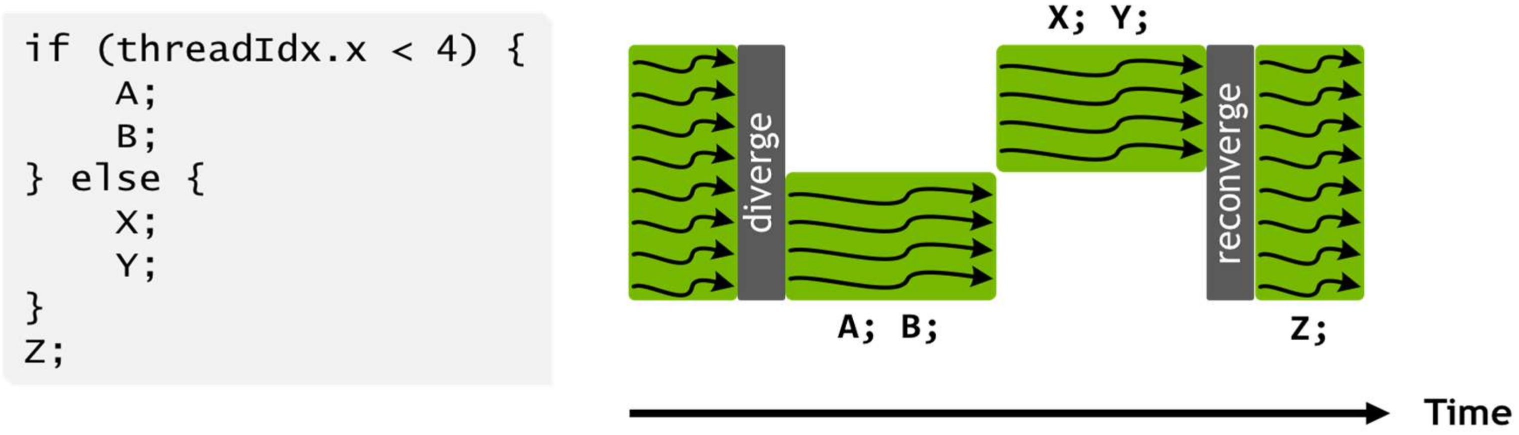
SPECULATIVE RECONVERGENCE FOR IMPROVED SIMT EFFICIENCY

Sana Damani, Daniel R. Johnson, Mark Stephenson, Stephen W. Keckler,
Eddie Yan, Michael McKeown and Olivier Giroux

CGO 2020

THREAD DIVERGENCE

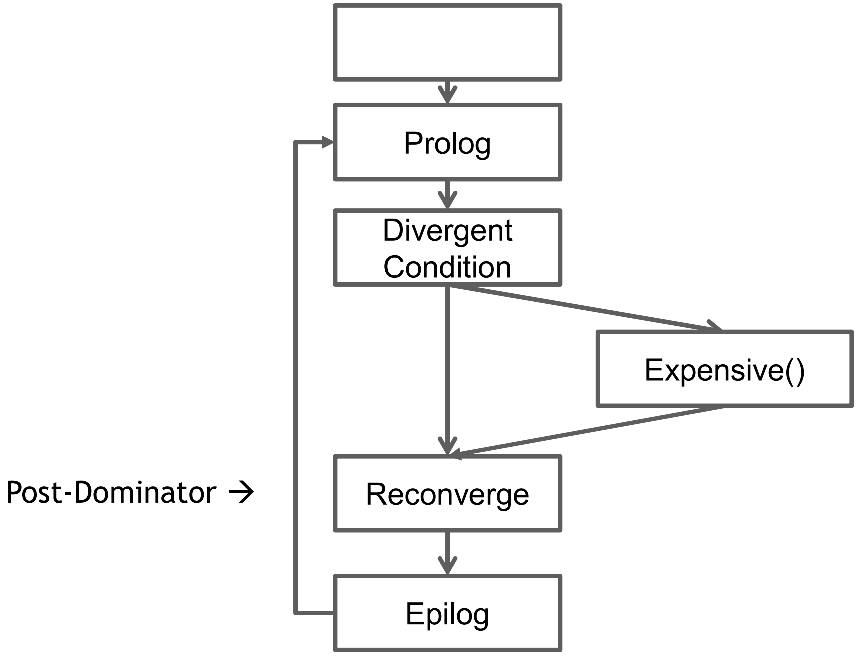
- **SIMT** Execution
- **Divergent Branch:** dependent on thread-local values
- **Solution:** If-conversion, Serialized execution
- Earliest **safe** reconvergence point: Branch **Post-Dominator**



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

Image credits: https://devblogs.nvidia.com/inside-volta

POST-DOMINATOR RECONVERGENCE



Prolog

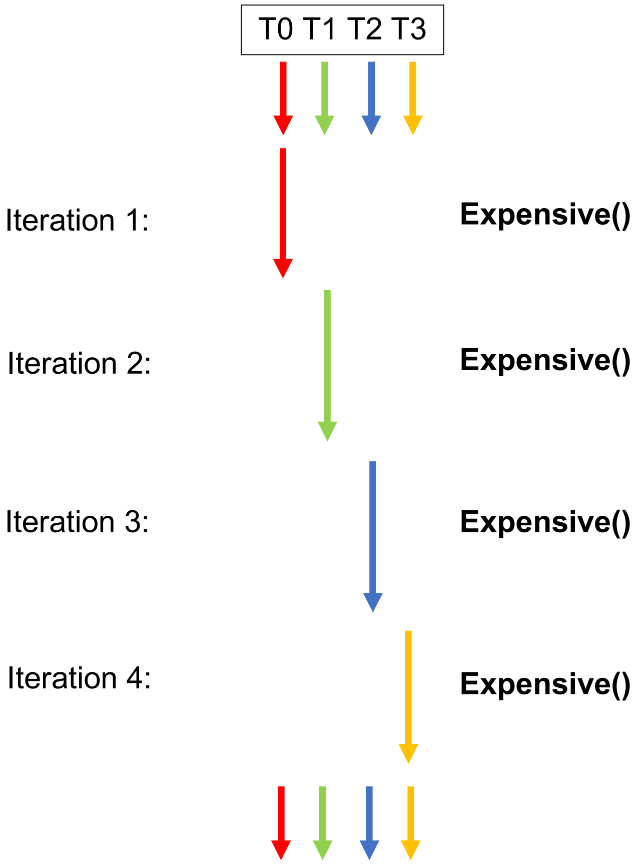
Divergent Condition

Expensive()

Post-Dominator →

Reconverge

Epilog



T0 T1 T2 T3

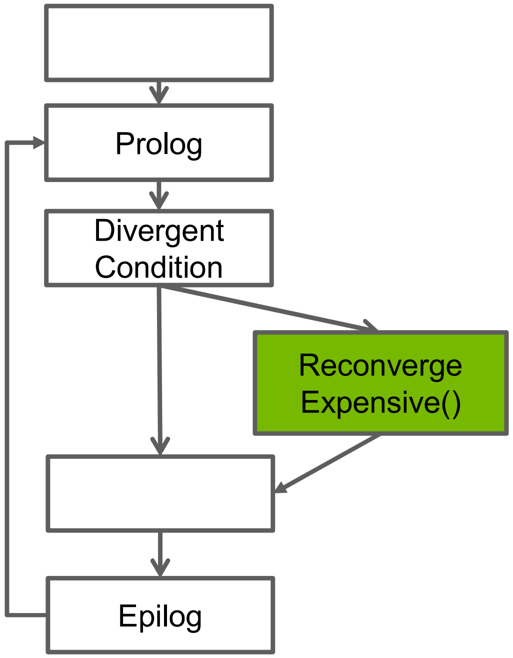
Iteration 1: **Expensive()**

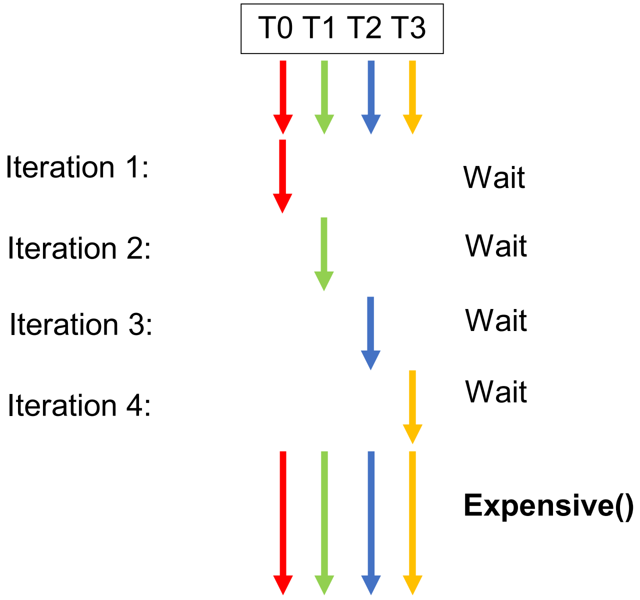
Iteration 2: **Expensive()**

Iteration 3: **Expensive()**

Iteration 4: **Expensive()**

ALTERNATIVE RECONVERGENCE





SPECULATIVE RECONVERGENCE

- **Observation:**
  - Diverged threads execute the same code serially instead of in parallel
  - Post-dominator reconvergence is not always ideal
- **Solution:**
  - Reconverge threads before executing common code
- **Goal:**
  - Increase convergence within expensive code paths

CODE PATTERNS

1. **Divergent condition inside loop**

```
while (loop condition) {
        if (divergent condition) {
                Expensive()         // converge here
        }
}
```

2. **Nested loop with divergent loop trip count**

```
while (loop condition) {
        while (divergent condition) {
                Expensive()          // converge here
        }
}
```

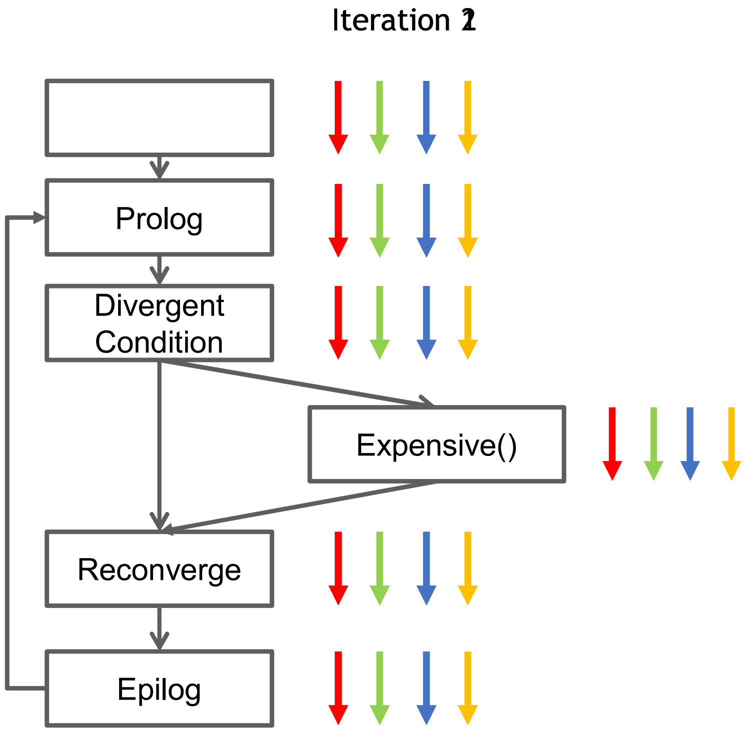
3. **Divergently executed common function call**

```
if (divergent condition) {
        foo()   // converge within common function
}
else {
        foo()   // converge within common function
}
```

DIVERGENT CONDITION INSIDE LOOP

Post-Dominator Reconvergence

```
while (loop condition) {
            if (divergent condition) {
                        Expensive()
            }
}
```

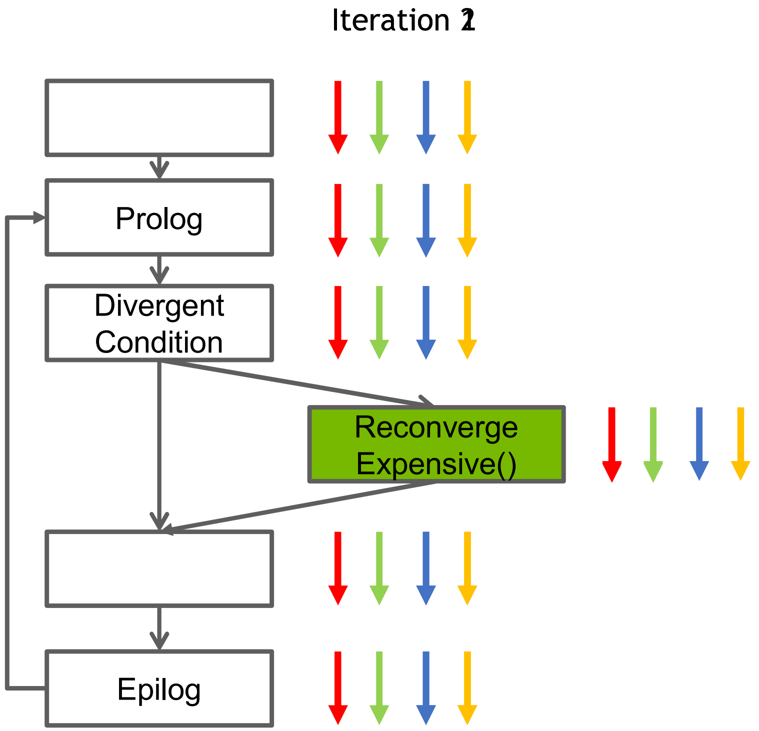


Copyright NVIDIA 2020

DIVERGENT CONDITION INSIDE LOOP

Speculative Reconvergence

```
while (loop condition) {
        if (divergent condition) {
                Expensive()
        }
}
```

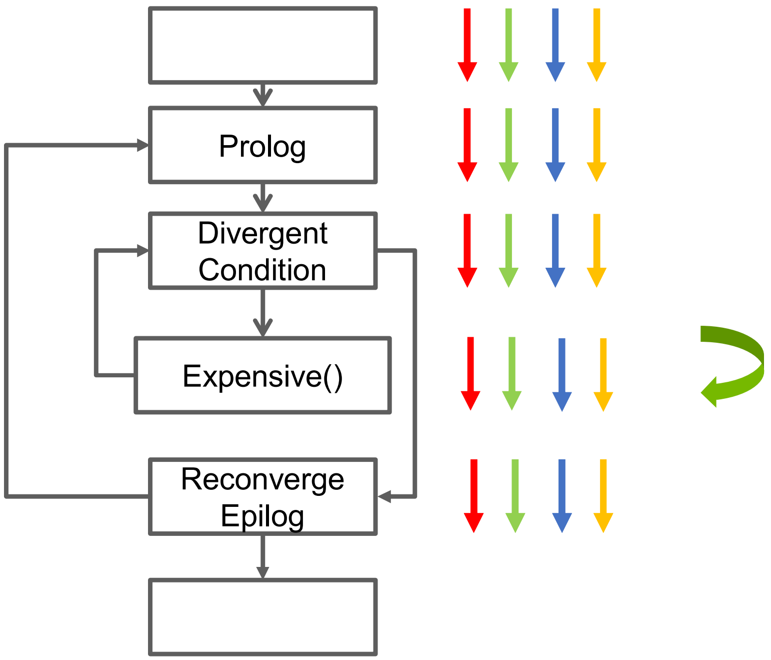


DIVERGENT LOOP TRIP CONDITION

Post-Dominator Reconvergence

```
while (loop condition) {
            while (divergent condition) {
                        Expensive()
            }
}
```

Iteration 1: Outer Loop

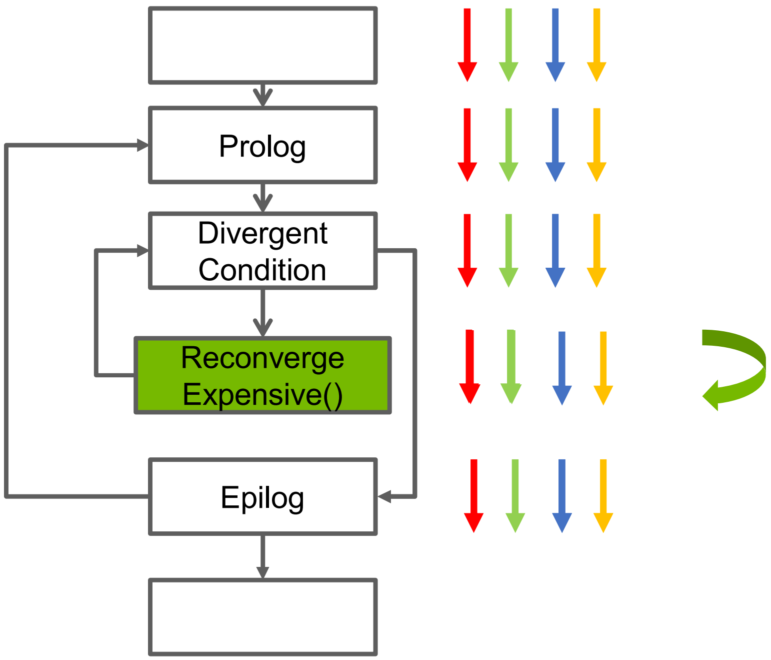


DIVERGENT LOOP TRIP CONDITION

Speculative Reconvergence

```
while (loop condition) {
          while (divergent condition) {
                    Expensive()
          }
}
```

Iteration 2: Outer Loop

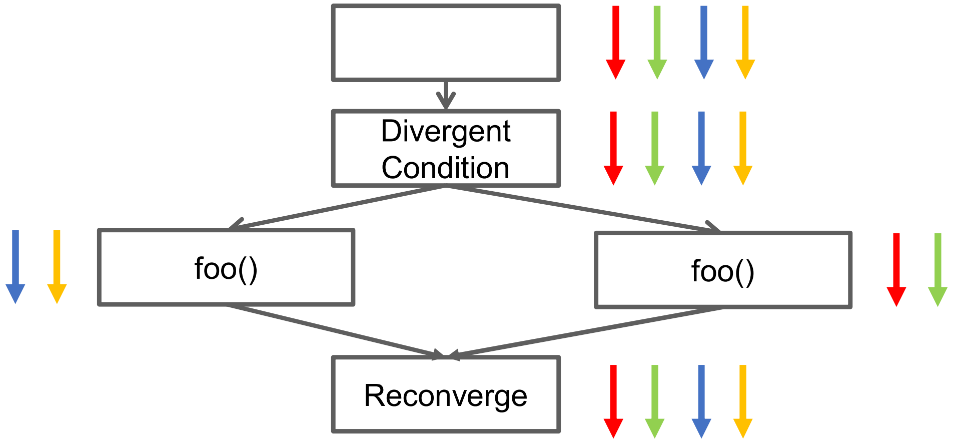


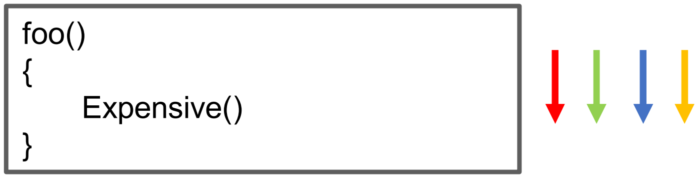
Copyright NVIDIA 2020

DIVERGENT COMMON FUNCTION CALL

Post-Dominator Reconvergence

```
if (divergent condition) {
            foo()
} else {
            foo()
}
```



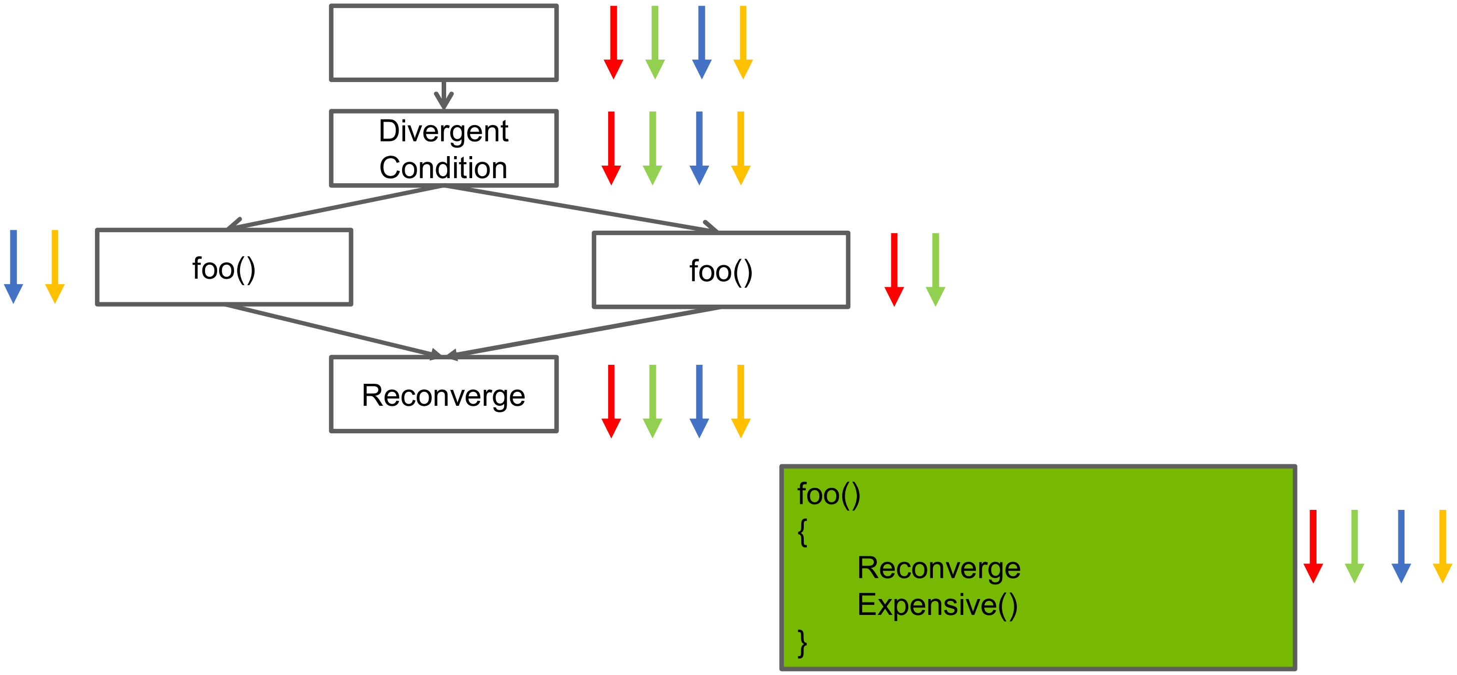


```
foo()
{
      Expensive()
}
```

DIVERGENT COMMON FUNCTION CALL

Speculative Reconvergence

```
if (divergent condition) {
            foo()
} else {
            foo()
}
```



```
foo()
{
    Reconverge
    Expensive()
}
```

DESIGN

Identify opportunity for speculative reconvergence

- User-directed
- Compiler-identified

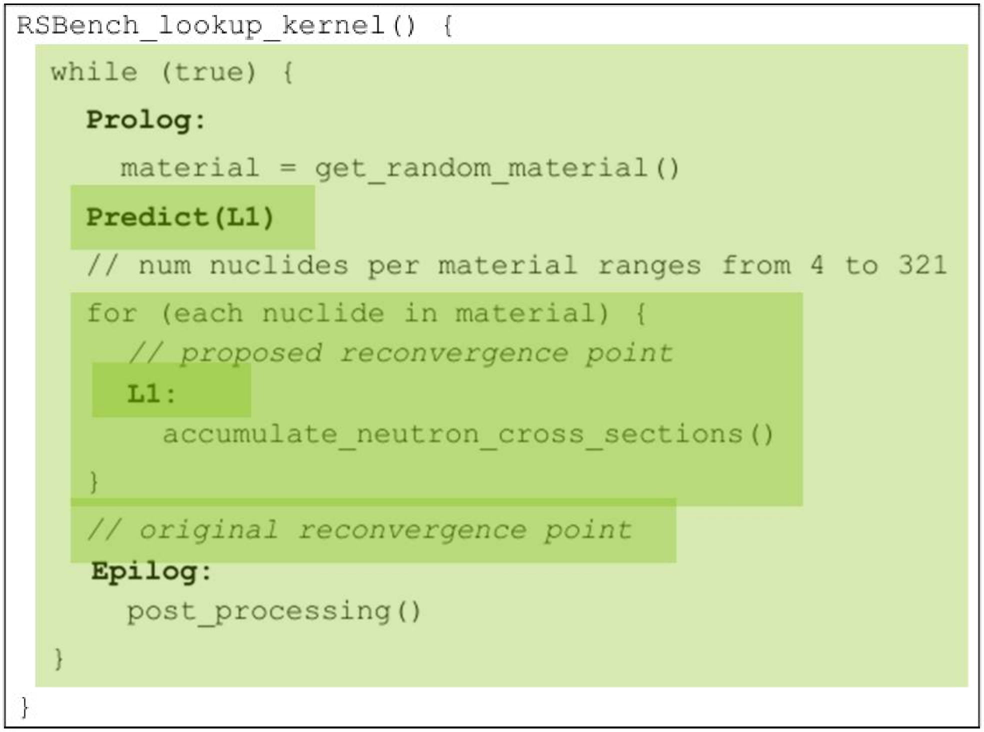
Compiler inserted reconvergence barriers

- Performance optimization (Forward progress guarantee)
- Forward and backward dataflow analysis to identify insertion points

IDENTIFY SPECULATIVE RECONVERGENCE POINT

Case Study: RSBench

- Monte Carlo neutron transport
- **Characteristics:**
  - Nested divergent loop
  - Expensive inner loop
  - Inexpensive prolog/epilog
  - High divergence



```
RSBench_lookup_kernel() {
  while (true) {
    Prolog:
      material = get_random_material()
    Predict(L1)
    // num nuclides per material ranges from 4 to 321
    for (each nuclide in material) {
      // proposed reconvergence point
      L1:
        accumulate_neutron_cross_sections()
    }
    // original reconvergence point
    Epilog:
      post_processing()
  }
}
```

TRANSFORM OBJECTIVES

Synchronization Primitives

- Ensure all participating threads **join** the reconvergence barrier at **region start** (JoinBarrier)
- Threads **wait** for participating threads at the **new reconvergence point** (WaitBarrier)
- **Exiting** threads must **cancel** out of the reconvergence barrier (CancelBarrier)
- Threads that **re-enter** the region must **rejoin** the reconvergence barrier (RejoinBarrier)

INSERTING SYNCHRONIZATION PRIMITIVES

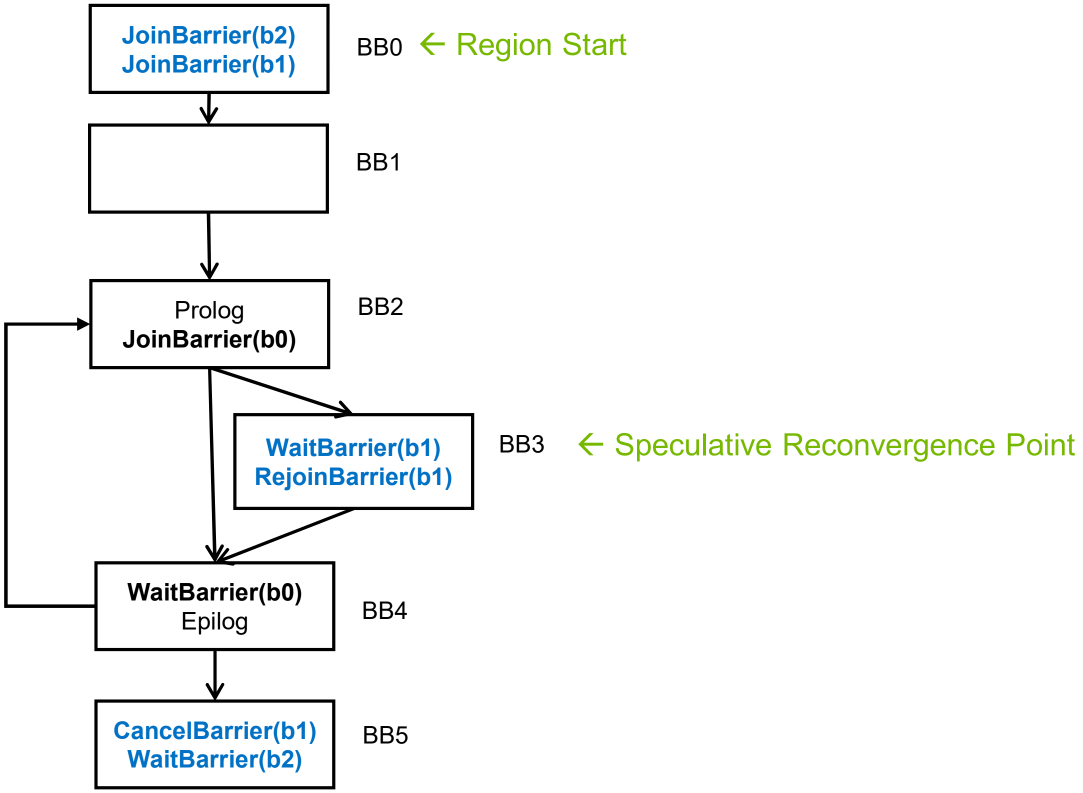
Participating threads register [illegible] barrier at region start

Exiting threads must cancel out of the reconvergence barrier

Threads wait for participating threads at the new reconvergence point

Ensure reconvergence at the region end

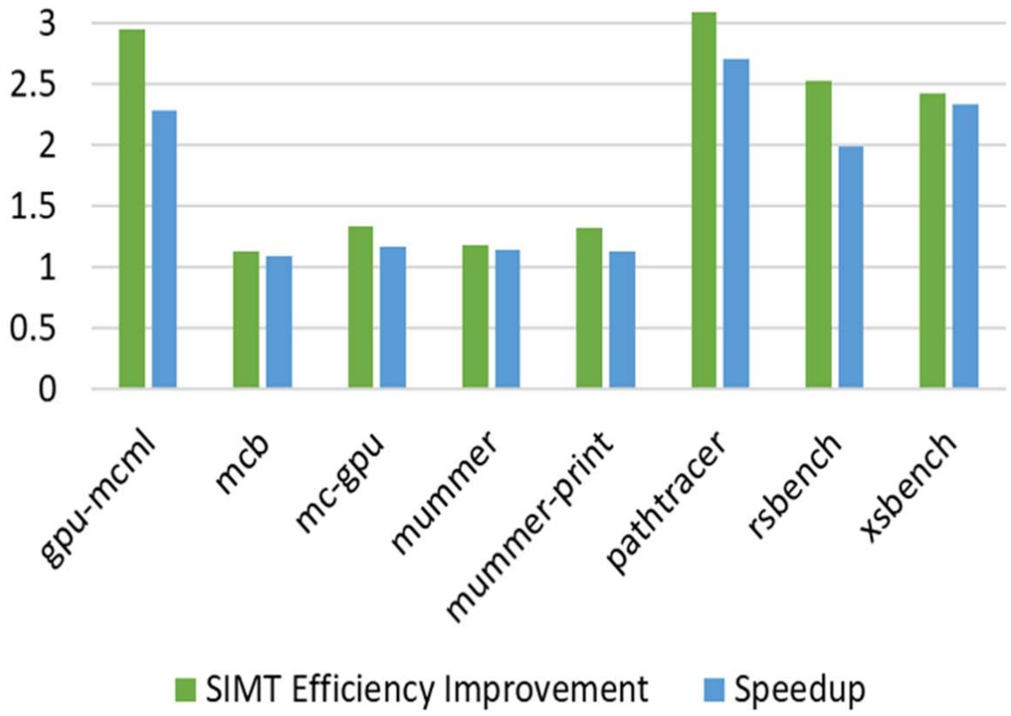
Last [illegible] must rejoin the barrier



EXPERIMENTAL SETUP

- **Hardware**: Volta GPU (V100)
- **Implementation:**
    - Production GPU compiler
    - User-directed and automatically detected opportunities
- **Benchmarks**: Mini-apps and internal benchmarks (Monte-carlo, ray tracing)
- **Metrics**: SIMT efficiency (avg threads active per issued instruction), Speedup

USER-DIRECTED SPECULATIVE RECONVERGENCE



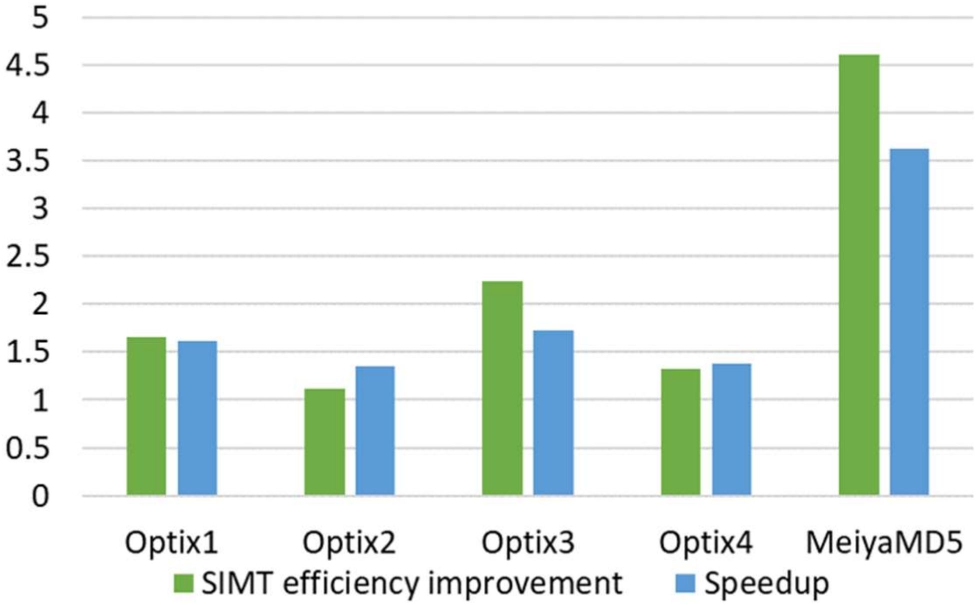
Apps: Monte Carlo, Path tracing

SIMT efficiency improvement: 1.2x to 3.4x

Speedup: 1.1x to 2.5x

Automatic reconvergence performed identically for these apps

AUTOMATIC SPECULATIVE RECONVERGENCE



Apps: Optix, MayaMD5

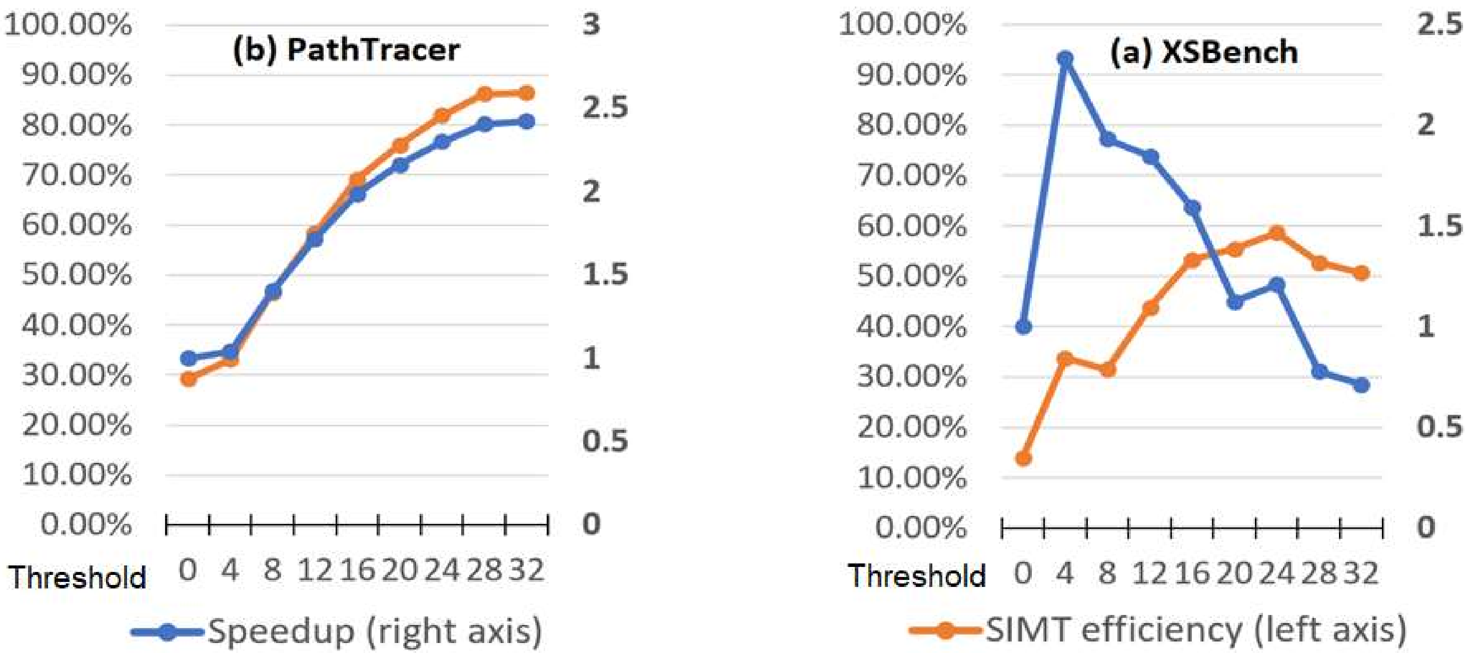
SIMT efficiency improvement: 1.2x to 4.7x

Speedup: 1.4x to 3.75x

SOFT BARRIER

Threshold Selection

Goal: Wait for *enough* threads to arrive at reconvergence point instead of *all* threads.

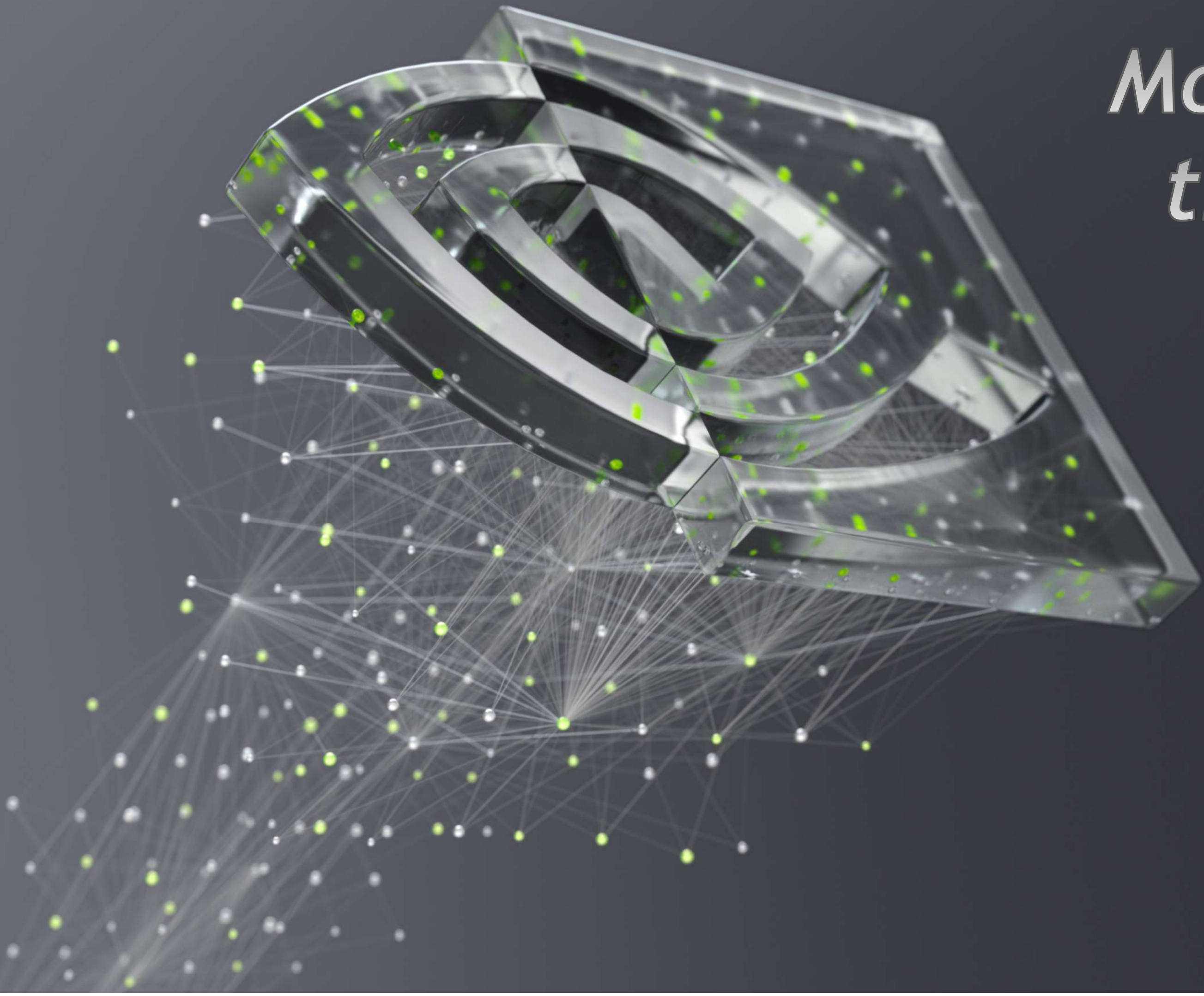


DISCUSSION

- Interaction with compiler optimizations:
  - Thread coarsening, loop unrolling, interchange, fission and fusion
  - If-conversion, function inlining, code refactoring
- Interaction with architectural features:
  - Scalar datapaths
  - Warp synchronous instructions (e.g. SHFL)

CONCLUSIONS

- *Sometimes, code that could execute in parallel is serialized with PDOM reconvergence*
- *Reconverge at alternative locations for better SIMT efficiency and performance*
- *Speculative reconvergence is a sharp-edged tool*



More in common than we think

NVIDIA.