Timing predictability of GPUs: challenges and advances

The Nvidia Pascal case

Thomas Carle, Univ. Toulouse 3, IRIT
TRACES group

Introduction

- Embedded Real-Time systems
  - Need for strong **timing guarantees** (WCET)
  - Measurements/probabilities vs **Static Analysis**
    - Microarchitectural details (**HW model**) + execution model (**Program semantics**)
    - Mature techniques for CPU targets

- Models for static WCET analysis
  - Hardware model
    - Pipeline depth, number and latency of functional units, instruction queues, etc.
    - Caches size and configuration
    - Memory hierarchy
    - Predictors, prefetchers
    - Traditionally obtained from the documentation
  - Software model
    - Control Flow Graph of the disassembled binary program

Introduction

- Graphical Processing Units: highly parallel accelerators
  - Major differences w.r.t. CPUs
    - Built for **average throughput** maximization (vs latency minimization)
      - Exploit parallelism to **hide latencies**
        - Instruction Level Parallelism
        - Data parallelism
      - Many functional units (aka CUDA Cores – CCs)
      - Feed the beast: complex **memory hierarchy**
    - Handle thousands of threads:
      - **Hierarchical scheduler** (kernels, blocks, warps)
      - **SIMT** execution model
  - Very efficient for e.g. matrix/tensor computations
    - Embedded AI applications (e.g. autonomous vehicles)

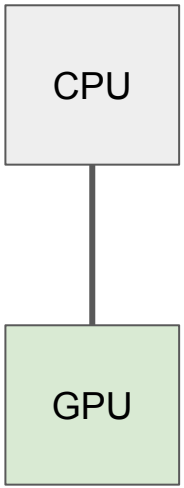
Introduction

- Graphical Processing Units: highly parallel accelerators
  - Major differences w.r.t. CPUs
    - Built for **average throughput** maximization (vs latency minimization)
      - Exploit parallelism to **hide latencies**
        - Instruction Level Parallelism
        - Data parallelism
      - Many functional units (aka CUDA Cores – CCs)
      - Feed the beast: complex **memory hierarchy**
    - Handle thousands of threads:
      - **Hierarchical scheduler** (kernels, blocks, warps)
      - **SIMT** execution model
  - Very efficient for e.g. matrix/tensor computations
    - Embedded AI applications (e.g. autonomous vehicles)

Can we build an accurate model to predict the timing behavior of a program running on a GPU ?

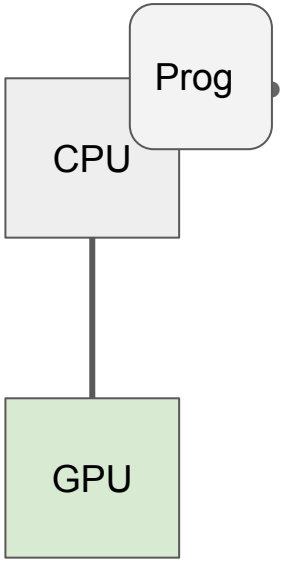
A quick introduction to GPUs and CUDA nomenclature

- Coarse grain vision



A quick introduction to GPUs and CUDA nomenclature

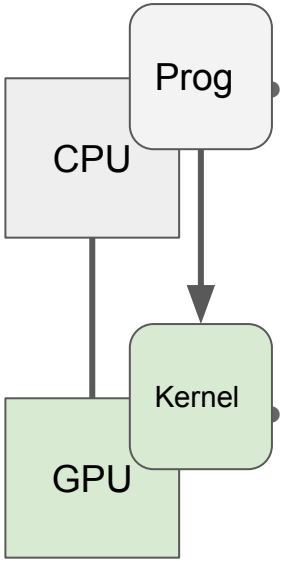
- Coarse grain vision



Main program written and compiled for the CPU target
  - Sequential or parallel
  - Makes calls to GPU functions a.k.a. **kernels**

A quick introduction to GPUs and CUDA nomenclature

- Coarse grain vision

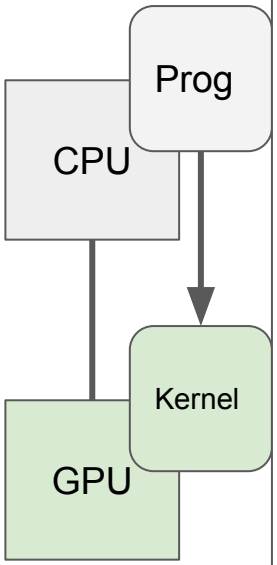


Main program written and compiled for the CPU target
- Sequential or parallel
- Makes calls to GPU functions a.k.a. **kernels**

Kernels written and compiled for the GPU
- **Inherently parallel**
- Written in C/C++ with extensions (e.g. CUDA, OpenCL, Vulkan)
- Multiple threads execute the **same code** on **different data**
  - Each thread has a **unique identifier**

A quick introduction to GPUs and CUDA nomenclature

- Coarse g



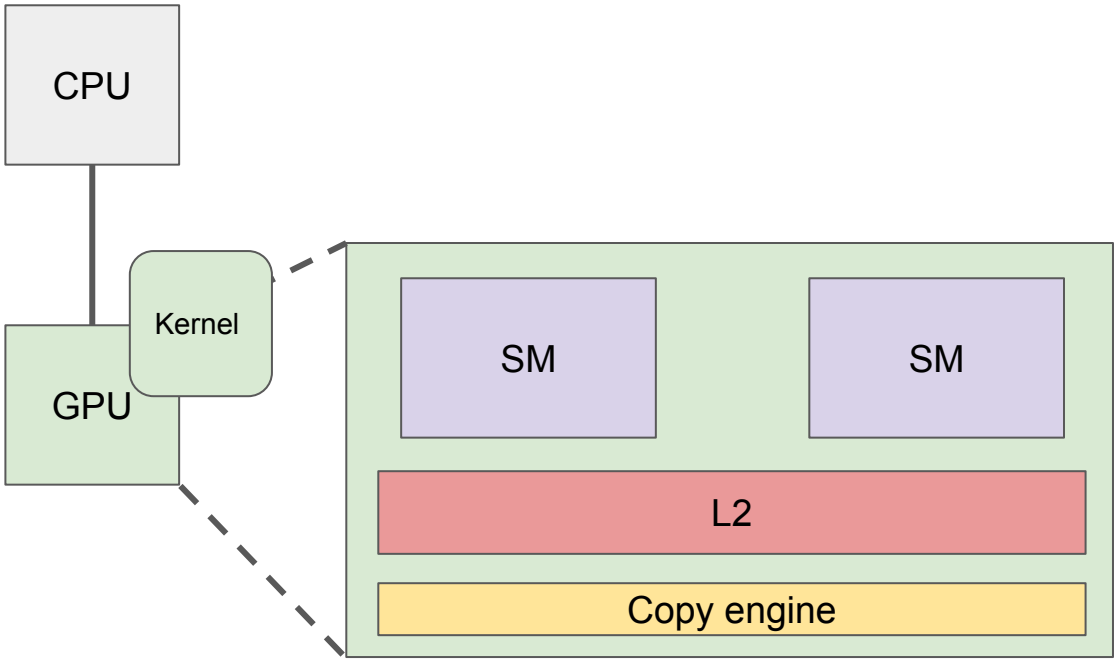

```
int main(){
      int t[2048];
      int* d_t;
      cudaMalloc(d_t, 2048*sizeof(int));
      init_table<<<2,1024>>>(d_t, 2048);
      cudaMemcpy(t, d_t, 2048*sizeof(int), cudaMemcpyDeviceToHost);
      return 0;
}


__global__ void init_table(int* table, int size_table){
      int tx = threadIdx.x;
      int bx = blockIdx.x;
      int idxX = bx * blockDim.x + tx;

      if( idxX >= size_table)
            return;
      table[idxX] = 0;
}
```

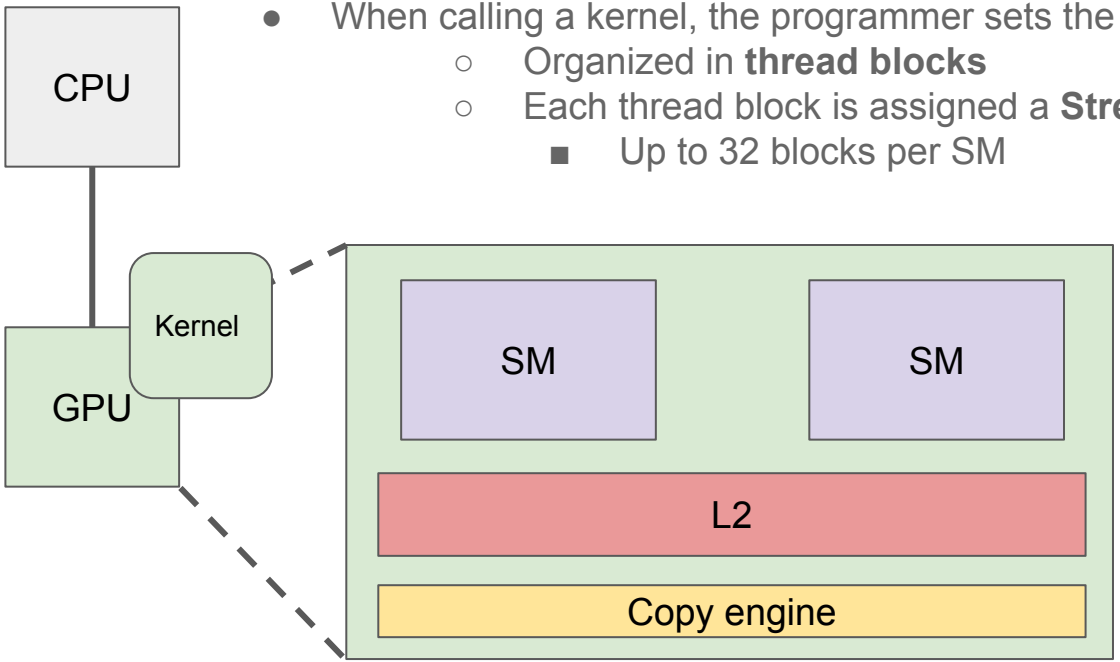
A quick introduction to GPUs and CUDA nomenclature

- Entering the GPU



A quick introduction to GPUs and CUDA nomenclature

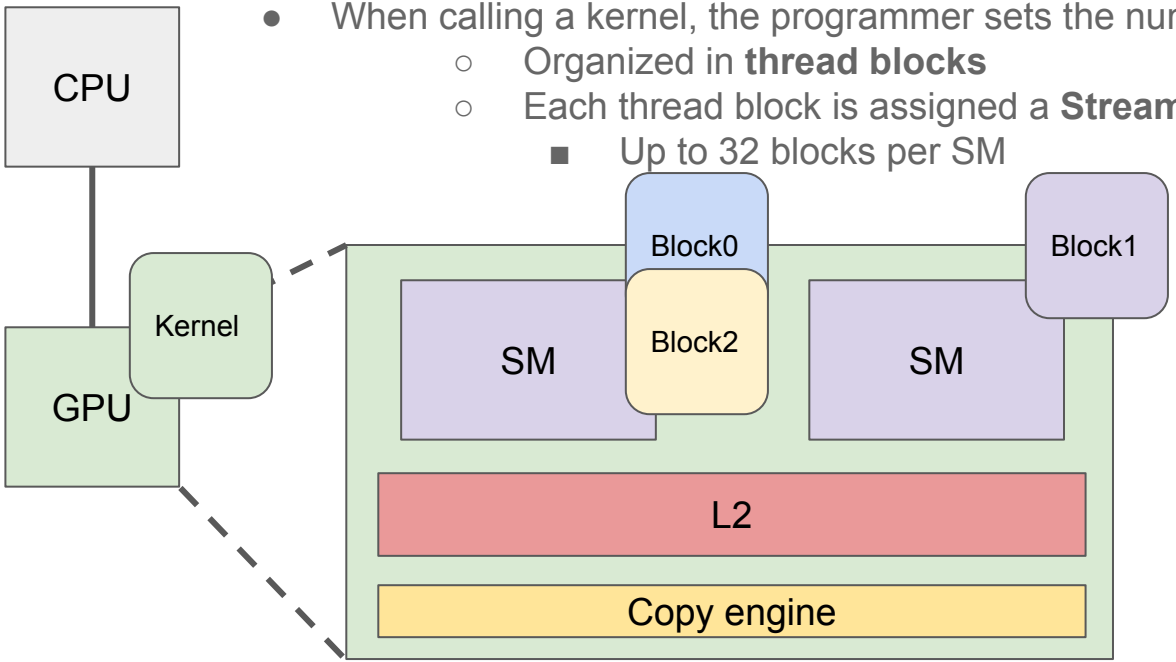
- Entering the GPU



- When calling a kernel, the programmer sets the number of threads to execute it
  - Organized in **thread blocks**
  - Each thread block is assigned a **Streaming Multiprocessor**
    - Up to 32 blocks per SM

A quick introduction to GPUs and CUDA nomenclature

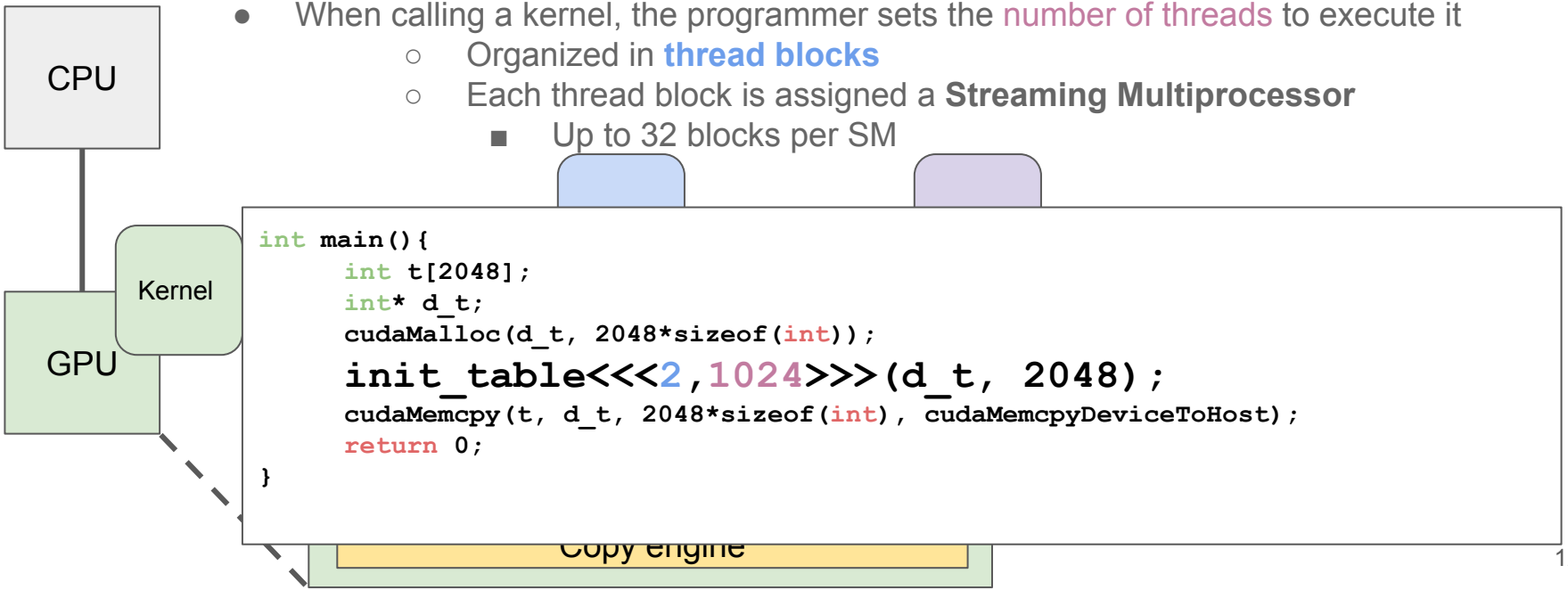
- Entering the GPU



- When calling a kernel, the programmer sets the number of threads to execute it
  - Organized in **thread blocks**
  - Each thread block is assigned a **Streaming Multiprocessor**
    - Up to 32 blocks per SM

A quick introduction to GPUs and CUDA nomenclature

- Entering the GPU

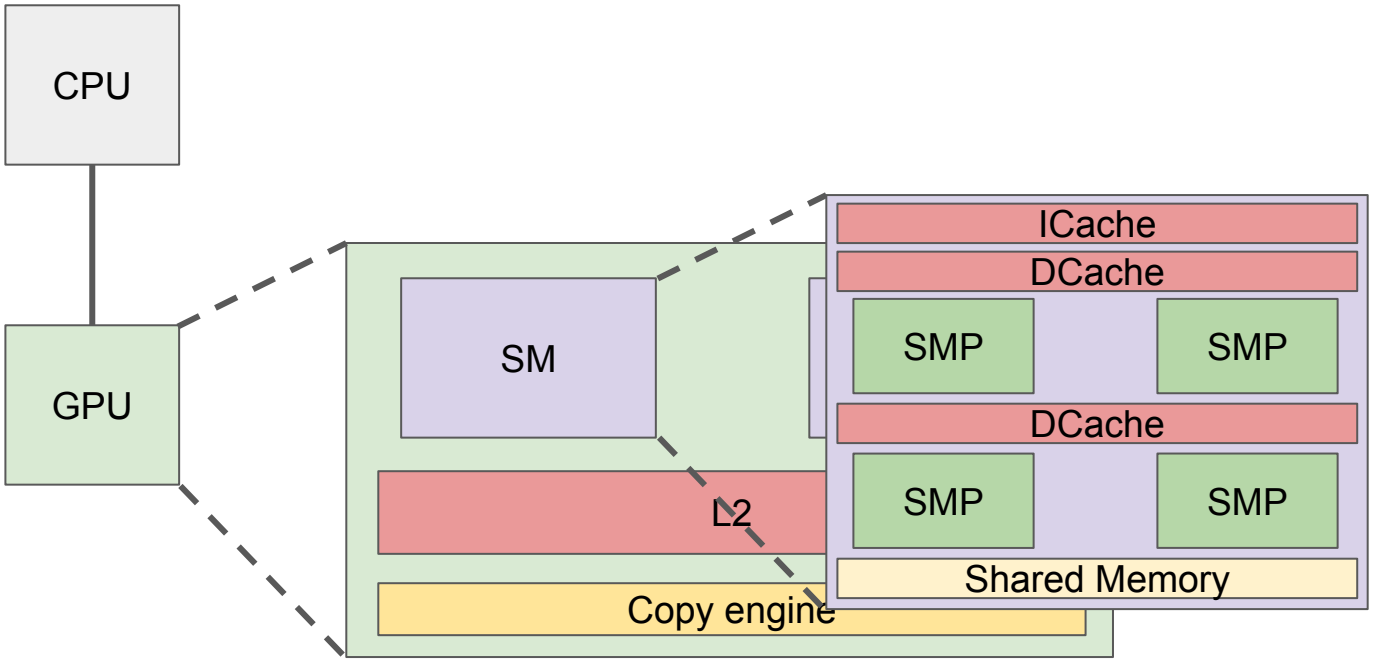


- When calling a kernel, the programmer sets the number of threads to execute it
  - Organized in **thread blocks**
  - Each thread block is assigned a **Streaming Multiprocessor**
    - Up to 32 blocks per SM

```
int main(){
      int t[2048];
      int* d_t;
      cudaMalloc(d_t, 2048*sizeof(int));
      init_table<<<2,1024>>>(d_t, 2048);
      cudaMemcpy(t, d_t, 2048*sizeof(int), cudaMemcpyDeviceToHost);
      return 0;
}
```

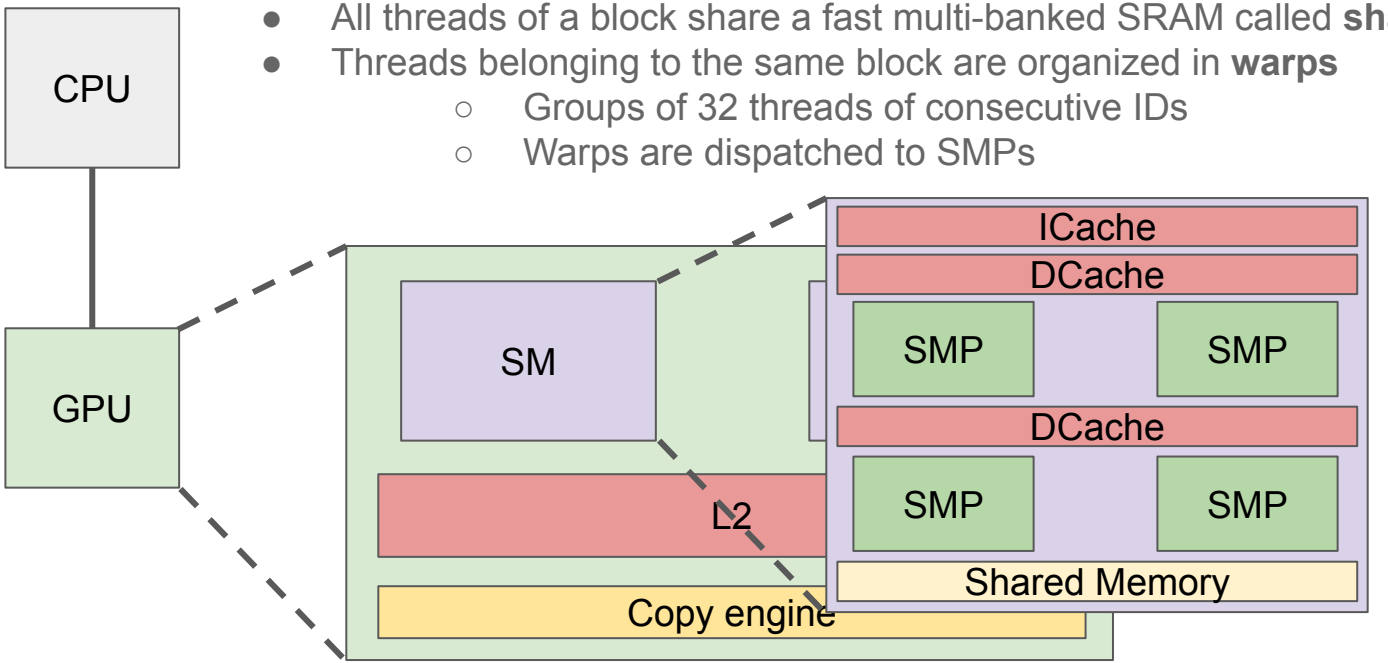
A quick introduction to GPUs and CUDA nomenclature

- Inside a SM



A quick introduction to GPUs and CUDA nomenclature

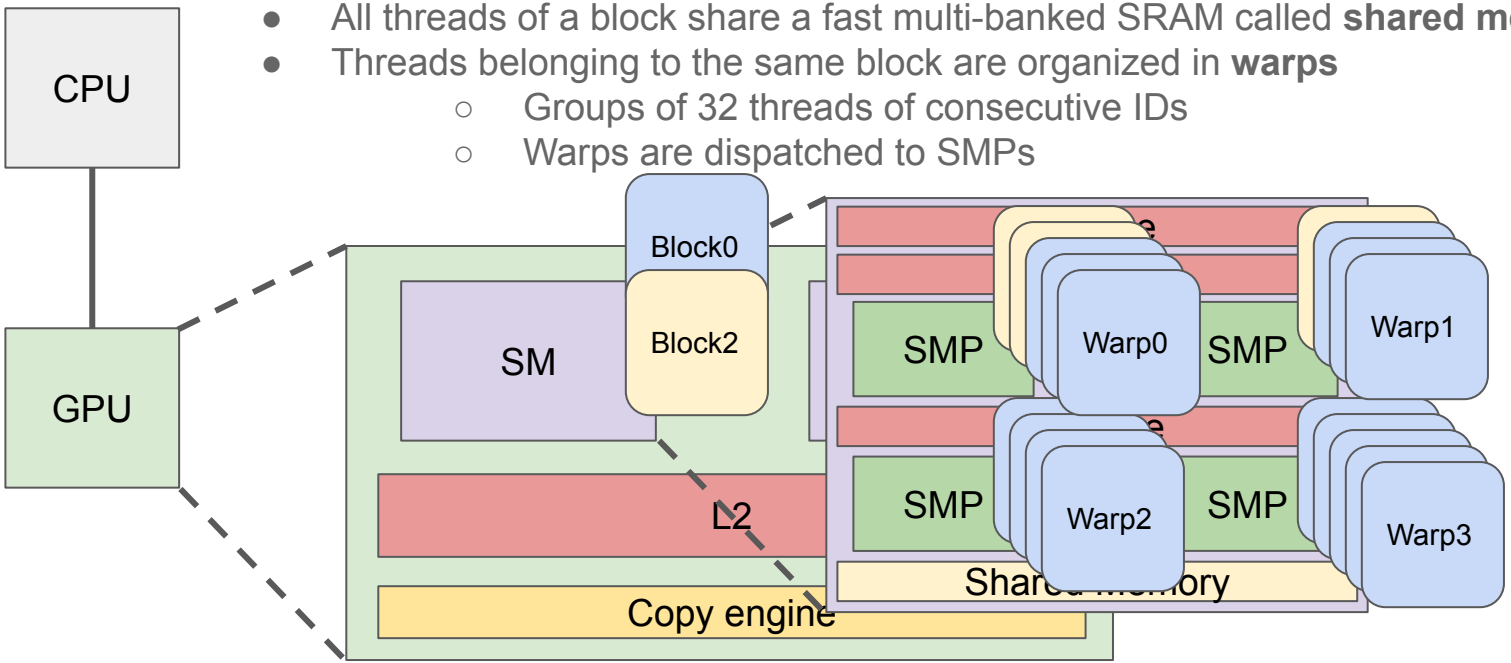
- Inside a SM



- All threads of a block share a fast multi-banked SRAM called **shared memory**
- Threads belonging to the same block are organized in **warps**
  - Groups of 32 threads of consecutive IDs
  - Warps are dispatched to SMPs

A quick introduction to GPUs and CUDA nomenclature

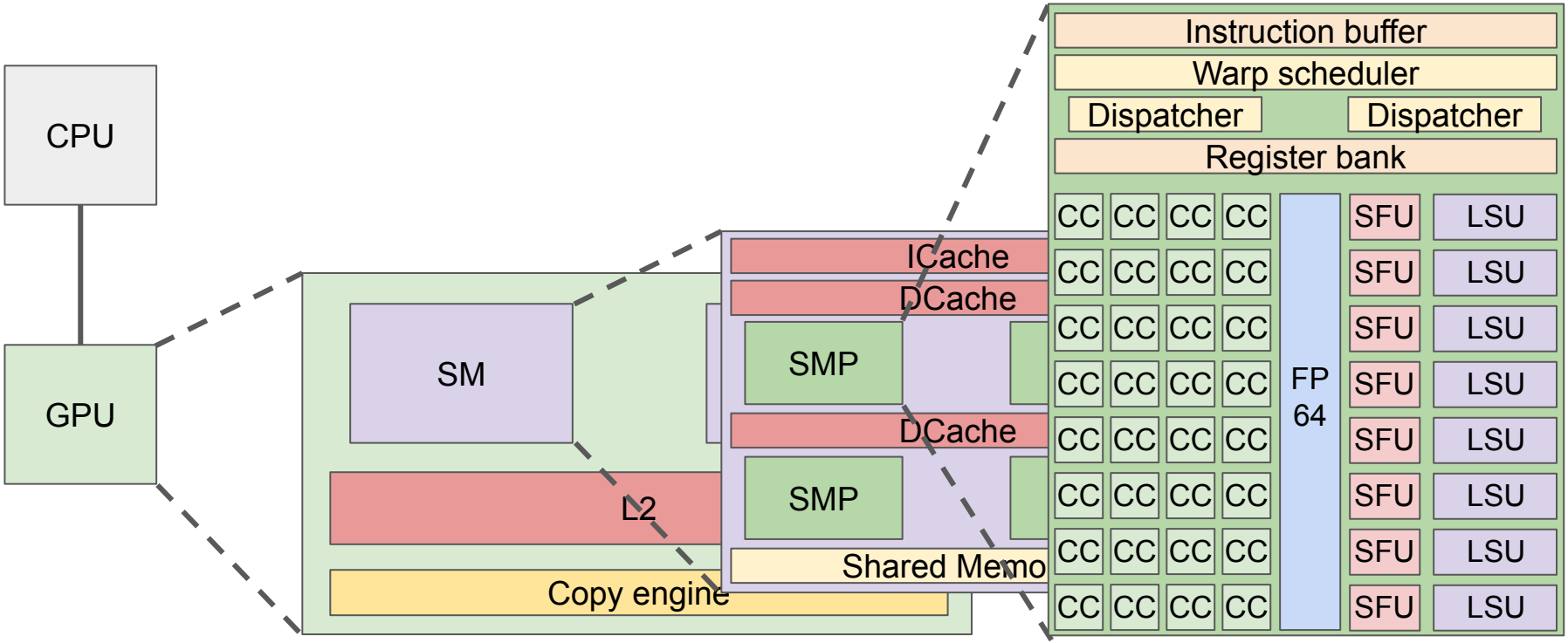
- Inside a SM



- All threads of a block share a fast multi-banked SRAM called **shared memory**
- Threads belonging to the same block are organized in **warps**
  - Groups of 32 threads of consecutive IDs
  - Warps are dispatched to SMPs

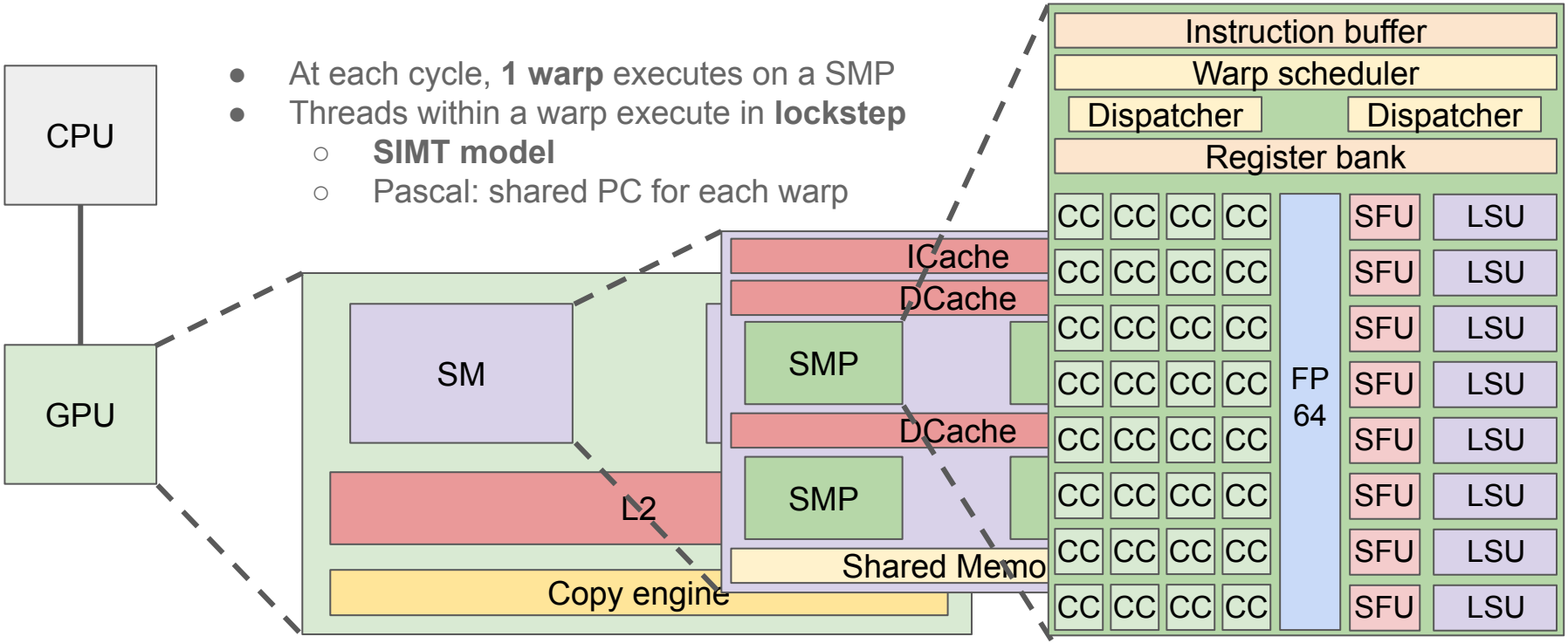
A quick introduction to GPUs and CUDA nomenclature

- Inside a SMP



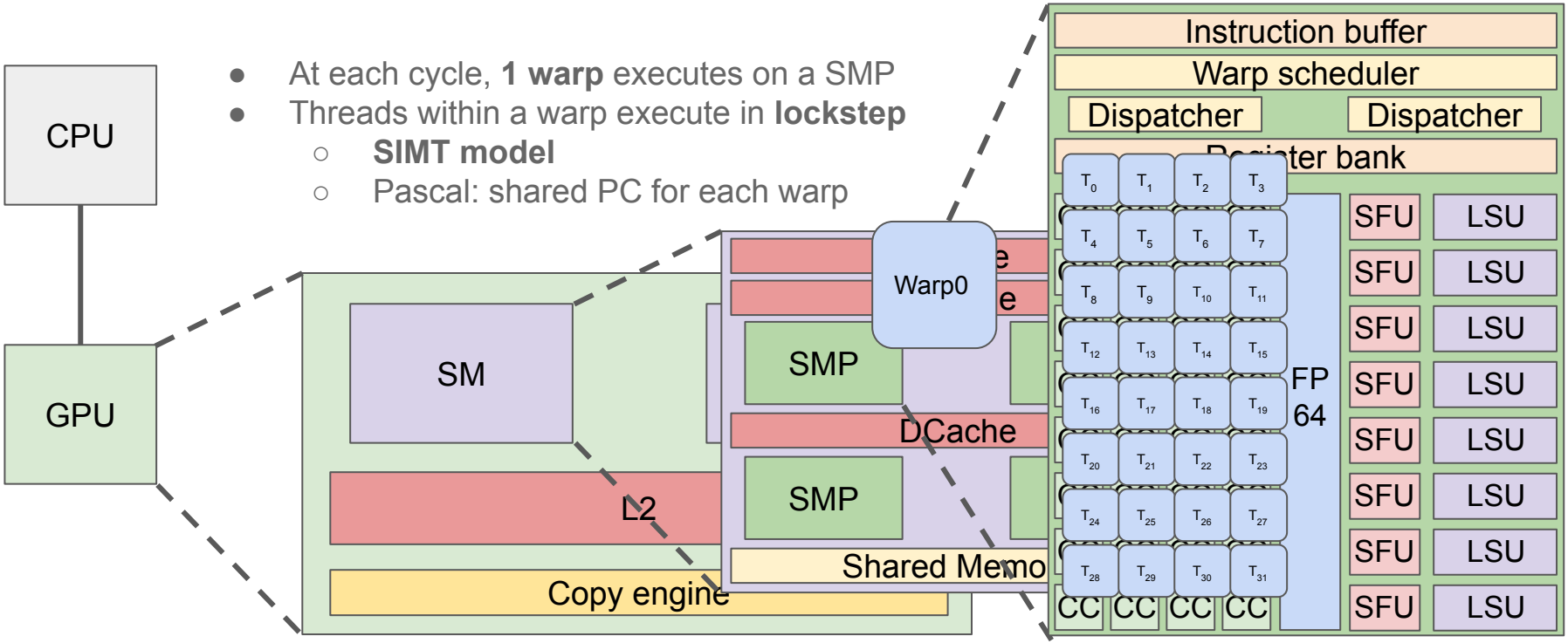
A quick introduction to GPUs and CUDA nomenclature

- Inside a SMP



A quick introduction to GPUs and CUDA nomenclature

- Inside a SMP



Nvidia Jetson TX2

- Embedded System-on-Chip
  - 6 ARM cores
  - 1 Pascal GPU
    - 2 SMs, 8 SMPs, 256 Cuda Cores
    - "Embedded GPU"
- Launched in 2016
  - Branded as go-to chip for embedded applications, including autonomous driving
  - Research results now available – mostly based on reverse engineering
  - Newer models introduce new features, but built on the same base mechanisms

Latency hiding mechanisms

- Software-defined scoreboard: **exploit ILP**
  - Upon long latency instructions, continue execution until the produced data is actually needed
  - SCHI instructions + DEPBAR instructions
    - Compiler optimizations
- Warp scheduler: **exploit data parallelism**
  - Swap the active warp when stalled
  - 1 per SMP
  - Looks for ready warps
    - Instruction buffer for each active warp
      - Contains the next instruction(s) for each warp
    - Software-defined scoreboard
  - Scheduling policy is unknown
    - Allegedly **Loose Round-Robin** or **Greedy-Then-Oldest**
    - Potentially **Two-level scheduler**
    - Very hard to assess experimentally
    - Influences **capacity to hide latency + cache contents / locality**

Latency hiding mechanisms

- Software-defined **scoreboard: exploit ILP**
  - Upon long latenc… … data is actually needed
  - SCHI instructions…
    - Compiler o…
- Warp scheduler: **e…**
  - Swap the active …
  - 1 per SMP
  - Looks for ready …
    - Instruction …
      - Conta…
    - Software-d…
  - Scheduling policy…
    - Allegedly **L…**
    - Potentially …
    - Very hard t…
    - Influences … **…ity**

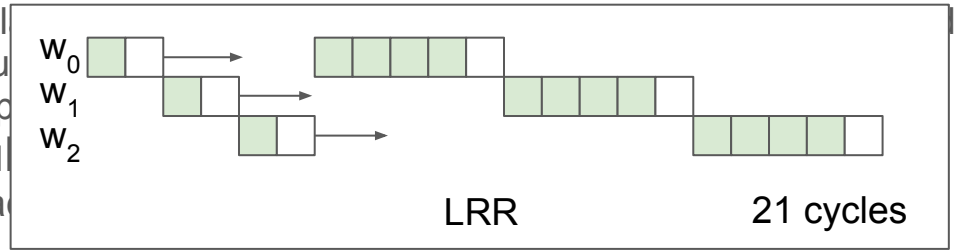
```
SCHI % each instr increments SB1
LD R1, [R2] % SB1 = 1
LD R3, [R4] % SB1 = 2
LD R5, [R6] % SB1 = 3
SCHI % not relevant
ADD R7, R8, R9 % not dependent
DEPBAR SB1, 2
FMUL R1, R1, R10 % dependent on R1
SCHI % not relevant
DEPBAR SB1, 1
FMUL R3, R3, R11 % dependent on R3
DEPBAR SB1, 0
SCHI % not relevant
FMUL R5, R5, R12 % dependent on R5
```

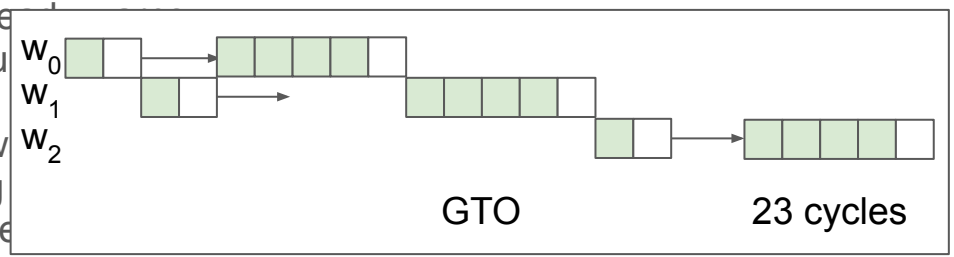
Latency hiding mechanisms

- Software-defined scoreboard: **exploit ILP**
  - Upon long latency instructions, continue execution until the produced data is actually needed
  - SCHI instructions + DEPBAR instructions
    - Compiler optimizations
- Warp scheduler: **exploit data parallelism**
  - Swap the active warp when stalled
  - 1 per SMP
  - Looks for ready warps
    - Instruction buffer for each active warp
      - Contains the next instruction(s) for each warp
    - Software-defined scoreboard
  - Scheduling policy is unknown
    - Allegedly **Loose Round-Robin** or **Greedy-Then-Oldest**
    - Potentially **Two-level scheduler**
    - Very hard to assess experimentally
    - Influences **capacity to hide latency + cache contents / locality**

Latency hiding mechanisms

- Software-defined scoreboard: **exploit ILP**
  - Upon long l… …l data is actually needed
  - SCHI instru…
    - Comp…
- Warp schedul…
  - Swap the a…
  - 1 per SMP
  - Looks for re…
    - Instru…
      - …
    - Softw…
  - Scheduling …
    - Allege…
    - Potentially **Two-level scheduler**
    - Very hard to assess experimentally
    - Influences **capacity to hide latency + cache contents / locality**



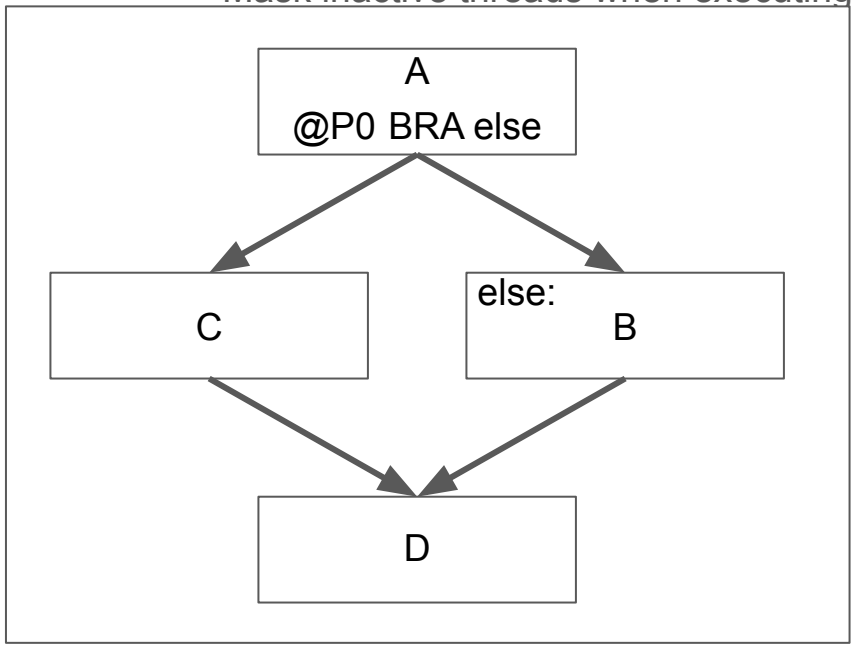


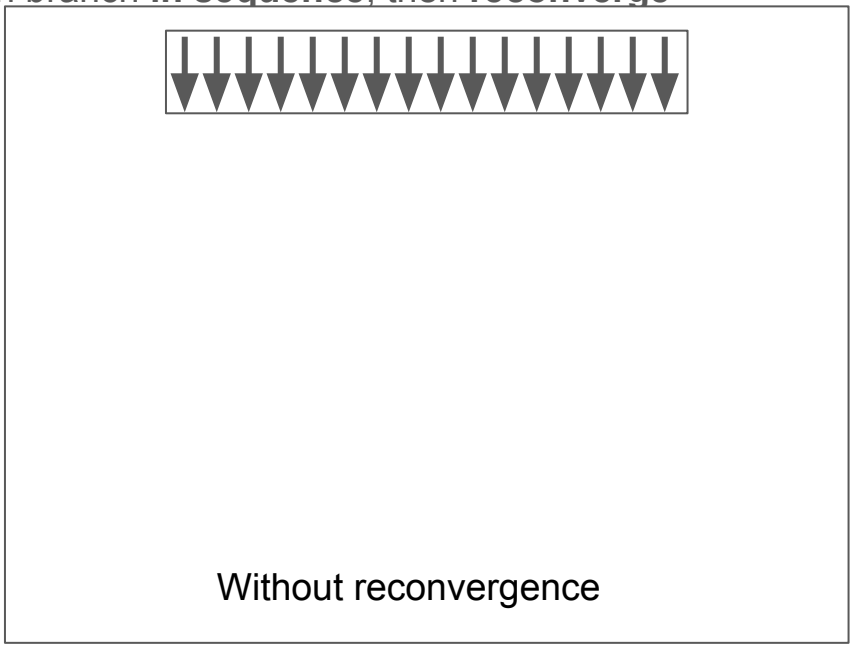
SIMT model and thread divergence

- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

SIMT model and thread divergence

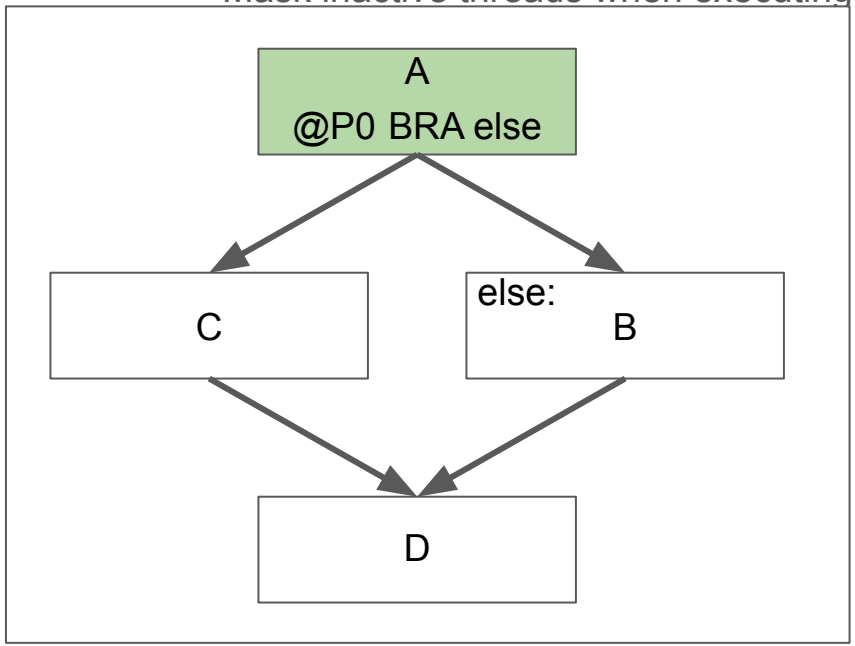
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

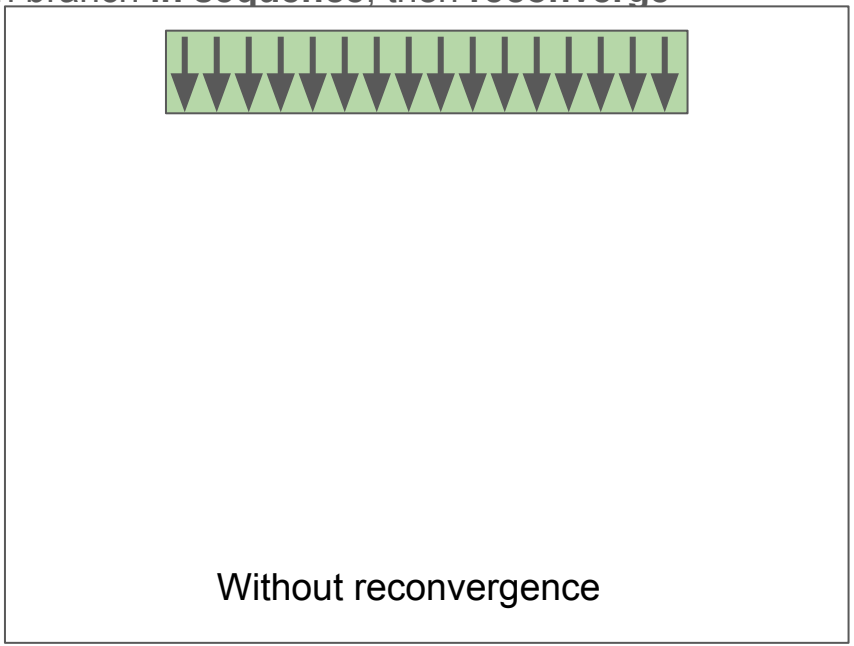




SIMT model and thread divergence

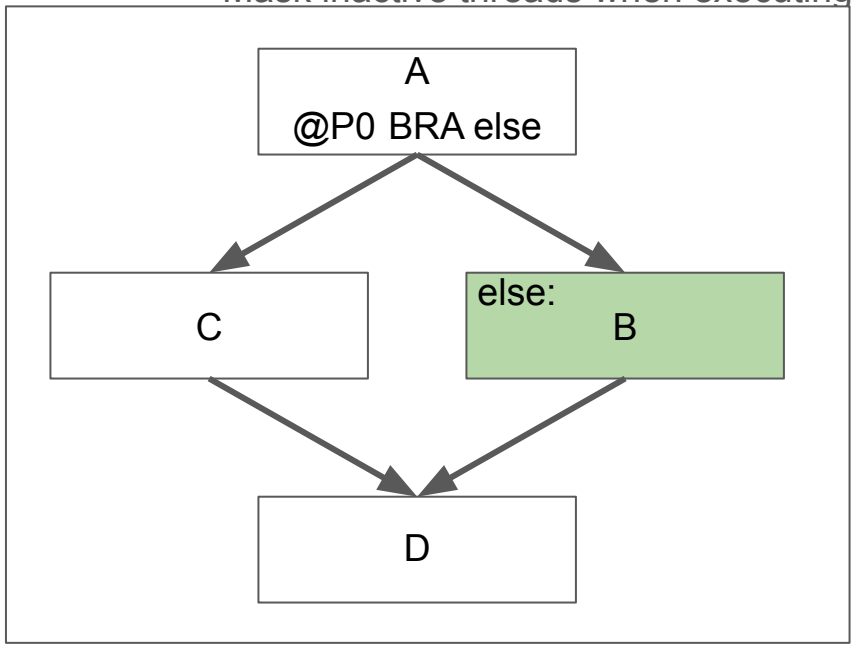
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

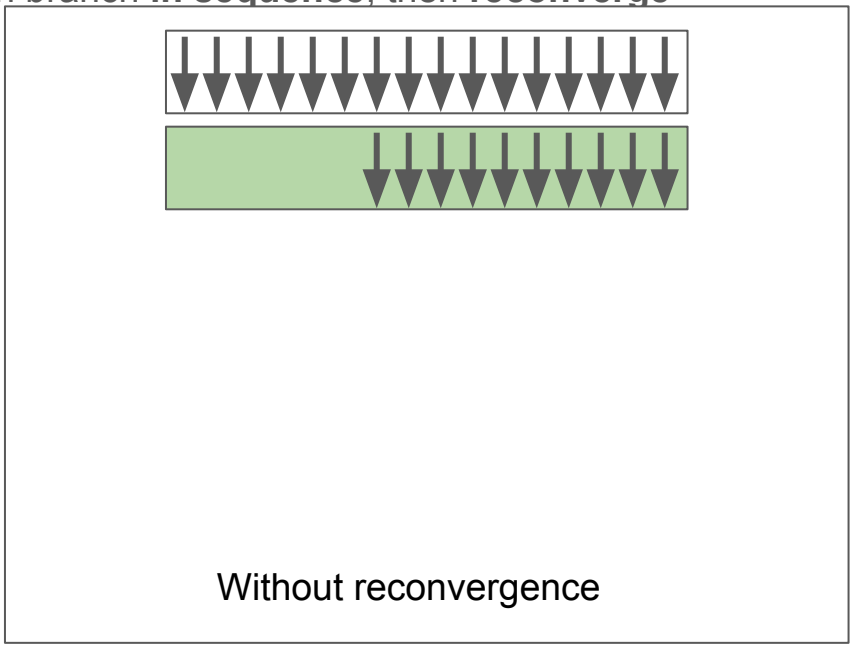




SIMT model and thread divergence

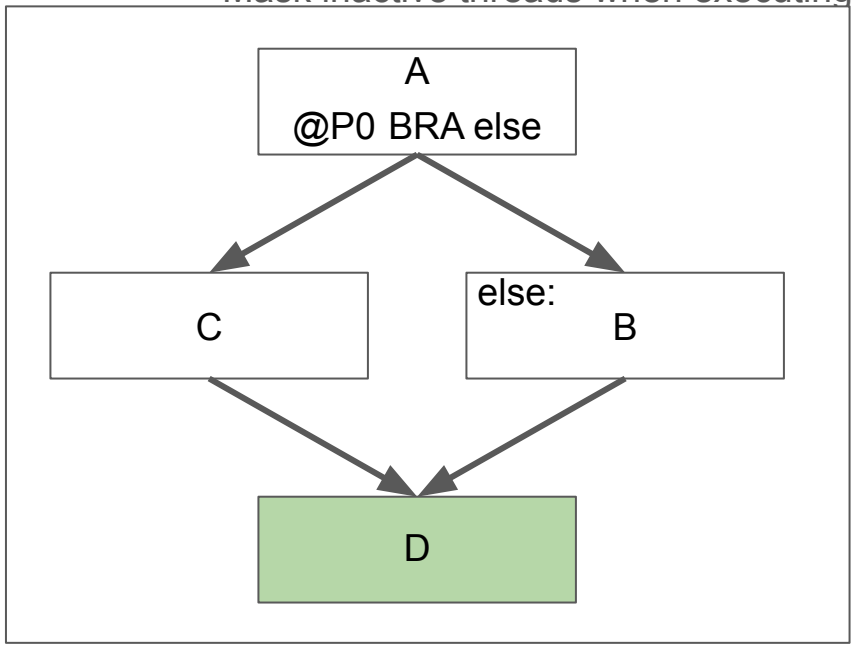
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

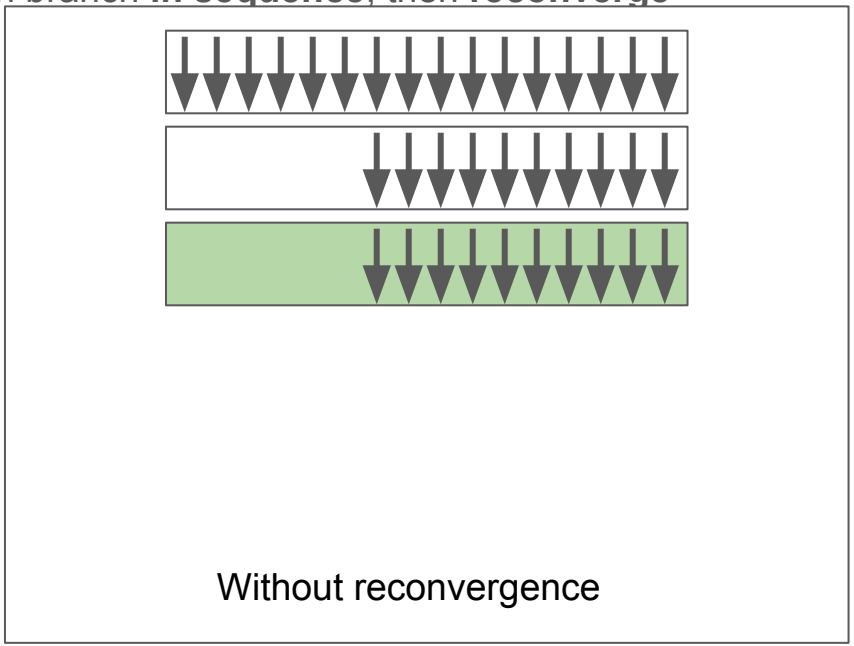




SIMT model and thread divergence

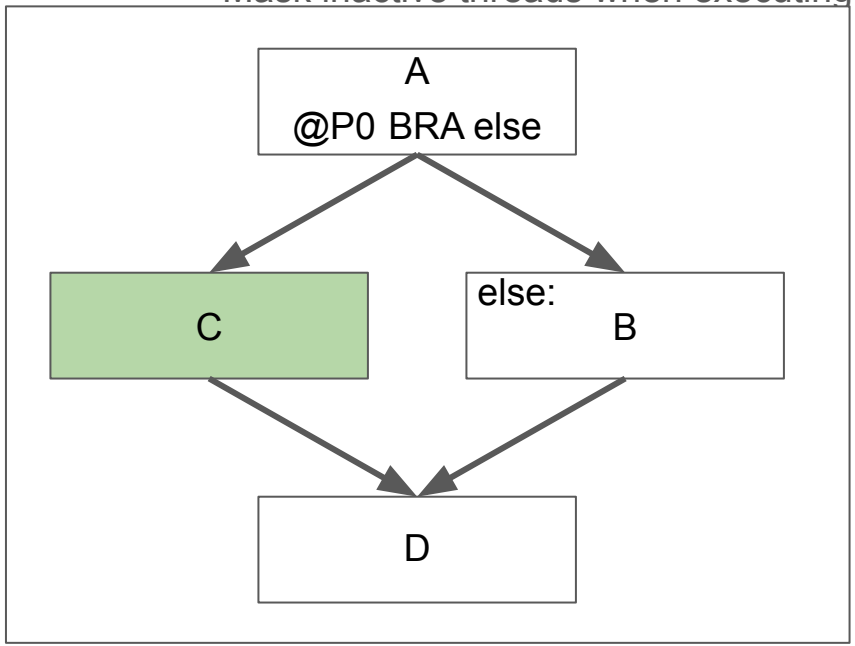
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

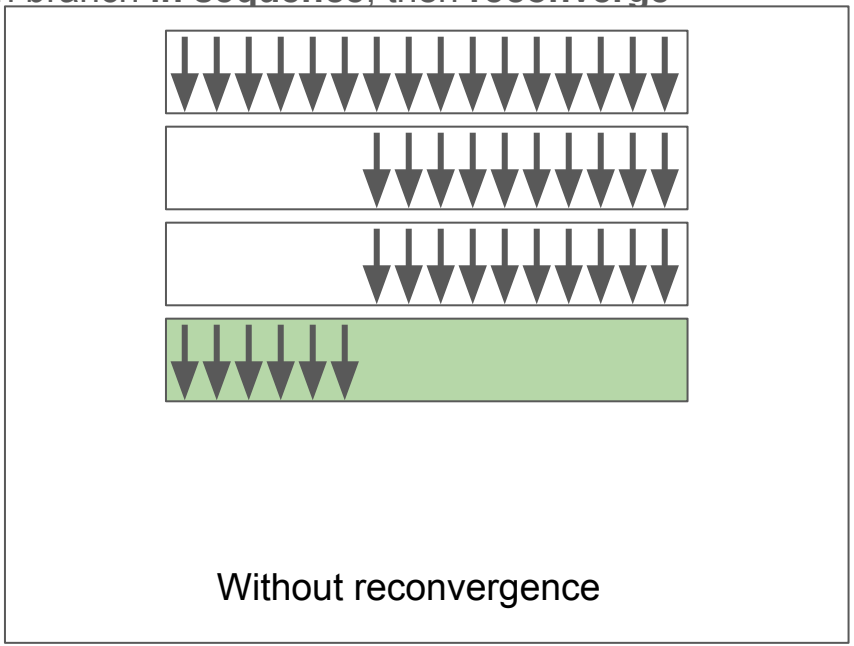




SIMT model and thread divergence

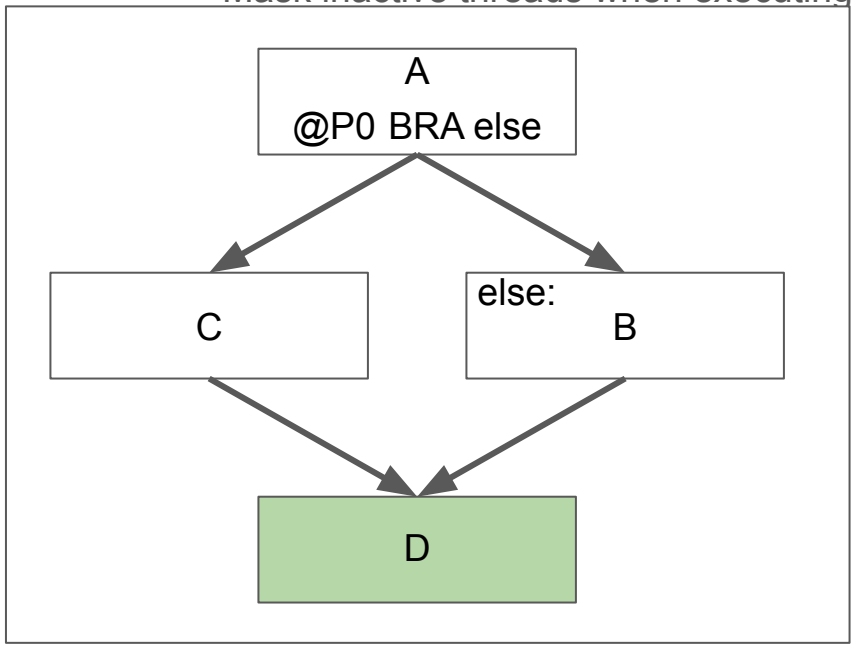
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

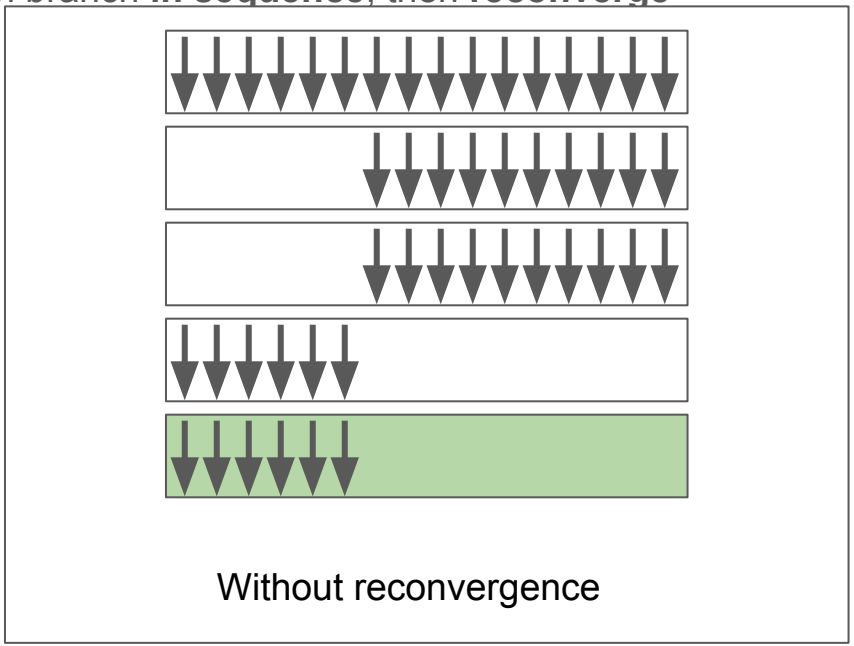




SIMT model and thread divergence

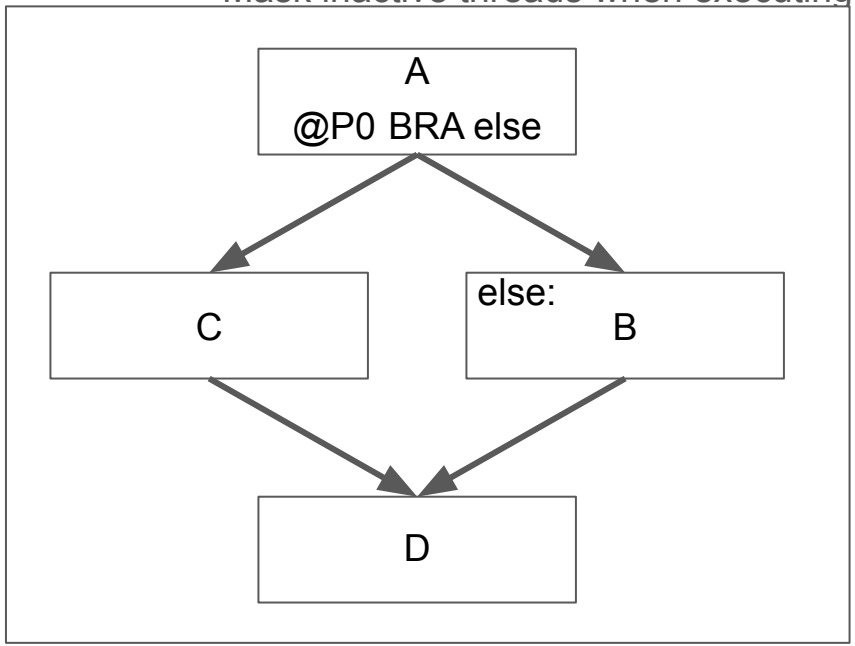
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

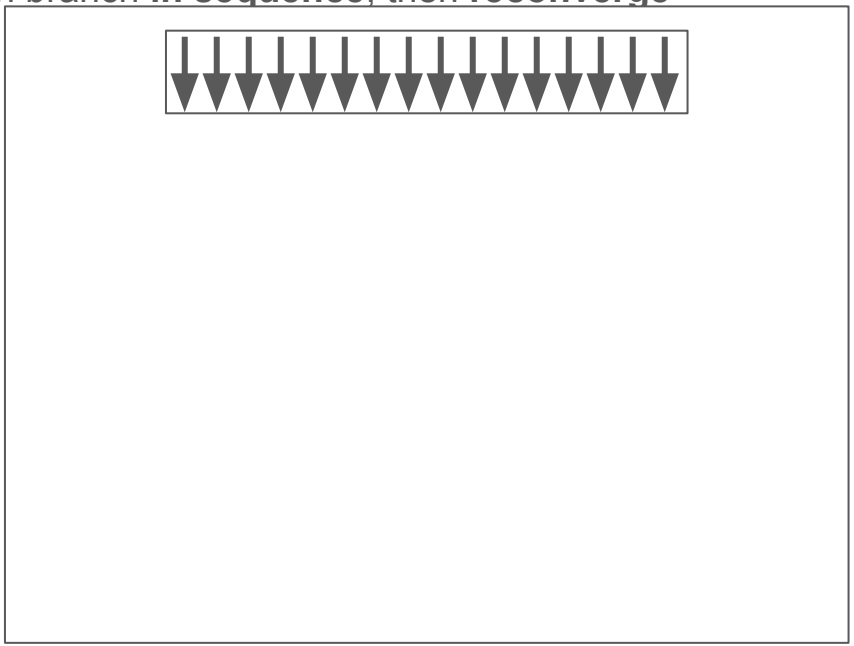




SIMT model and thread divergence

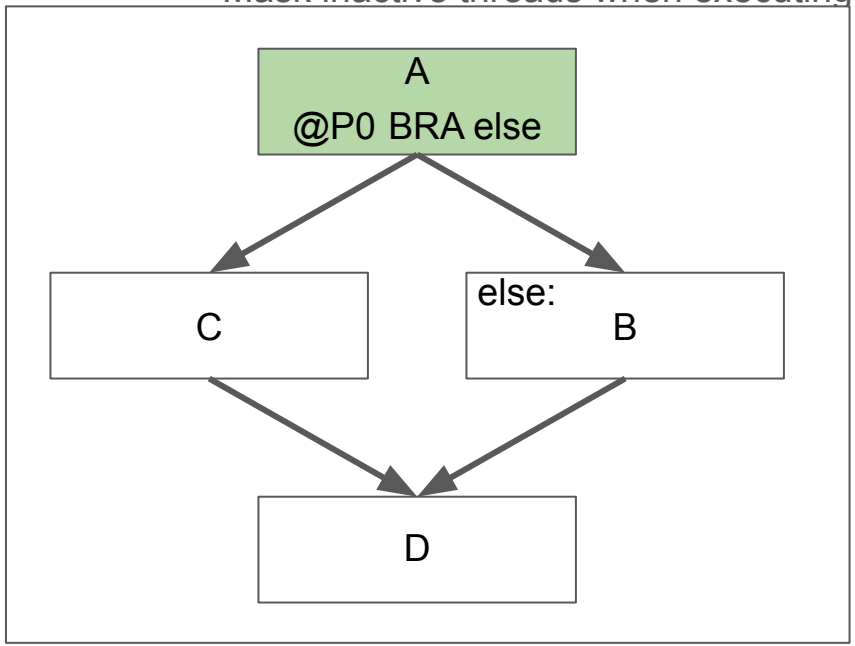
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

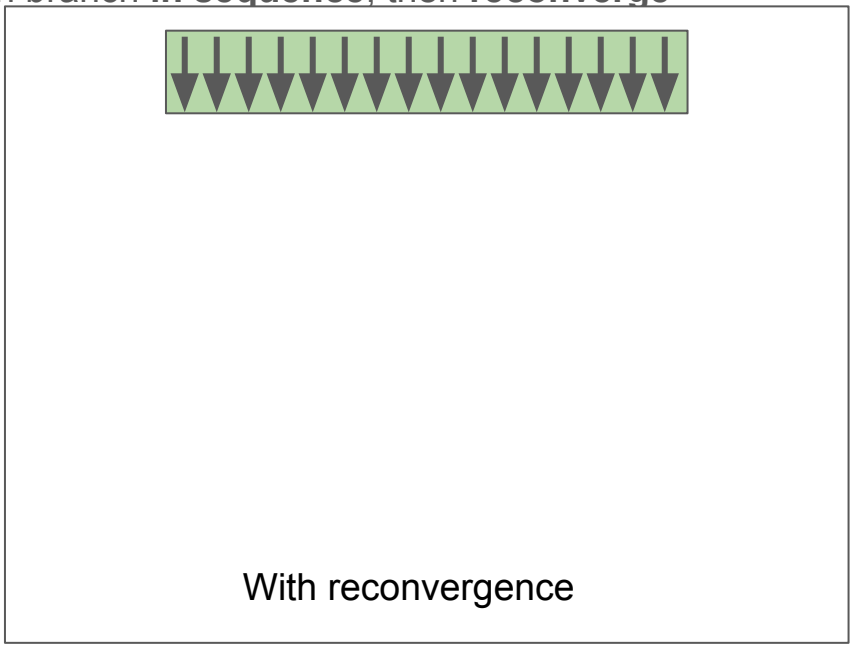




SIMT model and thread divergence

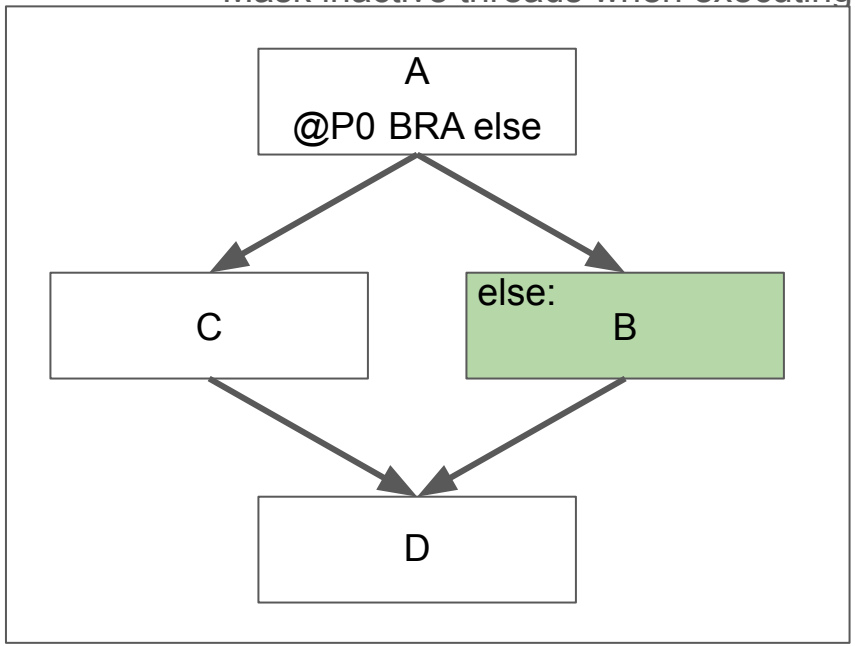
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

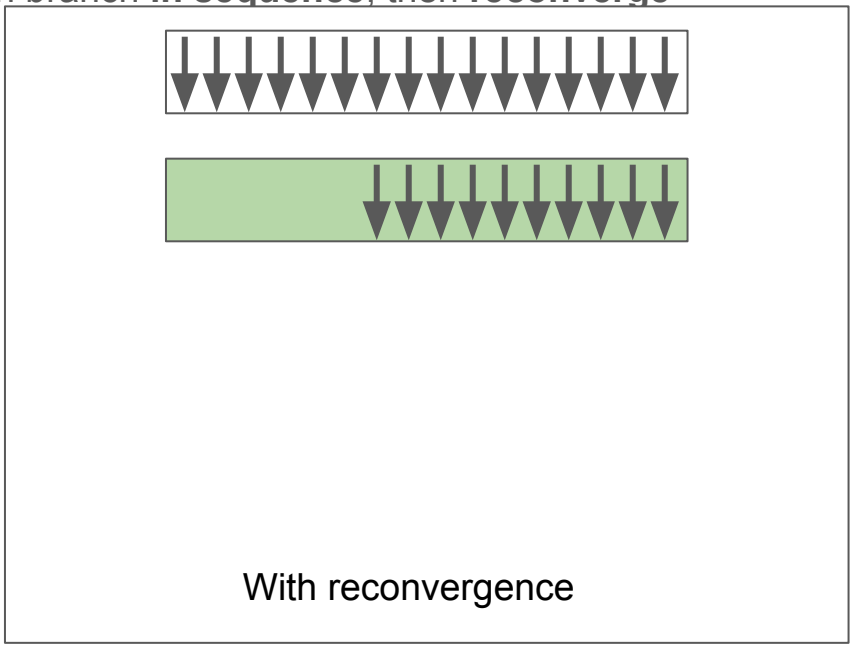




SIMT model and thread divergence

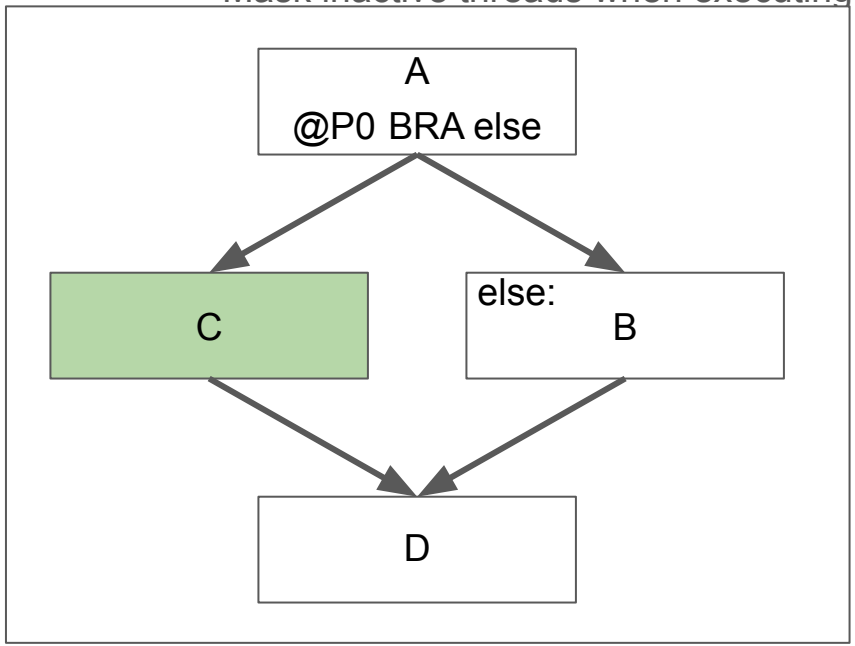
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

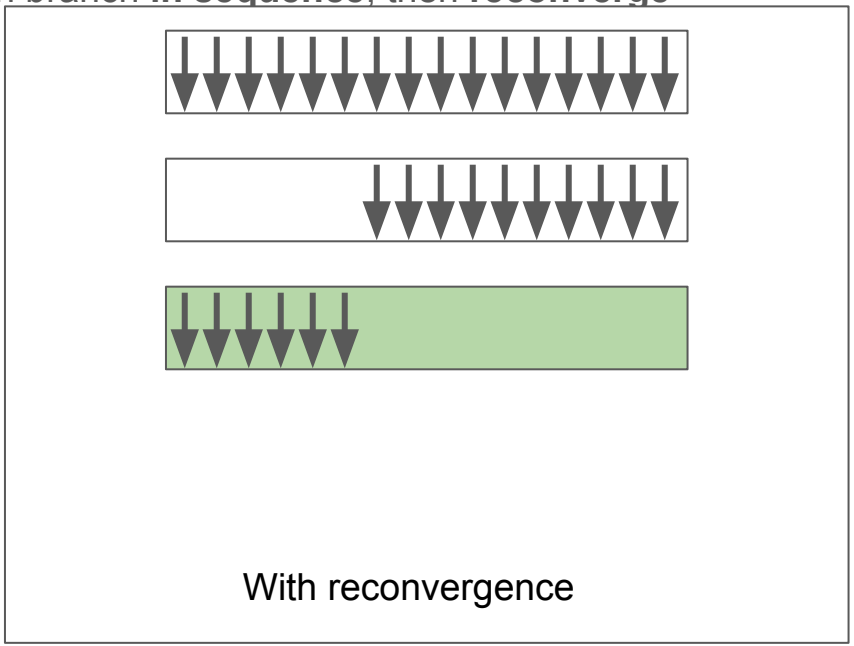




SIMT model and thread divergence

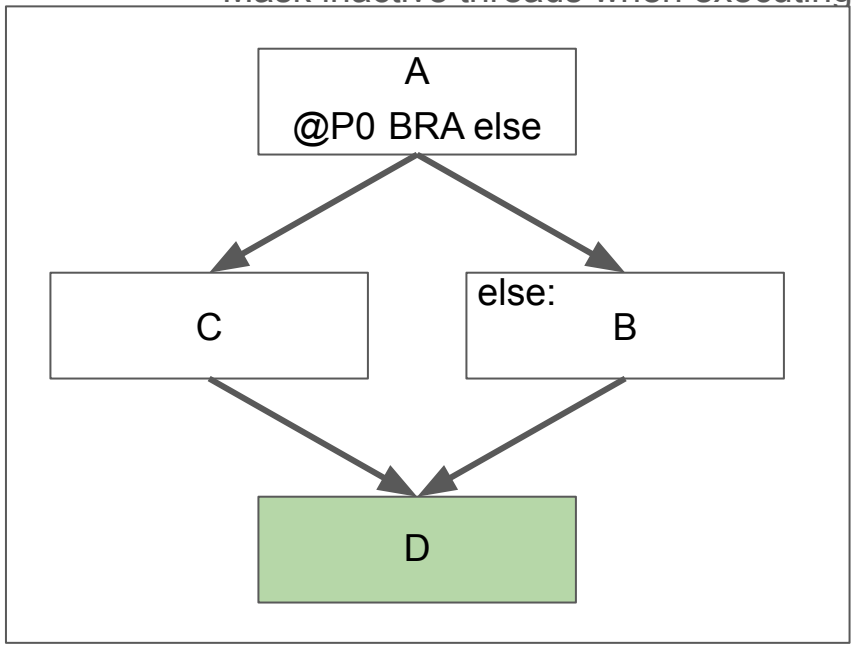
- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

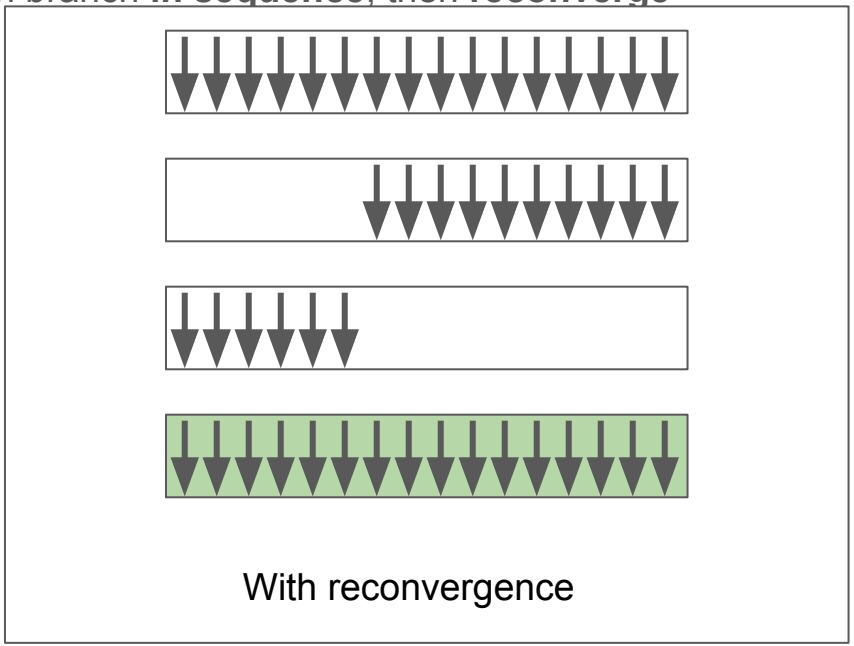




SIMT model and thread divergence

- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**

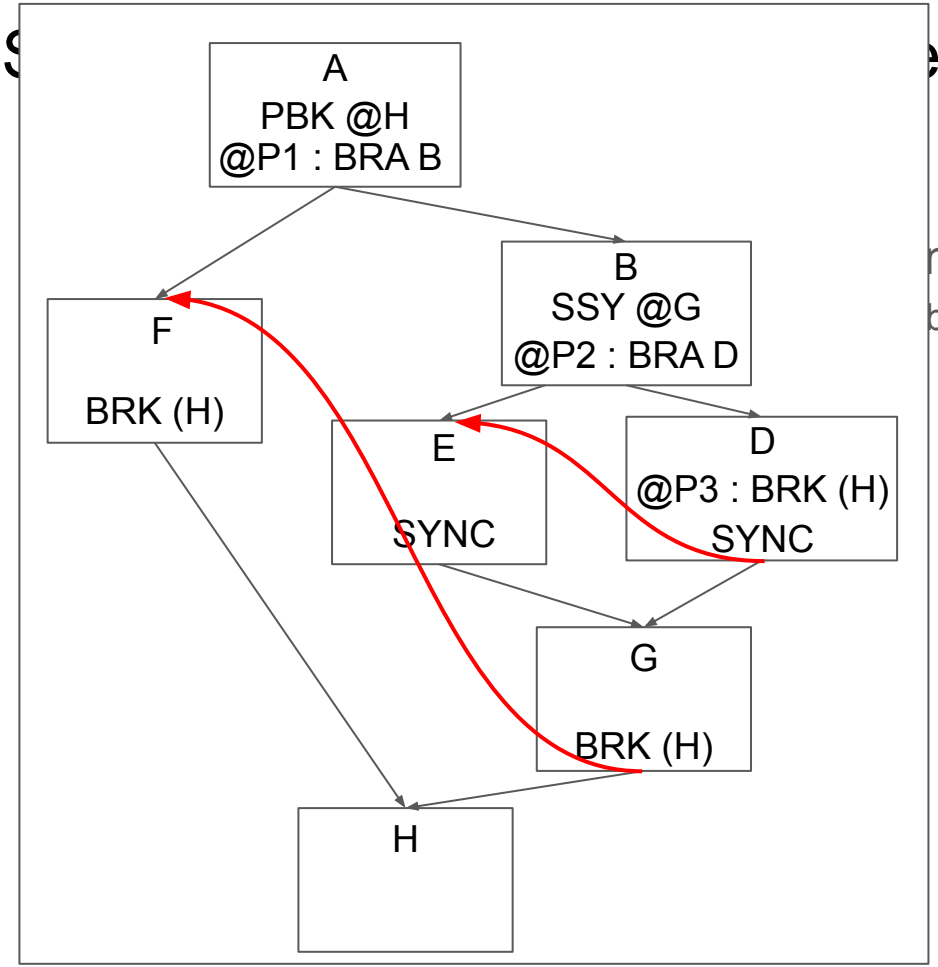




SIMT model and thread divergence

- Lockstep execution inside warps
- Conditional branch => divergence within warp
  - Mask inactive threads when executing each branch **in sequence**, then **reconverge**
  - Two kinds of divergence/convergence
    - SSY/SYNC, PBK/BRK
  - SIMT stack
    - Token based (NIL, SSY, PBK)
    - Software managed => compiler

S ence



n warp

branch **in sequence**, then **reconverge**

- Warp CFG requires **additional edges**
  - Using the SIMT semantics
  - Some may be avoided using static analyses
    - Warp1 and warp2 may not execute the same path, but all threads in each warp execute the same path

Low-level scheduling

- Still some unknowns
  - e.g. warp scheduler policy
- Complex, but can be modelled
  - Provided NVIDIA provides the implementation details
  - Hardest part: memory hierarchy and interference between warps
    - Cache contents
    - Interconnect/memory bank contentions
    - Same problems as multi-core CPUs, with larger scale

Conclusion

- Can we build an accurate model to predict the timing behavior of a program running on a GPU ?
  - Yes ! And we're working on it
    - **Timing behavior of a single warp**
      - **Static analyses to build accurate CFG**
      - **Architectural model**
    - **Compositional approach for multiple warps**
    - Same approach as for **multi-core CPUs**
    - Missing details => make hypotheses
  - What if some mechanisms are too dynamic ?
    - **Build our own predictable GPU**

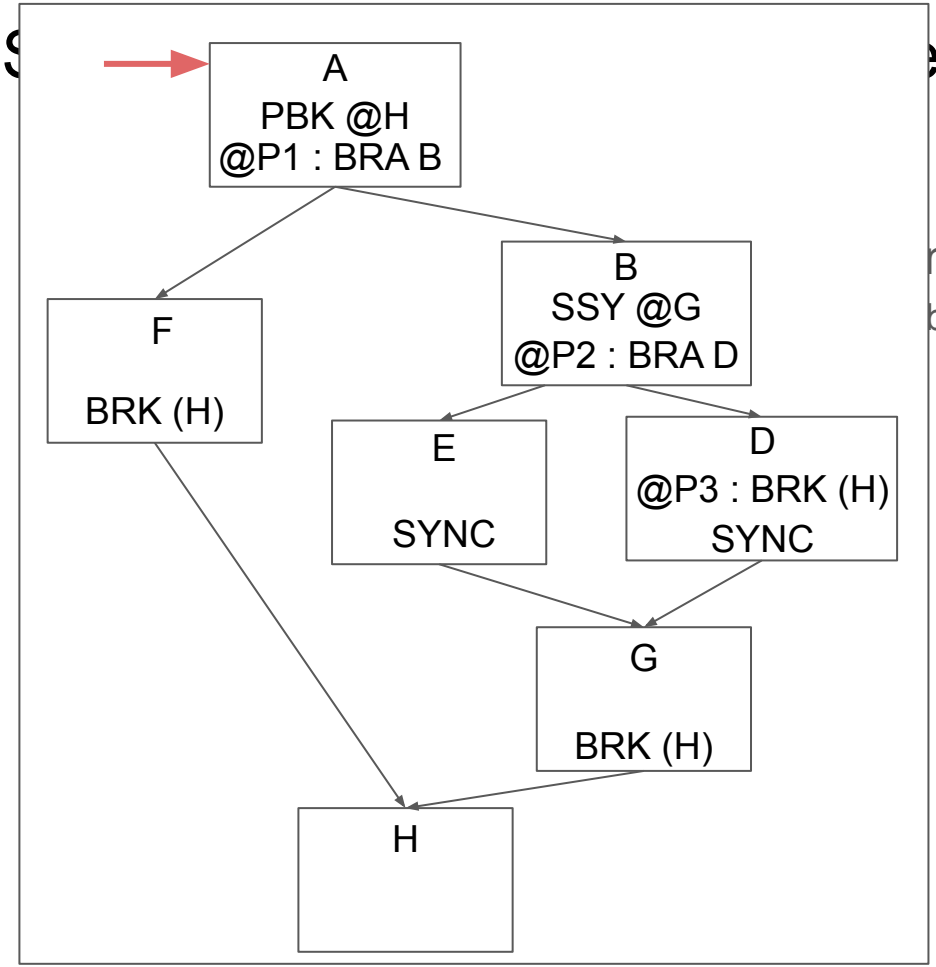
Thanks for your attention

CAPITAL Workshop: sCalable And PrecIse Timing AnaLysis for multicore platforms

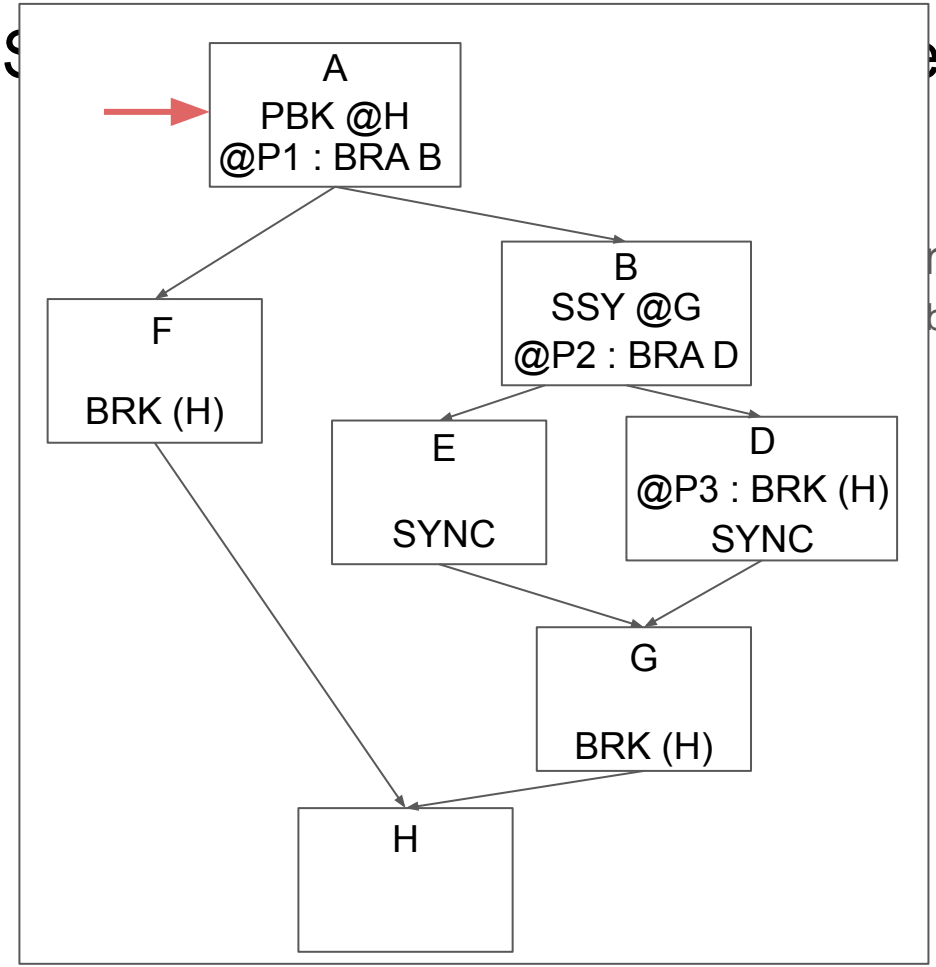
IRIT, Toulouse, **June 13th**

In-person or remote attendance

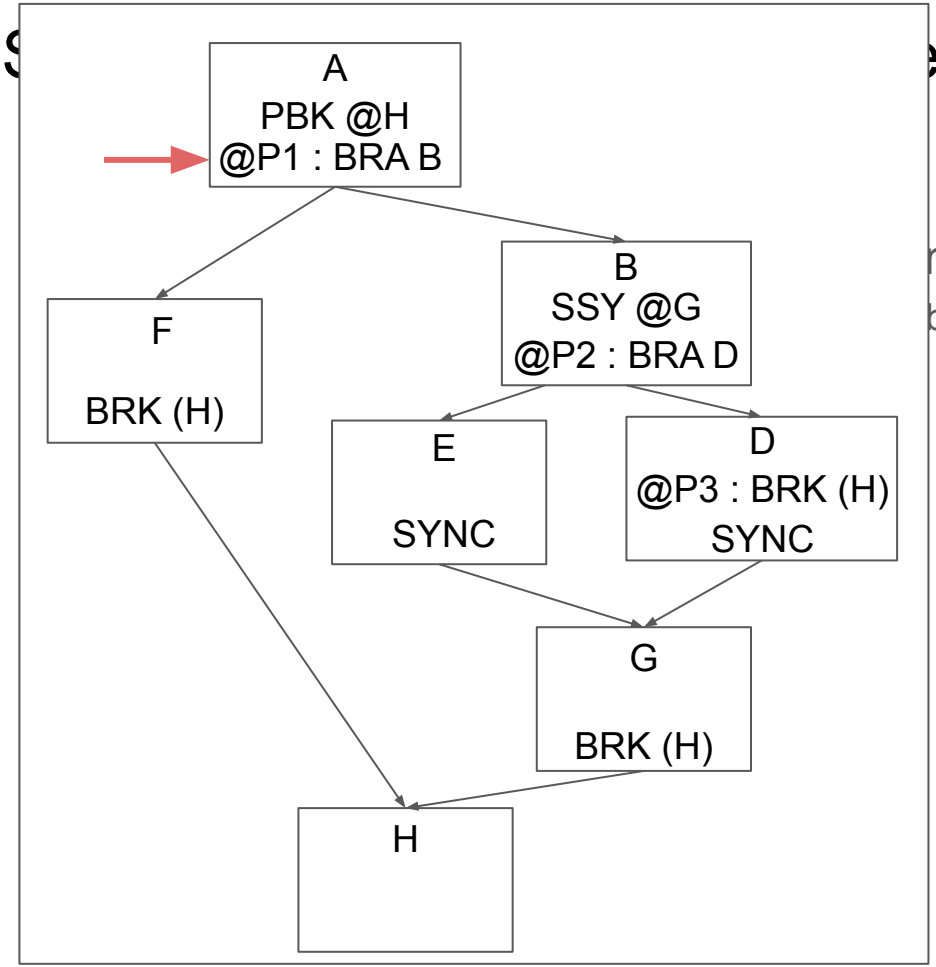
Free registration : **https://www.irit.fr/TRACES/site/capital-workshop-2023/**



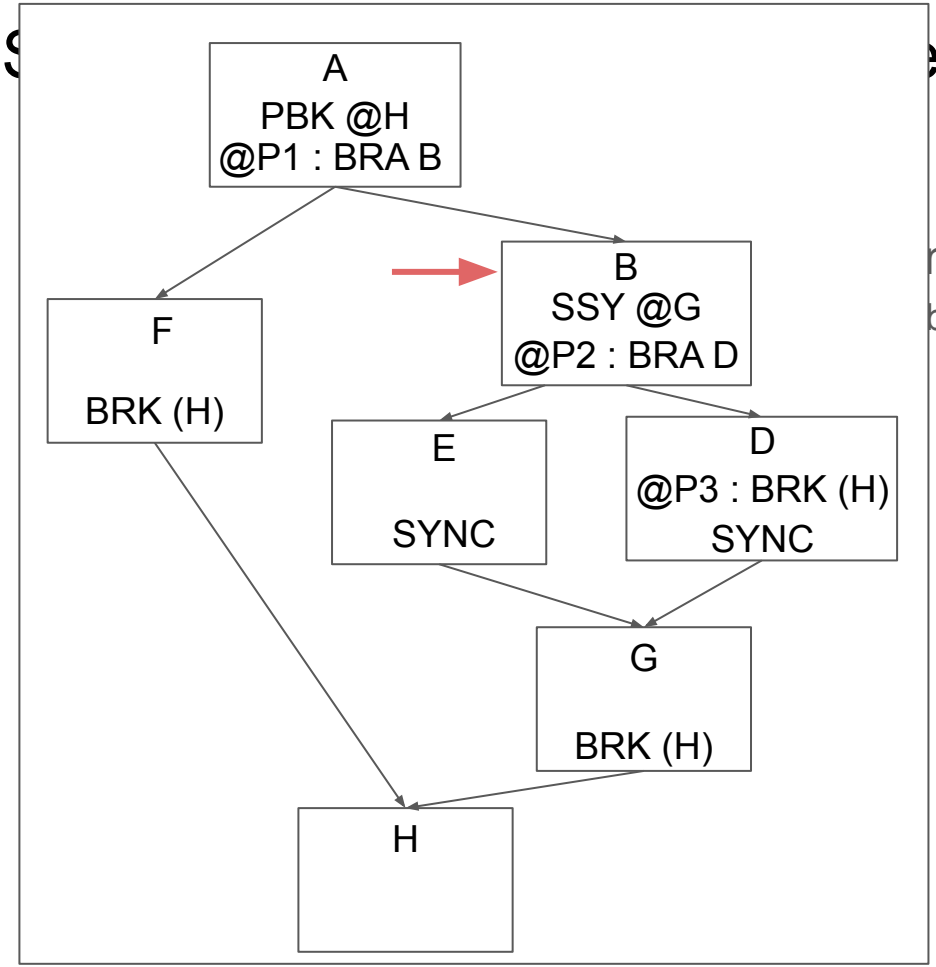
A | NIL | 0xFFFFFFFF



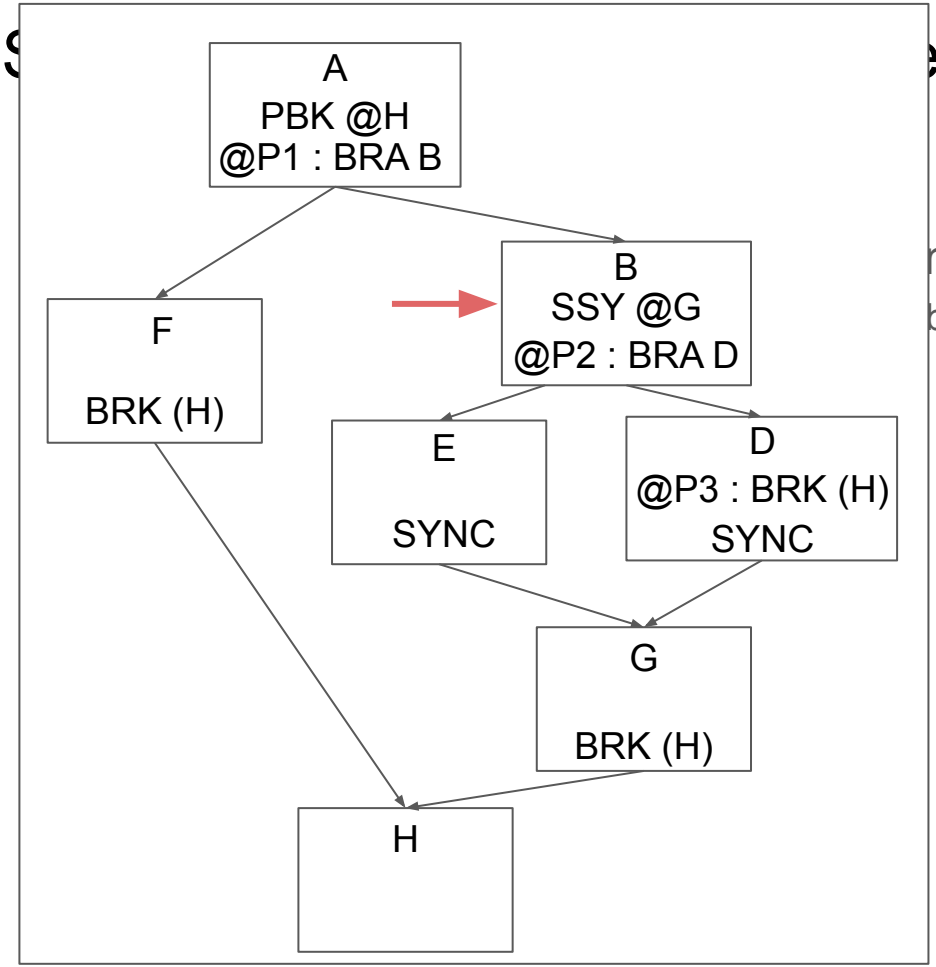
A	**NIL**	**0xFFFFFFFF**
H	**PBK**	0xFFFFFFFF



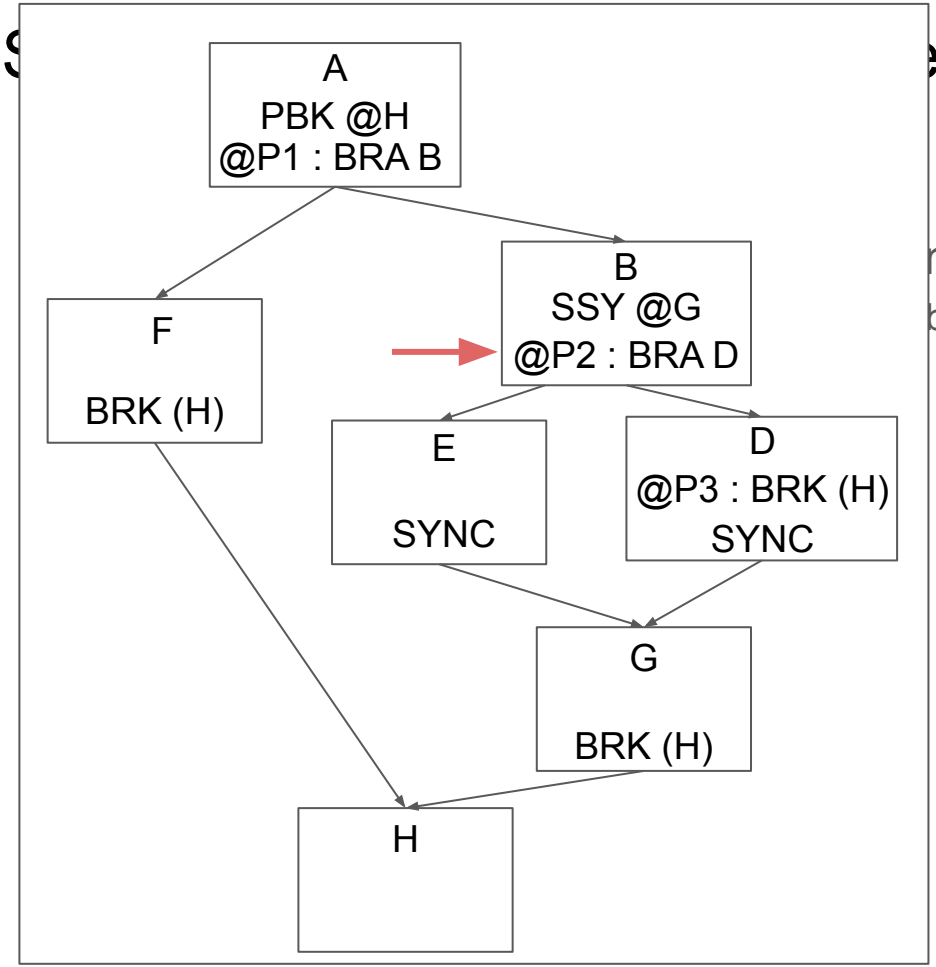
B	**NIL**	**0xFFFF0000**
F	NIL	0x**0000**FFFF
H	PBK	0xFFFFFFFF



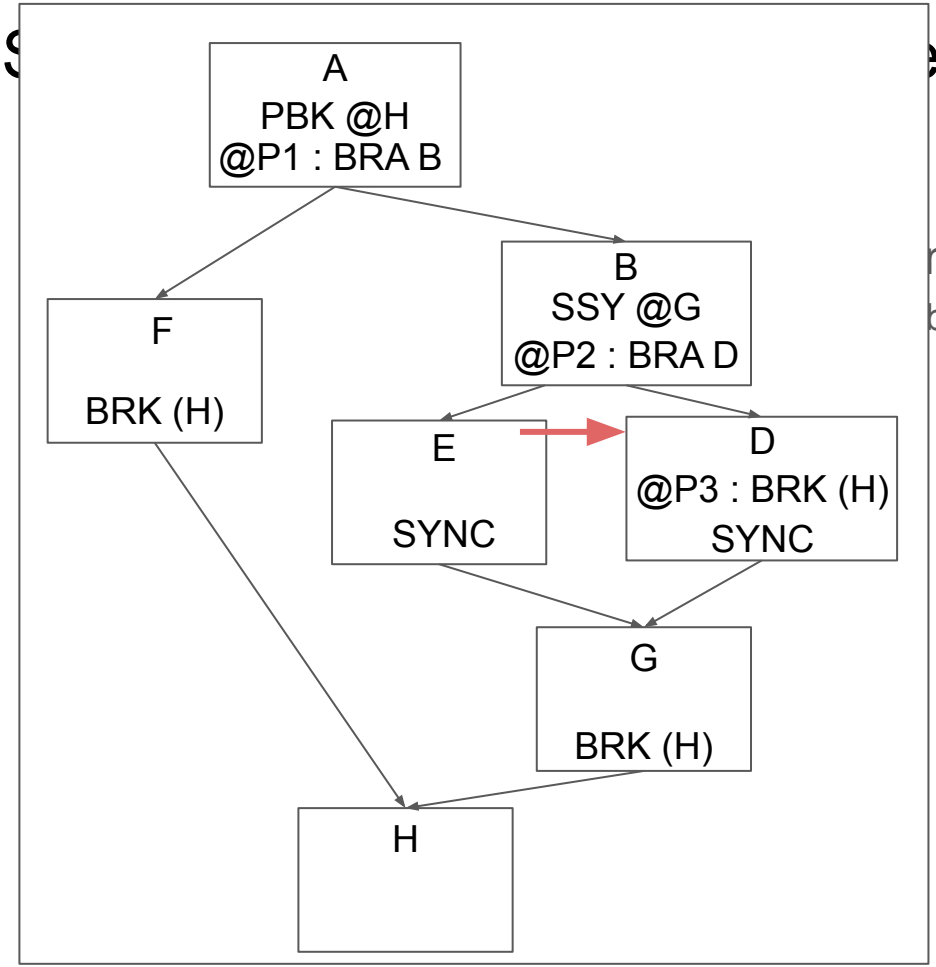
B	NIL	0xFFFF0000
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



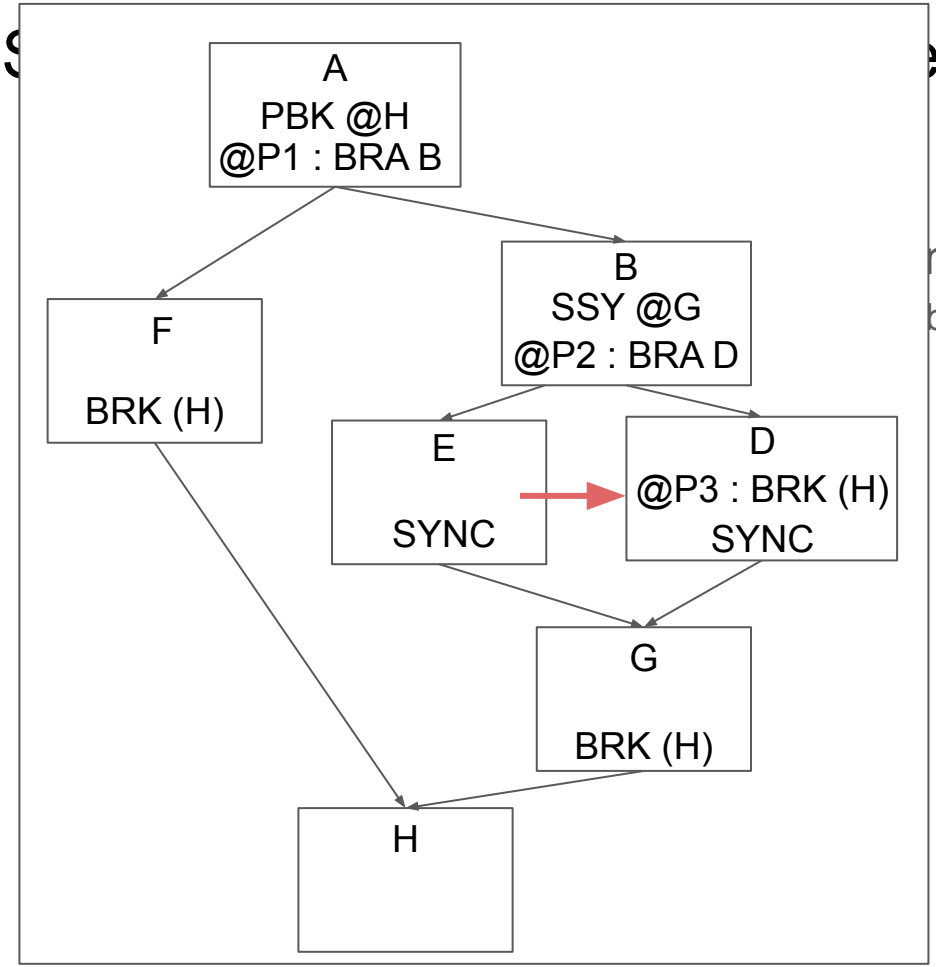
B	**NIL**	**0xFFFF0000**
G	**SSY**	0xFFFF0000
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



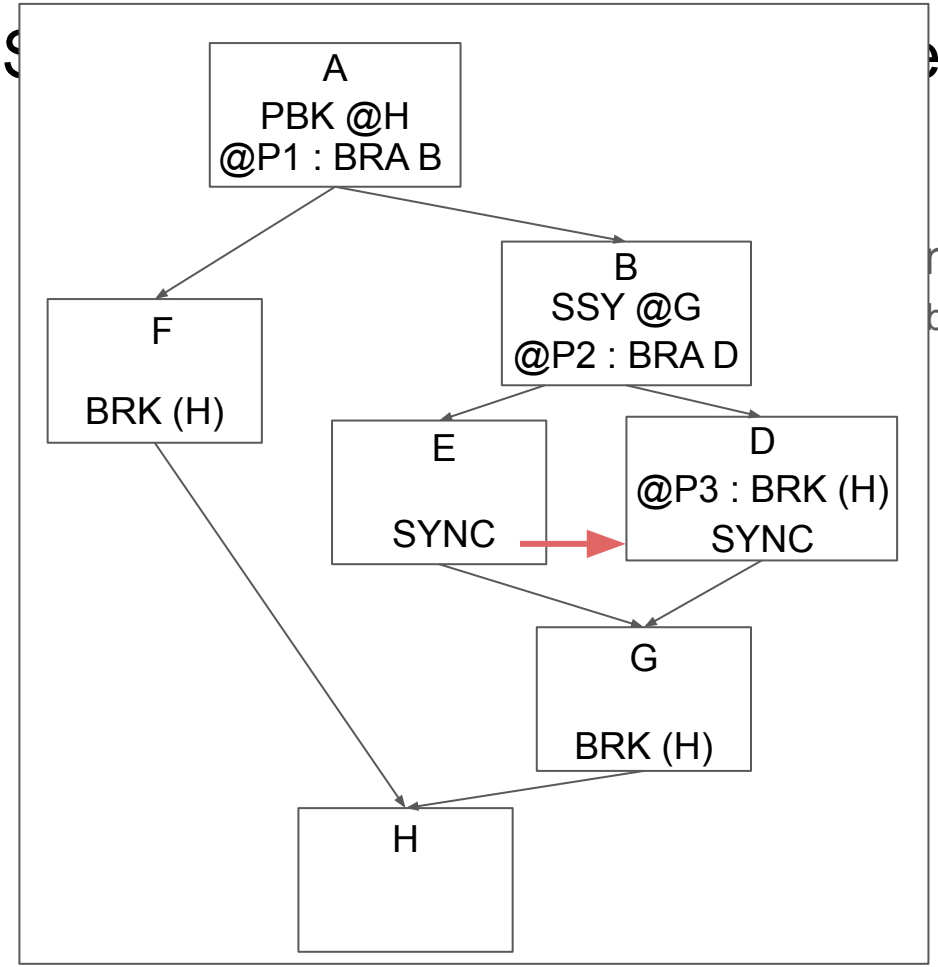
D	**NIL**	**0xFF000000**
E	NIL	0x**00**FF0000
G	SSY	0xFFFF0000
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



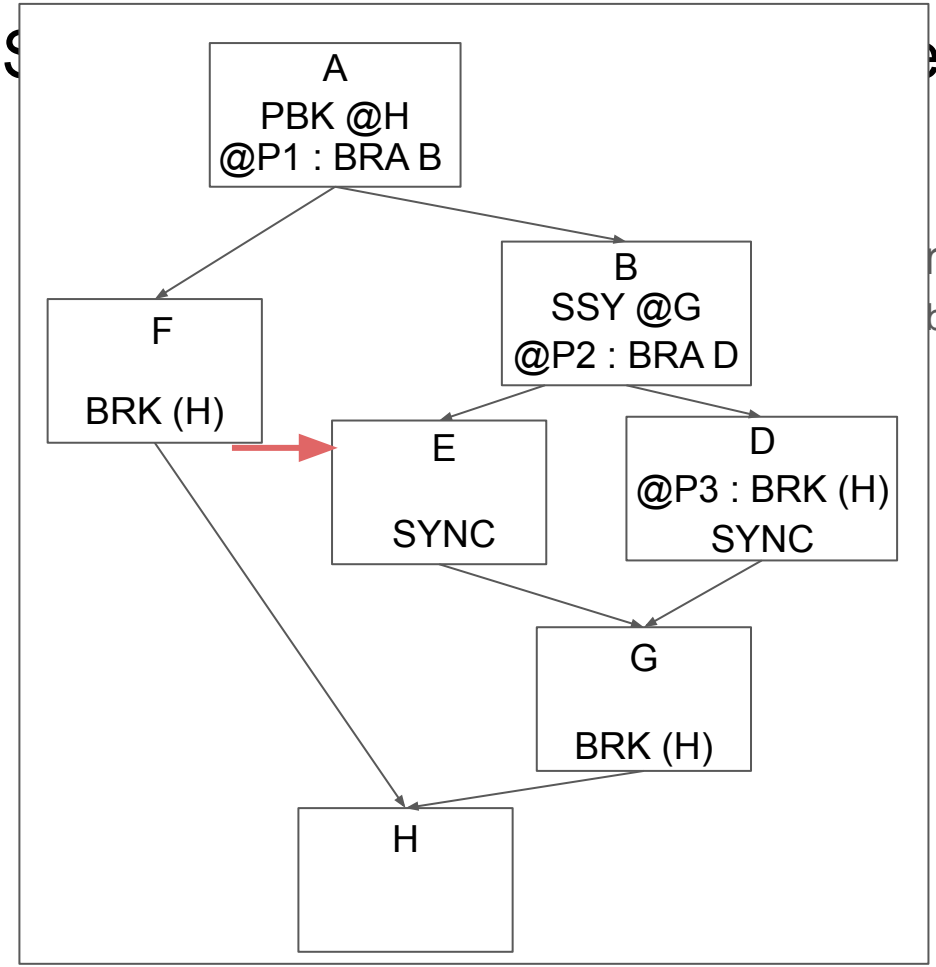
D	NIL	0xFF000000
E	NIL	0x00FF0000
G	SSY	0xFFFF0000
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



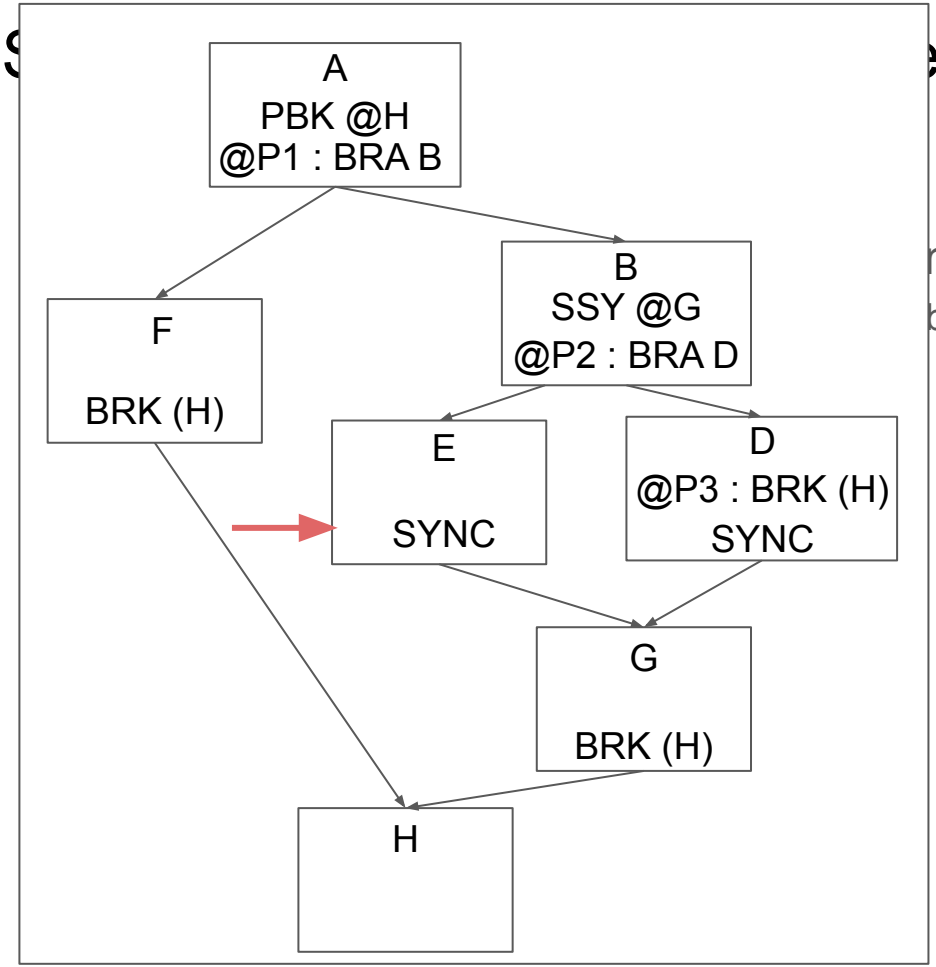
D	NIL	0xF**0**000000
E	NIL	0x00FF0000
G	SSY	0xF**0**FF0000
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



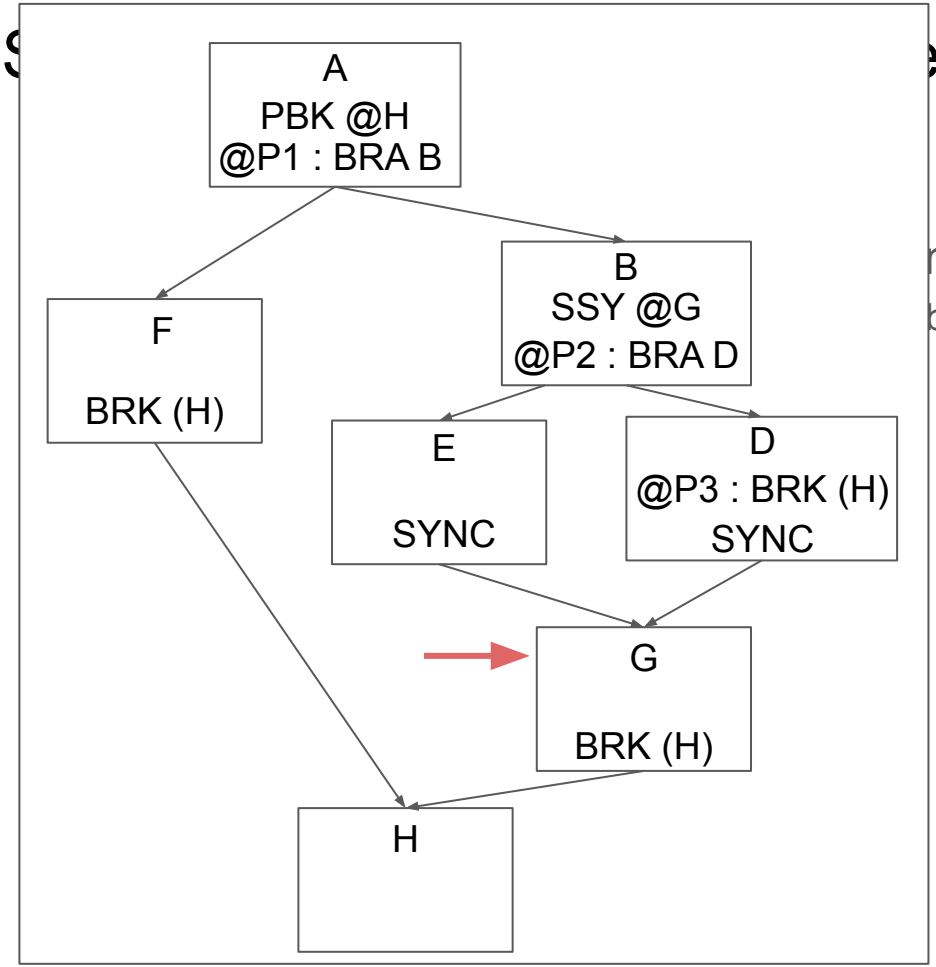
E	NIL	0x00FF0000
G	SSY	0xF0FF0000
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



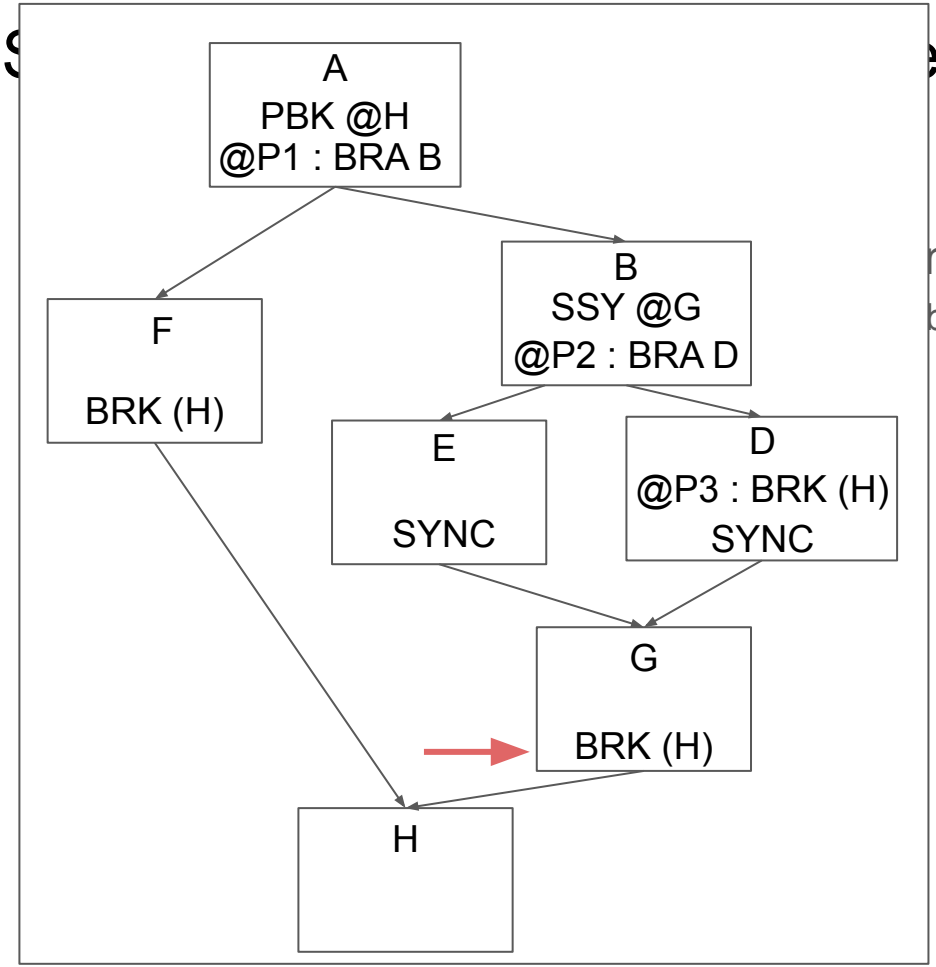
E	NIL	0x00FF0000
G	SSY	0xF0FF0000
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



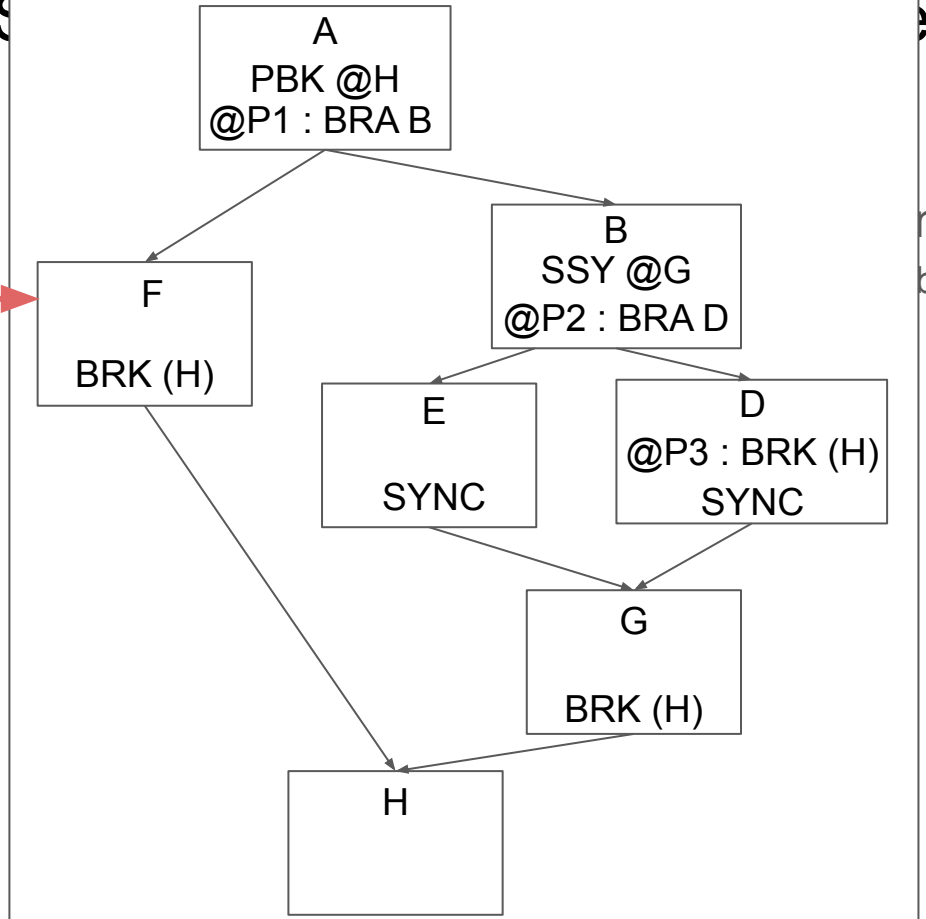
G	SSY	0xF0FF0000
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



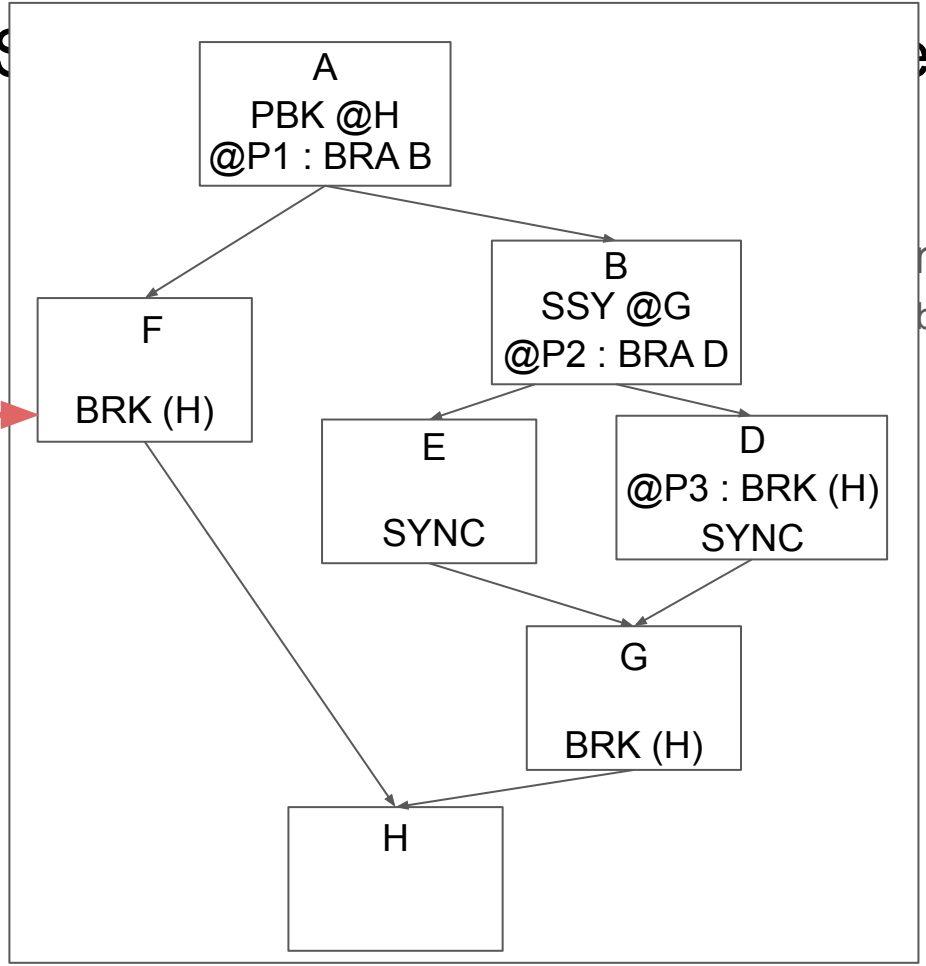
G	**NIL**	0xF0FF0000
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



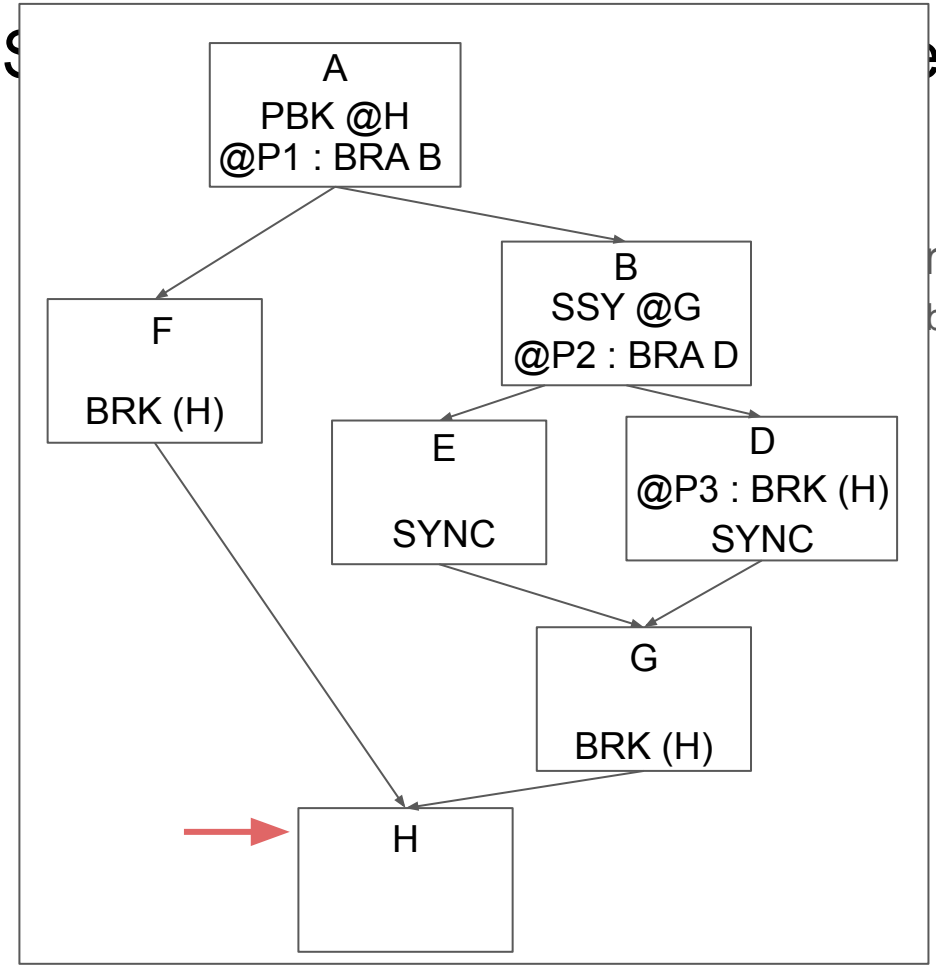
F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



F	NIL	0x0000FFFF
H	PBK	0xFFFFFFFF



H | PBK | 0xFFFFFFFF



H | **NIL** | 0xFFFFFFFF