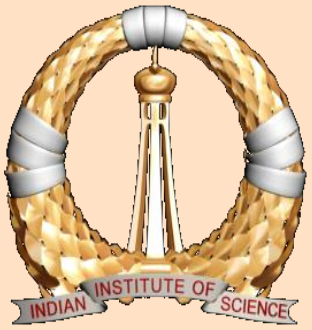
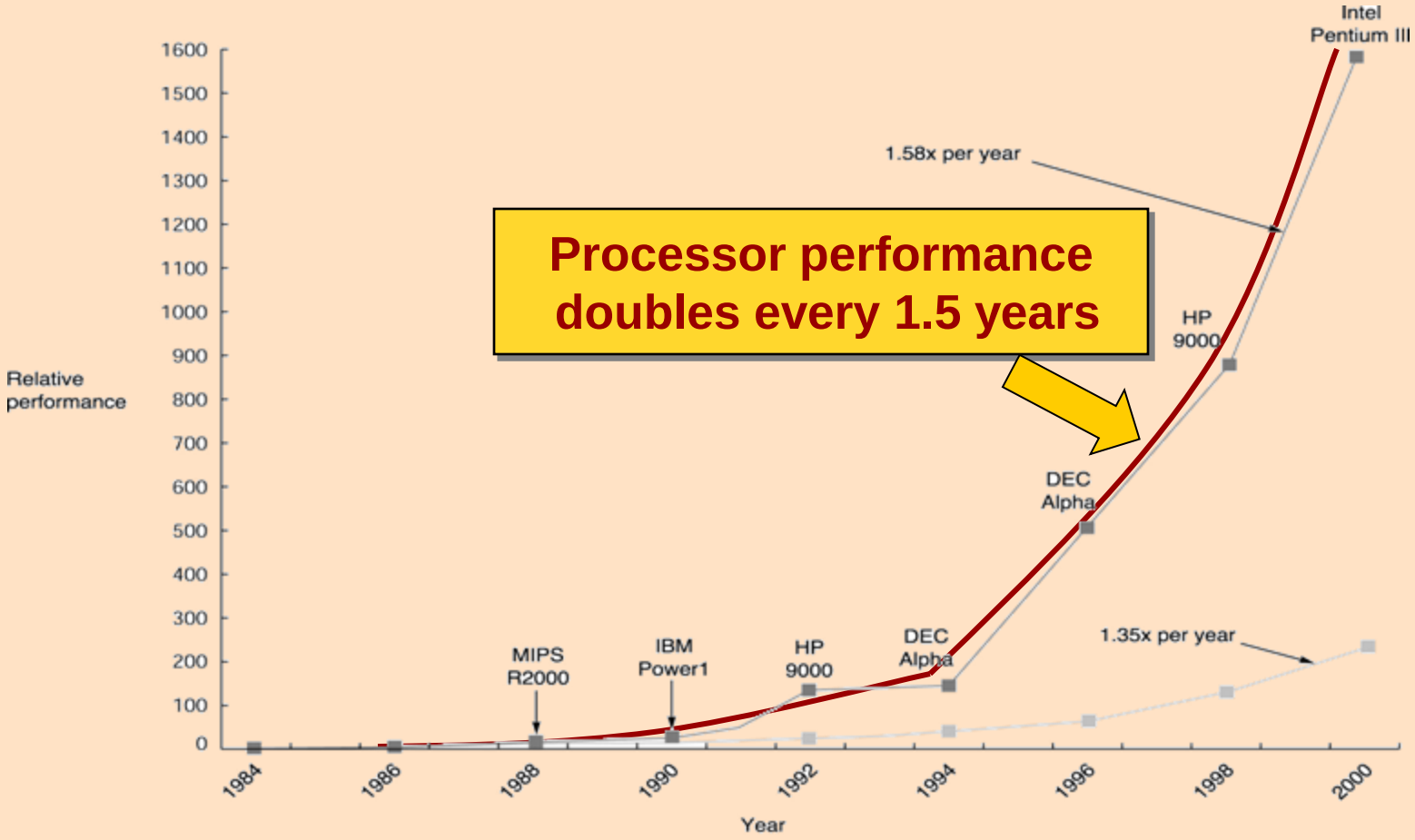
GPU Architecture

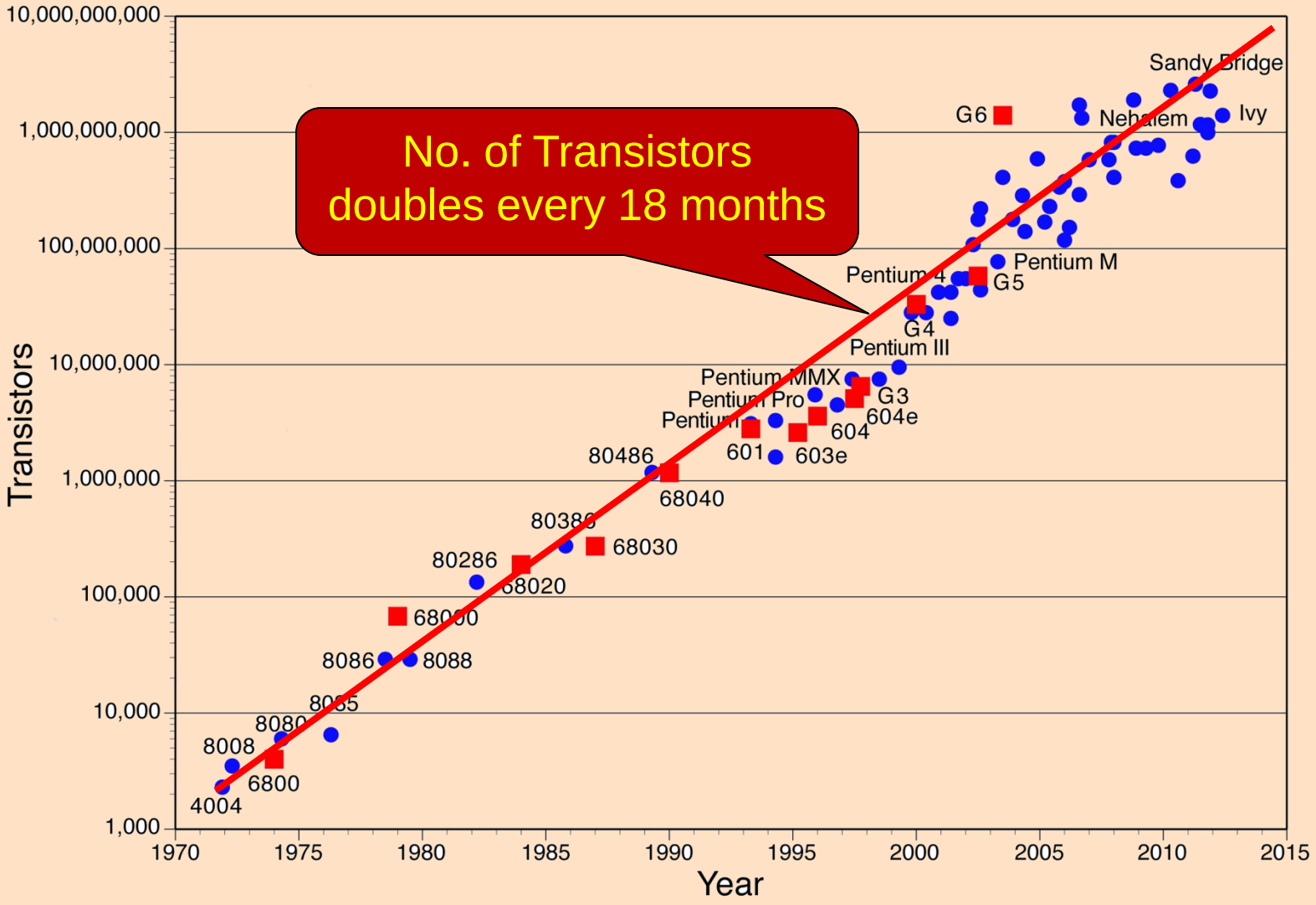
R. Govindarajan
Computer Science & Automation
Supercomputer Edn. & Res. Centre
Indian Institute of Scinece, Bangalore
govind@iisc.ac.in



Moore's Law : Performance

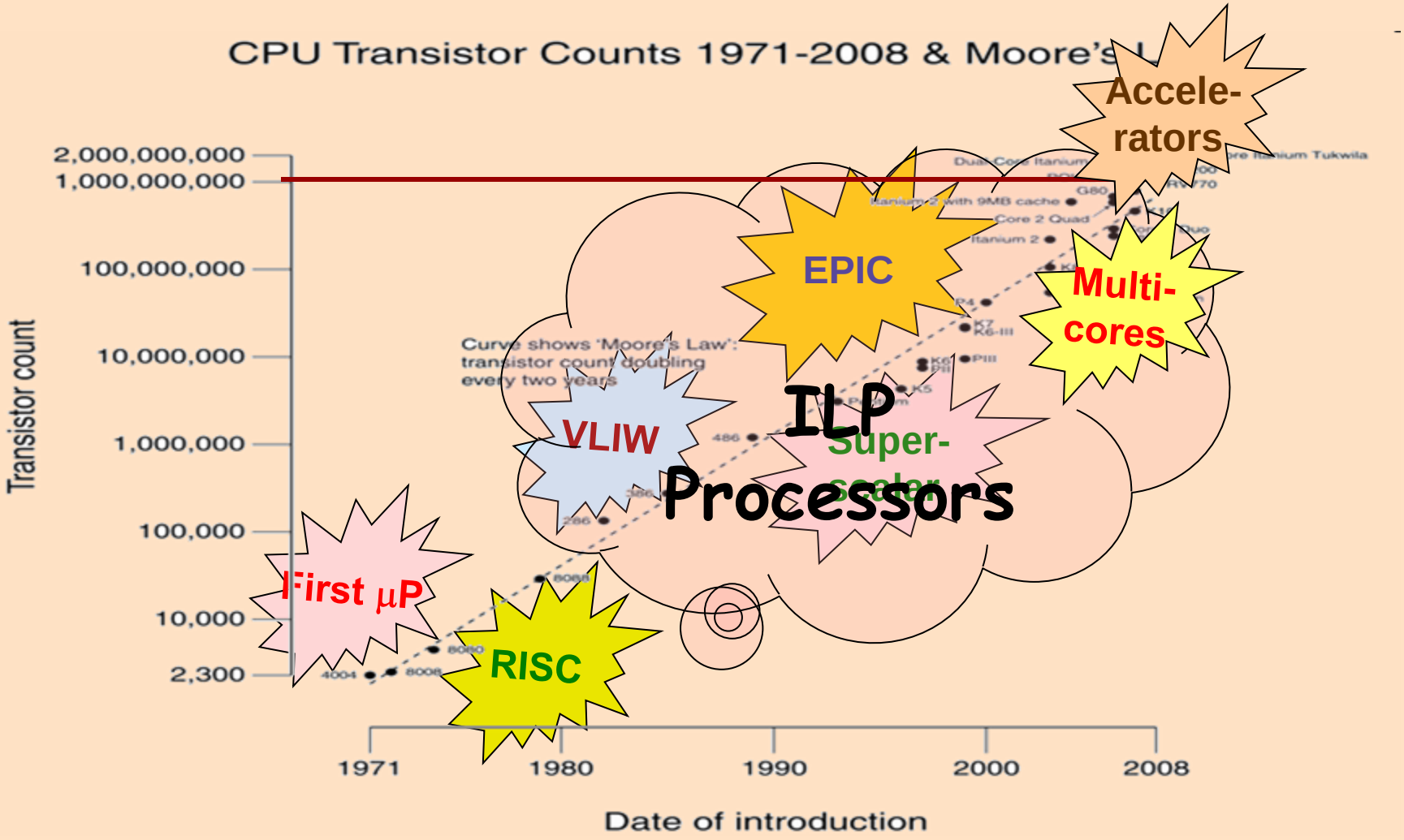


Moore's Law



Source: Univ. of Wisconsin

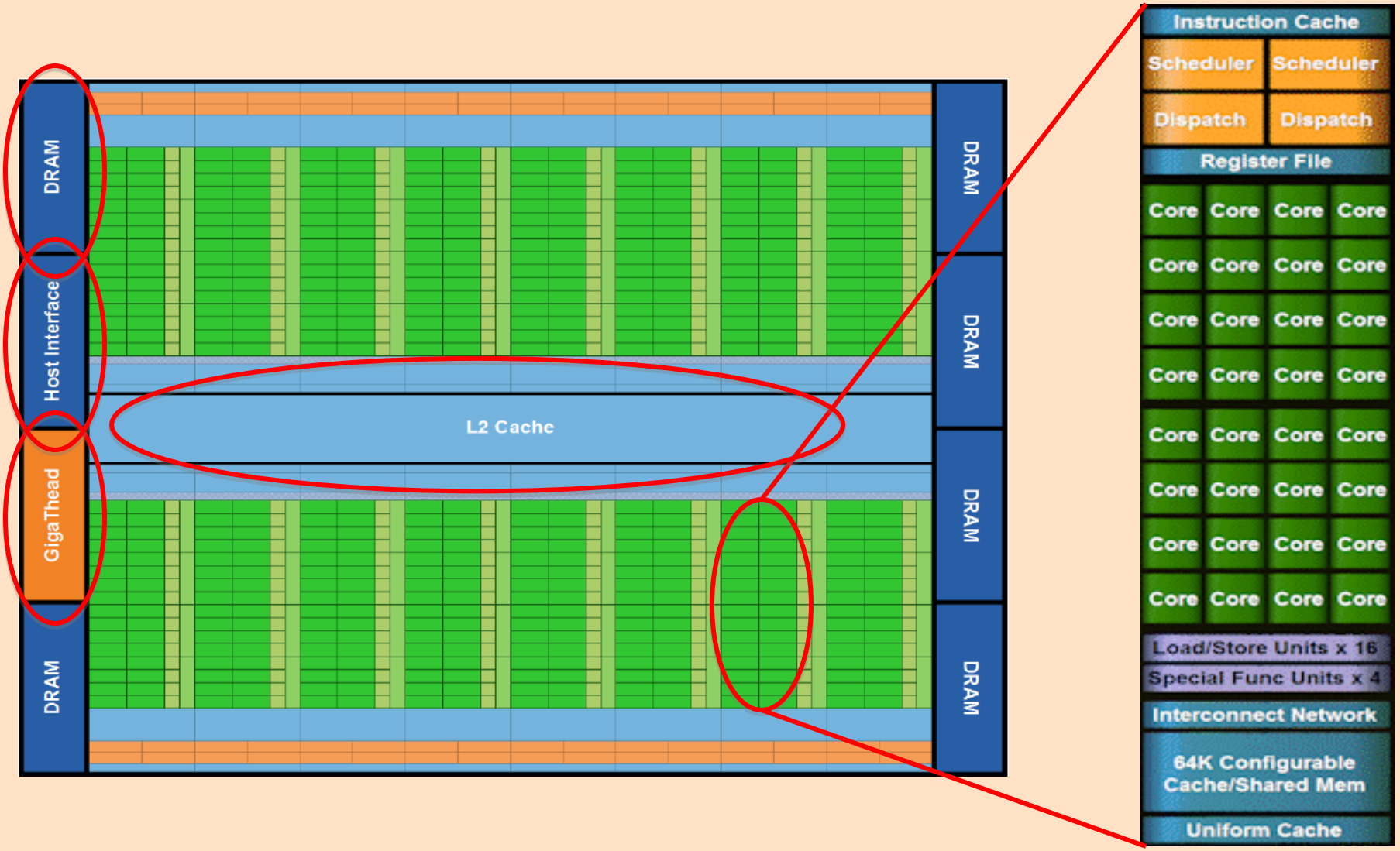
Moore's Law: Processor Architecture Roadmap (Pre-2000)



Rise of GPU Computing

- GPUs have become a popular platform for general purpose applications
- New Programming Models
  - CUDA
  - ATI Stream Technology
  - OpenCL
- Order of magnitude speedup over single-threaded CPU

Accelerator - Fermi S2050

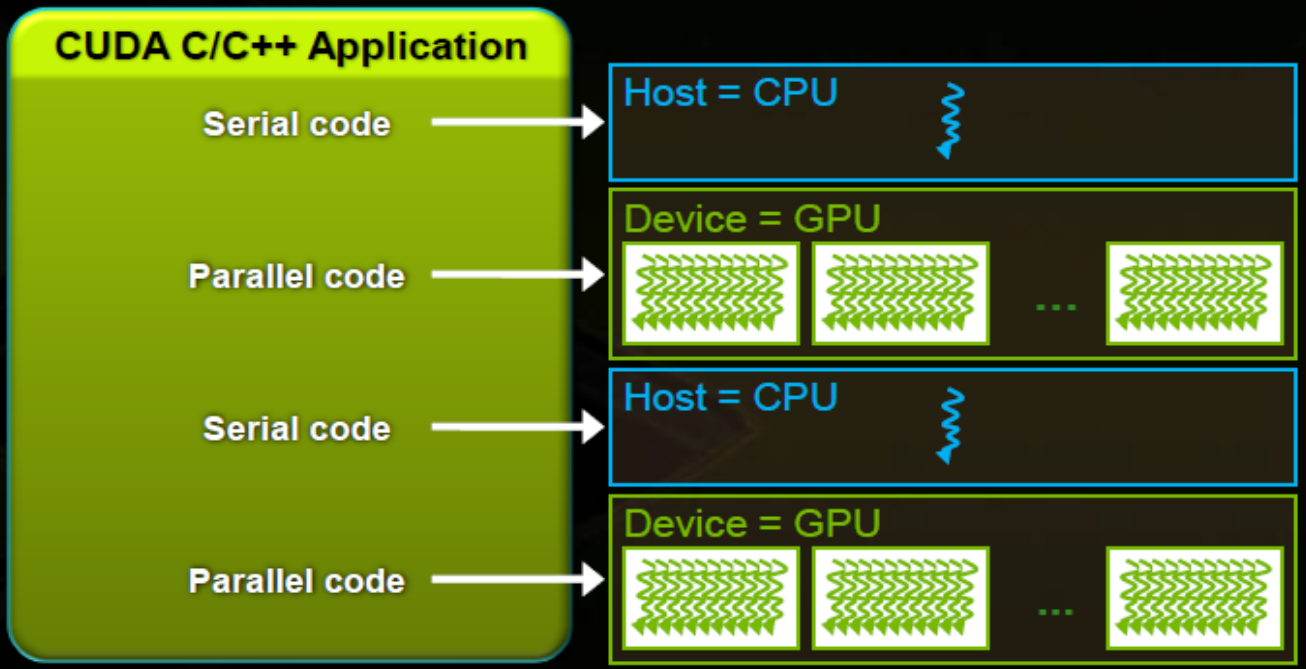


Next Few slides from

CUDA Programming

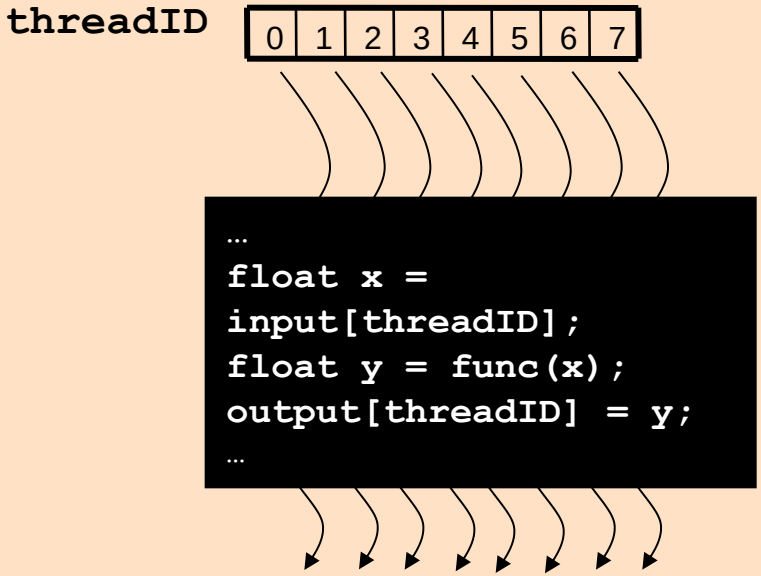
Anatomy of a CUDA C/C++ Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements



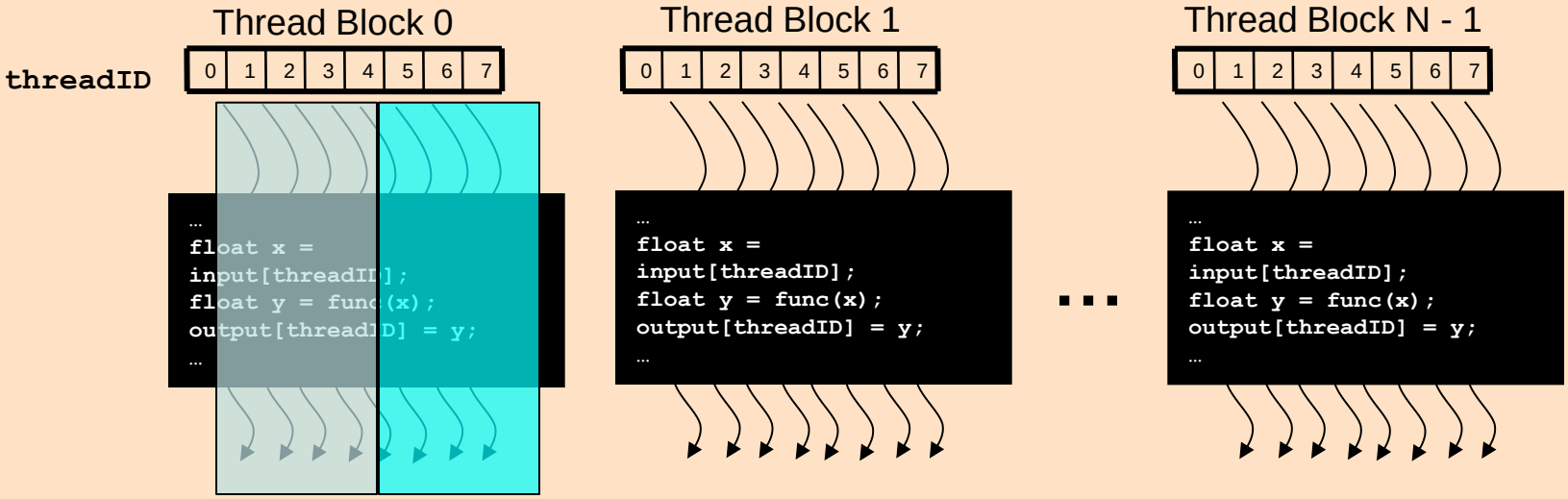
Execution of CUDA on GPUs

- A CUDA kernel consists of an array of light-weight threads
  - All threads run the same code (SPMD), but on different data
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



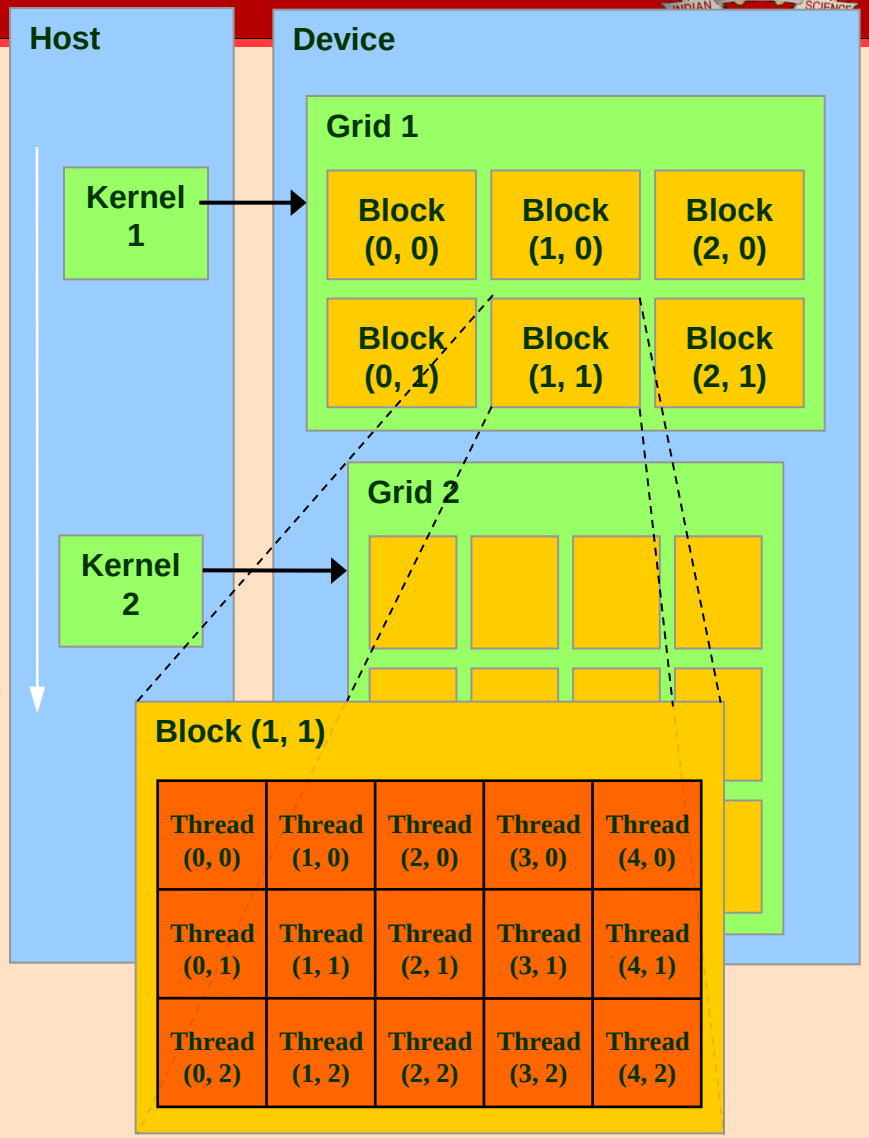
Threads, Warps, Thread Blocks

- Group threads into multiple blocks
  - Threads within a block cooperate via shared memory and barrier synchronization
  - Thread block scheduled on a single SM
  - Consecutive k(=32) threads (within a block) form a warp
  - Instrn. from a ready Warp is scheduled each cycle



Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - Grid can be a 2-dimensional array of thread blocks
- A thread block is a batch of threads that can cooperate with each other
  - Synchronizing their execution
  - Efficiently sharing data through a low latency shared memory
- Each thread block can be a 1-D, 2-D or 3-D array of threads



Source : "NVIDIA CUDA Programming Guide"

CUDA Programming

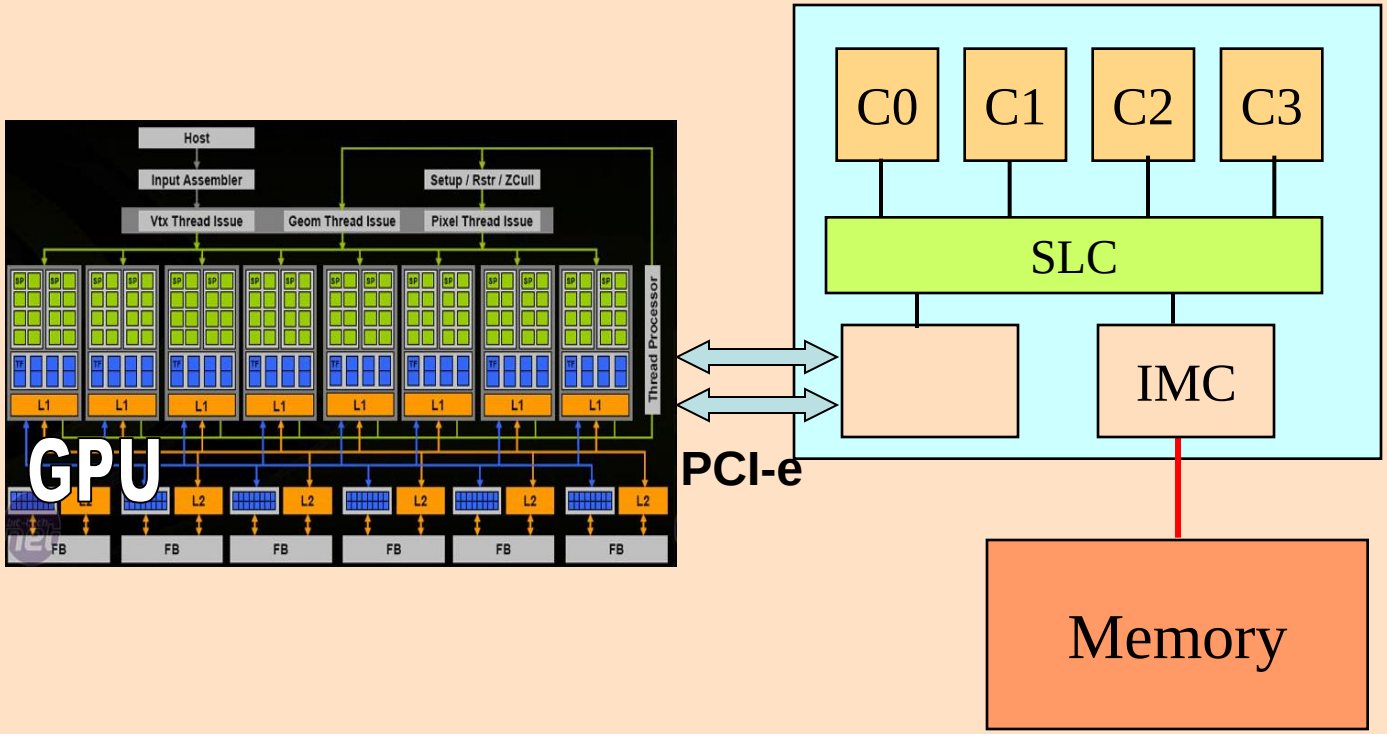

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

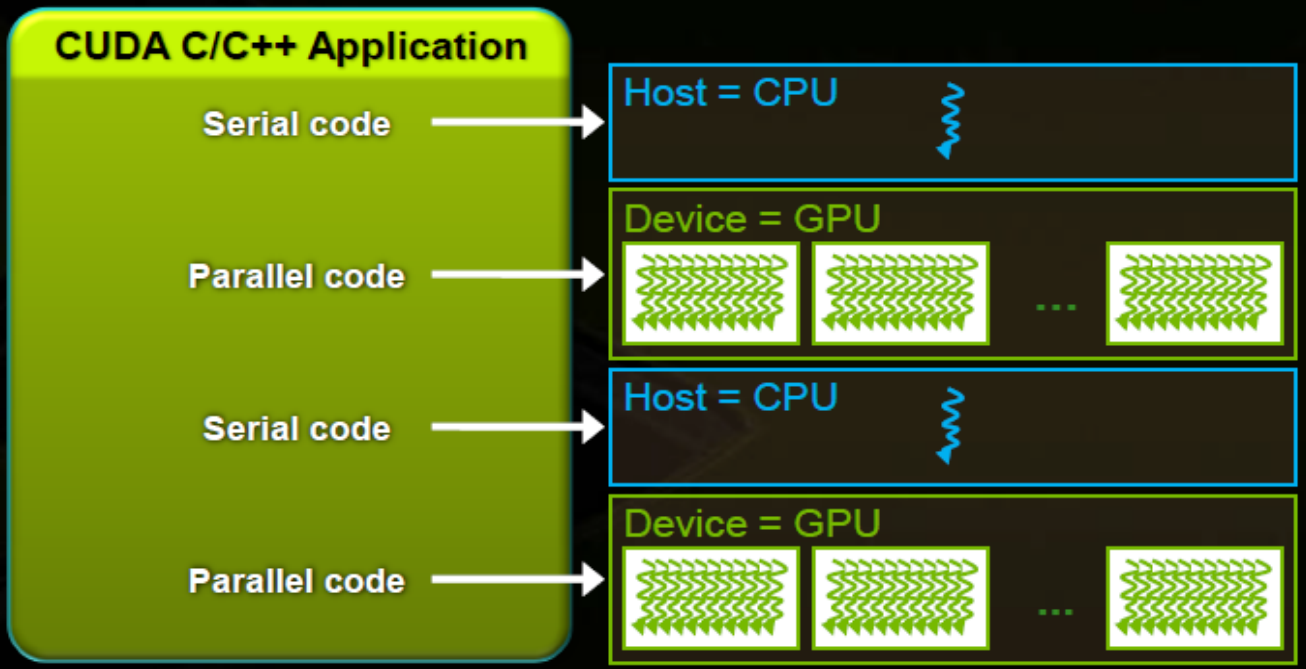
How is the GPU connected?



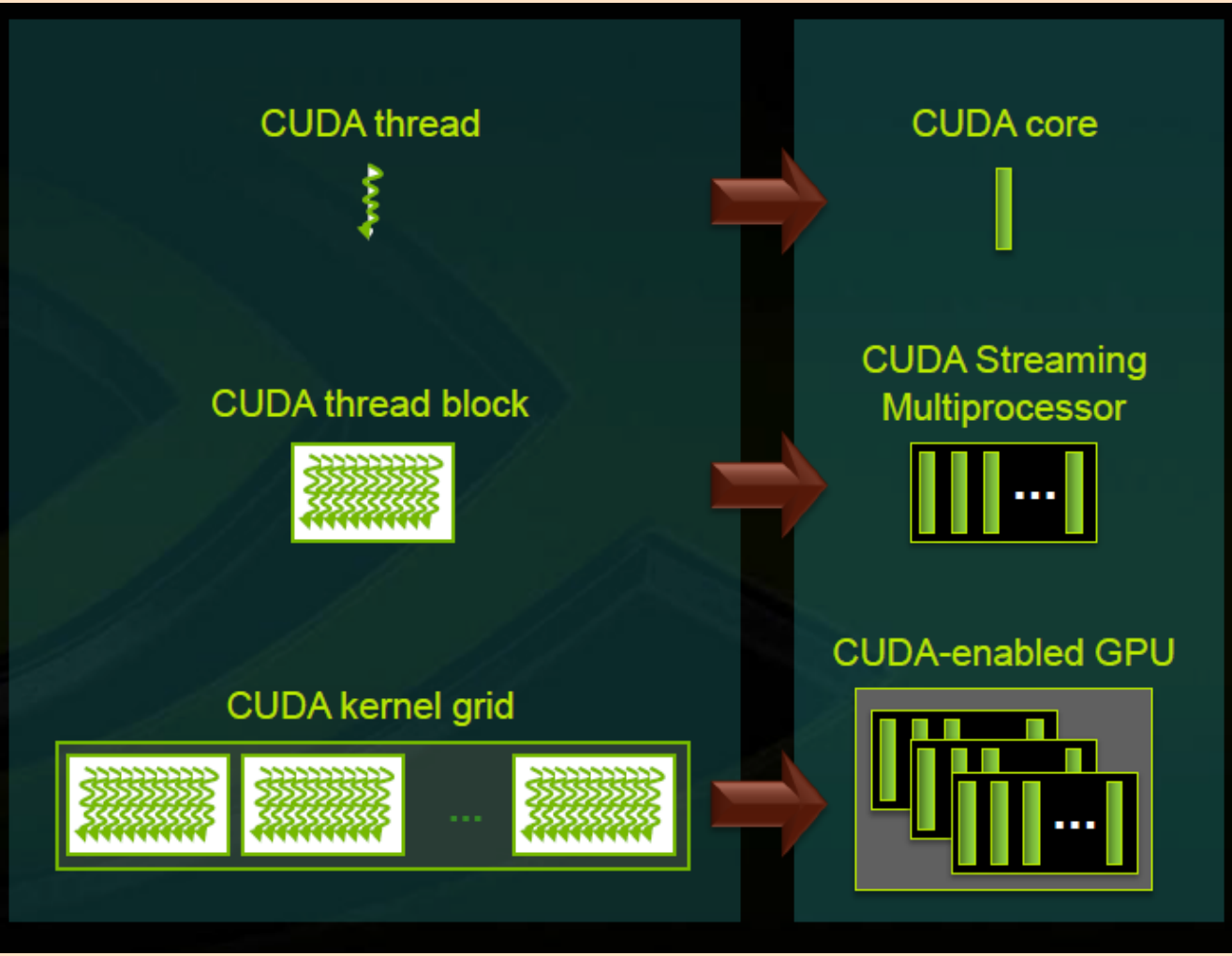
CUDA Programming

Anatomy of a CUDA C/C++ Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements



Kernel Execution

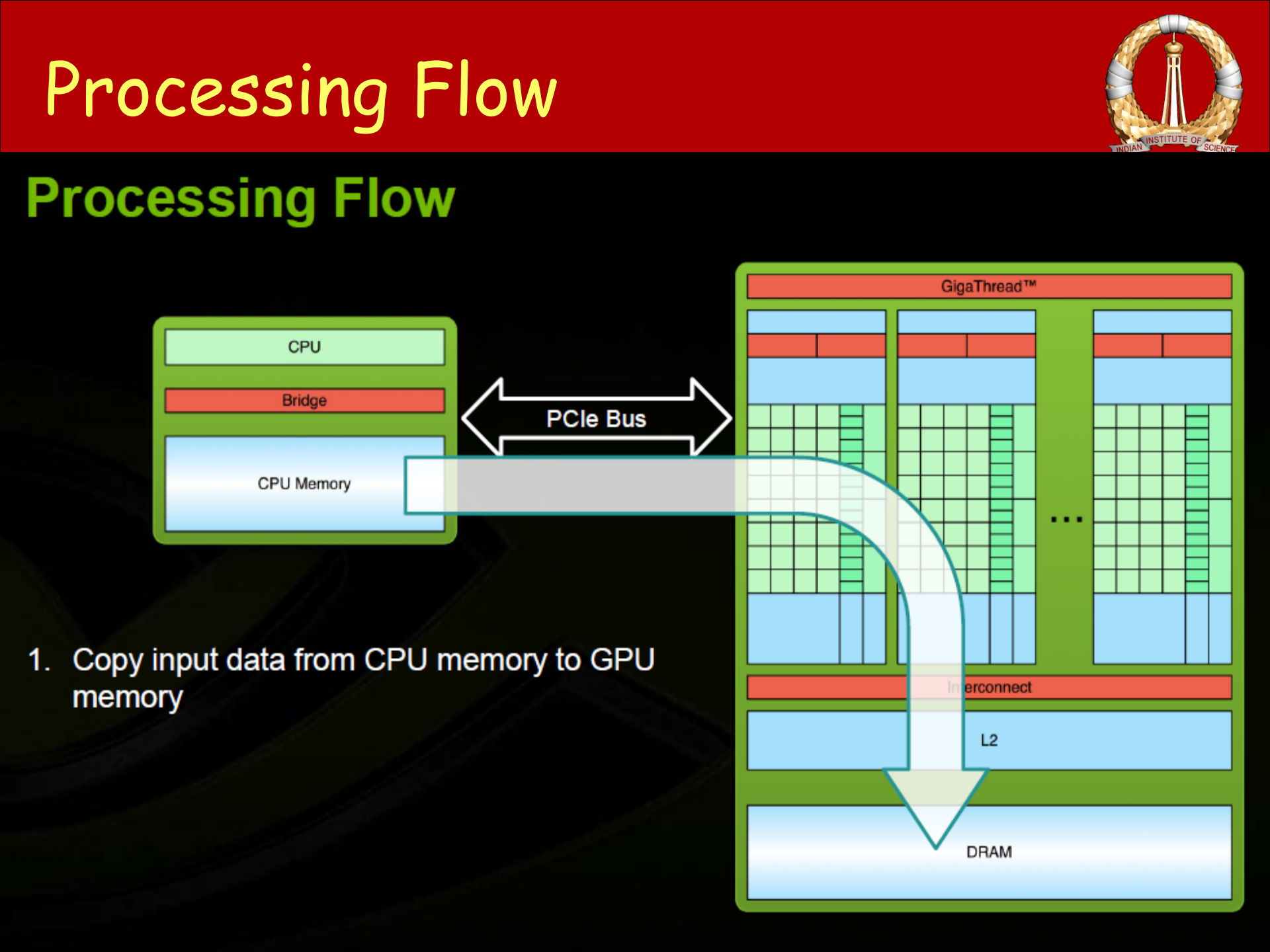


- Each thread is executed by a core

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

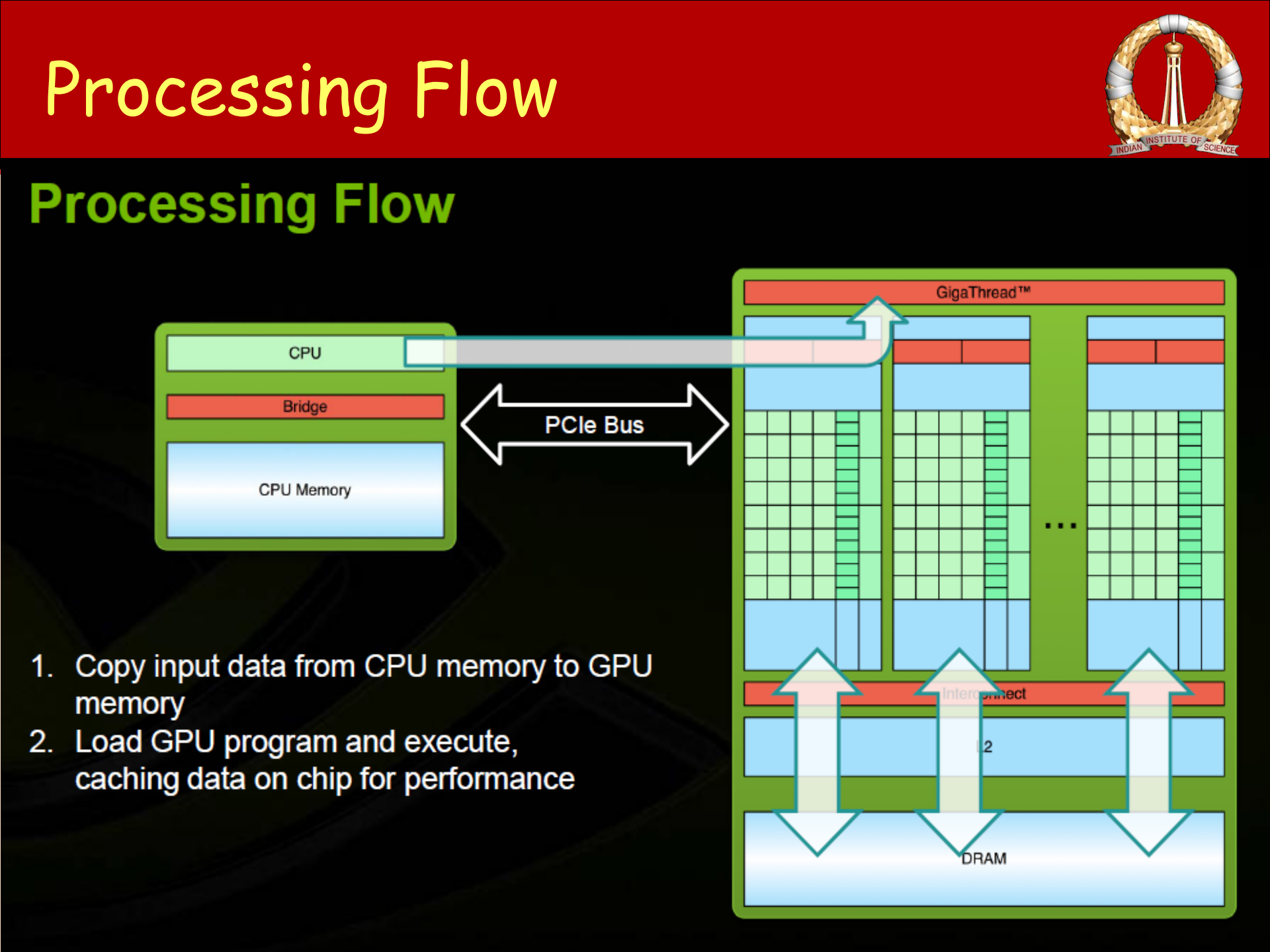
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Processing Flow

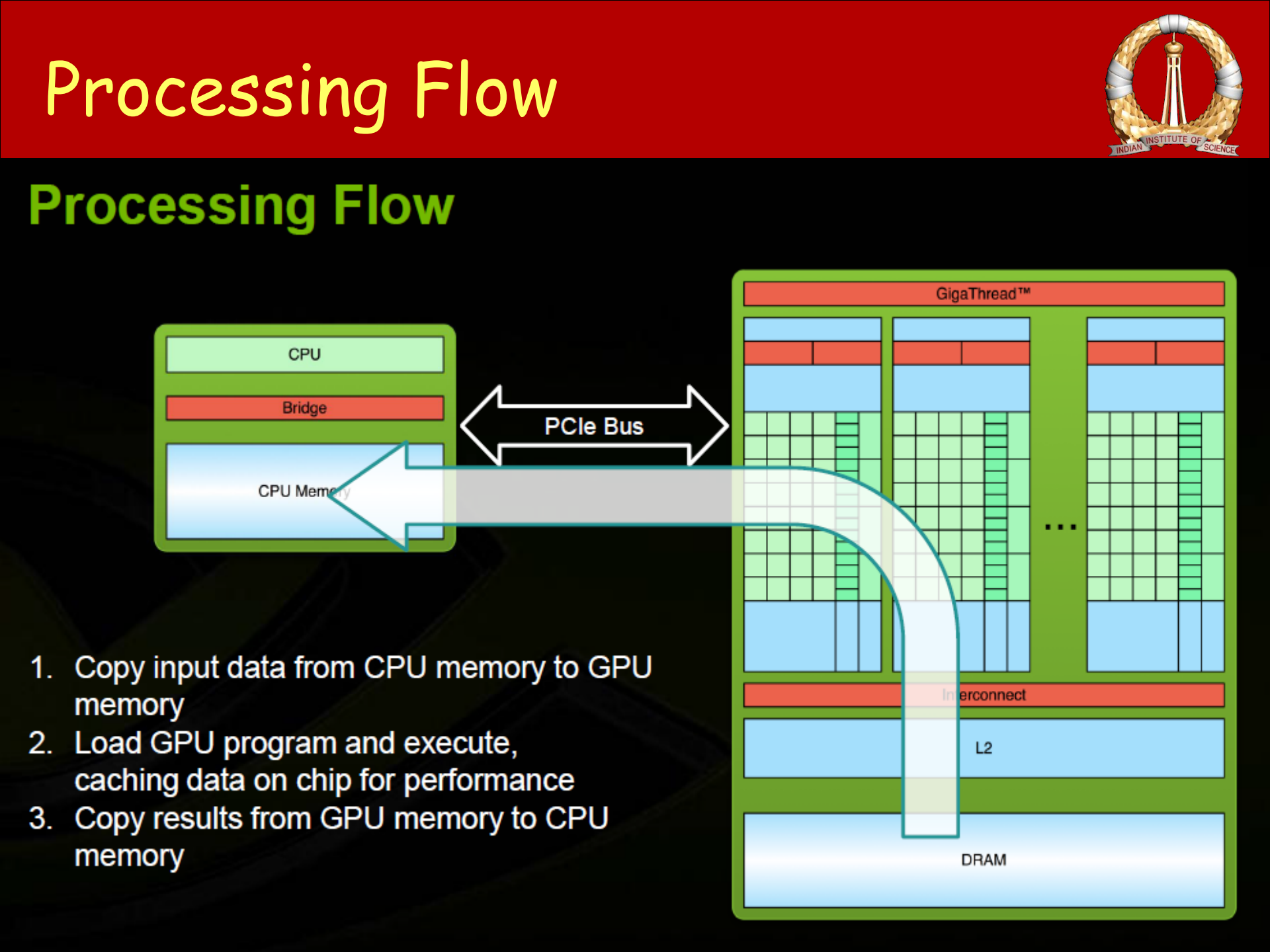


1. Copy input data from CPU memory to GPU memory

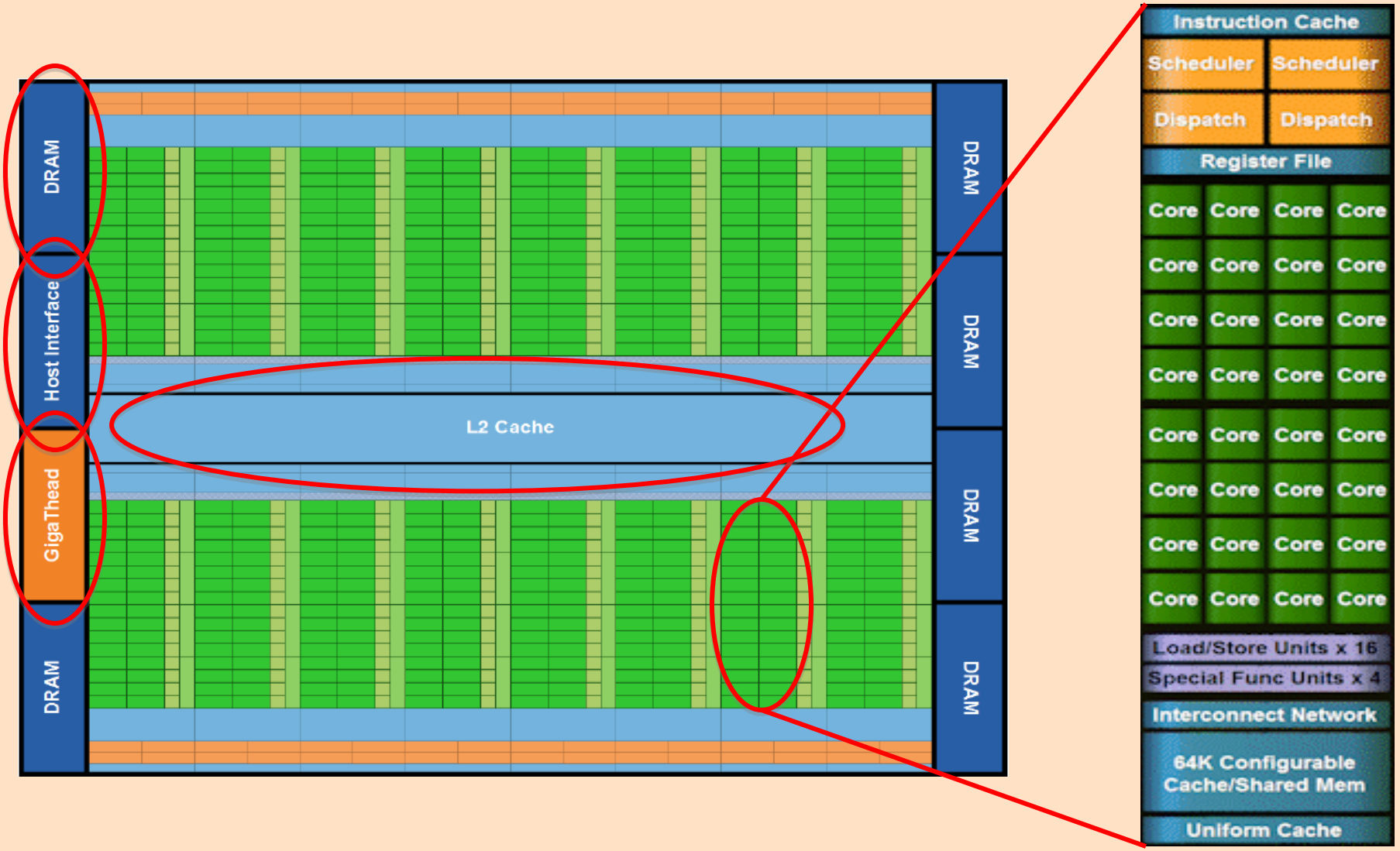
Processing Flow



Processing Flow



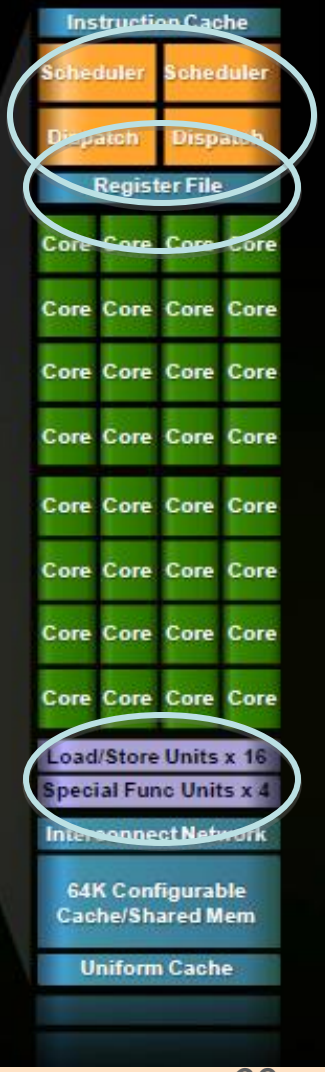
Accelerator - Fermi S2050



Streaming Multiprocessor

GPU Architecture – Fermi: Streaming Multiprocessor (SM)

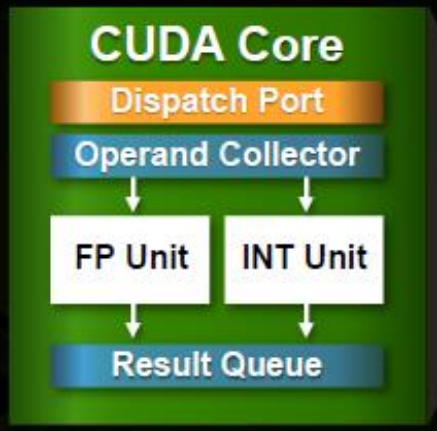
- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers



Inside a CUDA Core

GPU Architecture – Fermi: CUDA Core

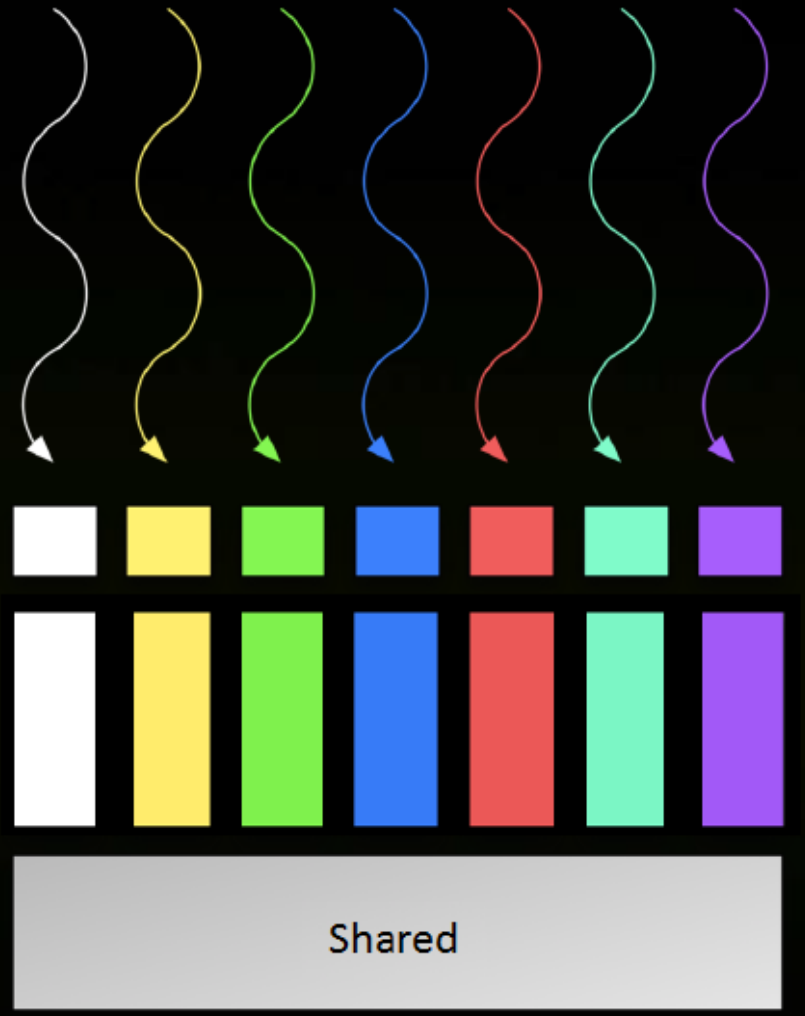
- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit



Memory Hierarchy in CUDA

Memory hierarchy

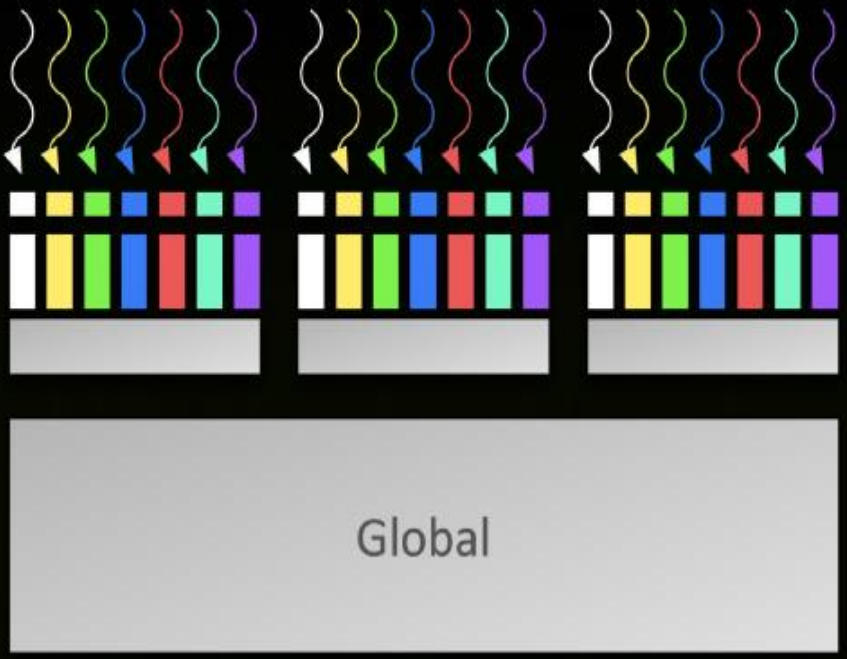
- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory



Global Memory

Memory hierarchy : Global memory

- Accessible by all threads of any kernel
- Data lifetime: from allocation to deallocation by host code
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)
- Latency: 400-800 cycles
- Bandwidth: 156 GB/s
  - Note: requirement on access pattern to reach peak performance



Global Memory Accesses

- Each thread issues memory accesses to data types of varying sizes, perhaps as small as 1 byte entities
- Given an address to load or store, memory returns/updates "segments" of either 32 bytes, 64 bytes or 128 bytes
- Maximizing bandwidth:
  - Operate on an ***entire*** 128 byte segment for each memory transfer

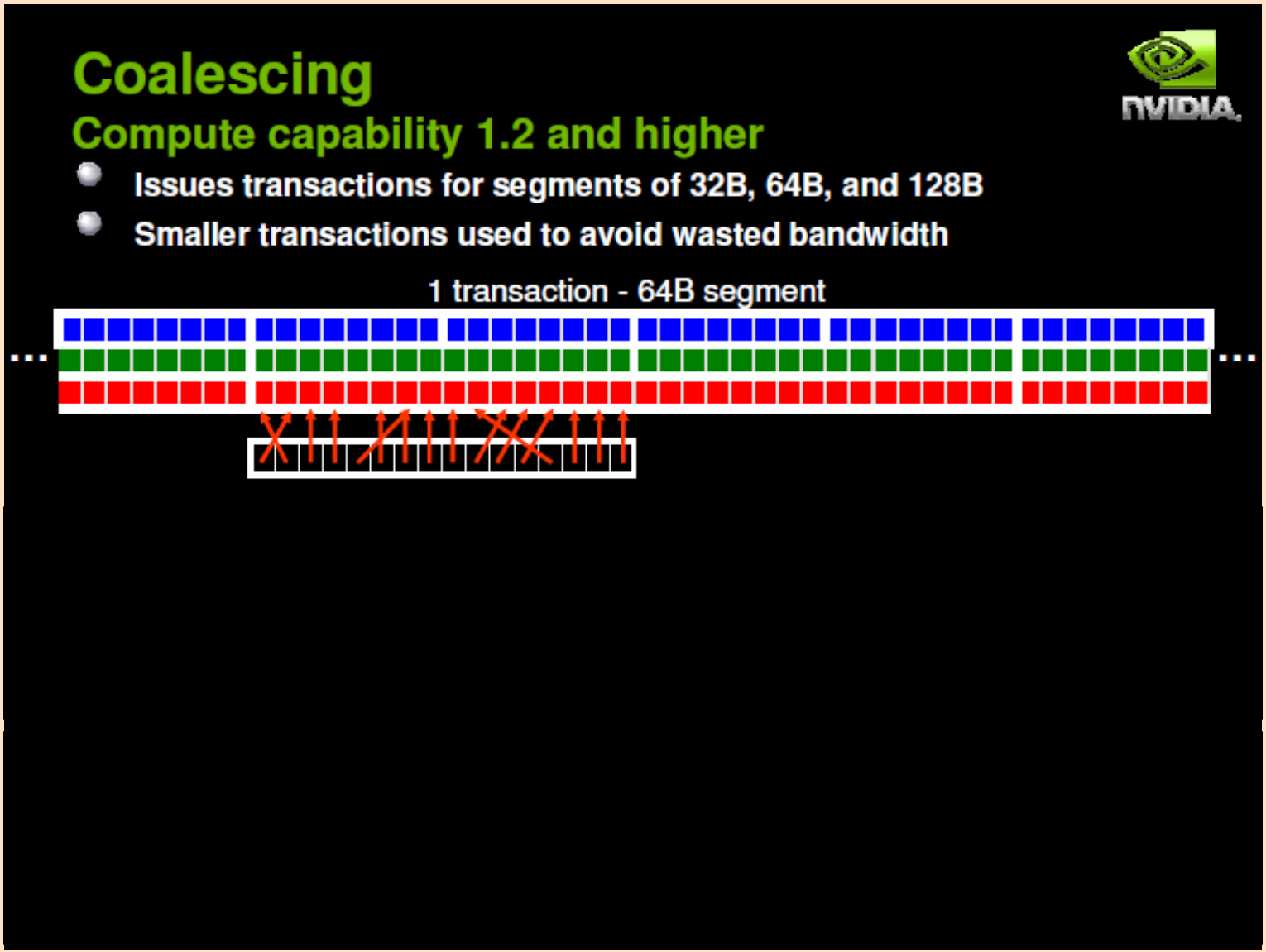
Understanding Global Memory Accesses

Memory protocol for compute capability 1.2 and 1.3* (CUDA Manual 5.1.2.1 and Appendix A.1)

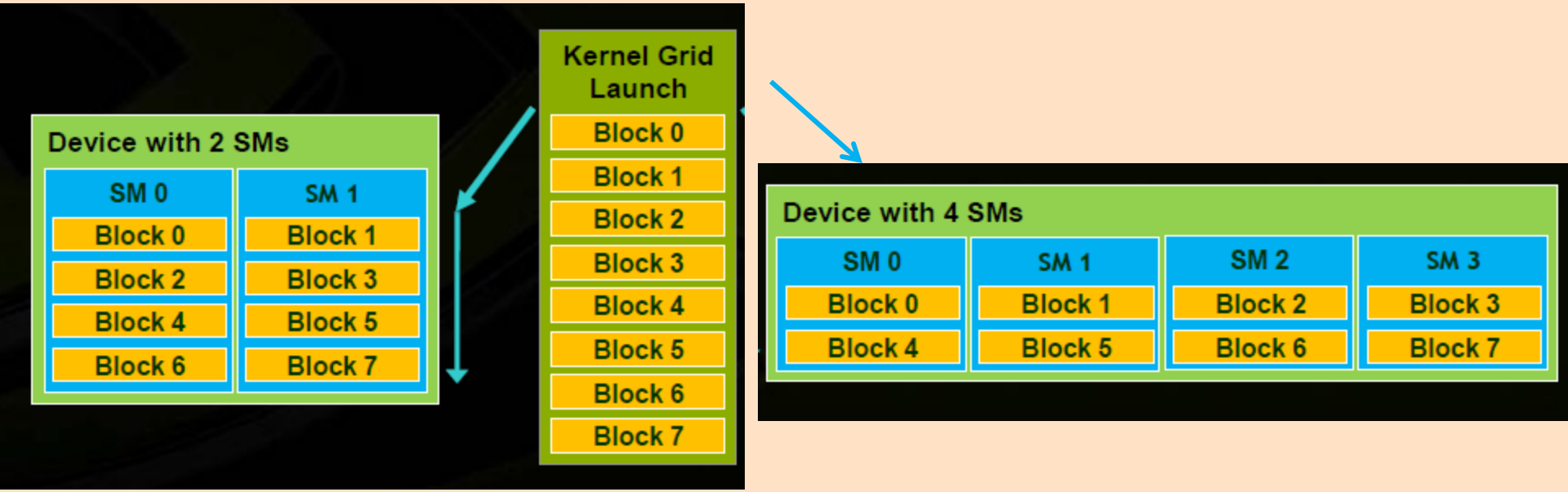
- Start with memory request by smallest numbered thread. Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)
- Find other active threads requesting addresses within that segment and ***coalesce***
- Reduce transaction size if possible
- Access memory and mark threads as "inactive"
- Repeat until all threads ***in half-warp*** are serviced

***Includes Tesla and GTX platforms as well as new Linux machines!**

Memory Coalescing



Threadblock Scheduling



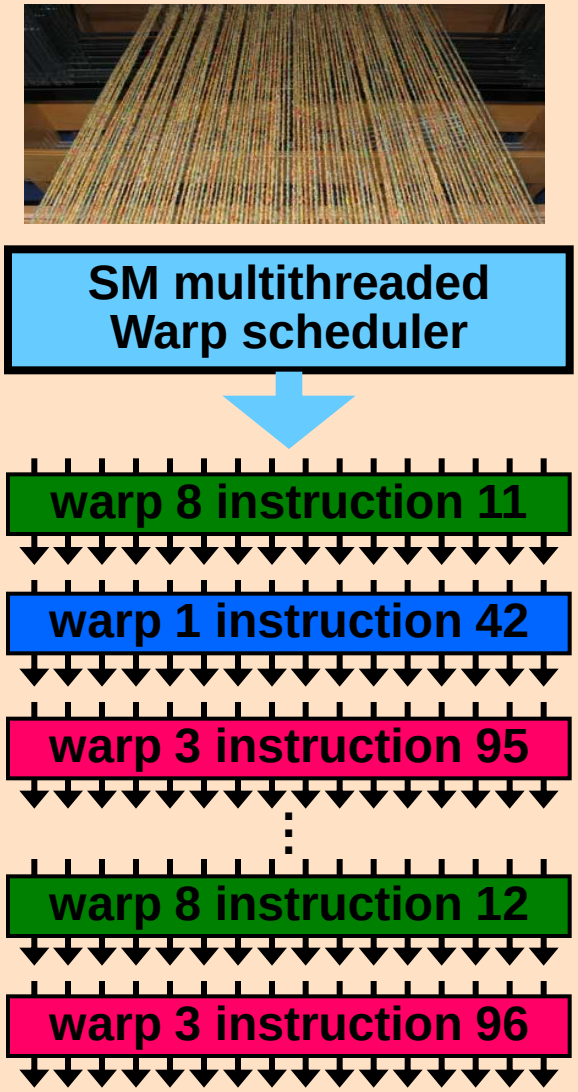
- Each TB allocated to an SM; Multiple TBs in a SM
- Threads in a threadblock may need to cooperate
  - Load/Store from memory together
  - Share results (through shared memory)
  - Synchronize with each other
- Thread Blocks can execute in any order

Resident Thread Blocks

- No. of threadblocks allocated to SM is limited by
  - Max. Threadblocks
  - Max. no. of threads
  - Registers (Regs. per thread * Thrds per Block * # Blks)
  - Shared Memory (Shared Memory per Block * # Blks)
- Warps of resident threadblocks are context switched to hide (long) latency

SM Warp Scheduling

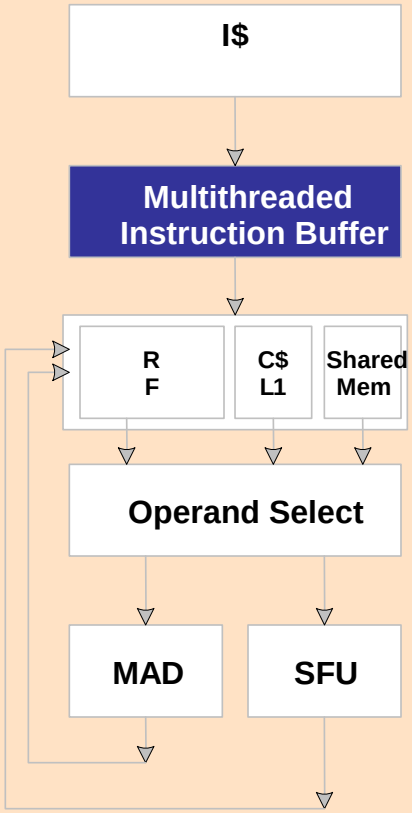
- Each SM launches Warps of Threads
  - 2 levels of parallelism
  - Shared instruction fetch per 32 threads (warps)
- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected



Source : "Kirk-Hwu Lecture Notes"

SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction cache
  - into any instruction buffer slot
- Issue one "ready-to-go" warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand ***scoreboarding*** used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp

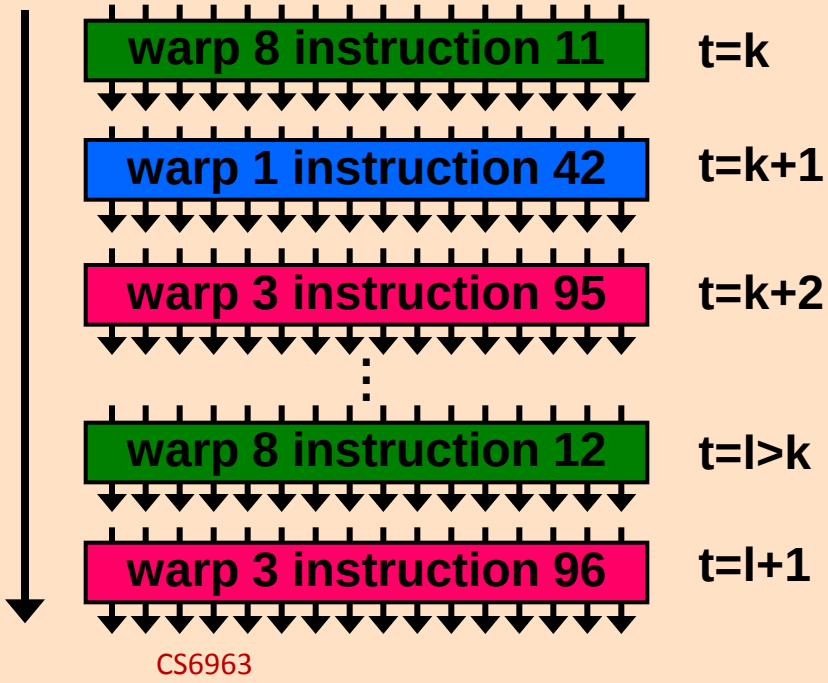


Scoreboarding

- How to determine if a thread is ready to execute?
- A ***scoreboard*** is a table in hardware that tracks
  - instructions being fetched, issued, executed
  - resources (functional units and operands) they need
  - which instructions modify which registers
- Old concept from CDC 6600 (1960s) to separate memory and computation

Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8

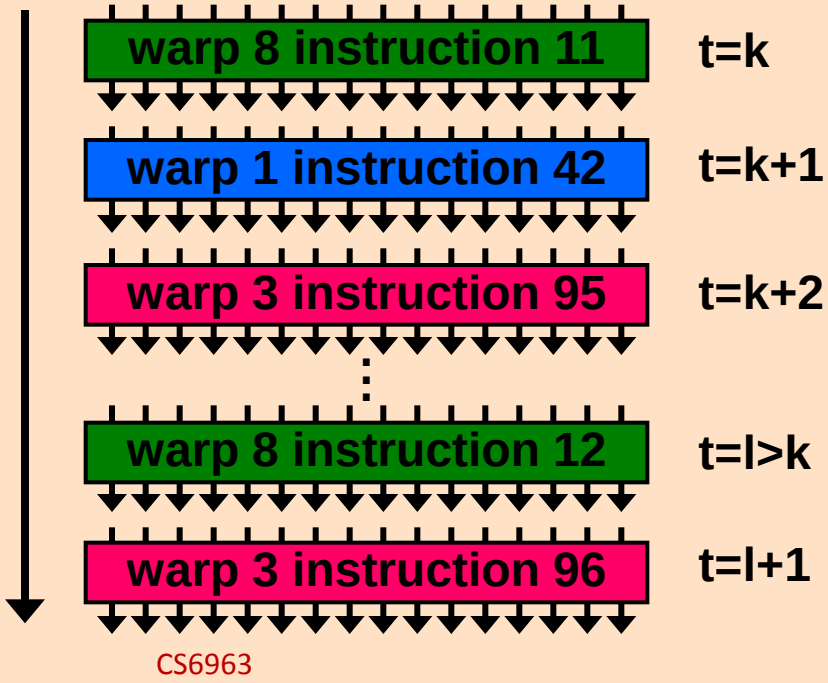


Warp	Current Instruction	Instruction State
Warp 1	42	Computing
Warp 3	95	Computing
Warp 8	11	Operands ready to go
…		

Schedule at time k ←

Scoreboarding from Example

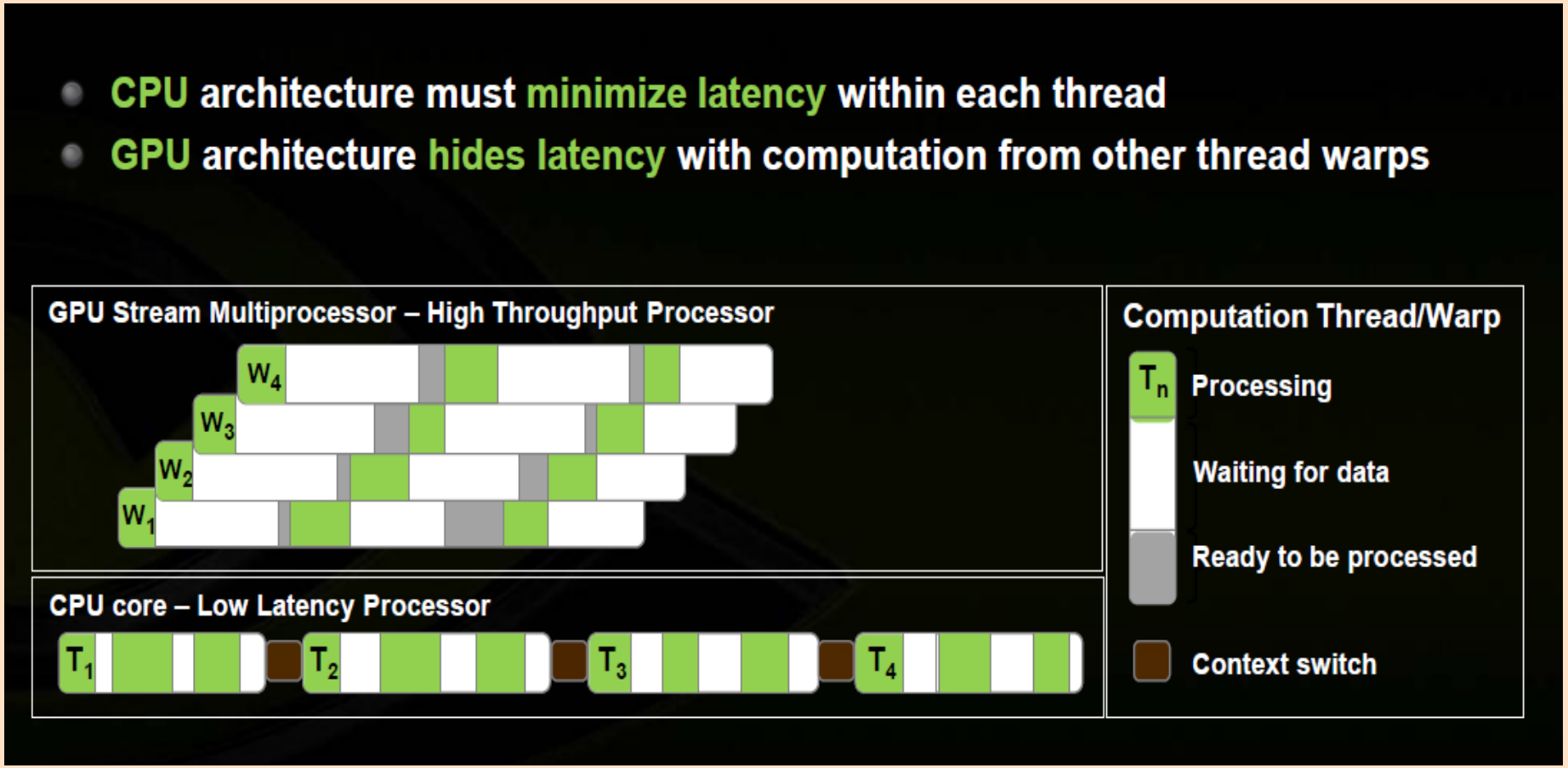
- Consider three separate instruction streams: warp1, warp3 and warp8



Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Computing
Warp 8	11	Computing
…		

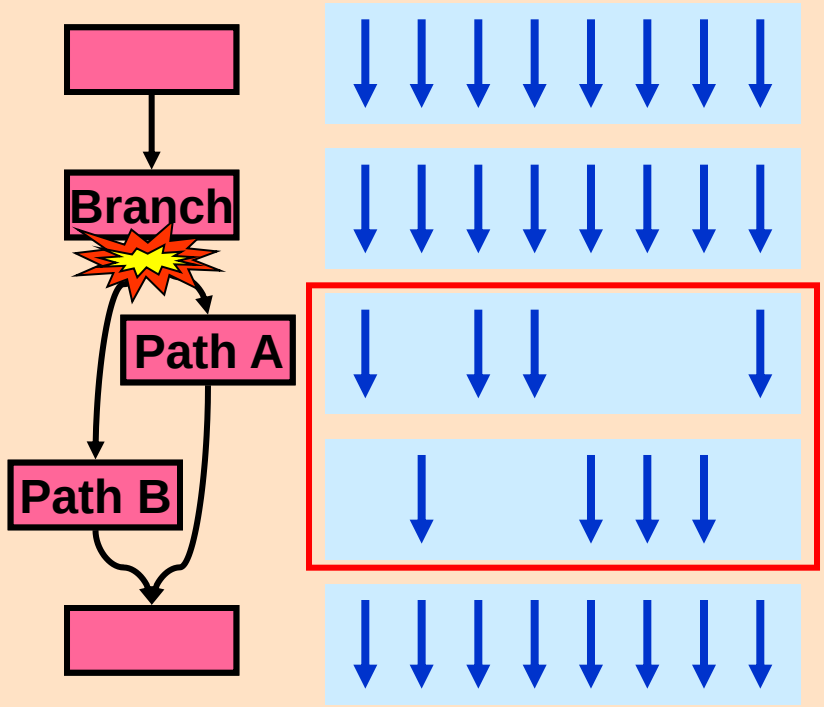
Schedule at time k+1

Context Switching to hide Latency



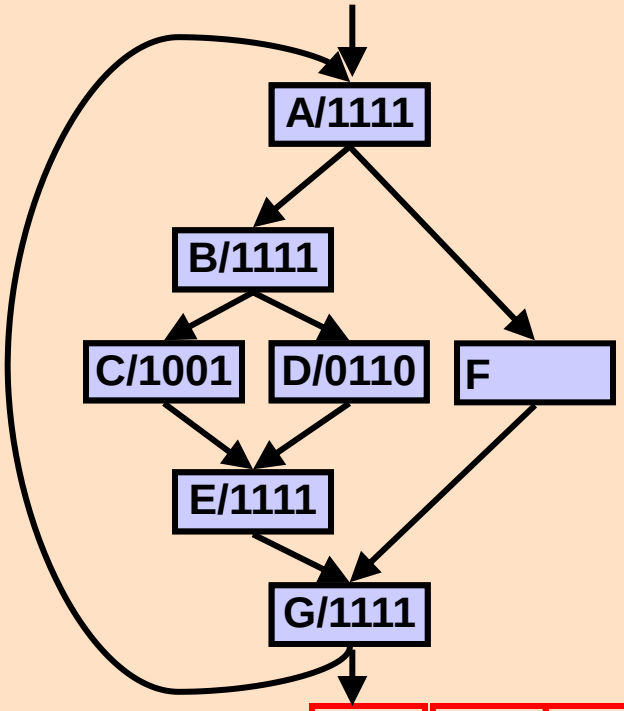
Branch Divergence Problem

- GPU uses SIMD pipeline to save area on control logic.
  - Group scalar threads into warps
- *Branch divergence* occurs when threads inside warps branches to different execution paths.



Performance loss with SIMD width = 16

PDOM-Based Reconvergence

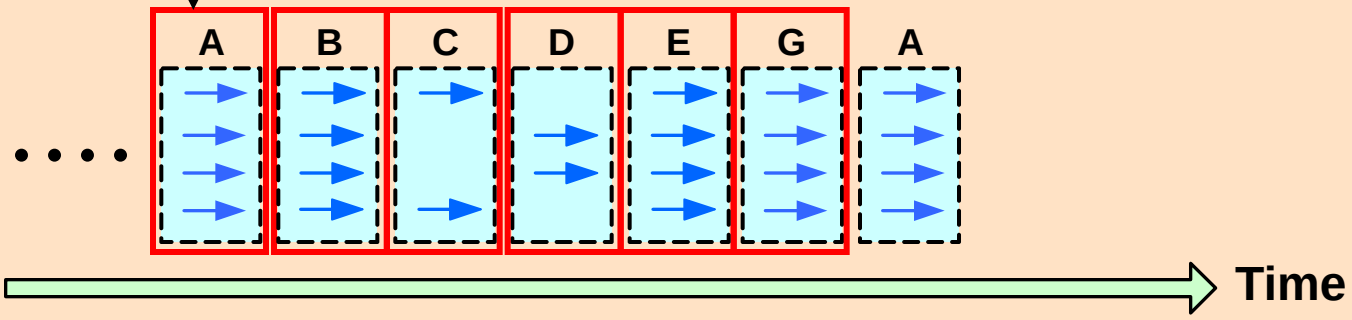


Stack

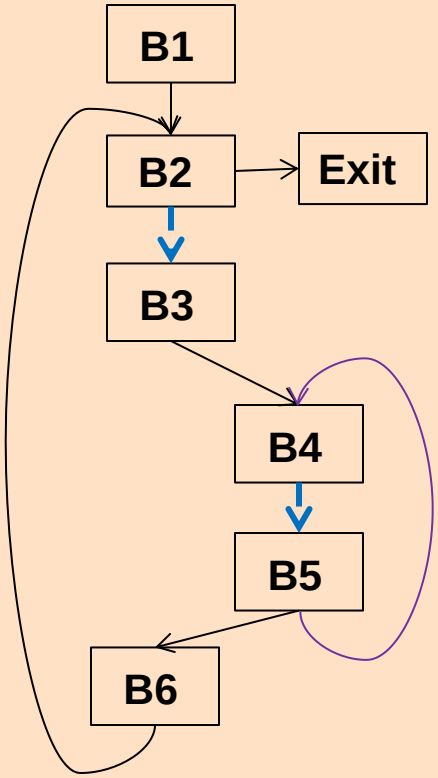
	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001

Thread Warp — **Common PC**

Thread 1	Thread 2	Thread 3	Thread 4



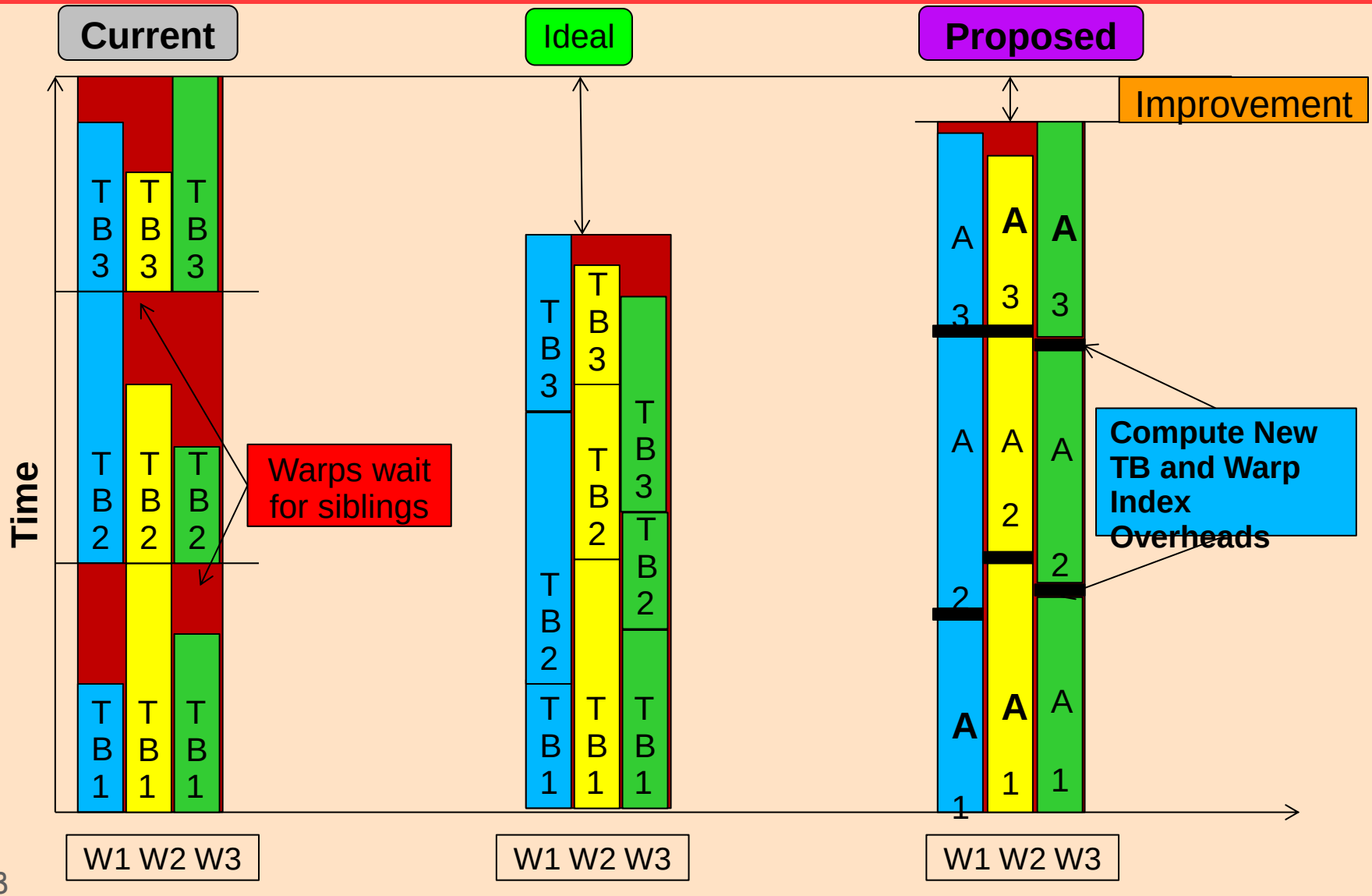
Warp Divergence




```
for (i = 0; i < 2; i++) {
    S1;
    for (j = 0; j < cond; j++)
        S2;
    S3;
}
```

	TB 1	TB 2	TB 3
W1	cond = 5	cond = 15	cond = 10
W2	cond = 15	cond = 10	cond = 5
W3	cond = 10	cond = 5	cond = 15

Warp Divergence



Research work on GPUs

- New warp and threadblock scheduling
- Improving Cache efficiency, prefetching, ..
- Memory management in GPUs
- Programming Languages/compilers for GPUs
- Integrated Heterogeneous Architectures
- Shared resources management
- …