Warp Scheduling and Divergence

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

February 13, 2020

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

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

Mapping to Hardware

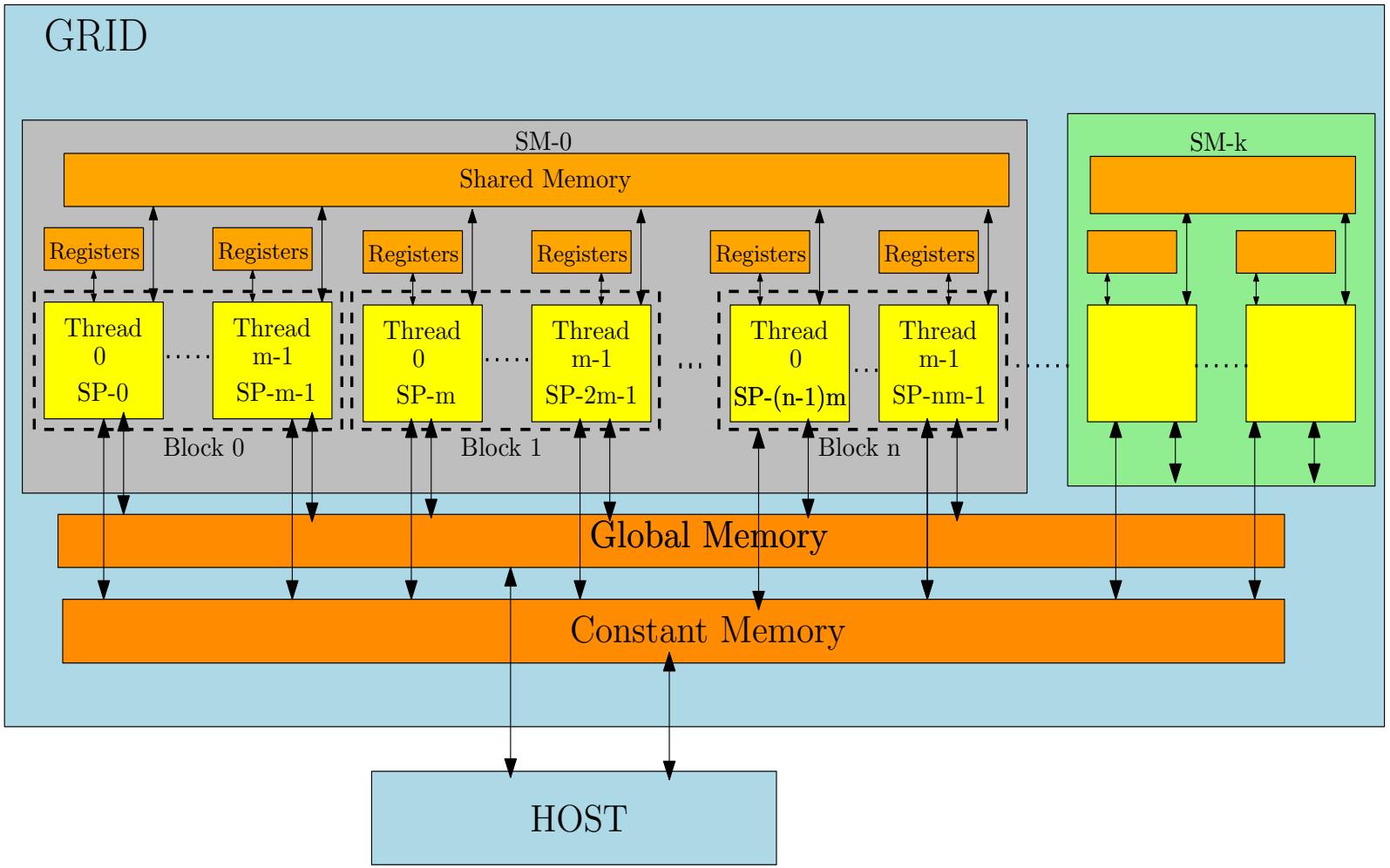


Figure: Mapping Kernel Grids to Architecture

Example: CUDA Thread and Block Definition

NumCols = blockDim.x * gridDim.x

NumRows = blockDim.y * gridDim.y

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

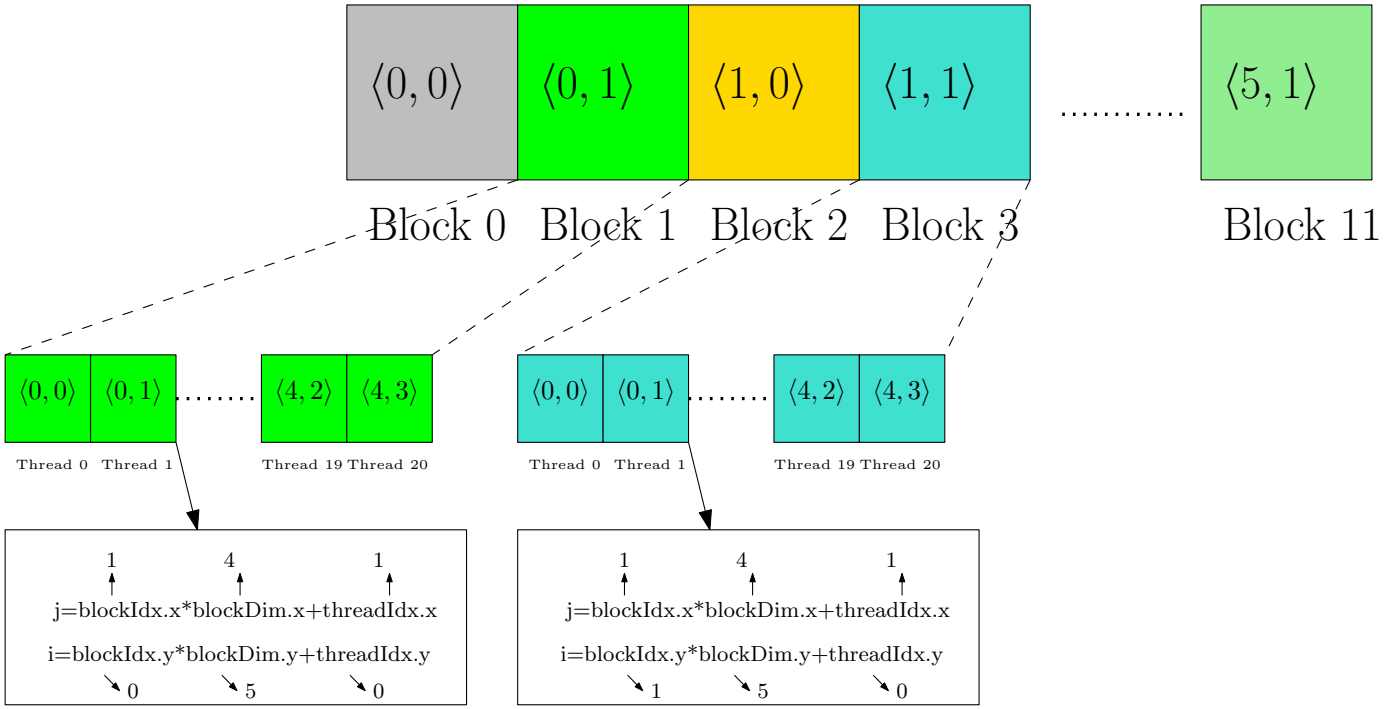


Figure: Kernel Grid Specification

Generalized Mapping Scenario

- 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

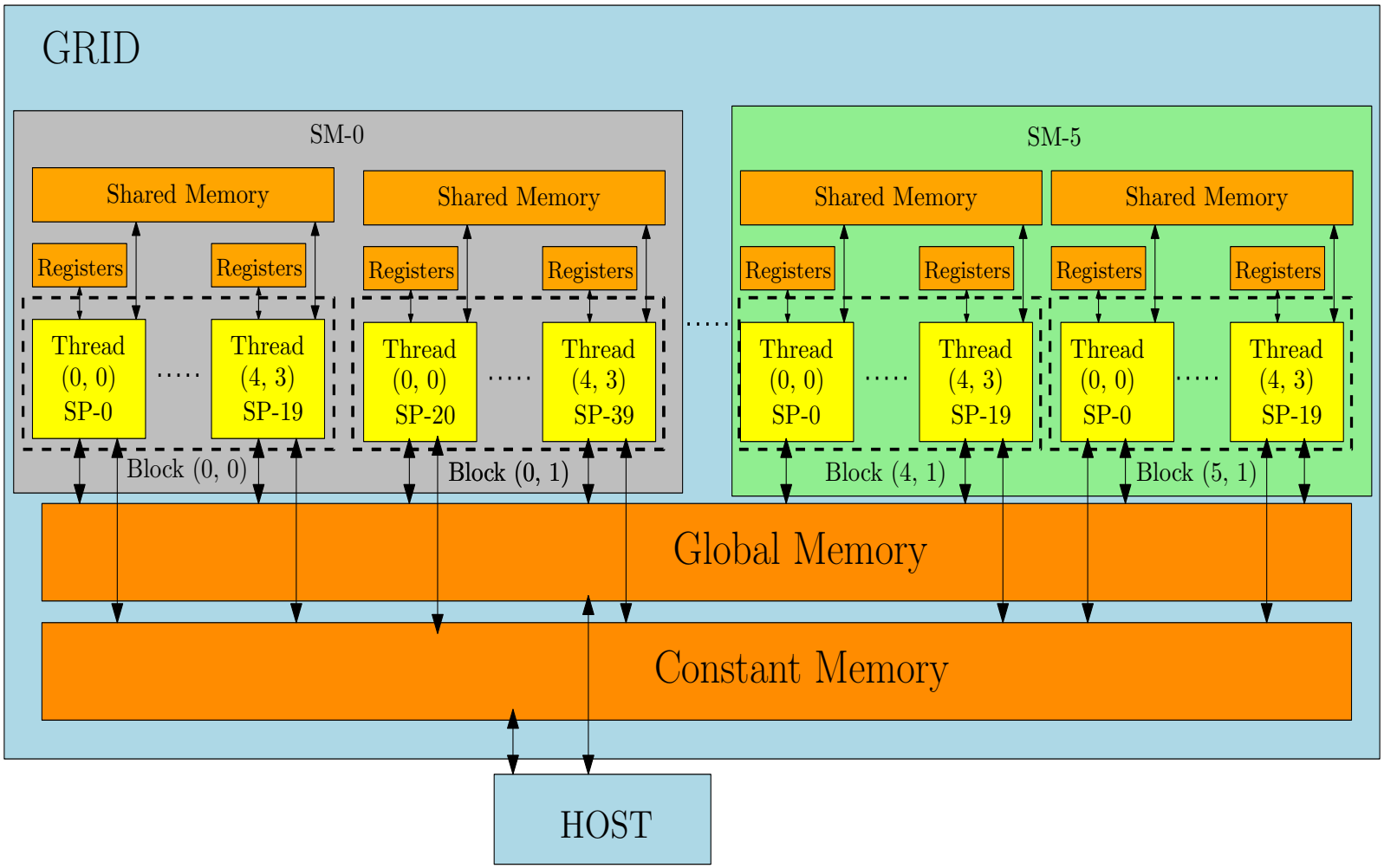


Figure: Mapping Kernel Grids to Architecture

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.

Mapping to Hardware

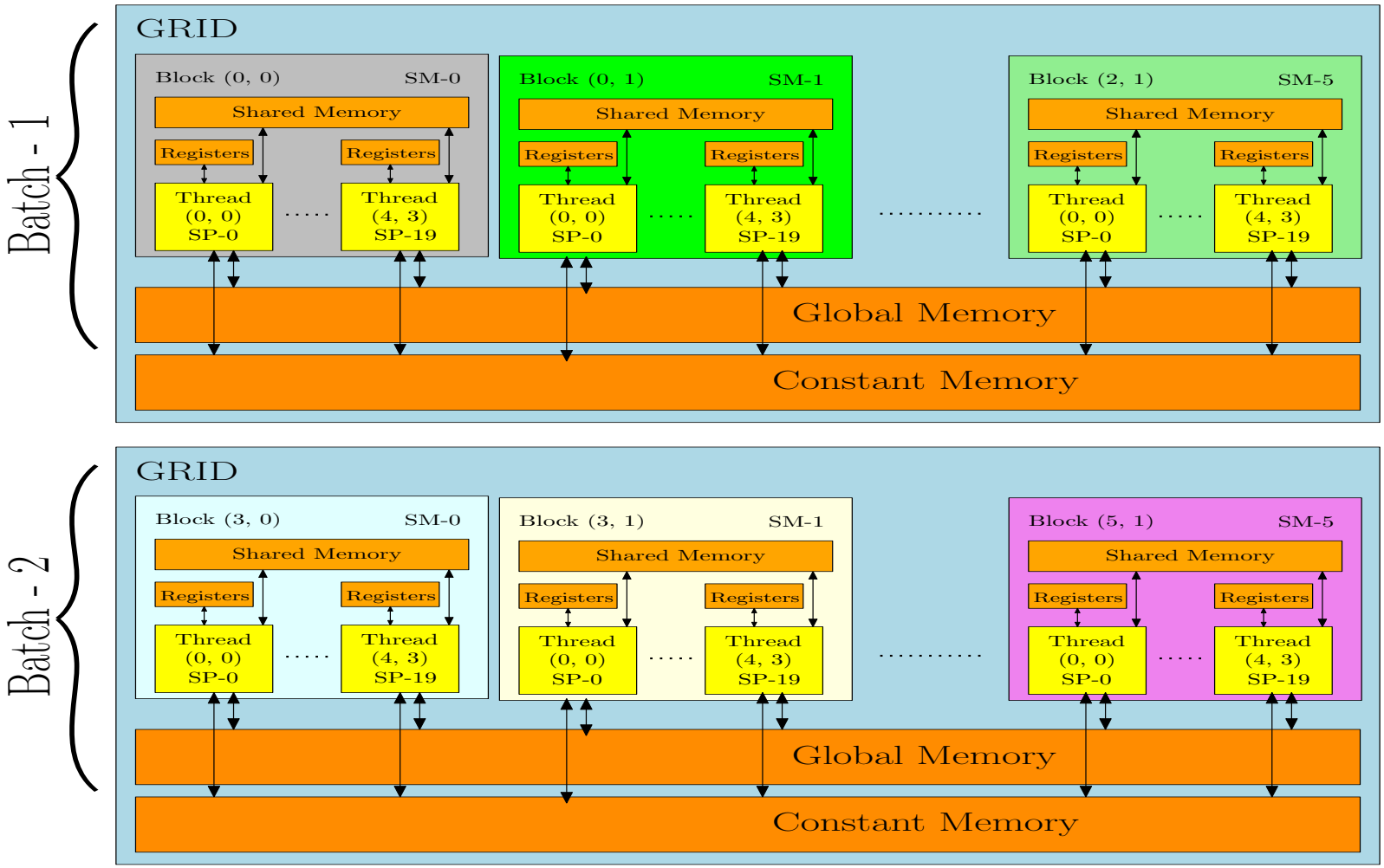


Figure: Mapping Kernel Grids

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

SM, SP, Block and thread

- 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"

SM: A closer look

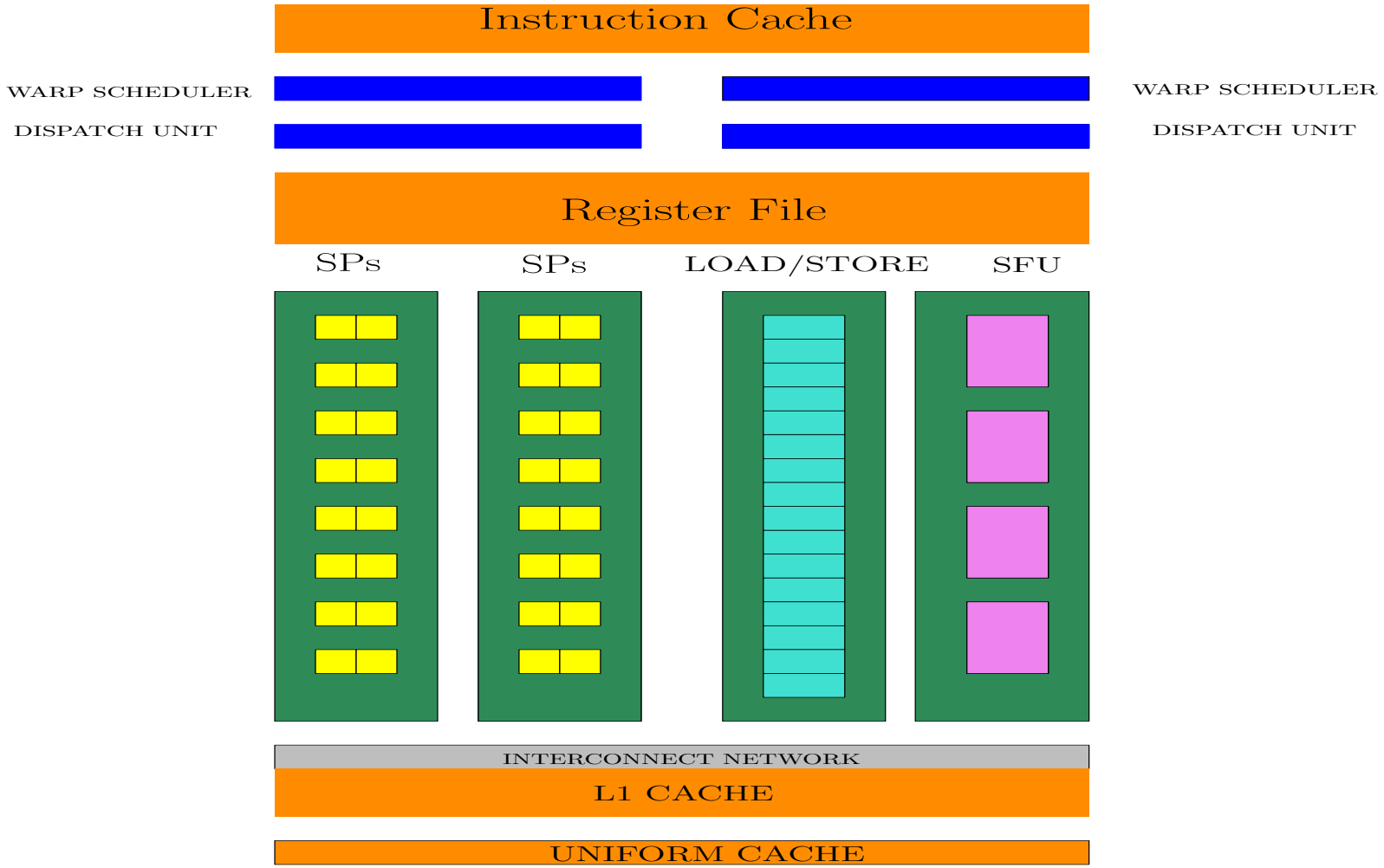


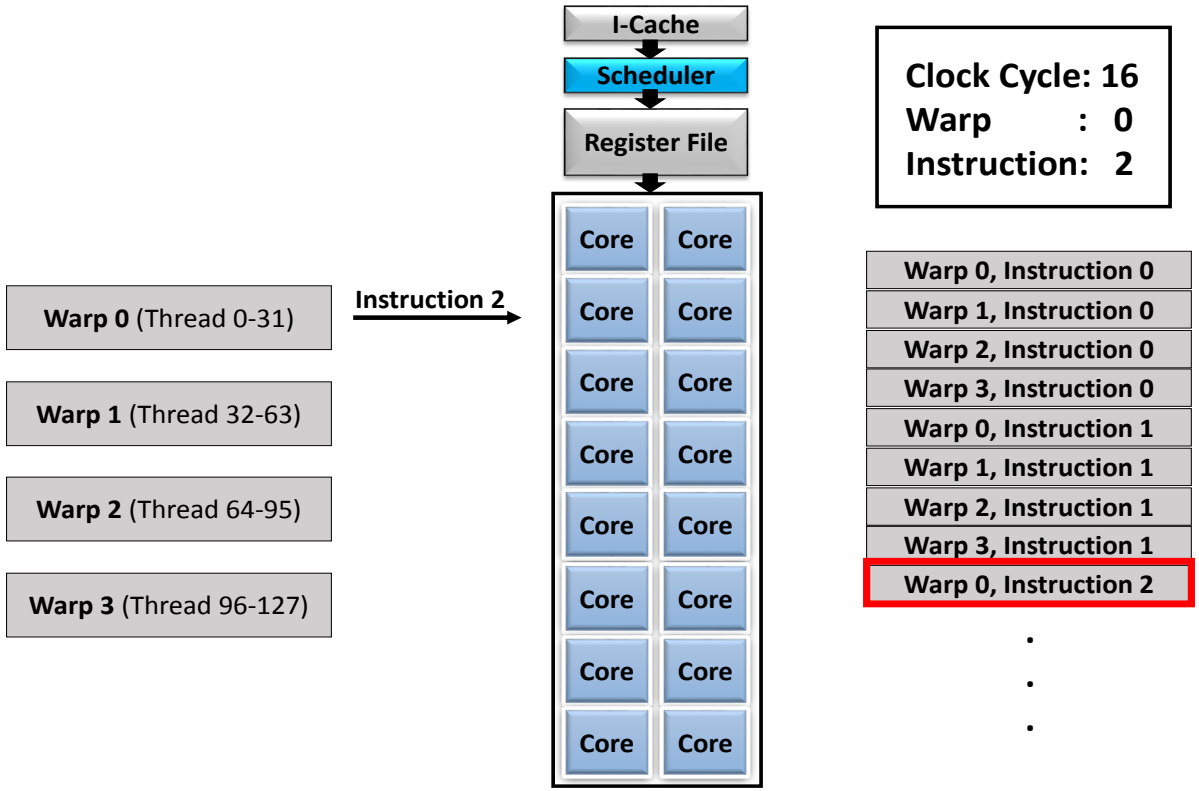
Figure: Streaming Multiprocessor

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.

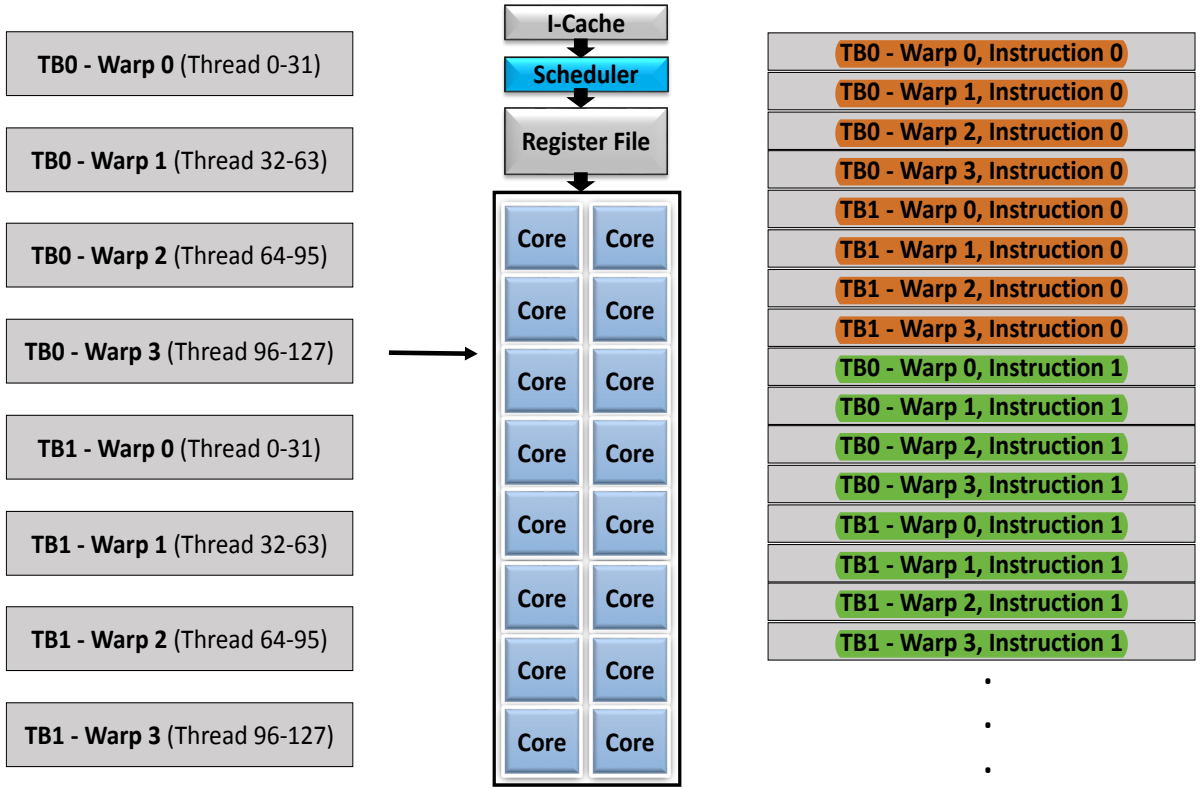
Ex : If a SM has 128 SPs, it can execute 4 Warps at a given time (one Warp has 32 Threads)

Warp Scheduling in SM



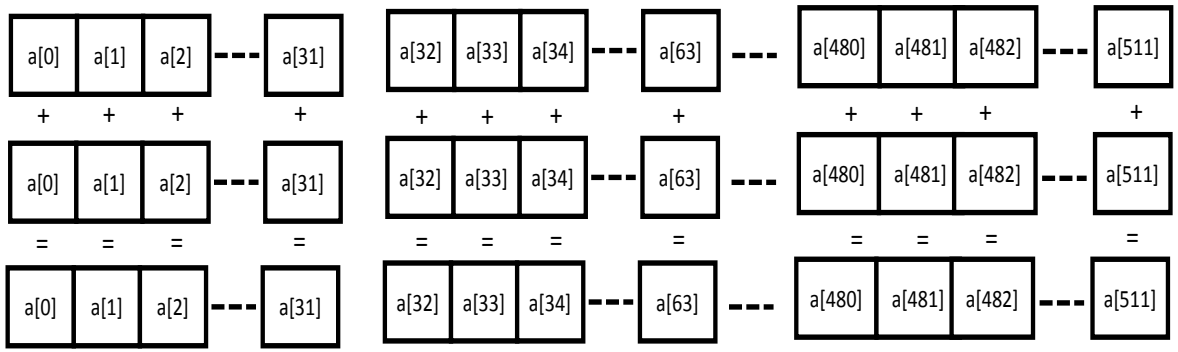
Ref : Henk Corporaal, Gert-Jan van den Braak - "Introduction to GPGPU Architectures"

Warp Scheduling in SM



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

Warp Scheduling in SM




```
__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

Warp Scheduling in SM

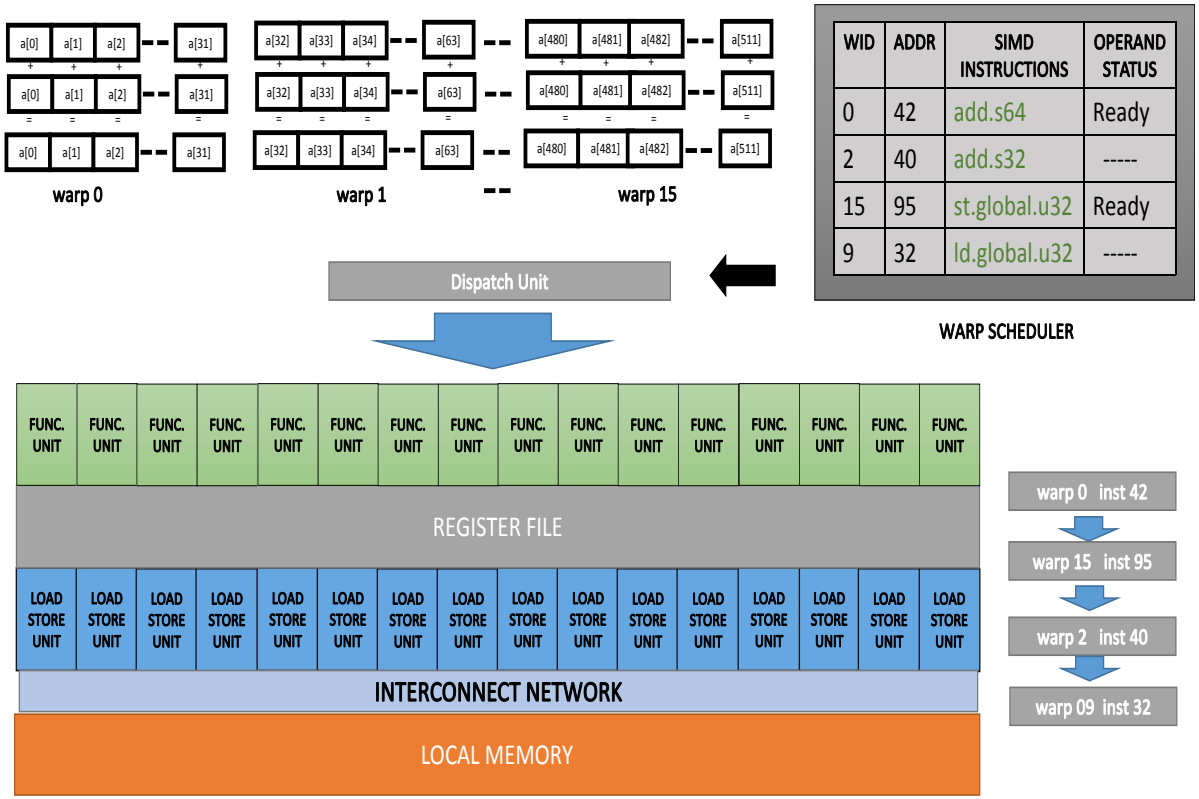


Figure: Warp Scheduler

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.

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)

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

Querying Device Properties

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;
}
```

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]);
```

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;
}
```

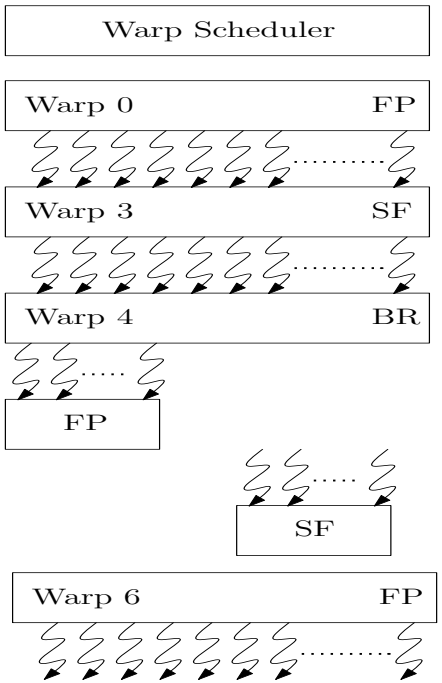
Example: Tesla K40m Characteristics

```
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
```

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.

Warp Scheduling



Threads inside a warp executing same instruction - efficient

Threads inside a warp executing different instruction - inefficient

Figure: Warp Divergence

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;
}
```

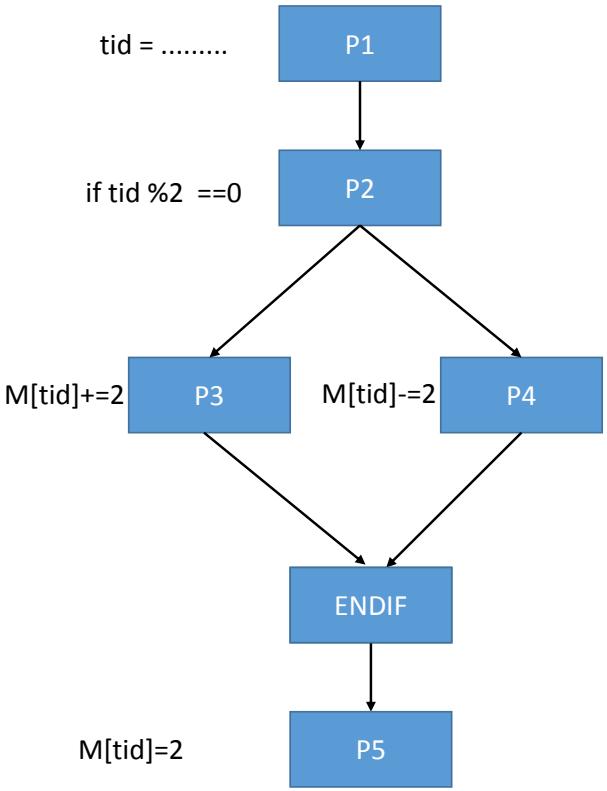
Half the threads of a warp execute the addition instruction while the other half execute the subtraction instruction.

The Hardware's Job

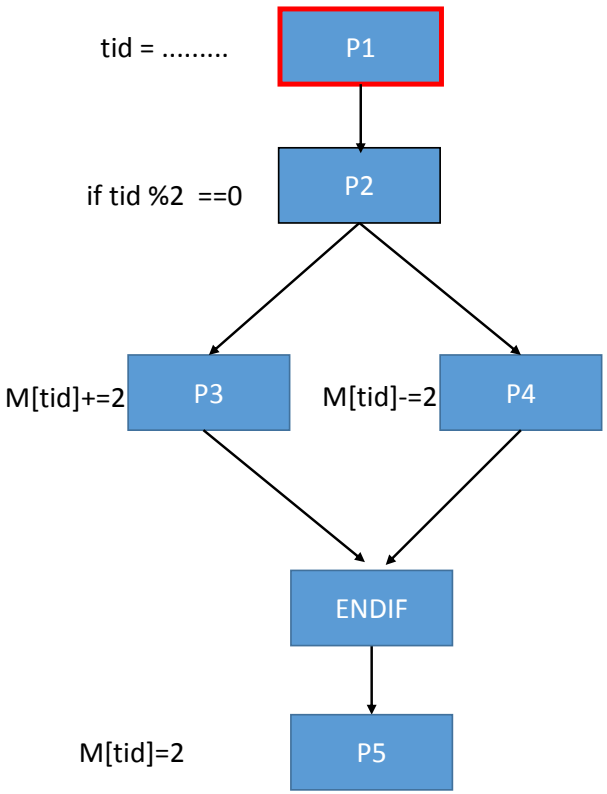
The GPU has hardware support for handling divergent branch instructions in code.

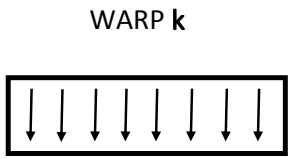
- The PTX Assembler maintains internal masks, a branch synchronization stack and special markers
- The PTX Assembler sets a **branch synchronization marker** first for the divergent `if` statement that pushes the active mask on a stack inside each SIMD thread
- Depending on the value of the mask relevant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

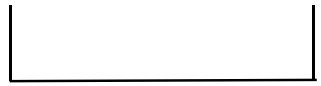
Divergent Code 1



Divergent Code 1



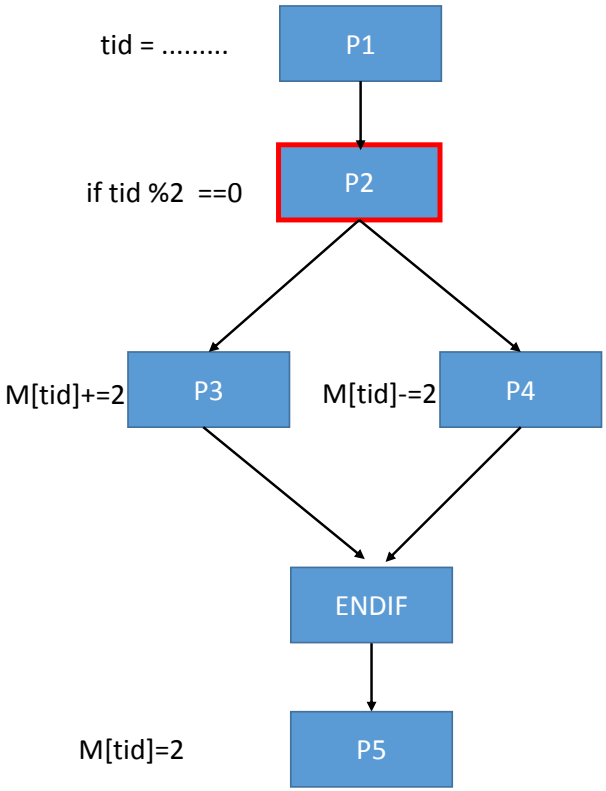


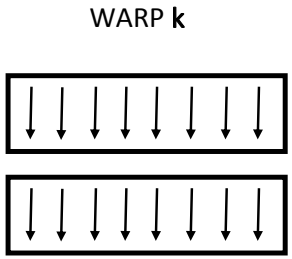


RECONVERGE TARGET

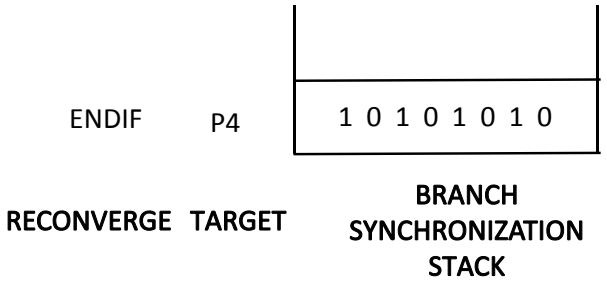
BRANCH SYNCHRONIZATION STACK

Divergent Code 1

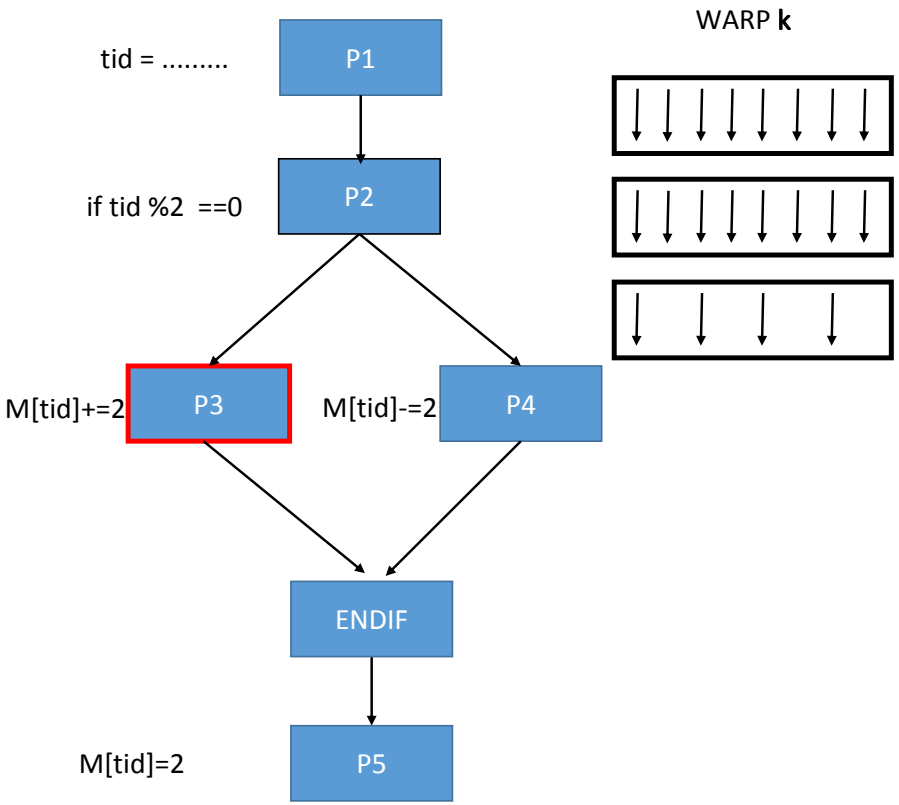


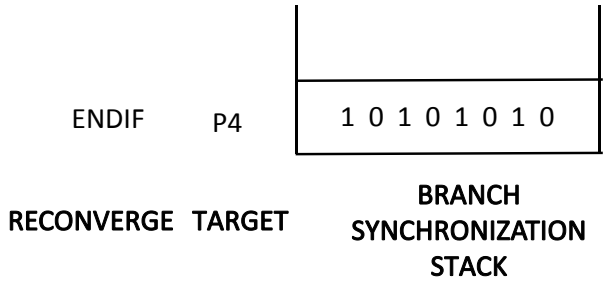


INITIALIZE ACTIVE MASK

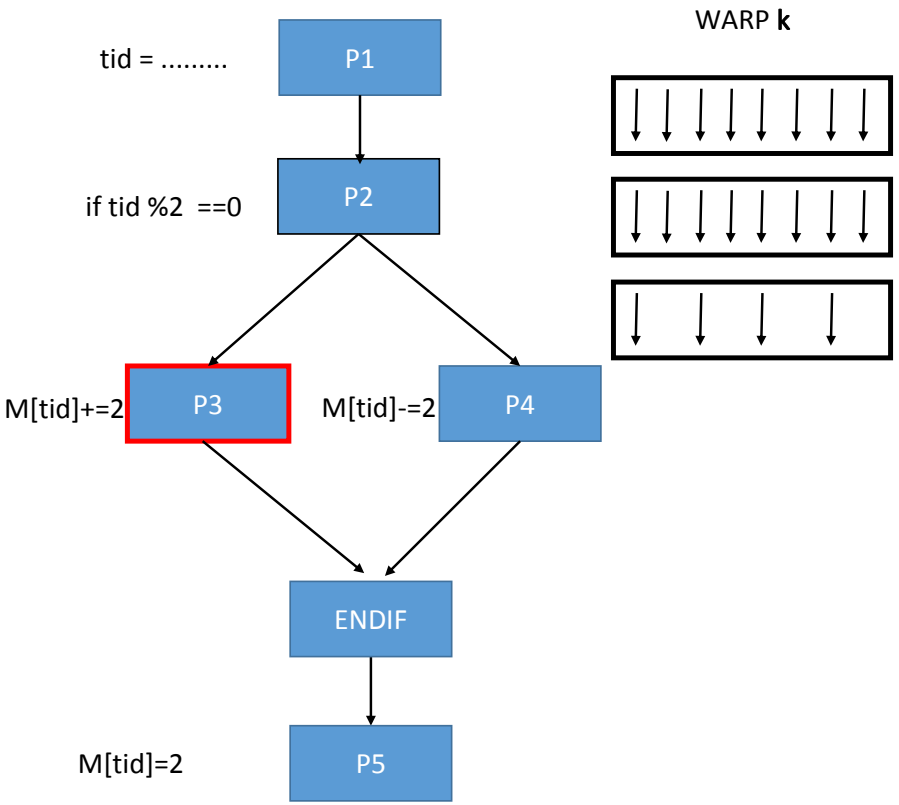


Divergent Code 1



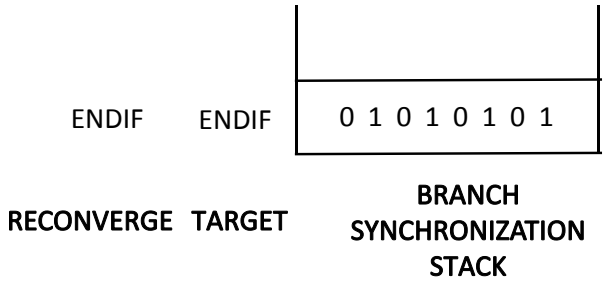


Divergent Code 1

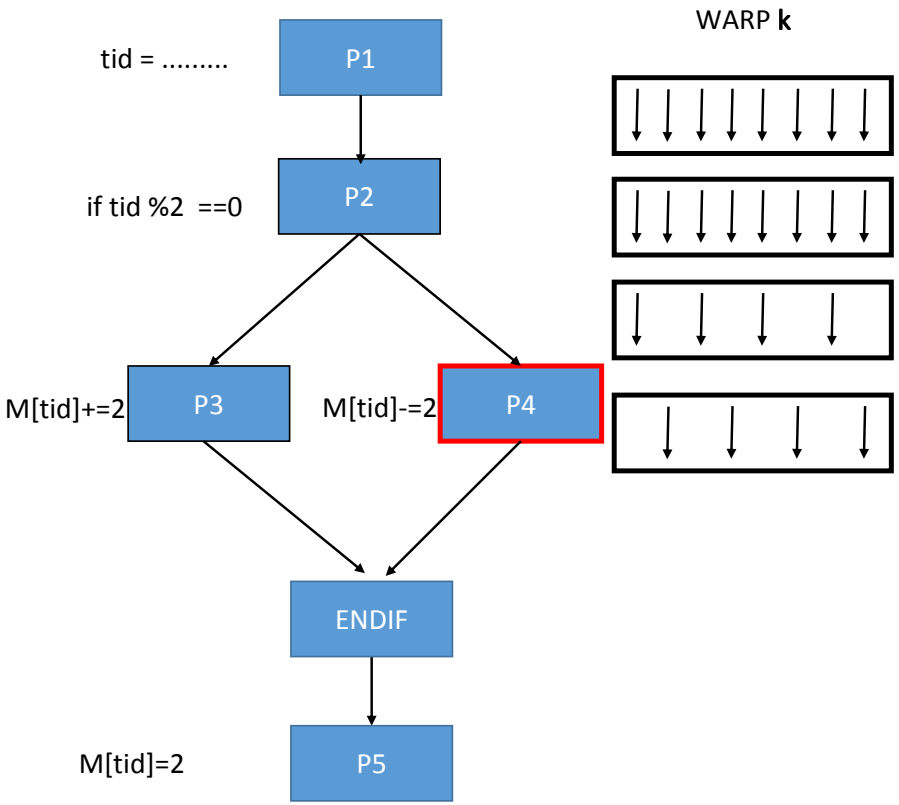


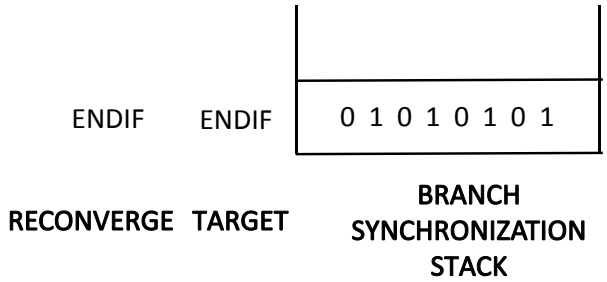
COMPLEMENT SET MASK BITS

SET TARGET

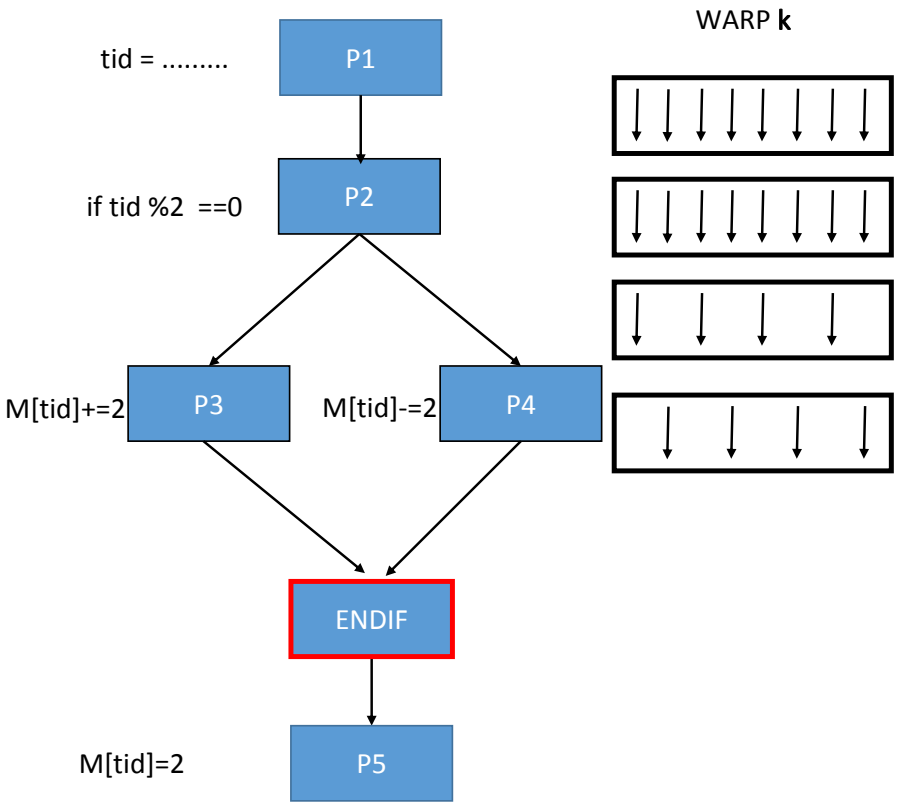


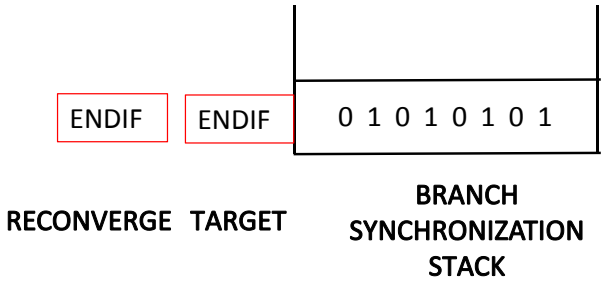
Divergent Code 1



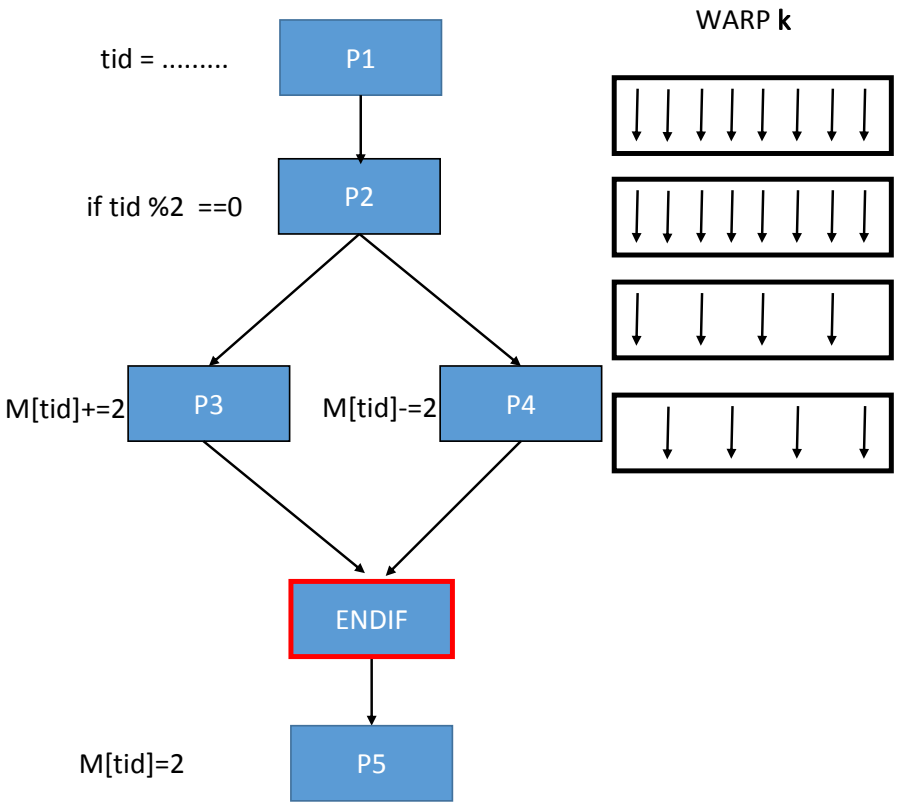


Divergent Code 1





Divergent Code 1

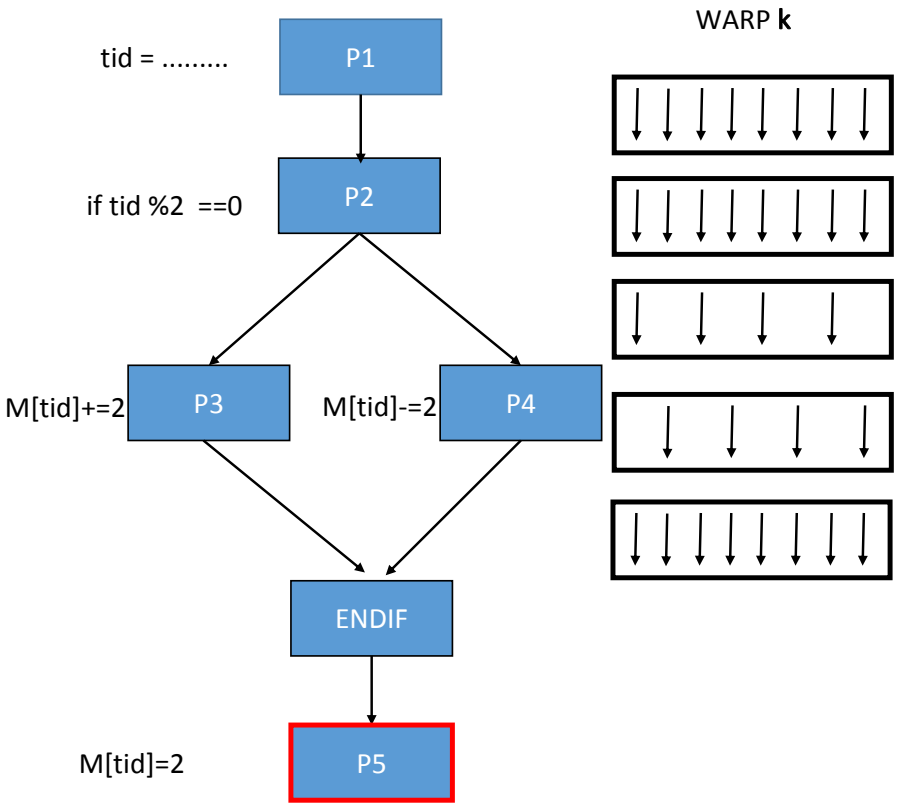


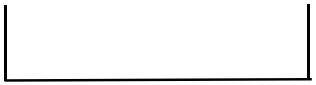
POP MASK

RECONVERGE TARGET

BRANCH SYNCHRONIZATION STACK

Divergent Code 1





RECONVERGE TARGET

BRANCH SYNCHRONIZATION STACK

Observations

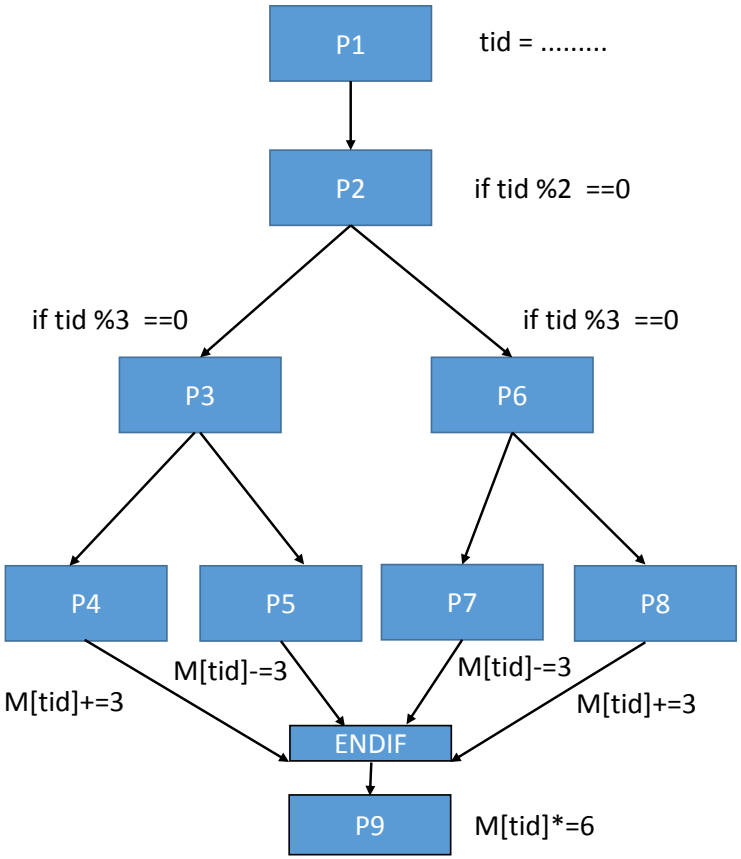
- The target value represents the address of the instruction to be executed by the warp, once the current conditional block of instructions has finished.
- The reconvergence value represents the address of the convergence statement, once both the conditional blocks associated with an `if-else` statement has finished execution.
- The mask is popped from the stack once the conditional block is executed both way (which is known from the target and reconvergence marker).

Divergent Code 2

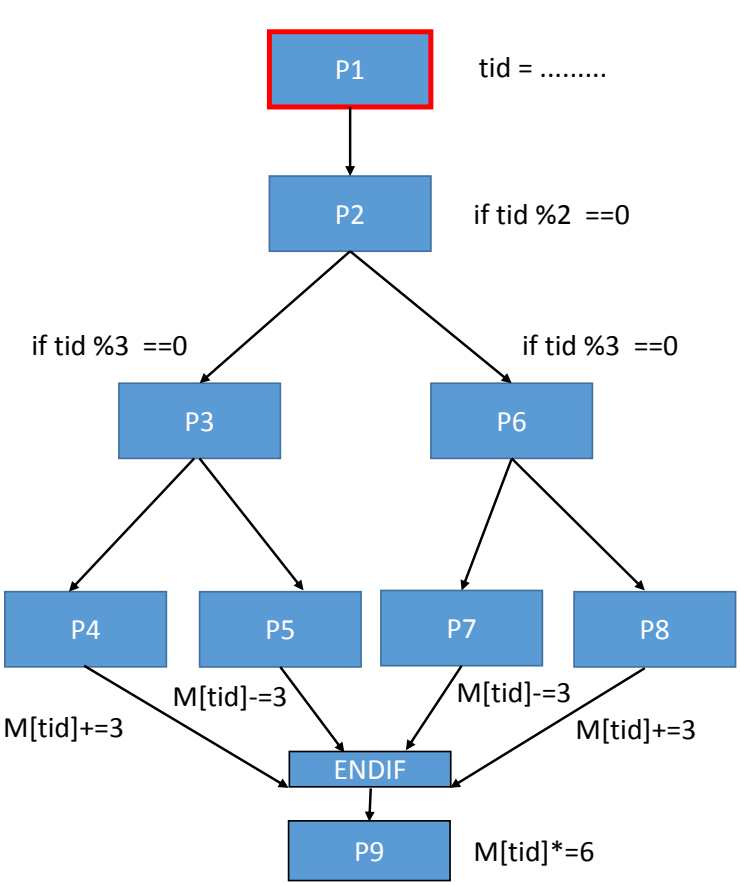
Let us consider an example that has nested if/else statements.

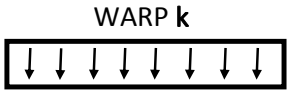
```
__global__
void divergence(float *M)
{
/*P1*/     int tid=blockIdx.x*blockDim.x+threadIdx.x;
/*P2*/     if(tid%2==0)
           {
/*P3*/      if(tid%3==0)
/*P4*/        M[tid]+=3;
            else
/*P5*/        M[tid]-=3;
           }
           else
           {
/*P6*/      if(tid%3==0)
/*P7*/ 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

Divergence Code 2

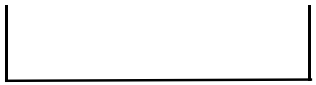


Divergence Code 2



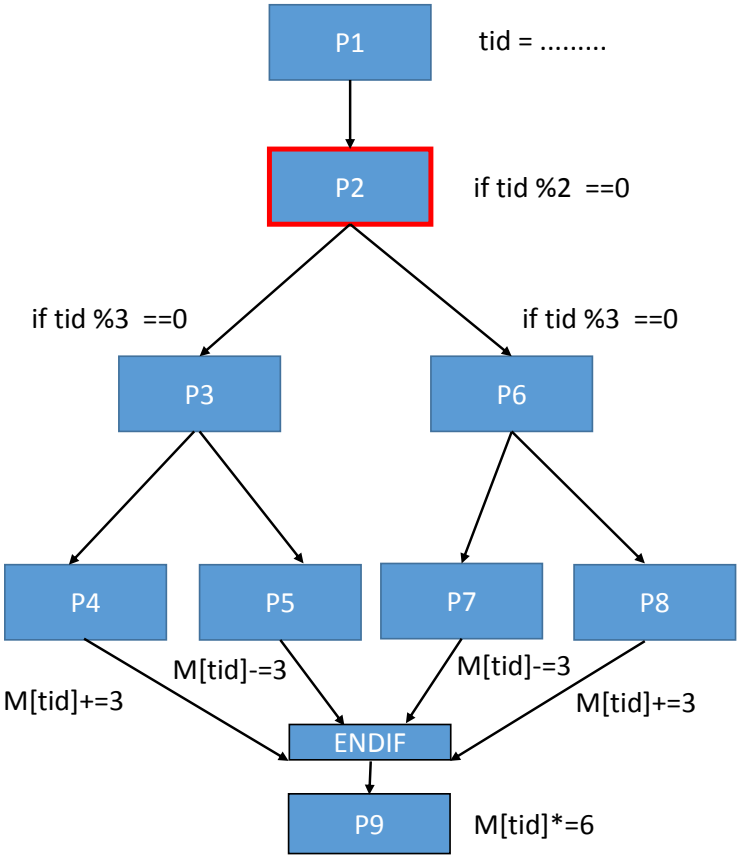


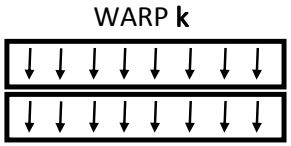
RECONVERGE TARGET



BRANCH SYNCHRONIZATION STACK

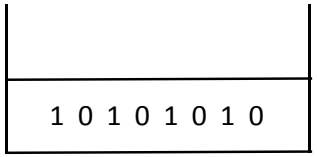
Divergence Code 2





INITIALIZE ACTIVE MASK

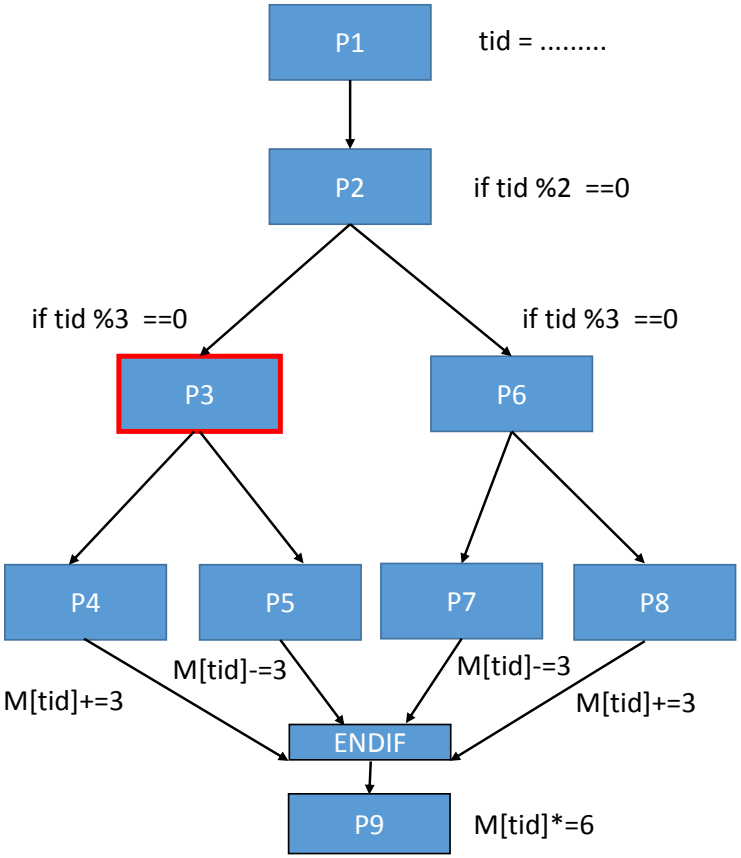
ENDIF P6

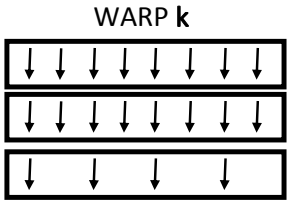


BRANCH SYNCHRONIZATION STACK

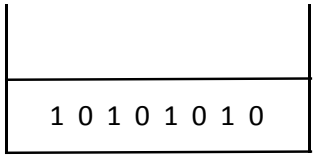
RECONVERGE TARGET

Divergence Code 2



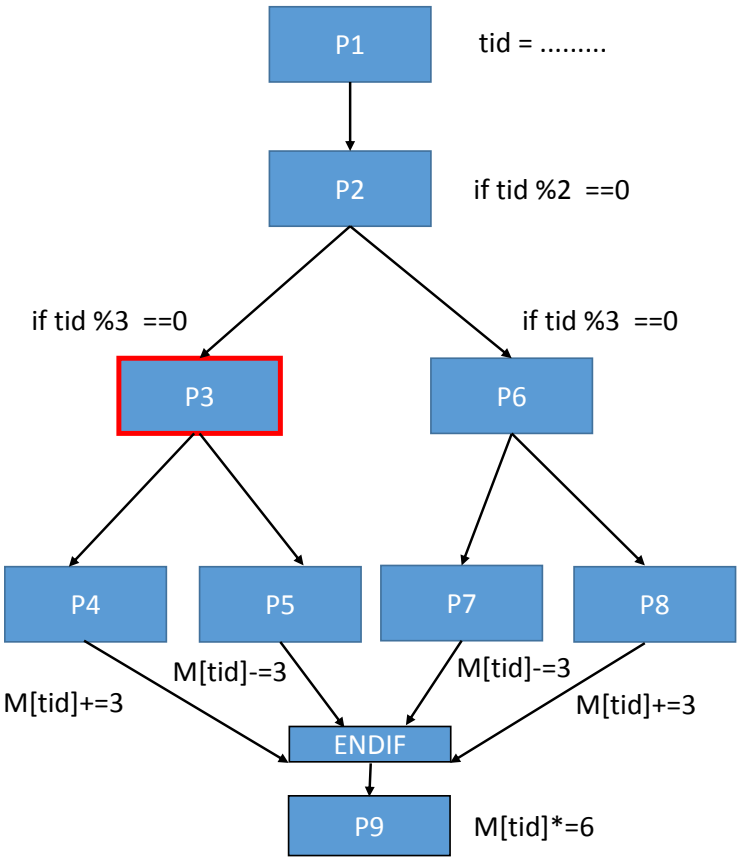


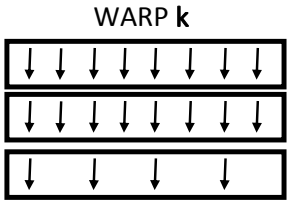
ENDIF P6



RECONVERGE TARGET

Divergence Code 2



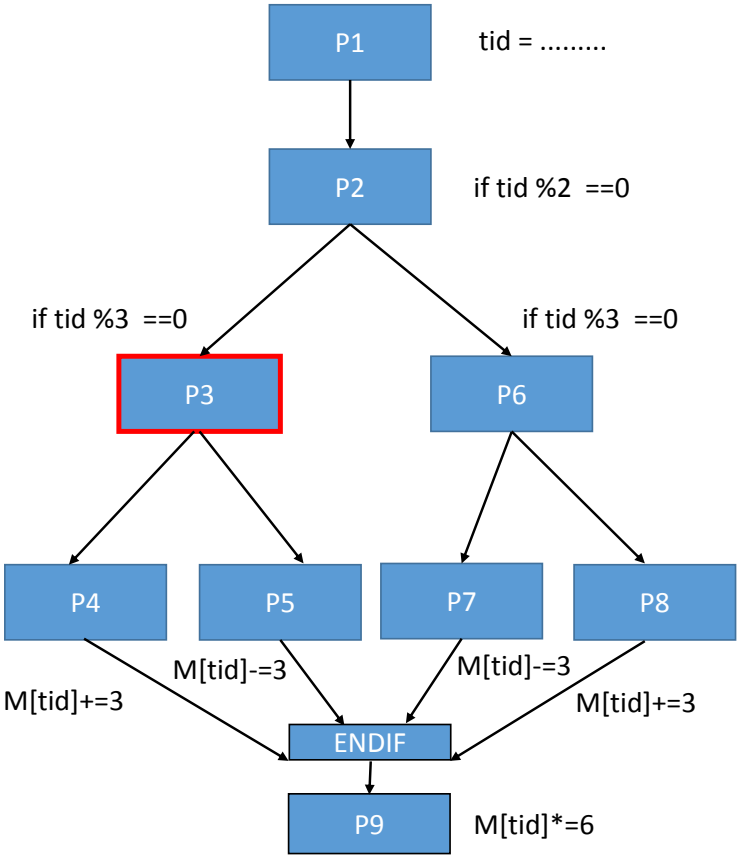


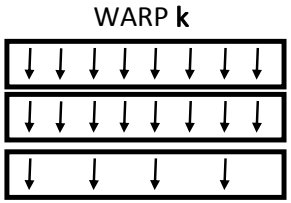
PUSH OLD MASK

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

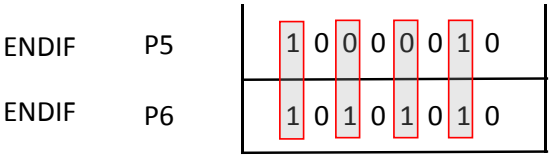
RECONVERGE TARGET

Divergence Code 2



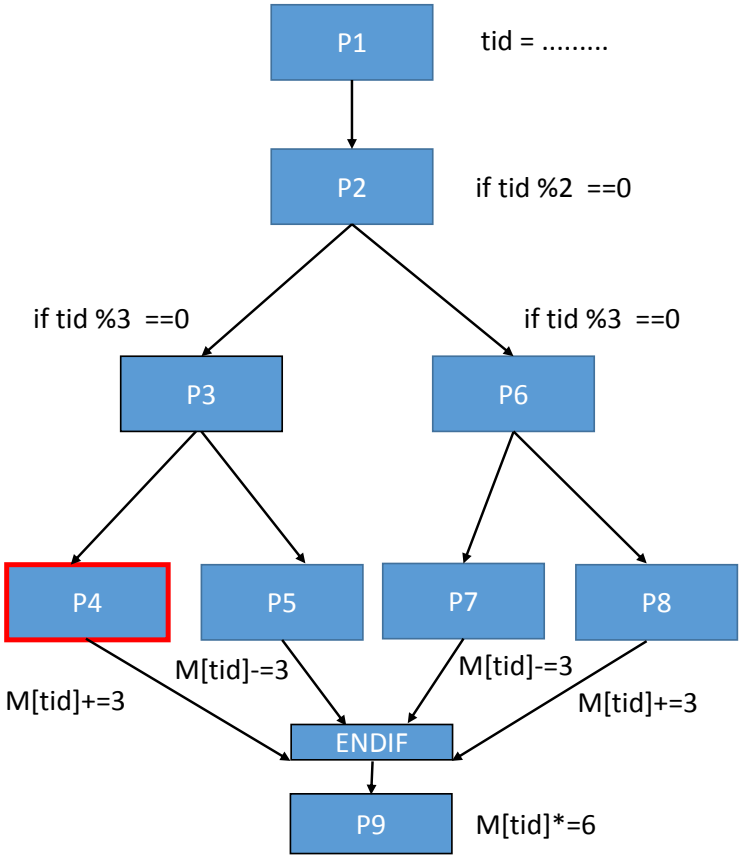


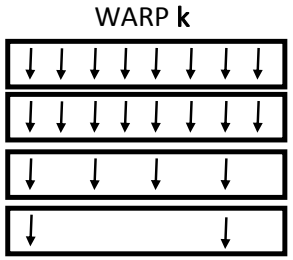
SET RELEVANT BITS

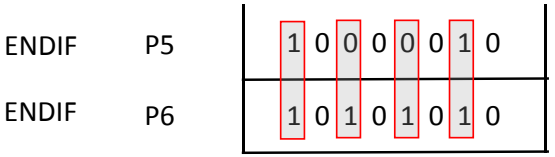


RECONVERGE TARGET

Divergence Code 2

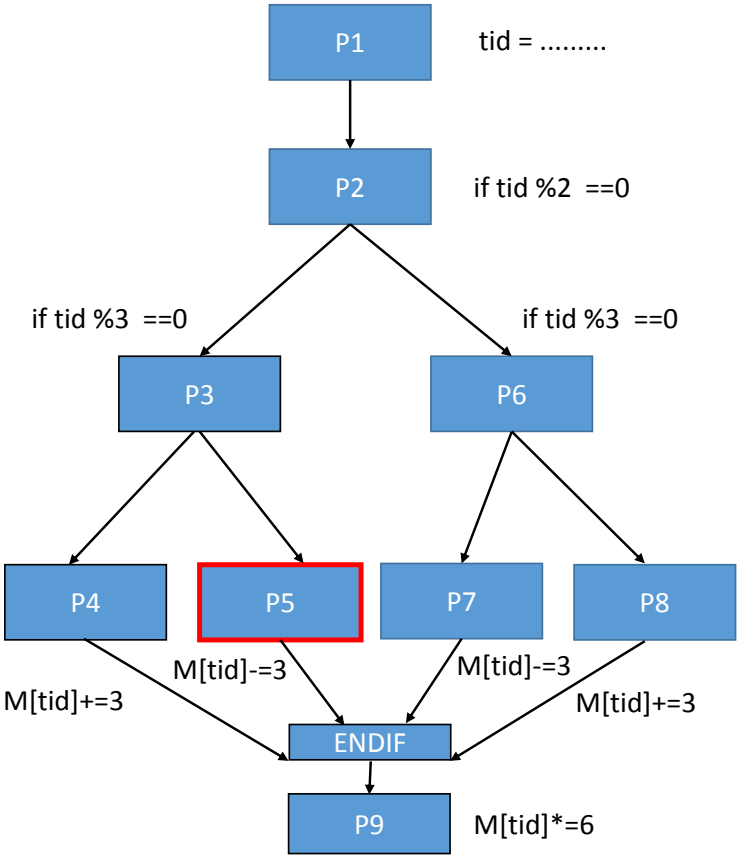


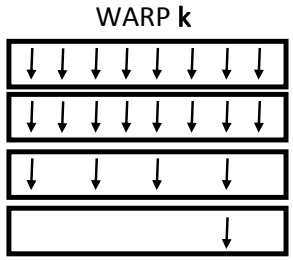


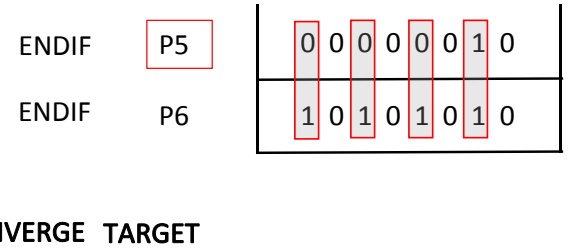


RECONVERGE TARGET

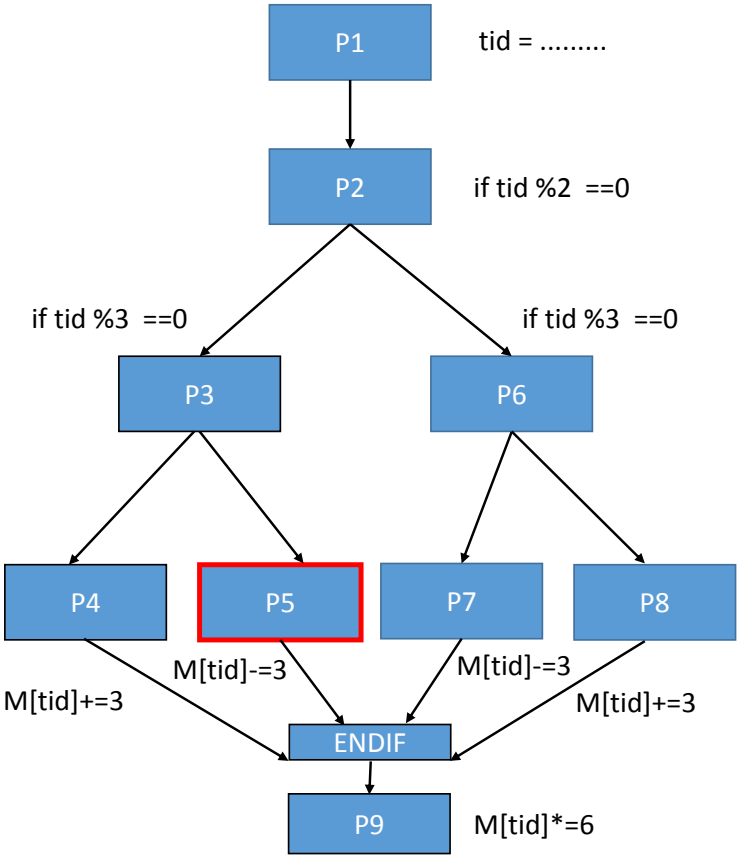
Divergence Code 2

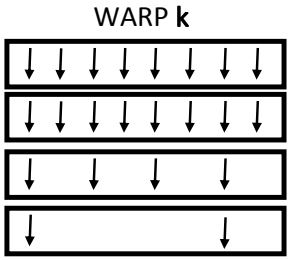






Divergence Code 2

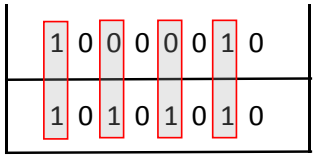




COMPLEMENT RELEVANT BITS

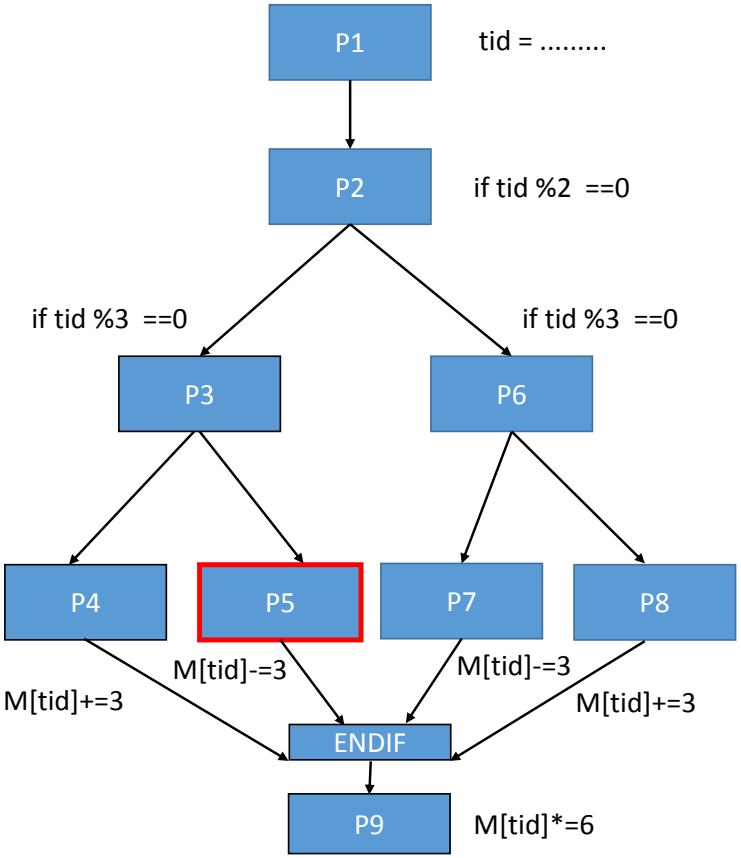
ENDIF P5

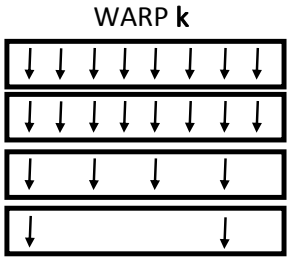
ENDIF P6



RECONVERGE TARGET

Divergence Code 2

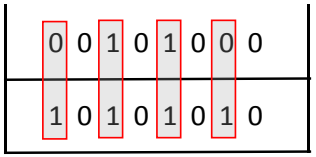




COMPLEMENT RELEVANT BITS

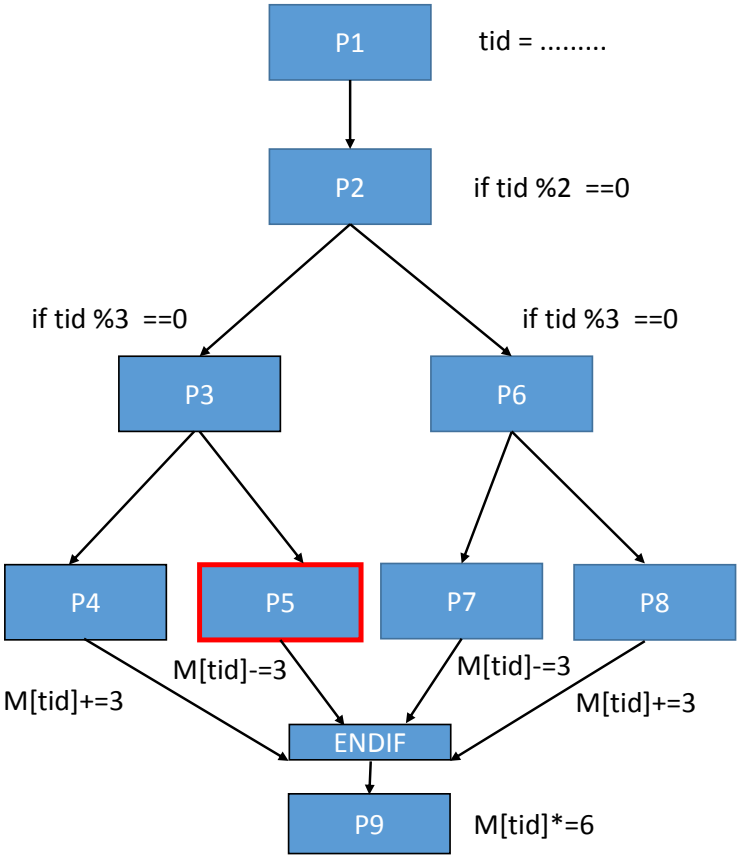
ENDIF ENDIF

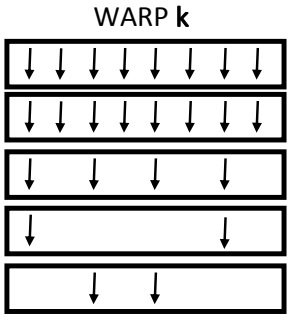
ENDIF P6



RECONVERGE TARGET

Divergence Code 2

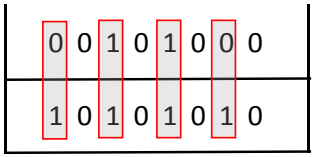




COMPLEMENT RELEVANT BITS

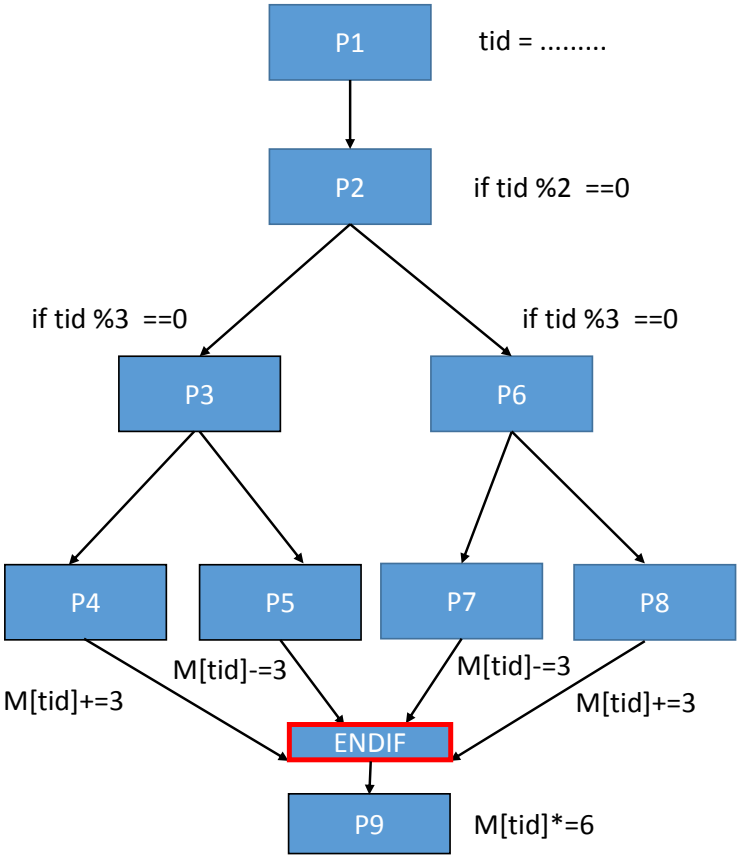
ENDIF ENDIF

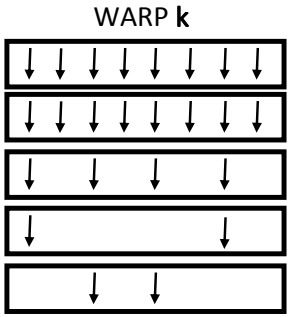
ENDIF P6

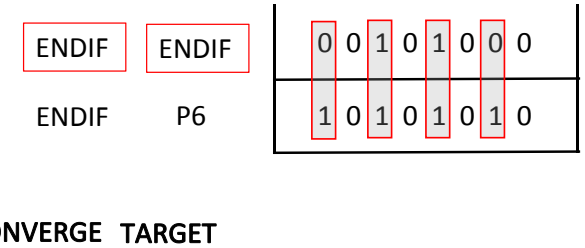


RECONVERGE TARGET

Divergence Code 2

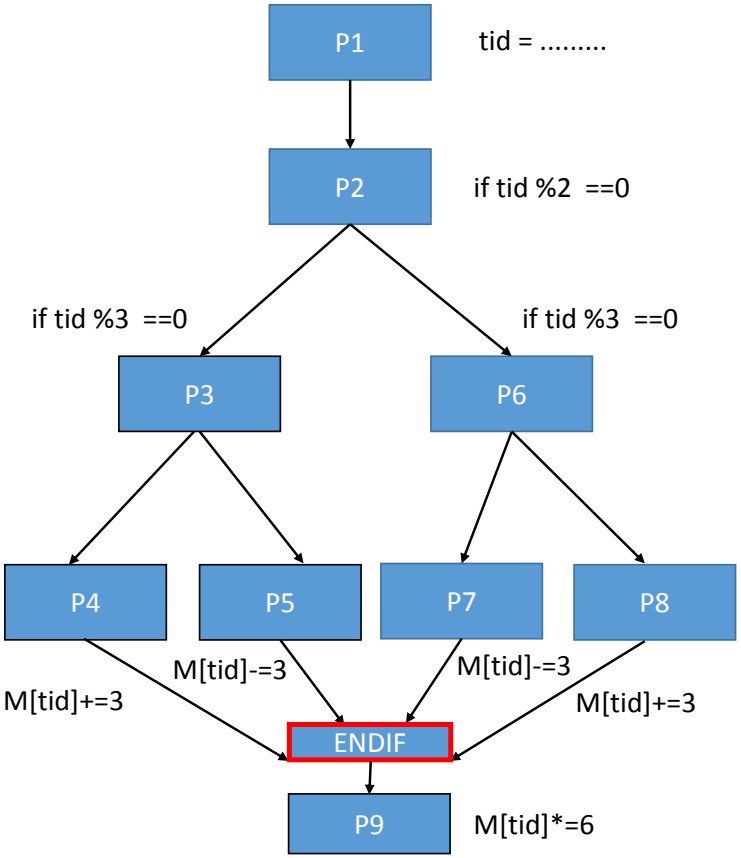


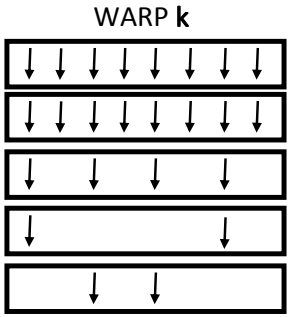




RECONVERGE TARGET

Divergence Code 2



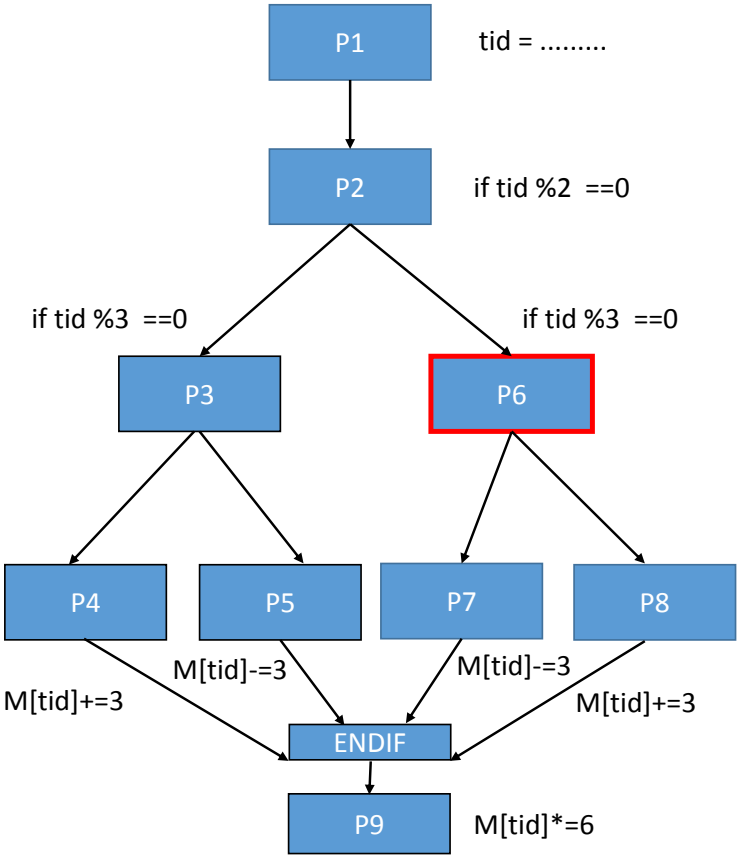


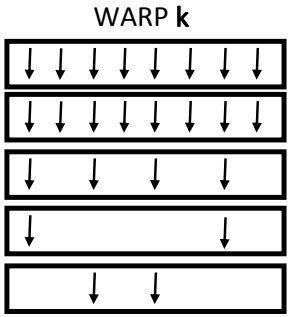
POP STACK

ENDIF P6 | 1 0 1 0 1 0 1 0

RECONVERGE TARGET

Divergence Code 2

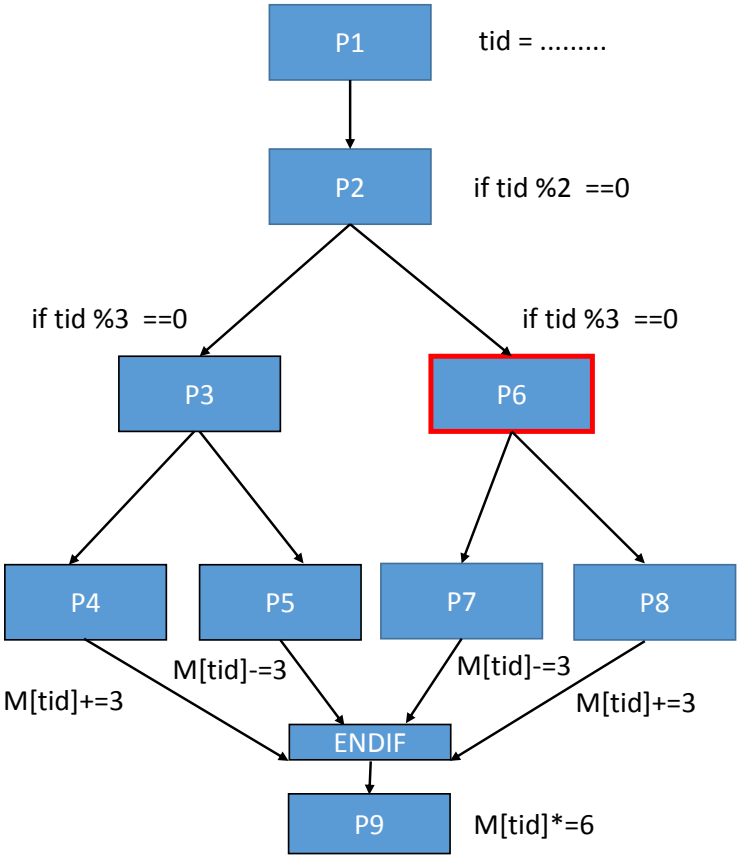


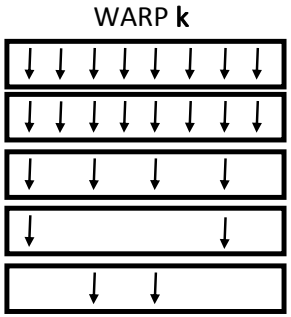


ENDIF P6 1 0 1 0 1 0 1 0

RECONVERGE TARGET

Divergence Code 2





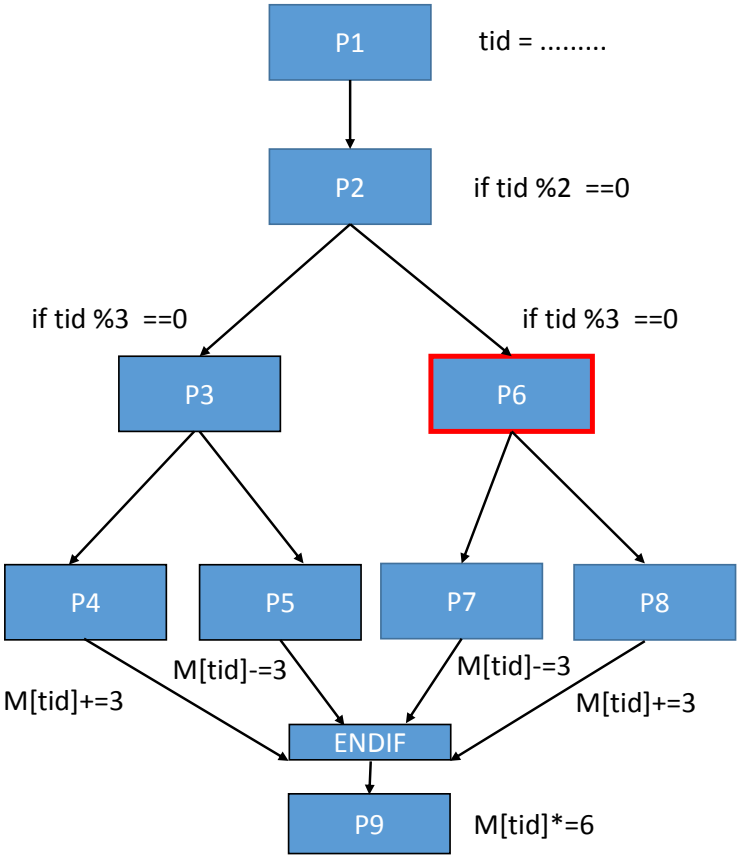
COMPLEMENT RELEVANT BITS

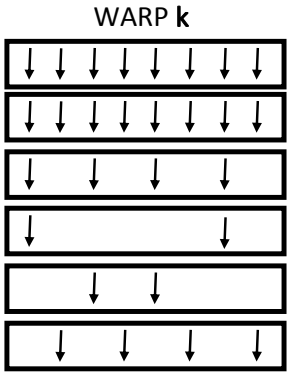
SET TARGET

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

RECONVERGE TARGET

Divergence Code 2

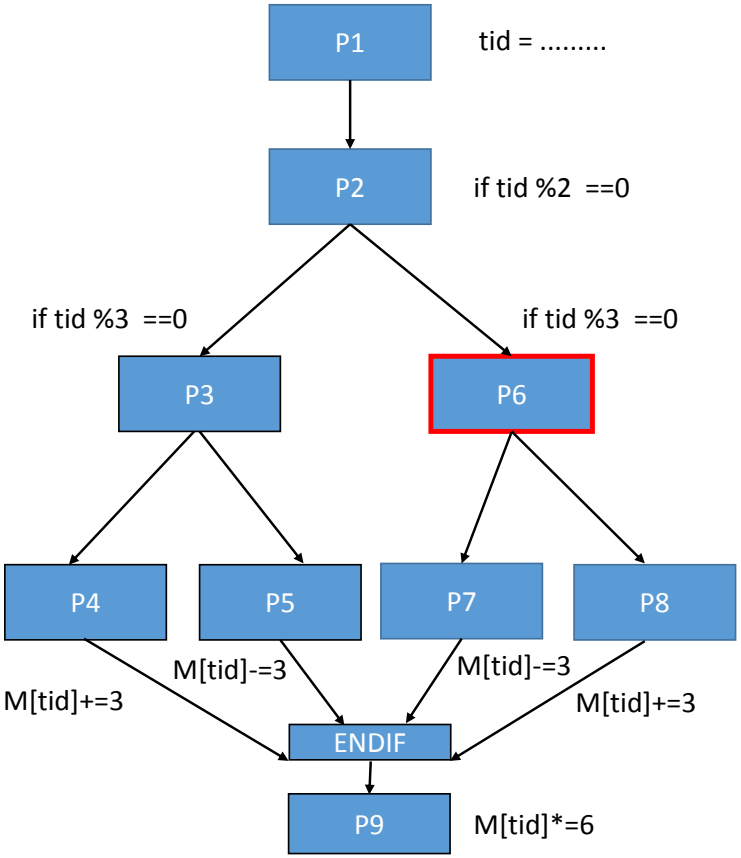


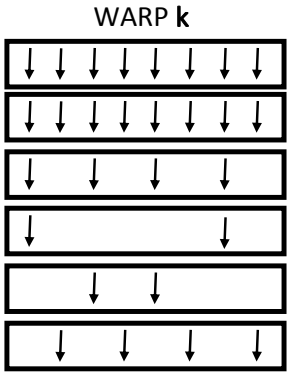


ENDIF ENDIF | 0 1 0 1 0 1 0 1

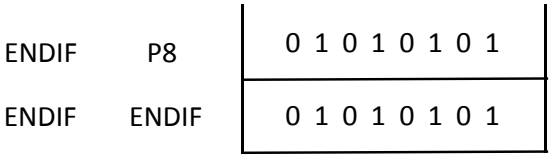
RECONVERGE TARGET

Divergence Code 2



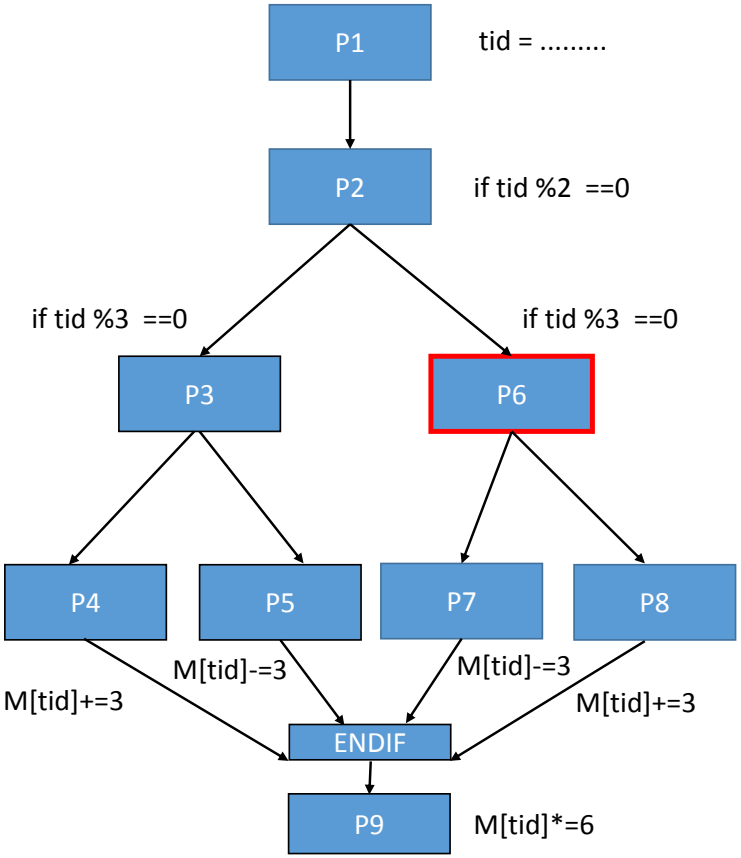


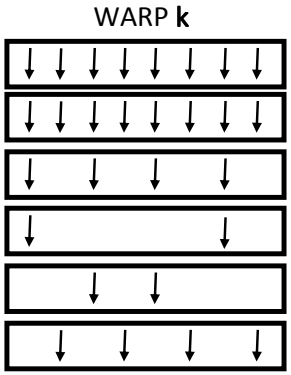
PUSH OLD MASK



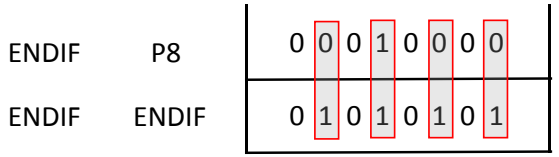
RECONVERGE TARGET

Divergence Code 2



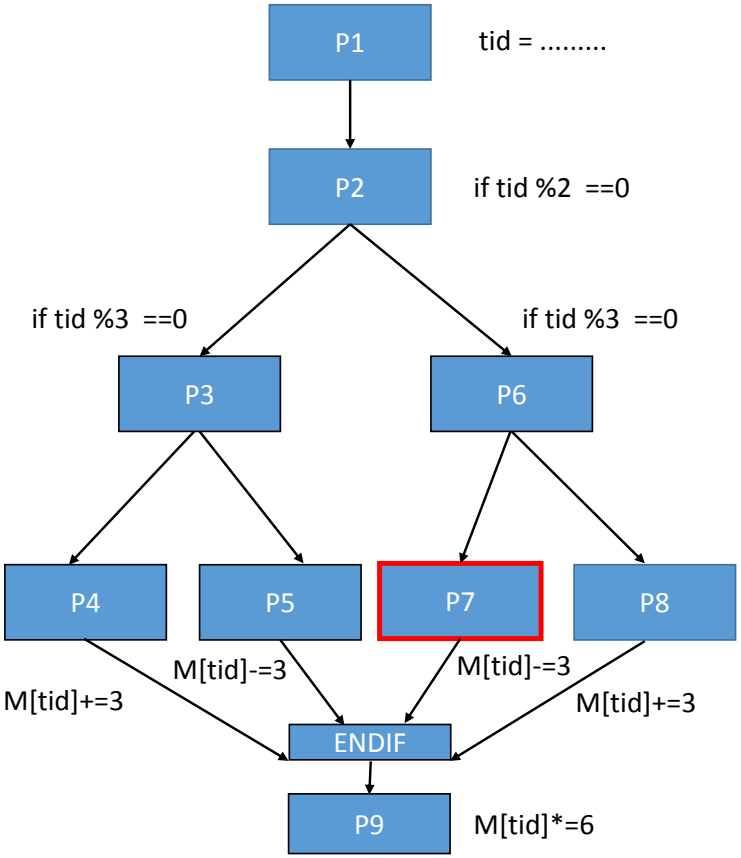


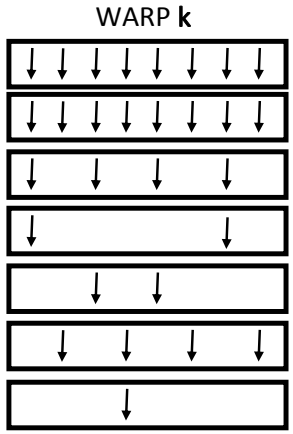
SET RELEVANT BITS

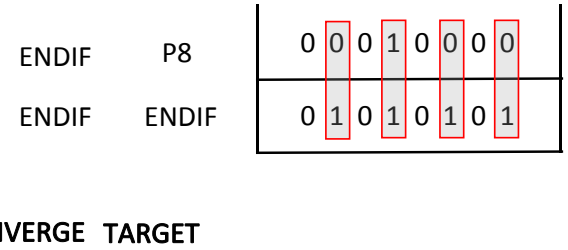


RECONVERGE TARGET

Divergence Code 2

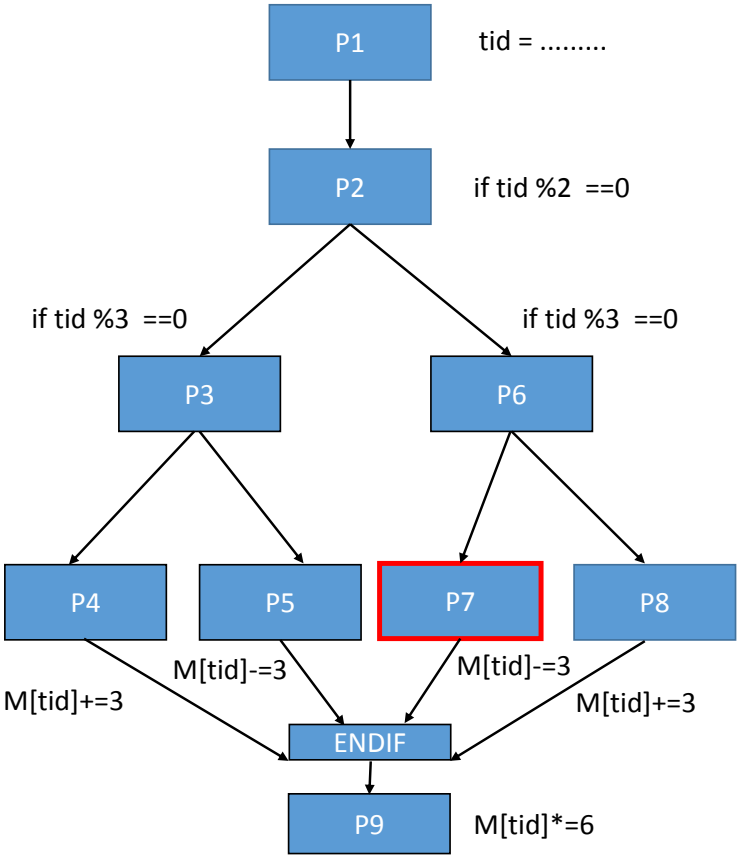


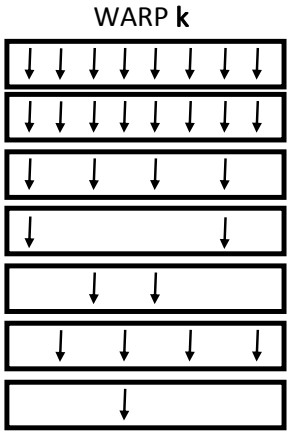




RECONVERGE TARGET

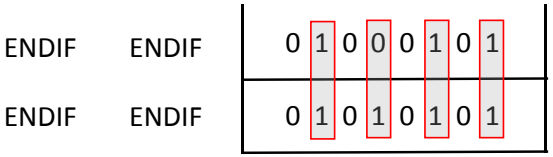
Divergence Code 2





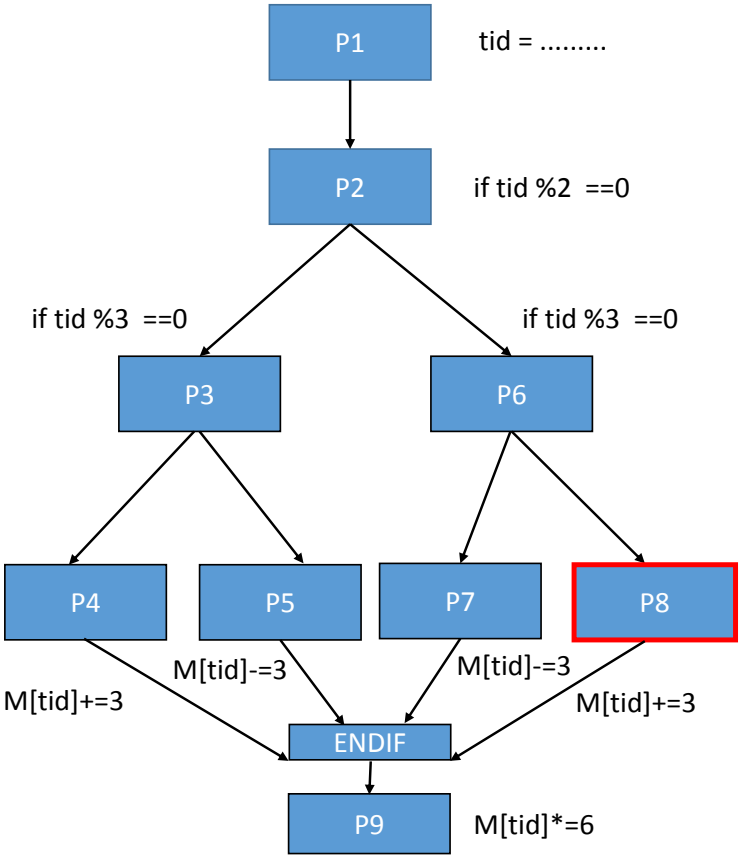
COMPLEMENT RELEVANT BITS

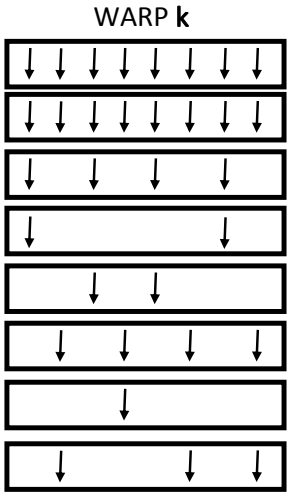
SET TARGET

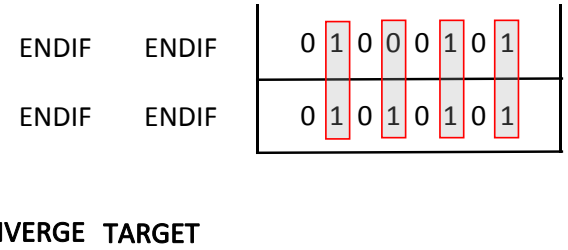


RECONVERGE TARGET

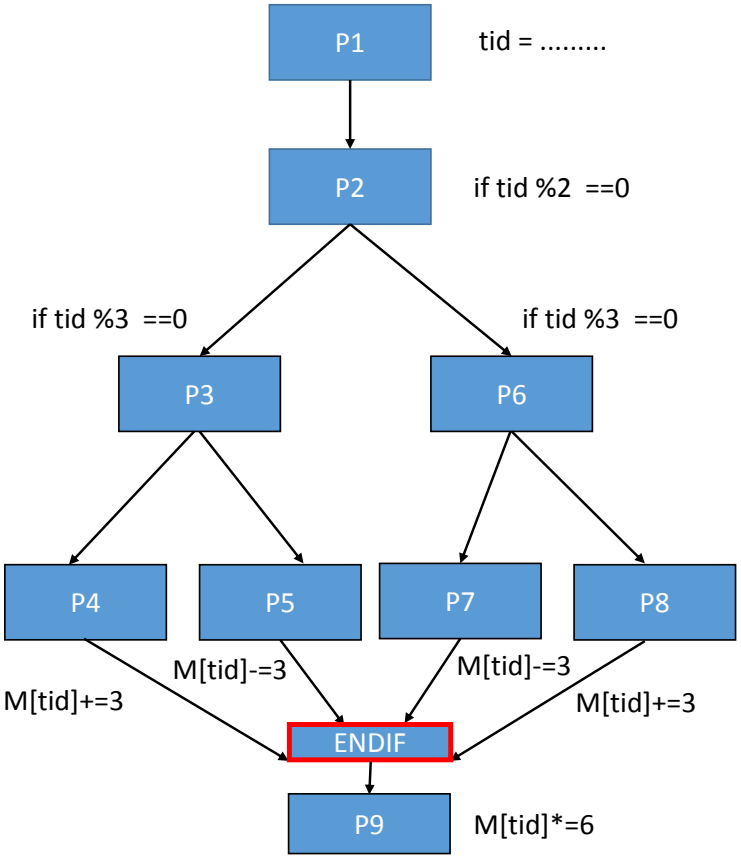
Divergence Code 2

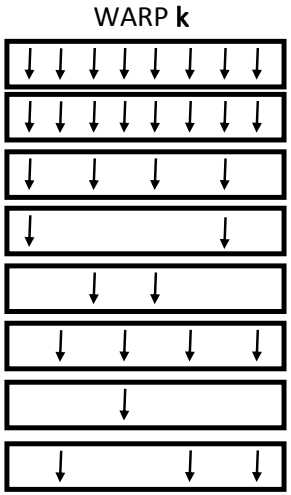


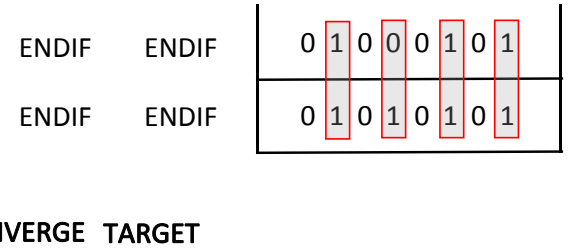




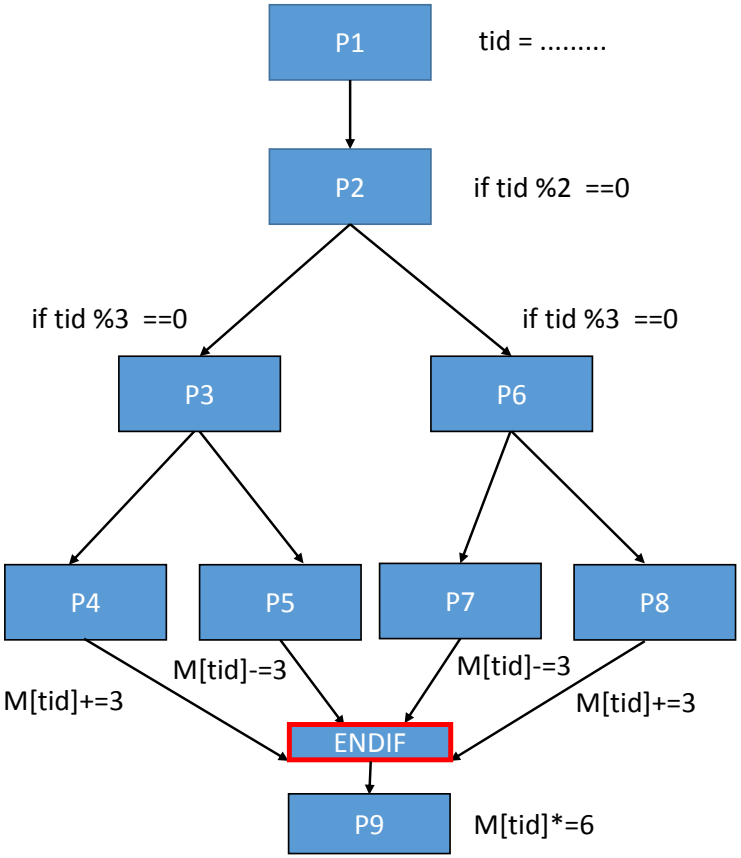
Divergence Code 2

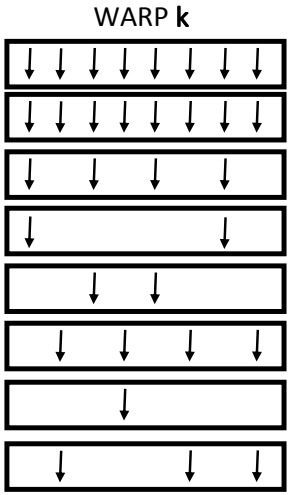




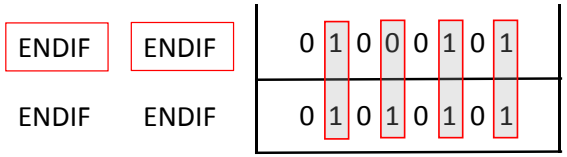


Divergence Code 2



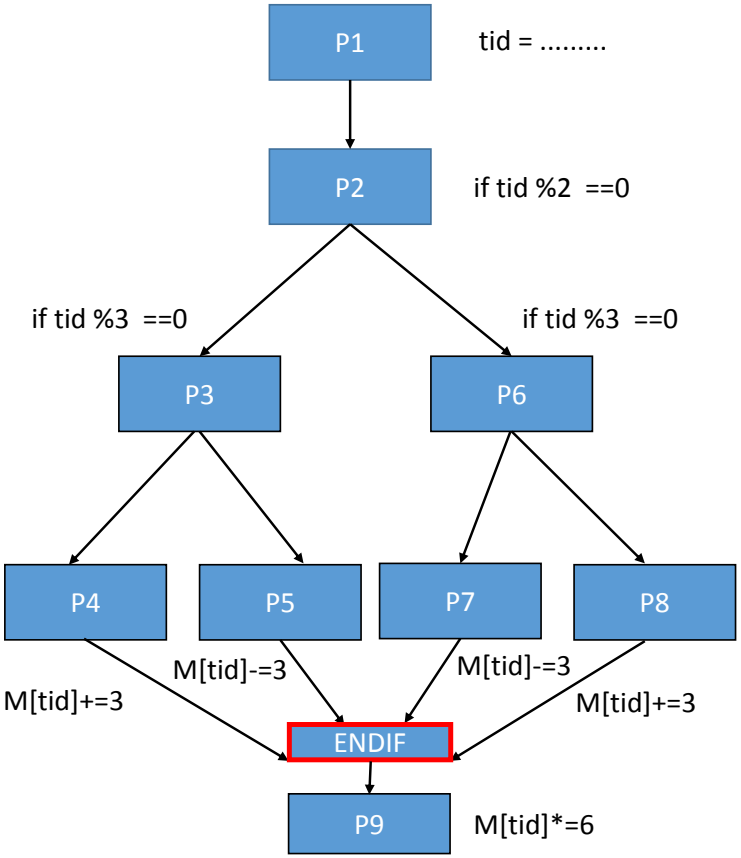


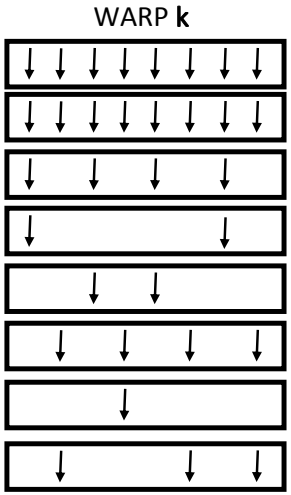
POP STACK



RECONVERGE TARGET

Divergence Code 2



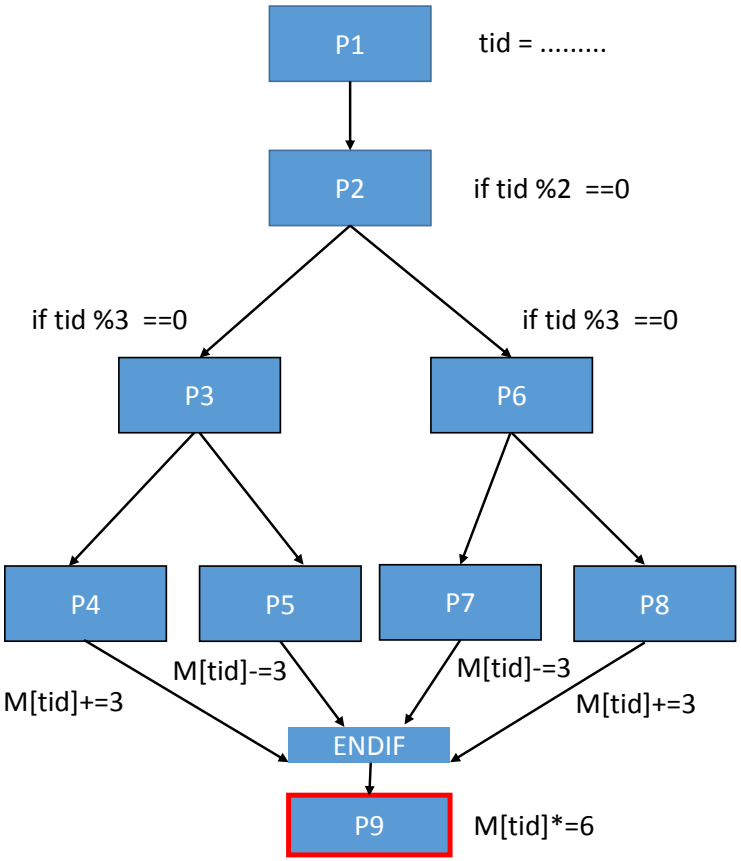


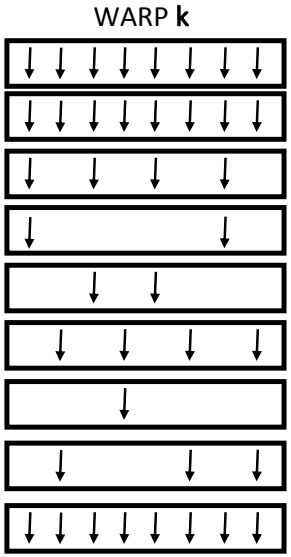
POP STACK AGAIN

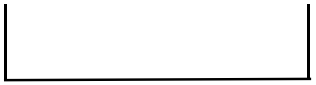
ENDIF ENDIF 0 1 0 1 0 1 0 1

RECONVERGE TARGET

Divergence Code 2







RECONVERGE TARGET

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