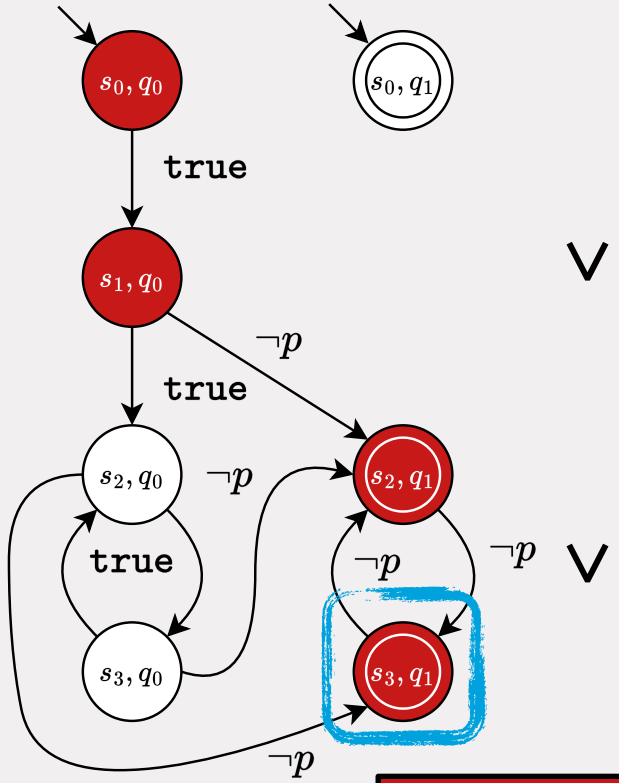
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

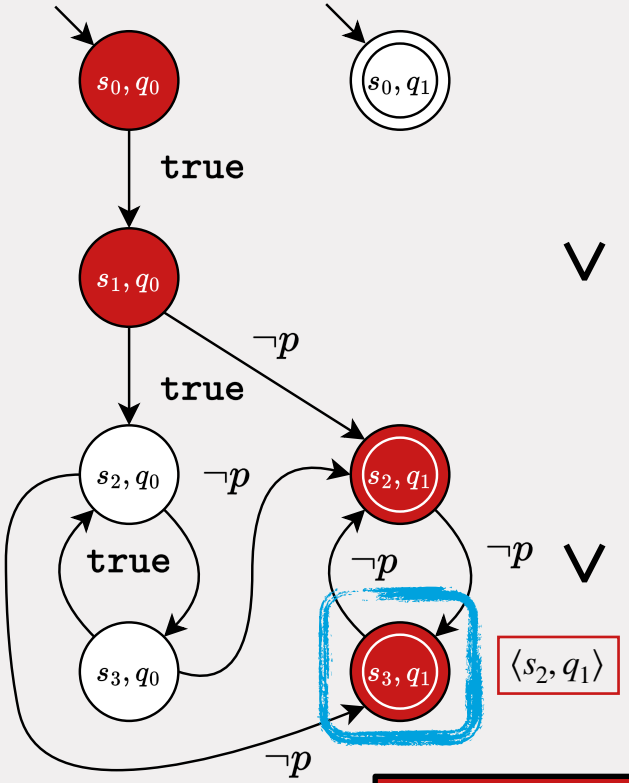
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

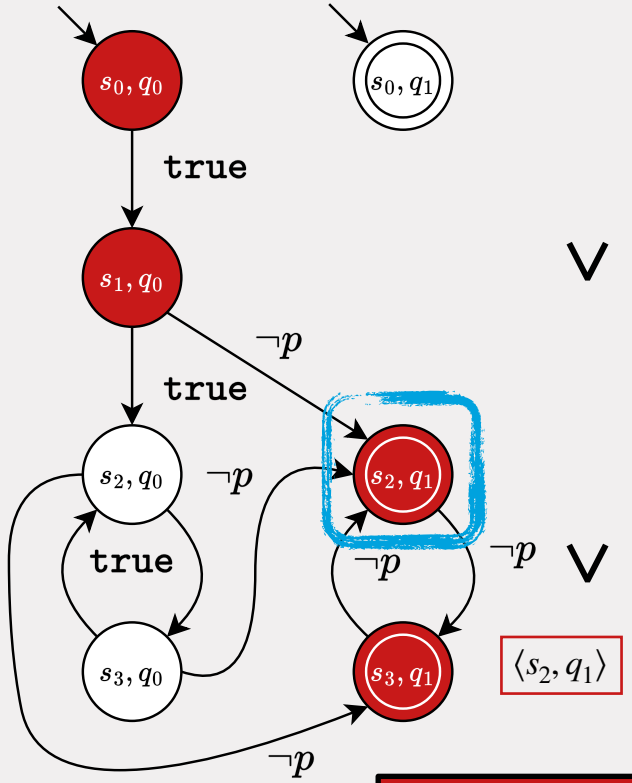
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

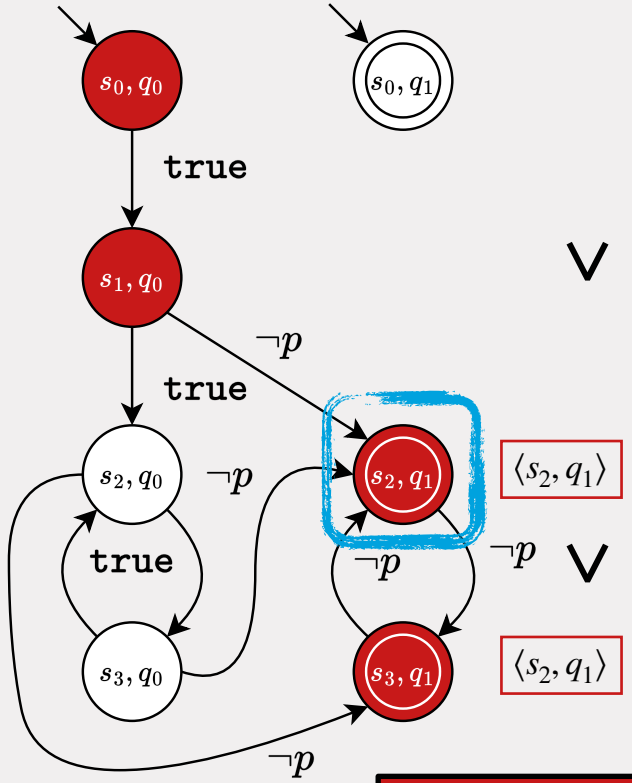
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

MAP algorithm

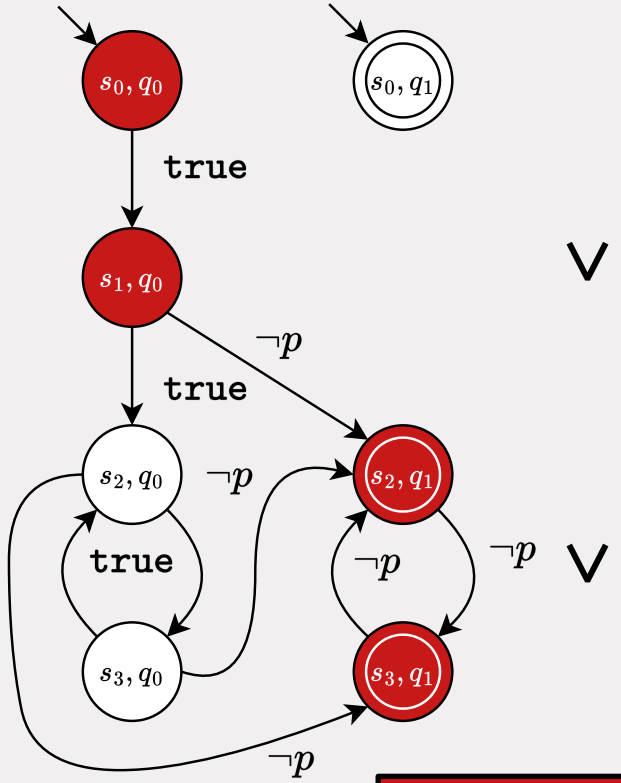


- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

TU/e

MAP algorithm

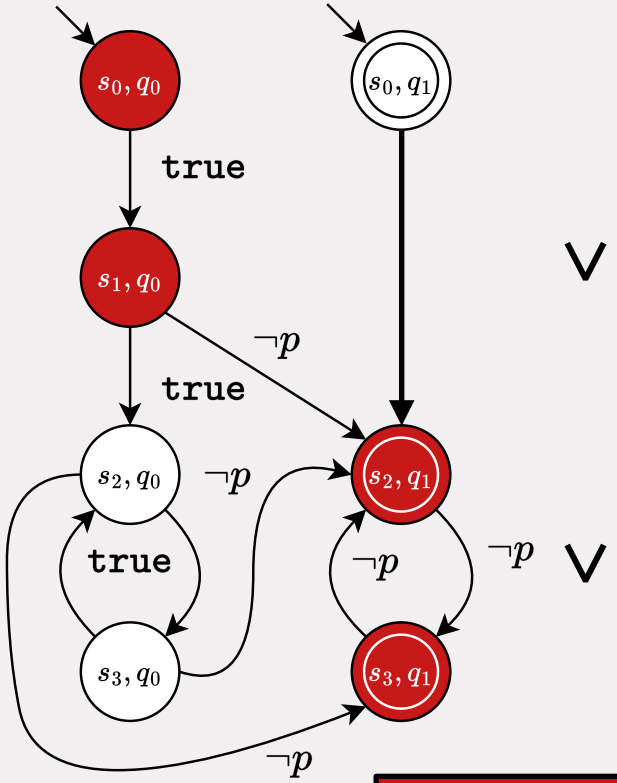


- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

TU/e

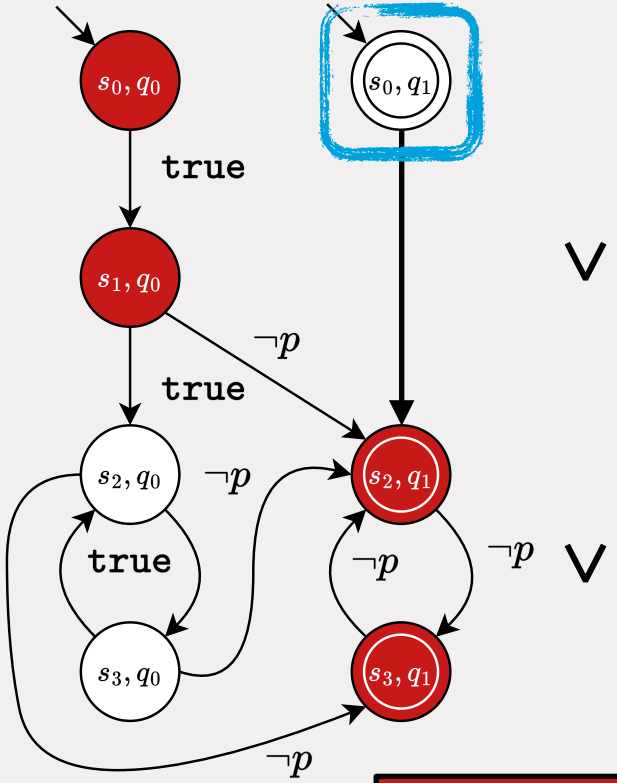
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

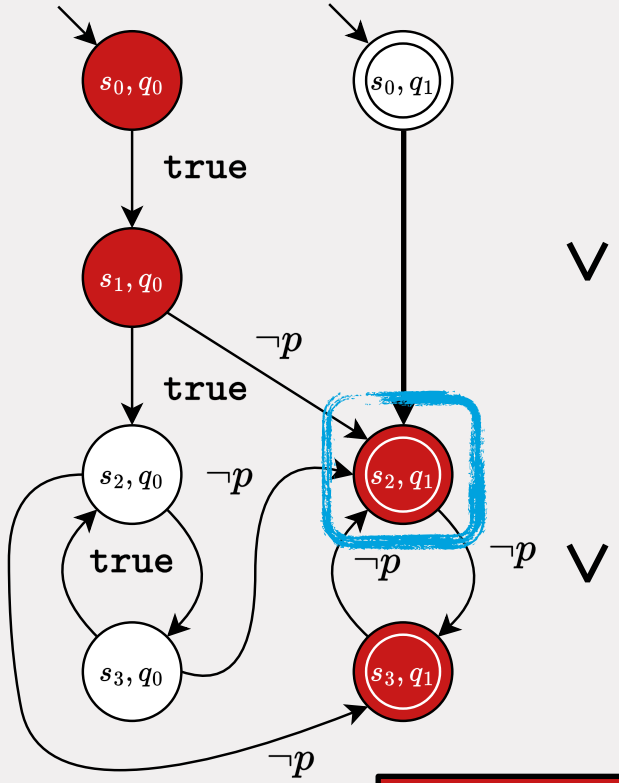
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

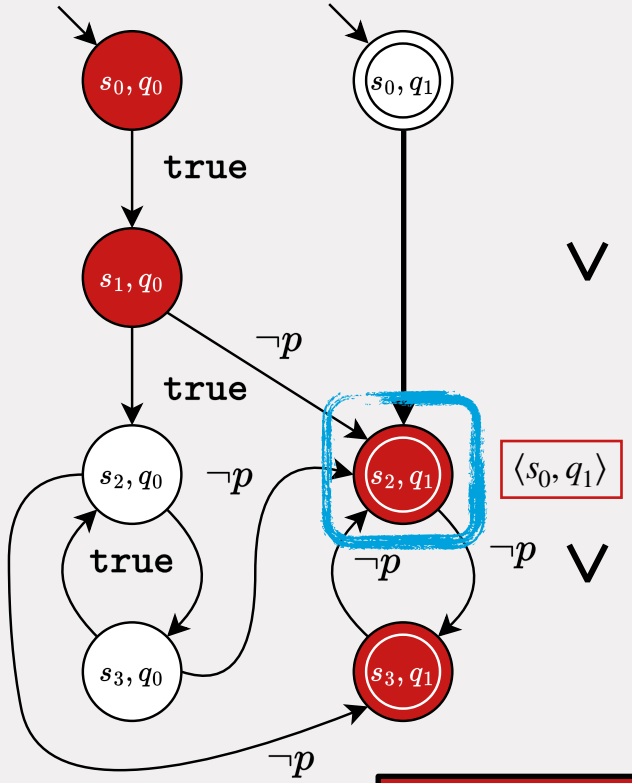
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

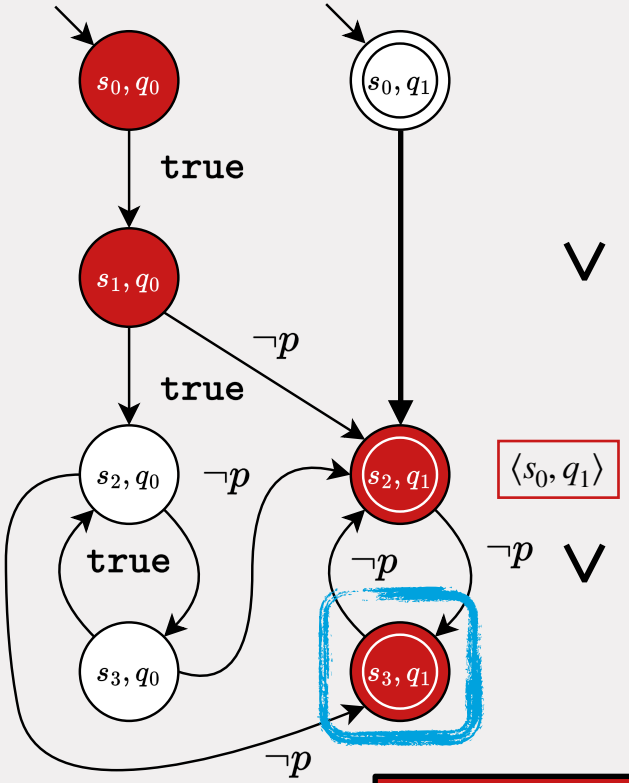
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

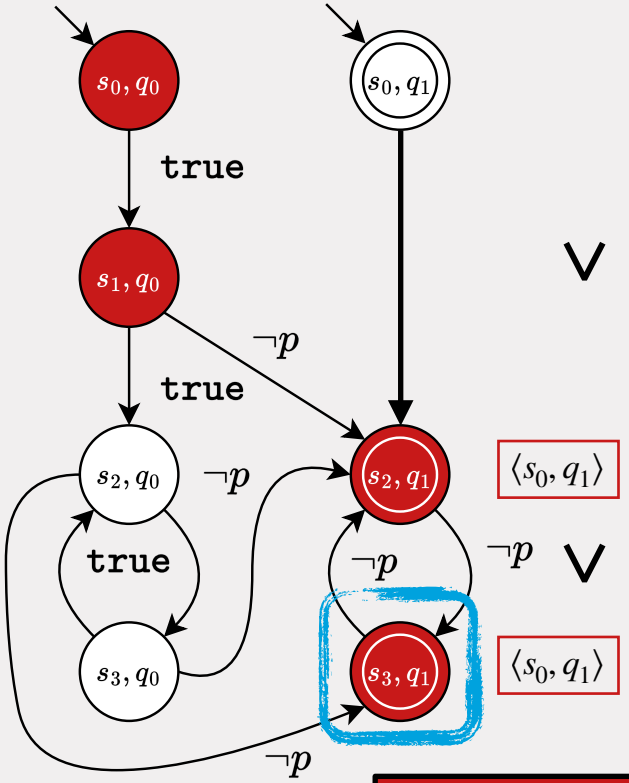
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

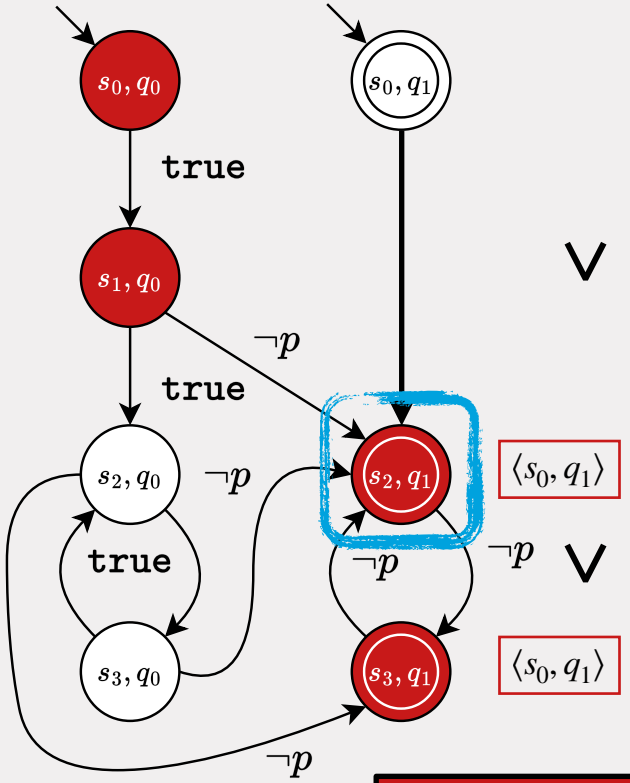
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

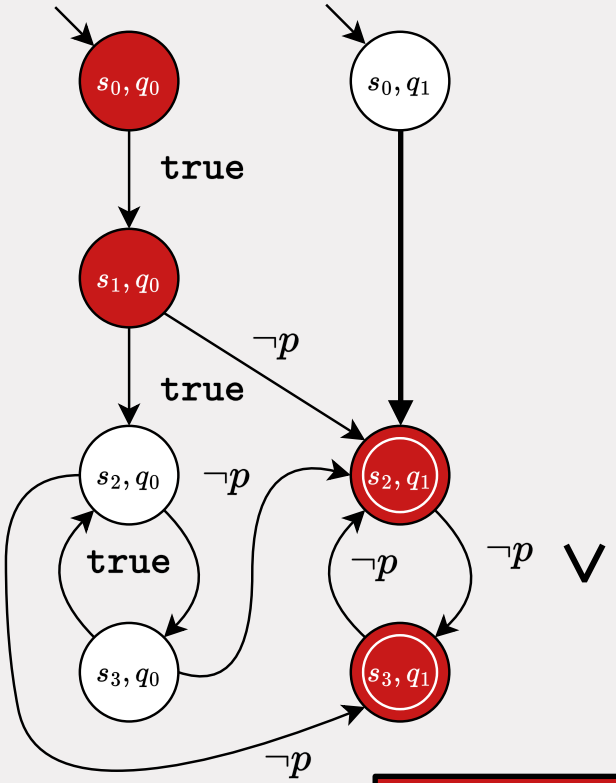
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

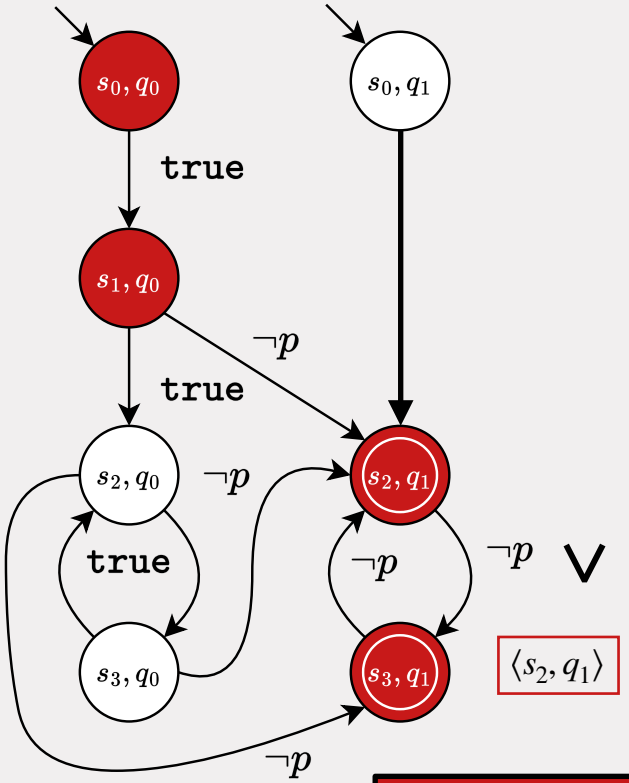
MAP algorithm



- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

MAP algorithm

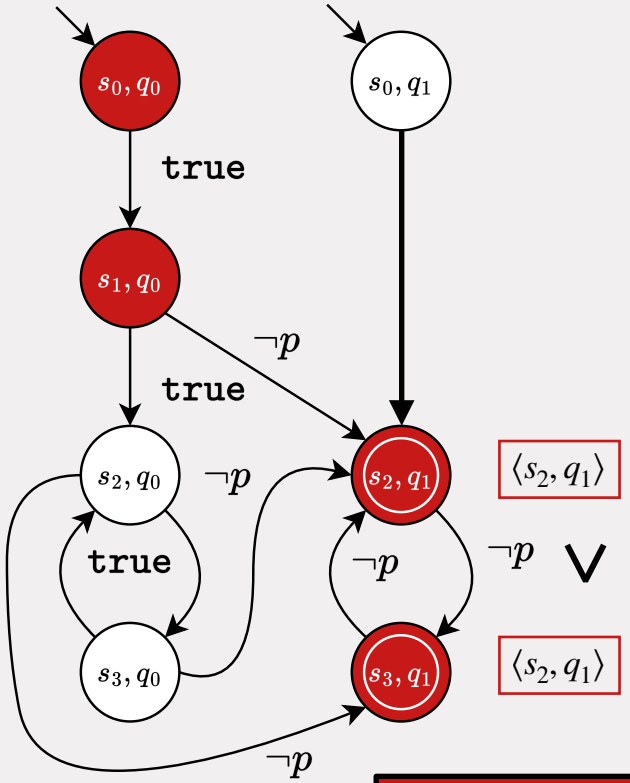


- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

TU/e

MAP algorithm

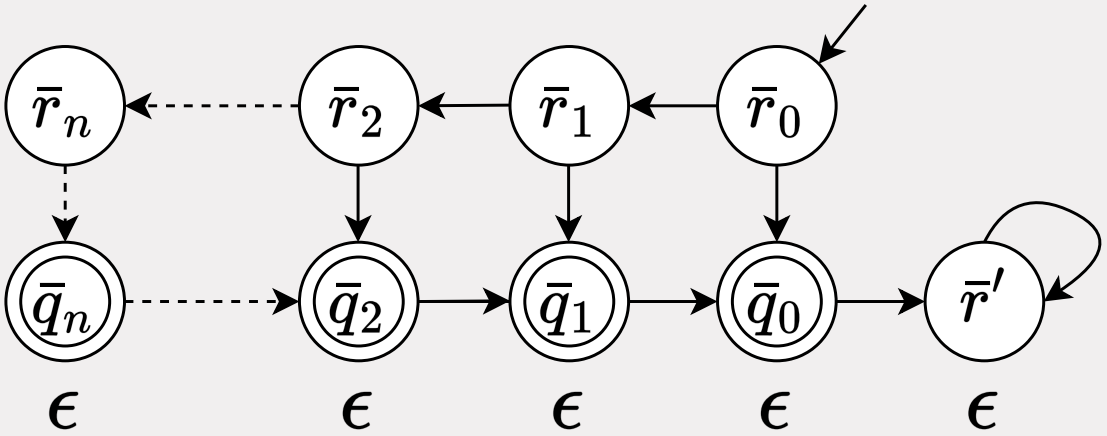


- **If the states in $\mathcal{Q}_F$ are *totally ordered***
  - For instance, by comparing the hash table addresses at which they are stored
- **then if an accepting state is its own maximal accepting predecessor, it is in a cycle**
  - In MAP, a reference to the maximal predecessor of a state is propagated along the search
- However, if a state is *not* its own MAP, we do not know anything
- Therefore, MAP works in rounds; in each round, states in $\mathcal{Q}_F$ for which it has been determined that they are not in a cycle are no longer considered accepting ($\langle s_0, q_1 \rangle$ is not in a cycle)
  - As in each round, at least one state is no longer considered accepting, MAP terminates

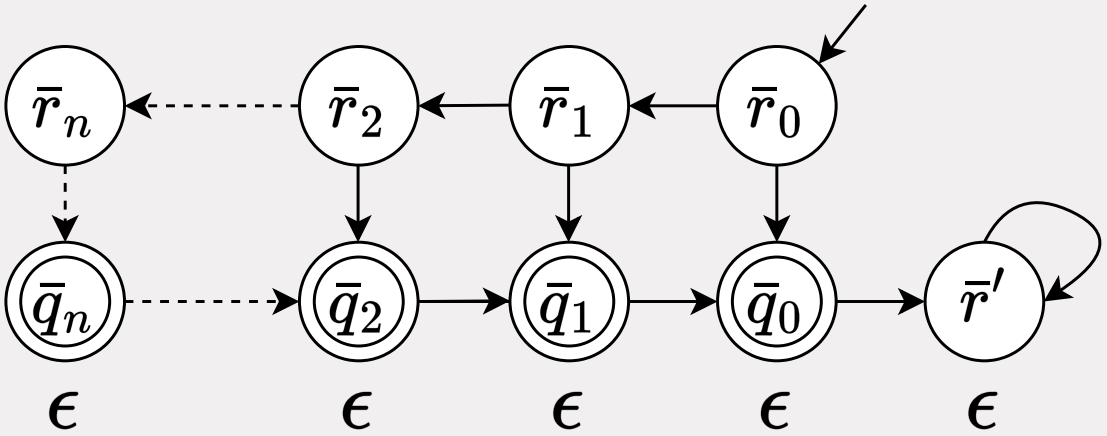
$$\langle s_0, q_1 \rangle > \langle s_2, q_1 \rangle > \langle s_3, q_1 \rangle$$

TU/e

However, MAP is very conservative



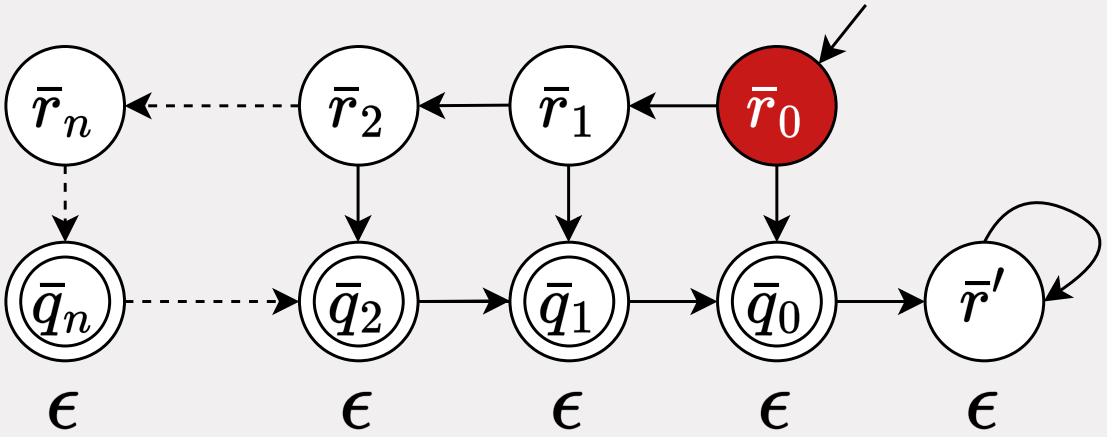
However, MAP is very conservative



- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- $p(\bar{q})$ is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with $p(\bar{q}) \neq \epsilon$ and $p(\bar{q}) \neq \bar{q}$ should be reopened in the next round

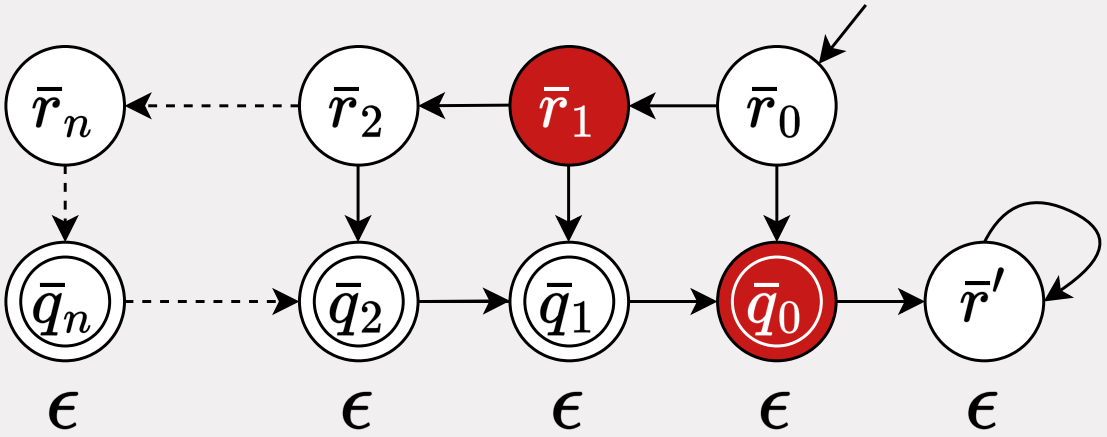
TU/e

However, MAP is very conservative



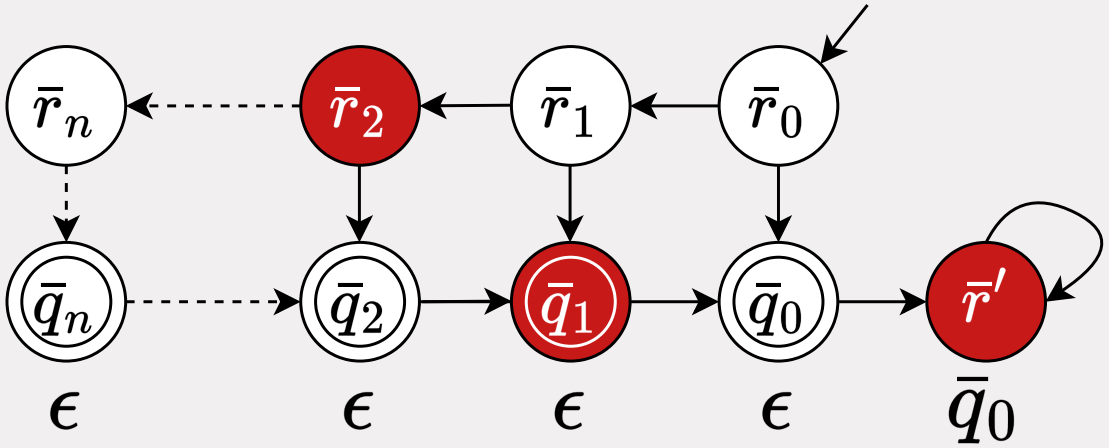
- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- $p(\bar{q})$ is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with $p(\bar{q}) \neq \epsilon$ and $p(\bar{q}) \neq \bar{q}$ should be reopened in the next round

However, MAP is very conservative



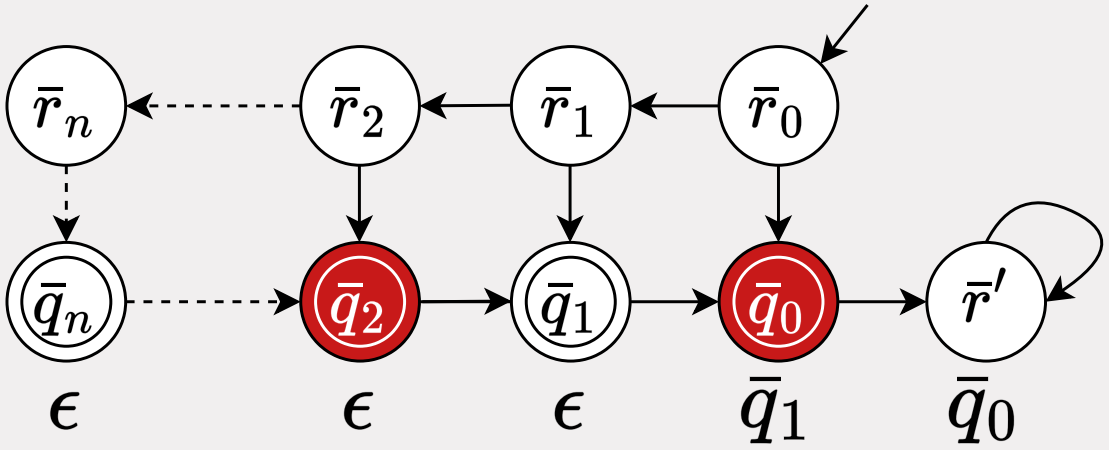
- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- $p(\bar{q})$ is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with $p(\bar{q}) \neq \epsilon$ and $p(\bar{q}) \neq \bar{q}$ should be reopened in the next round

However, MAP is very conservative



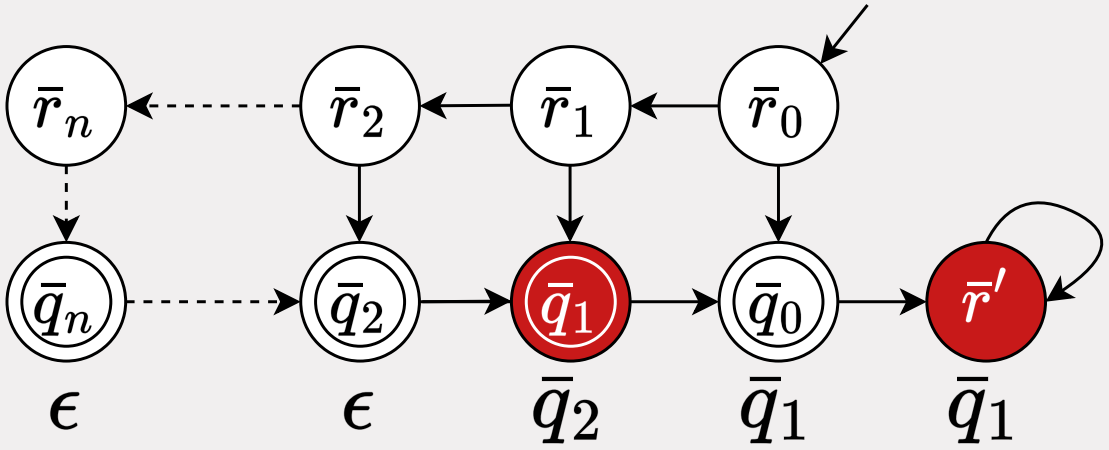
- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- $p(\bar{q})$ is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with $p(\bar{q}) \neq \epsilon$ and $p(\bar{q}) \neq \bar{q}$ should be reopened in the next round

However, MAP is very conservative



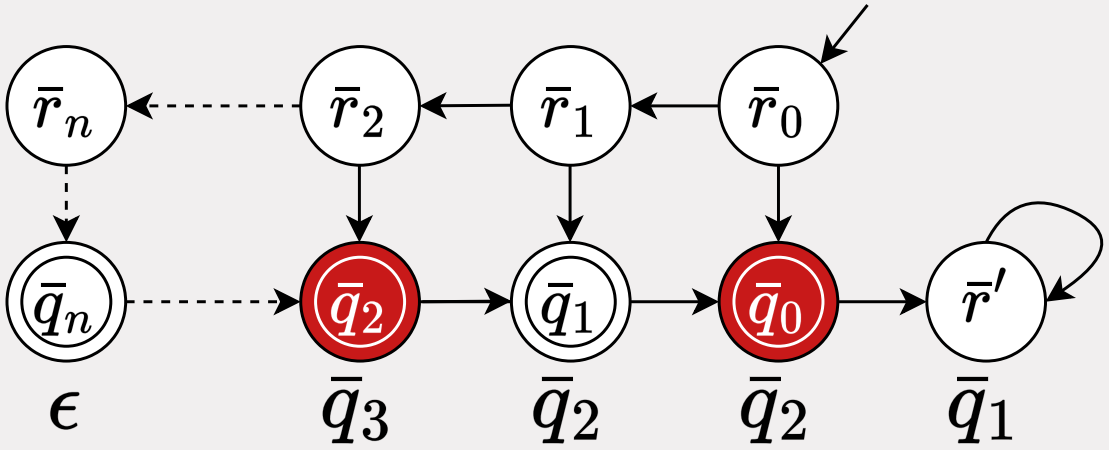
- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- $p(\bar{q})$ is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with $p(\bar{q}) \neq \epsilon$ and $p(\bar{q}) \neq \bar{q}$ should be reopened in the next round

However, MAP is very conservative



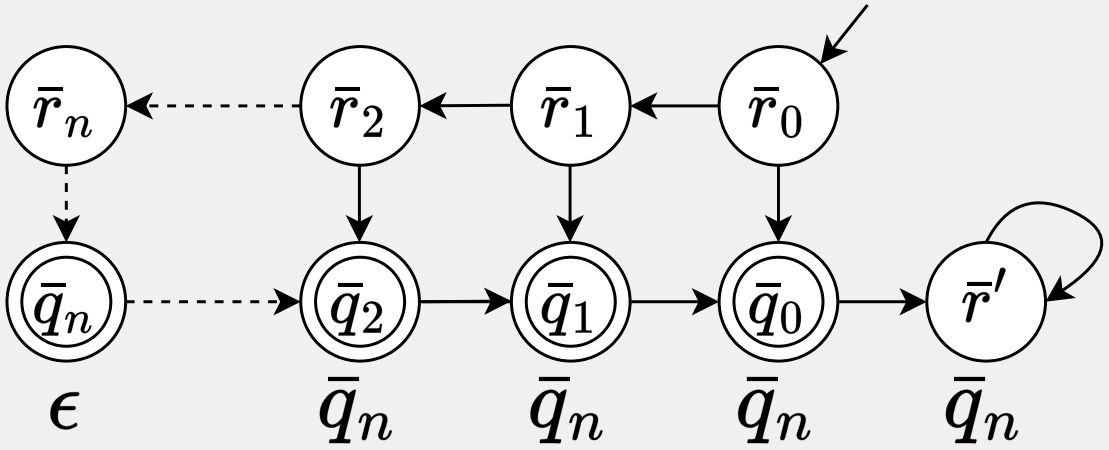
- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- $p(\bar{q})$ is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with $p(\bar{q}) \neq \epsilon$ and $p(\bar{q}) \neq \bar{q}$ should be reopened in the next round

However, MAP is very conservative



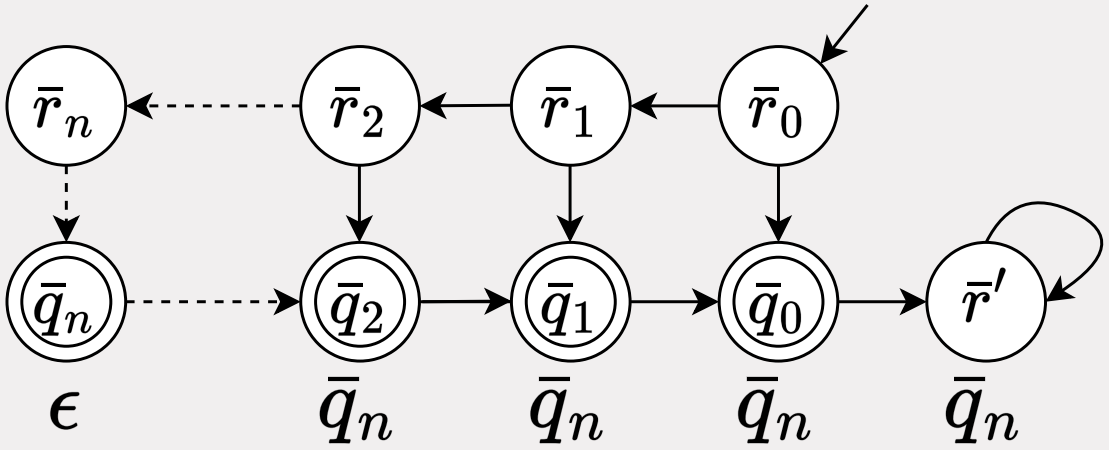
- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- $p(\bar{q})$ is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with $p(\bar{q}) \neq \epsilon$ and $p(\bar{q}) \neq \bar{q}$ should be reopened in the next round

However, MAP is very conservative



- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- p($\bar{q}$) is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with p($\bar{q}$) $\neq \epsilon$ and p($\bar{q}$) $\neq \bar{q}$ should be reopened in the next round

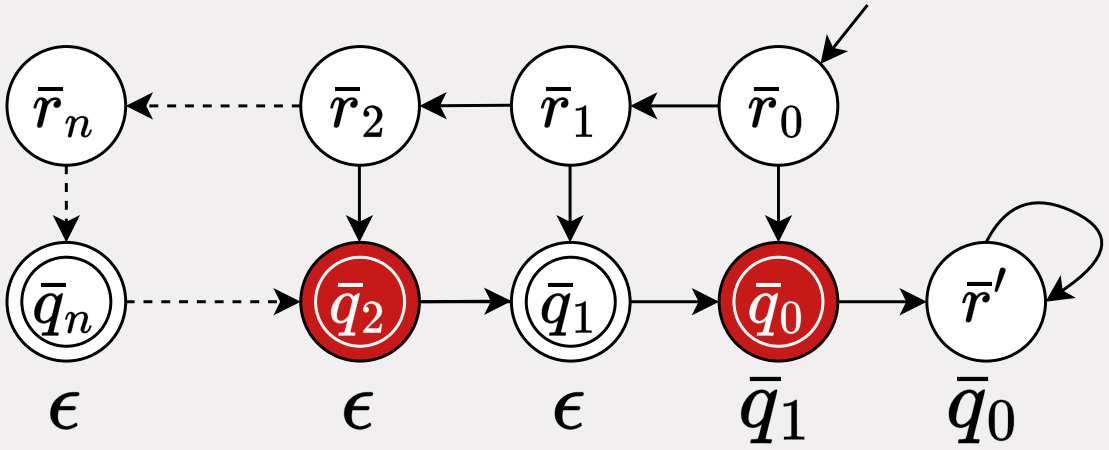
However, MAP is very conservative



- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- $p(\bar{q})$ is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with $p(\bar{q}) \neq \epsilon$ and $p(\bar{q}) \neq \bar{q}$ should be reopened in the next round

- Here, MAP requires *n* rounds
  - Can we do better?
    - Yes, **Hitchhiking**

However, MAP is very conservative



Observation: as the propagation of $\bar{q}_0$ was not interrupted, there is no need to propagate it again in the next round

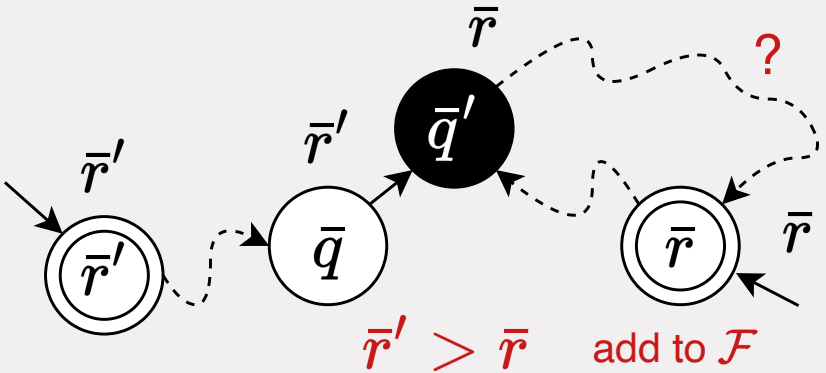
- **Let us consider a BFS**
- The $\bar{r}_i$ and $\bar{q}_i$ states are tuples
- For all $0 \leq i < j \leq n$, $\bar{q}_i < \bar{q}_j$
- $p(\bar{q})$ is current reference of $\bar{q}$
- At the end of a round, every $\bar{q} \in \mathcal{Q}_F$ with $p(\bar{q}) \neq \epsilon$ and $p(\bar{q}) \neq \bar{q}$ should be reopened in the next round

- Here, MAP requires *n* rounds
  - Can we do better?
    - Yes, **Hitchhiking**

TU/e

Hitchhiking

- Keep track of a set of *active* states $\mathcal{A}$ and a set of *interrupted* states $\mathcal{F}$
  - Initially, $\mathcal{A} = \mathcal{Q}_F$, and $\mathcal{A}$ strictly becomes smaller after each round
  - Initially, $\mathcal{F} = \emptyset$, states are added to $\mathcal{F}$ when 'their' search is interrupted
  - When a state $\bar{q} \in \mathcal{Q}_F$ is reached for the first time, we set $\mathrm{p}(\bar{q}) = \bar{q}$

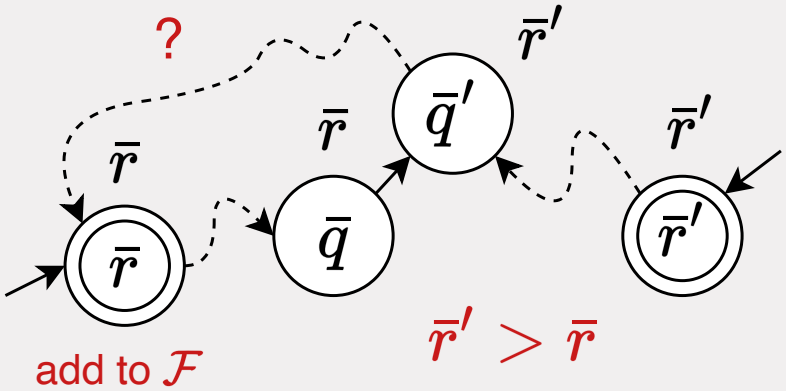


$\bar{r}$-search is interrupted; add $\bar{r}$ to $\mathcal{F}$

TU/e

Hitchhiking

- Keep track of a set of *active* states $\mathcal{A}$ and a set of *interrupted* states $\mathcal{F}$
  - Initially, $\mathcal{A} = \mathcal{Q}_F$, and $\mathcal{A}$ strictly becomes smaller after each round
  - Initially, $\mathcal{F} = \emptyset$, states are added to $\mathcal{F}$ when 'their' search is interrupted
  - When a state $\bar{q} \in \mathcal{Q}_F$ is reached for the first time, we set $p(\bar{q}) = \bar{q}$

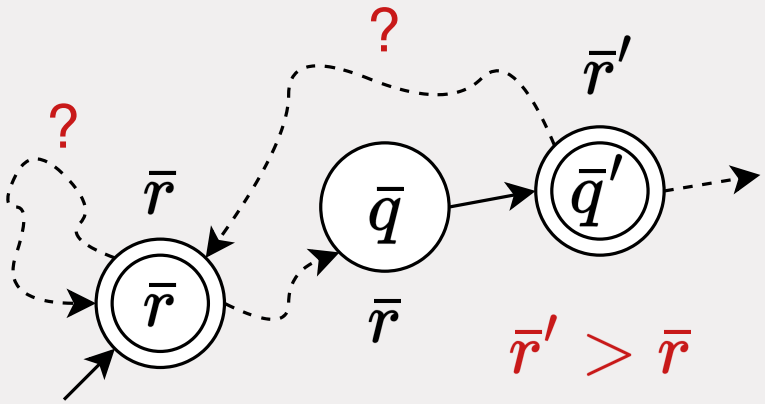


$\bar{r}$-search is interrupted; add $\bar{r}$ to $\mathcal{F}$

TU/e

Hitchhiking

- Keep track of a set of *active* states $\mathcal{A}$ and a set of *interrupted* states $\mathcal{F}$
  - Initially, $\mathcal{A} = \mathcal{Q}_F$, and $\mathcal{A}$ strictly becomes smaller after each round
  - Initially, $\mathcal{F} = \emptyset$, states are added to $\mathcal{F}$ when 'their' search is interrupted
  - When a state $\bar{q} \in \mathcal{Q}_F$ is reached for the first time, we set $\mathrm{p}(\bar{q}) = \bar{q}$



$\bar{r}$-search is interrupted; However, do *not* add $\bar{r}$ to $\mathcal{F}$

- If $\bar{q}' = \bar{r}'$, the $\bar{q}'$-search may detect cycle
- If $\bar{q}'$-search is interrupted, either $\bar{q}'$ is added to $\mathcal{F}$, or same situation applies for $\bar{q}'$ (but as cycle is finite and $\mathcal{A}$ is totally ordered, this cannot be applicable for all involved states in $\mathcal{A}$)

TU/e

Hitchhiking — Postprocessing at end of round

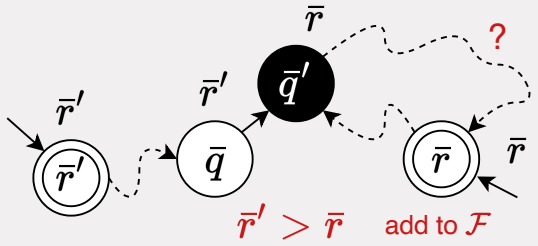
- $\mathcal{O}$: set of *open* states (to be explored)

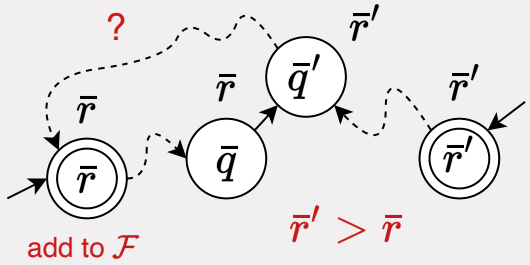
if $\mathcal{F} \neq \emptyset$ **then**
  for all $\bar{r} \in \mathcal{Q}_{\otimes}$ **do in parallel**
    if $\bar{r} \in \mathcal{A}$ **then**
      if $\bar{r} \in \mathcal{F}$ **and** $\mathrm{p}(\bar{r}) \neq \bar{r}$ **then** $\mathcal{O} \leftarrow \mathcal{O} \cup \{\bar{r}\}$
      else $\mathcal{A} \leftarrow \mathcal{A} \setminus \{\bar{r}\}$
      if $\bar{r} \in \mathcal{F}$ **then** $\mathcal{F} \leftarrow \mathcal{F} \setminus \{\bar{r}\}$
    if $\bar{r} \in \mathcal{A}$ **then** $\mathrm{p}(\bar{r}) \leftarrow \bar{r}$ **else** $\mathrm{p}(\bar{r}) \leftarrow \epsilon$

Hitchhiking — Postprocessing at end of round

- $\mathcal{O}$: set of *open* states (to be explored)

if $\mathcal{F} \neq \emptyset$ **then**
  for all $\bar{r} \in \mathcal{Q}_{\otimes}$ **do in parallel**
    if $\bar{r} \in \mathcal{A}$ **then**
      if $\bar{r} \in \mathcal{F}$ **and** $\mathrm{p}(\bar{r}) \neq \bar{r}$ **then** $\mathcal{O} \leftarrow \mathcal{O} \cup \{\bar{r}\}$
      else $\mathcal{A} \leftarrow \mathcal{A} \setminus \{\bar{r}\}$
      if $\bar{r} \in \mathcal{F}$ **then** $\mathcal{F} \leftarrow \mathcal{F} \setminus \{\bar{r}\}$
    if $\bar{r} \in \mathcal{A}$ **then** $\mathrm{p}(\bar{r}) \leftarrow \bar{r}$ **else** $\mathrm{p}(\bar{r}) \leftarrow \epsilon$

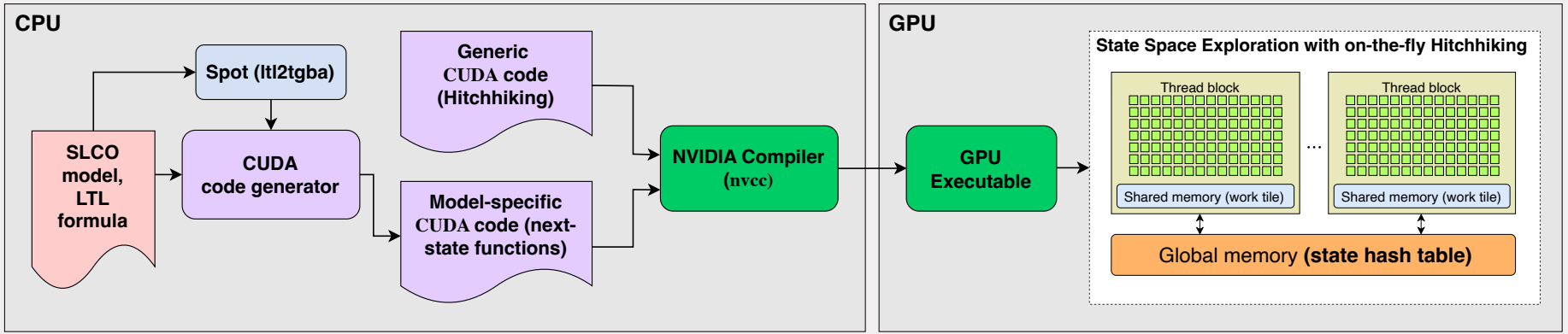




If $\mathrm{p}(\bar{r})$ was never updated, then $\bar{r}$ cannot be in a cycle!

TU/e

Hitchhiking implementation in GPUexplore 3.0

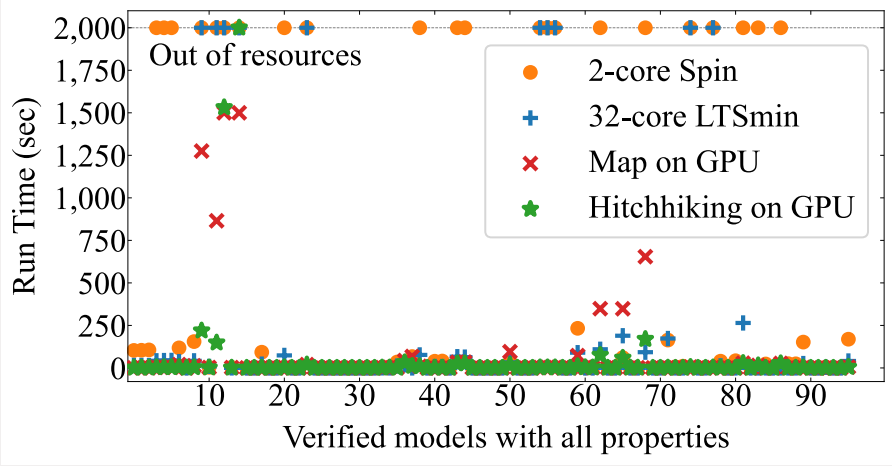


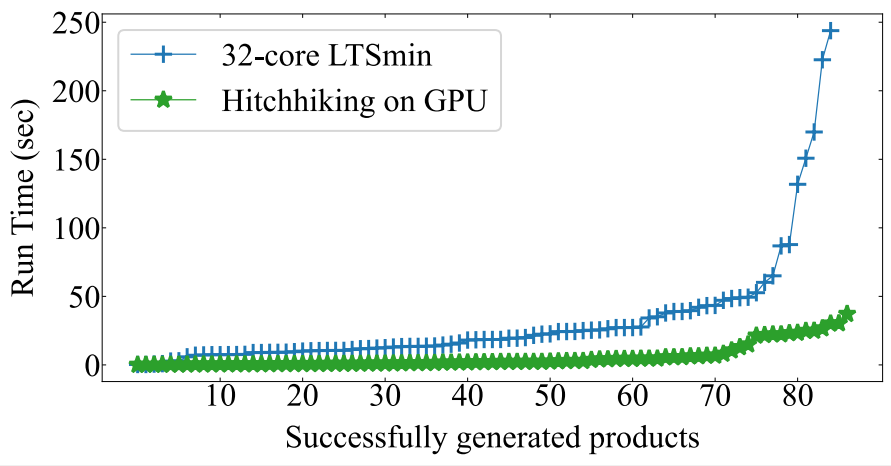
- **Spot library** used to generate NBAs **[Duret-Lutz *et al.*, 2022]**
- **CUDA C++ code generator** developed in Python + TextX + Jinja2

Experiments

- Compare Hitchhiking with MAP on GPU
  - Existing GPU MAP implementation (not on-the-fly) no longer maintained
  - We added MAP (on-the-fly) to GPUexplore 3.0
- Compare Hitchhiking with state-of-the-art CPU algorithms
  - 2-core NDFS in the SPIN model checker **[Holzmann & Bošnački, 2007]**
  - *n*-core Combined NDFS (CNDFS) in the LTSmin model checker **[Evangelista *et al.*, 2012]**
  - All tools use state compression, but imprecise methods (bit state hashing, …) have been disabled
- **Benchmarks:** 32 (translated) models from the BEEM benchmark suite **[Pelánek, 2007]**
  - Some scaled up to make them more interesting for parallel model checking
    - State spaces ranging between 150,000 and 1.2 billion states
  - Per model, 3 LTL formulae were checked

Experiments





- **Used hardware:**
  - **GPU:** Titan RTX GPU: 4,608 cores (1.35 GHz), 24 GB memory, 2018
    - Generated code compiled with CUDA C++ 12.2 compiler
  - **CPU:** 32-core AMD EPYC 7R13 (2.65 GHz), 2021
    - Out of memory set at 32 GB

Table 1: Runtime (sec.) of LTL checking on GPU vs. contemporary multi-core tools. : Model was altered to remove deadlocks. m.: Out of memory. t.: Time out. : φ is *satisfied*. : φ is *not satisfied*. ✗: Incorrect result. Times in bold indicate that HITCHHIKING was faster than SPIN and LTSMIN. Significantly worse MAP times (at least 0.1 second slower) are marked ▼.

Model	SPIN (2-core NDFS)			LTSMIN (1-core CNDFS)			LTSMIN (32-core CNDFS)			GPUEXPLORE HITCHHIKING			GPUEXPLORE MAP		
	φ_1	φ_2	φ_3	φ_1	φ_2	φ_3	φ_1	φ_2	φ_3	φ_1	φ_2	φ_3	φ_1	φ_2	φ_3
adding.20+	103	104	106	110.7	110.8	112.8	14.65	14.47	14.2	**0.88**	**0.88**	**0.899**	0.879	0.91	0.898
adding.50+	m.	m.	m.	677.5	673.3	667.2	42.24	43.55	44.34	**3.66**	**3.67**	**3.679**	3.663	3.684	3.679
anderson.6	118	0.17	155	162.4	2.62	195.2	37.43	0.02	41.95	**4.57**	0.168	**3.238**	▼ 19.704	0.168	▼ 12.85
anderson.7	m.	3.9	m.	m.	50.63	m.	m.	9.07	m.	**219**	**1.963**	**148.39**	▼ 1275	▼ 2.381	▼ 865.2
anderson.8	m.	3.84	m.	m.	29.11	m.	m.	0.08	m.	**1531**	0.945	m.	▼ t.	▼ 2.062	m.
at.5	0.06	0.06	92.5	0.001	0.001	120.3	0.001	0.001	22.53	0.144	0.001	**0.6**	0.145	0.001	0.602
at.6	0.06	0.06	m.	0.001	0.001	911.1	0.001	0.01	73.74	0.148	0.001	**2.706**	0.148	0.001	2.708
at.7	0.06	0.06	m.	0.001	0.001	m.	0.001	0.001	m.	0.165	0.001	**19.63**	0.167	0.001	19.67
bakery.5	0.05	0.05	0.04	0.001	0.001	0.001	0.01	0.001	0.001	0.042	0.042	0.041	0.042	0.045	0.044
bakery.6	0.05	0.05	0.05	0.001	0.001	0.001	0.01	0.001	0.001	0.042	0.043	0.047	0.043	0.044	0.068
bakery.7	0.04	0.05	0.05	0.001	0.001	0.001	0.01	0.001	0.001	0.041	0.042	0.042	0.043	0.042	0.042
elevator2.3	3.21	5.87	34.9	2.5	0.01	34.24	0.46	0.04	11.28	1.63	1.965	**0.608**	1.678	▼ 2.946	0.633
elevator2.4+	32.5	66.9	m.	33.94	0.001	570.1	18.21	0.05	76.64	32.39	11.18	**9.083**	▼ 43.66	▼ 67.57	▼ 9.432
frogs.4	0.07	39.5	40.6	0.001	43.16	42.23	0.001	8.65	8.42	0.052	**0.69**	**0.698**	0.062	0.698	0.699
frogs.5	0.05	m.	m.	0.001	1141	1105	0.01	66.27	63.75	0.046	**26.1**	**26.4**	0.046	▼ 33.6	▼ 34.49
lamport.6	0.05	0.05	0.04	0.001	0.001	0.001	0.01	0.01	0.001	0.043	0.043	0.042	0.046	0.043	0.042
lamport.7	0.05	0.05	36.4	0.01	0.01	289.7	0.01	0.001	33.2	0.192	0.204	**17.02**	0.193	0.204	▼ 95.82
lamport.8	0.05	0.05	0.05	0.001	0.001	0.001	0.01	0.001	0.001	0.043	0.042	0.042	0.043	0.043	0.042
loyd.3	m.	m.	m.	m.	m.	m.	m.	m.	m.	**2.59**	**2.6**	**2.627**	2.59	2.6	2.631
mcs.5	0.12	0.12	233	0.001	1.02	m.	0.01	0.001	89.8	0.219	0.229	**9.915**	0.22	0.229	▼ 72.56
peterson.5	0.12	0.12	m.	0.001	0.001	1449	0.02	0.02	109.9	3.425	2.094	**75.78**	3.425	2.094	▼ 348.6
peterson.6	0.07	0.07	60.6	0.001	0.001	m.	0.01	0.01	189.4	0.54	0.543	**58.85**	0.565	0.54	▼ 348
peterson.7	0.18	0.17	m.	0.04	0.001	1232	0.06	0.04	92.64	1.202	0.96	169.4	1.198	0.961	▼ 654.2
phils.6	1.1	1.36	164	0.001	0.001	190.6	0.001	0.001	173.1	0.041	0.031	**1.156**	0.042	0.042	1.155
phils.7	5.99	10.3	m.	0.001	0.02	m.	0.001	0.001	m.	0.001	0.031	**5.156**	0.001	0.034	▼ 5.228
phils.8	3.09	2.58	m.	0.001	0.01	m.	0.01	0.001	m.	0.001	0.026	**2.443**	0.041	0.028	2.484
telephony.4	39.7	0.05	42.4	59.81	0.001	✗	12.51	0.001	✗	**0.44**	0.051	**0.265**	0.442	0.052	0.265
telephony.5	m.	0.05	m.	m.	0.001	✗	264.5	0.01	✗	**28.5**	0.066	**14.6**	28.45	0.067	▼ 14.91
telephony.6	23.3	0.05	m.	0.04	0.001	✗	0.05	0.01	✗	0.314	0.057	**26.56**	0.324	0.058	▼ 26.63
szymanski.5	25.1	24.3	152	10.62	0.01	268	0.01	0.01	25.93	0.287	0.555	**2.363**	▼ 1.122	0.555	2.388
leader_filters.5	0.05	0.05	5.34	0.001	0.001	7.46	0.01	0.01	3.71	0.058	0.07	**0.145**	0.054	0.07	0.151
leader_filters.6	0.05	0.05	169	0.001	0.001	583	0.01	0.01	40.93	0.051	0.045	**5.294**	0.051	0.045	5.336

Conclusions and future work

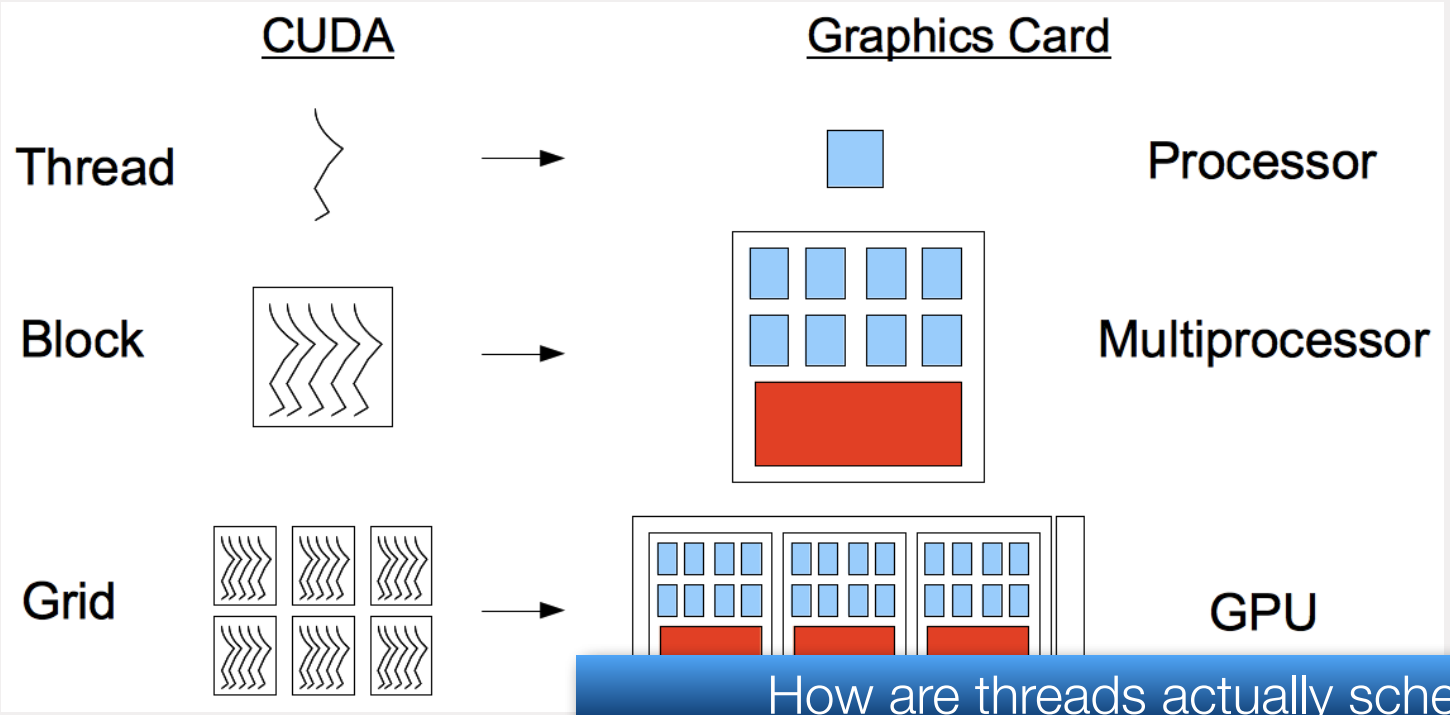
- We presented **Hitchhiking**, a new algorithm for massively parallel LTL model checking
  - Based on MAP, but keeps track of search interruptions
- Implemented in GPUexplore 3.0
  - Speedups up to 150x compared to 32-core CNDFS LTSmin
- First time GPU accelerated LTL model checking has been applied to state spaces with **more than a billion states**

- **Future work:**
  - Shortest counter-example construction
  - Other temporal logics (CTL)
  - Probabilistic Model Checking (NWO project)

Beyond the CUDA Programming model

Implementation of model

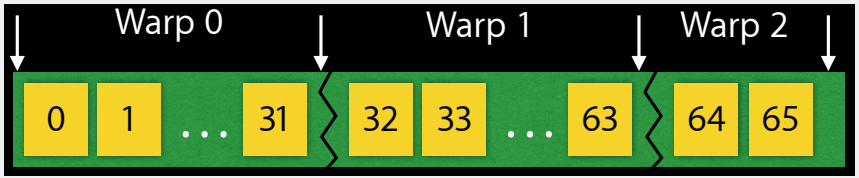
Implementation of model

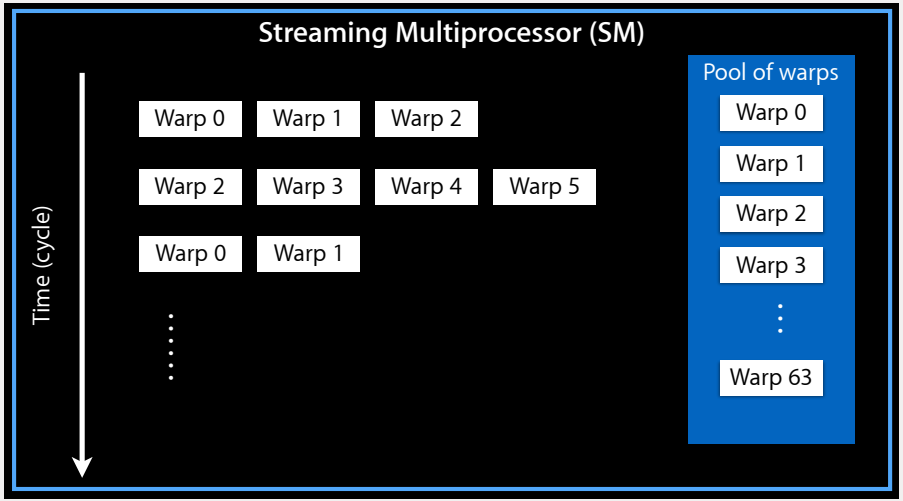


How are threads actually scheduled?

Warps as Scheduling Units

- Each block is executed as a set of 32-thread **warps**
- Warps are scheduling units in SMs
- The threads in a warp execute in **lockstep**: SIMT (single instruction multiple threads)
- Be aware of warps when optimising code, to limit:
  - thread divergence
  - uncoalesced memory accesses





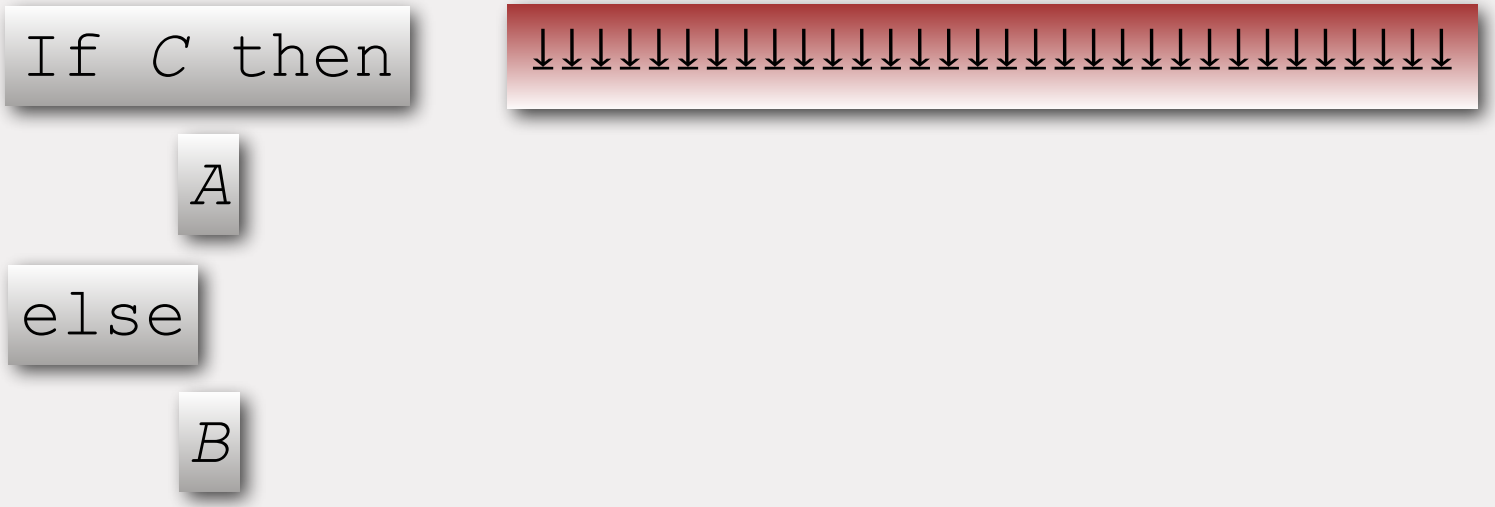
[Images by Ming Yang]

Thread divergence

- If executions of threads in a warp branch in different directions, they have to wait for each other

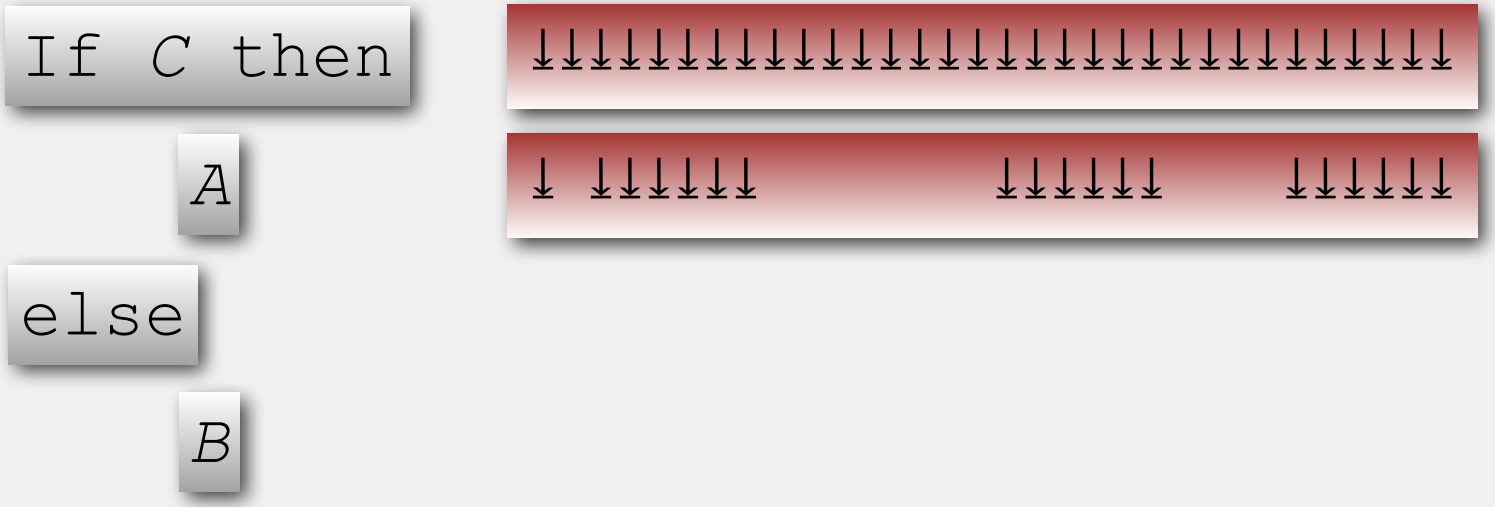
Thread divergence

- If executions of threads in a warp branch in different directions, they have to wait for each other



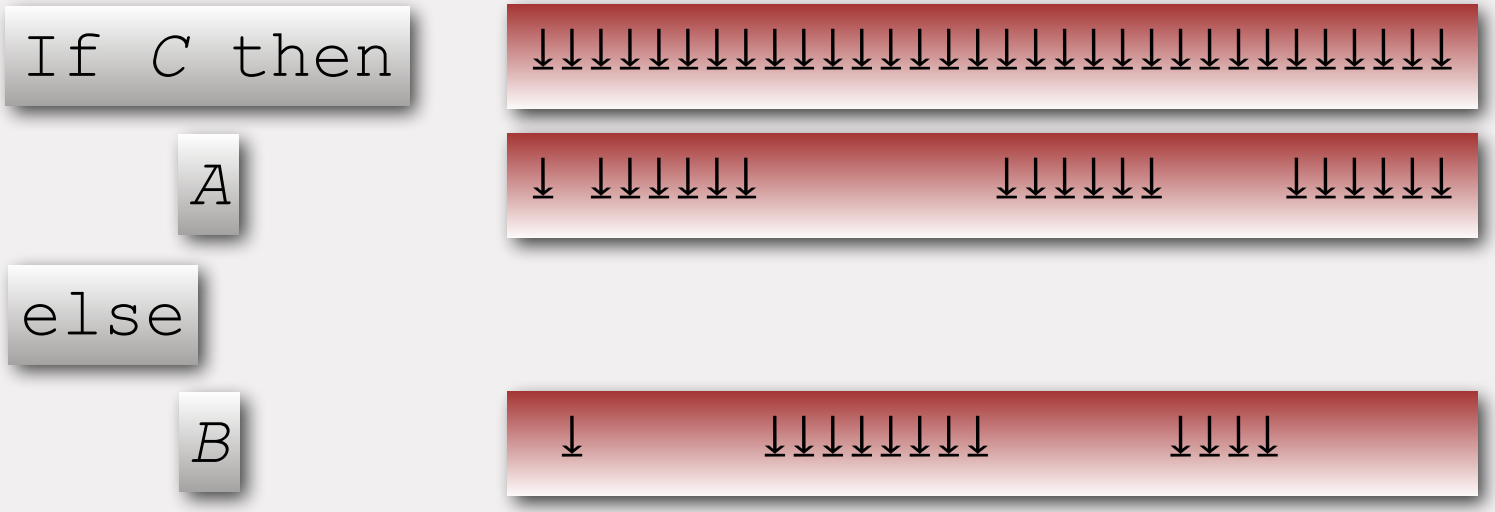
Thread divergence

- If executions of threads in a warp branch in different directions, they have to wait for each other



Thread divergence

- If executions of threads in a warp branch in different directions, they have to wait for each other



Thread divergence

- Avoid identical computations in different branches

```
if(tid % 2) {
    s += 1.0f/tid;
}
else {
    s -= 1.0f/tid;
}
```

```
float t = 1.0f/tid;
if(tid % 2) {
    s += t;
}
else {
    s -= t;
}
```

Thread divergence and program correctness

- Synchronisation barriers should be called by either none or all of the threads
- They **cannot** be called in a divergent block

```
if(p) {
   ...
   // Sync here?
   ...
}
```

```
if(p) {
   ...
}
__syncthreads();
if(p) {
   ...
}
```

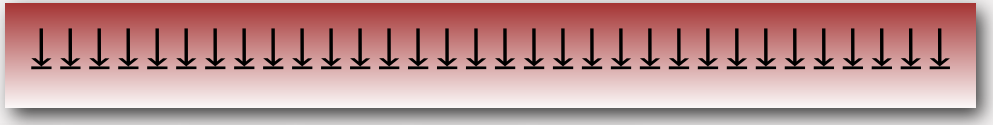
Uncoalesced memory accesses

- Memory accesses of threads in a warp are combined as much as possible

Uncoalesced memory accesses

- Memory accesses of threads in a warp are combined as much as possible

```
y = R[TId*2]
```

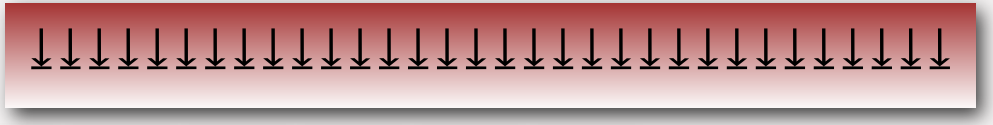


serialised to **2 accesses**, each fetching 16 entries

Uncoalesced memory accesses

- Memory accesses of threads in a warp are combined as much as possible

```
y = R[TId*2]
```



serialised to **2 accesses**, each fetching 16 entries

```
y = R[TId]
```

only **1 access**

TU/e

Occupancy

- Occupancy = # of active warps / Maximum number of resident warps per SM

Technical Specifications	Compute Capabilities						
	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum number of resident warps per SM	48	64					

- Occupancy limiters:
  - Register usage
  - Shared memory usage
  - Block size

[Based on slides by Ming Yang]

Occupancy limiter: Register usage

- Example 1 (capability = 3.0)
- Kernel uses 21 registers per thread
- # of active threads = 65536 / 21 ≈ 3120
  - # of warps = 3120 / 32 = 97
  - > 2048 / 32 = 64, therefore occupancy of 100%

Technical Specifications	Compute Capabilities						
	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum number of 32-bit registers per thread block	32 K	**64 K**						32 K
Maximum number of resident threads per SM	1536	**2048**						

Occupancy limiter: Register usage

- Example 2 (capability = 3.0)
- Kernel uses 64 registers per thread
- # of active threads = 65536 / 64 = 1024
  - # of warps = 1024 / 32 = 32
  - Occupancy = 32 / 64 = 50%

Technical Specifications	Compute Capabilities						
	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum number of 32-bit registers per thread block	32 K	64 K						32 K
Maximum number of resident threads per SM	1536	2048						

Occupancy limiter: Shared memory

- Example 1 (capability = 3.0)
- Kernel uses 16 bytes shared memory per thread
- # of active threads = 49152 / 16 = 3072
  - # of warps = 3072 / 32 = 96
  - > 64, therefore occupancy of 100%

Technical Specifications	Compute Capabilities							
	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum amount of shared memory per SM	**48 KB**				112 KB	64 KB	96 KB	64 KB
Maximum number of resident threads per SM	1536	**2048**						
Maximum number of resident warps per SM	48	**64**						

Occupancy limiter: Shared memory

- Example 1 (capability = 3.0)
- Kernel uses 32 bytes shared memory per thread
- # of active threads = 49152 / 32 = 1536
  - # of warps = 1536 / 32 = 48
  - Occupancy = 48 / 64 = 75%

Technical Specifications	Compute Capabilities							
	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum amount of shared memory per SM	**48 KB**				112 KB	64 KB	96 KB	64 KB
Maximum number of resident threads per SM	1536	**2048**						
Maximum number of resident warps per SM	48	**64**						

Occupancy

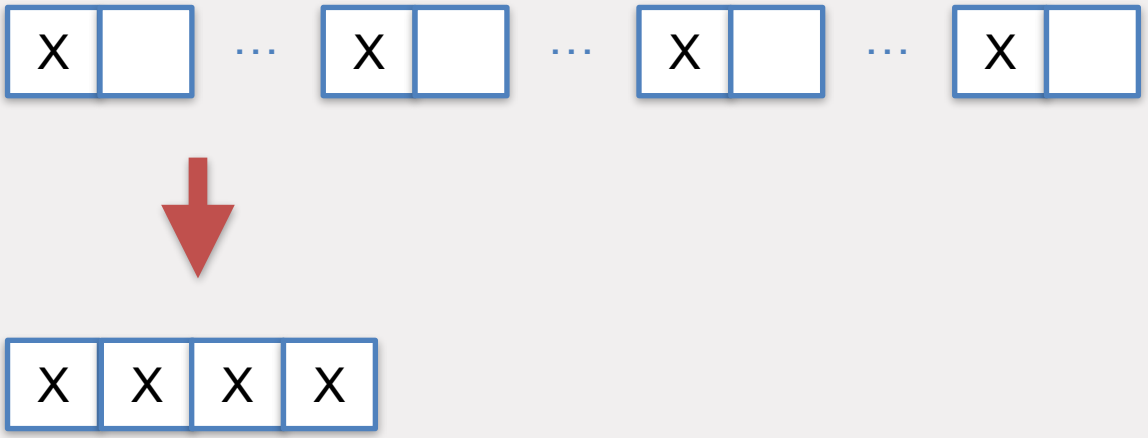
- Do we want higher occupancy?
  - Yes. Latency can be hidden better when more threads are running

- Is occupancy a metric of performance?
  - No! It is just one of the contributing factors
  - There are examples where lowering the occupancy positively affects performance
    - Matrix multiplication and FFT

Fifth hands-on session

- Go to folder **`5-matrix-sum`**, look at the source files
- Task:
  - Restructure the input data, and rewrite the kernel for matrix row summation, to achieve coalesced memory accesses
- Hints:
  - To each row, one thread is assigned
  - Consider which elements in the input array need to be accessed by a single thread

Hint



Revisiting BFS

Consider *stepsize* = 32

Require: initial state is in search frontier
Ensure: if state i is in search frontier, then the successors of i are added to search frontier, and i is moved to the explored set

```
    stepsize ← 1
 2: for (i ← Global-ThreadId; i < |V|; i ← i + NrOfThreads) do
        srcinfo ← offsets[i]
 4:     if inFrontier(srcinfo) then
            offsets[i] ← moveToExplored(srcinfo)
 6:         offset1 ← getOffset(srcinfo)
            offset2 ← getOffset(offsets[i + stepsize − (i mod stepsize)])
 8:         for (j ← offset1; j < offset2; j ← j + stepsize) do
                t ← trans[j]
10:             if t ≠ empty then
                    tgtstate ← getTgtstate(t)
12:                 tgtinfo ← offsets[tgtstate]
                    if isNew(tgtinfo) then
14:                     offsets[tgtstate] ← addToFrontier(tgtinfo)
```

Suggestion for data structures

- Design such that accesses are coalesced most of the time
- For instance, often better to have *structure of arrays* than *array of structures*

```
struct Pt {
      float x;
      float y;
      float z;
};

struct Pt myPts[N];
```

```
struct Pt {
      float x[N];
      float y[N];
      float z[N];
};

struct Pt myPts;
```

Suggestion for data structures

- Design such that accesses are coalesced most of the time
- For instance, often better to have *structure of arrays* than *array of structures*

```
struct Pt {
      float x;
      float y;
      float z;
};

struct Pt myPts[N];
```

```
struct Pt {
      float x[N];
      float y[N];
      float z[N];
};

struct Pt myPts;
```

Consider access pattern when each thread ti computes myPts[ti].x + myPts[ti].y + myPts[ti].z

TU/e

Inter-thread/inter-warp communication

- Barriers (`__syncthreads()`)
  - Easy to use
  - Coarse-grained at block level
- Atomics
  - Fine-grained
  - Can be used to implement wait-free algorithms
- Communication via memory
  - Beware of memory consistency

[Based on slides by Sylvain Collange]

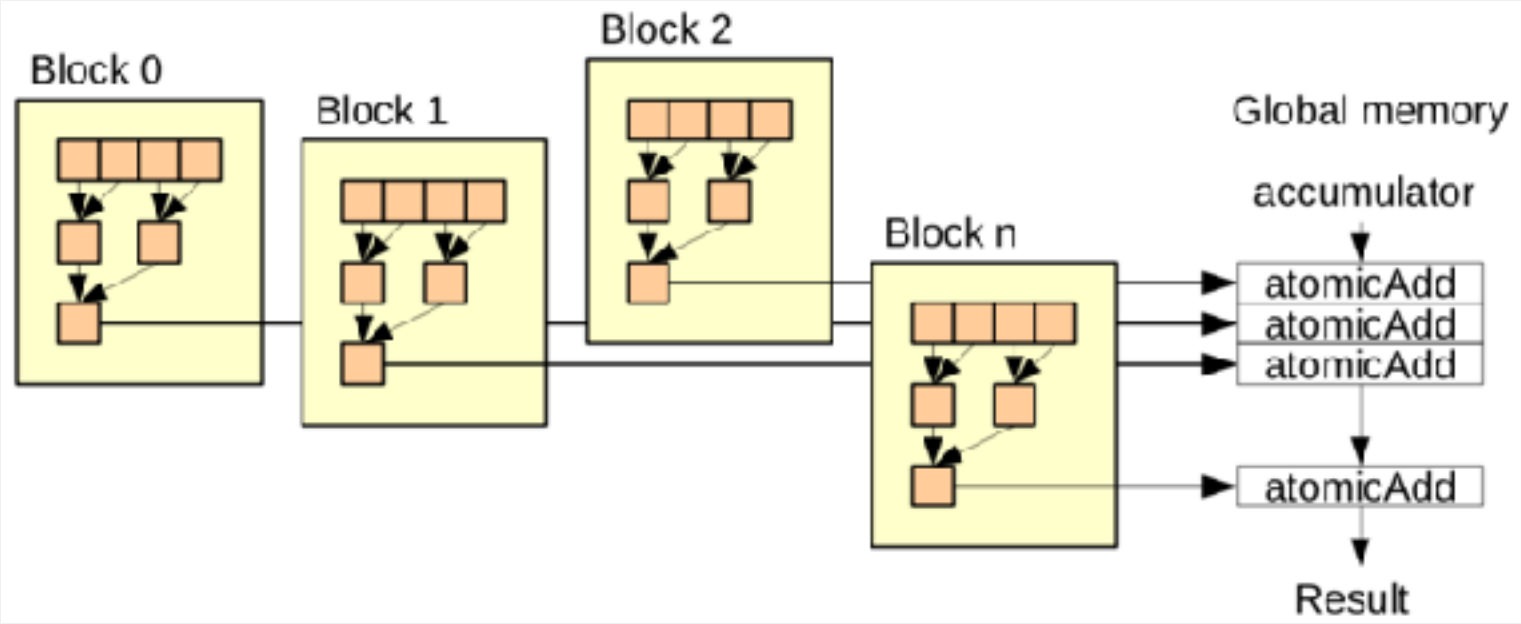
Atomics

- Read, modify, write in one operation
  - Cannot be interleaved with accesses of other threads
- Available operators (both 32-bit and 64-bit):
  - `atomicAdd, atomicSub, atomicInc, atomicDec`
  - `atomicMin, atomicMax`
  - `atomicExch, atomicCAS`
  - `atomicAnd, atomicOr, atomicXor`
- Both for global and shared memory

- Beware for performance!
  - Atomic operations to the same memory address are serialised

Example: reduction

- Each block performs local reduction of data, atomics are used to accumulate the result



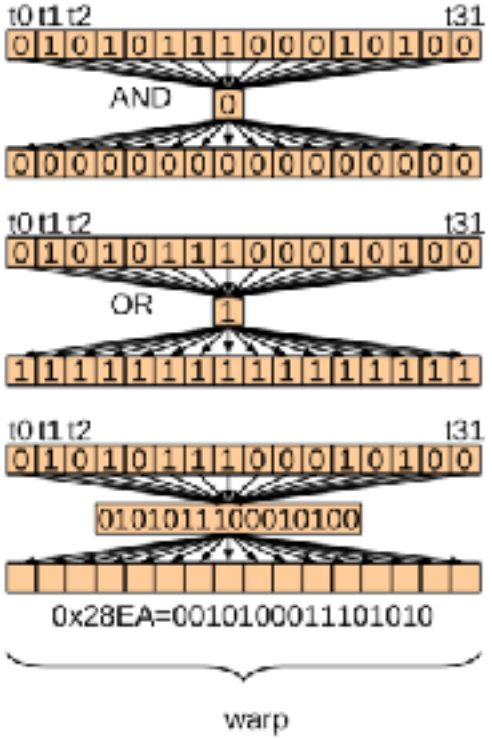
Warp-synchronous programming

- Threads in a warp run synchronously, therefore no need to synchronise them explicitly
- Warps can be treated as threads with SIMD capabilities, similar to threads on a Xeon Phi, for instance
- Special **intra-warp instructions** can be used to efficiently exchange data among threads in a warp

Warp vote instructions

- `p2 = __all(p1)`
  horizontal AND between predicates p1 of all threads in the warp
- `p2 = __any(p1)`
  OR between all p1
- `n = __ballot(p)`
  Set bit *i* of integer n to value of p for thread *i*
  i.e. get bit mask as an integer

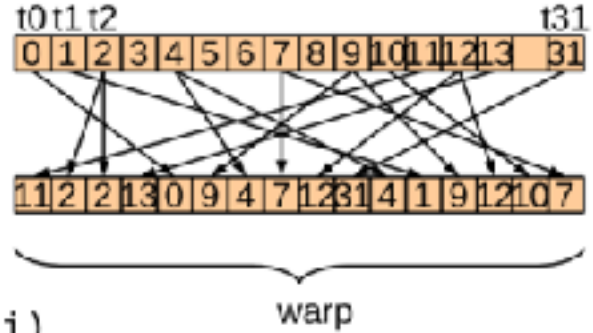
Like __syncthreads_{and,or} for a warp
Use: take control decisions for the whole warp



Warp shuffle instructions

Exchange data between lanes

- `__shfl(v, i)`
  Get value of thread i in the warp
  - Use: 32 concurrent lookups in a 32-entry table
  - Use: arbitrary permutation...



- `__shfl_up(v, i)` ≈ `__shfl(v, tid-i)`,
  `__shfl_down(v, i)` ≈ `__shfl(v, tid+i)`
  Same, indexing relative to current lane
  - Use: neighbor communication, shift
- `__shfl_xor(v, i)` ≈ `__shfl(v, tid ^ i)`
  - Use: exchange data pairwise: "butterfly"

Other features

- NVIDIA GPUs implement a relaxed memory consistency model
  - *Thread fences* to enforce explicit ordering of memory accesses
  - Declare shared variables as `volatile`
  - `__syncthreads()` implies thread fence

- New features are added frequently
  - For instance, CUDA 9 (end 2017) introduced *cooperative groups*

Optimising Code

- Moving data around is more expensive than computing on it
- Start with a simple algorithm and keep it for readability and correctness checks
- Optimize only when needed
- Focus on the bottlenecks first (compute-bound or memory-bound)
- Auto-tune (automatically explore the parameter space)
  - Different loop orderings
  - Different tile sizes, on multiple levels L3, L2, and L1
  - Different number of threads, thread blocks, vector lengths, etc
  - e.g. using the Kernel Tuner (https://github.com/benvanwerkhoven/kernel_tuner)

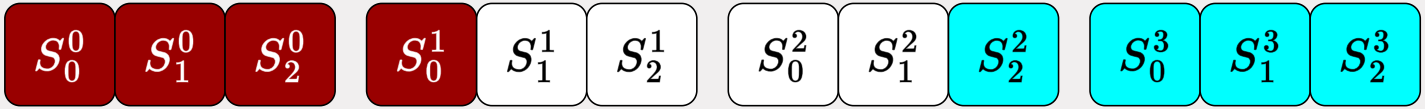
Optimisation in GPUexplore: different work distribution

Example: three FSMs

S_i^j: state of FSM i in state j

Colour indicates warp (size 4)

Thread-to-FSM work distribution:



Warp-to-FSM work distribution:

- GPUexplore 2 work distribution *Thread-to-FSM*
  - If threads focussing on the same state reside in the same warp, they can efficiently communicate
  - **Network of LTSs:** Each thread does (more or less) the same operation, so no thread divergence
  - **SLCO models:** each thread will call a different function, so **thread divergence!**
- GPUexplore 3 work distribution *Warp-to-FSM*
  - Now threads in the same warp focus on the same process, so they execute the same function
  - **Threads diverge on the state of their process. Can we reduce this effect?**
  - Sort the states based on the state of this process
    - Hou *et al.*. *Fast Segmented Sort on GPUs*, ICS (2017): **intra-warp bitonic merge-sort**

TU/e

GPU hash tables

- Hash table implemented as (integer) array with hash function(s)
- To achieve coalesced accesses, partition hash table into *buckets* consisting of 32 integers
  - Let a warp handle the lookup / insertion of an element

```
bool cooperative_find(int e, int *T) {
  int warptid = threadIdx.x % 32;
  int bucketid = h(e);
  int entry = h(e) * 32 + warptid;
  return ballot(entry == e);
}
```

GPU hash tables

- Hash table implemented as (integer) array with hash function(s)
- To achieve coalesced accesses, partition hash table into *buckets* consisting of 32 integers
  - Let a warp handle the lookup / insertion of an element

```
int cooperative_insert(int e, int *T) {
  int warptid = threadIdx.x % 32;
  int bucketid = h(e);
  int entry = h(e) * 32 + warptid;
  int e_lane = __ffs(ballot(entry == EMPTY));
  if (warptid == e_lane) {
    atomicExch(h(e) * 32 + warptid, e);
  }
  return h(e) * 32 + e_lane;
}
```

Wrap-up

- GPUs provide **enormous potential to accelerate computations**
  - **In formal verification:** bisimulation checking, BFS, SCC/MEC decomposition, explicit-state model checking
  - **Not covered:** SAT solving, bounded model checking, term rewriting, theorem proving (ask Jan!), probabilistic model checking (again, ask Jan!)
- A GPU program is not hard to write. However, optimising one requires hardware knowledge
- Think of:
  - barriers
  - types of memory
  - warps
  - intra-warp instructions
- **Can GPUs help to accelerate your research?**