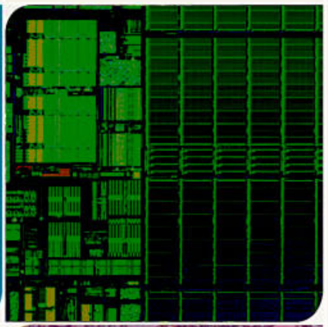
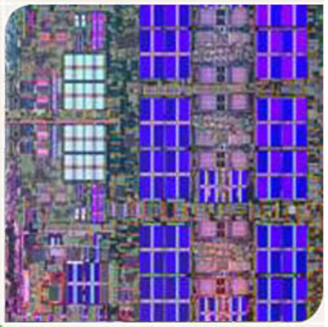
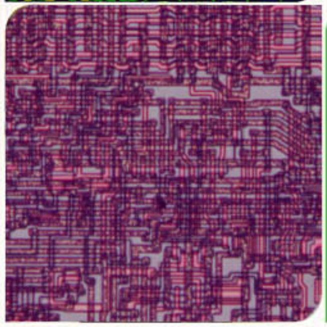
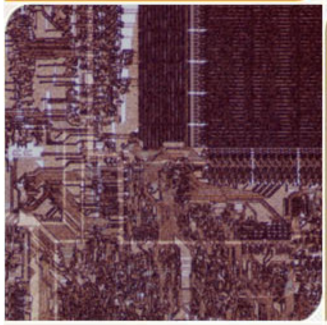





CS4803DGC Design Game Consoles

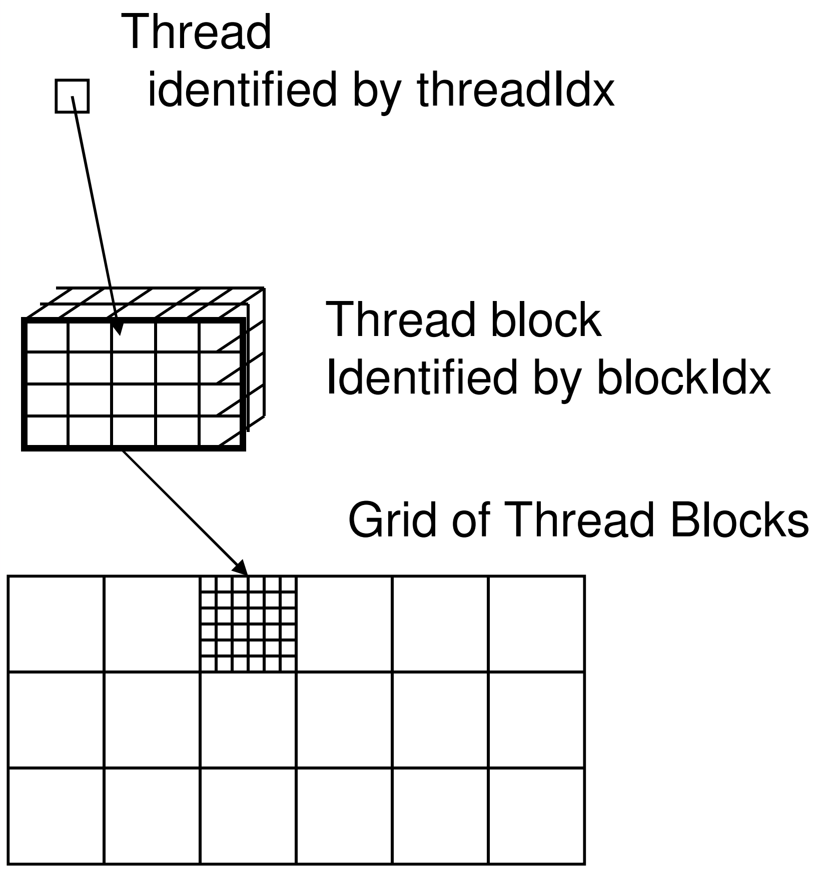
Spring 2009

Prof. Hyesoon Kim



Georgia Tech College of Computing

Execution Model



Multiple levels of parallelism

-Thread block

 -Up to 512 threads per block

 -Communicate through shared memory

 -Threads guaranteed to be resident

 -threadIdx, blockIdx

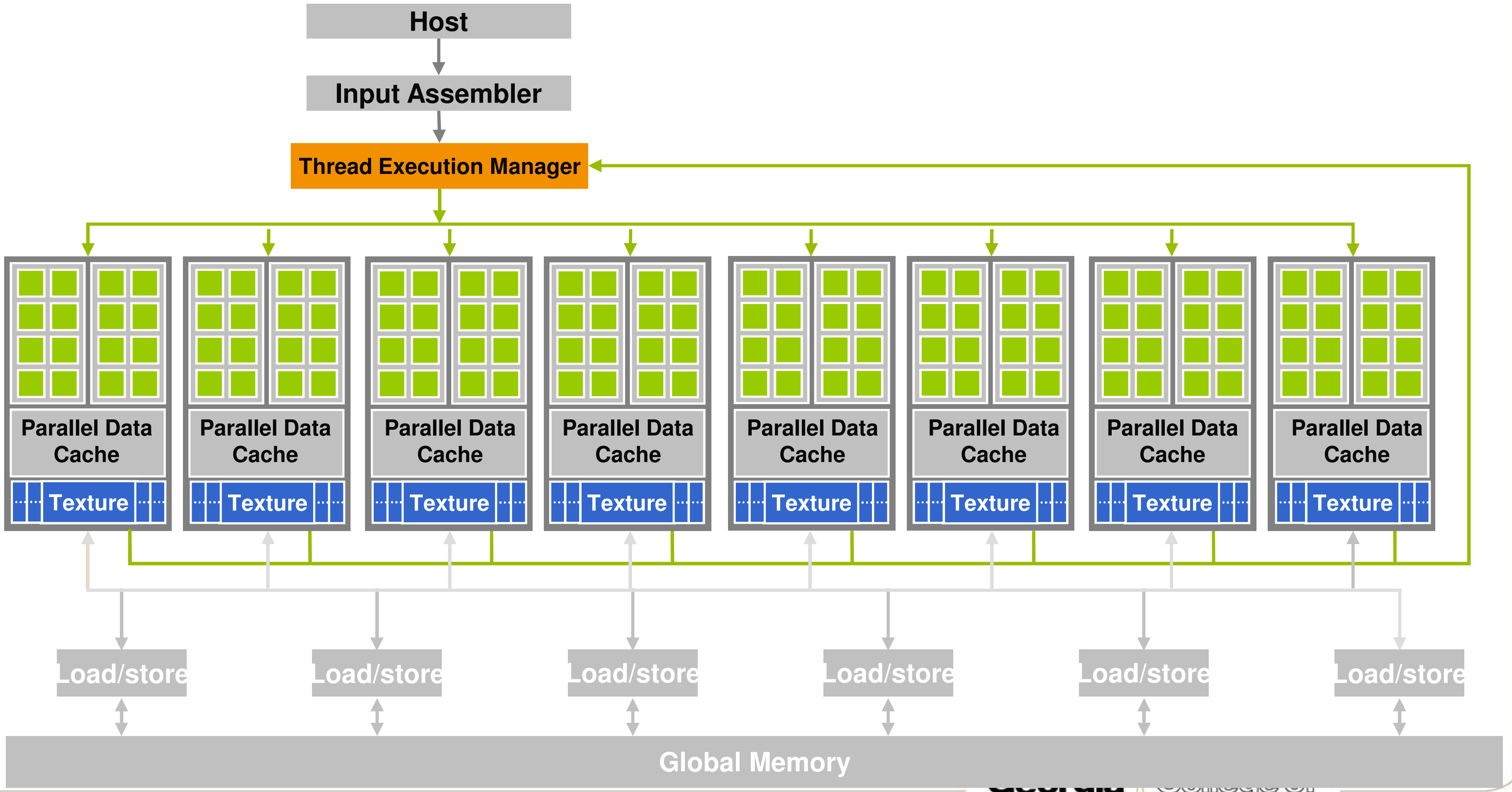
 -__syncthreads()

-Grid of thread blocks

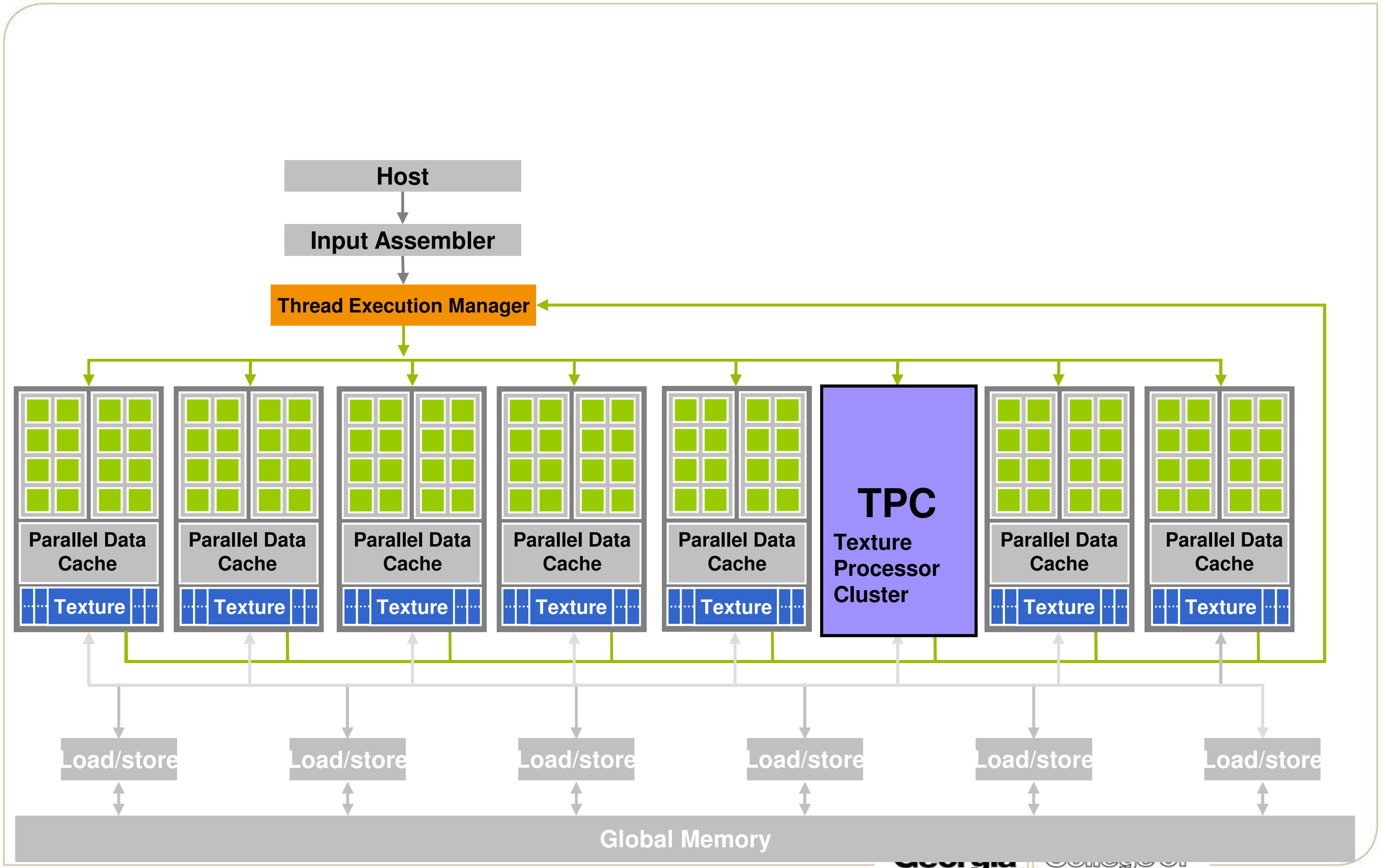
 -F <<< nblocks, nthreads >>> (a, b, c)

GeForce 8800 GTX

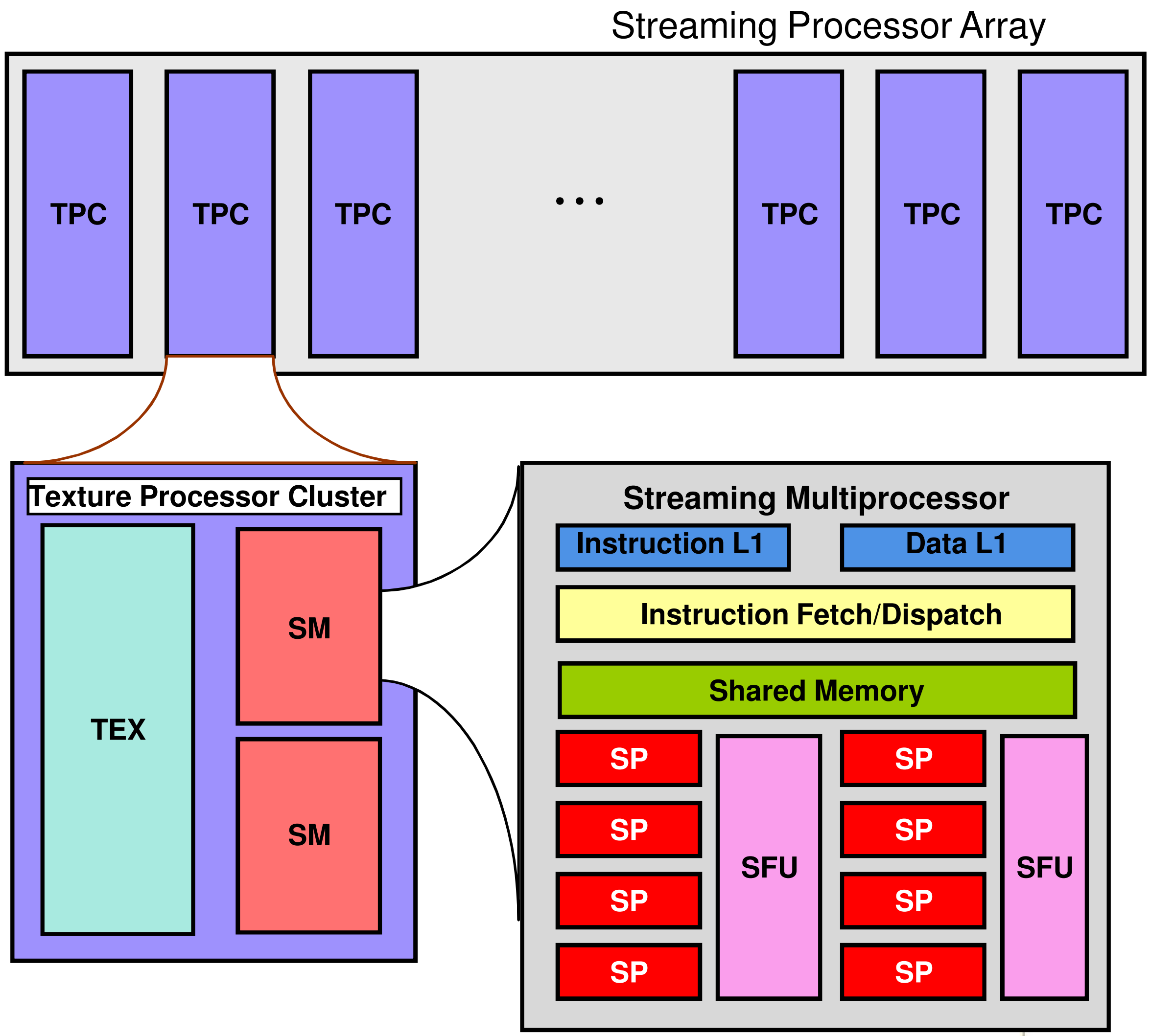
16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU



GeForce 8800 GTX

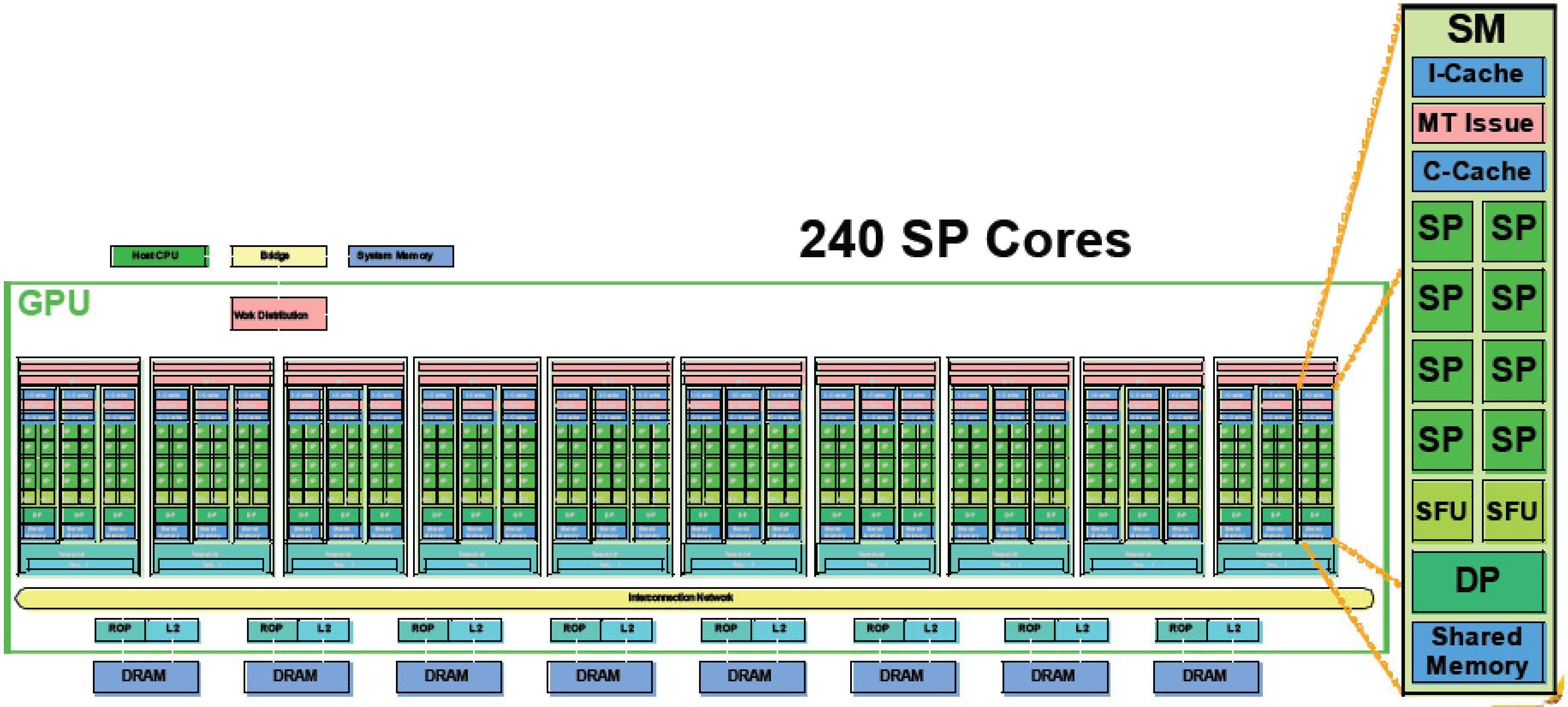


GeForce-8 Series HW Overview



T10P Series

240 SP Cores



CUDA Processor Terminology

- SPA: Streaming Processor Array
  - Array of TPCs
    - 8 TPCs in GeForce8800
  - TPC: Texture Processor Cluster
    - Cluster of 2 SMs+ 1 TEX
    - TEX is a texture processing unit
  - SM: Streaming Multiprocessor
    - Array of 8 SPs
    - Multi-threaded processor core
    - Fundamental processing unit for a thread block
  - SP: Streaming Processor
    - Scalar ALU for a single thread
    - With 1K of registers

GeForce 8800 Series Technical Specs

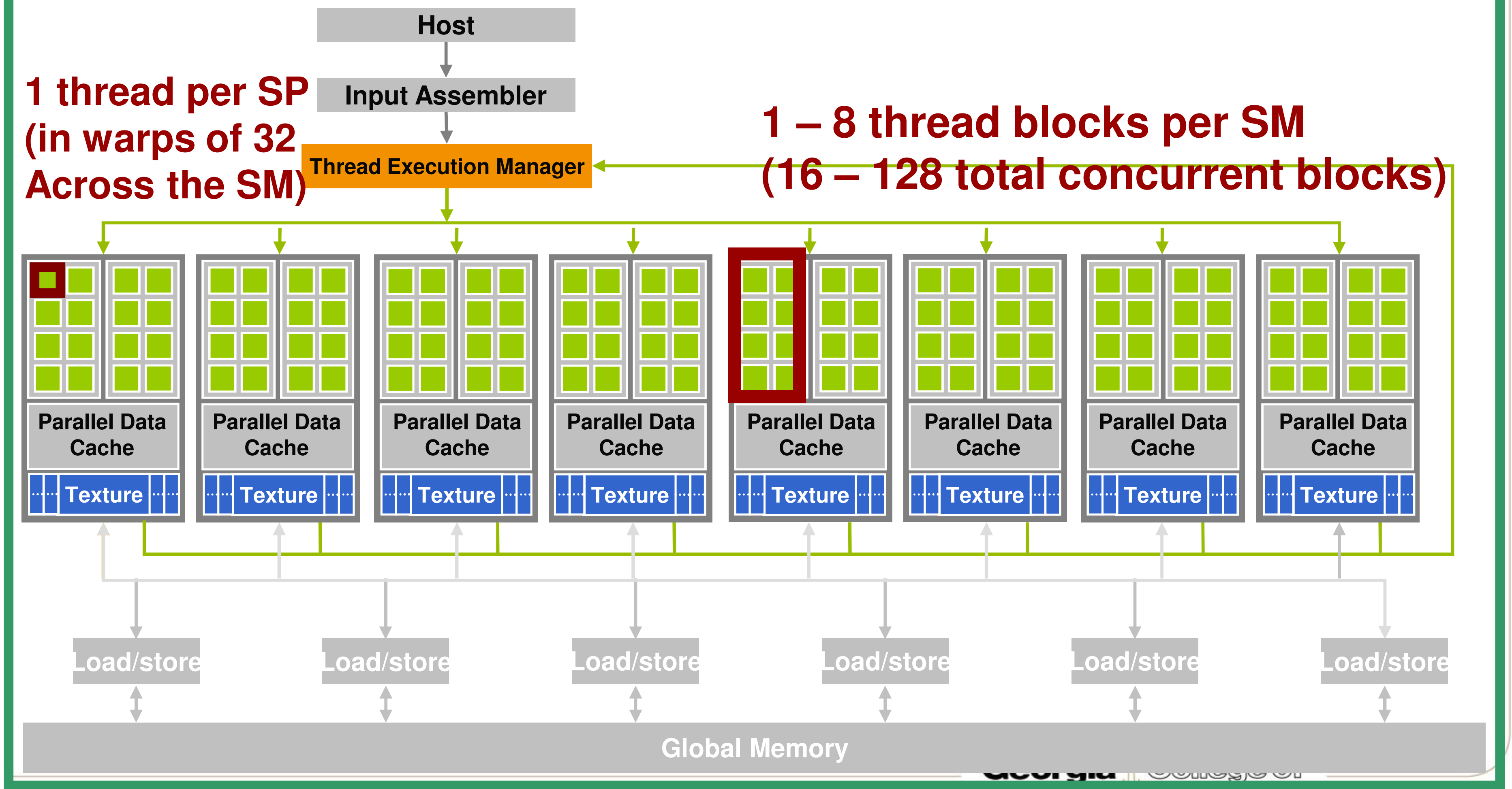
- Maximum number of threads per block: 512
- Maximum size of each dimension of a grid: 65,535
- Number of streaming multiprocessors (SM):
  - GeForce 8800 GTX: 16 @ 675 MHz
  - GeForce 8800 GTS: 12 @ 500 MHz
  - GeForce 8800 GT: 14@ 600MHZ
- Device memory:
  - GeForce 8800 GTX: 768 MB@86.4GB/sec
  - GeForce 8800 GTS: 640 /320MB@64GB/sec
  - GeForce 8800 GT: 512MB@57.5GB/sec
- Shared memory per multiprocessor: 16KB divided in 16 banks
- Constant memory: 64 KB
- Warp size: 32 threads (16 Warps/Block)
- Maximum number of active blocks per multiprocessor: 8
- Maximum number of active threads per multiprocessor: 768 (24 warps)
- Limit on kernel size : 2 M instructions

GeForce 8800

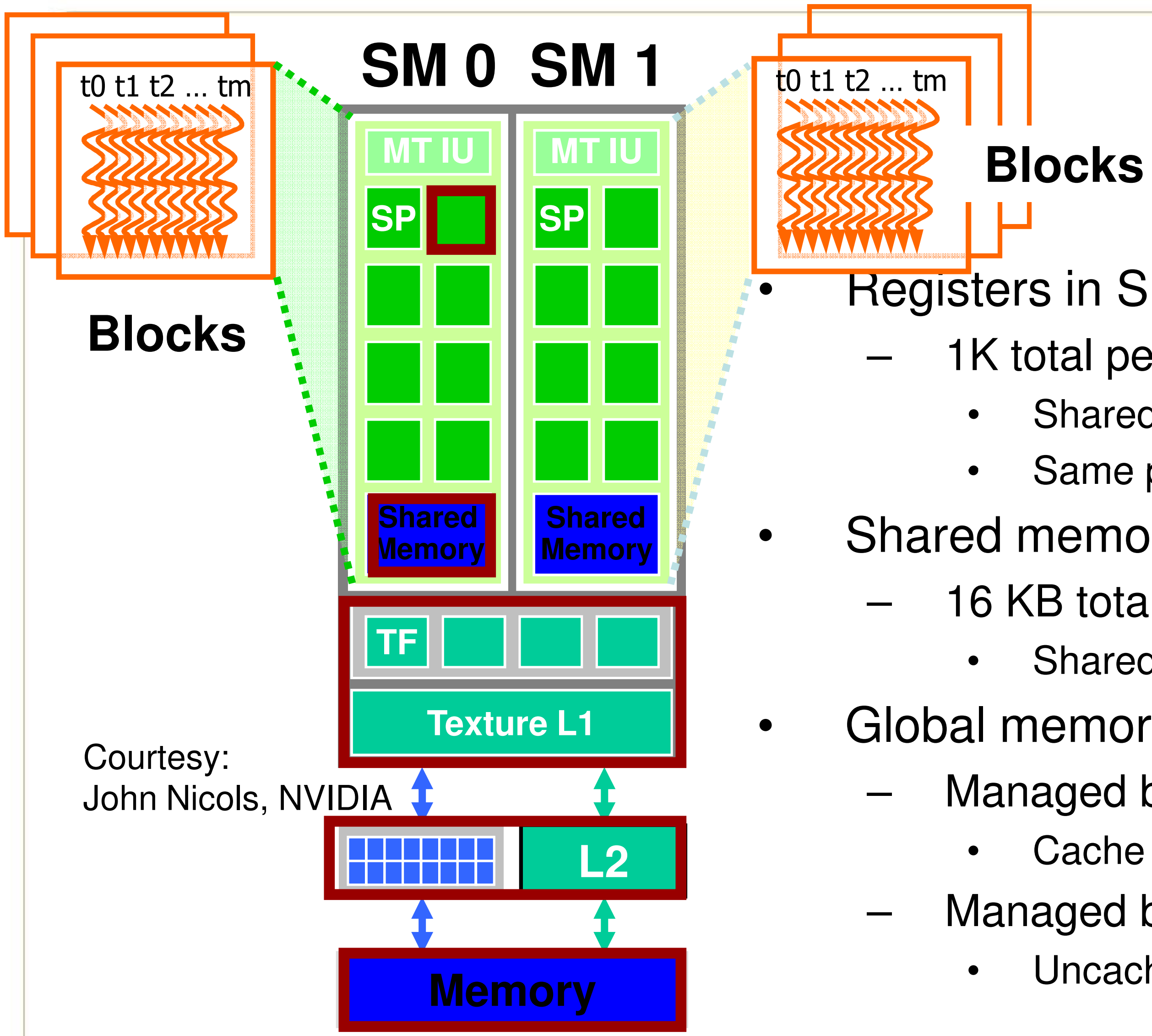
1 Grid (kernel) at a time

1 thread per SP (in warps of 32 Across the SM)

1 – 8 thread blocks per SM (16 – 128 total concurrent blocks)



SM Memory Architecture



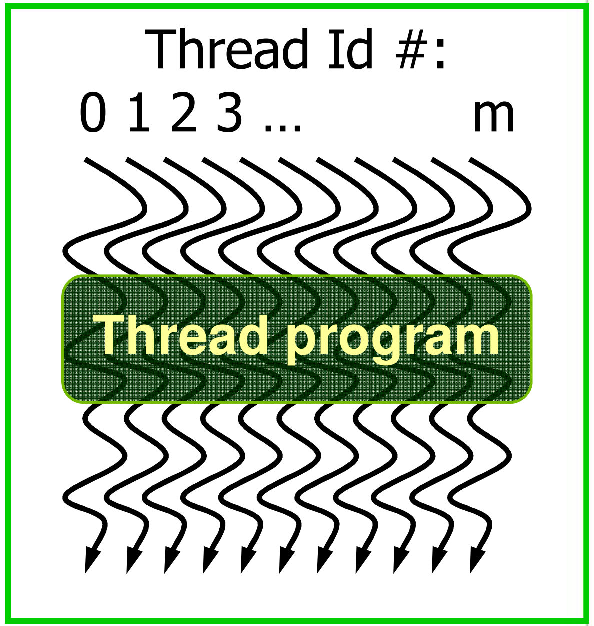
Courtesy:
John Nicols, NVIDIA

- Registers in SP
  - 1K total per SP
    - Shared between thread
    - Same per thread in a block
- Shared memory in SM
  - 16 KB total per SM
    - Shared between blocks
- Global memory
  - Managed by texture units
    - Cache – read only
  - Managed by LD/ST ROP unit
    - Uncached – read/write

CUDA Thread Block: Review

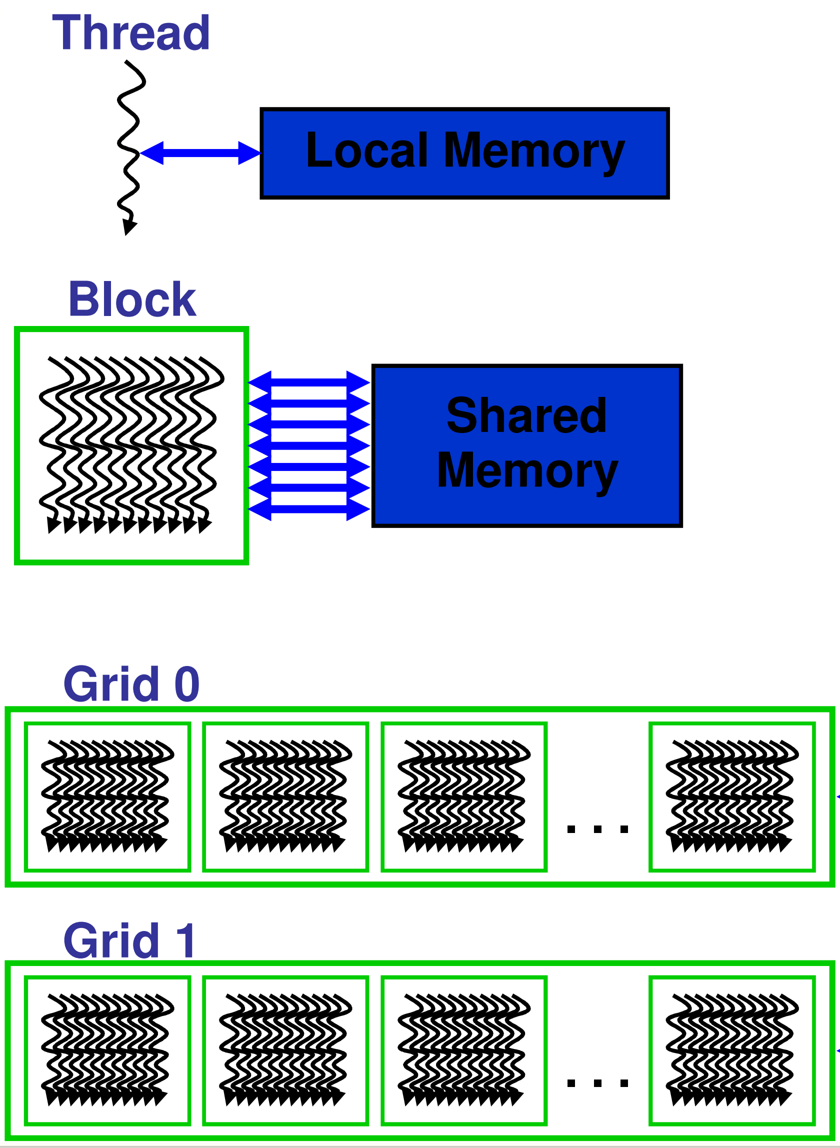
- Programmer declares (Thread) Block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- All threads in a Block execute the same thread program
- Threads have thread id numbers within Block
- Threads share data and synchronize while doing their share of the work
- Thread program uses thread id to select work and address shared data

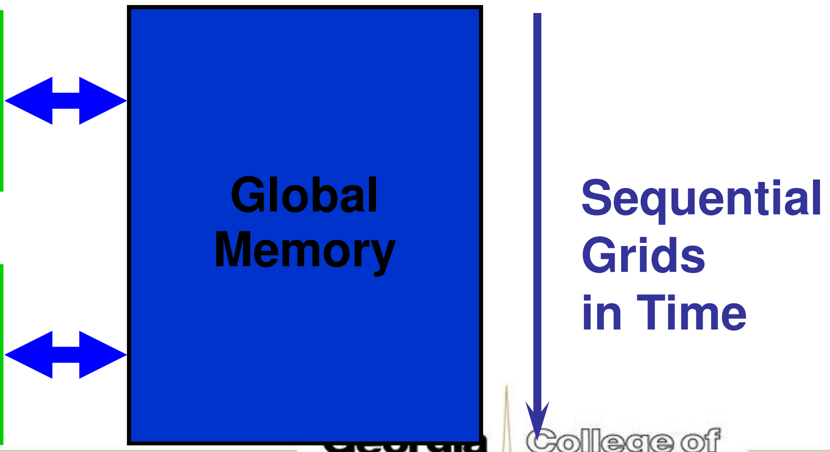
CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

Parallel Memory Sharing



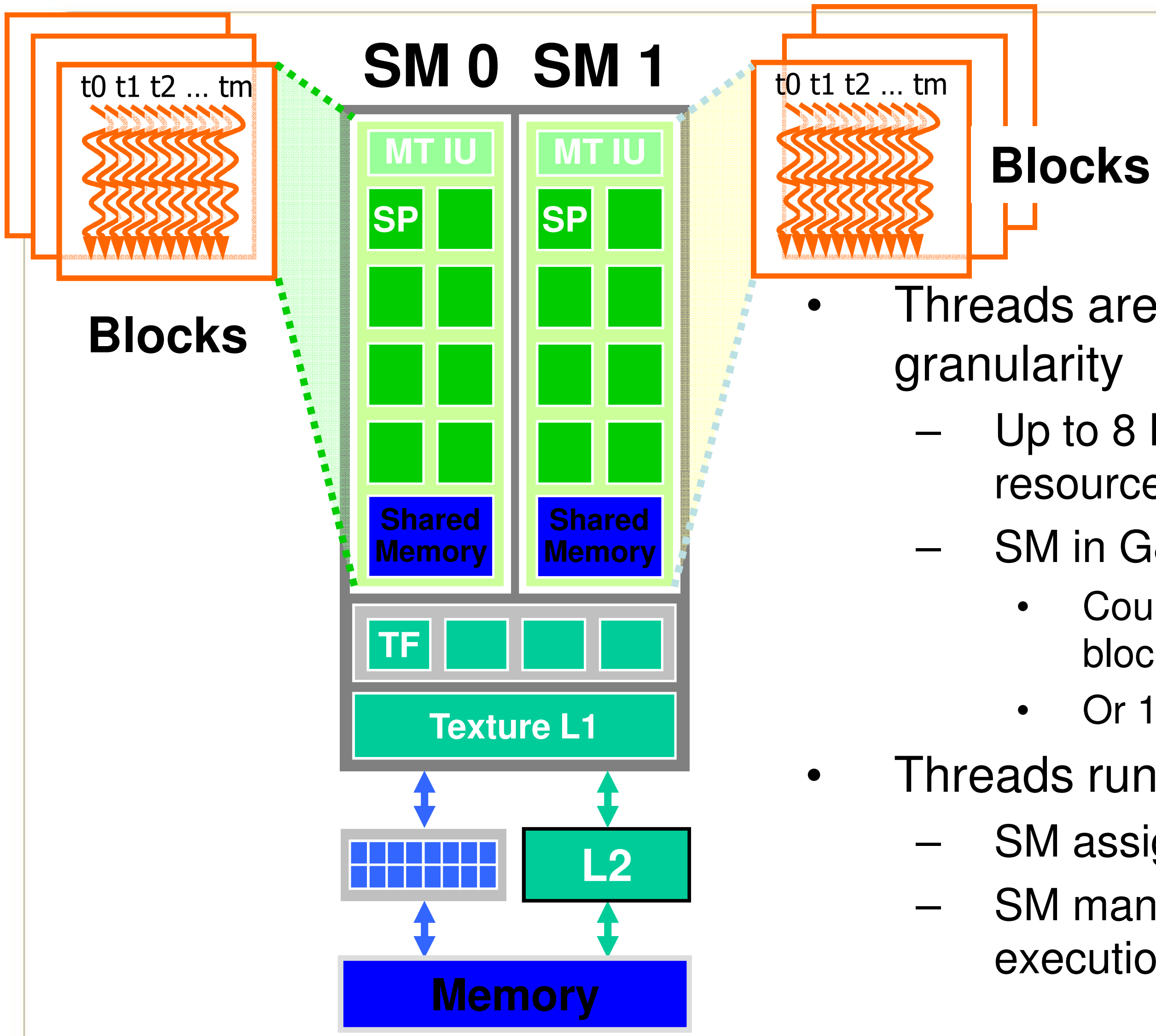


- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication

Bandwidths of GeForce 8800 GTX

- Frequency
  - 575 MHz with ALUs running at 1.35 GHz
- ALU bandwidth (GFLOPs)
  - (1.35 GHz) X (16 SM) X ((8 SP)X(2 MADD) + (2 SFU)) = ~388 GFLOPs
- Register BW
  - (1.35 GHz) X (16 SM) X (8 SP) X (4 words) = 2.8 TB/s
- Shared Memory BW
  - (575 MHz) X (16 SM) X (16 Banks) X (1 word) = 588 GB/s
- Device memory BW
  - 1.8 GHz GDDR3 with 384 bit bus: 86.4 GB/s
- Host memory BW
  - PCI-express: 1.5GB/s or 3GB/s with page locking

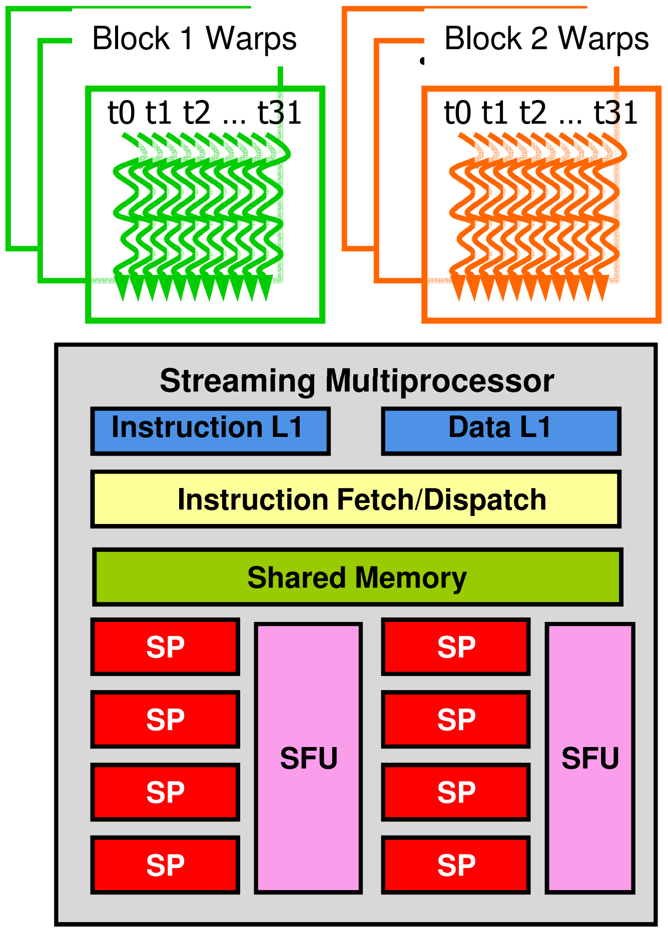
SM Executes Blocks



- 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 id #s
  - SM manages/schedules thread execution

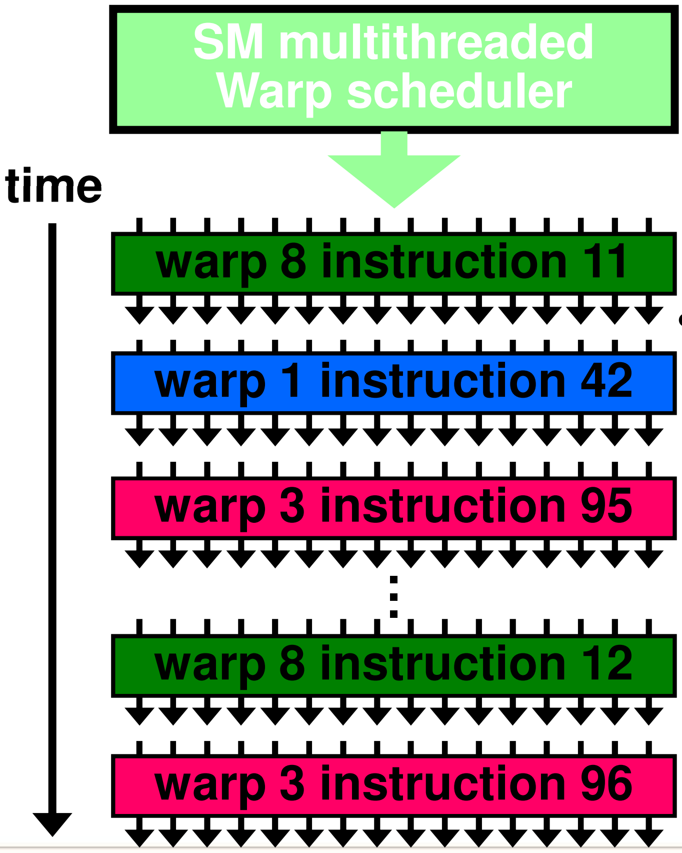
Thread Scheduling/Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- 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.



Georgia Tech College of Computing

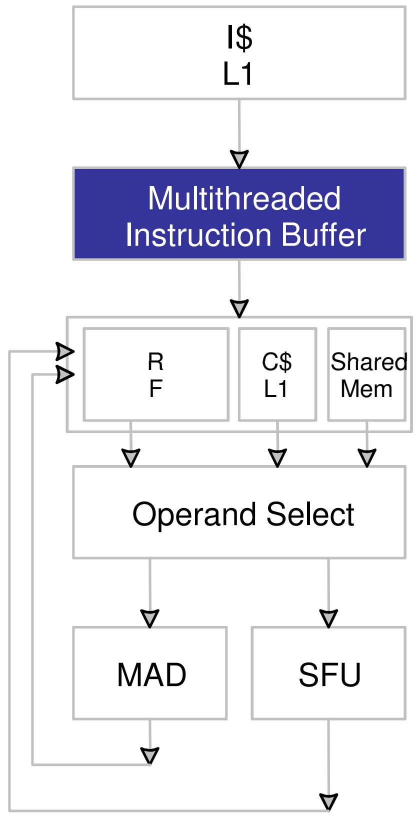
SM Warp Scheduling



- 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
- 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

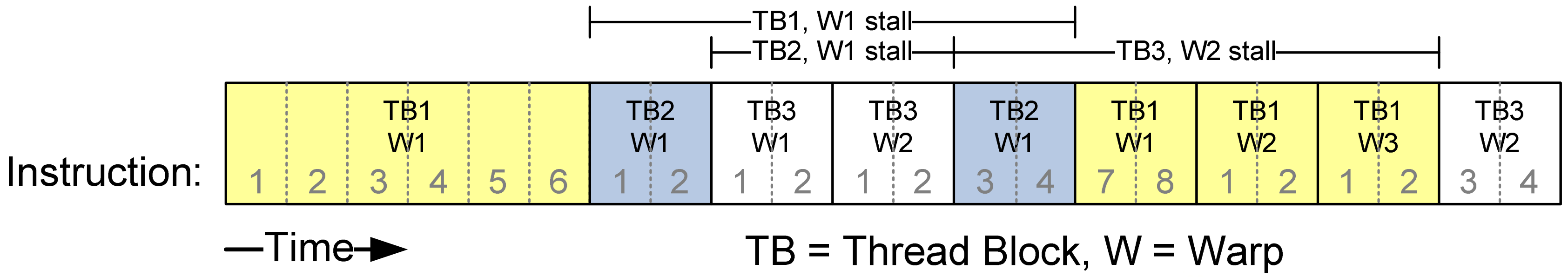
SM Instruction Buffer – Warp Scheduling

- 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



TB = Thread Block, W = Warp

Granularity Considerations

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?
  - 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 can fit into an SM!

Control

- Each SM has its own warp scheduler
- Schedules warps OoO based on hazards and resources
- Warps can be issued in any order within and across blocks
- Within a warp, all threads always have the same position
  - Current implementation has warps of 32 threads
  - Can change with no notice from NVIDIA

Conditionals within a Thread

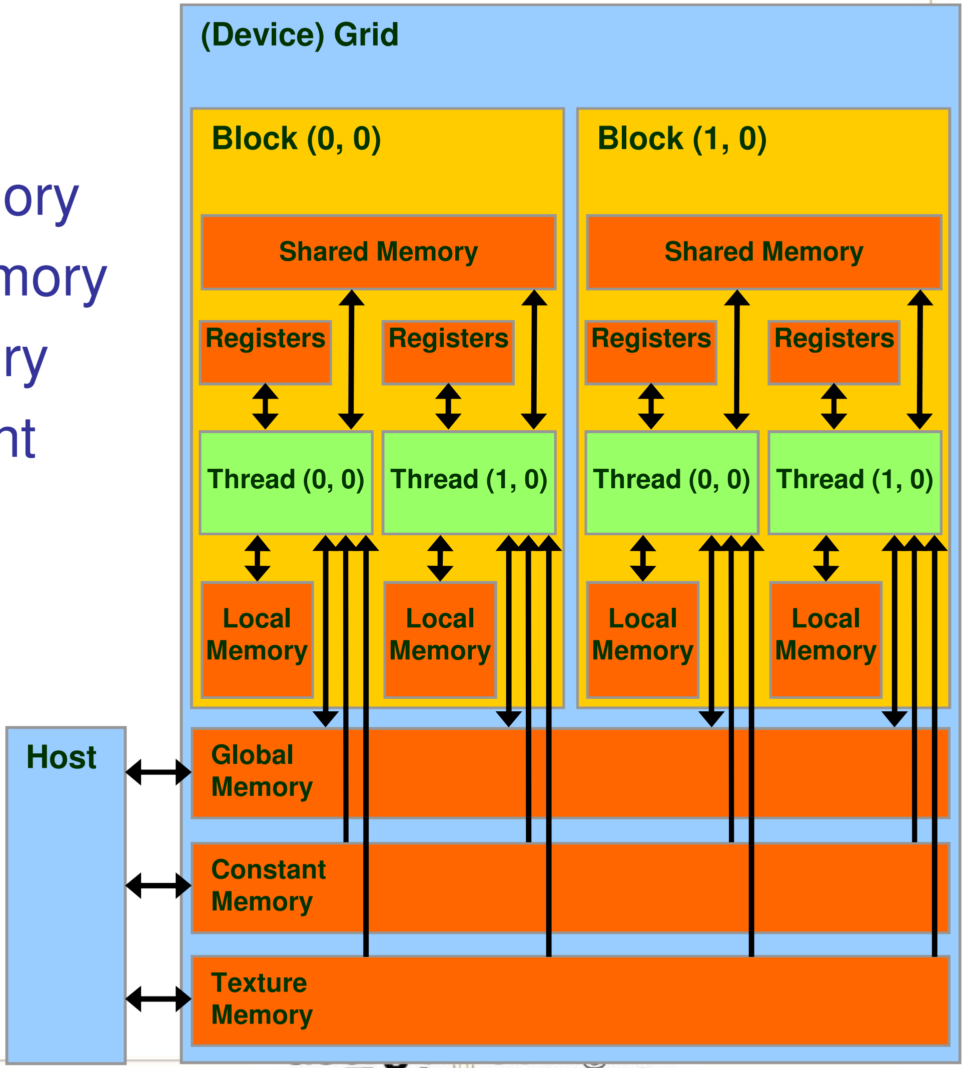
- What happens if there is a conditional statement within a thread?
- No problem if all threads in a warp follow same path
- ***Divergence***: threads in a warp follow different paths
  - HW will ensure correct behavior by (partially) serializing execution
  - Compiler can add predication to eliminate divergence
- Try to avoid divergence
  - If (TID > 2) {…} →If(TID/ warp_size> 2) {…}

Control Flow

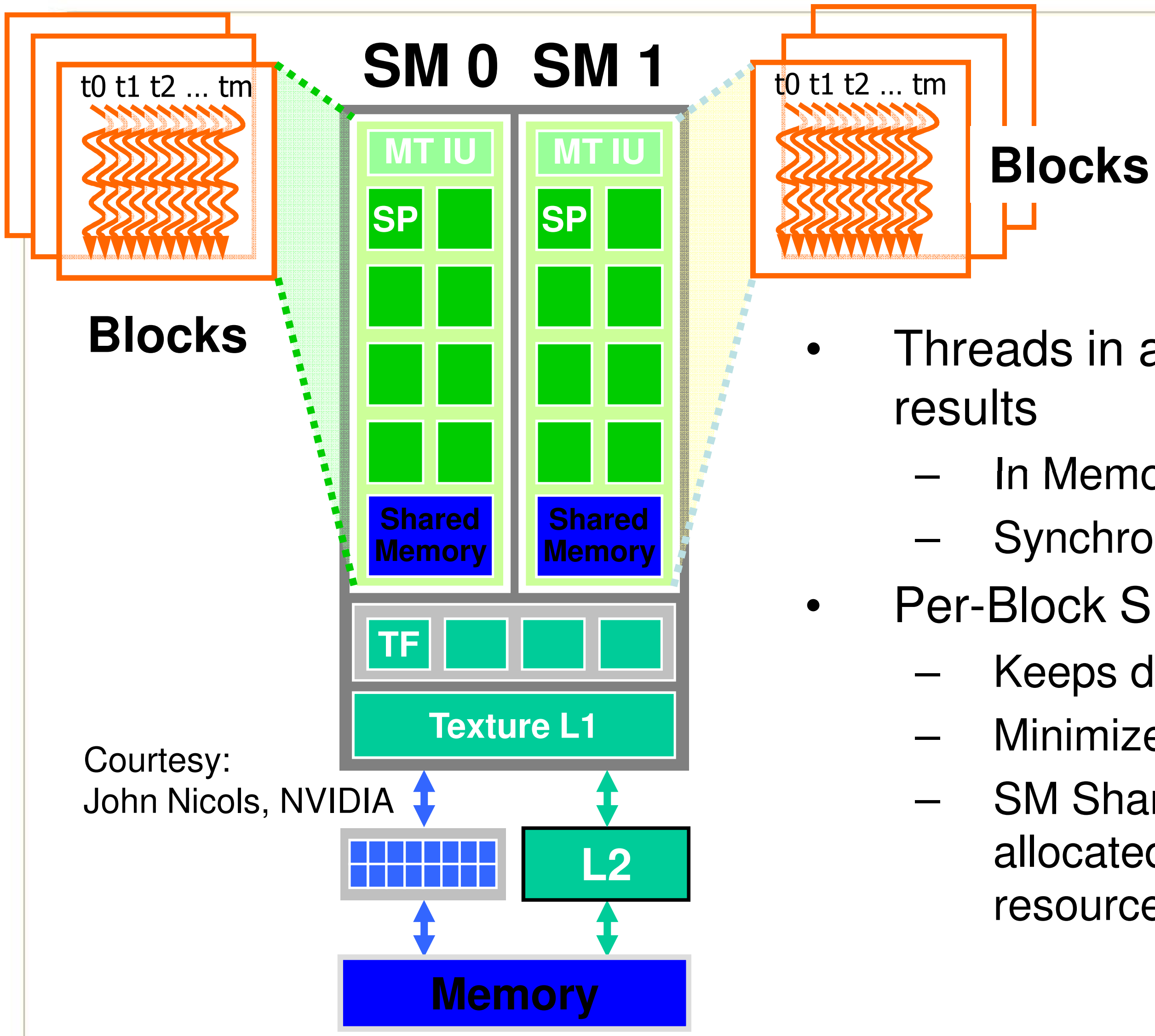
- Recap:
  - 32 threads in a warm are executed in SIMD (share one instruction sequencer)
  - Threads within a warp can be disabled (masked)
    - For example, handling bank conflicts
  - Threads contain arbitrary code including conditional branches
- How do we handle different conditions in different threads?
  - No problem if the threads are in different warps
  - Control ***divergence***
  - ***Predication***

CUDA Device Memory Space: Review

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



SM Memory Architecture

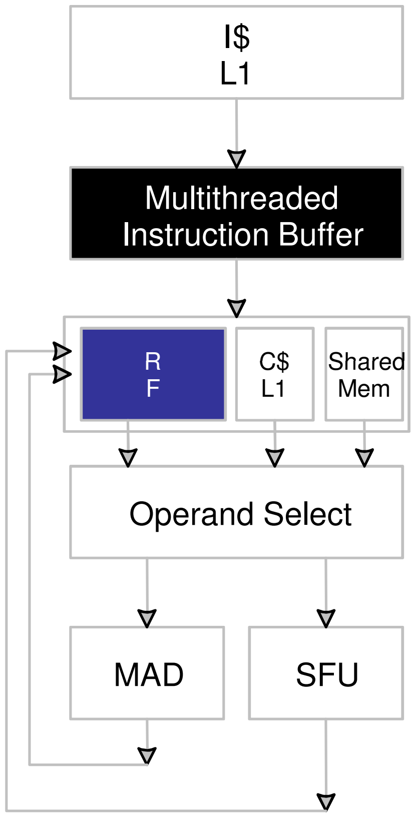


Courtesy:
John Nicols, NVIDIA

- Threads in a Block share data & results
  - In Memory and Shared Memory
  - Synchronize at barrier instruction
- Per-Block Shared Memory Allocation
  - Keeps data close to processor
  - Minimize trips to global Memory
  - SM Shared Memory dynamically allocated to Blocks, one of the limiting resources

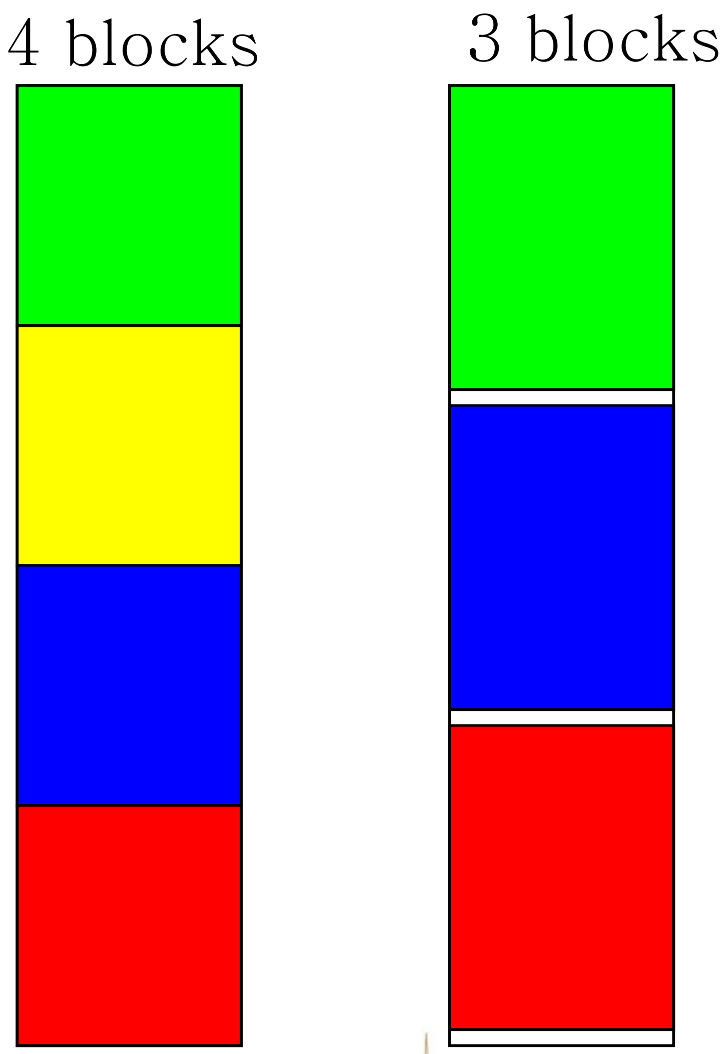
SM Register File

- Register File (RF)
  - 32 KB
  - 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 Block only access registers assigned to itself



Matrix Multiplication Example

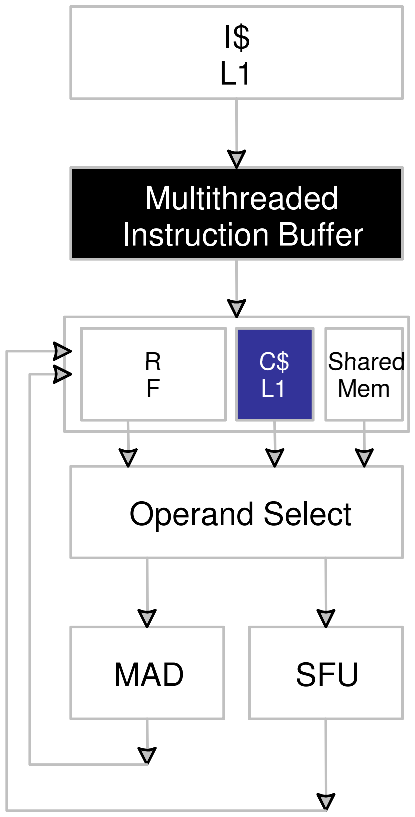
- If each Block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?
  - Each Block requires 10*256 = 2560 registers
  - 8192 = **3** * 2560 + change
  - So, three blocks can run on an SM as far as registers are concerned
- How about if each thread increases the use 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

Constants

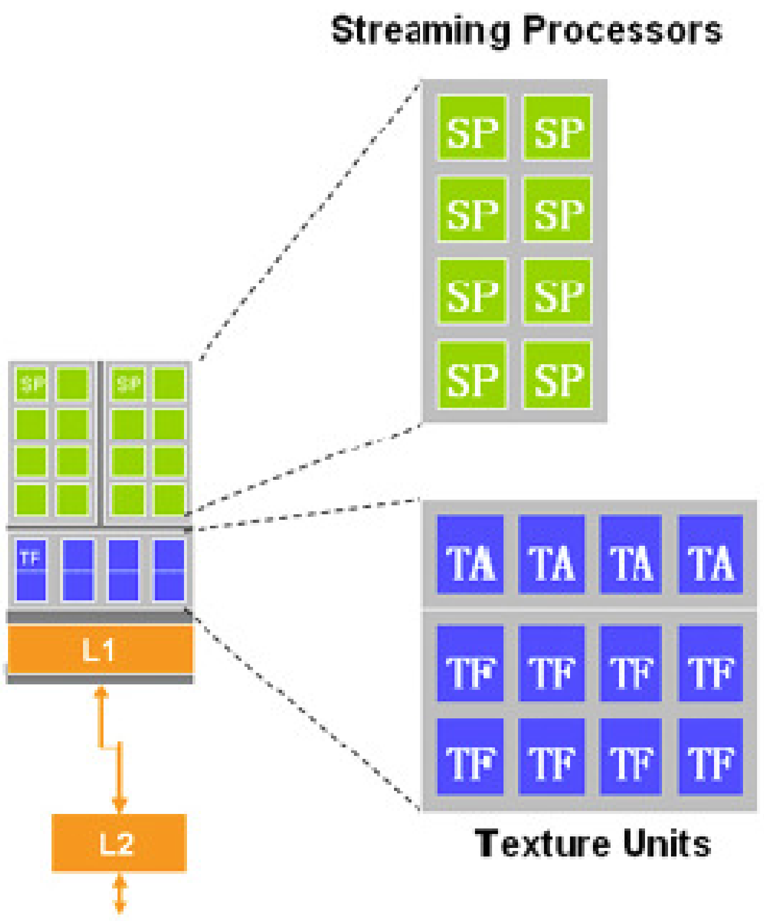
- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!



Textures

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

Streaming Processors, Texture Units, and On-chip Caches



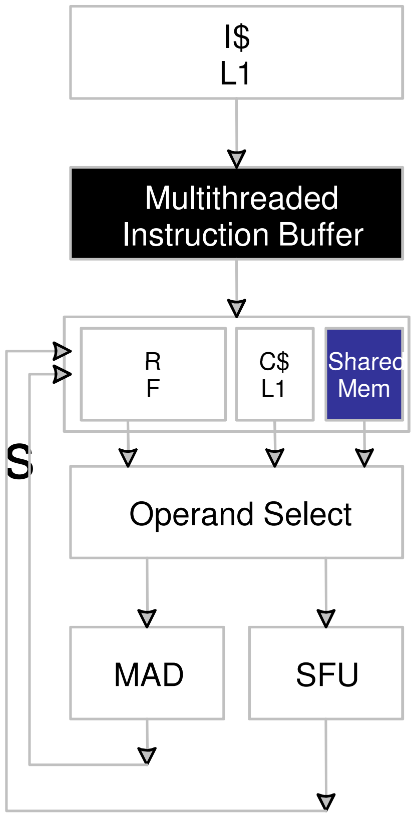
- SP = Streaming Processors
- TF = Texture Filtering Unit
- TA = Texture Address Unit
- L1/L2 = Caches

Exploiting the Texture Samplers

- Designed to map textures onto 3D polygons
- Specialty hardware pipelines for:
  - Fast data sampling from 1D, 2D, 3D arrays
  - Swizzling of 2D, 3D data for optimal access
  - Bilinear filtering in zero cycles
  - Image compositing & blending operations
- Arrays indexed by u,v,w coordinates – easy to program
- Extremely well suited for multigrid & finite difference methods

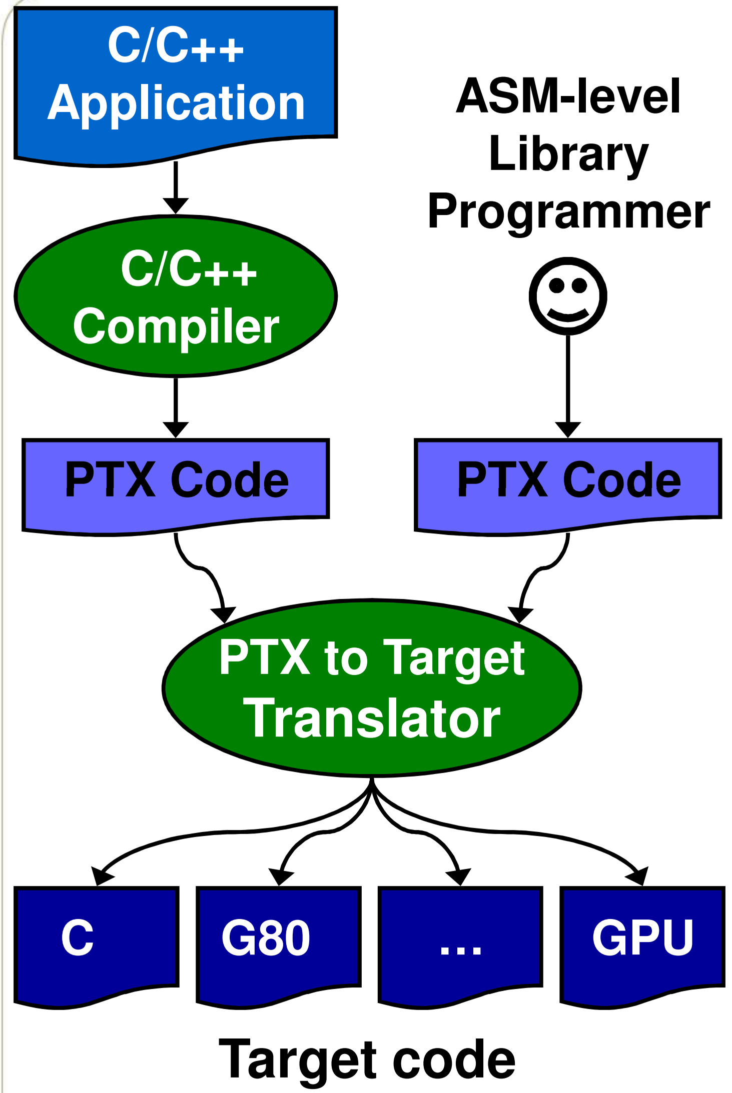
Shared Memory

- 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 ☺



Sample CUDA / PTX Programs

PTX Virtual Machine and ISA



- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state
  - An intermediate language
- ISA – Instruction Set Architecture
  - Variable declarations
  - Instructions and operands
- Translator is an optimizing compiler
  - Translates PTX to Target code
  - Program install time
- Driver implements VM runtime
  - Coupled with Translator

In PTX World

- CTA = (block in CUDA programming domain) : Cooperative Thread Array
- Special registers
  - ctaid: each CTA has a unique CTA id
  - ntcaid: 1D, 2D, 3D?
  - gridid: each grid has a unique temporal grid id
  - %tid, %ntid, %ctaid, %nctaid, and %grid

Compiling CUDA to PTX

CUDA

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```

PTX

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
# 174      me.x += me.y * me.z;
mad.f32              $f1, $f5, $f3, $f1;
```

CUDA Function

- CUDA

```
__device__ void interaction(
                float4 b0, float4 b1, float3 *accel)
{
    r.x = b1.x - b0.x;
    r.y = b1.y - b0.y;
    r.z = b1.z - b0.z;
    float distSqr = r.x * r.x + r.y * r.y + r.z * r.z;
    float s = 1.0f/sqrt(distSqr);
    accel->x += r.x * s;
    accel->y += r.y * s;
    accel->z += r.z * s;
}
```

No register Spills/Fills in PTX level

- PTX

```
sub.f32      $f18, $f1, $f15;
sub.f32      $f19, $f3, $f16;
sub.f32      $f20, $f5, $f17;
mul.f32      $f21, $f18, $f18;
mul.f32      $f22, $f19, $f19;
mul.f32      $f23, $f20, $f20;
add.f32      $f24, $f21, $f22;
add.f32      $f25, $f23, $f24;
rsqrt.f32    $f26, $f25;
mad.f32      $f13, $f18, $f26, $f13;
mov.f32      $f14, $f13;
mad.f32      $f11, $f19, $f26, $f11;
mov.f32      $f12, $f11;
mad.f32      $f9, $f20, $f26, $f9;
mov.f32      $f10, $f9;
```

Compiling a loop that calls a function

- Cuda
  - sx is shared
  - mx, accel are local

```
for (i = 0; i < K; i++) {
   if (i != threadIdx.x) {
       interaction(
            sx[i], mx, &accel
      );
   }
}
```

- PTX

```
     cvt.s32.u16 $r5, %ctaid.x

   mov.s32        $r12, 0;
$Lt_0_26:
   setp.eq.u32        $p1, $r12, $r5;
   @$p1 bra    $Lt_0_27;
   mul.lo.u32  $r13, $r12, 16;
   add.u32     $r14, $r13, $r1;
   ld.shared.f32      $f15, [$r14+0];
   ld.shared.f32      $f16, [$r14+4];
   ld.shared.f32      $f17, [$r14+8];
```

[func body from previous slide inlined here]

```
$Lt_0_27:
   add.s32     $r12, $r12, 1;
   mov.s32     $r15, 128;
   setp.ne.s32        $p2, $r12, $r15;
   @$p2 bra    $Lt_0_26;
```