WARPS AND VOLTA ARCHITECTURE

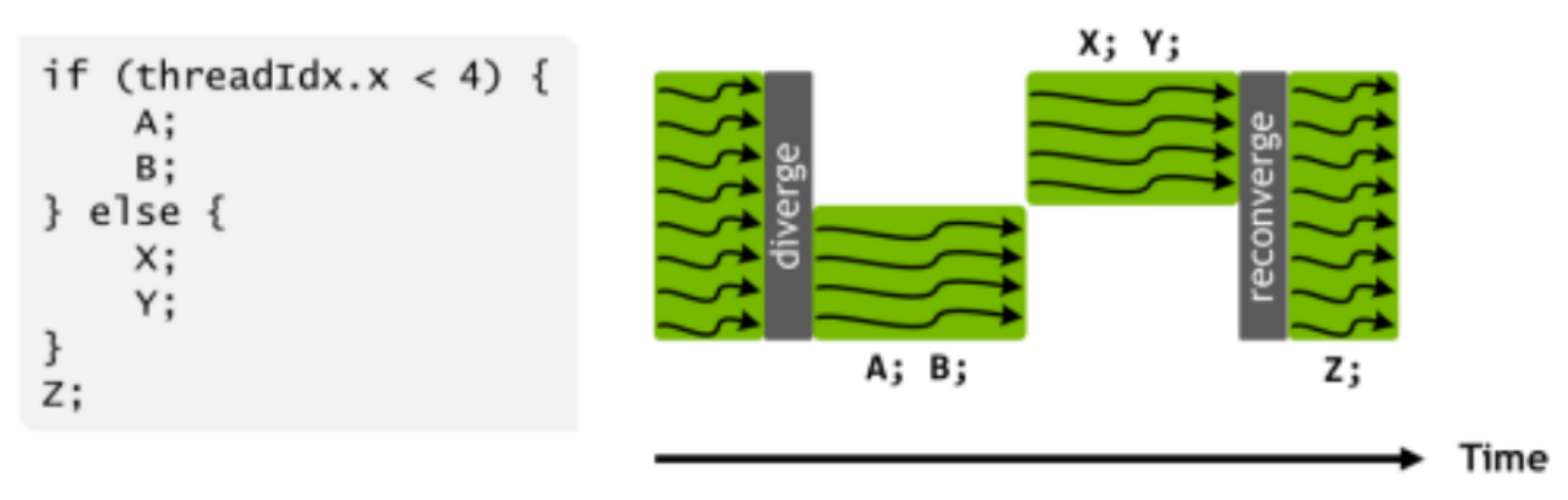
25 May 2025 00:10

INDEPENDENT THREAD SCHEDULING

The Volta architecture is designed to be significantly easier to program than prior GPUs, enabling users to work productively on more complex and diverse applications. Volta GV100 is the first GPU to support independent thread scheduling, which enables finer-grain synchronization and cooperation between parallel threads in a program. One of the major design goals for Volta was to reduce the effort required to get programs running on the GPU, and to enable greater flexibility in thread cooperation, leading to higher efficiency for fine-grained parallel algorithms.

Prior NVIDIA GPU SIMT Models

Pascal and earlier NVIDIA GPUs execute groups of 32 threads (known as warps) in SIMT (Single Instruction, Multiple Thread) fashion. The Pascal warp uses a single program counter shared amongst all 32 threads, combined with an *active mask* that specifies which threads of the warp are active at any given time. This means that divergent execution paths leave some threads inactive, serializing execution for different portions of the warp as shown in Figure 20. The original mask is stored until the warp reconverges, typically at the end of the divergent section, at which point the mask is restored and the threads run together once again.



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

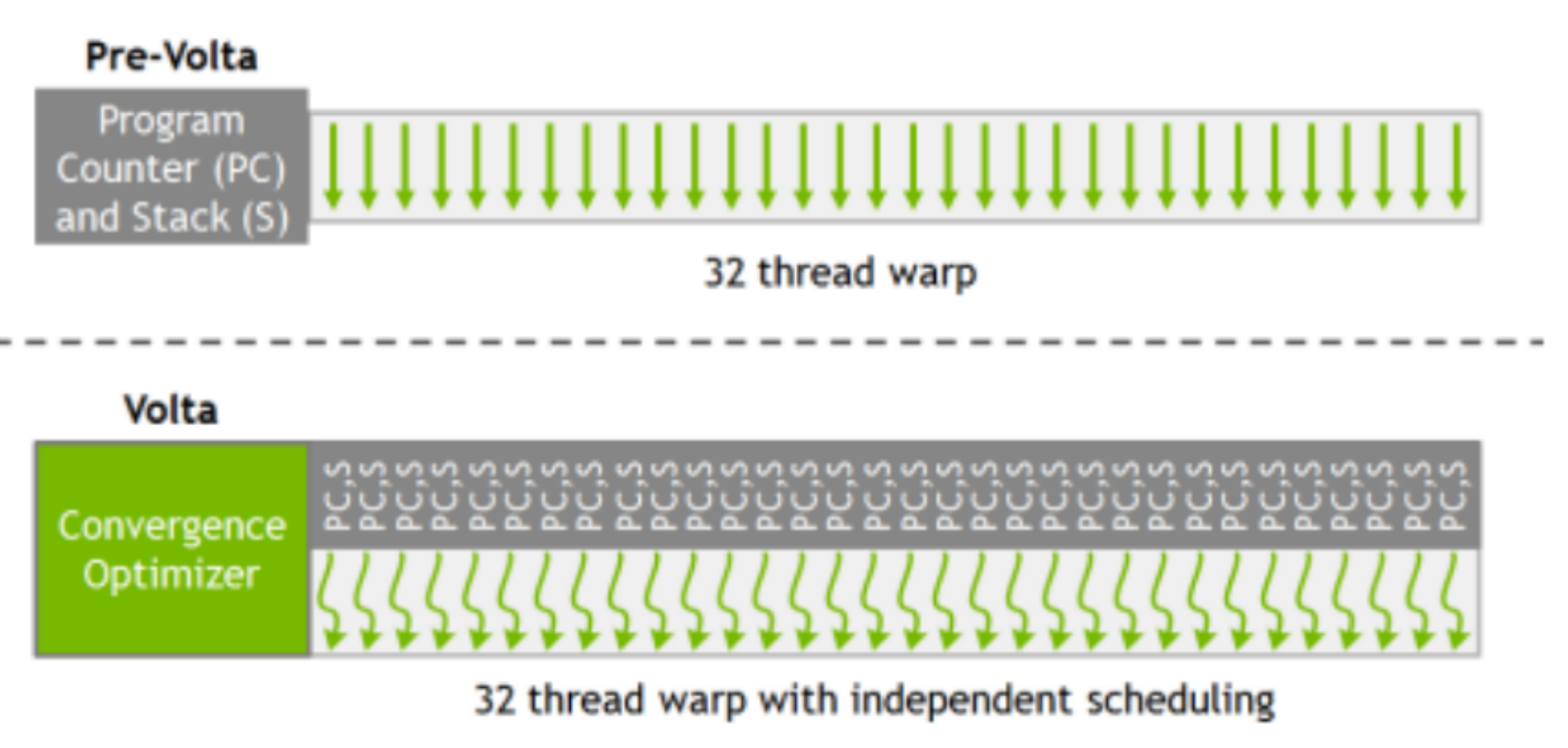
Thread scheduling under the SIMT warp execution model of Pascal and earlier NVIDIA GPUs. Capital letters represent statements in the program pseudocode. Divergent branches within a warp are serialized so that all statements in one side of the branch are executed together to completion before any statements in the other side are executed. After the else statement, the threads of the warp will typically reconverge.

Figure 20. SIMT Warp Execution Model of Pascal and Earlier GPUs

The Pascal SIMT execution model maximizes efficiency by reducing the quantity of resources required to track thread state and by aggressively reconverging threads to maximize parallelism. Tracking thread state in aggregate for the whole warp, however, means that when the execution pathway diverges, the threads which take different branches lose concurrency until they reconverge. This loss of concurrency means that threads from the same warp in divergent regions or different states of execution cannot signal each other or exchange data. This presents an inconsistency in which threads from different warps continue to run concurrently, but diverged threads from the same warp run sequentially until they reconverge. This means, for example, that algorithms requiring fine-grained sharing of data guarded by locks or mutexes can easily lead to deadlock, depending on which warp the contending threads come from. Therefore, on Pascal and earlier GPUs, programmers need to avoid fine-grained synchronization or rely on lock-free or warp-aware algorithms.

Volta SIMT Model

Volta transforms this picture by enabling equal concurrency between all threads, regardless of warp. It does this by maintaining execution state per thread, including a program counter and call stack, as shown in Figure 21.



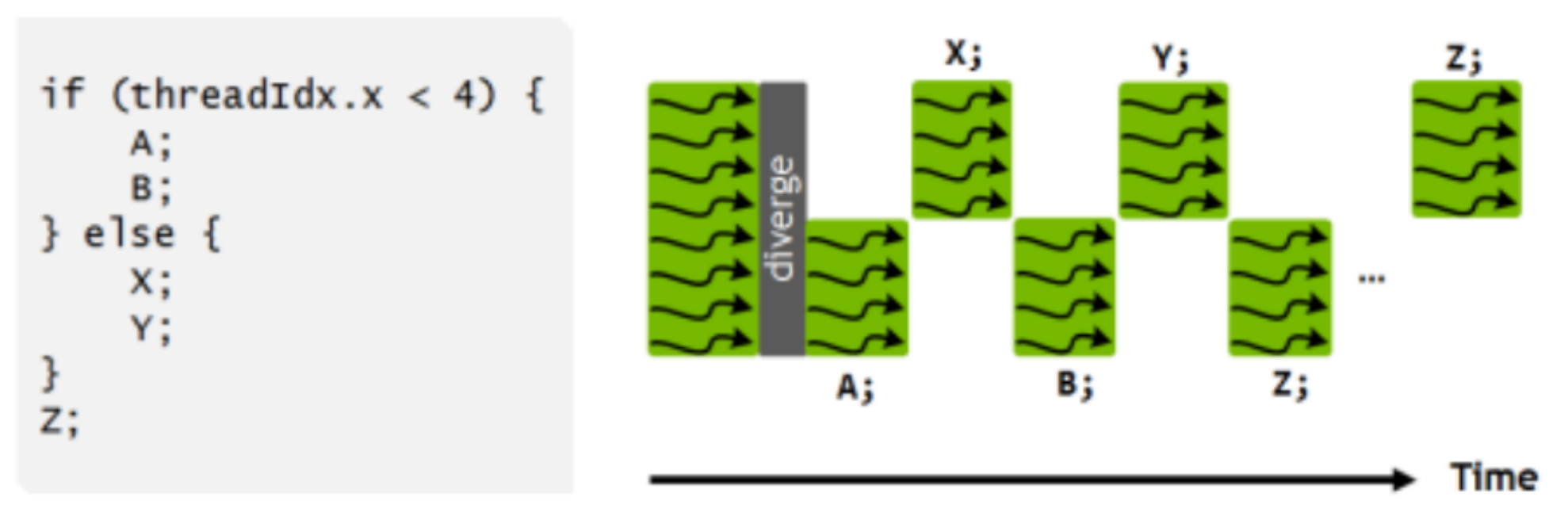
Pre-Volta
Program Counter (PC) and Stack (S)
32 thread warp
Volta
Convergence Optimizer
PC,S
32 thread warp with independent scheduling

Volta (bottom) independent thread scheduling architecture block diagram compared to Pascal and earlier architectures (top). Volta maintains per-thread scheduling resources such as program counter (PC) and call stack (S), while earlier architectures maintained these resources per warp.

Figure 21. Volta Warp with Per-Thread Program Counter and Call Stack

Volta's independent thread scheduling allows the GPU to yield execution of any thread, either to make better use of execution resources or to allow one thread to wait for data to be produced by another. To maximize parallel efficiency, Volta includes a schedule optimizer which determines how to group active threads from the same warp together into SIMT units. This retains the high throughput of SIMT execution as in prior NVIDIA GPUs, but with much more flexibility: threads can now diverge and reconverge at sub-warp granularity, while the convergence optimizer in Volta will still group together threads which are executing the same code and run them in parallel for maximum efficiency

Execution of the code example from Figure 20 looks somewhat different on Volta. Statements from the *if* and *else* branches in the program can now be interleaved in time as shown in Figure 22. Note that execution is still SIMT: at any given clock cycle, CUDA cores execute the same instruction for all active threads in a warp just as before, retaining the execution efficiency of previous architectures. Importantly, Volta's ability to independently schedule threads within a warp makes it possible to implement complex, fine-grained algorithms and data structures in a more natural way. While the scheduler supports independent execution of threads, it optimizes non-synchronizing code to maintain as much convergence as possible for maximum SIMT efficiency.



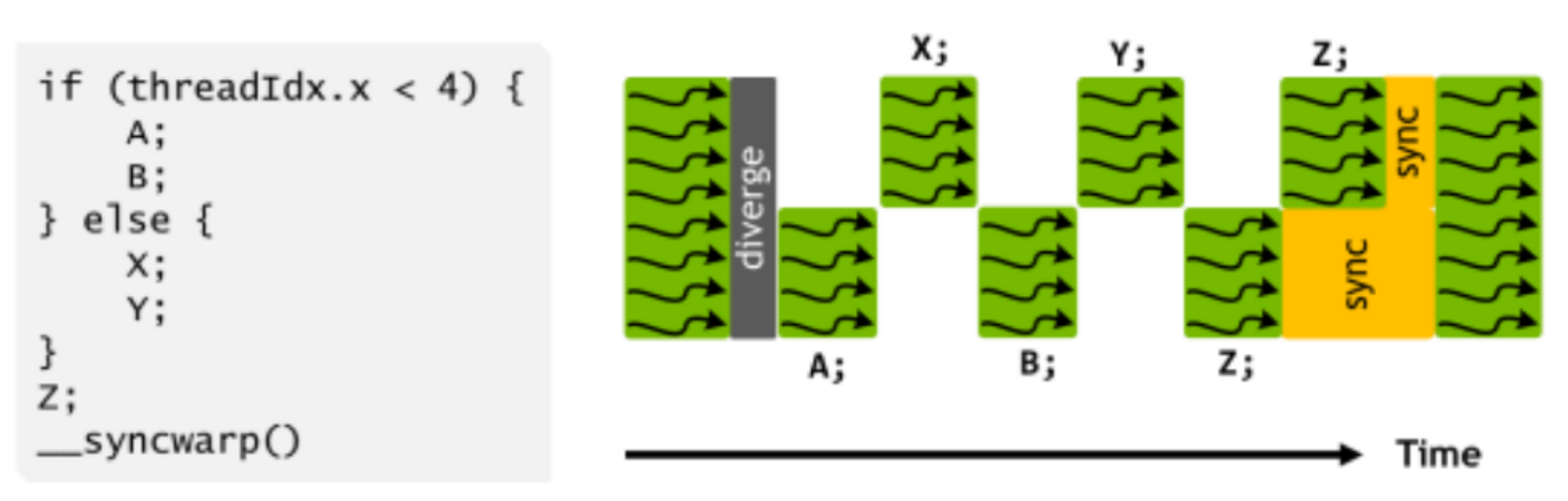
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

Volta independent thread scheduling enables interleaved execution of statements from divergent branches. This enables execution of fine-grain parallel algorithms where threads within a warp may synchronize and communicate.

Figure 22. Volta Independent Thread Scheduling

It is interesting to note that Figure 22 does not show execution of statement **Z** by all threads in the warp at the same time. This is because the scheduler must conservatively assume that **Z** may produce data required by other divergent branches of execution, in which case it would be unsafe to automatically enforce reconvergence. In the common case where **A**, **B**, **X**, and **Y** do not consist of synchronizing operations, the scheduler can identify that it is safe for the warp to naturally reconverge on **Z**, as in prior architectures.

Programs can call the new CUDA 9 warp synchronization function **__syncwarp()** to force reconvergence, as shown in Figure 23. In this case, the divergent portions of the warp might not execute **Z** together, but all execution pathways from threads within a warp will complete before any thread reaches the statement after the **__syncwarp()**. Similarly, placing the call to **__syncwarp()** before the execution of **Z** would force reconvergence before executing **Z**, potentially enabling greater SIMT efficiency if the developer knows that this is safe for their application.



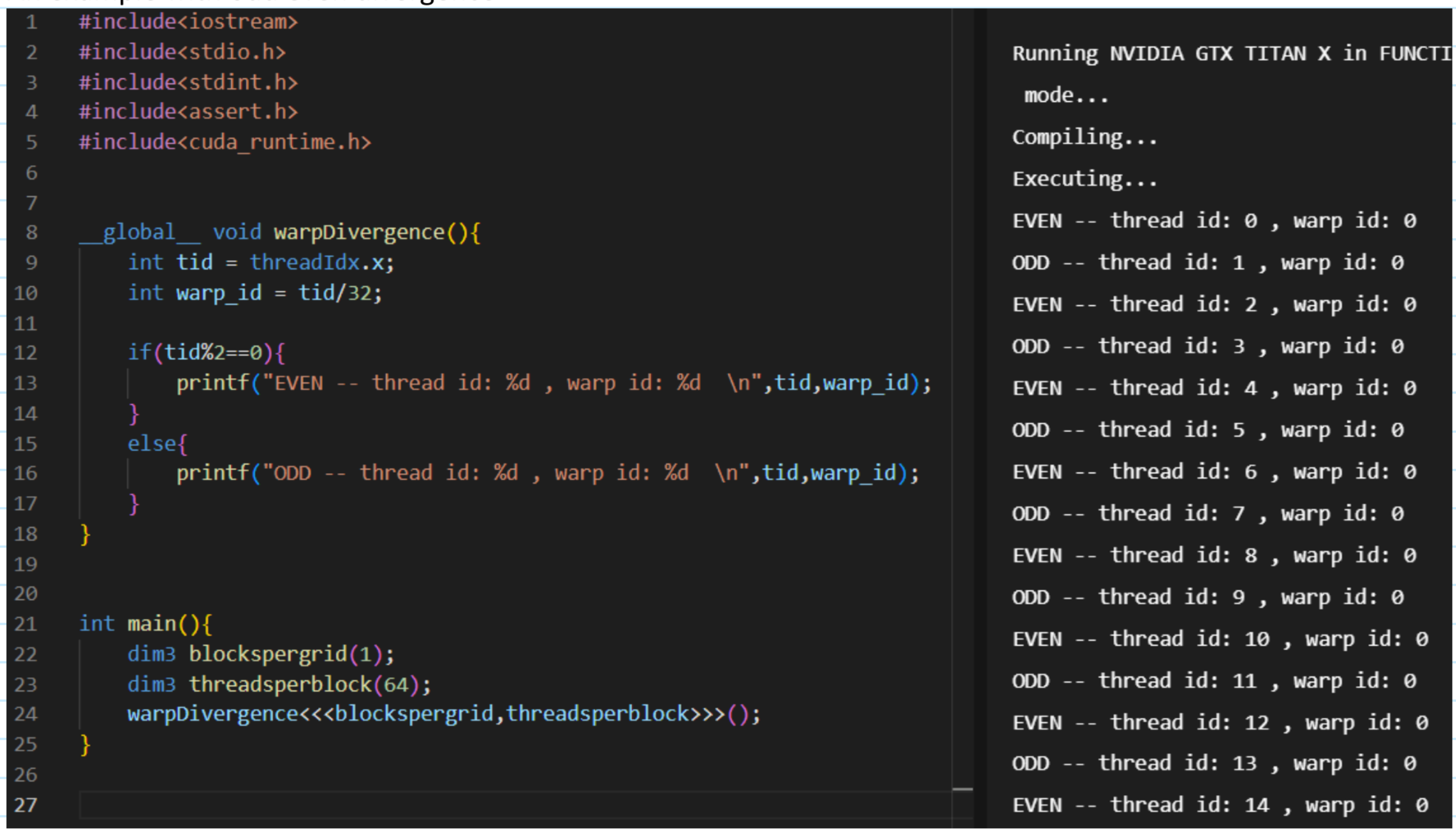
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```

Figure 23. Programs use Explicit Synchronization to Reconverge Threads in a Warp

In CUDA, a **warp** is a group of **32 threads** that execute the same instruction simultaneously in a SIMT (Single Instruction, Multiple Threads) fashion. When you launch a kernel, CUDA schedules threads in warps of 32.

- **Divergence happens at warp-level only**, not across warps.
- **NVIDIA compilers try to minimize divergence**, but explicit conditionals can still cause it.
- Performance degrades with **frequent and nested divergent branches** in the same warp.
- Avoid divergence in performance-critical kernels if possible, especially in inner loops.

An example with odd even divergence



```
#include<iostream>
#include<stdio.h>
#include<stdint.h>
#include<assert.h>
#include<cuda_runtime.h>


__global__ void warpDivergence(){
    int tid = threadIdx.x;
    int warp_id = tid/32;

    if(tid%2==0){
        printf("EVEN -- thread id: %d , warp id: %d  \n",tid,warp_id);
    }
    else{
        printf("ODD -- thread id: %d , warp id: %d  \n",tid,warp_id);
    }
}


int main(){
    dim3 blockspergrid(1);
    dim3 threadsperblock(64);
    warpDivergence<<<blockspergrid,threadsperblock>>>();
}

```

```
Running NVIDIA GTX TITAN X in FUNCTI
 mode...
Compiling...
Executing...
EVEN -- thread id: 0 , warp id: 0
ODD -- thread id: 1 , warp id: 0
EVEN -- thread id: 2 , warp id: 0
ODD -- thread id: 3 , warp id: 0
EVEN -- thread id: 4 , warp id: 0
ODD -- thread id: 5 , warp id: 0
EVEN -- thread id: 6 , warp id: 0
ODD -- thread id: 7 , warp id: 0
EVEN -- thread id: 8 , warp id: 0
ODD -- thread id: 9 , warp id: 0
EVEN -- thread id: 10 , warp id: 0
ODD -- thread id: 11 , warp id: 0
EVEN -- thread id: 12 , warp id: 0
ODD -- thread id: 13 , warp id: 0
EVEN -- thread id: 14 , warp id: 0
```

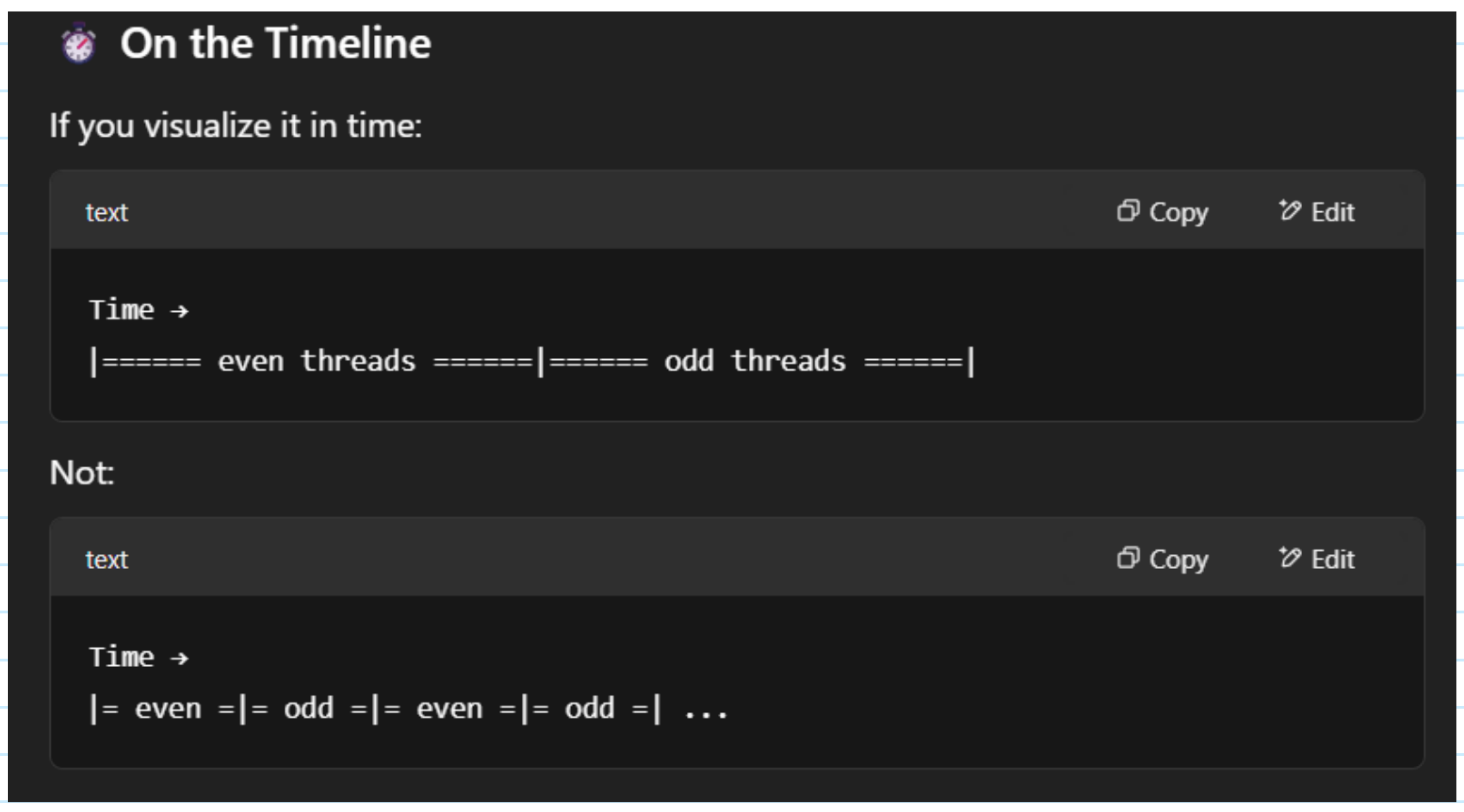
Now guess whether the GPU first execute all even threads, then all odd threads, or does it switch back and forth?

ANSWER - It executes **one branch path completely first**, then switches to the other.

The warp scheduler will:

1. **Mask off** threads that don't satisfy the if condition (i.e., odd threads).
2. **Execute the if block** for all **even threads**.
3. Then **mask off** even threads.
4. **Execute the else block** for all **odd threads**.

- First mask: [1 0 1 0 1 ...] → only even threads active → run if block
- Second mask: [0 1 0 1 0 ...] → only odd threads active → run else block



On the Timeline

If you visualize it in time:

```
Time →
|====== even threads ======|====== odd threads ======|
```

Not:

```
Time →
|= even =|= odd =|= even =|= odd =| ...
```