Accelerated Verification - part 1

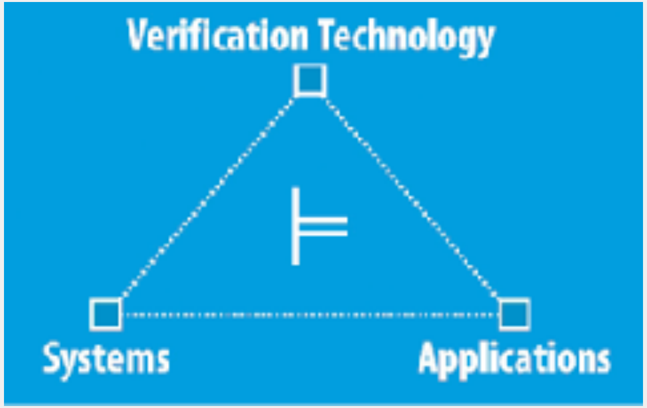
Anton Wijs

VTSA Summer school 2024 / 8 & 9 July

Anton Wijs

- Associate Professor on Parallel Software Development (Software Engineering & Technology)
- Topics: system verification, **model checking**, annotation checking, **parallel programming (**CPUs, **GPUs)**
- A.J.Wijs@tue.nl

It's great to be back!



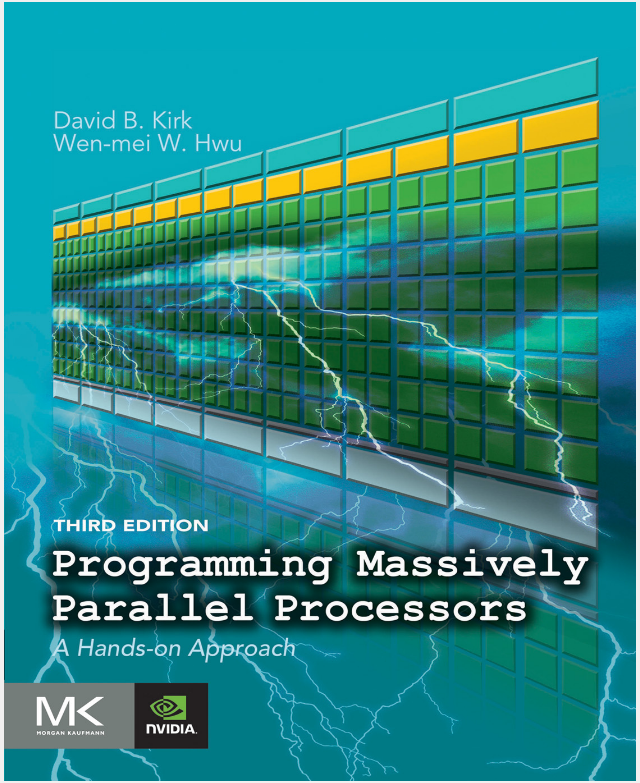
- **VTSA summer school 2010 in Luxembourg**
  - Attended as a post-doc researcher

Schedule Accelerated Verification

- **8 July 2024: 14:00 — 17:30**
  - Introduction to GPU computing (with applications to formal verification)
- **9 July 2024: 09:00 — 12:30**
  - Optimised GPU computing (with applications to formal verification)

Schedule 8 July 2024

- **14:00 – 14:30** Introduction to GPU Computing / High-level intro to CUDA Programming
- **14:30 – 15:00** 1st Hands-on Session
- **15:00 – 15:15** Solution to first Hands-on Session
- **15:15 – 15:45** PRAM model and a linear parallel bisimulation algorithm
- **15:45 – 16:15** CUDA Programming part 2, with 2nd Hands-on Session + solution
- **16:15 – 16:45** 3rd Hands-on Session + solution
- **16:45 – 17:15** A parallel algorithm for Strongly Connected Component detection in graphs
- **17:15 – 17:30** CUDA Program execution



We will cover approx. first five chapters

Introduction to GPU Computing

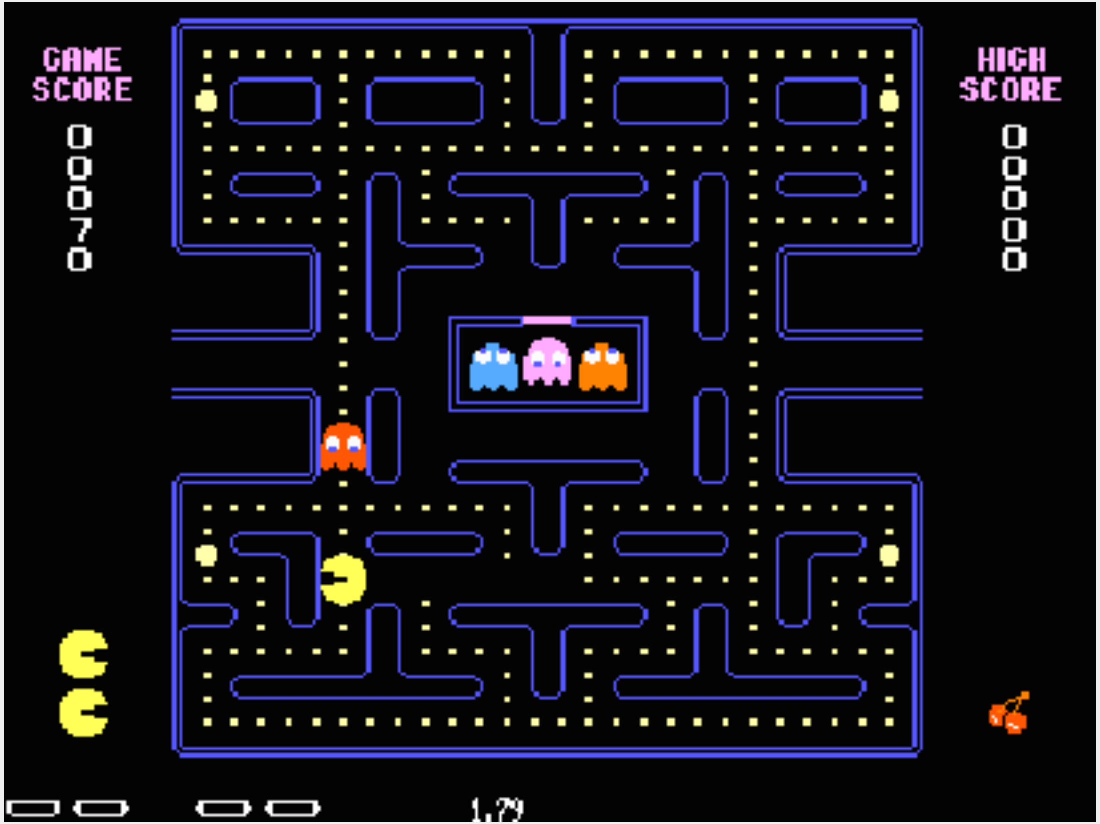
What is a GPU?

- Graphics Processing Unit –

  The computer chip on a graphics card

- *General Purpose* GPU (GPGPU)

Graphics in 1980



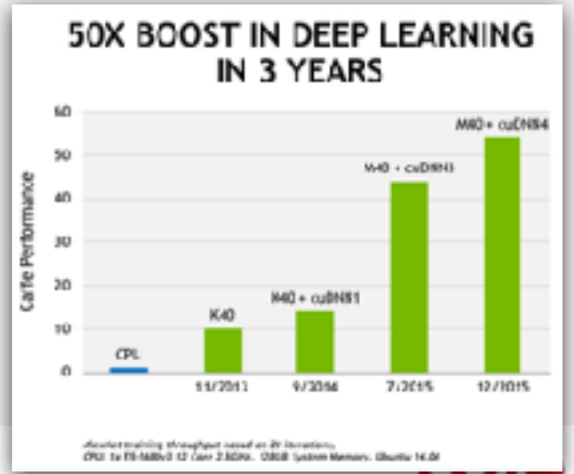
Graphics in 2000

Graphics now

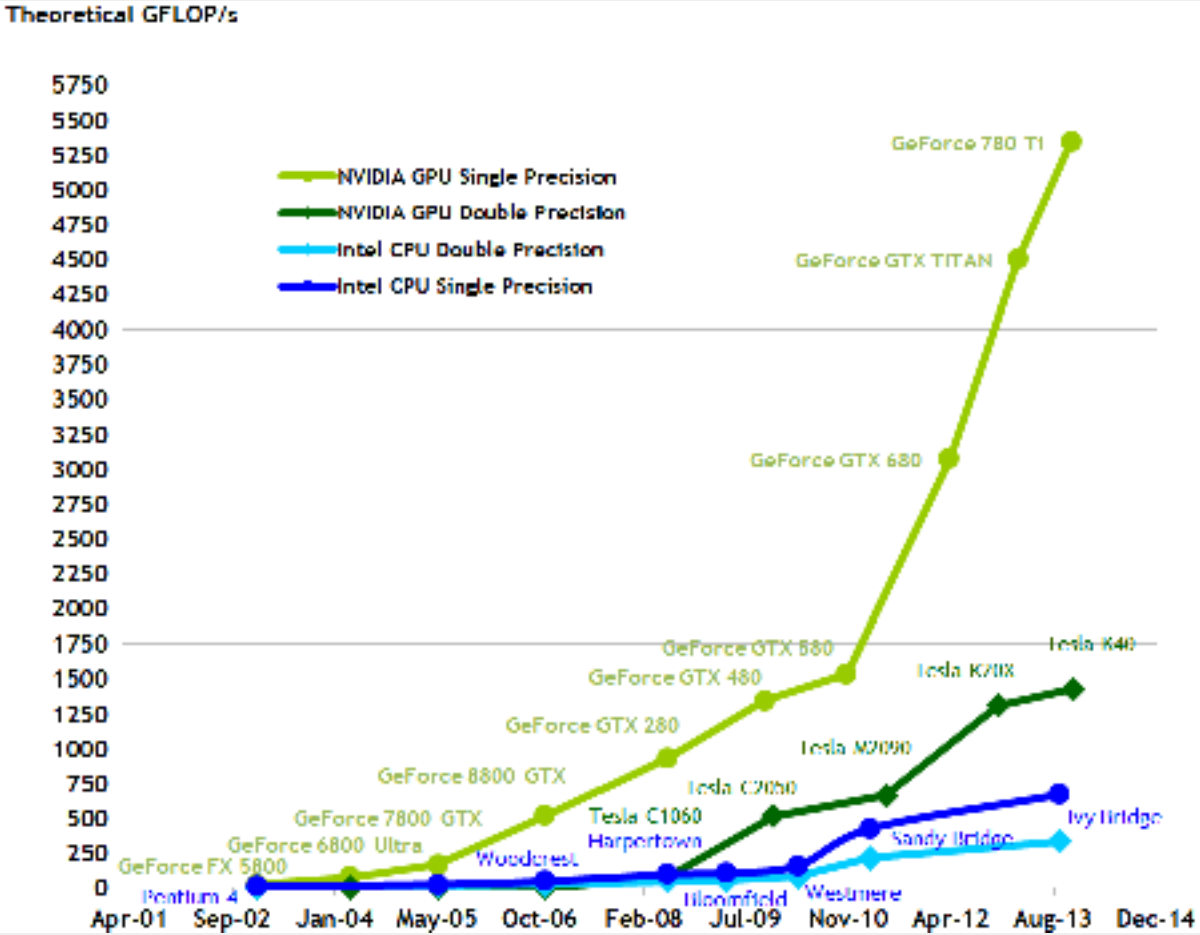
The impact of Graphics Processors (GPUs)

- Numerical simulation, media processing, medical imaging, machine learning, …
- *Communications of the ACM 59(9):14-16 (sep.'16)*
  - "GPUs are a gateway to the future of computing"
  - Example: **deep learning**
    - 2011-12: GPUs dramatically increase performance

Leiserson *et al.* There's plenty of room at the top: What will drive computer performance after Moore's law? *Science* 368(6495), 2020:
Major computational advances increasingly need to come from parallelism

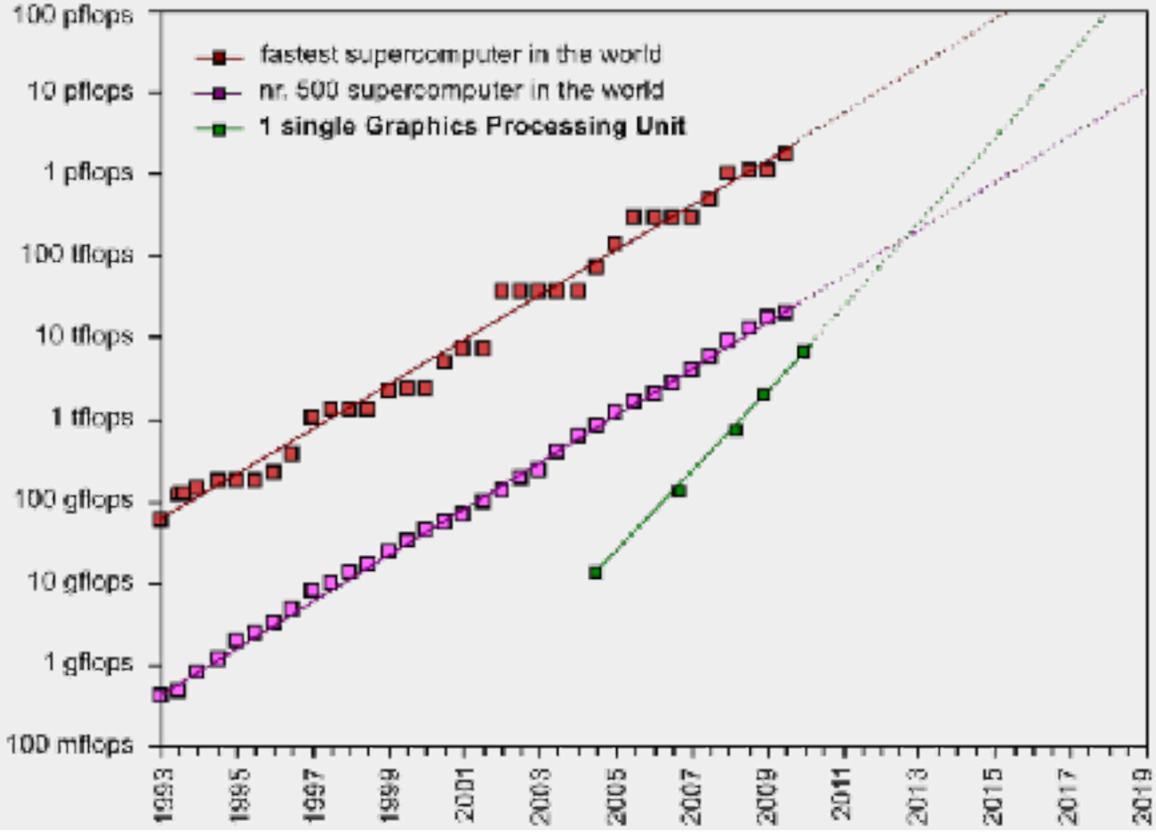


Compute performance



(According to Nvidia)

GPUs vs supercomputers ?

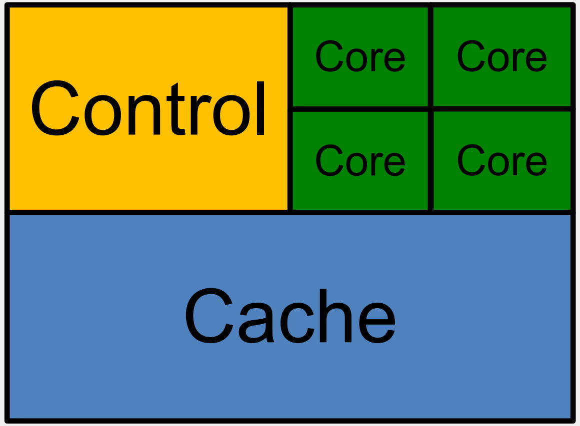


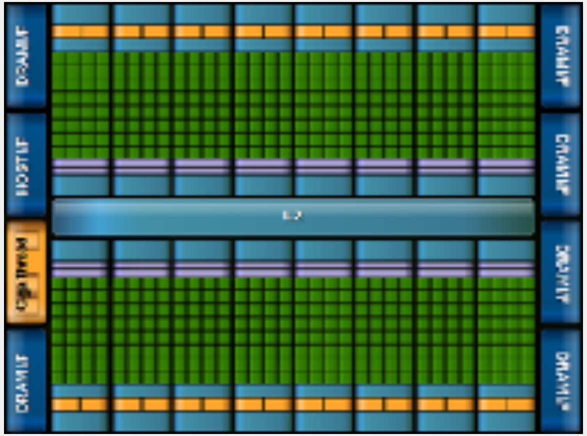
Oak Ridge’s Frontier (2022)

- Number 1 in top500 list (2024): **1.5 eflops** peak (15^{18} flops), 22.8 MW power
- 9,472 AMD Epyc 7713 “Trento” processors x 64 cores = **606,208 cores**
- 37,888 Instinct MI250X GPUs x 220 cores = **8,335,360 cores**

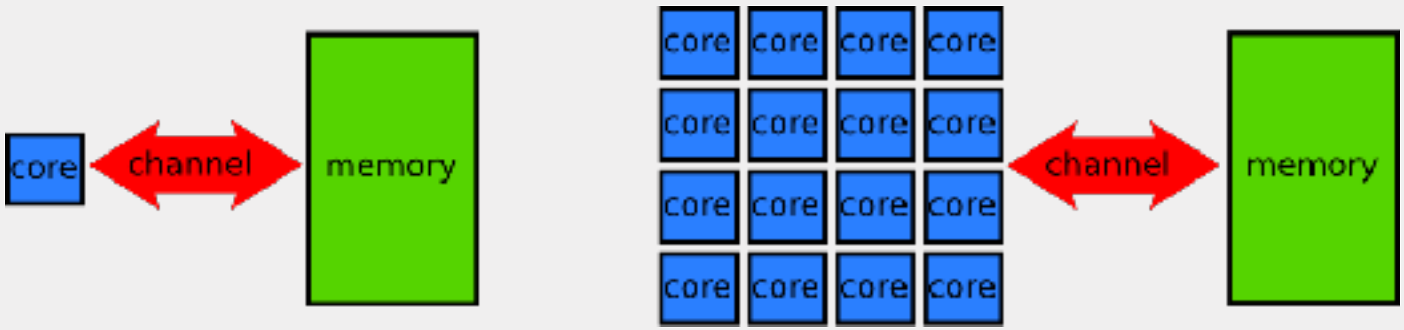
CPU vs GPU Hardware

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: minimize latency experienced by 1 thread
  - Big on-chip caches
  - Sophisticated control logic
- GPU: maximize throughput of all threads
  - Multithreading can hide latency, so no big caches
  - Control logic
    - Much simpler
    - Less: share control logic across many threads





It's all about the memory

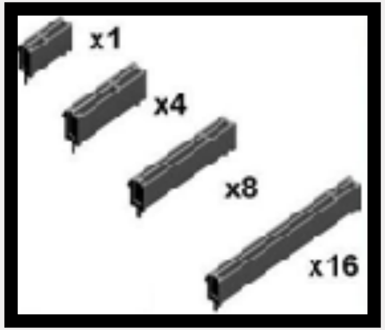


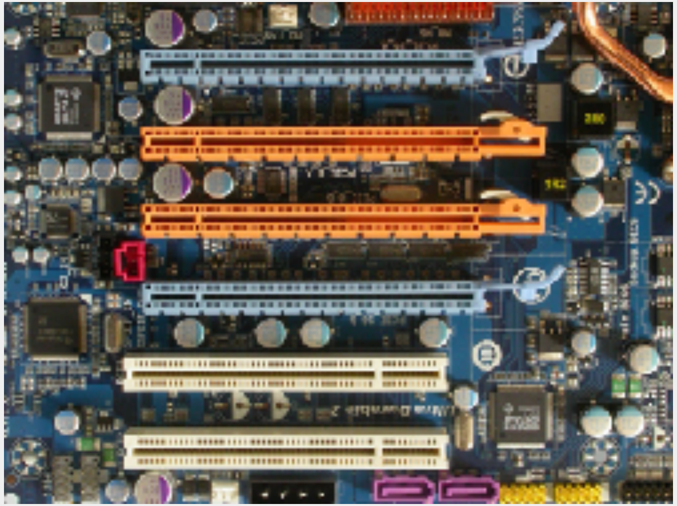
Many-core architectures

From Wikipedia: “A many-core processor is a multi-core processor in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient — largely because of issues with congestion in supplying instructions and data to the many processors.”

Integration into host system

- PCI-e 3.0 achieves about 16 GB/s
- Comparison: GPU device memory bandwidth is 320 GB/s for GTX1080





Why GPUs?

- Performance
  - Large scale parallelism
- Power Efficiency
  - Use transistors more efficiently
  - #1 in green 500 uses NVIDIA Grace Hopper Superchip 72C (June 2024)
- Price (GPUs)
  - Huge market
  - Mass production, economy of scale
  - Gamers (and AI engineers / users) pay for our HPC needs!

When to use GPU Computing?

- When:
  - Thousands or even millions of elements that can be processed in parallel

- Very efficient for algorithms that:
  - have high arithmetic intensity (lots of computations per element)
  - have regular data access patterns
  - do not have a lot of data dependencies between elements
  - do the same set of instructions for all elements

A high-level intro to CUDA Programming (Part 1)

CUDA Programming Model

Before we start:

- I’m going to explain the CUDA Programming model
- I’ll try to avoid talking about the hardware as much as possible
- For the moment, make no assumptions about the backend or how the program is executed by the hardware
- I will be using the term ‘thread‘ a lot, this stands for ‘thread of execution’ and should be seen as a parallel programming concept. Do not compare them to CPU threads.

CUDA Programming Model

- The CUDA programming model separates a program into a *host* (CPU) and a *device* (GPU) part.

- The host part: allocates memory and transfers data between host and device memory, and starts GPU functions

- The device part consists of functions that will execute on the GPU, which are called *kernels*
- Kernels are executed by huge amounts of threads at the same time
- The data-parallel workload is divided among these threads
- The CUDA programming model allows you to code for each thread individually

Data management

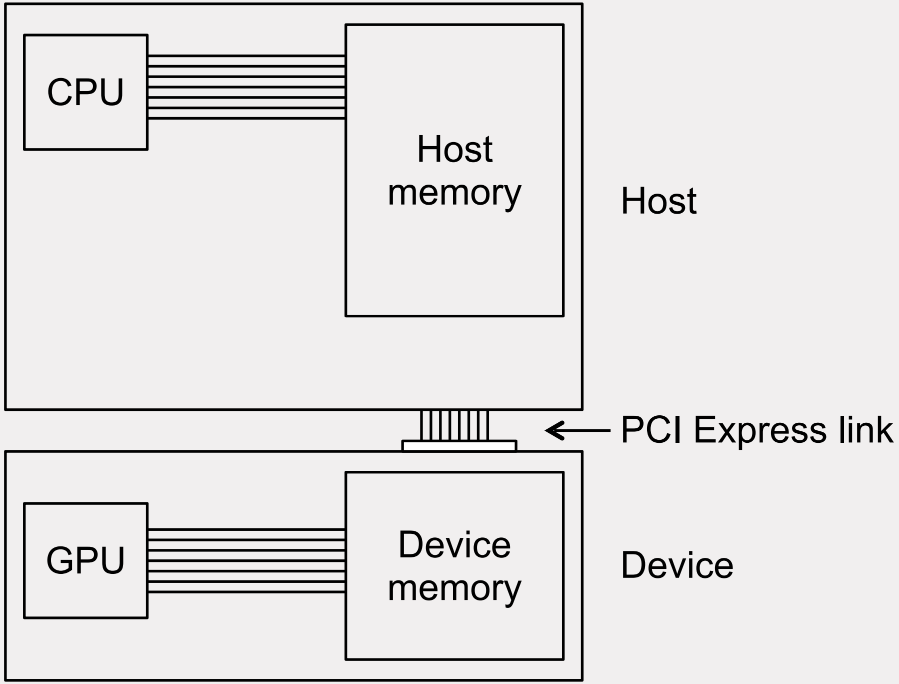
- The GPU is located on a separate device
- The host program manages the allocation and freeing of GPU memory

  C:
  - cudaMalloc()
  - cudaFree()

  Python:
  - mem_alloc()
- Host program also copies data between different physical memories

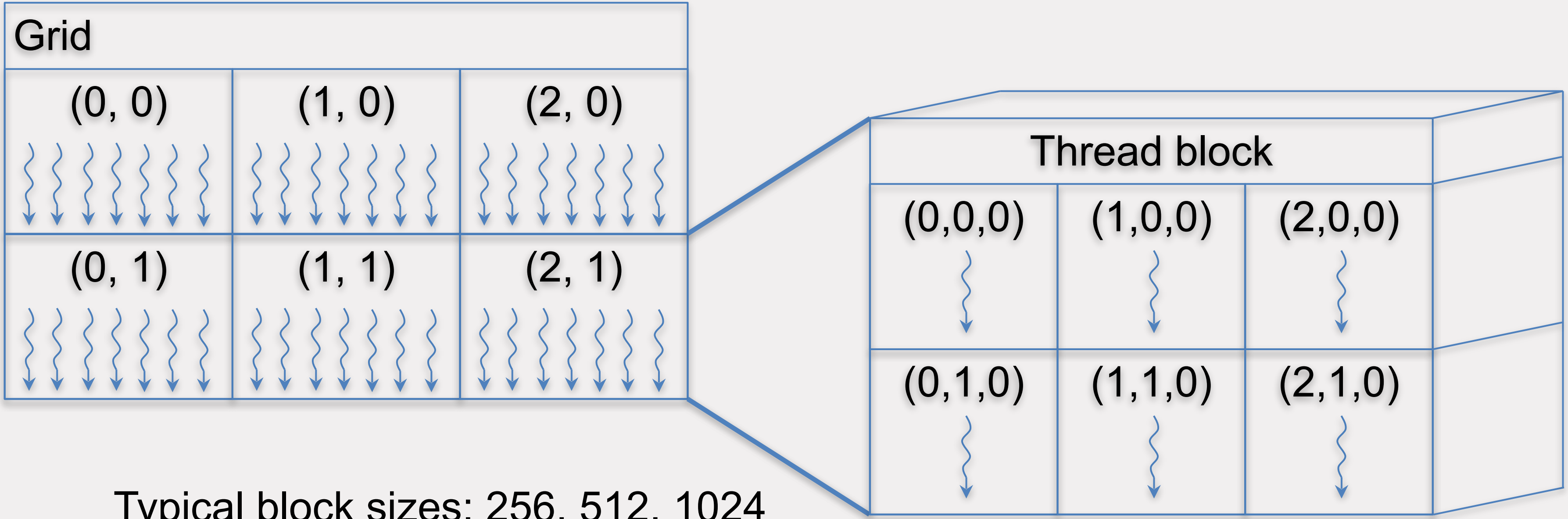
  C:
  - cudaMemcpy()

  Python:
  - memcpy_htod() or memcpy_dtoh()



Thread Hierarchy

- Kernels are executed in parallel by possibly millions of threads, so it makes sense to try to organize them in some manner



Typical block sizes: 256, 512, 1024

Threads

- In the CUDA programming model a thread is the most fine-grained entity that performs computations
- Threads direct themselves to different parts of memory using their built-in variables `threadIdx.x, y, z` (thread index *within* the thread block)
- Example:

```
for (i=0; i<N; i++) {
  c[i] = a[i] + b[i];
}
```

Create a single thread block of N threads:

```
i = threadIdx.x;
c[i] = a[i] + b[i];
```

- Effectively the loop is ‘unrolled’ and spread across N threads

TU/e

Threads

- In the CUDA programming model a thread is the most fine-grained entity that performs computations
- Threads direct themselves to different parts of memory using their built-in variables `threadIdx.x, y, z` (thread index *within* the thread block)
- Example:

```
for (i=0; i<N; i++) {
  c[i] = a[i] + b[i];
}
```

Create a single thread block of N threads:

```
i = threadIdx.x;
c[i] = a[i] + b[i];
```

Single Instruction Multiple Data (SIMD) principle

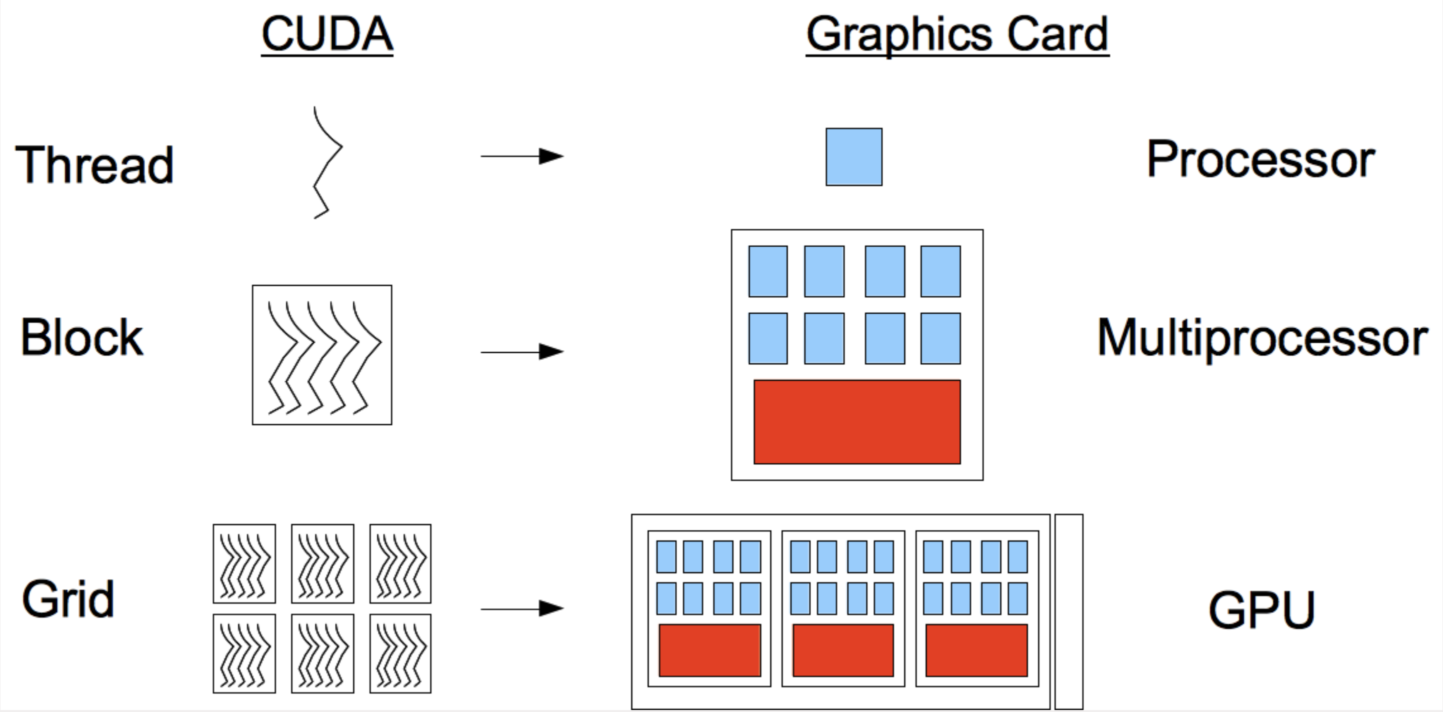
- Effectively the loop is ‘unrolled’ and spread across N threads

Thread blocks

- Threads are grouped in thread blocks, allowing you to work on problems larger than the maximum thread block size
- Thread blocks are also numbered, using the built-in variables `blockIdx.x`, `y` containing the index of each block within the grid.
- Total number of threads created is always a multiple of the thread block size, possibly not exactly equal to the problem size
- Other built-in variables are used to describe the thread block dimensions `blockDim.x`, `y`, `z` and grid dimensions `gridDim.x`, `y`

Mapping to hardware

Mapping to hardware



Starting a kernel

- The host program sets the number of threads and thread blocks when it launches the kernel

```
//create variables to hold grid and thread block dimensions
dim3 threads(x, y, z)
dim3 grid(x, y)

//launch the kernel
vector_add<<<grid, threads>>>(c, a, b);

//wait for the kernel to complete
cudaDeviceSynchronize();
```

CUDA function declarations

- `__global__` defines a kernel function
  - Each "`__`" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

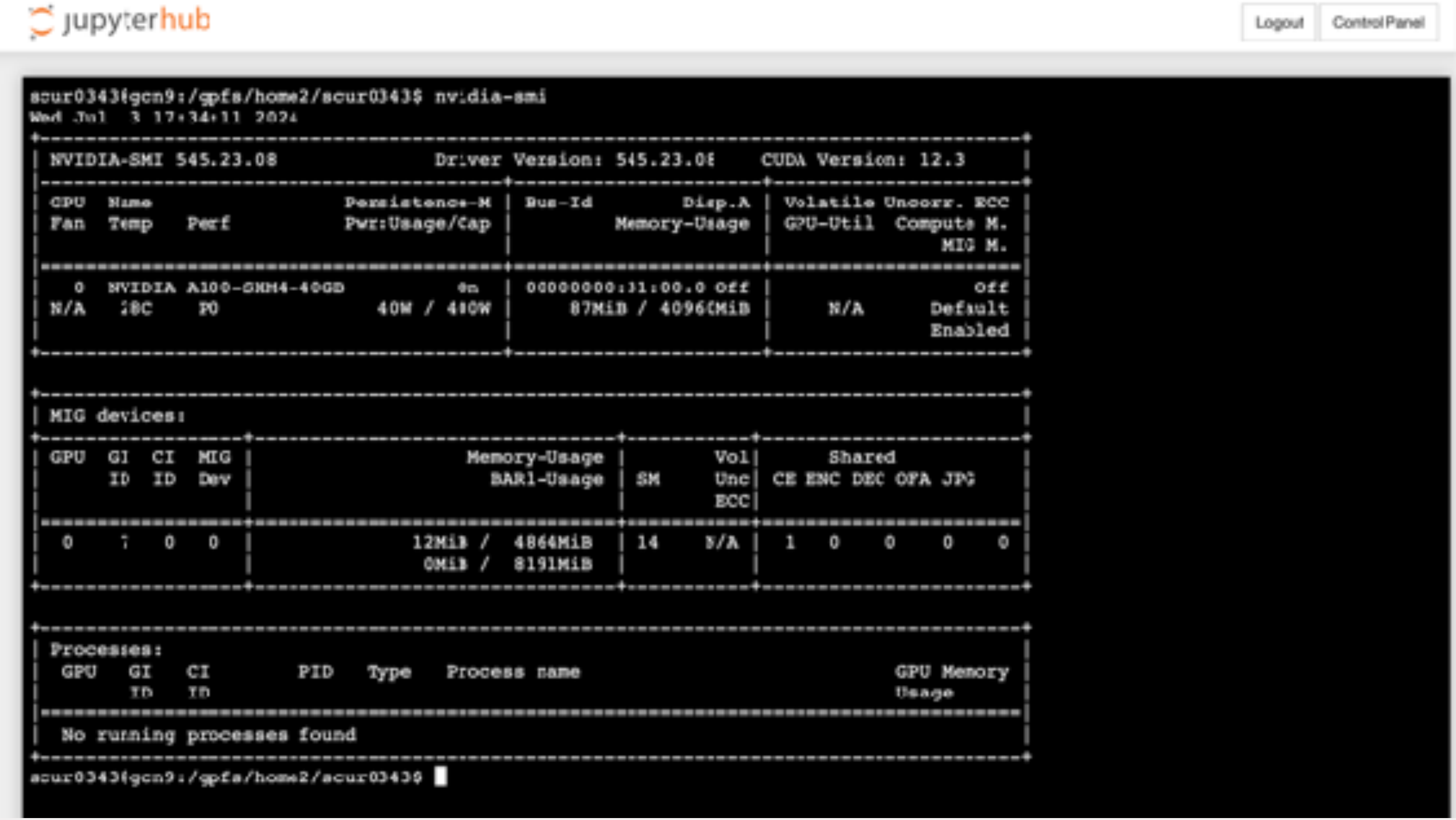
Setup hands-on sessions

- Go to https://jupyter.snellius.surf.nl/jhssrf012
- Log in with username / password given by SURF (SURFcua)
  - If password expired, request new password (Forgot)
- You will log into JupyterHub

Setup hands-on session

- You get to play with **1/8th of an NVIDIA A100 (via Multi-Instance GPU (MIG))**
  - 5 GB global memory
- How to check this?
  - On the top right of screen, click on **New**, select **Terminal**
    - This opens a terminal, and will be used to compile our GPU programs
  - In the terminal, run **nvidia-smi**

Setup hands-on session



TU/e

First hands-on session

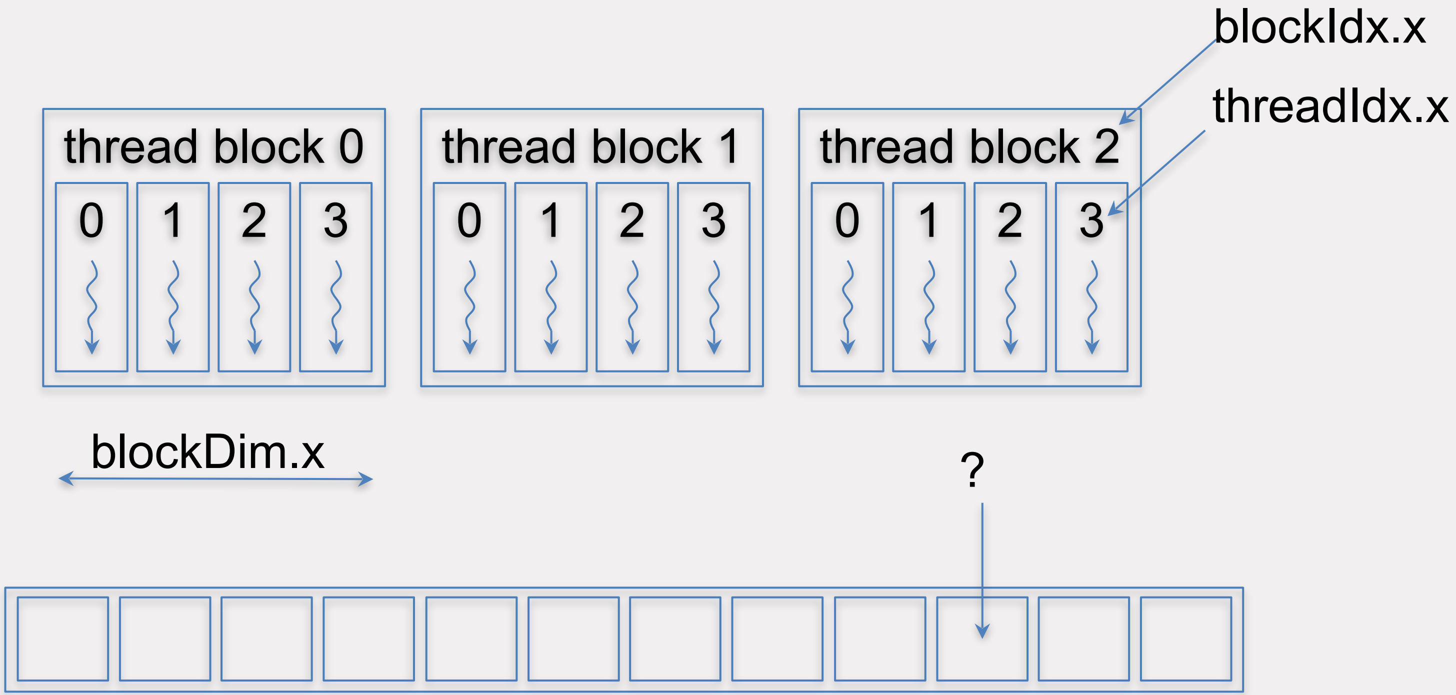
- Go back to the folders, open the folder **`1-vector-add`**

- **`timer.h`, `timer.cc`:** can be ignored, included to measure runtimes
- **`Makefile`:** can be ignored, used to compile our program
- **`vector_add.cu`:** the CUDA file we will work with, **open it and inspect**
  - Set **Language** to **C++** for syntax highlighting

First hands-on session

- Make sure you understand everything in the code, and complete the exercise!
- **Hints:**
  - Look at how the kernel is launched in the host program
  - `threadIdx.x` is the thread index within the thread block
  - `blockIdx.x` is the block index within the grid
  - `blockDim.x` is the dimension of the thread block (number of threads per block)

Hint


blockIdx.x
threadIdx.x
thread block 0
thread block 1
thread block 2
0 1 2 3
0 1 2 3
0 1 2 3
blockDim.x
?

Solution

- CPU implementation:

```
for (i=0; i<N; i++) {
  c[i] = a[i] + b[i];
}
```

- GPU implementation:

  Create a N threads using multiple thread blocks:

```
i = blockIdx.x * blockDim.x + threadIdx.x;
if (i<N) {
    c[i] = a[i] + b[i];
}
```

Single Instruction Multiple Data (SIMD) principle

But what if you have more data elements than threads?

- Use a **grid-stride loop**:

```
i = blockIdx.x * blockDim.x + threadIdx.x;
for (i=0; i<N; i += blockDim.x * gridDim.x) {
    c[i] = a[i] + b[i];
}
```

- Look at how the kernel is launched in the host program
- `threadIdx.x` is the thread index within the thread block
- `blockIdx.x` is the block index within the grid
- `blockDim.x` is the dimension of the thread block
- `gridDim.x` is the dimension of the grid (number of blocks)

Single Instruction Multiple Data (SIMD) principle

The PRAM model and a parallel linear bisimulation algorithm

Joint work with Lars van den Haak, Jan Martens, Jan-Friso Groote & Pieter Hijma (TACAS 2015, FACS 2021)

Computational model — CRCW PRAM

- The Parallel Random Access Machine (PRAM) is an extension of 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** (simplification of CUDA warp-based execution, addressed tomorrow)
- A PRAM program comes with a function $P : \mathbb{N} \to \mathbb{N}$ defining how many processes are started, based on the size of the input

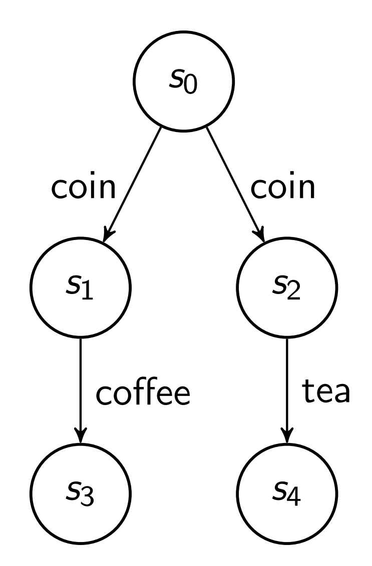
Computational model — memory contention

- Handling conflicts in read and writes of the common memory
  - Exclusive Read Exclusive Write (**EREW PRAM**)
  - Concurrent Read Exclusive Write (**CREW PRAM**)
  - Concurrent Read Concurrent Write (**CRCW PRAM**)
- In case of concurrent writes to the same memory cell further cases are distinguished:
  - **Priority CRCW:** The lowest indexed processor will write
  - **Arbitrary CRCW:** An arbitrary processor will complete the write
  - **Common CRCW:** Write will only succeed if all processors write the same value
- Our proposed algorithm works without changes on **Priority** and **Arbitrary** CRCW PRAM

Computational model — Computational Complexity

- The time complexity of a PRAM is given by the number of steps all the processors take
- **Optimal**
  - PRAM is called **optimal** w.r.t. a sequential algorithm if the total work done is equal. If T is the parallel run time and P is the number of processors, then it is optimal with an algorithm running in S steps if $P \cdot T \in \mathcal{O}(S)$ [Balcázar et al. 1992]
- Deciding bisimilarity is proven to be $\mathscr{P}$-complete. It is widely believed no PRAM algorithm running in polylogarithmic time exists for $\mathscr{P}$-complete problems

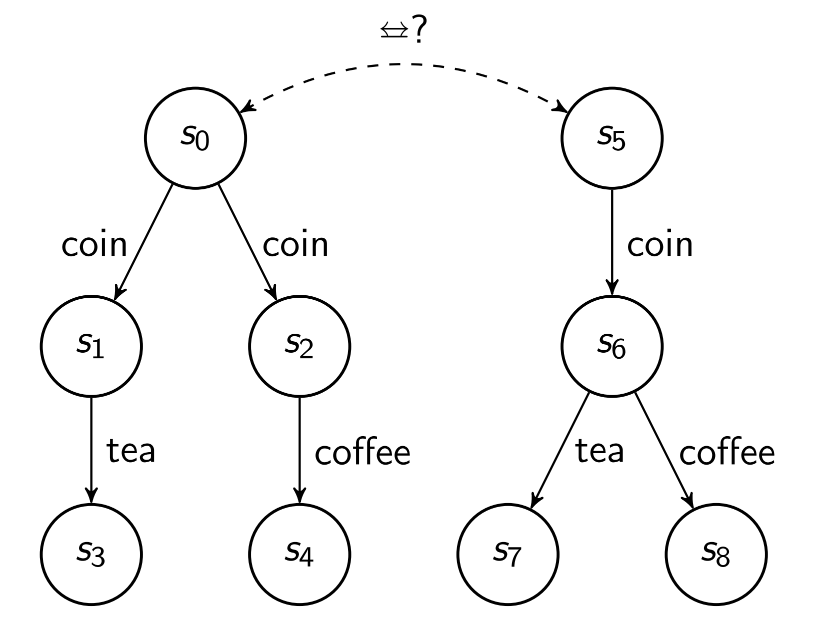
Labelled Transition System (LTS)



- LTS
    - (Finite) set of states S (*n* states)
    - (Finite) set of actions *Act*
    - Transition relation $T : S \times Act \times S$ (*m* transitions)
        - $s_0 \xrightarrow{coin} s_1$
- **Some terminology**
    - We say that a state s reaches a state t with action $a \in Act$ iff $s \xrightarrow{a} t$
    - A state s reaches a set of states $U \subseteq S$ with a iff there exists a state $t \in U$ such that s reaches t with a
    - A set of states V is *stable* under a set of states U iff for all actions a either all states in V reach U with a, or none of them do

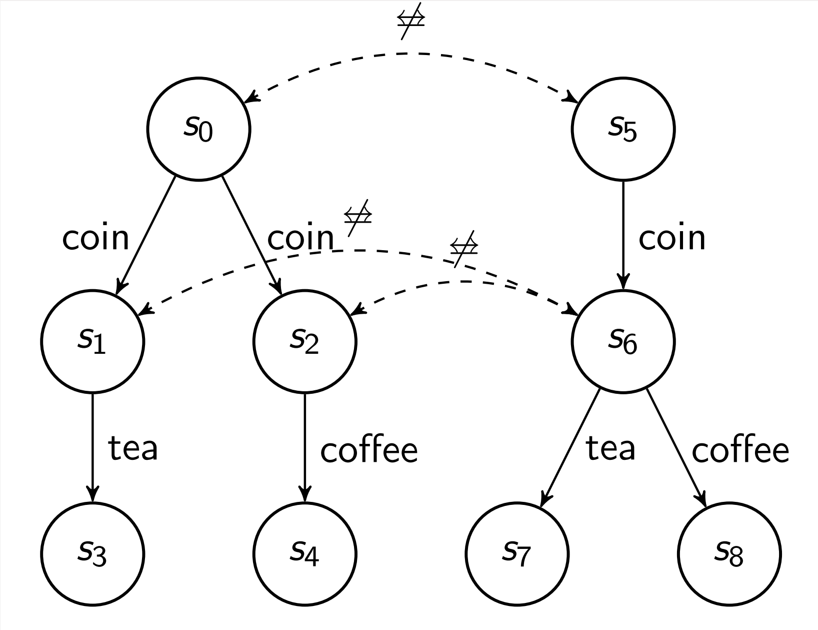
Strong bisimulation

- Two states s and t are **(strongly) bisimilar** iff there exists a relation $R : S \times S$ that is symmetric and $s\ R\ t$ implies that for all $s \xrightarrow{a} s'$ there exists a state t' such that $t \xrightarrow{a} t'$ and $s'\ R\ t'$
- We are interested in the **largest bisimulation relation**, which we refer to as $\underline{\leftrightarrow}$



Strong bisimulation

- Two states s and t are **(strongly) bisimilar** iff there exists a relation $R : S \times S$ that is symmetric and $s\ R\ t$ implies that for all $s \xrightarrow{a} s'$ there exists a state t' such that $t \xrightarrow{a} t'$ and $s'\ R\ t'$
- We are interested in the **largest bisimulation relation**, which we refer to as $\underline{\leftrightarrow}$



A comment about transition labels

- For LTSs with only a single transition label, the problem of bisimulation is also known as the **relational coarsest partition problem (RCPP)**

- **Fact:** RCPP is not significantly harder than finding the largest bisimulation for LTSs with multiple transition labels

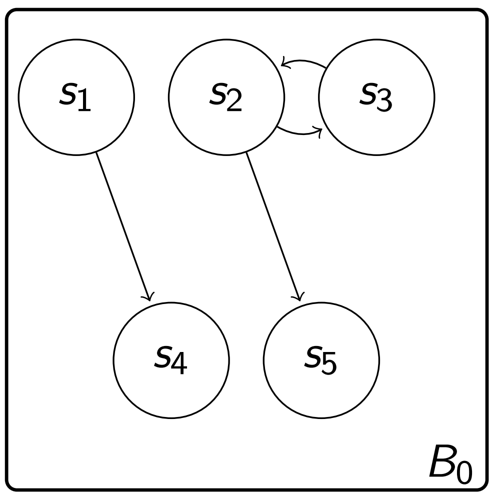
- For the sake of clarity, however, we will discuss algorithms in a setting **without** transition labels (equivalent to only one transition label)

Partition refinement

- A **partition** π of a set S is a disjoint cover of S, i.e, $\pi = \{B_0, B_1, \ldots, B_n\}$, and every pair of blocks $B_i, B_i' \in \pi$ is disjoint: $B_i \cap B_i' = \emptyset$ and all blocks together cover S: $\bigcup_{B_i \in \pi} B_i = S$
- A partition π' is a **refinement** of π if for all blocks $B \in \pi'$ there is a $B' \in \pi$ such that $B \subseteq B'$
- **Partition-based bisimilarity computation**
  - **Input:** An LTS $M = (S, Act, \rightarrow)$ and an initial partitioning π_0
  - **Output:** A partition π of S that defines $\underline{\leftrightarrow}$: $\forall s, t \in S \,.\, s \,\underline{\leftrightarrow}\, t \iff \exists B \in \pi \,.\, s, t \in B$

Partition-based bisimilarity computation

- **Idea**
  - Create a partition of states (sets of states, or *blocks*)
    - A partition π is *stable* under a set of states U iff each block $B \in \pi$ is stable under U
    - A partition π is stable iff it is stable under all its own blocks $B \in \pi$
  - Iteratively split blocks in smaller blocks (refine π) until bisimulation is achieved ($s \underline{\leftrightarrow} t$ iff $s, t \in B$)

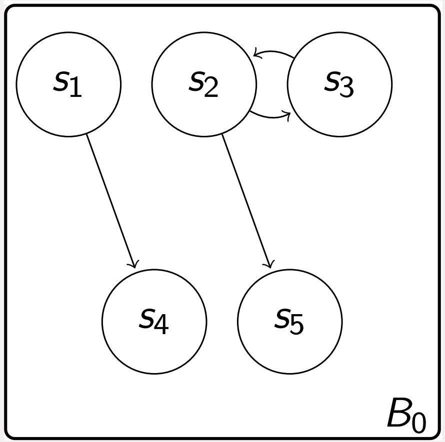


- **Fact:** Stability is inherited under refinement

The sequential Kanellakis-Smolka algorithm ($\mathcal{O}(mn)$)

$\pi := \pi_0$;
Unstable $:= \pi_0$;
while *Unstable* $\neq \emptyset$ **do**
  foreach $B \in$ *Unstable* **do**
    Delete$(B,$ *Unstable*$)$;
    $C := \{s | s \to t \text{ and } t \in B\}$;
    foreach $B' \in \pi$ *for which* $\emptyset \neq B' \cap C \neq B'$ **do**
      `// Split` B' `into` $B' \cap C$ `and` $B' \setminus C$
      Delete(B, π);
      $\pi := \pi \cup \{B' \cap C, B' \setminus C\}$;
      Unstable $:=$ *Unstable* $\cup \{B' \cap C, B' \setminus C\}$;
    end
  end
end

Algorithm 1: Sequential algorithm based on Kanellakis & Smolka. *Unstable* is set of *not necessarily* stable blocks.

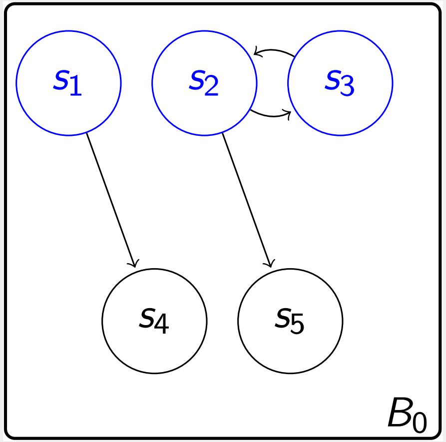


Select block B_0

The sequential Kanellakis-Smolka algorithm ($\mathcal{O}(mn)$)

$\pi := \pi_0$;
Unstable $:= \pi_0$;
while *Unstable* $\neq \emptyset$ **do**
    foreach $B \in$ *Unstable* **do**
        Delete$(B,$ *Unstable*$)$;
        $C := \{s \mid s \rightarrow t \text{ and } t \in B\}$;
        foreach $B' \in \pi$ *for which* $\emptyset \neq B' \cap C \neq B'$ **do**
            // Split B' into $B' \cap C$ and $B' \setminus C$
            Delete(B, π);
            $\pi := \pi \cup \{B' \cap C, B' \setminus C\}$;
            Unstable $:=$ *Unstable* $\cup \{B' \cap C, B' \setminus C\}$;
        end
    end
end

Algorithm 1: Sequential algorithm based on Kanellakis & Smolka. *Unstable* is set of *not necessarily* stable blocks.

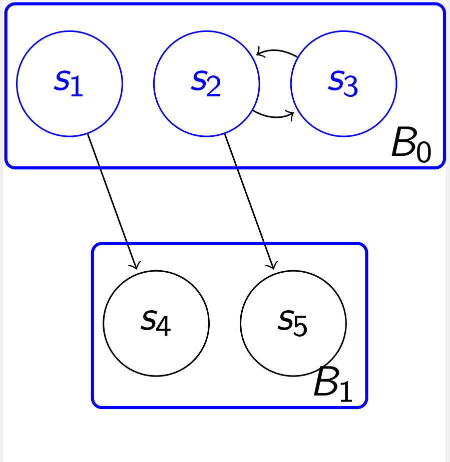


Calculate the reverse image of B_0: C

The sequential Kanellakis-Smolka algorithm ($\mathcal{O}(mn)$)

$\pi := \pi_0$;
$\mathit{Unstable} := \pi_0$;
while $\mathit{Unstable} \neq \emptyset$ **do**
    foreach $B \in \mathit{Unstable}$ **do**
        $\mathit{Delete}(B, \mathit{Unstable})$;
        $C := \{s | s \rightarrow t \text{ and } t \in B\}$;
        foreach $B' \in \pi$ *for which* $\emptyset \neq B' \cap C \neq B'$ **do**
            `// Split` B' `into` $B' \cap C$ `and` $B' \setminus C$
            $\mathit{Delete}(B, \pi)$;
            $\pi := \pi \cup \{B' \cap C, B' \setminus C\}$;
            $\mathit{Unstable} := \mathit{Unstable} \cup \{B' \cap C, B' \setminus C\}$;
        end
    end
end

Algorithm 1: Sequential algorithm based on Kanellakis & Smolka. *Unstable* is set of *not necessarily* stable blocks.



Split all blocks based on C

Parallel Algorithm (without action labels)

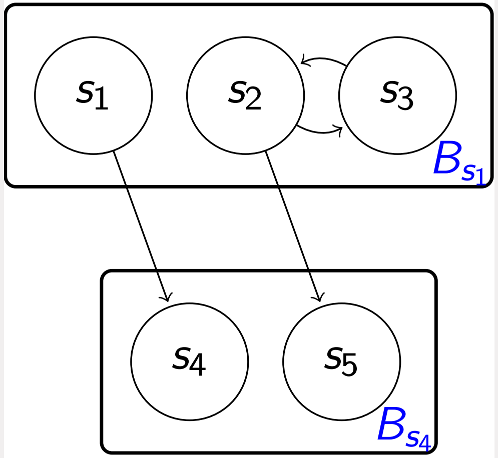
- We use an Arbitrary Concurrent Read Concurrent Write PRAM
  - Each processor runs program in lock-step, has shared memory
  - Write data races lead to random processor completing write
- **Idea:**
  - Perform steps of the sequential algorithm in $\mathcal{O}(1)$ time on $\max(n, m)$ processors
    - **This is not so straightforward**
  - We perform at most $\mathcal{O}(n)$ iterations
    - Total complexity of $\mathcal{O}(n)$ time

Data structures in common memory

- $N : \mathbb{N}$ the number of states of the input LTS
- $M : \mathbb{N}$ the number of transitions of the input LTS
- **The input, a list of transitions:** for every transition numbered $i \in \{0,\ldots,M\}$:
  - A source $s \in S$ and target $t \in S$ indicating it is the transition $s \to t$.
    - $s_i := s$
    - $t_i := t$
- *current_splitter :* $L_B \cup \{ \perp \}$ the current block that is used for splitting
- For each state $s \in S$:
  - mark$_s$: $\mathbb{B}$, the mark whether s is able to reach the current block
  - block$_s$: L_B, the block s is a member of. Initially, block$_s := 0$
- For each block label $b \in L_B$:
  - next_number$_b$: L_B, the leader of the new block when a split is performed
  - stable$_b$: $\mathbb{B}$, indicating whether the block is stable. Initially, stable$_0$ = **false** and for all blocks $b \in L_B$, $b \neq 0$, stable$_b$ = **true**

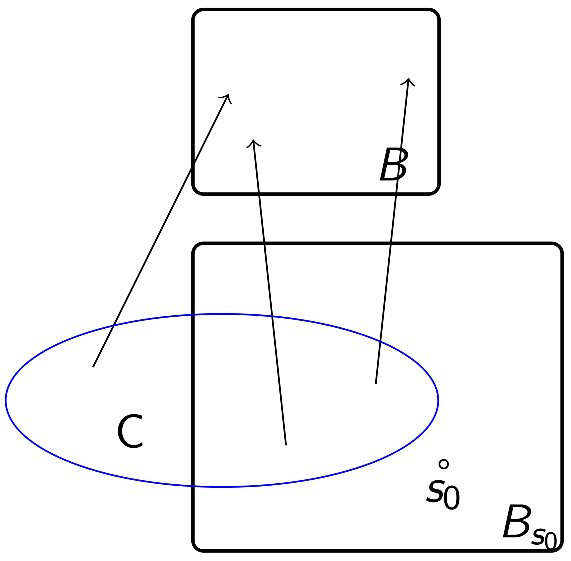
Parallel Algorithm (without action labels)

- **Block labels & leaders**
  - Each state s has a block label (block$_s$)
  - Same block label ≡ Same block
  - Labels are states themselves. (block$_s = s' \in S$)
  - The state that is the label of a block is called the *block leader*
- **Mark states**
  - Each state has a mark (mark$_s$) indicating if it reaches the splitter
- **Splitting blocks**
- Block leader remains in own block
- For each block, a *new leader* is elected from states that split off



New leader election

- *current_splitter* = B
- $C := \{s \mid s \to t \wedge t \in B\}$
- The block B_{s_0} with label s_0 will split in two blocks according to C
- The new block $C \cap B_{s_0}$ will elect a new leader
- A concurrent write in a variable next_number$_{s_0}$ will choose a state as new leader



Parallel Algorithm (without action labels)

1. Reset variables and choose splitter

```
if i ≤ N then
    current_splitter := ⊥;
    mark_i := false;
    if unstable_i then
        current_splitter := i;
    end
end
```

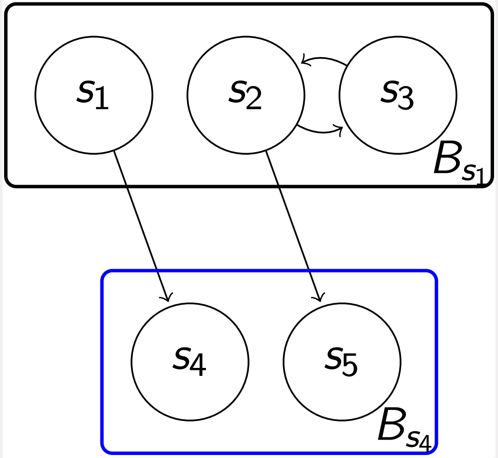
2. Mark states that reach the splitter

```
if i ≤ M and block_target_i = current_splitter then
    mark_source_i := true;
end
```

3. Perform splits based on marks & set unstable

```
if i ≤ N and current_splitter ≠ ⊥ then
    unstable_current_splitter := false;
    if mark_i ≠ mark_block_i then
        new_leader_block_i := i;
        unstable_block_i := true;
        block_i := new_leader_block_i;
        unstable_block_i := true;
    end
end
```

Repeat until fix-point is reached



Step 1: select

current_splitter := B_{s_4}

Parallel Algorithm (without action labels)

1. Reset variables and choose splitter

if $i \leq N$ **then**
  $current\_splitter := \perp$;
  $\mathrm{mark}_i := \mathrm{false}$;
  if $unstable_i$ **then**
    $current\_splitter := i$;
  end
end

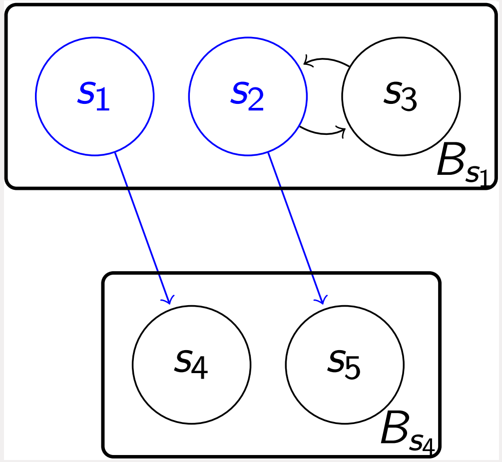
2. **Mark states that reach the splitter**

if $i \leq M$ *and* $block_{target_i} = current\_splitter$ **then**
  $\mathrm{mark}_{source_i} := \mathrm{true}$;
end

3. Perform splits based on marks & set unstable

if $i \leq N$ *and* $current\_splitter \neq \perp$ **then**
  $\mathrm{unstable}_{current\_splitter} := \mathrm{false}$;
  if $mark_i \neq mark_{block_i}$ **then**
    $\mathrm{new\_leader}_{block_i} := \mathrm{i}$;
    $\mathrm{unstable}_{block_i} := \mathrm{true}$;
    $\mathrm{block}_i := \mathrm{new\_leader}_{block_i}$;
    $\mathrm{unstable}_{block_i} := \mathrm{true}$;
  end
end

Repeat until fix-point is reached



Step 2: mark states s_1, s_2

Parallel Algorithm (without action labels)

1. Reset variables and choose splitter

```
if i ≤ N then
    current_splitter := ⊥;
    mark_i := false;
    if unstable_i then
        current_splitter := i;
    end
end
```

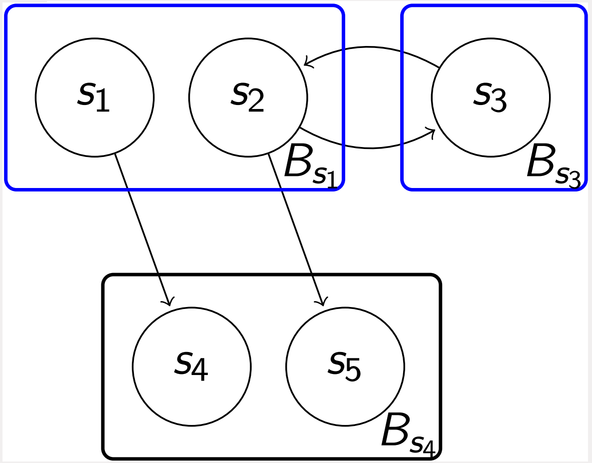
2. Mark states that reach the splitter

```
if i ≤ M and block_target_i = current_splitter then
    mark_source_i := true;
end
```

3. Perform splits based on marks & set unstable

```
if i ≤ N and current_splitter ≠ ⊥ then
    unstable_current_splitter := false;
    if mark_i ≠ mark_block_i then
        new_leader_block_i := i;
        unstable_block_i := true;
        block_i := new_leader_block_i;
        unstable_block_i := true;
    end
end
```

Repeat until fix-point is reached



Step 3: split B into B_1, B_2

Putting labels back in LTSs

- Translating LTSs to a transition system without labels results in a LTS with $\mathcal{O}(m)$ states. In the worse case, $m = n^2$, also the steps our algorithm has to take can grow quadratically
  We can do better!

- **Algorithm with labels**

1. Preprocess the states, such that states are grouped on outgoing actions
2. For every state s, keep track of a mark_s boolean for every outgoing action
3. Let the transitions compare these marks with the leading state

- Has $\mathcal{O}(n + |\text{Act}|)$ time complexity

Experimental results

Benchmark name	$\|Act\|$	$\|Blocks\|$	$\#It$	T_{pre}	T_{alg}	$\#It/n$	$\#It/Blocks$	$T_{Par\text{-}BCRP}/n$	$T_{alg}/\#It$	$T_{Par\text{-}BCRP}$	T_{LR}	T_{Wss}	T_{Wms}
Vasy_0_1	2	9	16	0.50	0.37	0.06	1.78	0.003	0.023	0.87	2.29	0.49	0.45
Cwi_1_2	26	1,132	2,786	0.63	56.5	1.43	2.46	0.029	0.020	57.1	17	18.8	21.8
Vasy_1_4	6	28	45	0.56	1.01	0.04	1.61	0.001	0.022	1.58	4.78	1.68	0.62
Cwi_3_14	2	62	122	0.63	2.68	0.03	1.97	0.001	0.022	3.30	60	3.80	3.72
Vasy_5_9	31	145	193	0.84	4.22	0.04	1.33	0.001	0.022	5.06	134	35.3	3.45
Vasy_8_24	11	416	664	0.70	13.9	0.07	1.59	0.002	0.021	15	277	31.5	3.03
Vasy_8_38	81	219	319	1.12	6.64	0.04	1.46	0.001	0.021	7.76	127	35.1	5.94
Vasy_10_56	12	2,112	3,970	0.73	82.0	0.37	1.88	0.008	0.021	82.7	860	40.9	4.6(0.2)
Vasy_18_73	17	4,087	6,882	1.01	142	0.37	1.68	0.008	0.021	143	1,354	211	21.7
Vasy_25_25	25,216	25,217	25,218	159	519	1.00	1.00	0.027	0.021	678	21,960	t.o.	416
Vasy_40_60	3	40,006	87,823	0.87	1,810	2.20	2.20	0.045	0.021	1,811	17,710	1,290	1,230
Vasy_52_318	17	8,142	15,985	2.52	338	0.31	1.96	0.007	0.021	340	11,855	368	152(20)
Vasy_65_2621	72	65,536	98,730	12.2	10,050	1.51	1.51	0.154	0.102	10,060	t.o.	27,000	1,230
Vasy_66_1302	81	66,929	91,120	6.70	5,745	1.36	1.36	0.086	0.063	5,752	480,600	20,450	240(20)
Vasy_69_520	135	69,754	113,246	4.13	3,780	1.62	1.62	0.054	0.033	3,780	94,800	16,090	35.4
Vasy_83_325	211	83,436	148,012	4.41	3,093	1.77	1.77	0.037	0.021	3,097	57,190	21,500	5,880
Vasy_116_368	21	116,456	210,537	2.50	5,900	1.81	1.81	0.051	0.028	5,900	80,900	6,360	2,930
Cwi_142_925	7	3,410	5,118	4.85	238	0.04	1.50	0.002	0.047	243	3,363	220(30)	140(20)
Vasy_157_297	235	4,289	9,682	4.58	201	0.06	2.26	0.001	0.021	206	1,058	1,240	579
Vasy_164_1619	37	1,136	1,630	8.34	125	0.01	1.43	0.001	0.077	134	8,173	470(30)	46.8
Vasy_166_651	211	83,436	145,029	6.13	5,710	0.87	1.74	0.034	0.039	5,720	80,210	29,660	9,560
Cwi_214_684	5	77,292	149,198	3.58	6,948	0.70	1.93	0.032	0.047	6,952	19,250	440(30)	450(50)
Cwi_371_641	61	33,994	85,858	4.72	4,050	0.23	2.53	0.011	0.047	4,050	26,940	6,970	1,548
Vasy_386_1171	73	113	199	7.38	14.0	0.00	1.76	0.000	0.070	21	334	30.6	34.8
Cwi_566_3984	11	15,518	23,774	16.0	3,707	0.04	1.53	0.007	0.156	3,723	98,200	6,700	2,200(200)
Vasy_574_13561	141	3,577	5,860	71.5	3,770	0.01	1.64	0.007	0.643	3,841	144,810	11,700	1,853
Vasy_720_390	49	3,292	3,782	3.97	143	0.01	1.15	0.0002	0.038	147	2,454	1,633	183
Vasy_1112_5290	23	265	365	24.0	99.3	0.0003	1.38	0.0001	0.272	123	4,570	293	36.8
Cwi_2165_8723	26	31,906	66,132	37.0	23,660	0.03	2.07	0.011	0.358	23,700	140,170	9,700	1,965
Cwi_2416_17605	15	95,610	152,099	64.1	96,400	0.06	1.59	0.040	0.634	96,500	257,200	16,300(1100)	15,300
Vasy_6020_19353	511	7,168	12,262	221	11,690	0.002	1.71	0.002	0.954	11,910	107,900	34,000(2000)	19,230
Vasy_6120_11031	125	5,199	10,014	74.0	6,763	0.002	1.93	0.001	0.675	6,837	55,750	7,010	1,280
Vasy_8082_42933	211	408	660	281	1,149	0.0001	1.62	0.0002	1.739	1,429	17,272	5,530	2,030

Branching bisimulation

- Idea can be extended to address **branching bisimulation**

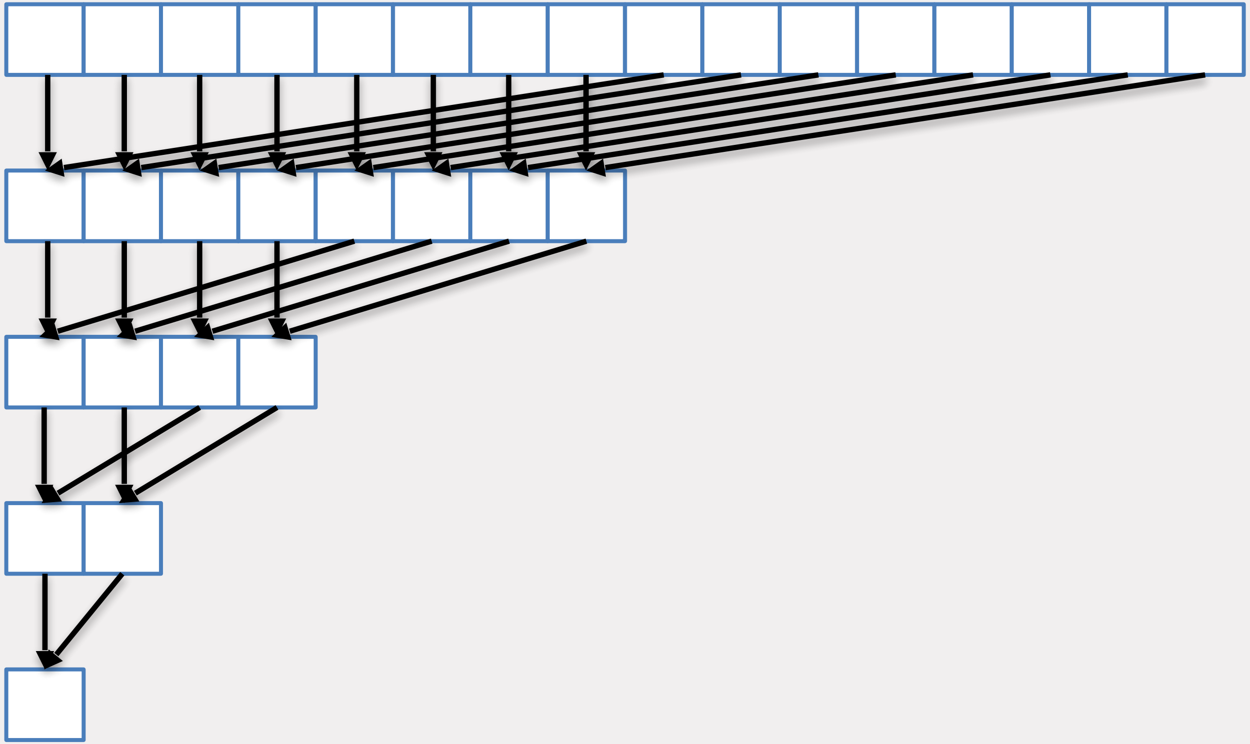
- **Branching bisimulation**
  - Given an LTS $M = (S, \mathrm{Act}, \rightarrow)$, a relation $R : S \times S$ is a branching bisimulation relation iff it is symmetric and for all $s, t \in S$ with $s\ R\ t$ and for all $a \in \mathrm{Act} \cup \{\tau\}$ with $s \xrightarrow{a} s'$, we have either
    - $a = \tau$ and $s'\ R\ t$, or
    - there is a sequence $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t'$ of zero or more τ-transitions such that $s\ R\ t'$, $t' \xrightarrow{a} t''$ and $s'\ R\ t''$
- **However, computation requires transitive closure of τ-transitions**
  - Calculate at pre-processing in $\mathcal{O}(n)$ time using $\mathcal{O}(n^2)$ processors
  - **Fundamental problem: the transitive closure bottleneck** [Kao & Klein 1993]

CUDA Programming
Part 2

Second hands-on session

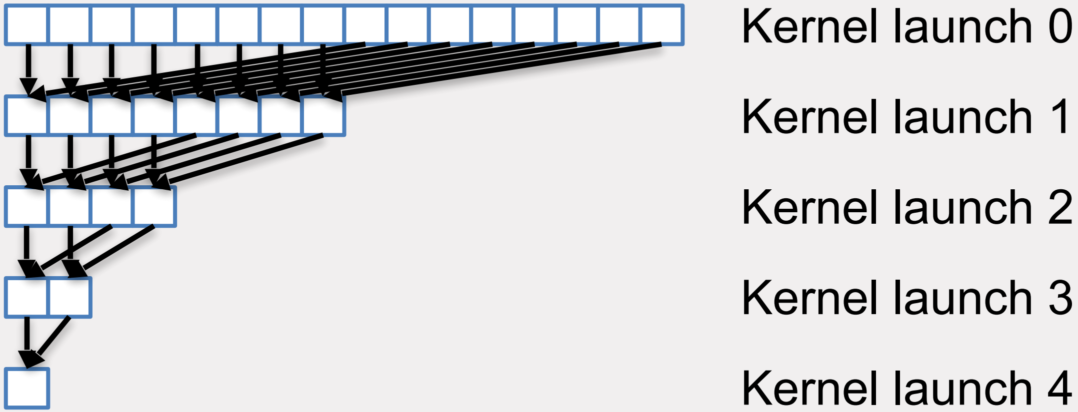
- Go to folder `2-reduction`, look at the source file `reduction.cu`
- Make sure you understand everything in the code
- **Task:**
  - Implement the kernel to perform a single iteration of parallel reduction
- **Hints:**
  - It is assumed that enough threads are launched such that each thread only needs to compute the sum of two elements in the input array
  - In each iteration, an array of size n is reduced into an array of size n/2
  - Each thread stores its result at a designated position in the output array

Hint – Parallel Summation



Global synchronisation

- CUDA has no mechanism to indicate global synchronisation of all threads across the grid
- Instead, enforce synchronisation points by breaking down computation into multiple kernel launches

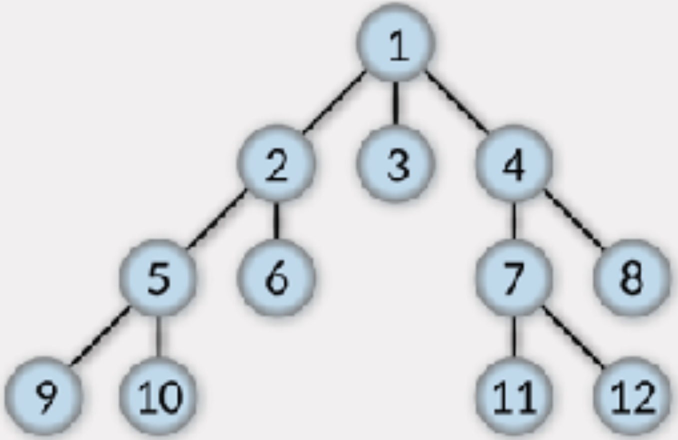


Barrier synchronisation

- Two forms:
  - **Global synchronisation:** achieved between kernel launches
  - **Intra-block synchronisation:** Contrary to global synchronisation, CUDA does provide a mechanism to synchronise all threads in the same block
    - `__syncthreads()`
    - All threads in the same block must reach the `__syncthreads()` before any of them can move on
    - Best used to split up computation of each block in several phases
    - Tightly linked to use of (block-local) **shared memory**, which we will address tomorrow afternoon

Third hands-on session

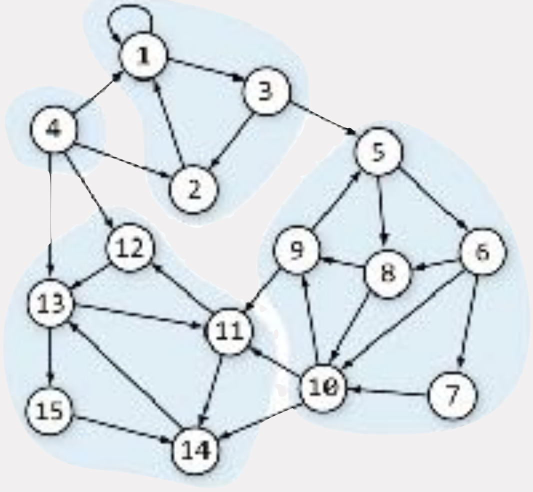
- Go to folder **`3-bfs`**, look at the source file **`bfs.cu`**
- Make sure you understand everything in the code
- **Task:**
  - Implement the kernel to perform a single iteration of parallel breadth-first search (BFS)
- **Hints:**
  - In BFS, a set OPEN is maintained, consisting of states that require exploration, i.e., of which the outgoing transitions need to be followed. Once a state is explored, it is moved to a set CLOSED



SCC decomposition of graphs
MEC decomposition of MDP graphs

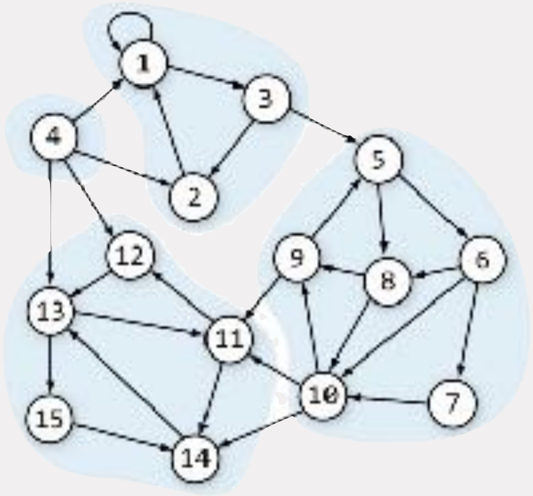
Joint work with Dragan Bošnački and Joost-Pieter Katoen
(CAV 2014)

Strongly Connected Components

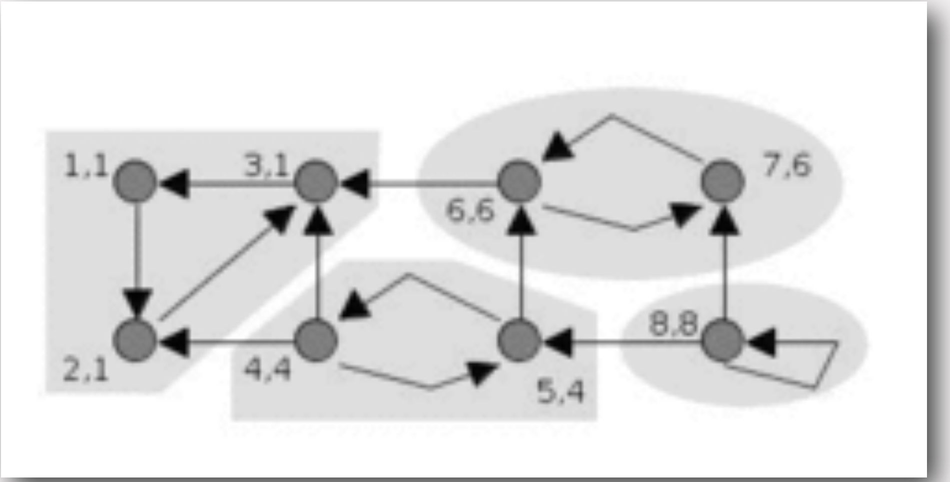


- **Model checking:** checking liveness (something good eventually happens)
- **Metabolic networks:** metabolites in SCC can be converted to each other

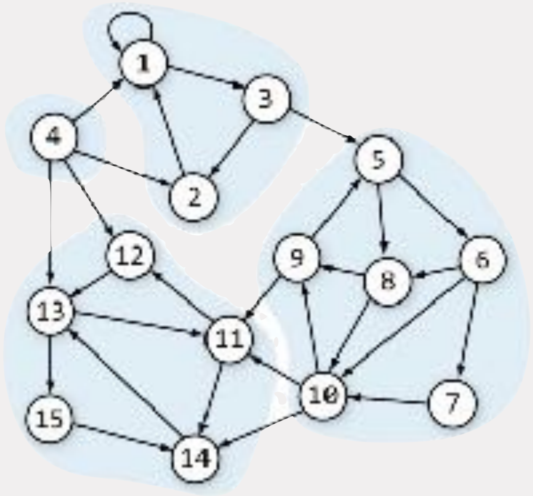
Strongly Connected Components



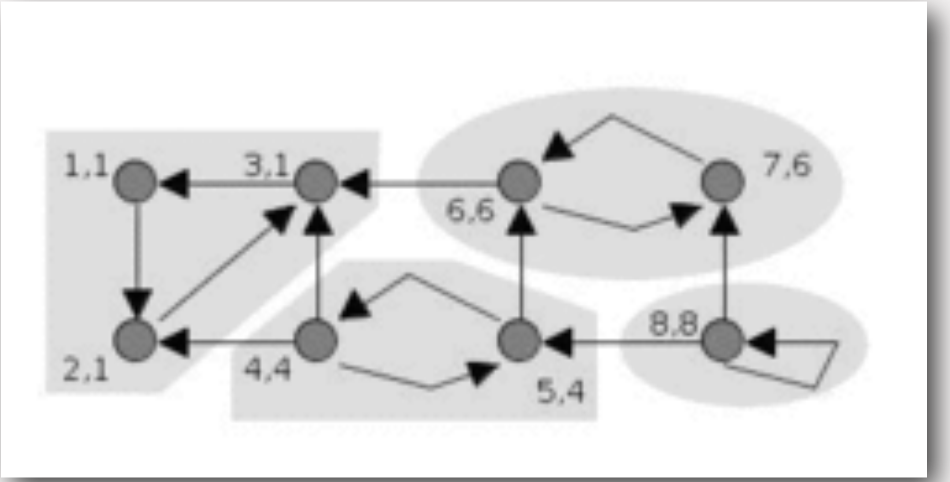
- **Sequential:** linear-time [Tarjan,'72], [Dijkstra & Feijen,'88]
- DFS based; hard to parallelise; next to impossible for many-core



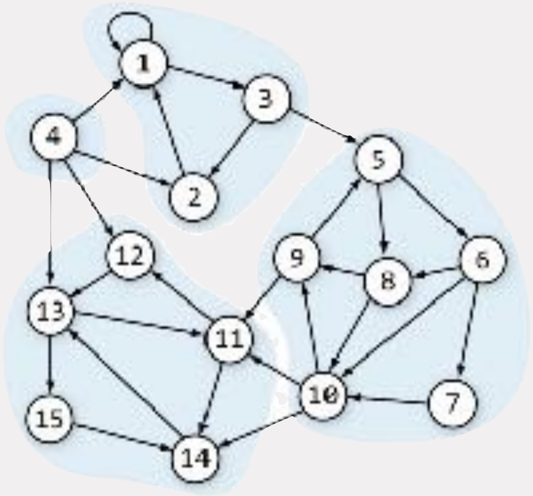
Strongly Connected Components



- **Sequential:** linear-time [Tarjan,'72], [Dijkstra & Feijen,'88]
- DFS based; hard to parallelise; next to impossible for many-core

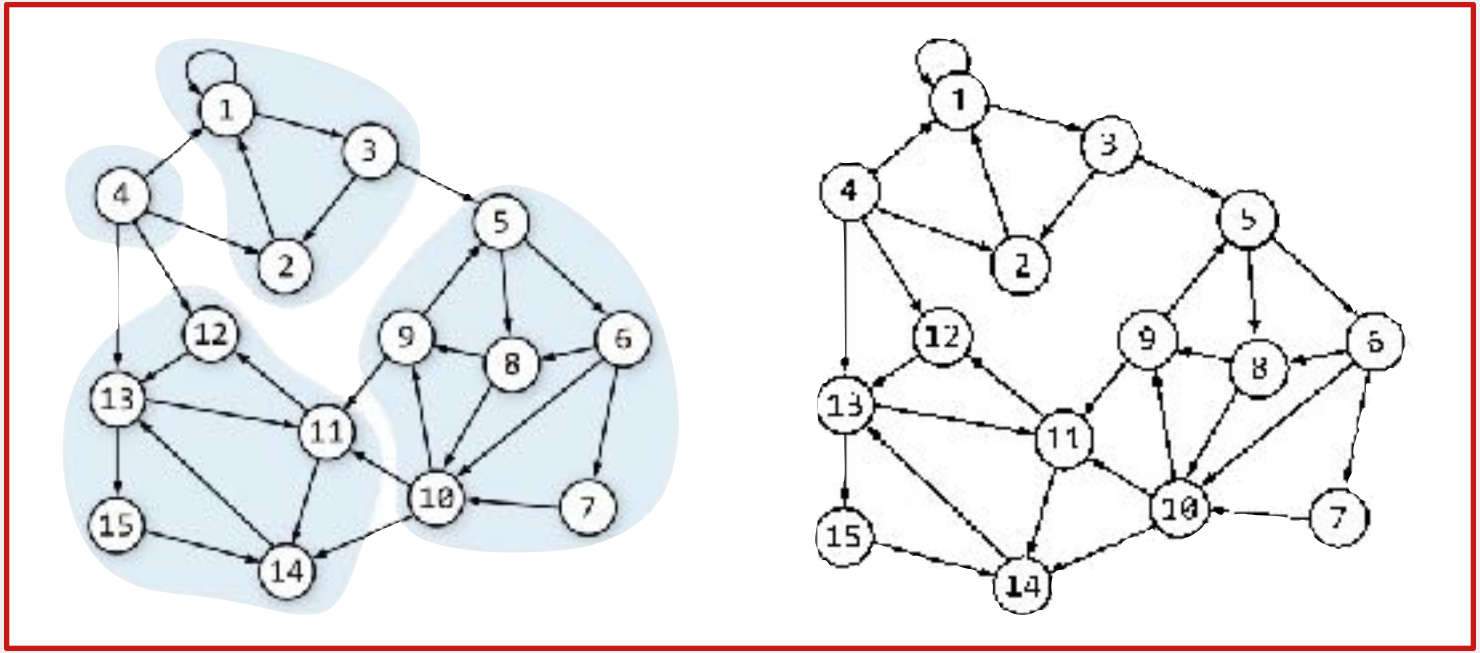


Strongly Connected Components

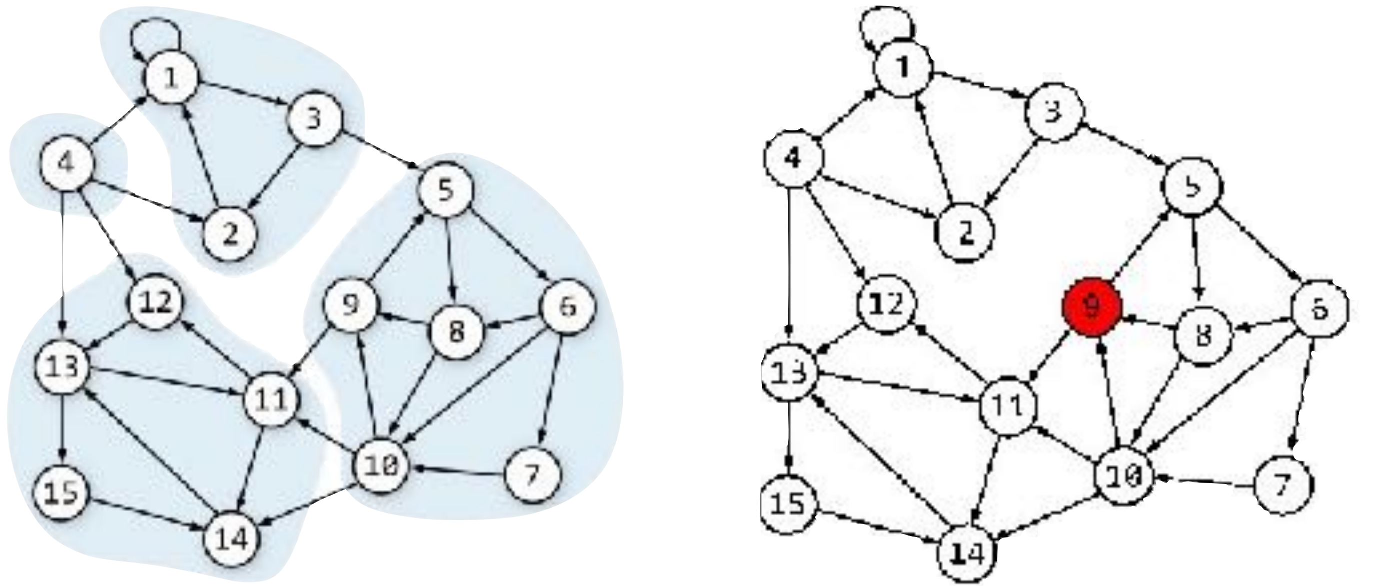


- Relevant for model checking, to detect potential for infinite (undesired) behaviour
- Study for alternatives [Barnat et al.,’11]
  - Forward-Backward BFS (with trimming) [Fleischer et al.’00]
  - For many graphs best option, disappointing for model checking problems (5x speedup)

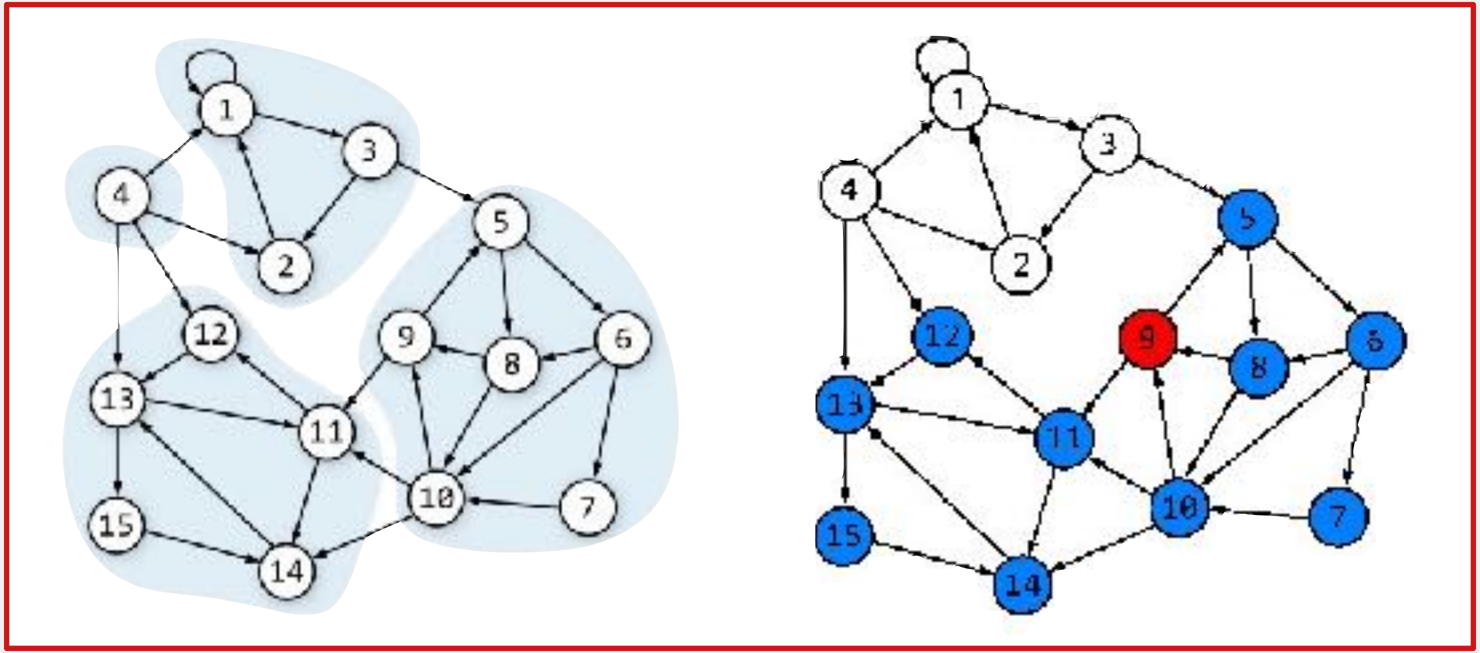
Strongly Connected Components



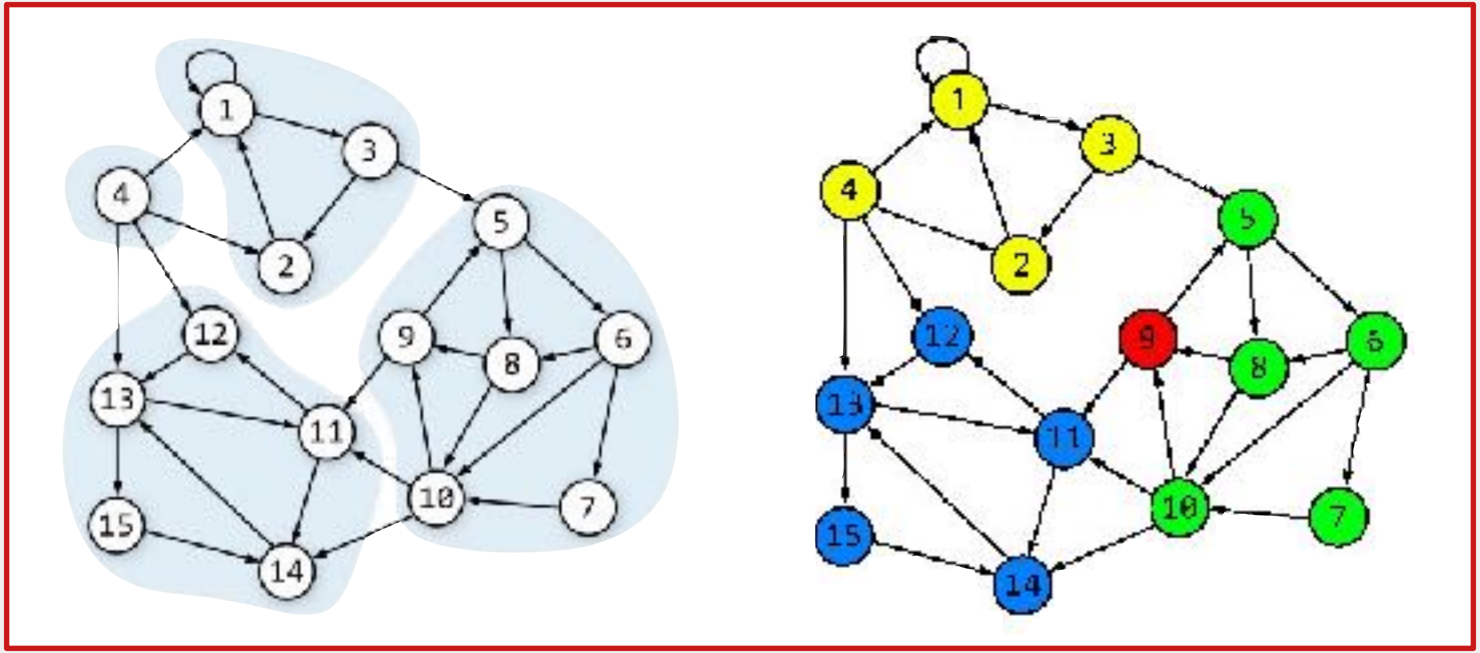
Strongly Connected Components



Strongly Connected Components



Strongly Connected Components



- Green states + state 9: an SCC
- Other SCCs either completely consist of yellow or blue states
- Trimming procedure: remove trivial SCCs (states with no in- or outgoing edge)

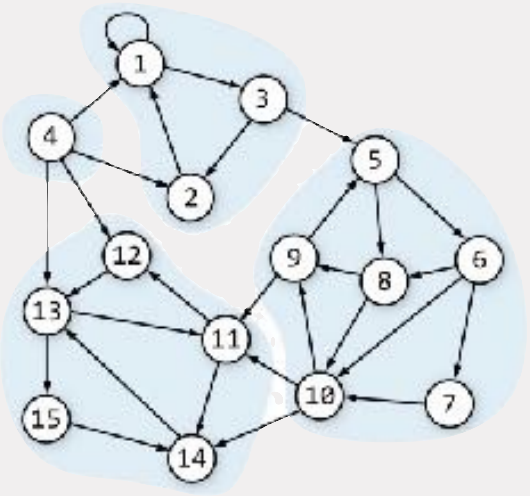
Forward / Backward BFS (with trimming)

Algorithm 1 FB with Trimming (FBT)

Require: graph $G = (V, E)$, set $J \subseteq V$
Ensure: SCC decomposition of G is given

$V' \leftarrow \textsc{Trim}(V)$ *produces trivial SCCs*
if $V' \neq \emptyset$ **then**
  $pivot \leftarrow \textsc{selectPivot}(V' \cap J)$
  $F \leftarrow \textsc{fwdBfs}(pivot, (V', E))$
  $B \leftarrow \textsc{bwdBfs}(pivot, (V', E))$
  remove SCC $F \cap B$ *from* V'
  do in parallel
    $\text{FBT}(((F \setminus B), E), J)$
    $\text{FBT}(((B \setminus F), E), J)$
    $\text{FBT}(((V' \setminus (B \cup F)), E), J)$

GPU SCC Decomposition



- Stored on GPU:
  - outgoing transitions per state
  - incoming transitions per state (transposed graph)
  - result per state (current search region)
    - Results[s] = Results[t] ⇒ s and t in same SCC
- Put pivots in search frontiers, scan transitions, put successors in frontiers, ...
- 3 bits per integer for search frontiers, closed sets, locking, ...

Data

- **Transitions contain state IDs, Results indicate SCCs**

			2									
Start of outgoing trans. of state *i* (offsets)			**5**	**7**								
Outgoing trans.						**X**	**X**					
Start incoming trans.												
Incoming trans.												
Results			**0**									

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0									

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0									

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0									

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			**5**	**7**								
Outgoing trans.						**5**	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0									

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0			0						

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0			0						

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			$\boxed{5}$	$\boxed{7}$								
Outgoing trans.						5	$\boxed{8}$					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0			0						

TU/e

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0			0			0			

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0			0			0			

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			$\boxed{3}$	5								
Incoming trans.				0	3							
Results			0			0			0			

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			$\boxed{3}$	$\boxed{5}$								
Incoming trans.				0	3							
Results			0			0			0			

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results			0			0			0			

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results	0		0			0			0			

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			$\boxed{3}$	$\boxed{5}$								
Incoming trans.				0	3							
Results	0		0			0			0			

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			$\boxed{3}$	$\boxed{5}$								
Incoming trans.				0	$\boxed{3}$							
Results	0		0			0			0			

Forward/Backward BFS

- One kernel launch: move both frontiers one step

Thread 2

Start of outgoing trans. of state *i* (offsets)			5	7								
Outgoing trans.						5	8					
Start incoming trans.			3	5								
Incoming trans.				0	3							
Results	0		0	0		0			0			

TU/e

BFS on a GPU

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
for (i ← Global-ThreadId; i < |V|; i ← i + NrOfThreads) do
    srcinfo ← offsets[i]
    if inFrontier(srcinfo) then
        offsets[i] ← movetoExplored(srcinfo)
        offset1 ← getOffset(srcinfo)
        offset2 ← getOffset(offsets[i + stepsize − (i mod stepsize)])
        for (j ← offset1; j < offset2; j ← j + stepsize) do
            t ← trans[j]
            if t ≠ empty then
                tgtstate ← getTgtstate(t)
                tgtinfo ← offsets[tgtstate]
                if isNew(tgtinfo) then
                    offsets[tgtstate] ← addToFrontier(tgtinfo)
```

Pivot selection

- Select for each new region **new pivot**
- Let threads with states in same region **race**
- **Reuse** outgoing trans as hash table
- Use atomic writes and write-lock bit

- **3 * Results[i] + inForward[i] + 2 * inBackward[i]**

Outgoing trans.												
Results			0			0						

Pivot selection

- Select for each new region **new pivot**
- Let threads with states in same region **race**
- **Reuse** outgoing trans as hash table
- Use atomic writes and write-lock bit

- **3 * Results[i] + inForward[i] + 2 * inBackward[i]**

Outgoing trans.												
Results			**0**			**0**						

Example for 2 and 5: inForward[i] = **true**, inBackward[i] = **false**

Pivot selection

- Select for each new region **new pivot**
- Let threads with states in same region **race**
- **Reuse** outgoing trans as hash table
- Use atomic writes and write-lock bit

- **3 * Results[i] + inForward[i] + 2 * inBackward[i]**

Outgoing trans.		**2**										
Results			**0**			**0**						

Example for 2 and 5: inForward[i] = **true**, inBackward[i] = **false**

Pivot selection

- Select for each new region **new pivot**
- Let threads with states in same region **race**
- **Reuse** outgoing trans as hash table
- Use atomic writes and write-lock bit

2 wins, 5 reads result

- **3 * Results[i] + inForward[i] + 2 * inBackward[i]**

Outgoing trans.		**2**										
Results			**0**			**0**						

Example for 2 and 5: inForward[i] = **true**, inBackward[i] = **false**

Pivot selection

- Select for each new region **new pivot**
- Let threads with states in same region **race**
- **Reuse** outgoing trans as hash table
- Use atomic writes and write-lock bit

2 wins, 5 reads result

- **3 * Results[i] + inForward[i] + 2 * inBackward[i]**

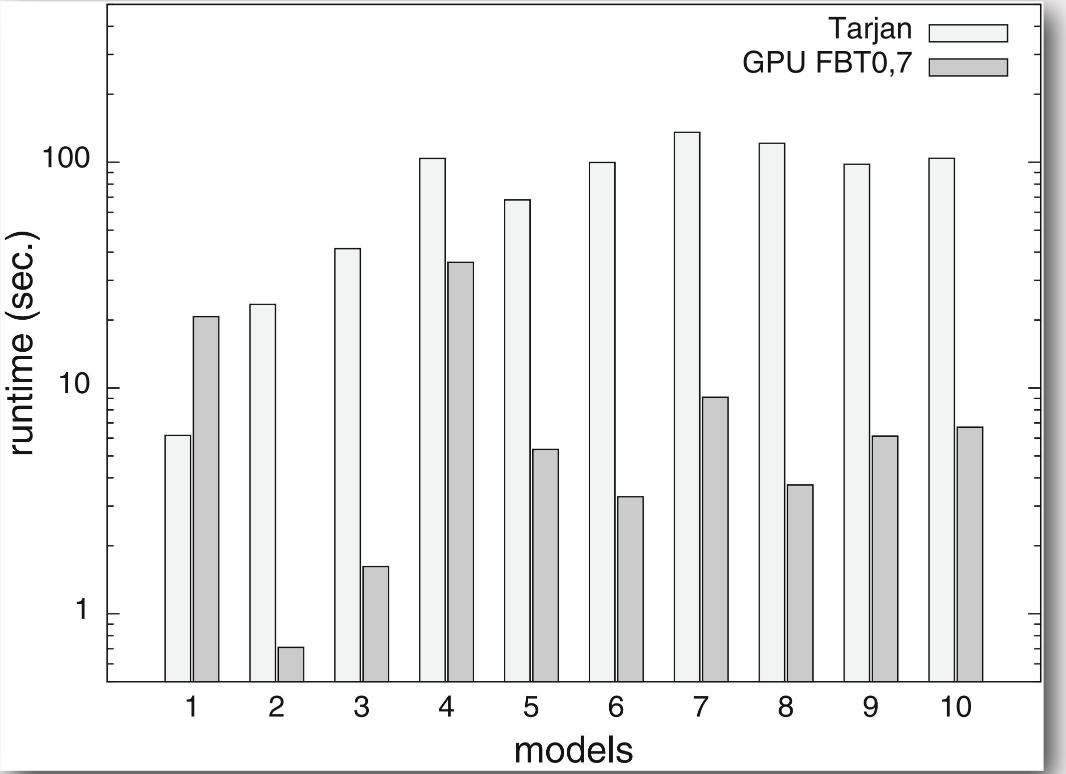
Outgoing trans.		**2**										
Results			**2**			**2**						

Example for 2 and 5: inForward[i] = **true**, inBackward[i] = **false**

Benchmark characteristics

Model		|V| (M)	|E| (M)	av. out	max. out	#SCCs
1	wlan.2500	12.6	28.1	2.23	129	12.5 M
2	phil.7	11.0	98.5	8.97	14	1
3	diningcrypt.10	42.9	279.4	6.51	20	42.9 M
4	test-and-set.7	51.4	468.5	9.12	17	4,672
5	leader.7	68.7	280.5	4.08	14	42.2 M
6	phil_lss.5.10	72.9	425.6	5.84	10	1
7	coin.8.3	87.9	583.0	6.63	16	5.4 M
8	mutual.7.13	76.2	653.7	8.58	14	1
9	zeroconf_dl.F.200.1k.6	118.6	273.5	2.31	10	118.6 M
10	firewire_dl.800.36.(0.2)	129.3	293.6	2.27	5	129.3 M

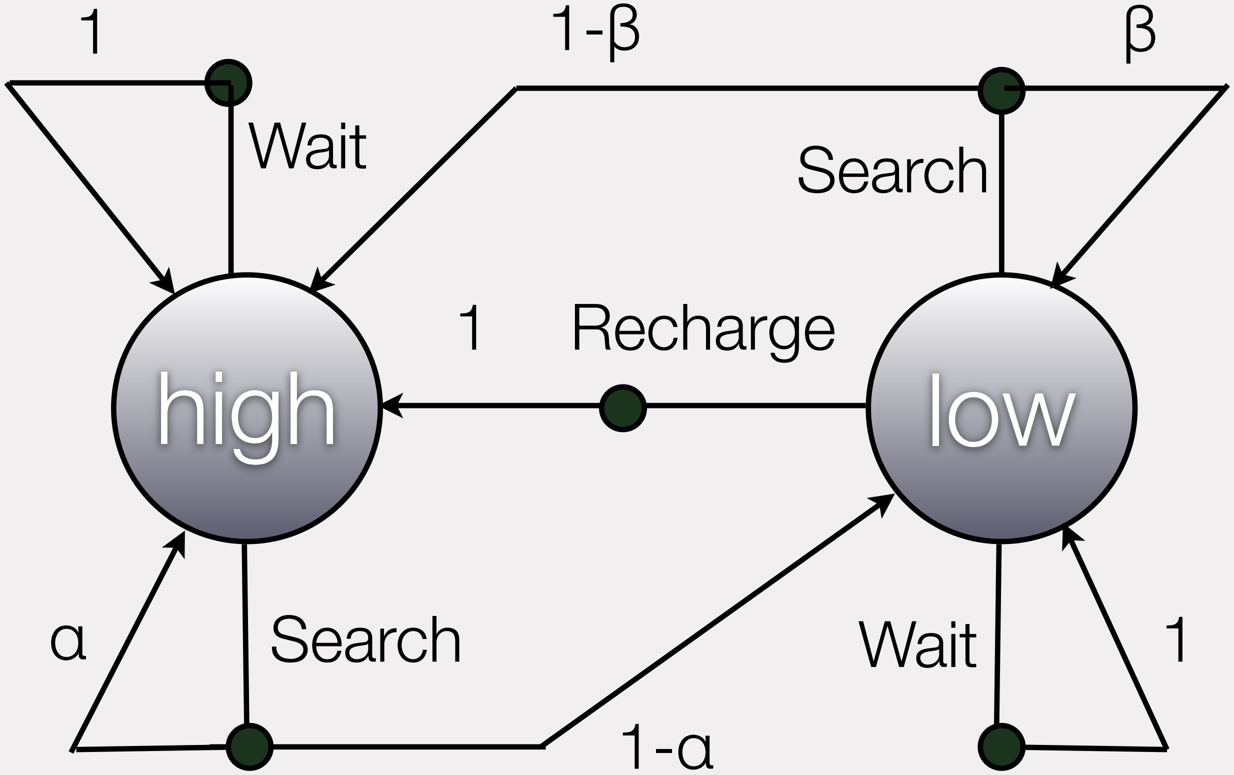
SCC decomposition results



15-30x speedup

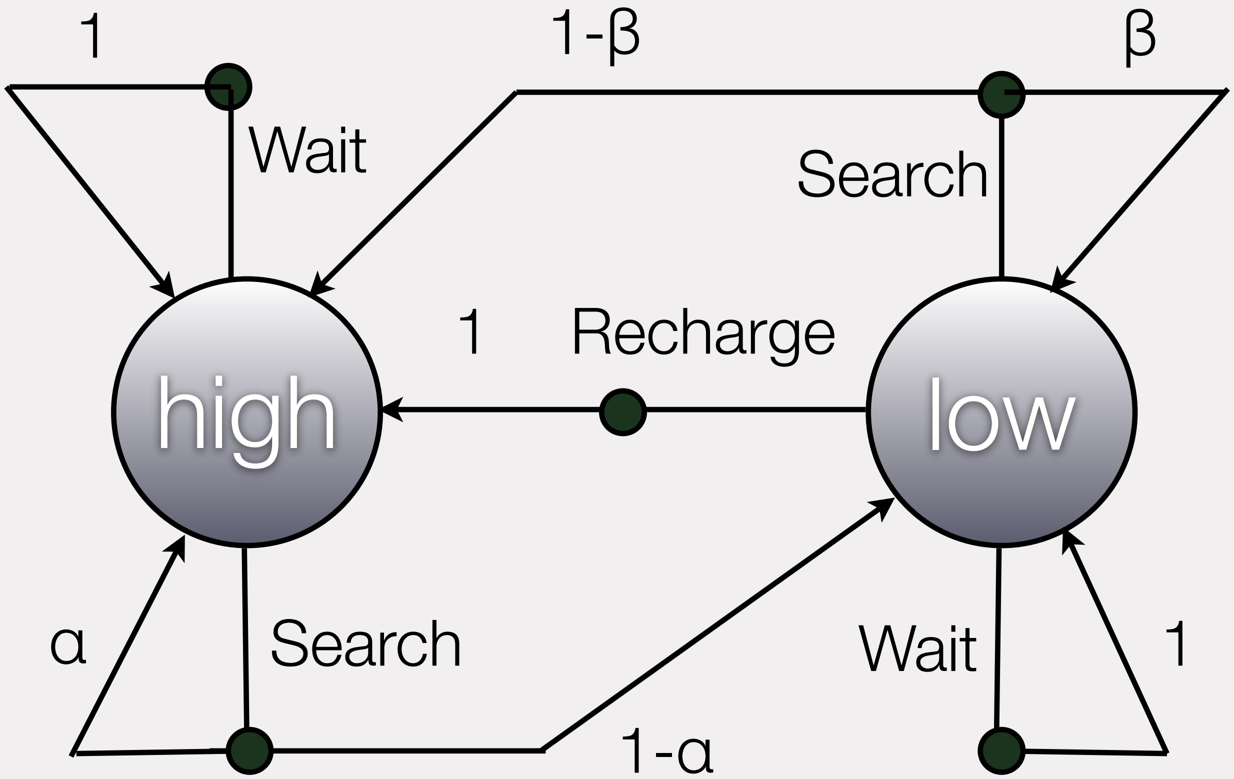
NVIDIA K20 GPU
5 GB global memory
13 SMs / 2,496 SPs

Markov Decision Process (MDP)



- **MDP:** model probabilistic systems with components
- Non-deterministic and probabilistic choice

Markov Decision Process (MDP)



- (Maximal) End Component (MEC) generalises SCC
- Limiting properties that hold with probability 1.0
- Randomised algorithms, stochastic games,…
- **V is EC iff (1) V is SCC, (2) for all v in V, exists probability distribution staying in V**

MEC decomposition

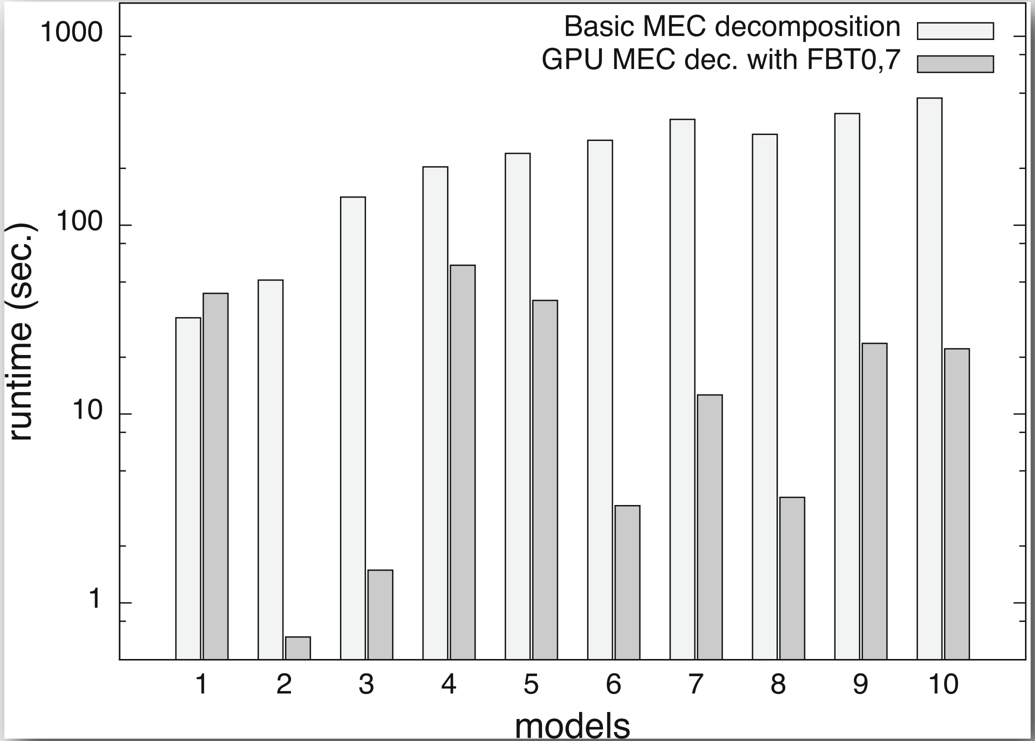
- [Chatterjee & Henzinger,'12]
    1. Compute SCCs
    2. For each SCC C, determine U with no probability distribution staying in C
    3. if U not empty: Remove Attr$(U \cap C)$
    4. else: C is MEC
    5. Goto step 1

- Attr(U) contains U plus all vertices that can reach U regardless of the resolution of the non-deterministic choice(s)

MEC decomposition results

- **Up to 79x speedup**

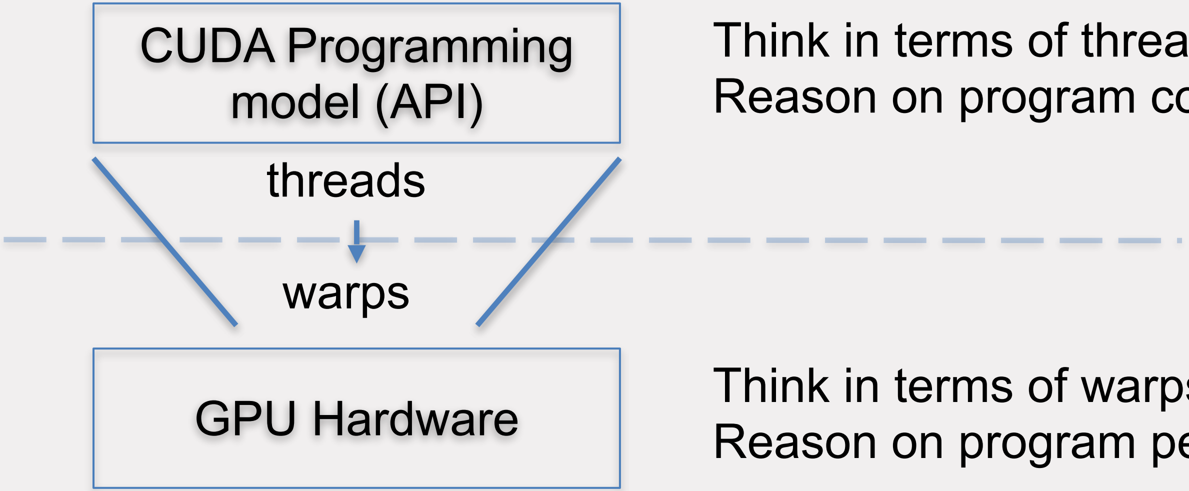
- Besides SCC decomposition, other steps are **pleasantly parallel** for GPUs



Other results

- GPU accelerated **Bounded Forward / Backward BFS**
  - When one BFS finishes, bound other BFS to same region
  - Better complexity, but limits potential for parallelism
- **Informed** pivot selection
- **No noticeable improvements on the GPU**

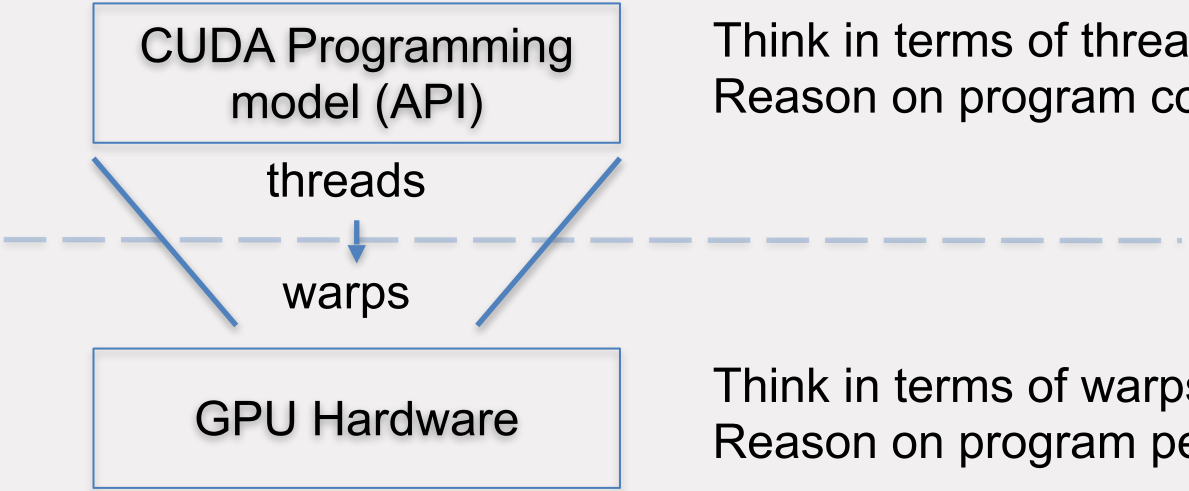
Overview



Think in terms of threads
Reason on program correctness

Think in terms of warps
Reason on program performance

Overview



Think in terms of threads
Reason on program correctness

Think in terms of warps
Reason on program performance

Tomorrow in Part 2 of Accelerated Verification!