EE382N (20): Computer Architecture - Parallelism and Locality
Fall 2011

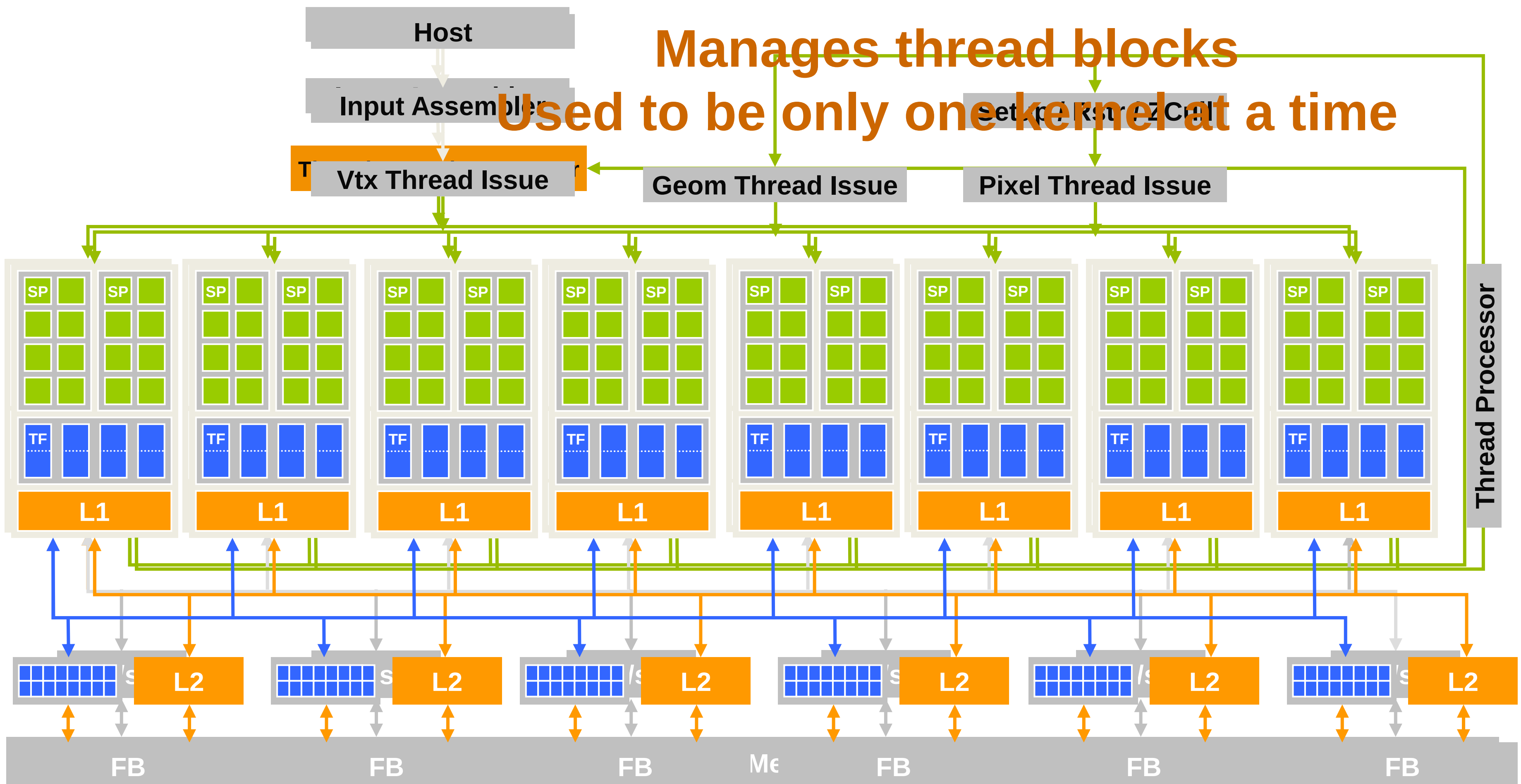
Lecture 18 – GPUs (III)

Mattan Erez

The University of Texas at Austin

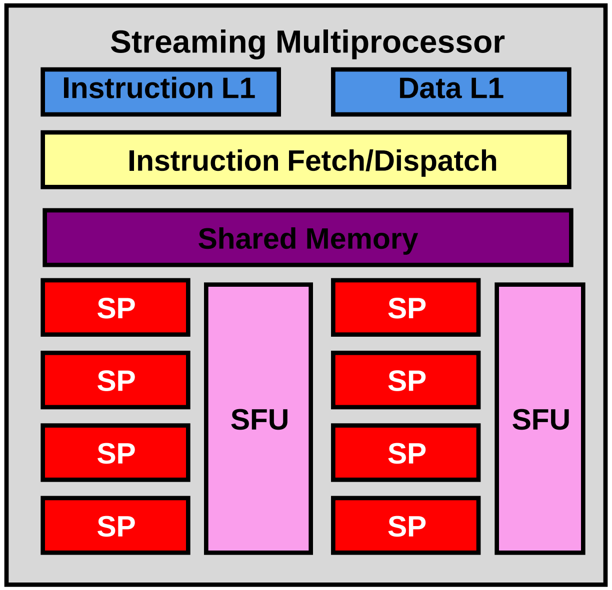
Make the Compute Core The Focus of the Architecture

- The future of GPUs is programmable processing
- Processors execute computing threads
- So – build the architecture around the processor
- Alternative operating mode specifically for computing



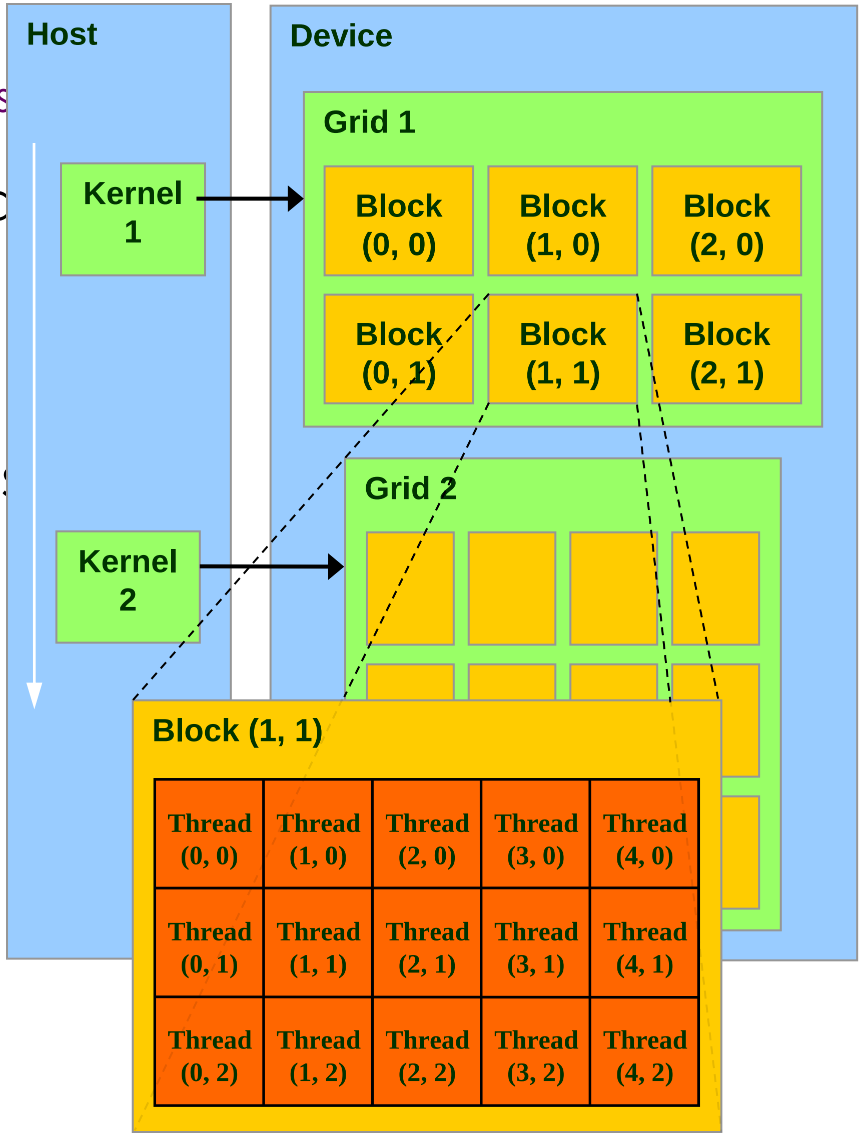
Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - Vectors of 32 threads (***warps***)
  - Up to 16 warps per thread block
    - HW masking of inactive threads in a warp
  - Threads cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- 32 KB in registers
- DRAM texture and memory access

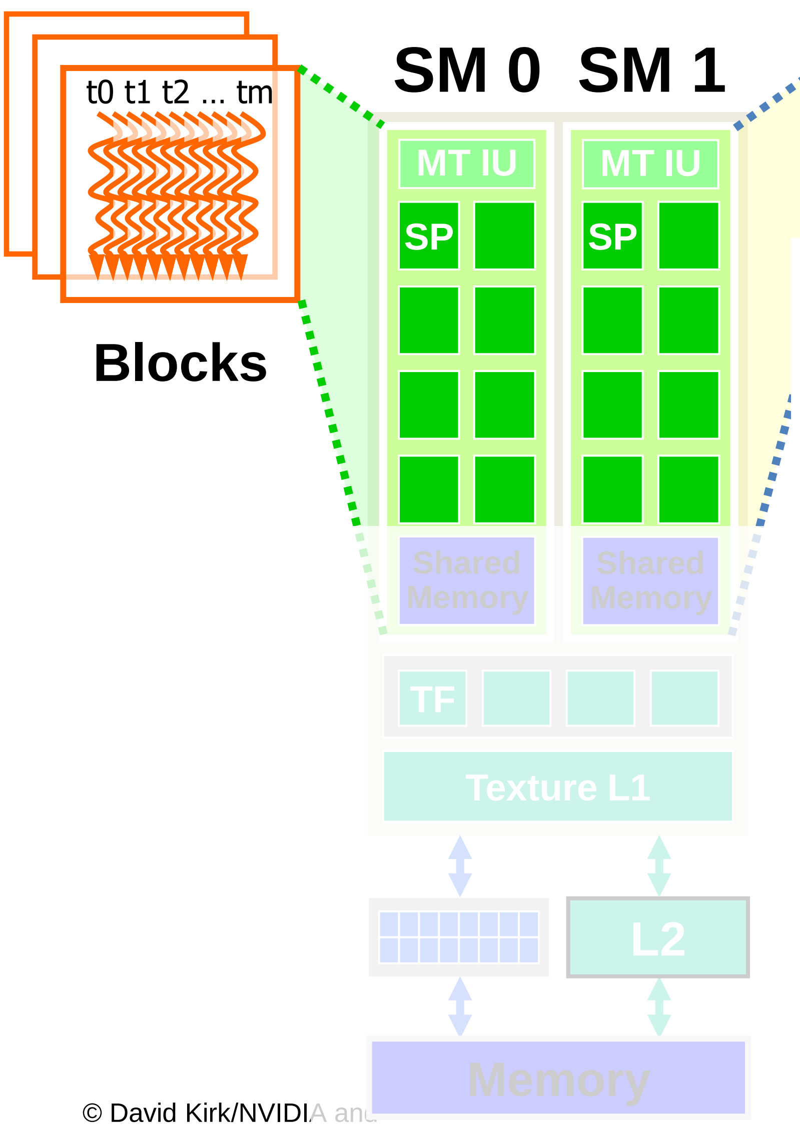


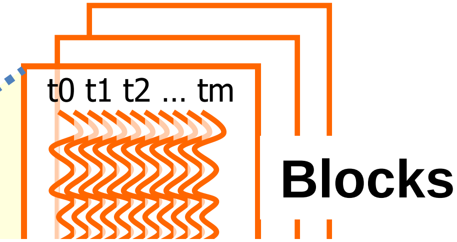
Thread Life Cycle in HW

- Kernel is launched on the SPA
  - Kernels known as ***grids*** of thread blocks
- Thread Blocks are serially distributed to all the SM's
  - Potentially >1 Thread Block per SM
  - At least 96 threads per block
- Each SM launches Warps of Threads
  - 2 levels of parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks



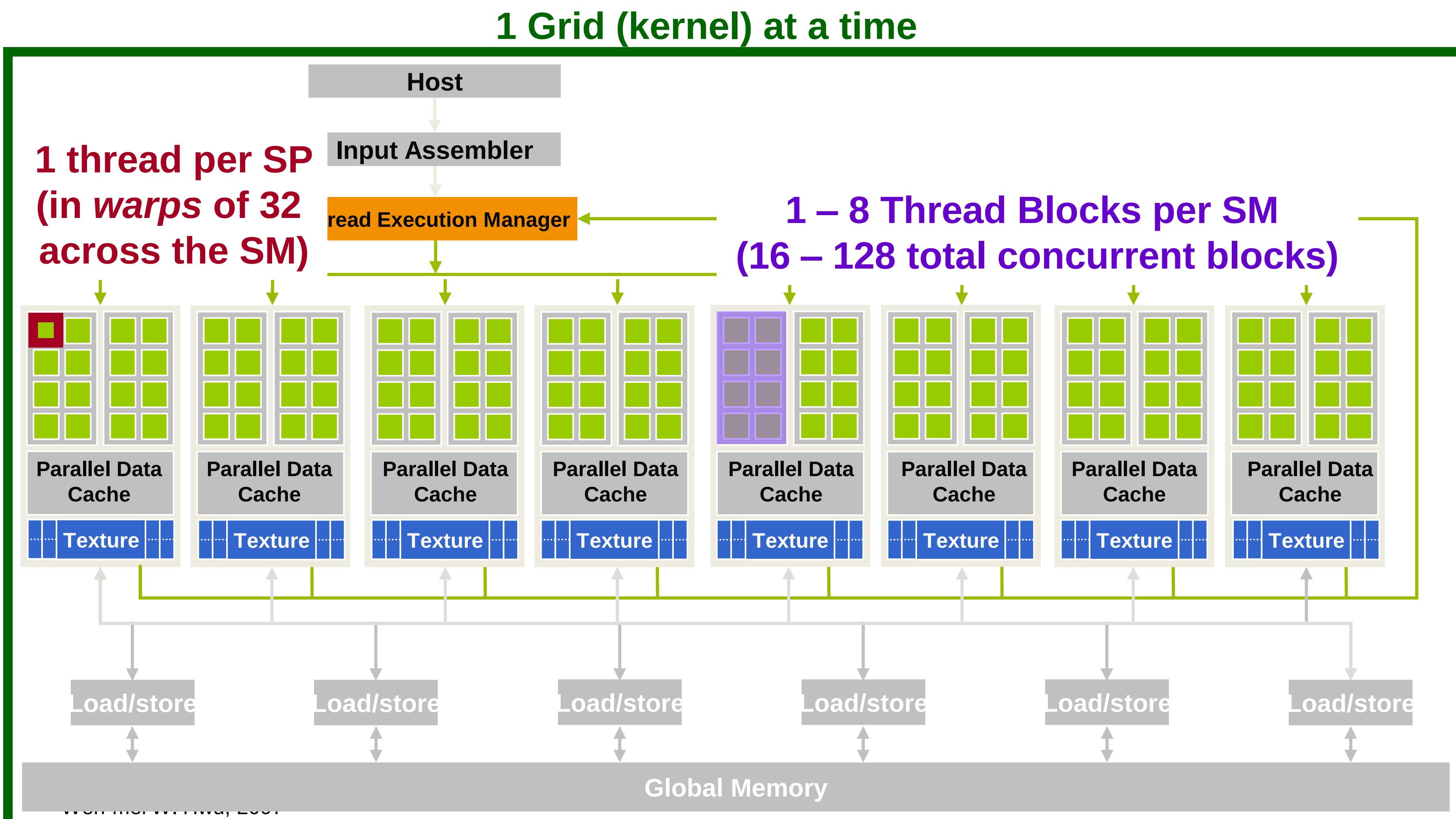
SM Executes Blocks





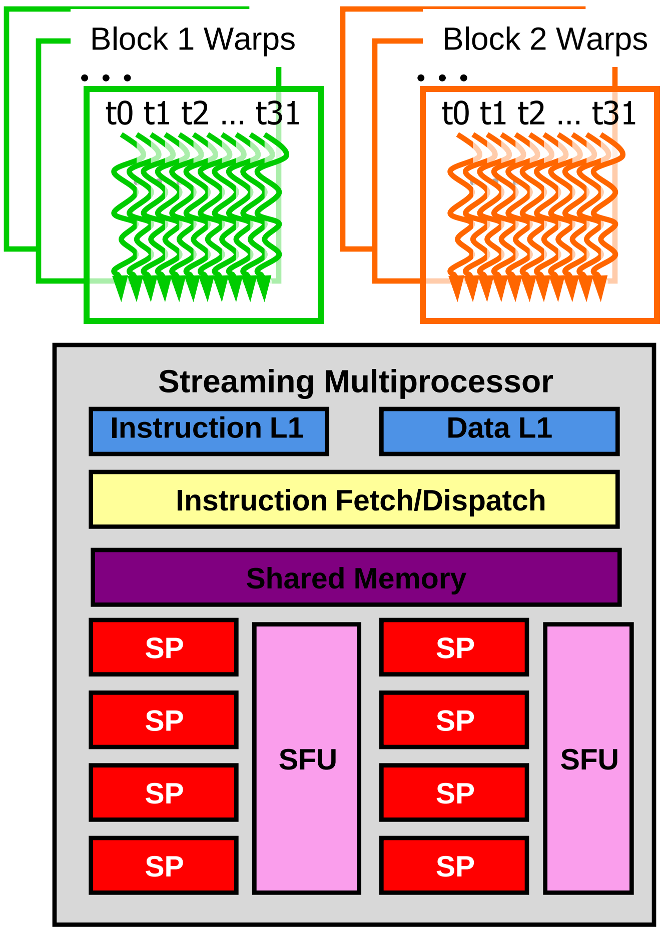
- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM assigns/maintains thread IDs
  - SM manages/schedules thread execution

Make the Compute Core The Focus of the Architecture



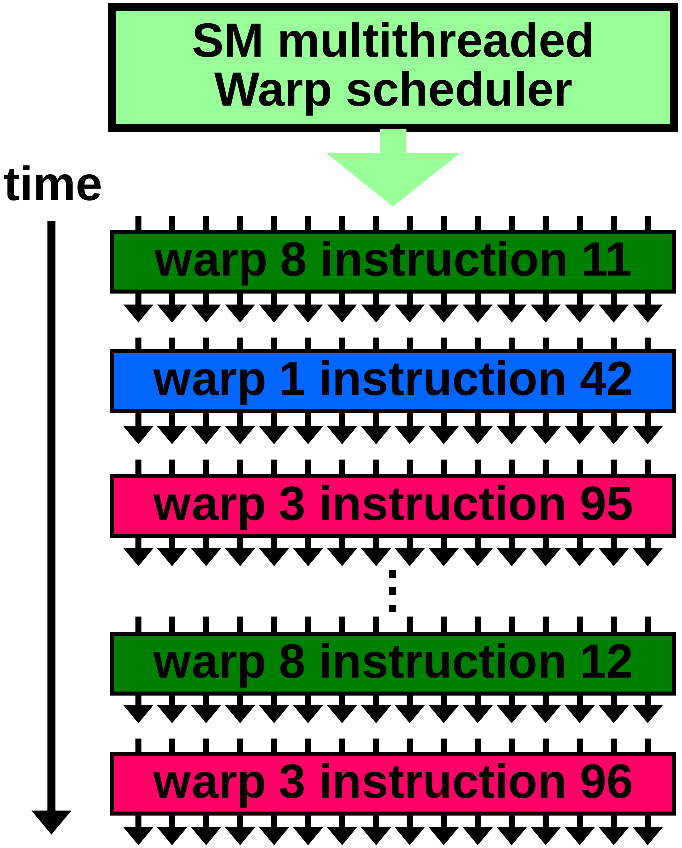
Thread Scheduling/Execution

- Each Thread Block is divided into 32-thread Warps
  - This is an implementation decision
- Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.



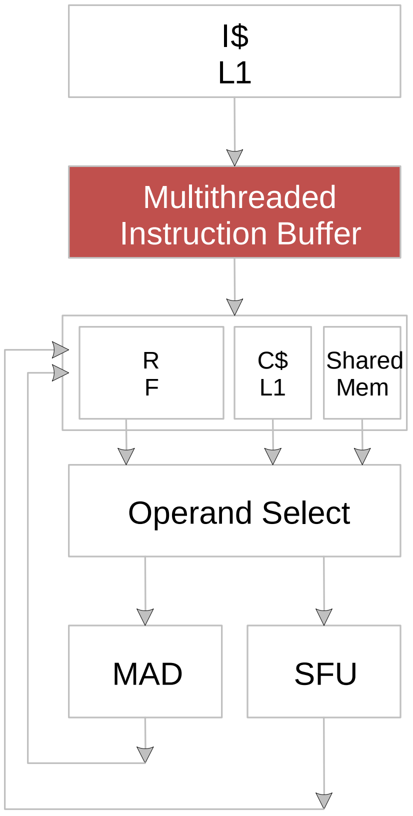
SM Warp Scheduling

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - All threads in a Warp execute the same instruction when selected
  - Scoreboard scheduler
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency



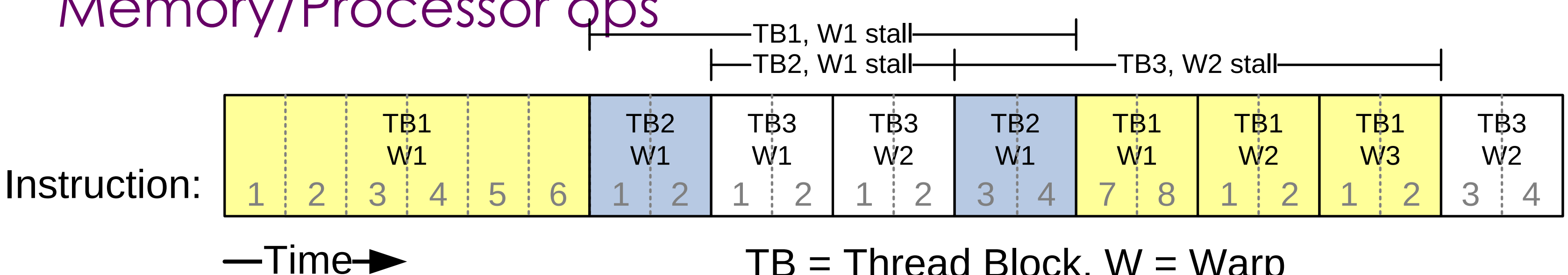
SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction L1 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

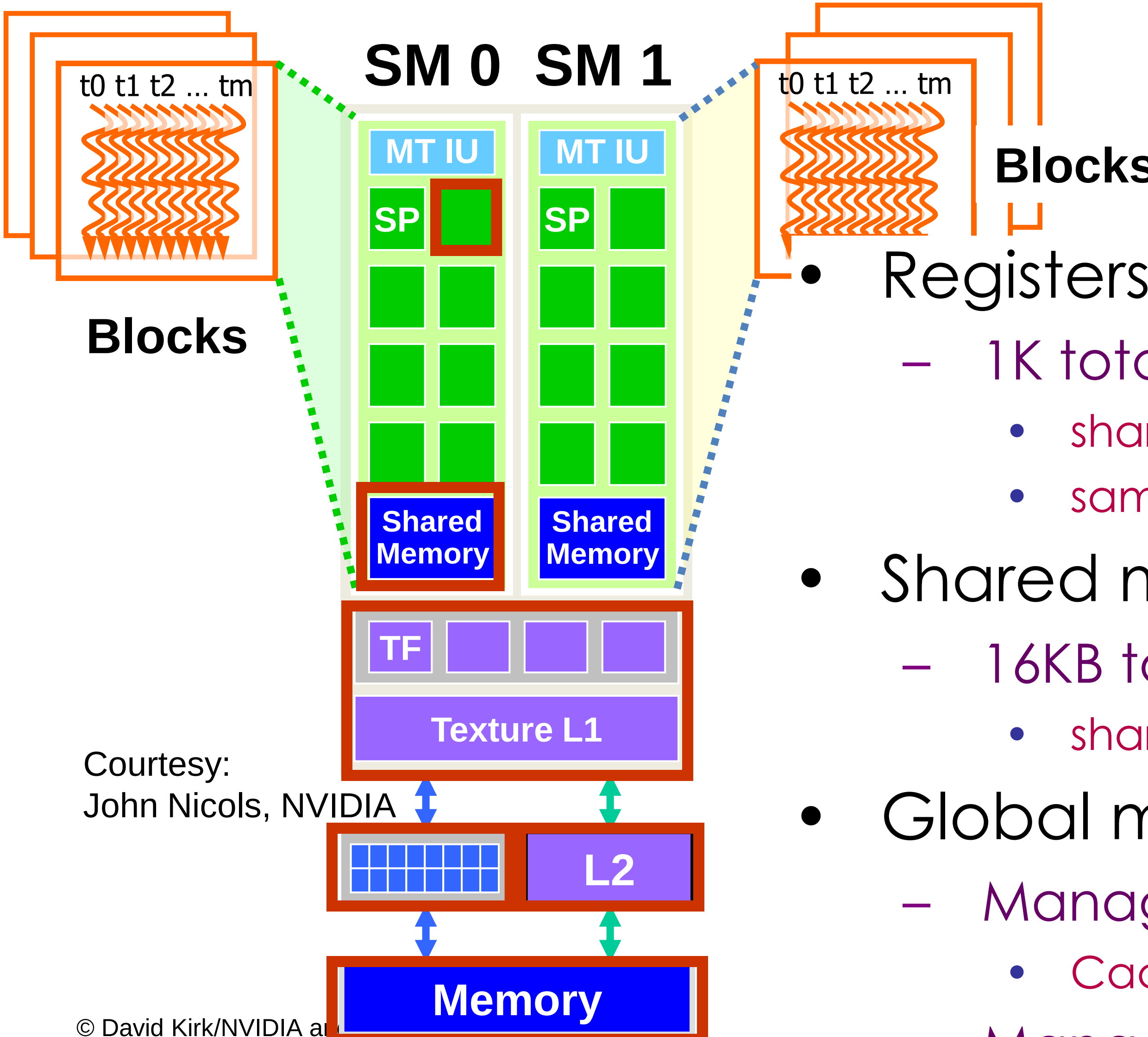
- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes ready after the needed values are deposited
  - prevents hazards
  - cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
  - any thread can continue to issue instructions until scoreboarding prevents issue
  - allows Memory/Processor ops to proceed in shadow of Memory/Processor ops



Granularity and Resource Considerations

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles (1 thread per tile element)?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, it can take up to 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Not even one
    nto an SM!

SM Memory Architecture

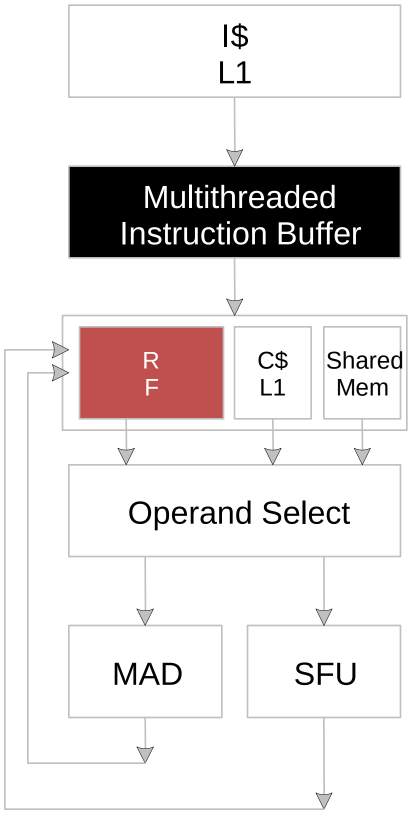


Courtesy:
John Nicols, NVIDIA

- Registers in SP
  - 1K total per SP
    - shared between thread
    - same per thread in a block)
- Shared memory in SM
  - 16KB total per SM
    - shared between blocks
- Global memory
  - Managed by Texture Units
    - Cache – read only
  - Managed by LD/ST ROP units
    - Uncached – read/Write

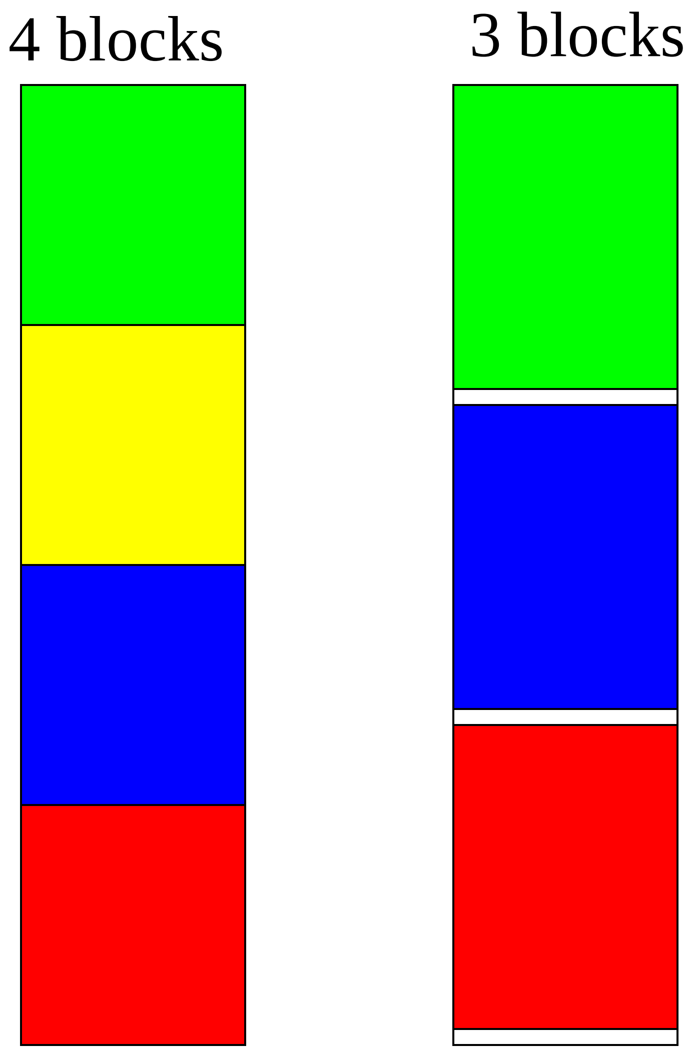
SM Register File

- Register File (RF)
  - 32 KB (1 Kword per SP)
  - Provides 4 operands/clock
- TEX pipe can also read/write RF
  - 2 SMs share 1 TEX
- Load/Store pipe can also read/write RF



Programmer View of Register File

- There are 8192 registers in each SM in G80
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all Blocks assigned to the SM
  - Once assigned to a Block, the register is NOT accessible by threads in other Blocks
  - Each thread in the same ...ly access ... assigned to itself



Matrix Multiplication Example

- If each Block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?
  - Each Block requires 10*256 = 2560 registers
  - 8192 = **3** * 2560 + change
  - So, three blocks can run on an SM as far as registers are concerned
- How about if each thread increases the use of registers by 1?
  - Each Block now requires 11*256 = 2816 registers
  - 8192 < 2816 *3
  - Only two Blocks can run on an SM, **1/3 reduction of parallelism**!!!

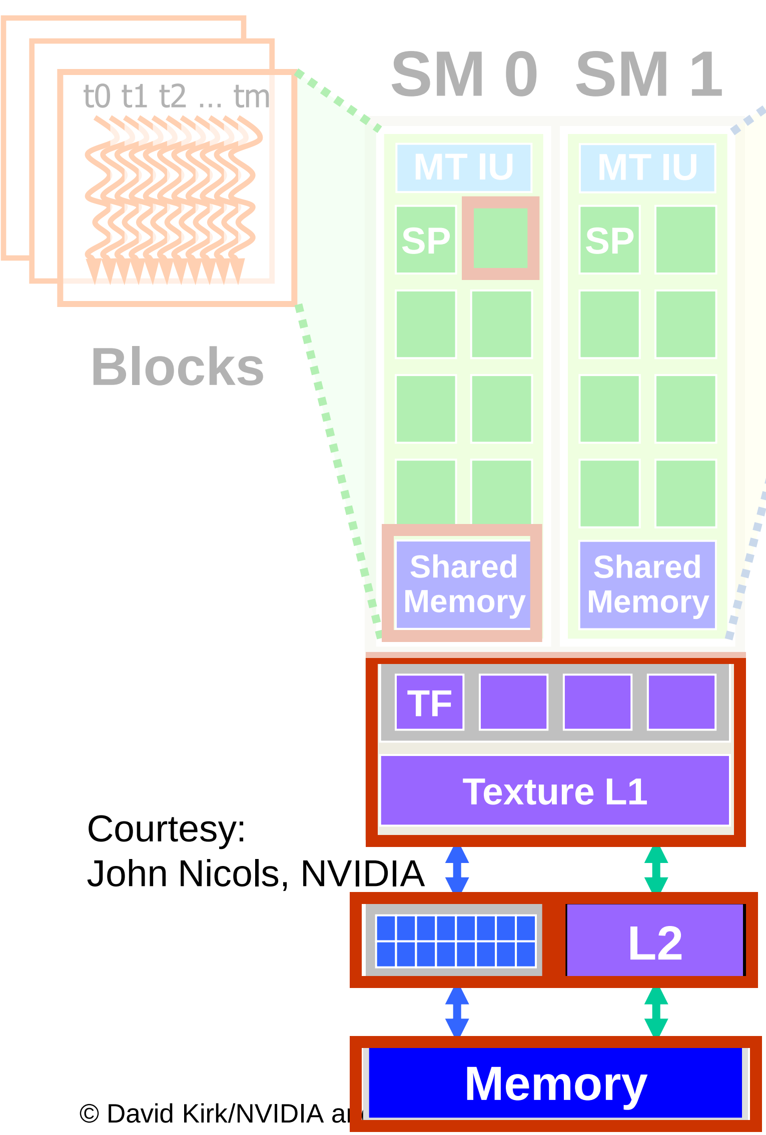
More on Dynamic Partitioning

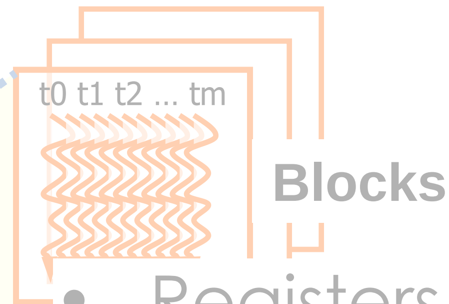
- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between instruction-level parallelism and thread level parallelism

ILP vs. TLP Example

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 10 registers, global loads have 200 cycles
  - 3 Blocks can run on each SM
- If a Compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load
  - Only two can run on each SM
  - However, one only needs 200/(8*4) = 7 Warps to tolerate the memory latency
  - Two Blocks have 16 Warps. The performance can actually be higher!

SM Memory Architecture



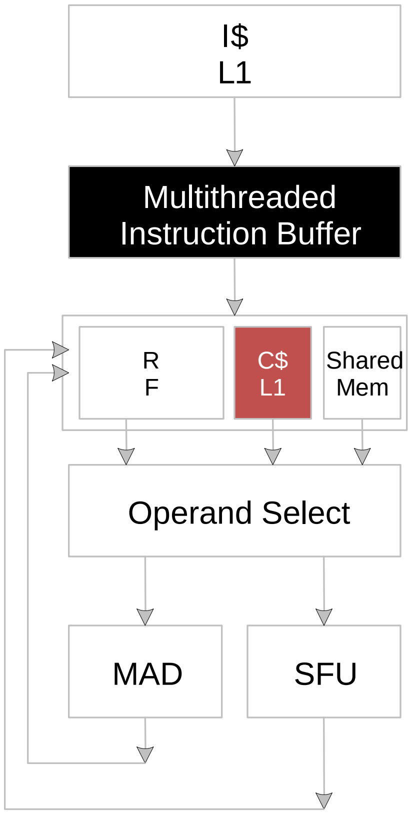


Courtesy:
John Nicols, NVIDIA

- Registers in SP
  - 1K total per SP
    - shared between thread
    - same per thread in a block)
- Shared memory in SM
  - 16KB total per SM
    - shared between blocks
- Global memory
  - Managed by Texture Units
    - Cache – read only
  - Managed by LD/ST ROP units
    - Uncached – read/Write

Constants

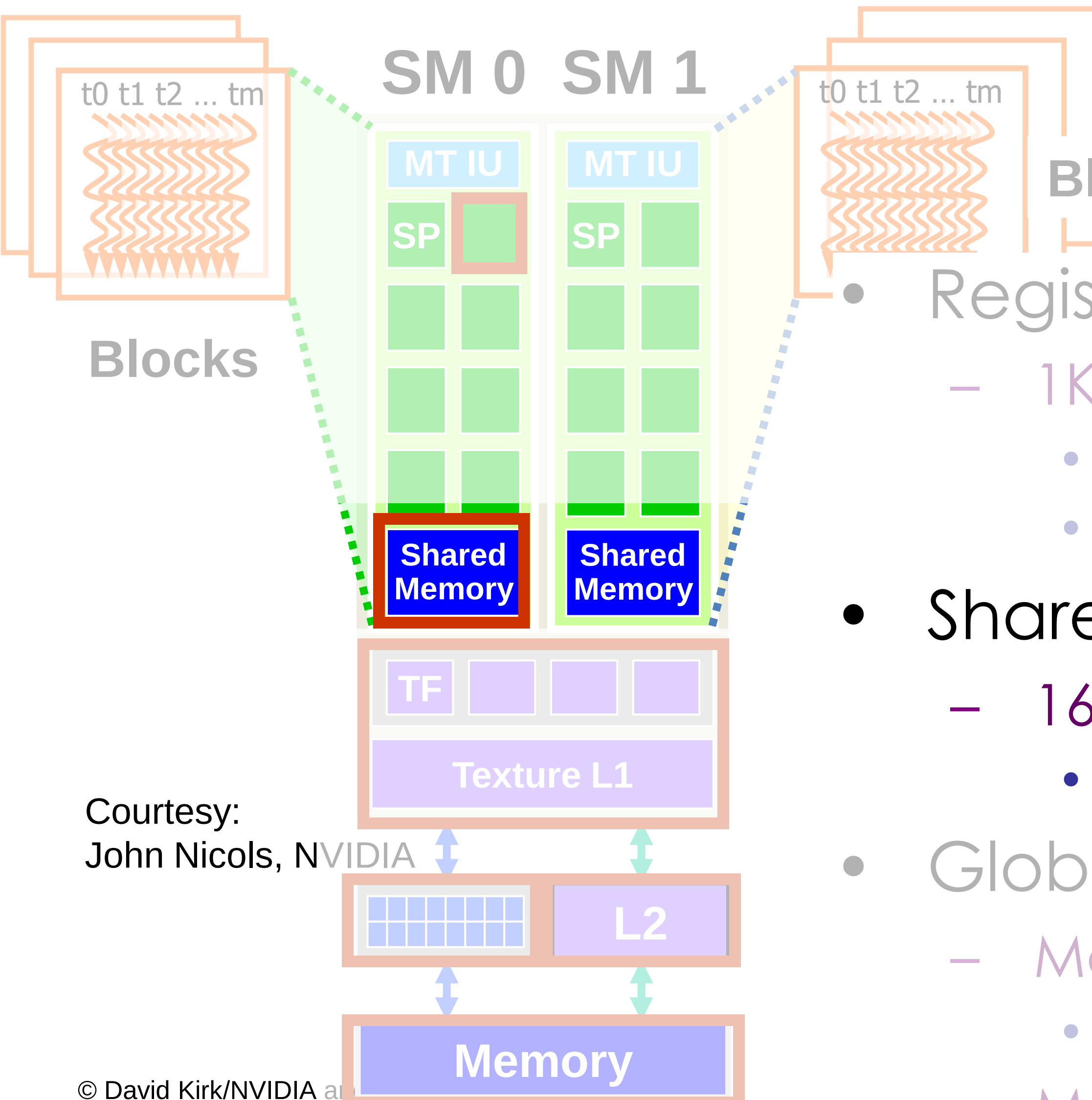
- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!



Textures

- Textures are 2D arrays of values stored in global DRAM
- Textures are cached in L1 and L2
- Read-only access
- Caches optimized for 2D access:
  - Threads in a warp that follow 2D locality will achieve better memory performance

SM Memory Architecture



Courtesy:
John Nicols, NVIDIA

- Registers in SP
  - 1K total per SP
    - shared between thread
    - same per thread in a block)
- Shared memory in SM
  - 16KB total per SM
    - shared between blocks
- Global memory
  - Managed by Texture Units
    - Cache – read only
  - Managed by LD/ST ROP units
    - Uncached – read/Write

Shared Memory

- Each SM has 16 KB of Shared Memory
  - 16 banks of 32bit words
- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - read and write access
- Not used explicitly for pixel shader programs
  - we dislike pixels talking to each other ☹