Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

December 10, 2019

NPTEL

Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

NPTEL

GPU can be viewed as an array of Streaming Multiprocessors (SMs) Each SM has the following elements

- Registers that can be partitioned among threads of execution
- Several Caches: Shared memory, Constant, Texture, L1 etc
- Warp Schedulers (More on this later)
- Scalar Processors(SPs) for integer and floating-point operations
- Special Function Units (SFUs) for single-precision floating-point transcendental functions

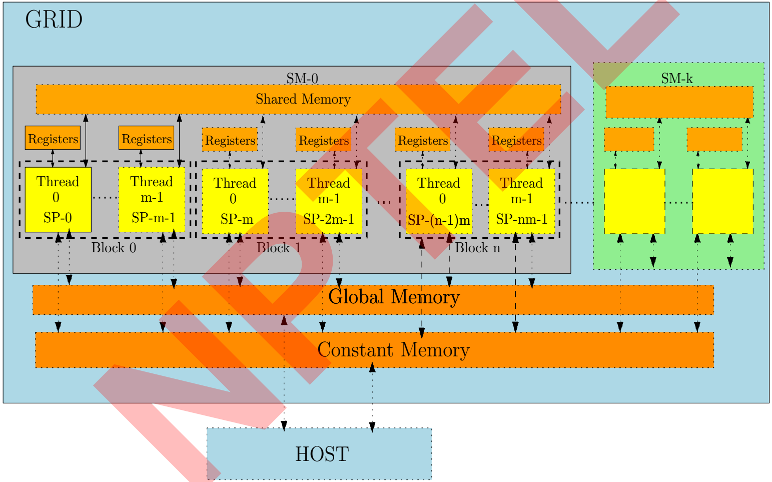
NPTEL

Table: CUDA Device Memory Types and Scopes

Variables Declaration	Memory	Scope	Lifetime
Automatic Variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
__device__ __shared__ int SharedVar	Shared	Block	Kernel
__device__ int GlobalVar	Global	Grid	Application
__device__ __constant__ int ConstVar	Constant	Grid	Application

NPTEL

Mapping to Hardware



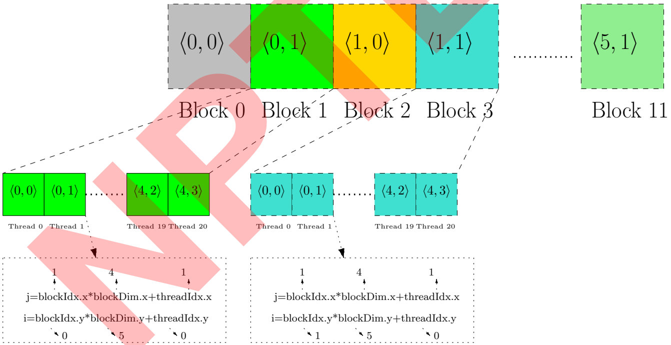
NPTEL

Example: CUDA Thread and Block Definition

NumCols = blockDim.x * gridDim.x

NumRows = blockDim.y * gridDim.y

$gridDim = \langle 6, 3 \rangle$ $blockDim = \langle 4, 5 \rangle$



NPTEL

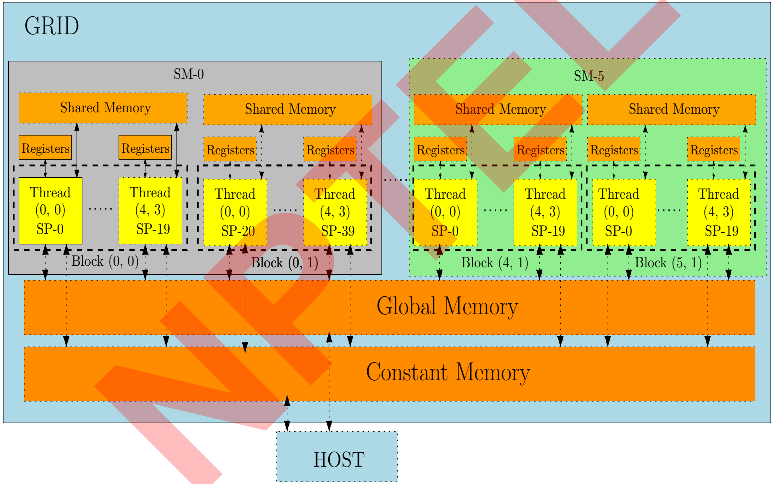
Figure: Kernel Grid Specification

Generalized Mapping Scenario

NPTEL

- Let us consider a scenario for the grid and block dimensions specified above.
- $gridDim = < 6, 2 >$ and $blockDim = < 5, 4 >$
- $\#SMs = 6$ $\#SPs$ per $SM = 40$
- Two Blocks are mapped to one SM at a time.
- Hardware resources are completely utilized.

Mapping to Hardware



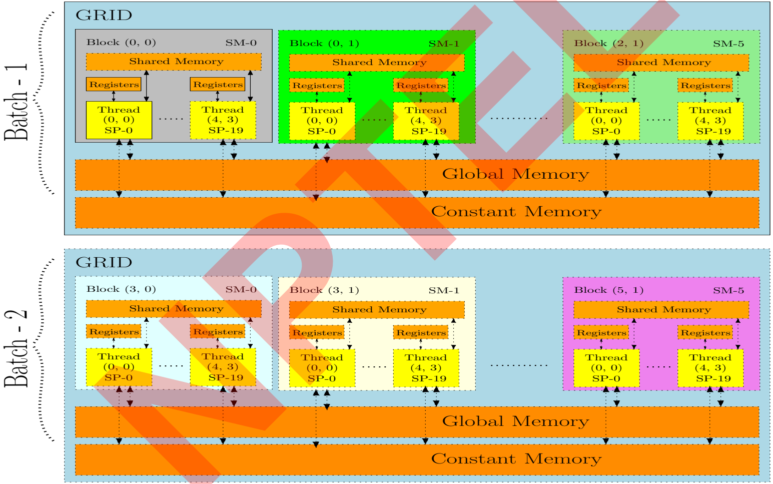
NPTEL

Mapping in a resource constrained setting

- Consider a scenario where the resources of the architecture are limited.
- $gridDim = < \mathbf{6}, 2 >$ and $blockDim = < 5, 4 >$
- $\#SMs = 6$ $\#SPs$ per $SM = 20$
- Thread Blocks are launched in batches sequentially.
- Execution is serialized to some extent.

NPTEL

Mapping to Hardware



NPTEL

Figure: Mapping Kernel Grid

SM, SP, Block and thread

- thread block max size : 1024 (modern archs 2048)
- SM can store max 1024 "thread contexts"
- can have much less than 1024 SPs
- GTX 970 : 13 SMs : 13 X 1024 thread contexts in parallel
- GTX 970 : 128 SP per SM

NPTEL

SM, SP, Block and thread

NPTEL

- One block in one SM
- One SM can have multiple blocks

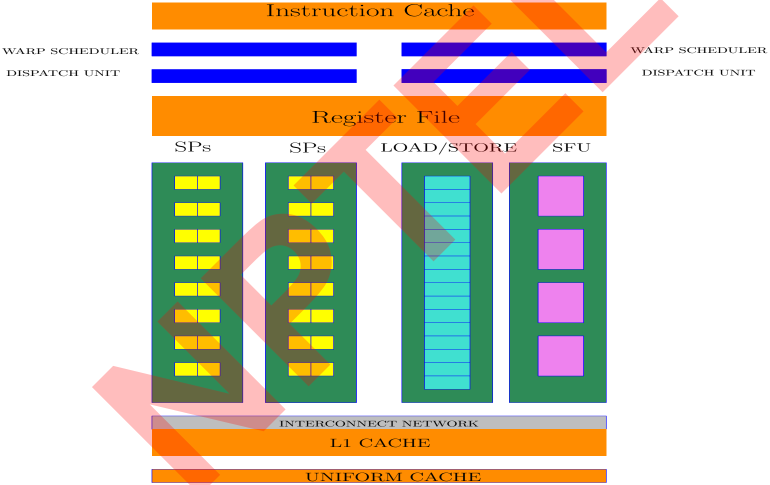
If SM can store max 1024 "thread contexts", and block size is 256, we have 4 blocks per SM.

GPU HW scheduler

- The hw scheduler decided which threads to map to a collection of SPs in SIMD fashion :: SIMT model of execution
- This collection is physically guaranteed to execute in parallel
- The unit of such collections is "warp"

NPTEL

SM: A closer look



NPTEL

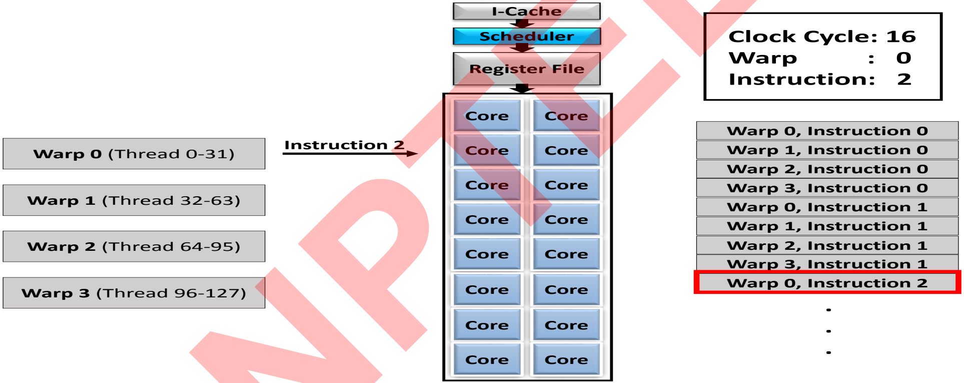
Warps

- Warp is a unit of thread Scheduling in SMs.
- Warp size is implementation specific (typically 32 threads)
- Warps are executed in an SIMD fashion i.e. the warp scheduler launches warps of threads and each warp typically executes one instruction across parallel threads.

Ex : If a SM has 128 SPs, it can execute 4 Warps at a given time (one Warp has 32 Threads)

NPTEL

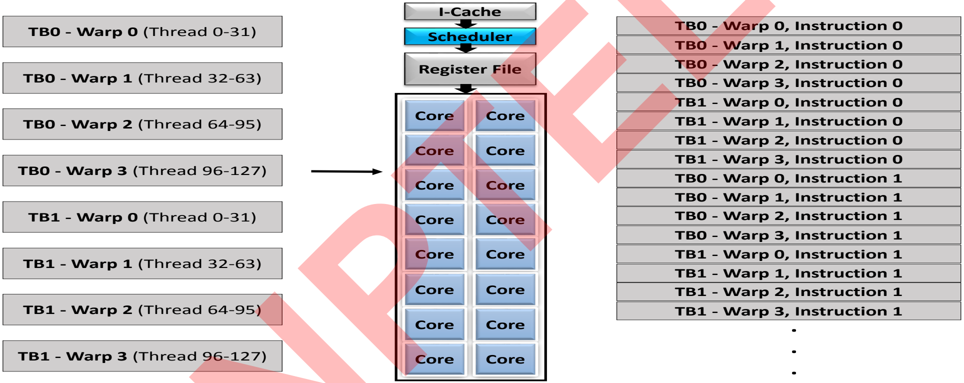
Warp Scheduling in SM



NPTEL

Ref : Henk Corporaal, Gert-Jan van den Braak - "Introduction to GPGPU Architectures"

Warp Scheduling in SM



TB0 - Warp 0 (Thread 0-31)

TB0 - Warp 1 (Thread 32-63)

TB0 - Warp 2 (Thread 64-95)

TB0 - Warp 3 (Thread 96-127)

TB1 - Warp 0 (Thread 0-31)

TB1 - Warp 1 (Thread 32-63)

TB1 - Warp 2 (Thread 64-95)

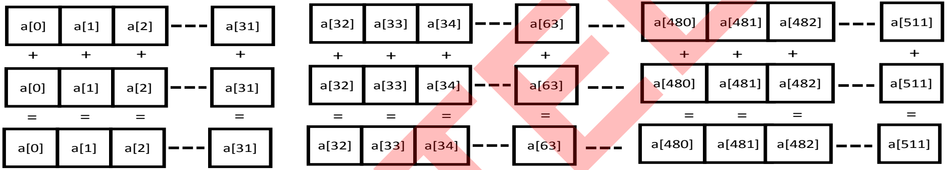
TB1 - Warp 3 (Thread 96-127)

TB0 - Warp 0, Instruction 0
TB0 - Warp 1, Instruction 0
TB0 - Warp 2, Instruction 0
TB0 - Warp 3, Instruction 0
TB1 - Warp 0, Instruction 0
TB1 - Warp 1, Instruction 0
TB1 - Warp 2, Instruction 0
TB1 - Warp 3, Instruction 0
TB0 - Warp 0, Instruction 1
TB0 - Warp 1, Instruction 1
TB0 - Warp 2, Instruction 1
TB0 - Warp 3, Instruction 1
TB1 - Warp 0, Instruction 1
TB1 - Warp 1, Instruction 1
TB1 - Warp 2, Instruction 1
TB1 - Warp 3, Instruction 1
.
.
.

- Thread block scheduler (TBS) is *believed to use* round robin policy to schedule thread blocks - implementation dependent

NPTEL

Warp Scheduling in SM


a[0] a[1] a[2] --- a[31]
+ + + +
a[0] a[1] a[2] --- a[31]
= = = =
a[0] a[1] a[2] --- a[31]
a[32] a[33] a[34] --- a[63]
+ + + +
a[32] a[33] a[34] --- a[63]
= = = =
a[32] a[33] a[34] --- a[63]
---
a[480] a[481] a[482] --- a[511]
+ + + +
a[480] a[481] a[482] --- a[511]
= = = =
a[480] a[481] a[482] --- a[511]

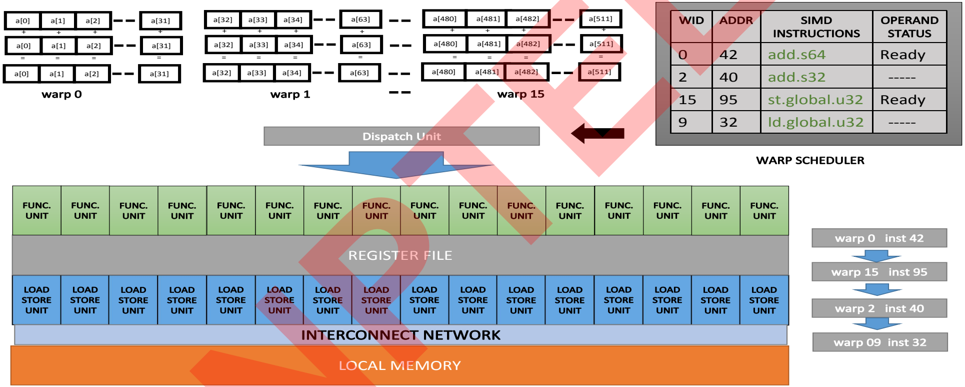
```
__global__ void twice(int *array)
{
 int tid = blockDim.x*blockIdx.x+threadIdx.x;
 array[tid] = array[tid] + array[tid];
}
```

```
mov.u32          %r1, %ntid.x;
mov.u32          %r2, %ctaid.x;
mov.u32          %r3, %tid.x;
mad.lo.s32        %r4, %r2, %r1, %r3;
mul.wide.s32     %rd3, %r4, 4;
add.s64          %rd4, %rd2, %rd3;
ld.global.u32    %r5, [%rd4];
add.s32          %r6, %r5, %r5;
st.global.u32    [%rd4], %r6;
```

Figure: Simple CUDA Kernel

NPTEL

Warp Scheduling in SM



WID	ADDR	SIMD INSTRUCTIONS	OPERAND STATUS
0	42	add.s64	Ready
2	40	add.s32	-----
15	95	st.global.u32	Ready
9	32	ld.global.u32	-----

Figure: Warp Scheduler

NPTEL

Warp Scheduling in SM

- Issue one “ready-to-go” warp instruction/cycle
- Use operand score-boarding to prevent hazards
- Issue selection based on round-robin/age of warp
- Score-boarding determines if a thread is ready to execute?
- Scoreboard is a HW implemented table that tracks - instrs fetched, resource availability for fetched instrs (FU and operand), register file modifications by instrs.

NPTEL

Latency Tolerance

- When threads in one warp execute a long-latency operation (read from global memory), the warp scheduler will dispatch and execute other warps until that operation is finished.
- Other long latency operations : FP units, Branch instructions
- After all, all threads in the same control-flow execute same instruction sequence on different data points !
- A common practice is to launch thread blocks of a size that is a multiple of the warp size to maximally utilize threads.
- Slow global memory accesses by threads in a warp may be optimized using coalescing (more on this later)

NPTEL

Efficient use of thread blocks

Target System Constraints

- A maximum of 8 blocks and 1024 threads per SM
- A maximum of 512 threads per block

Table: Solutions for various block scenarios

Input Block Size	Blocks per SM	Threads per Block	Remarks
8 * 8	12	64	SM execution resources will be underutilized
16*16	4	256	Achieves full thread capacity in SMs
32*32	1	1024	Exceeds the limit of 512 threads per block

NPTEL

Querying Device Properties

NPTEL

CUDA API provides constructs for obtaining properties of the target GPU.

- **cudaGetDeviceCount():** Obtains the number of devices in the system.
- **cudaGetDeviceProperties():** Returns the property values of a particular device

Querying Device Properties

```
int main()
{

    int devCount;
    cudaGetDeviceCount(&devCount);
    for (int i = 0; i < devCount; ++i)
    {
        cudaDeviceProp devp;
        cudaGetDeviceProperties(&devp, i);
        printDevProp(devp);
    }
        return 0;
}
```

NPTEL

Querying Device Properties

```
void printDevProp(cudaDeviceProp devProp)
{
 printf("Major revision number: %d\n",devProp.major);
 printf("Minor revision number: %d\n",devProp.minor);
 printf("Name: %s\n",devProp.name);
 printf("Total global memory: u\n",devProp.totalGlobalMem);
 printf("Total shared memory per block:%u\n", devProp.sharedMemPerBlock);
 printf("Total registers per block: %d\n", devProp.regsPerBlock);
 printf("Warp size: %d\n",devProp.warpSize);
 printf("Maximum memory pitch: %u\n",devProp.memPitch);
 printf("Maximum threads per block: %d\n",devProp.maxThreadsPerBlock);
 for (int i = 0; i < 3; ++i)
  printf("Maximum dimension %d of block: %d\n",i,devProp.maxThreadsDim[i]);
 for (int i = 0; i < 3; ++i)
  printf("Maximum dimension %d of grid:   %d\n", i, devProp.maxGridSize[i]);
```

NPTEL

Querying Device Properties

```
  printf("Clock rate: %d\n",devProp.clockRate);
  printf("Total constant memory:%u\n", devProp.totalConstMem);
  printf("Texture alignment: %u\n", devProp.textureAlignment);
  printf("Concurrent copy and execution: %s\n", (devProp.deviceOverlap ? "Yes"
      : "No"));
  printf("Number of multiprocessors: %d\n",devProp.multiProcessorCount);
  return;
}
```

NPTEL

Example: Tesla K40m Characteristics

```
Major revision number: 3
Minor revision number: 5
Name: Tesla K40m
Total global memory: 3405643776
Total shared memory per block:49152
Total registers per block: 65536
Warp size: 32
Maximum memory pitch: 2147483647
Maximum threads per block: 1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate: 745000
Total constant memory:65536
Texture alignment: 512
Concurrent copy and execution: Yes
Number of multiprocessors: 15
```

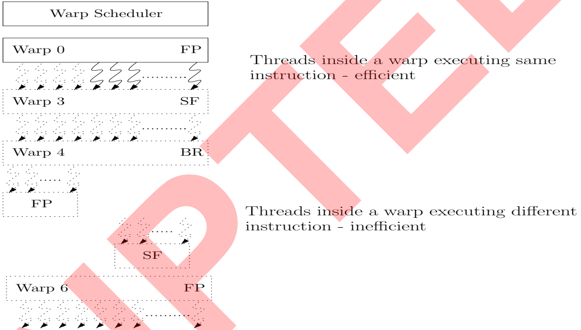
NPTEL

Control Flow Divergence

- Threads inside a warp execute the same instruction.
- How does a warp handle if statements / branch instructions?
- The GPU is not capable of running both the if else blocks at the same time.

NPTEL

Warp Scheduling



Threads inside a warp executing same instruction - efficient

Threads inside a warp executing different instruction - inefficient

Figure: Warp Divergence

NPTEL

Divergent Code 1

Consider the following kernel code

```
__global__
void divergence(float *M)
{
/*P1:*/ int tid=blockIdx.x*blockDim.x+threadIdx.x;
/*P2:*/ if(tid%2)
/*P3:*/         M[j]+=2;
        else
/*P4:*/    M[j]-=2;
/*P5:*/ M[j]*=2;
}
```

Half the threads of a warp execute the addition instruction while the other half execute the subtraction instruction.

NPTEL

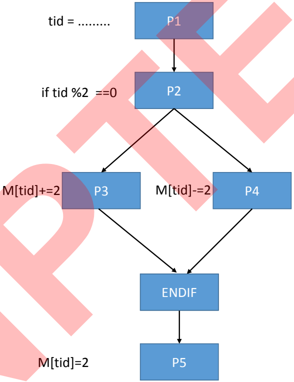
The Hardware's Job

The GPU has hardware support for handling divergent branch instructions in code.

- The PTX Assembler maintains internal masks, a branch synchronization stack and special markers
- The PTX Assembler sets a **branch synchronization marker** first for the divergent `if` statement that pushes the active mask on a stack inside each SIMD thread
- Depending on the value of the mask relevant threads execute instructions,
- Once the instructions in the `if` block are finished, the active mask is popped from the stack, flipped and pushed back.

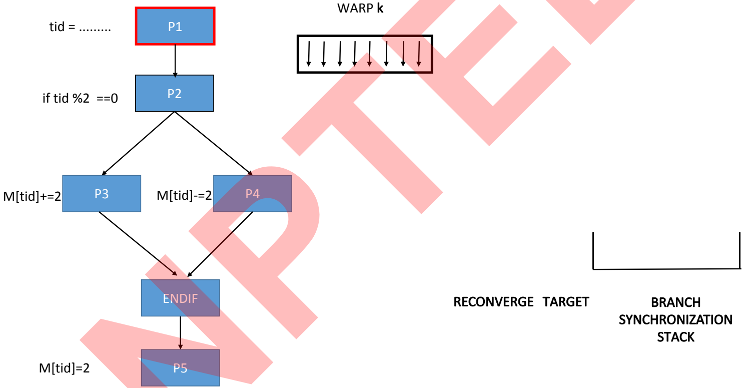
NPTEL

Divergent Code 1



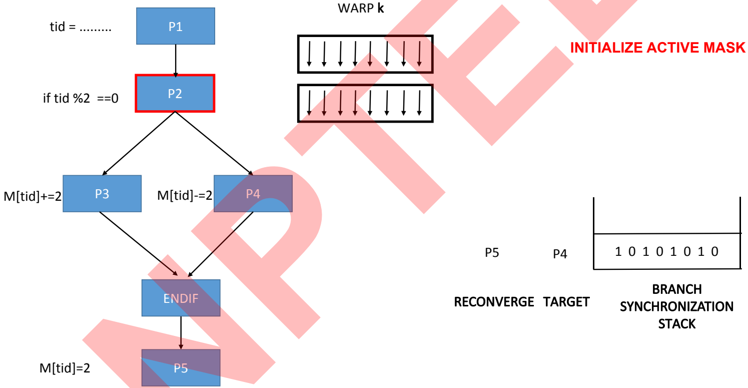
NPTEL

Divergent Code 1



NPTEL

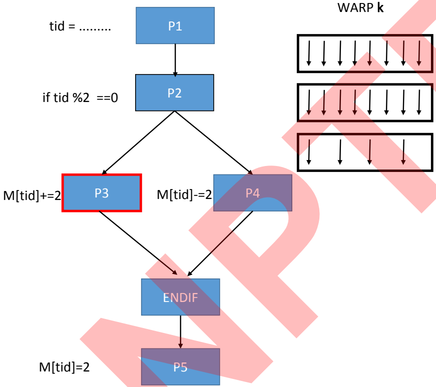
Divergent Code 1

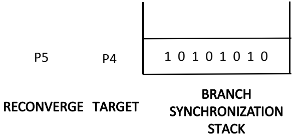


INITIALIZE ACTIVE MASK

NPTEL

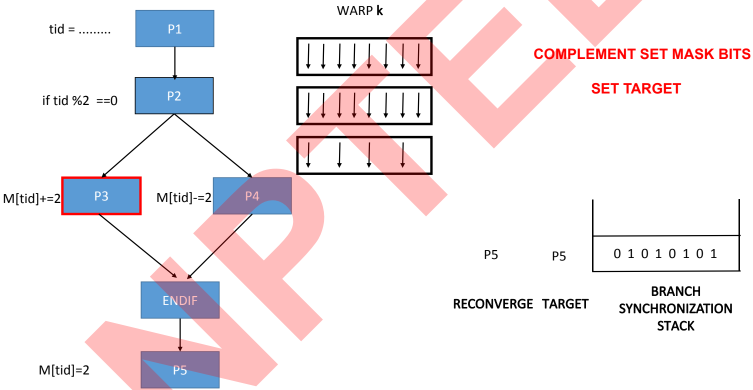
Divergent Code 1





NPTEL

Divergent Code 1

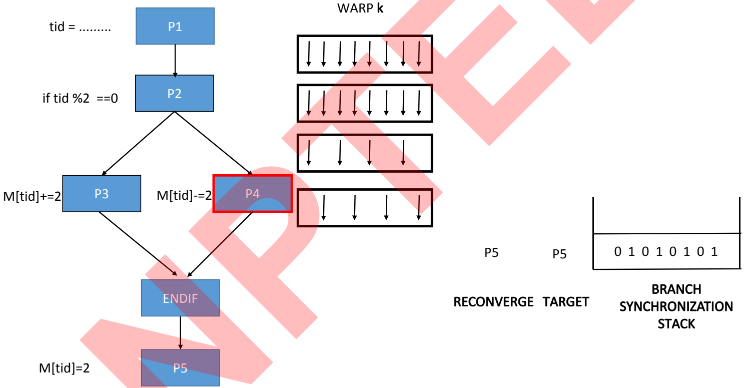


COMPLEMENT SET MASK BITS

SET TARGET

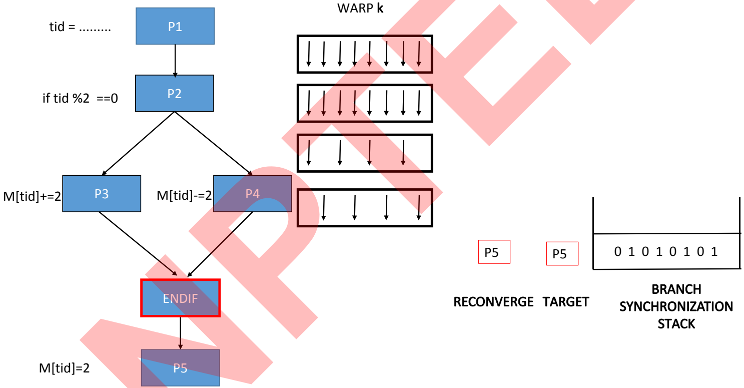
NPTEL

Divergent Code 1



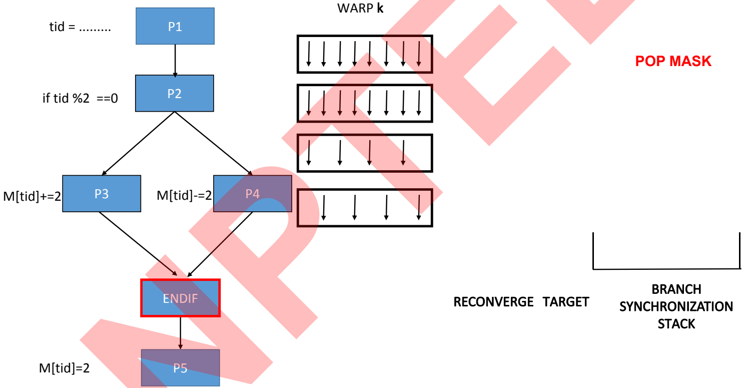
NPTEL

Divergent Code 1



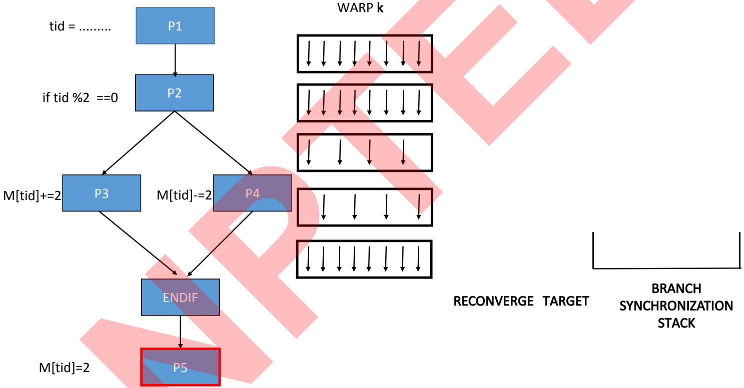
NPTEL

Divergent Code 1



NPTEL

Divergent Code 1



NPTEL

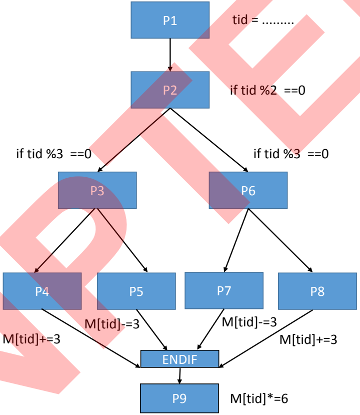
Divergent Code 2

Let us consider an example that has nested if/else statements.

```
__global__
void divergence(float *M)
{
/*P1*/      int tid=blockIdx.x*blockDim.x+threadIdx.x;
/*P2*/      if(tid%2==0)
            {
/*P3*/       if(tid%3==0)
/*P4*/         M[tid]+=3;
             else
/*P5*/         M[tid]-=3;
            }
            else
            {
/*P6*/       if(tid%3==0)
/*P7*/         M[tid]-=3;
             else
/*P8*/         M[tid]+=3;
            }
/*P9*/      M[tid]*=6;
}
```

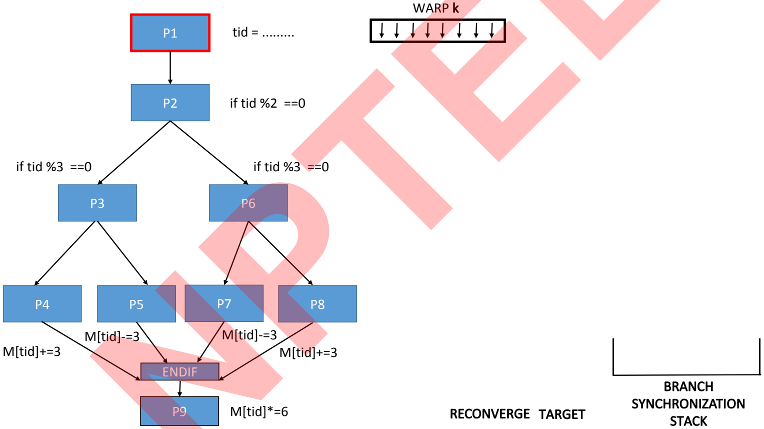
NPTEL

Divergence Code 2



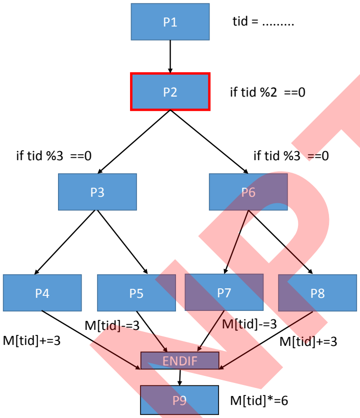
NPTEL

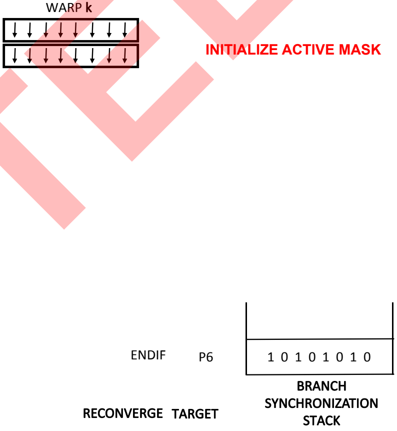
Divergence Code 2



NPTEL

Divergence Code 2



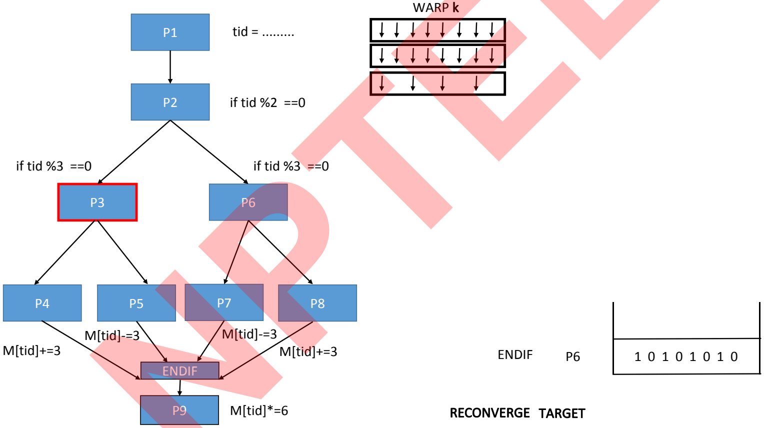


INITIALIZE ACTIVE MASK

RECONVERGE	TARGET	BRANCH SYNCHRONIZATION STACK
ENDIF	P6	1 0 1 0 1 0 1 0

NPTEL

Divergence Code 2



P1 tid = ...

P2 if tid %2 ==0

if tid %3 ==0 P3

if tid %3 ==0 P6

P4 M[tid]+=3

P5 M[tid]-=3

P7 M[tid]-=3

P8 M[tid]+=3

ENDIF

P9 M[tid]*=6

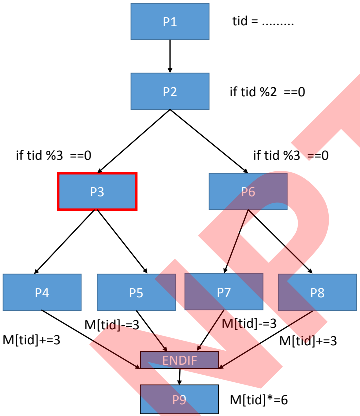
WARP k

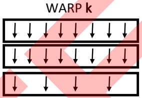
ENDIF P6 1 0 1 0 1 0 1 0

RECONVERGE TARGET

NPTEL

Divergence Code 2





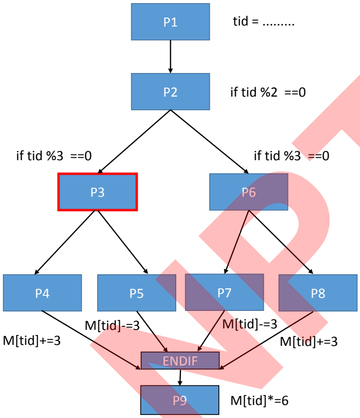
PUSH OLD MASK

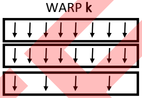
ENDIF	P5	1 0 1 0 1 0 1 0
P9	P6	1 0 1 0 1 0 1 0

RECONVERGE TARGET

NPTEL

Divergence Code 2



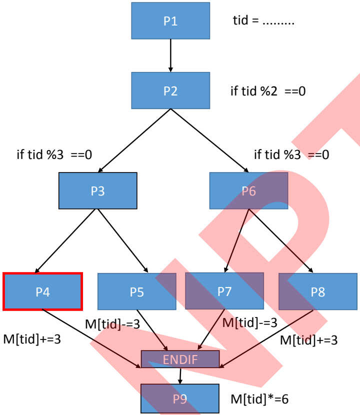


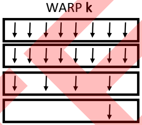
SET RELEVANT BITS

RECONVERGE	TARGET	
ENDIF	P5	0 0 0 0 0 0 1 0
ENDIF	P6	1 0 1 0 1 0 1 0

NPTEL

Divergence Code 2

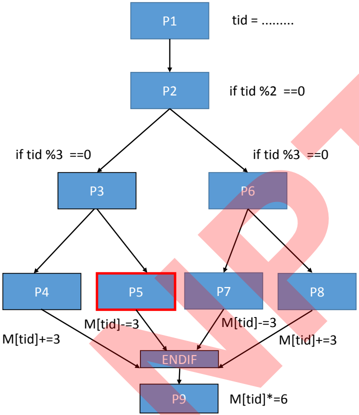


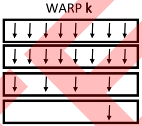


RECONVERGE	TARGET								
ENDIF	P5	0	0	0	0	0	0	1	0
ENDIF	P6	1	0	1	0	1	0	1	0

NPTEL

Divergence Code 2

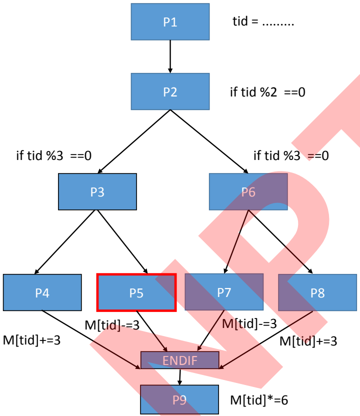


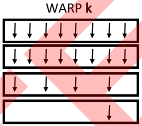


RECONVERGE	TARGET								
ENDIF	P5	0	0	0	0	0	0	1	0
ENDIF	P6	1	0	1	0	1	0	1	0

NPTEL

Divergence Code 2



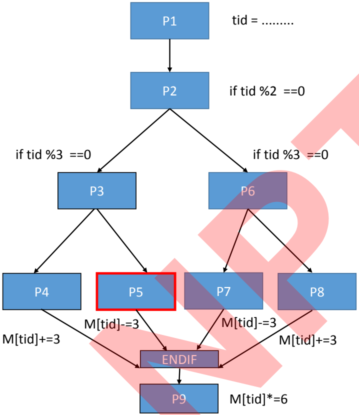


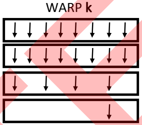
COMPLEMENT RELEVANT BITS

RECONVERGE	TARGET								
ENDIF	P5	0	0	0	0	0	0	1	0
ENDIF	P6	1	0	1	0	1	0	1	0

NPTEL

Divergence Code 2



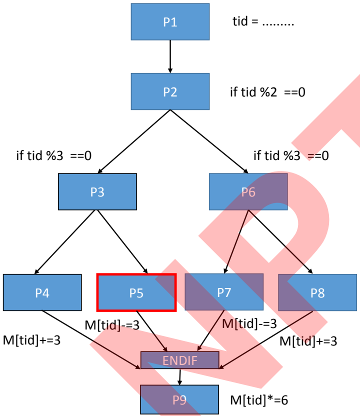


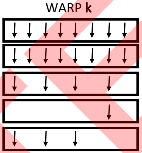
COMPLEMENT RELEVANT BITS

RECONVERGE	TARGET	
ENDIF	ENDIF	1 0 1 0 1 0 0 0
ENDIF	P6	1 0 1 0 1 0 1 0

NPTEL

Divergence Code 2



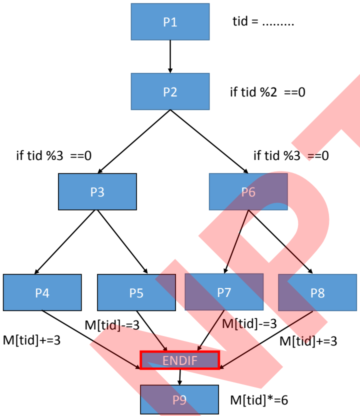


COMPLEMENT RELEVANT BITS

RECONVERGE	TARGET	
ENDIF	ENDIF	1 0 1 0 1 0 0 0
ENDIF	P6	1 0 1 0 1 0 1 0

NPTEL

Divergence Code 2



P1 tid = ...

P2 if tid %2 ==0

if tid %3 ==0

if tid %3 ==0

P3

P6

P4

P5

P7

P8

M[tid]+=3

M[tid]-=3

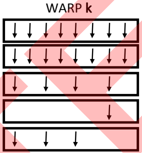
M[tid]-=3

M[tid]+=3

ENDIF

P9 M[tid]*=6

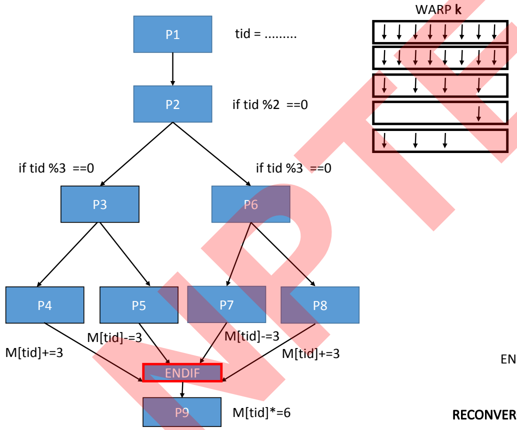
WARP **k**



RECONVERGE	TARGET	
ENDIF	ENDIF	1 0 1 0 1 0 0 0
ENDIF	P6	1 0 1 0 1 0 1 0

NPTEL

Divergence Code 2

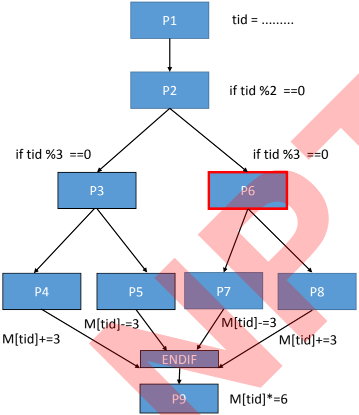


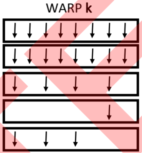
POP STACK

RECONVERGE	TARGET	
ENDIF	P6	1 0 1 0 1 0 1 0

NPTEL

Divergence Code 2



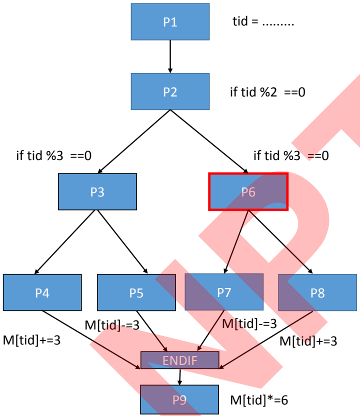


ENDIF	P6	1 0 1 0 1 0 1 0

RECONVERGE TARGET

NPTEL

Divergence Code 2



P1 tid = ...

P2 if tid %2 ==0

if tid %3 ==0

P3

if tid %3 ==0

P6

P4

P5

P7

P8

M[tid]+=3

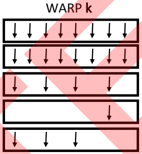
M[tid]-=3

M[tid]-=3

M[tid]+=3

ENDIF

P9 M[tid]*=6



COMPLEMENT RELEVANT BITS

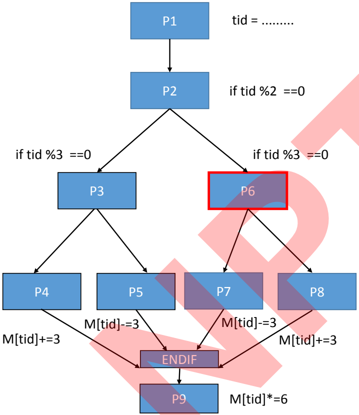
SET TARGET

ENDIF ENDIF 0 1 0 1 0 1 0 1

RECONVERGE TARGET

NPTEL

Divergence Code 2



P1

tid = ...

P2

if tid %2 ==0

if tid %3 ==0

P3

if tid %3 ==0

P6

P4

P5

P7

P8

M[tid]+=3

M[tid]-=3

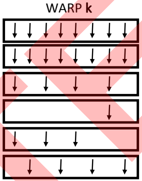
M[tid]-=3

M[tid]+=3

ENDIF

P9

M[tid]*=6



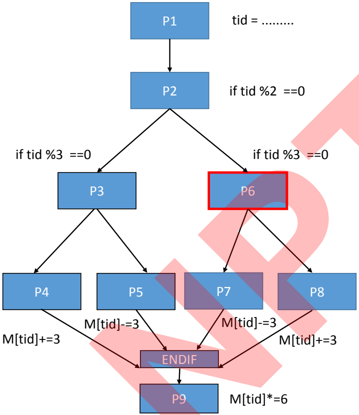
WARP k

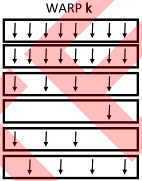
ENDIF ENDIF 0 1 0 1 0 1 0 1

RECONVERGE TARGET

NPTEL

Divergence Code 2





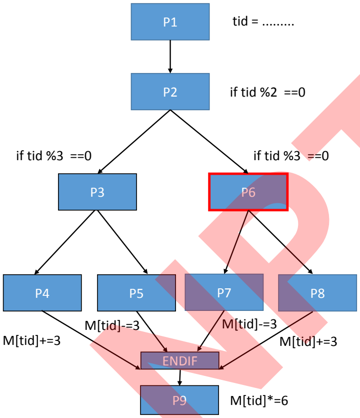
PUSH OLD MASK

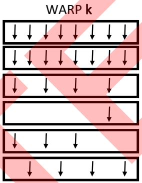
ENDIF	P8	0 1 0 1 0 1 0 1
ENDIF	ENDIF	0 1 0 1 0 1 0 1

RECONVERGE TARGET

NPTEL

Divergence Code 2



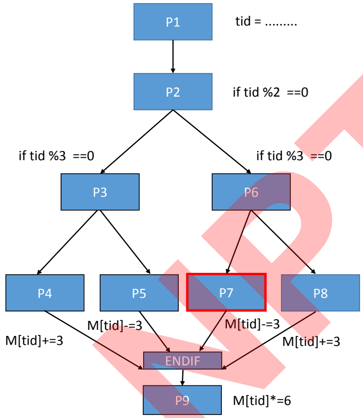


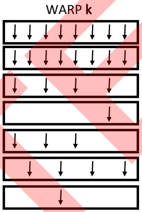
SET RELEVANT BITS

RECONVERGE	TARGET								
ENDIF	P8	0	0	0	1	0	0	0	0
ENDIF	ENDIF	0	1	0	1	0	1	0	1

NPTEL

Divergence Code 2

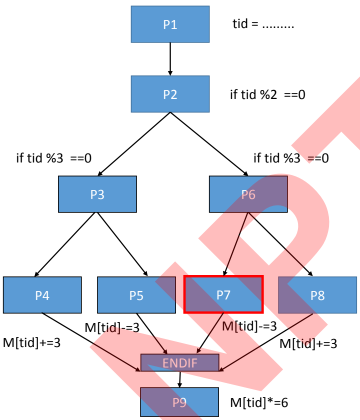


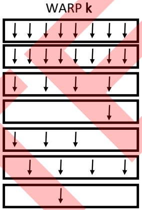


RECONVERGE	TARGET								
ENDIF	P8	0	0	0	1	0	0	0	0
ENDIF	ENDIF	0	1	0	1	0	1	0	1

NPTEL

Divergence Code 2





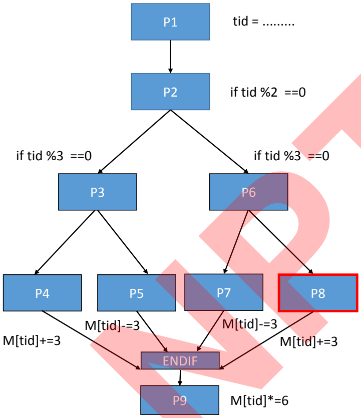
COMPLEMENT RELEVANT BITS

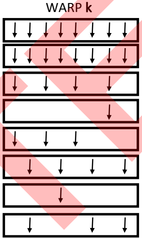
SET TARGET

RECONVERGE	TARGET	
ENDIF	ENDIF	0 1 0 0 0 1 0 1
ENDIF	ENDIF	0 1 0 1 0 1 0 1

NPTEL

Divergence Code 2

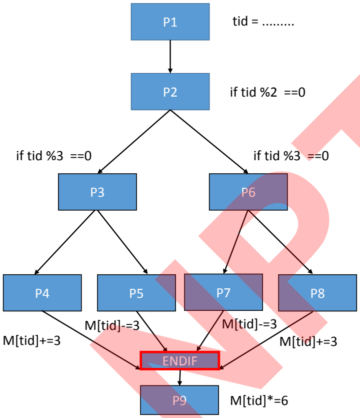


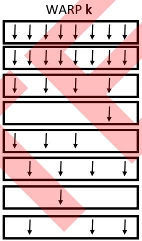


RECONVERGE	TARGET	
ENDIF	ENDIF	0 1 0 0 0 1 0 1
ENDIF	ENDIF	0 1 0 1 0 1 0 1

NPTEL

Divergence Code 2

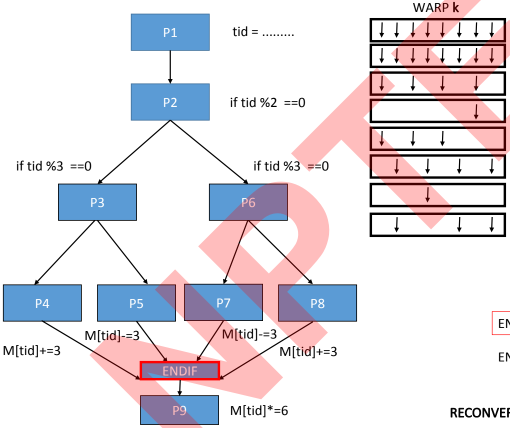




RECONVERGE	TARGET	
ENDIF	ENDIF	0 1 0 0 0 1 0 1
ENDIF	ENDIF	0 1 0 1 0 1 0 1

NPTEL

Divergence Code 2

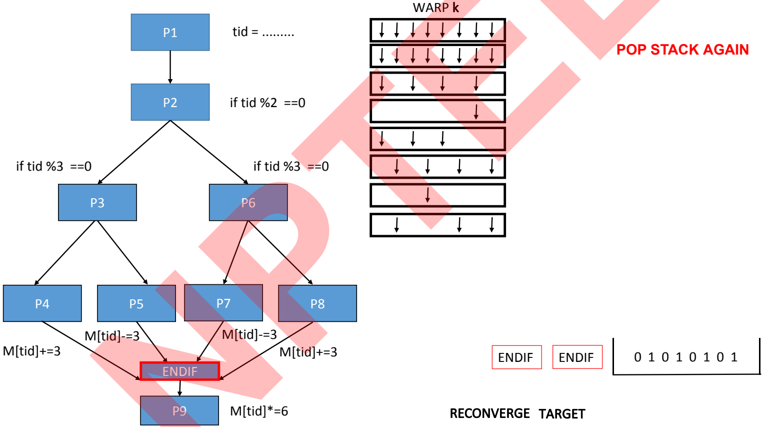


POP STACK

RECONVERGE	TARGET								
ENDIF	ENDIF	0	1	0	0	0	1	0	1
ENDIF	ENDIF	0	1	0	1	0	1	0	1

NPTEL

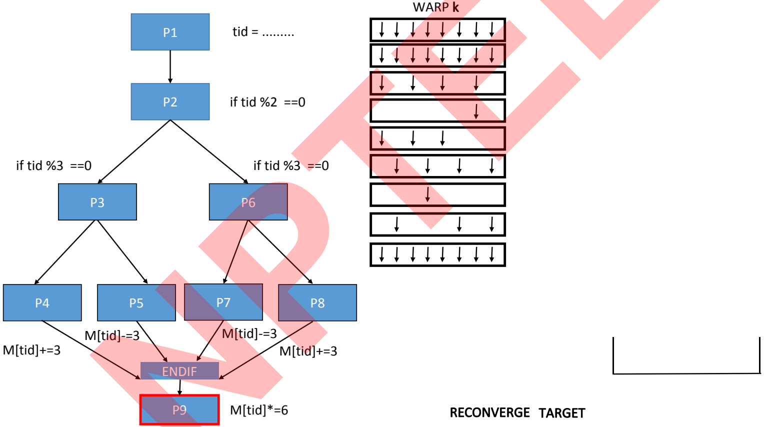
Divergence Code 2



POP STACK AGAIN

NPTEL

Divergence Code 2



P1 tid = ...

P2 if tid %2 ==0

if tid %3 ==0

P3

if tid %3 ==0

P6

P4 M[tid]+=3

P5 M[tid]-=3

P7 M[tid]-=3

P8 M[tid]+=3

ENDIF

P9 M[tid]*=6

WARP k

RECONVERGE TARGET

NPTEL

Programming tips

- GPU programmer has to be aware of hardware imposed restrictions - threads/SM, blocks/SM, threads/blocks, threads/warps
- The only safe way to synchronize threads from different blocks is to terminate kernel and make a fresh launch at the target synchronization point

NPTEL