CS516: Parallelization of Programs

CUDA Threads Organization

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

2023-24-W

Recap

- CUDA Programming
  - Thread organizations:
  - Examples

Today's outline

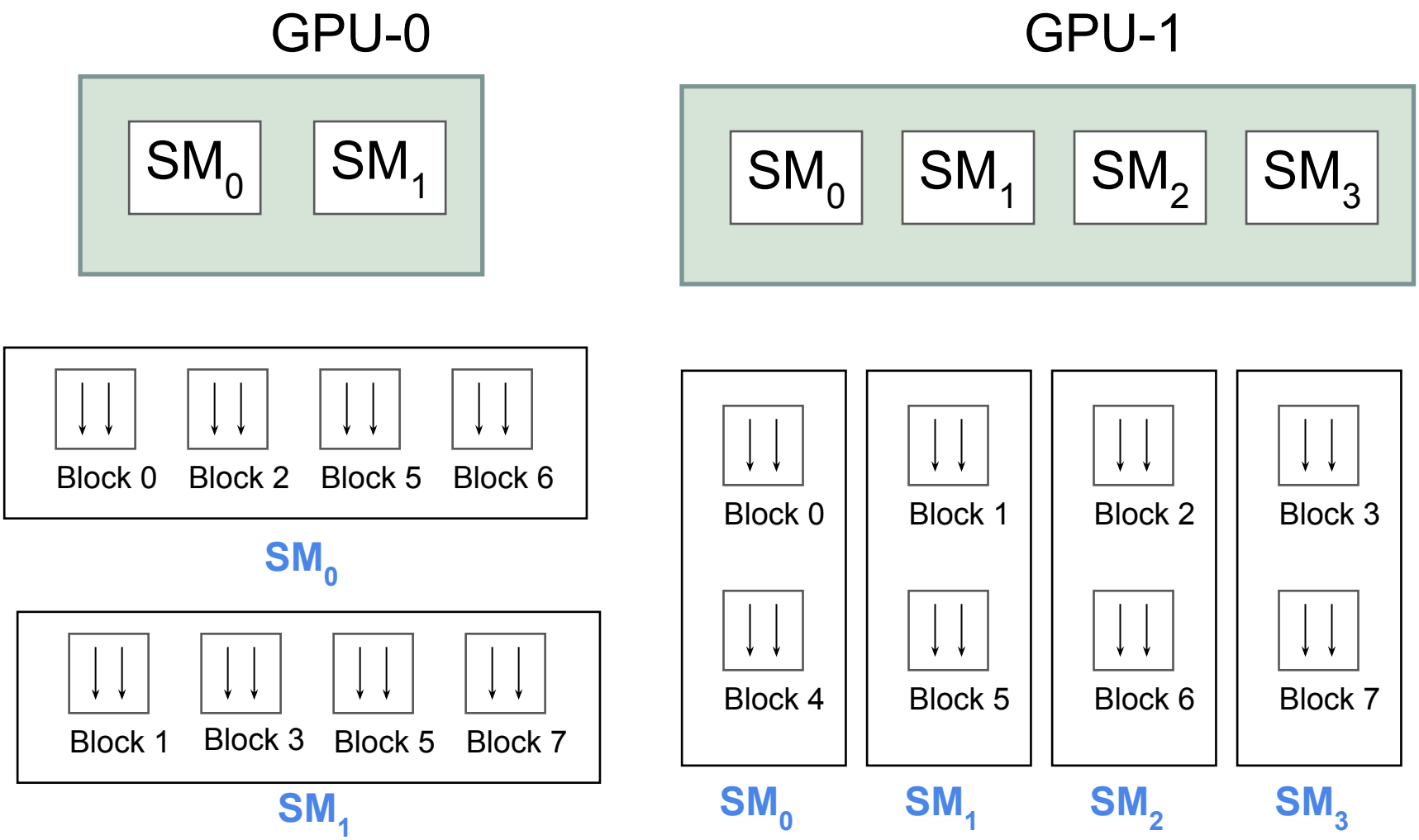
- This Lecture
  - Thread organization (2D & 3D)
  - GPU Instruction Execution

Thread Configuration

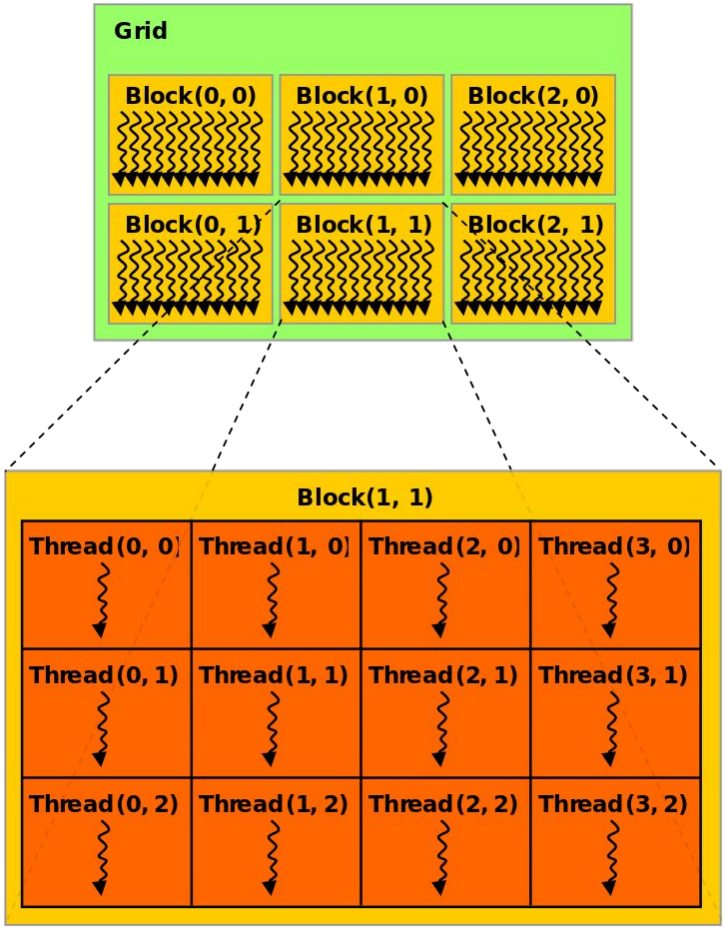
```
add<<<ThreadConfig>>> (dev_a, dev_b, dev_c);
```

```
add<<<ThreadBlocks, Threads>>> (dev_a, dev_b, dev_c);
```

Scalability



Threads and Blocks in 2D



Thread Indexing

- A kernel is launched as a grid of threads.
- A grid is a 3D array of thread-blocks (gridDim.x, gridDim.y and gridDim.z).
  - Thus, each block has blockIdx.x, .y, .z.
- A thread-block is a 3D array of threads (blockDim.x, .y, .z).
  - Thus, each thread has threadIdx.x, .y, .z.

Example: Thread Blocks and Thread in 3D

```
int main(void) {
    dim3 grid(2,3,4);
    dim3 block(5,6,7);
    dkernel <<< grid, block>>> ();
    cudaDeviceSynchrnoize();
    return 0;
}
```

No of thread blocks = 2*3*4
No of threads per block = 5*6*7
Total number of threads = 2*3*4*5*6*7

Accessing Dimensions

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
        if (threadIdx.x == 0 && blockIdx.x == 0 &&
            threadIdx.y == 0 && blockIdx.y == 0 &&
            threadIdx.z == 0 && blockIdx.z == 0) {
                printf("%d %d %d %d %d %d.\n", gridDim.x, gridDim.y, gridDim.z,
                                                blockDim.x, blockDim.y, blockDim.z);
        }
}
int main() {
        dim3 grid(2, 3, 4);
        dim3 block(5, 6, 7);
        dkernel<<<grid, block>>>();
        cudaThreadSynchronize();
        return 0;
}
```

Exercise: 2D Thread Organization

- For a given matrix A of size N*M, write a CUDA program to initialize the matrix elements as below

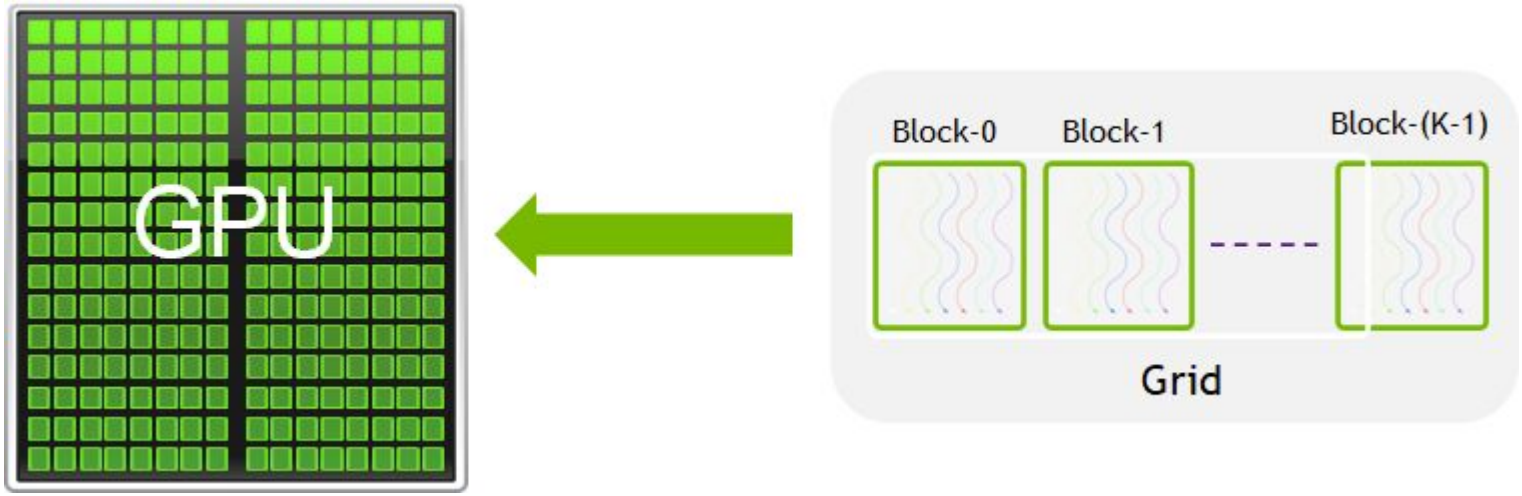
0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

- Assumptions:
  - Matrix is stored in the single dimensional array.
  - No. of thread blocks = 1
  - Each thread block has the dimensions: (N, M, 1)

2D

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *matrix) {
        unsigned id = threadIdx.x * blockDim.y + threadIdx.y;
        matrix[id] = id;
}
#define N       5
#define M       6

int main() {
        dim3 block(N, M, 1);
        unsigned *matrix, *hmatrix;

        cudaMalloc(&matrix, N * M * sizeof(unsigned));
        hmatrix = (unsigned *)malloc(N * M * sizeof(unsigned));

        dkernel<<<1, block>>>(matrix);
        cudaMemcpy(hmatrix, matrix, N * M * sizeof(unsigned), cudaMemcpyDeviceToHost);

        for (unsigned ii = 0; ii < N; ++ii) {
                for (unsigned jj = 0; jj < M; ++jj) {
                        printf("%2d ", hmatrix[ii * M + jj]);
                }
                printf("\n");
        }
        return 0;
}
```

GPU Thread Blocks



Few Constraints

- Thread block size has limit
- Max number of threads blocks reside per SM
- Max threads reside per SM

Compute Capabilities

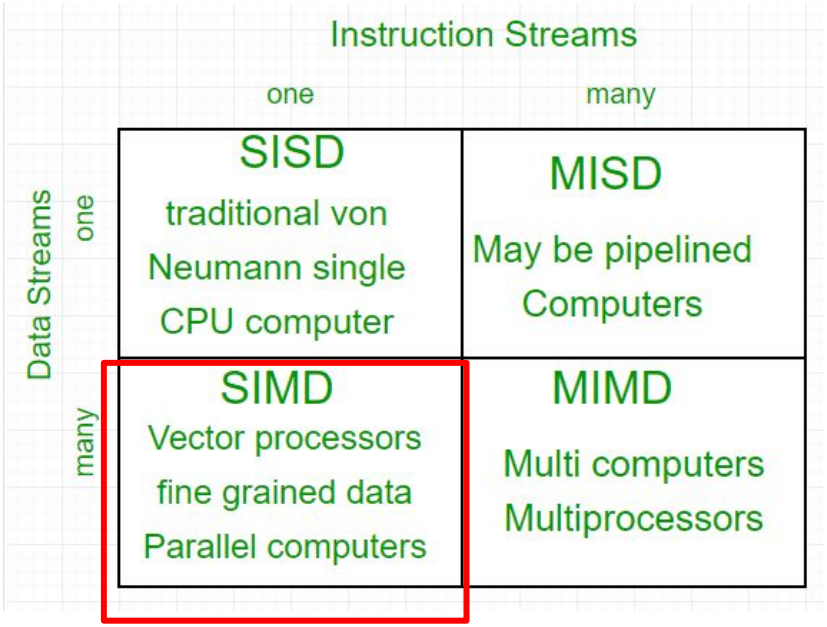
GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255[1]
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

Source: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

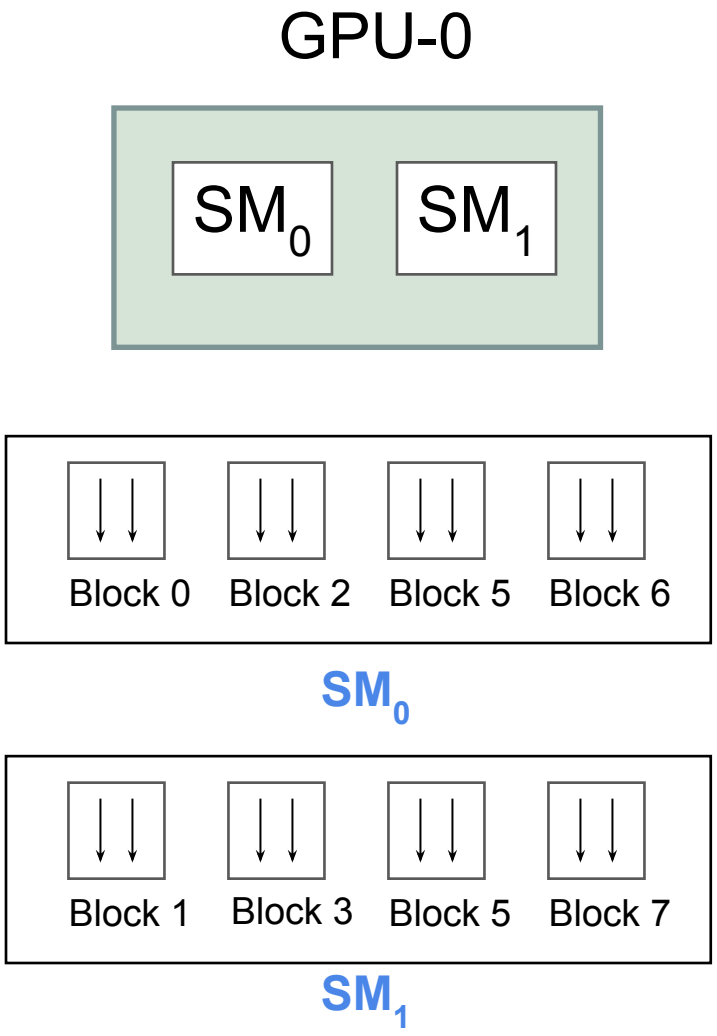
Instruction Execution

Flynn's Taxonomy

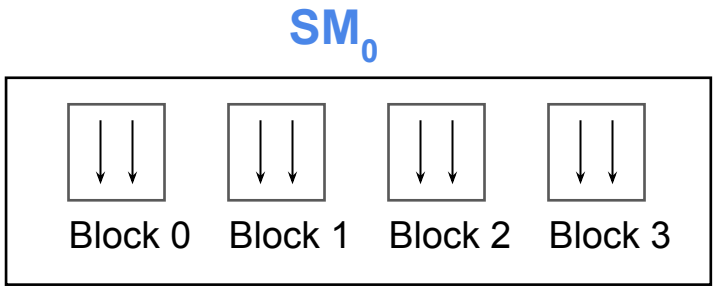
- Flynn's classification of computer architecture



Scalability

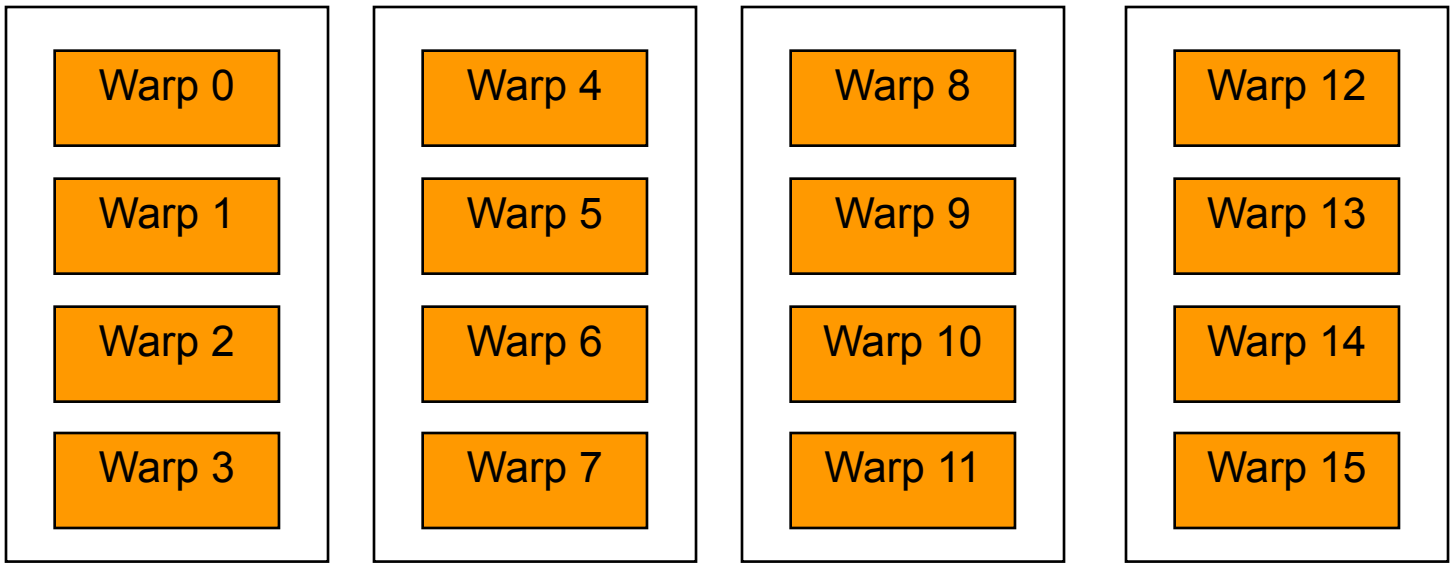


Execution in GPU

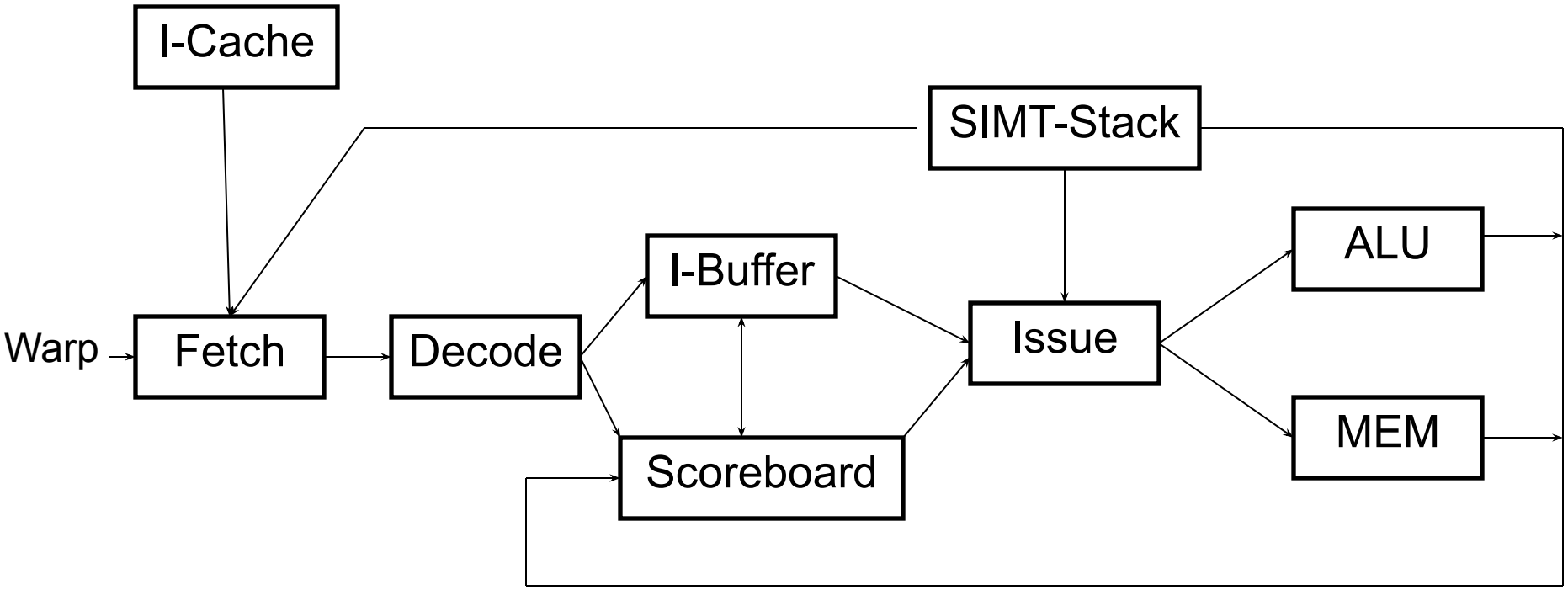


Block Size: 128 threads
Number of Blocks: 4
Total threads: 512

Threads grouped into warps
Size of each warp = 32 threads

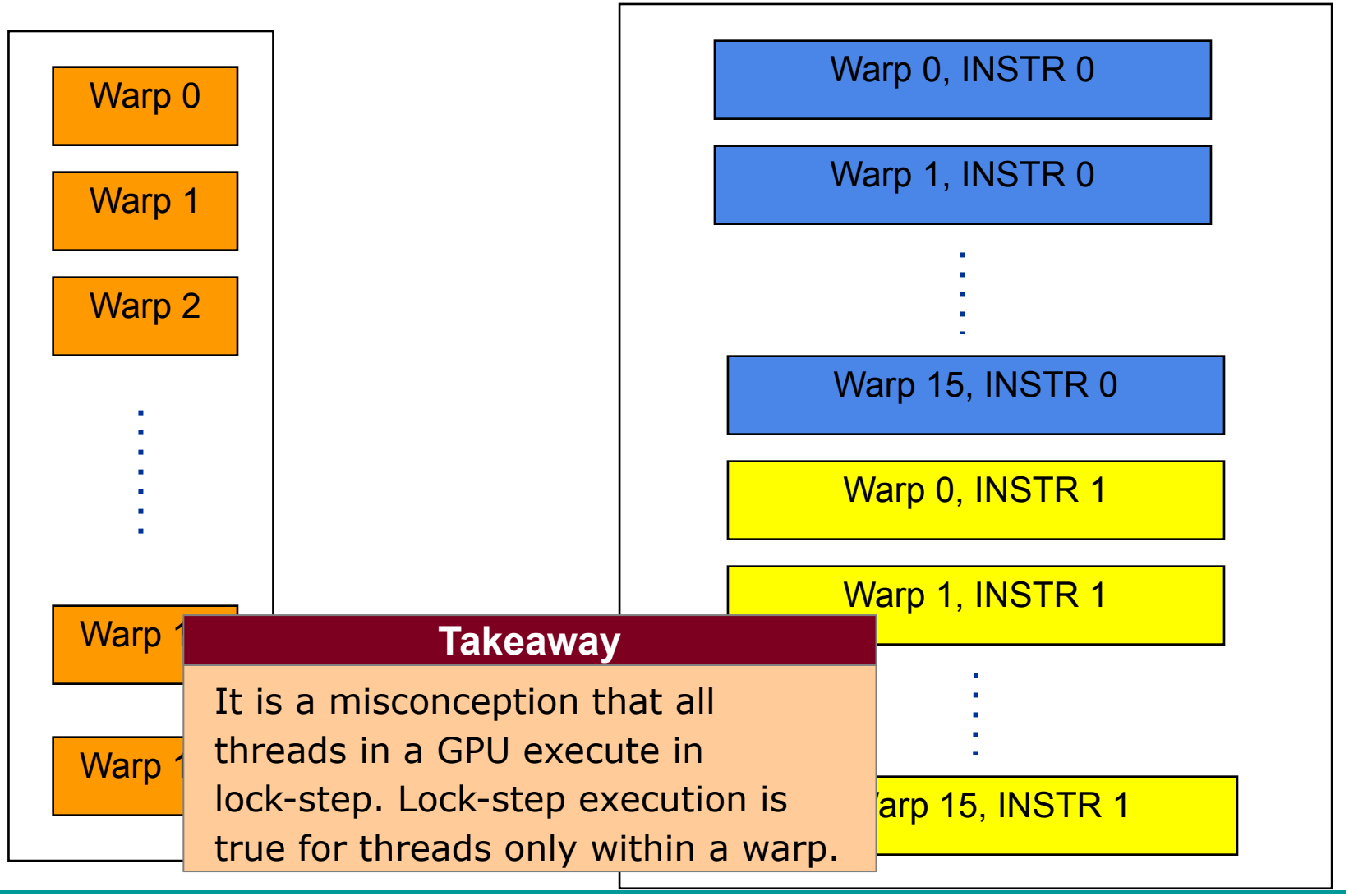


GPU Pipeline*



*As per the GPGPU-Sim Simulator

Instruction Execution



Takeaway

It is a misconception that all threads in a GPU execute in lock-step. Lock-step execution is true for threads only within a warp.

Warp Scheduler

Simple Example (Revisit)

```
__global__ void Square_Kernel(float a){
      /* Compute index i based on thread id */
      a[i] = a[i] * a[i];
}
int main() {
        h_a = malloc(..)//host array
        cudaMalloc(d_a,….) //device

         /* Initialize h_a */

        cudaMemcpy(d_a, h_a, cudaMemcpyHostToDevice)

        Square_Kernel<<<ThreadConfig>>> (d_a);

        cudaMemcpy(h_a, d_a, cudaMemcpyDeviceToHost)
        process(h_a);
        cudaFree(d_a);
        free(h_a);
}
```

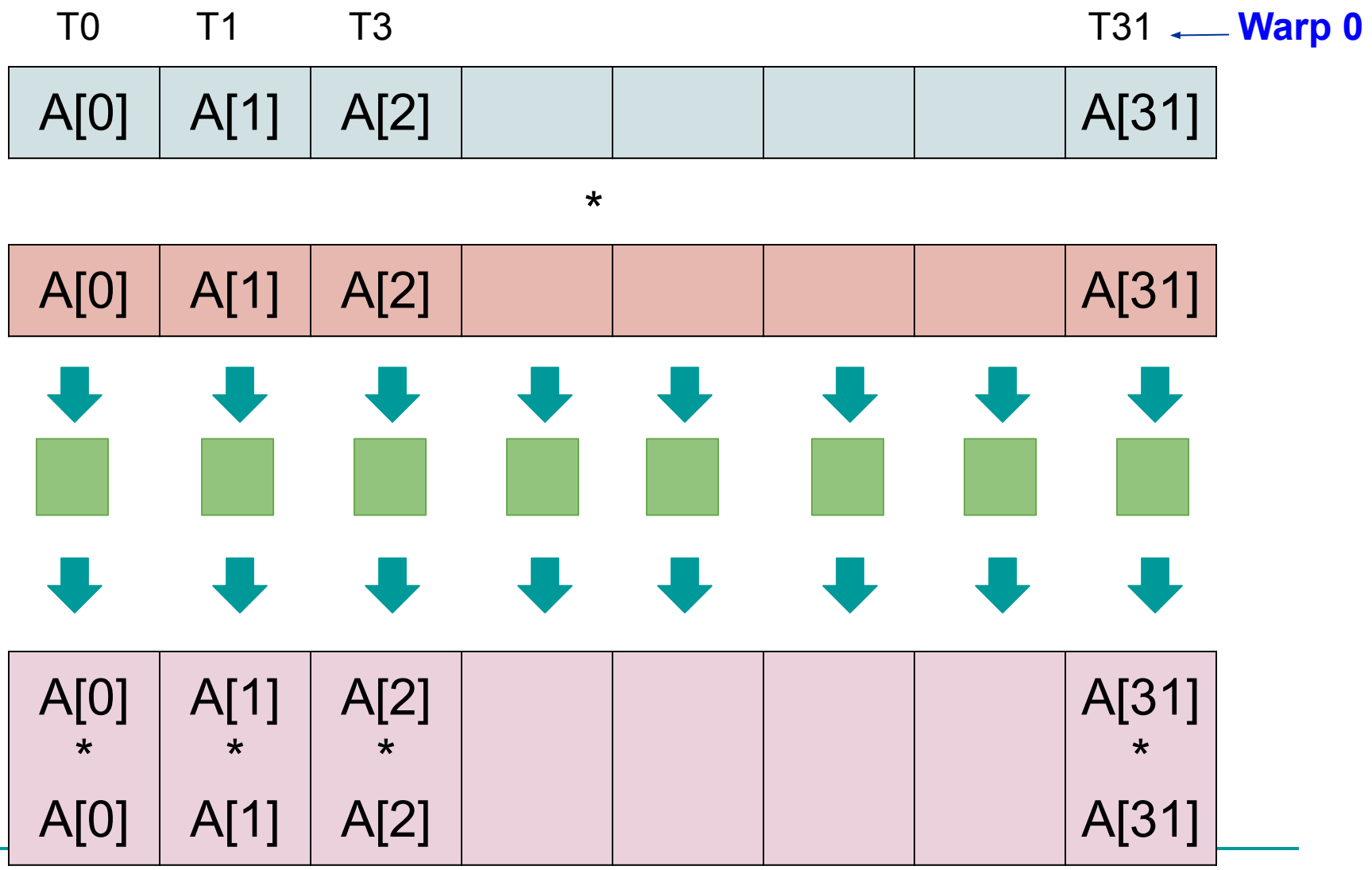
Compute Kernel

CPU to GPU Data transfer

Invoke Kernel

GPU to CPU Data transfer

Warp Execution



GPU Execution Pipeline Example

- 5 Pipeline stages: IF, ID, EX, M and W
- IF, ID, EX, and W stages if no memory
- Warp size is 32 threads
- ADD, SUB, DIV, MUL: 1 cycle per warp.
- Warp scheduler algorithm: round robin
- 3 Warps in the GPU

GPU Execution Pipeline Example

Instruction	Semantics
I1: MOV R0, tid	Moves the thread id (tid) to register R0

GPU Execution Pipeline Example

	1	2	3	4	5	6
W1:I1	IF	ID	EX	W		
W2:I1		IF	ID	EX	W	
W3:I1			IF	ID	EX	W

GPU Execution Pipeline Example

Instruction	Semantics
I1: MOV R0, tid	Moves the thread id (tid) to register R0
I2: Load R2, R0 (R1)	Load operation R2 ← [R1+R0]

Each thread takes 3 cycles for takes memory operation

GPU Execution Pipeline Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	..	..	103	104	105
W1:I1	IF	ID	EX	W															
W1:I2				IF	ID	EX	M	M	M	M	M	M	M	M	..	..	W		
W2:I1		IF	ID	EX	W														
W2:I2					IF	ID	EX	M	M	M	M	M	M	M	..	..	M	W	
W3:I1			IF	ID	EX	W													
W3:I2						IF	ID	EX	M	M	M	M	M	M	..	..	M	M	W

GPU Execution Pipeline Example

Instruction	Semantics
I1: MOV R0, tid	Moves the thread id (tid) to register R0
I2: Load R2, R0 (R1)	Load operation R2 ← [R1+R0]
I3: ADD R3,R1,R2	R3 ← R1+R2

GPU Execution Pipeline Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	..	..	103	104	105	106	107
W1:I1	IF	ID	EX	W																	
W1:I2				IF	ID	EX	M	M	M	M	M	M	M	M	..	..	W				
W1:I3							IF	ID	EX	EX	EX	EX	EX	EX	..	..	EX	W			
W2:I1		IF	ID	EX	W																
W2:I2					IF	ID	EX	M	M	M	M	M	M	M	..	..	M	W			
W2:I3								IF	ID	EX	EX	EX	EX	EX	..	..	EX	EX	W		
W3:I1			IF	ID	EX	W															
W3:I2						IF	ID	EX	M	M	M	M	M	M	..	..	M	M			
W3:I3									IF	ID	EX	EX	EX	EX	..	..	EX	EX	EX	W	

Warp Execution with Conditions

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize)
{
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id % 2) vector[id] = id;
    else vector[id] = vectorsize * vectorsize;
    vector[id]++;
}
```

S0

S1

S2

S3

Warp Execution with Conditions

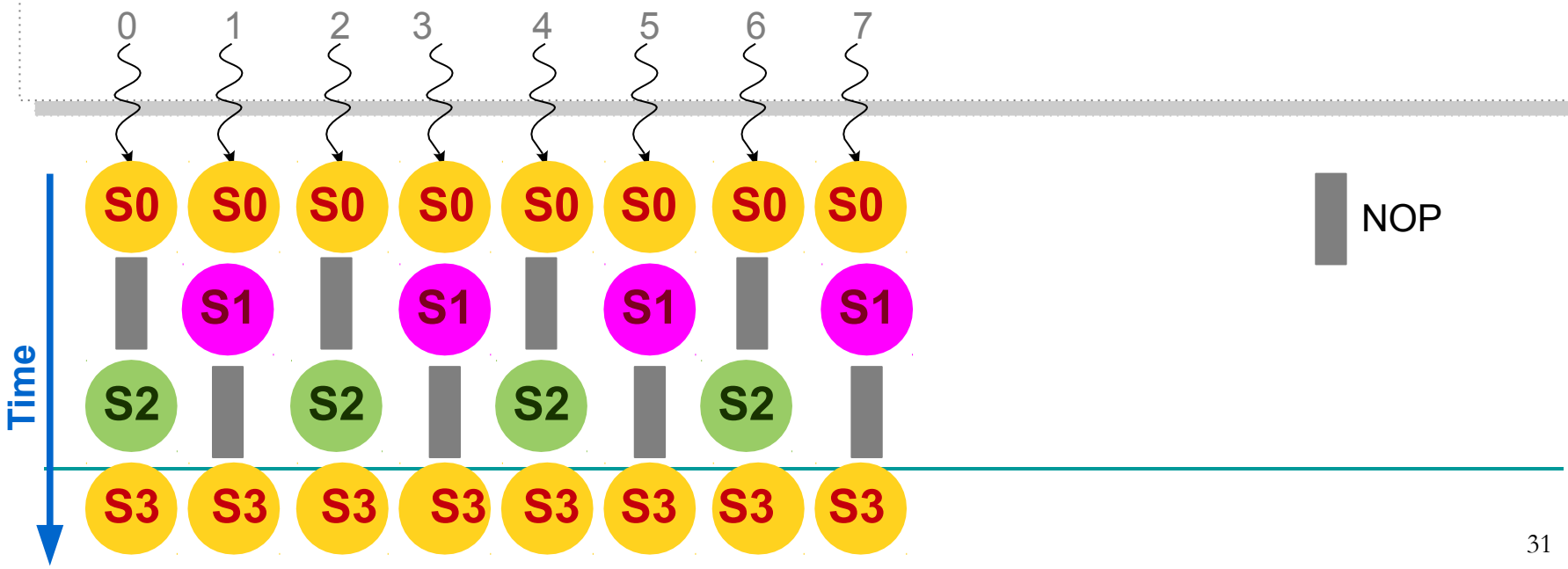
```
__global__ void dkernel(unsigned *vector, unsigned vectorsize)
{
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id % 2) vector[id] = id;
    else vector[id] = vectorsize * vectorsize;
    vector[id]++;
}
```

S0

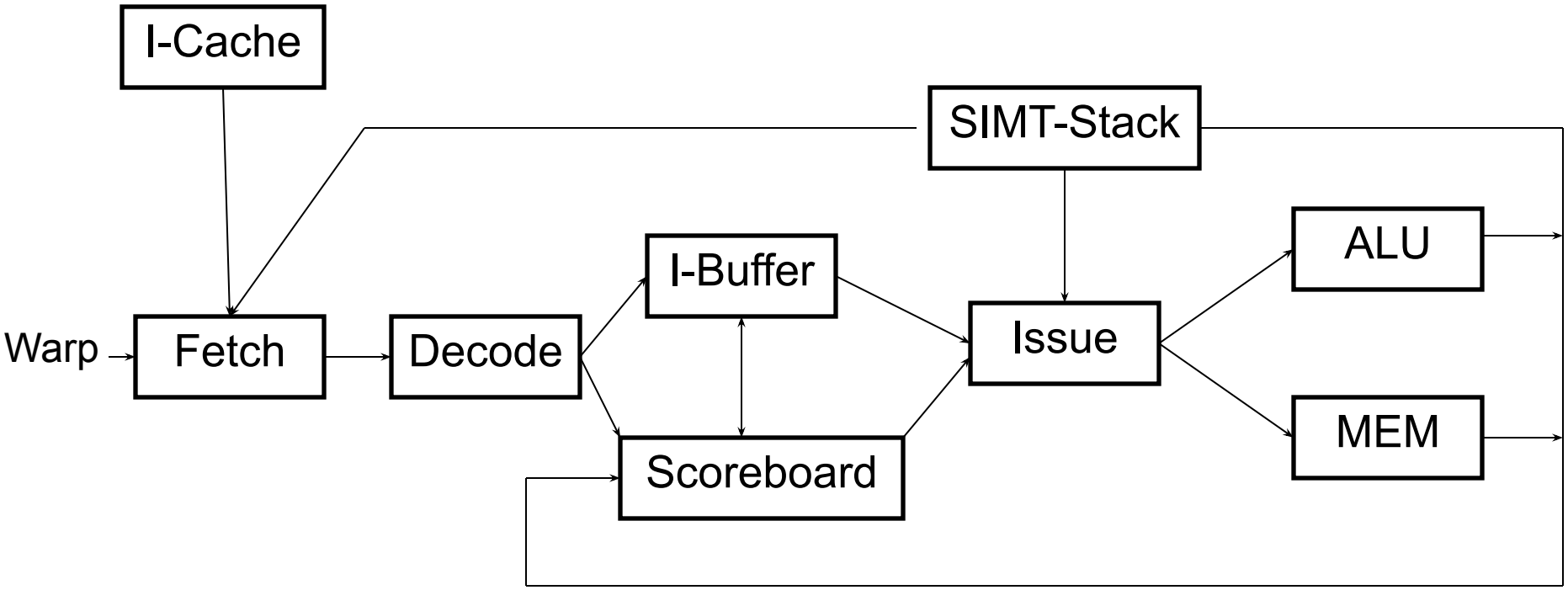
S1

S2

S3



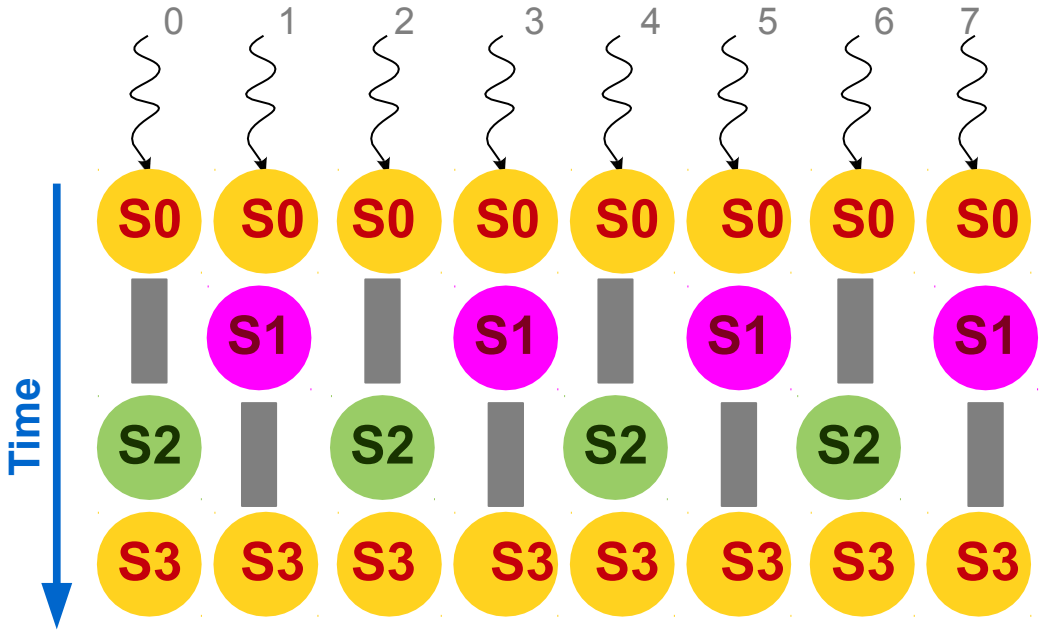
GPU Pipeline*



*As per the GPGPU-Sim Simulator

Warp Execution with Conditions

- When different warp-threads execute different instructions, threads are said to diverge.
- Hardware executes threads satisfying same condition together, ensuring that other threads execute a no-op.
- This adds sequentiality to the execution.
- This problem is termed as **thread-divergence**.



Degree of Divergence

- DoD for a warp is the number of steps required to complete one instruction for each thread in the warp.
- Without any thread-divergence, DoD = 1.
- For fully divergent code, DoD = 32.

Thread-Divergence

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize)
        {
         unsigned id = //compute id.
      if (id % 2 == 0) vector[id] = id;

      else vector[id] = vectorsize * vectorsize;
}
```

What is the degree of divergence?

Exercise: Thread-Divergence

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize)
    {
     unsigned id = //compute id.
  if (id % 3 == 0) vector[id] = id;
  else if (id % 3 == 1) vector[id] = id+1;
  else vector[id]=id+2;
```

What is the degree of divergence?

Exercise: Thread-Divergence

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize)
        {
         unsigned id = //compute id.
        switch (id) {
            case 0: vector[id] = 0;                          break;
            case 1: vector[id] = vector[id];                 break;
            case 2: vector[id] = vector[id - 2];             break;
            case 3: vector[id] = vector[id + 3];             break;
            case 4: vector[id] = 4 + 4 + vector[id];         break;
            case 5: vector[id] = 5 - vector[id];             break;
            case 6: vector[id] = vector[6];                  break;
            case 7: vector[id] = 7 + 7;                      break;
            case 8: vector[id] = vector[id] + 8;             break;
            case 9: vector[id] = vector[id] * 9;             break;
            default: vector[id] = vector[id] + vector[id];    break;
                    What is the degree of divergence?
    }
}
```

References

- CS6023 GPU Programming
  - https://www.cse.iitm.ac.in/~rupesh/teaching/gpu/jan20/
- Miscellaneous resources from internet