

Special thanks to Assoc Prof Nicholas Vun (Associate Chair in Academic)  
for his initiative and contribution to this module

# CE/CZ 3001: Advanced Computer Architecture

## **Module 6: GPU Architecture and CUDA Programming** - GPU Internal Operation

Asst Prof Liu Weichen  
School of Computer Science and Engineering  
Nanyang Technological University, Singapore

# Outline

- GPU operation – hardware perspectives
- Hardware resources utilization

# Software/Hardware Perspectives

CUDA provides the programming model on how to create software with parallelism for the GPU

- that can scale transparently to large numbers of CUDA cores.

Threads, thread blocks (and grid) are essentially a programmer's (software) perspective.

But how is a CUDA program executed when it is translated into its corresponding machine code?

- handled by the CUDA runtime system, which knows the underlying architecture of the GPU.

# Cores and Threads

Recall that a GPU organizes the streaming processor (SP, or CUDA cores) in groups of SMs

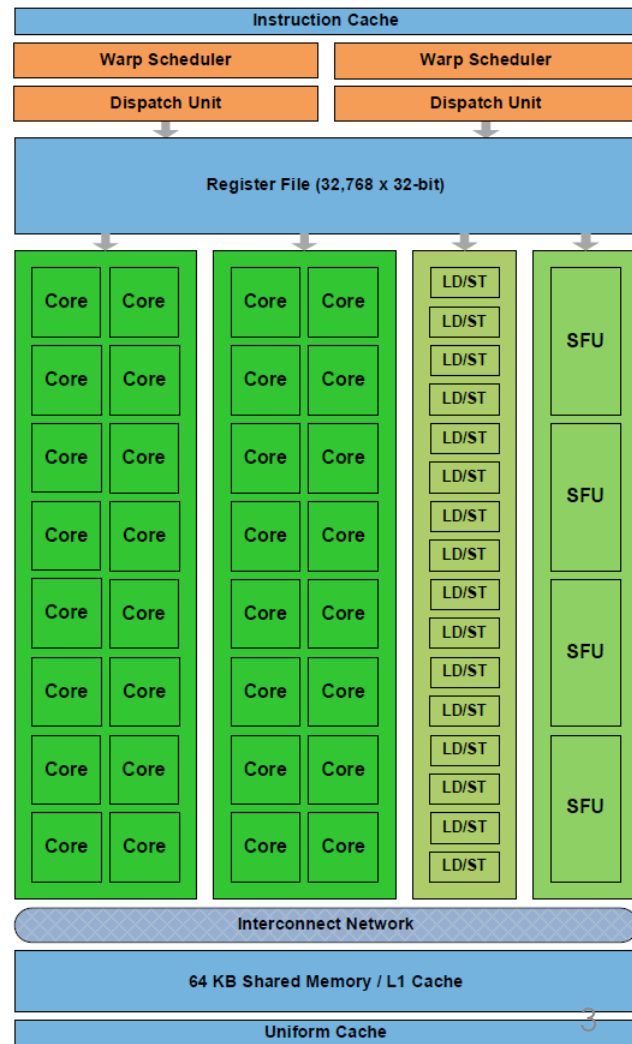
- each SM can contain from 32 SPs (Fermi) to 64 SPs (Turing)

When a kernel is launched

- each SP core will execute one thread

How are the threads assigned to the cores?

- kernel <<<2, 200>>>
- Kernel <<<5, 100>>>



# SM Block Allocation

During runtime

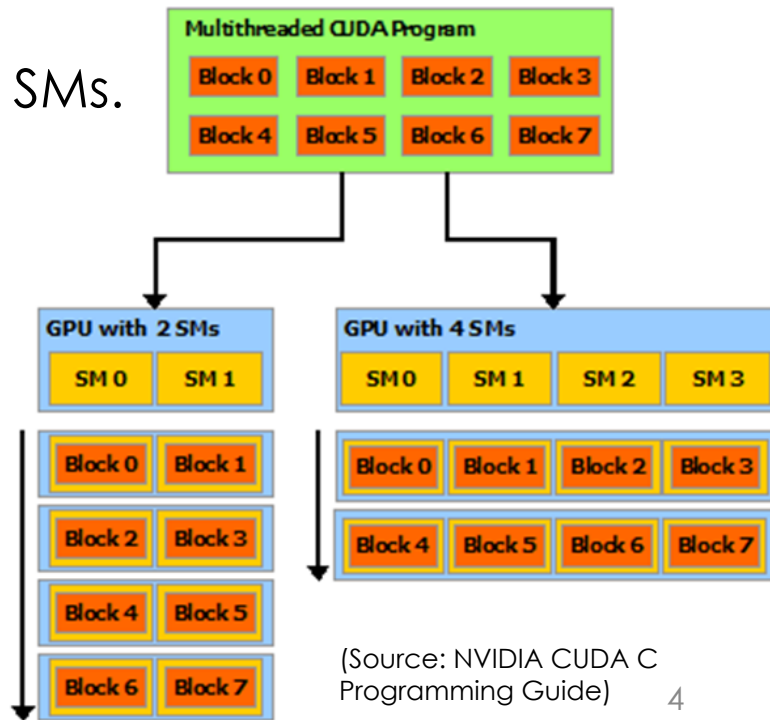
- each **block of threads** can be scheduled on any of the available SM within a GPU, in any order, concurrently or sequentially.
- but block **cannot split** among multiple SMs.

This is **assigned by the runtime system**

- which knows about the internal detail of the underlining GPU
- enable automatic scalability

There is also a maximum number of blocks that each SM can handle (e.g. 8)

- but actual number will be constrained by the available resource.



(Source: NVIDIA CUDA C Programming Guide)

# Blocks vs Threads

Earlier on we asked which one we should use:

- multiple parallel blocks with 1 thread each?
- 1 block with multiple parallel threads?

From hardware resource perspective & performance consideration

- both options are not ideal
- 1 block => only 1 SM is used per GPU
- 1 thread => potential only 1 core is used per SM (if blocks spread among multiple SMs)

GPU parallel computing is hence only effective if we have massive amount of parallel data

- which also compensates for overhead of copying data between host and device

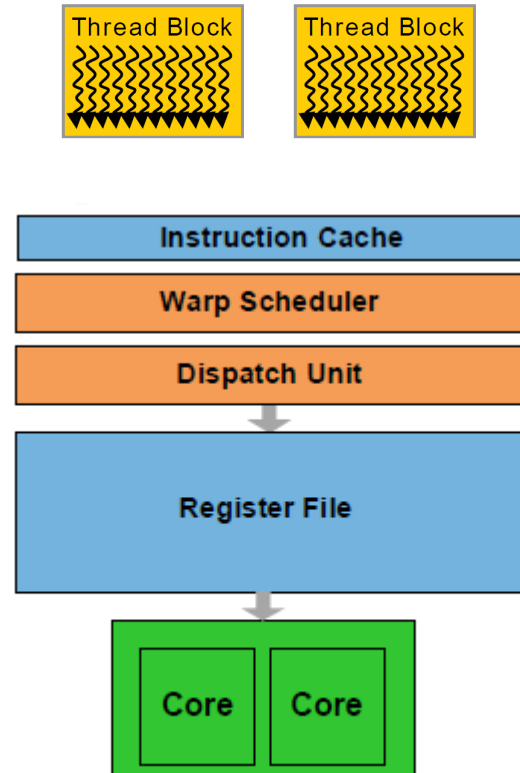
# Warp

In NVIDIA GPU, when a SM is given one or more thread blocks to execute

- it first partitions the threads into **group of 32** known as a **Warp**
  - **all threads** in a warp must execute the **same instruction** but with different data - i.e. SIMD

Each warp then gets scheduled by a **warp scheduler** for execution.

- i.e. threads are always scheduled in a warp group.





# Warp Scheduler

Why the need of scheduling?

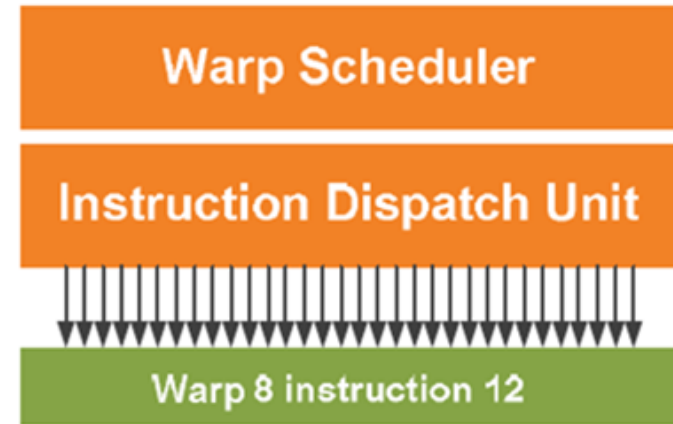
- the number of CUDA cores in a SM is usually less than the total number of threads that are assigned to it.

At any one time

- only 1 warp will be selected by the Warp Scheduler for execution

A warp consists of 32 threads with consecutive threadidx values

- e.g. threads of first warp will have threads ids between 0-31, second warp will have thread ids from 32-63 etc.



(Source: NVIDIA Fermi Compute Architecture Whitepaper)



# Warp Execution

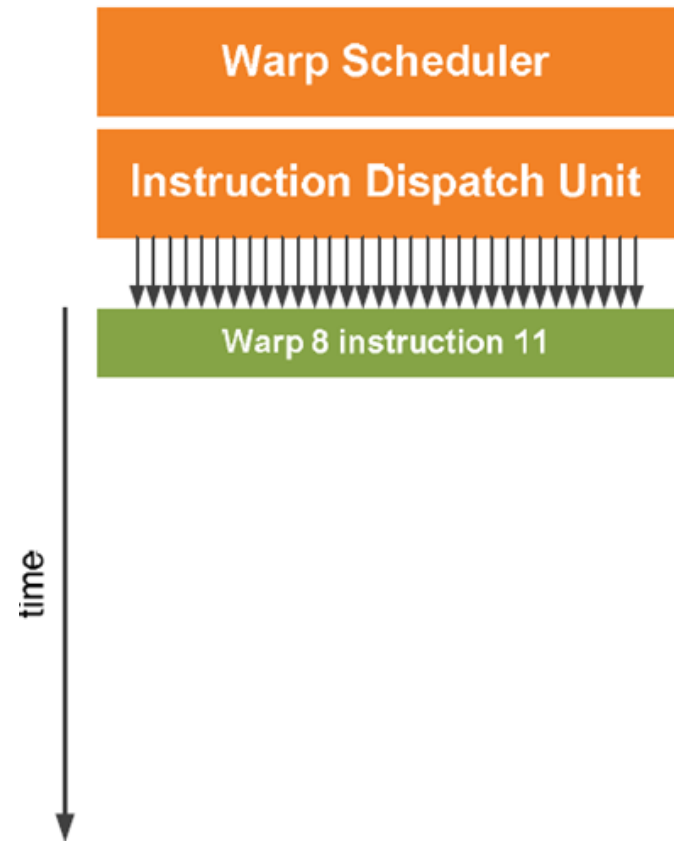
While a warp is waiting for the results of a previously executed instruction

- E.g. LD/ST data from memory
- a different warp that is ready is selected for execution.
- hence 'hiding' the latency

So it is more efficient if we have many threads and many blocks.

Warp selection is based on certain prioritized scheduling policy

- E.g. Round Robin, Least Recently Fetched

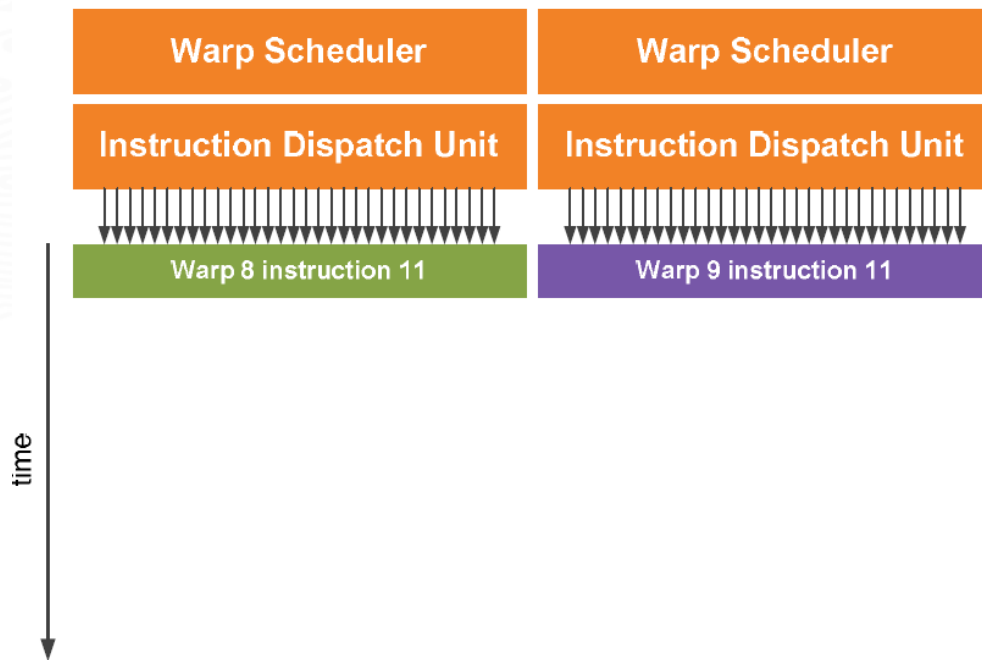


(Source: NVIDIA Fermi Compute Architecture Whitepaper)

## SM with multiple Warp Schedulers

A SM can contain one or more warp scheduler

- E.g. A SM with Dual Warp Schedulers allowing two warps to be issued and executed concurrently.



(Source: NVIDIA Fermi Compute Architecture Whitepaper)

# SIMT Architecture

SM employs a unique architecture called **SIMT** (hardware perspective)

- Single-Instruction, Multiple-Thread
- i.e. a warp executes one common instruction for all its threads at a time.

Within **a single thread**, its instructions are

- **pipelined** to achieve **instruction-level parallelism**
- issued in order, with no branch prediction and speculative execution

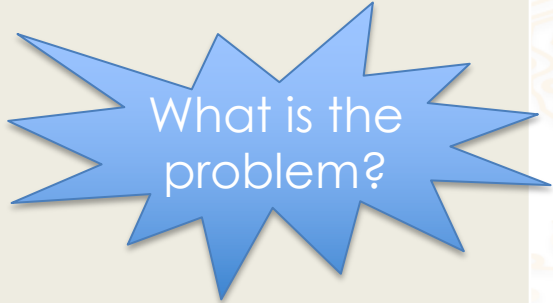
Individual threads in a warp start together, at the same instruction address

- but each has **its own instruction address counter and registers**
- free to branch and **execute independently** when the thread diverges, such as due to data-dependent conditional execution and branch.

# Thread Divergence On Performance

Consider the If-then-else statements in a kernel code as shown below.

```
1  __global__  
2  void kernel_x(int n, int x){  
3      int i;  
4      i = n * 20;  
5      if (i < 300)  
6          x = n;  
7      else  
8          x = n- 300;  
9      }  
10  
11 int main(void){  
12     :  
13     kernel_x<<<2, 64>>>(d_n, d_x);  
14     :
```



What is the problem?

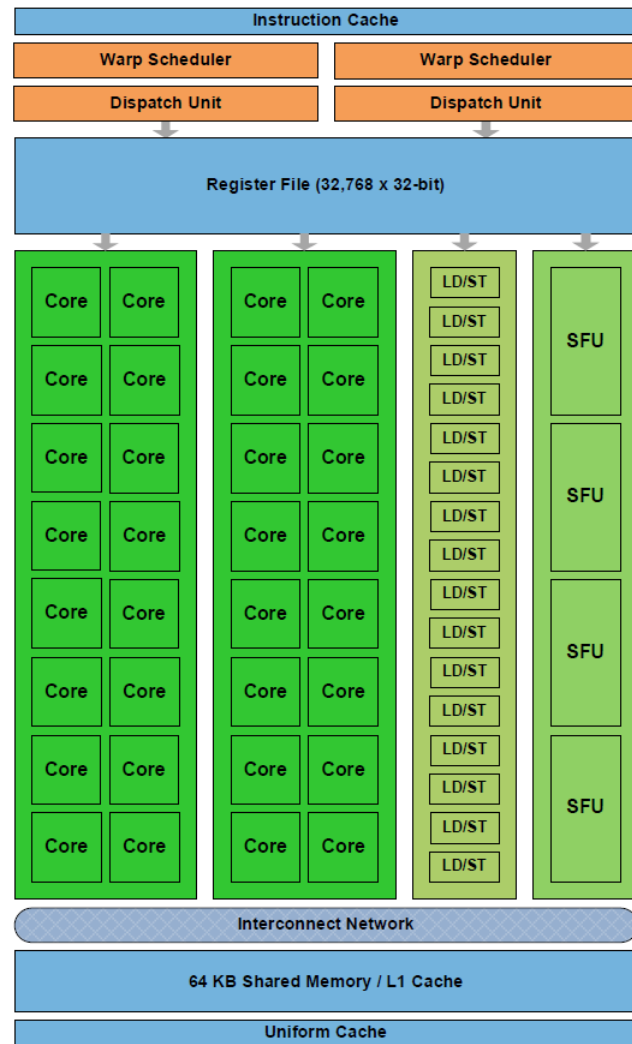
# Hardware Resource Allocation

On-chip resources are used to store the **execution context** for each warp processed by a SM

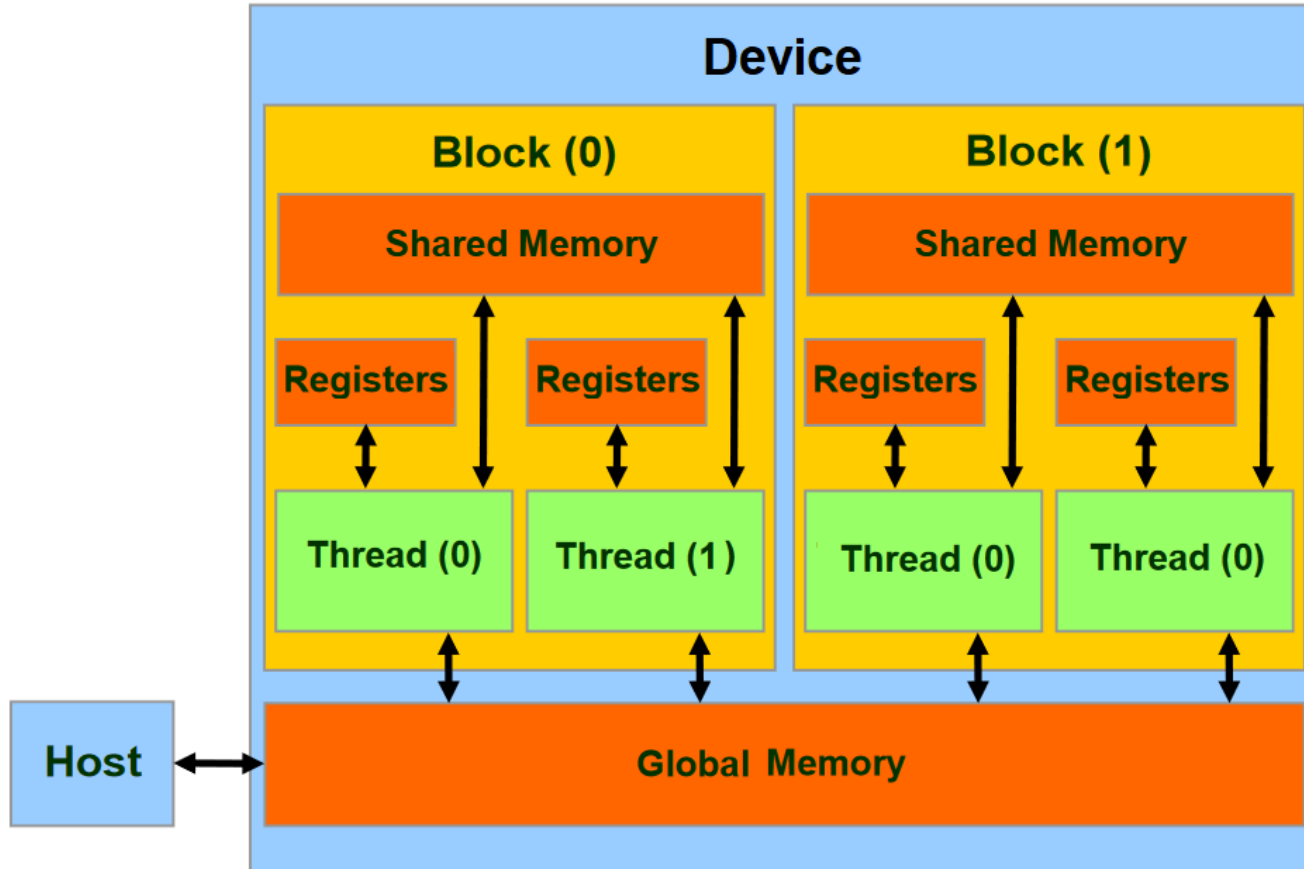
- E.g. program counters, registers, shared memory, etc.

The context is **maintained on-chip** during the entire lifetime of the warp

- allow fast switching between warps
- without the need of the conventional CPU's context switching.



# Memory Resources



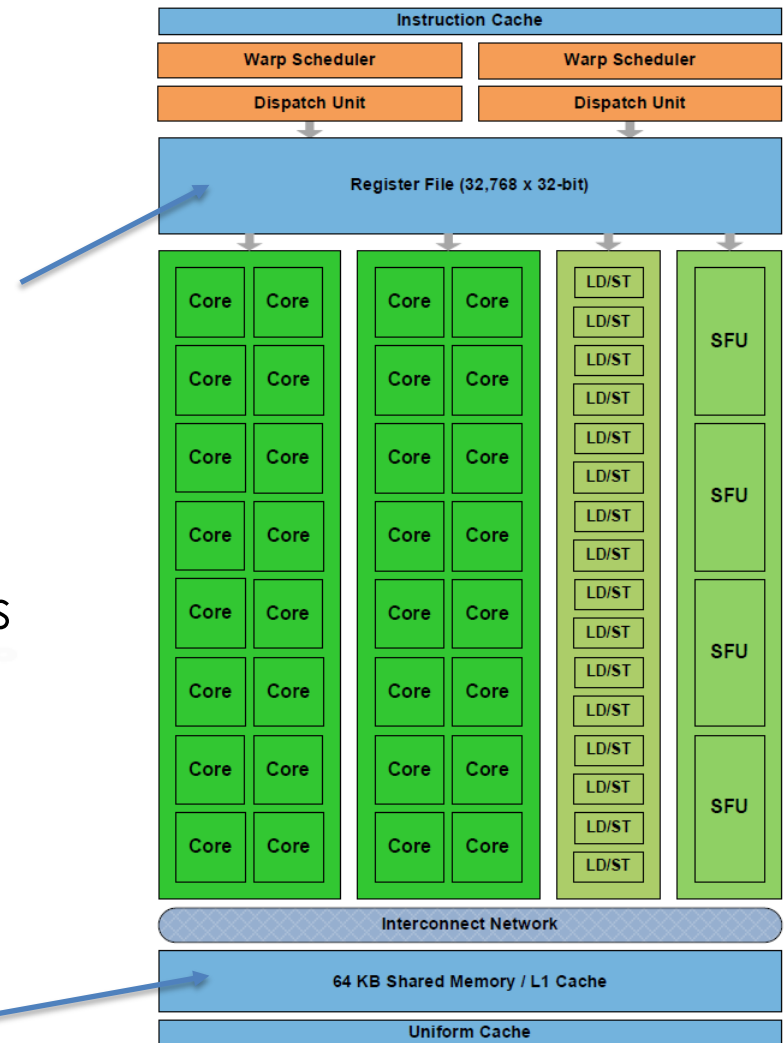
# Resource Usage

But with **limited amount of resources** such as the registers and shared memory in the SM

- number of threads that can actually be executed in a SM is capped for a given application.

The more resources a thread requires

- the less the number of threads that can simultaneously reside in the SM.





# Hardware Resource Usage

Example: Register Usage

Supposed a SM can accommodate upto 1536 threads and has 16384 registers.

To accommodate 1536 threads

- each thread can use no more than  $16384/1536 = 10$  registers.

If each threads requires 12 registers

- the number of threads that can simultaneously reside in the SM has to be reduced.

Reduction is done by removing a whole block

- if each block contains 128 threads
- reduction of threads will be done by reducing 128 threads at a time.

# Hardware Resource Usage

Another example: Shared memory usage

Suppose that a CUDA GPU has 24KB of shared memory per SM .

Suppose that each SM can support up to 8 blocks.

- each block must use no more than 3KB of shared memory.

If each block uses 5KB of shared memory for a particular application

- no more than 4 blocks can reside in a SM.

What happen if there are not enough registers or shared memory available per SM for at least one block?

- the kernel will fail to launch.

# Hardware Resource

Once a thread block is launched on a SM

- all its warps are resident until their executions finish.

Thus a new block is not launched on an SM

- until there is sufficient number of free registers for all warps of the new block,
- and until there is enough free shared memory for the new block.

# Warp Issuing Efficiency

Each scheduler must have at least one warp eligible to issue an instruction every clock cycle (pipelined design)

- to **hide latencies between instructions**

Warp could be not ready due to

- input operands are not ready
- loading of data from global memory
- waiting at synchronizing point

To avoid situations where all warps are stalled and no instructions are issued

- maintain as many active warps as possible throughout the execution of the kernel

# Summary

Number of blocks and warps residing on each SM for a given kernel call depends on

- the execution configuration of the kernel call
- the memory resources of the SM
- the resource requirements of the kernel

Due to the limited amount of resources in SM

- the **higher** the **demands** of a thread, the **lower** the **number** of threads that can actually run in parallel inside the SM
- need to reduce the number of threads

To extract the highest performance from the underlying architecture provided by the GPUs, it is essential to

- have careful design of the program and its decomposition
- make judicious choice of the number of threads and blocks