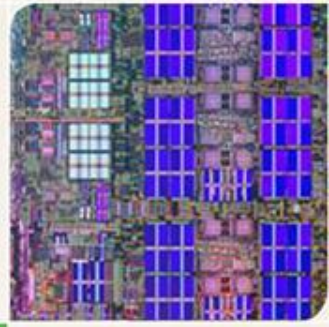
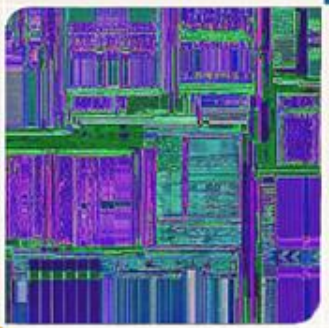
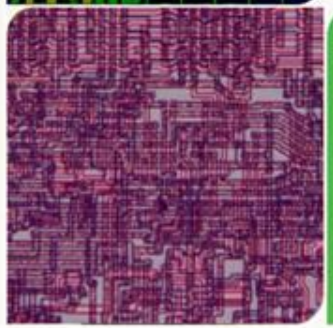




CS4803DGC Design Game Consoles

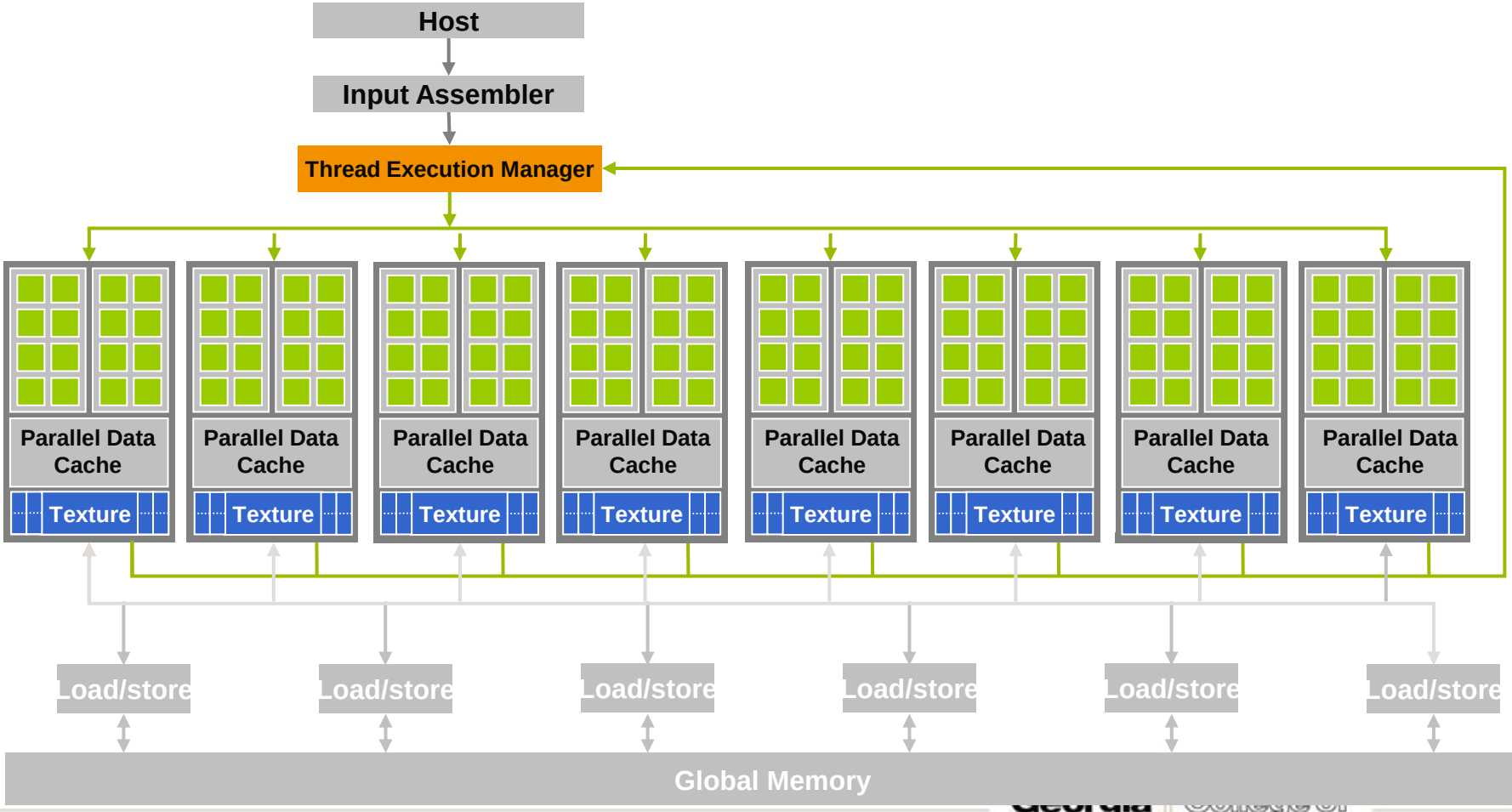
Spring 2010

Prof. Hyesoon Kim

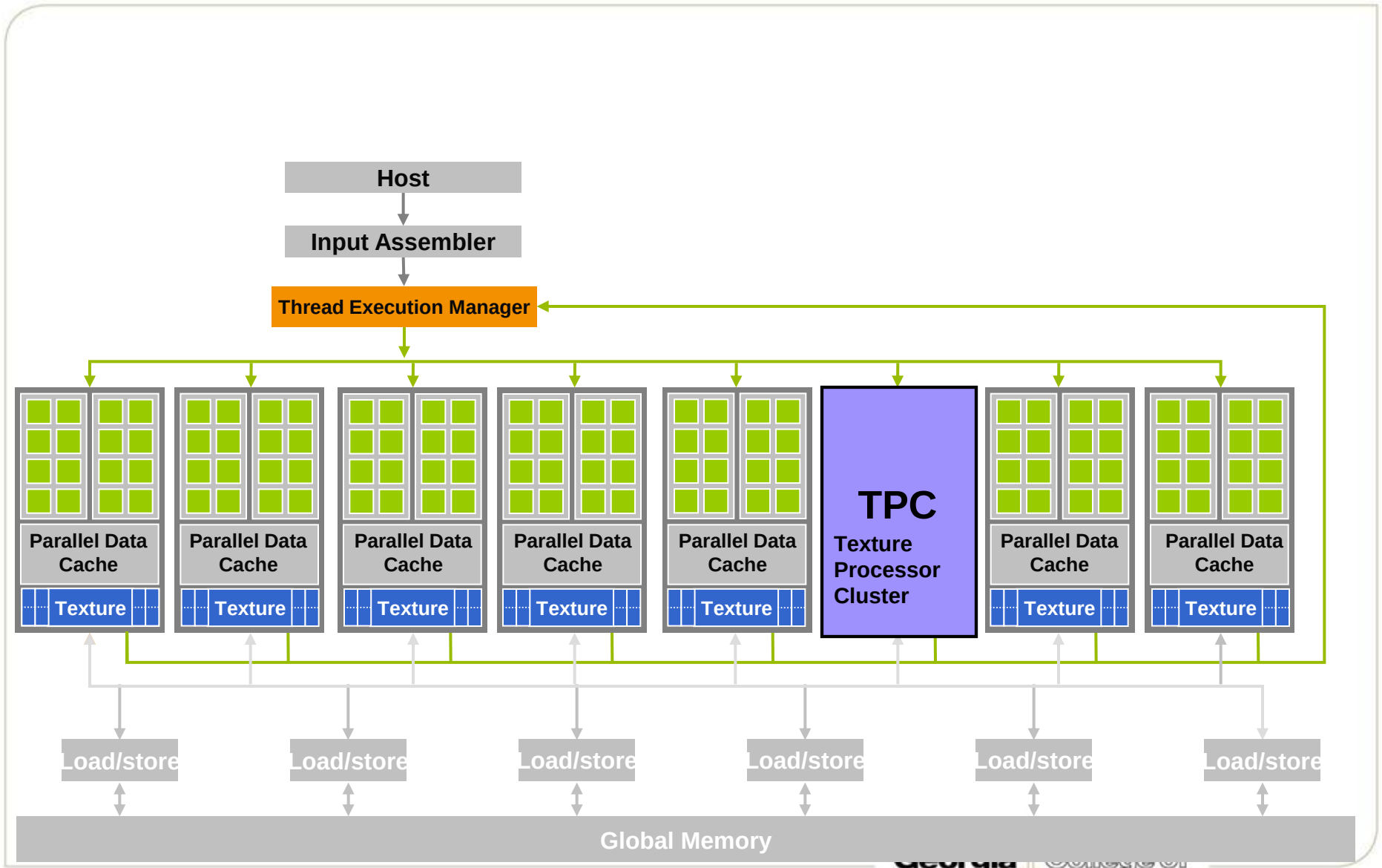
Georgia Tech College of Computing

GeForce 8800 GTX

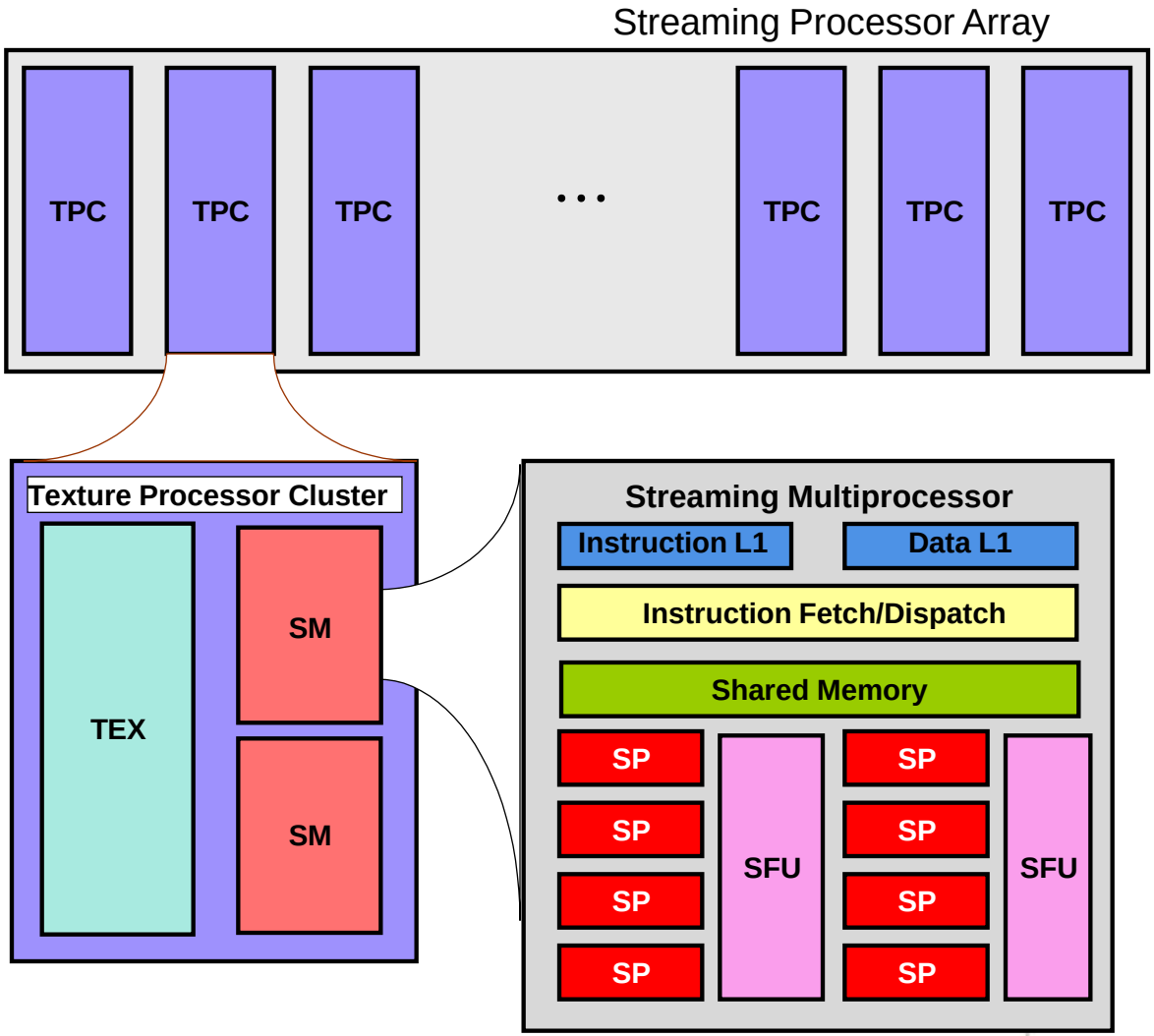
16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU



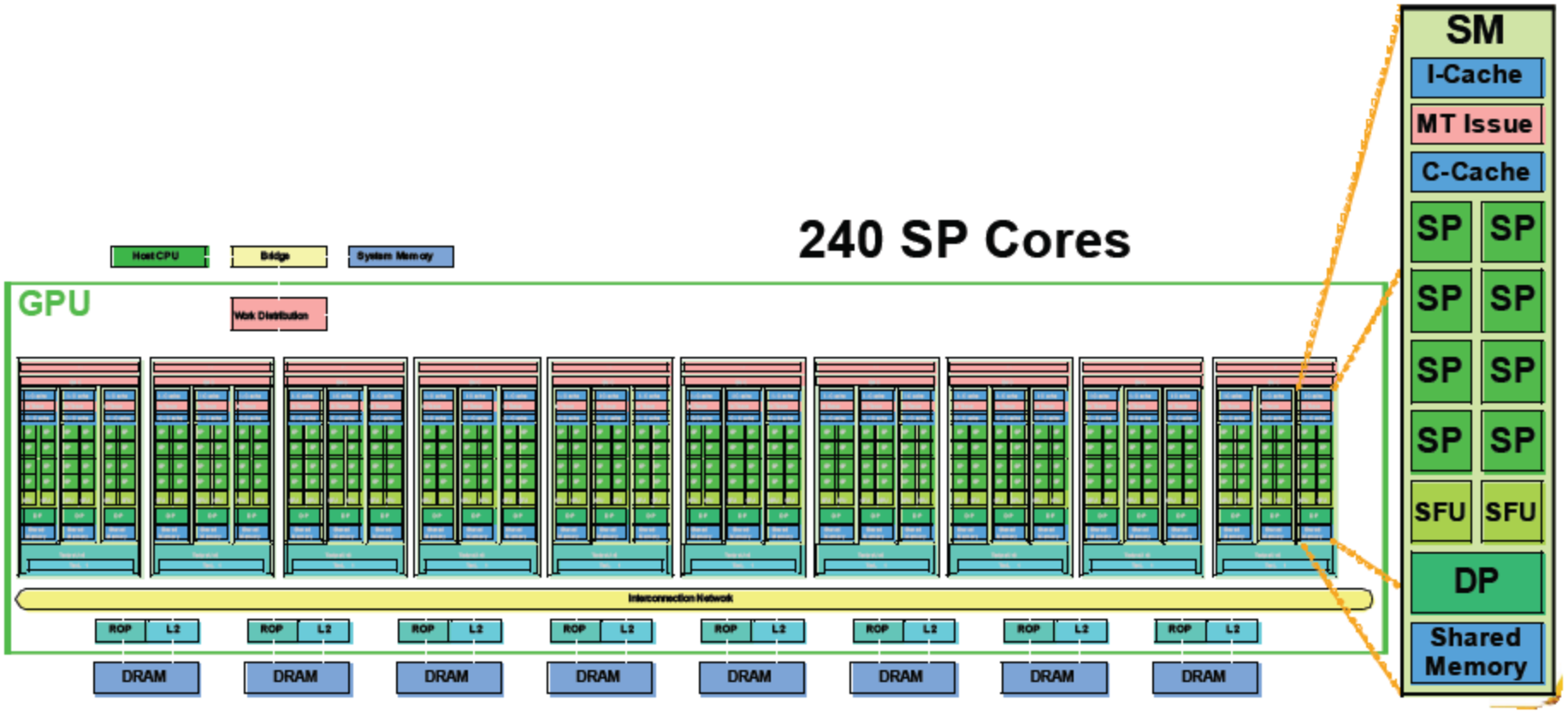
GeForce 8800 GTX



GeForce-8 Series HW Overview



T10P Series

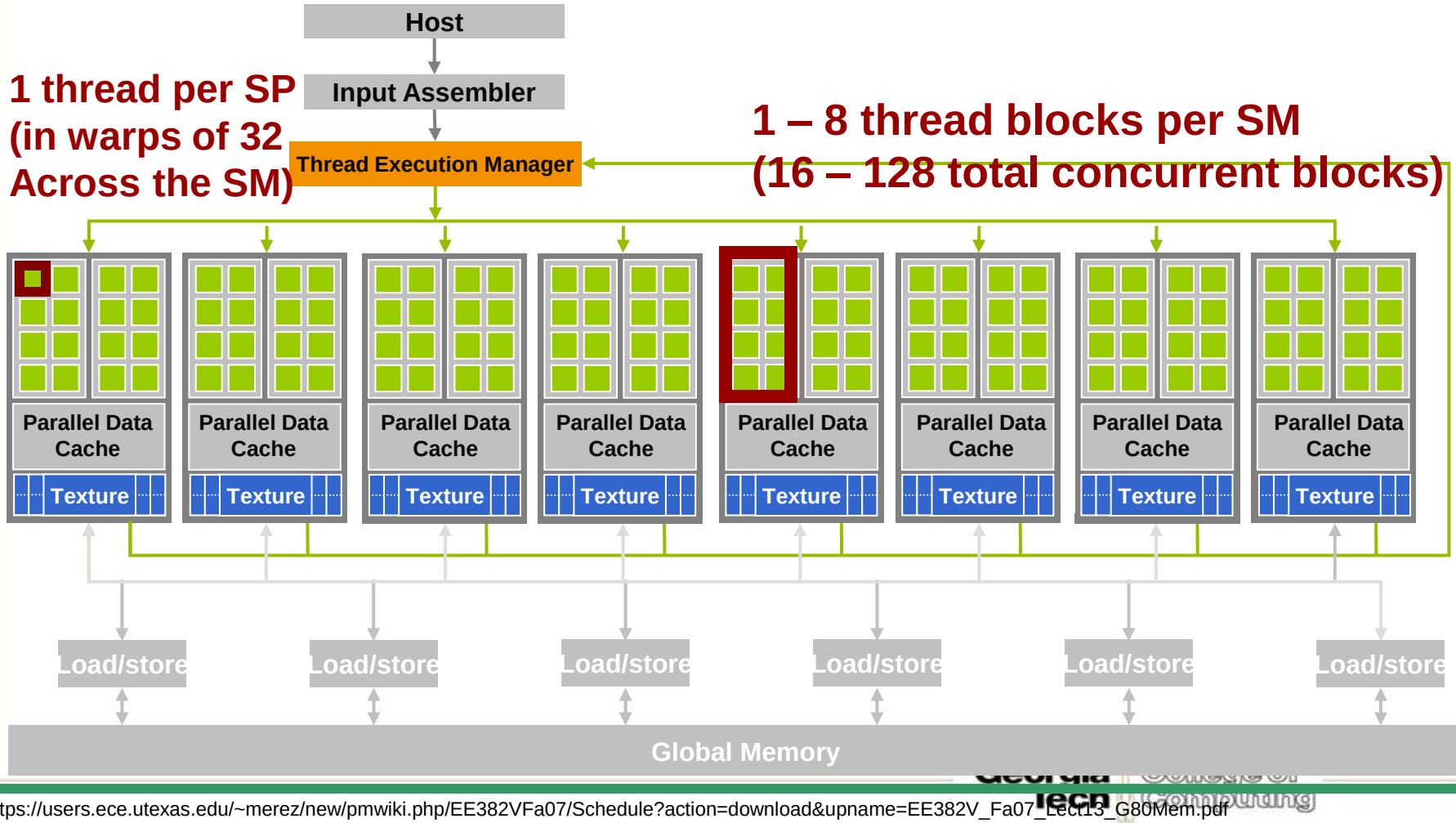

240 SP Cores
Host CPU
Bridge
System Memory
GPU
Work Distribution
Interconnection Network
ROP
L2
DRAM
SM
I-Cache
MT Issue
C-Cache
SP
SFU
DP
Shared Memory

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

1 Grid (kernel) at a time

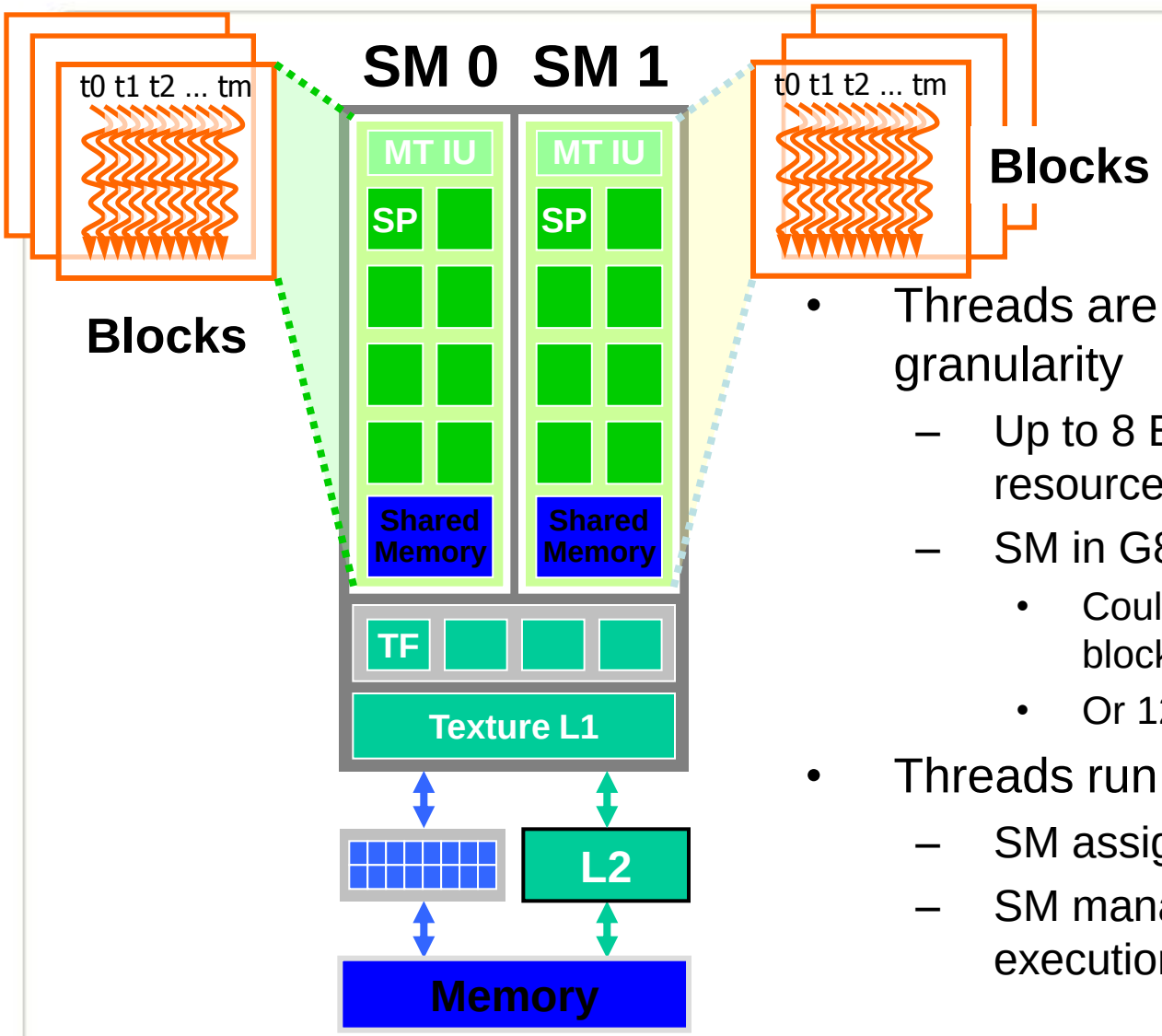


https://users.ece.utexas.edu/~merez/new/pmwiki.php/EE382VFa07/Schedule?action=download&upname=EE382V_Fa07_Lect13_G80Mem.pdf

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

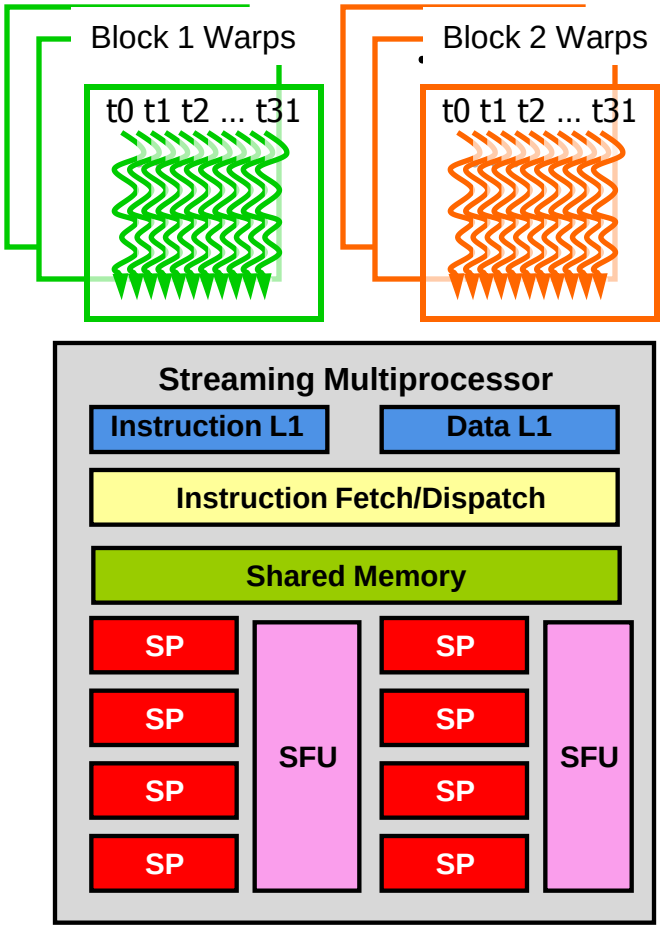
SM Executes Blocks



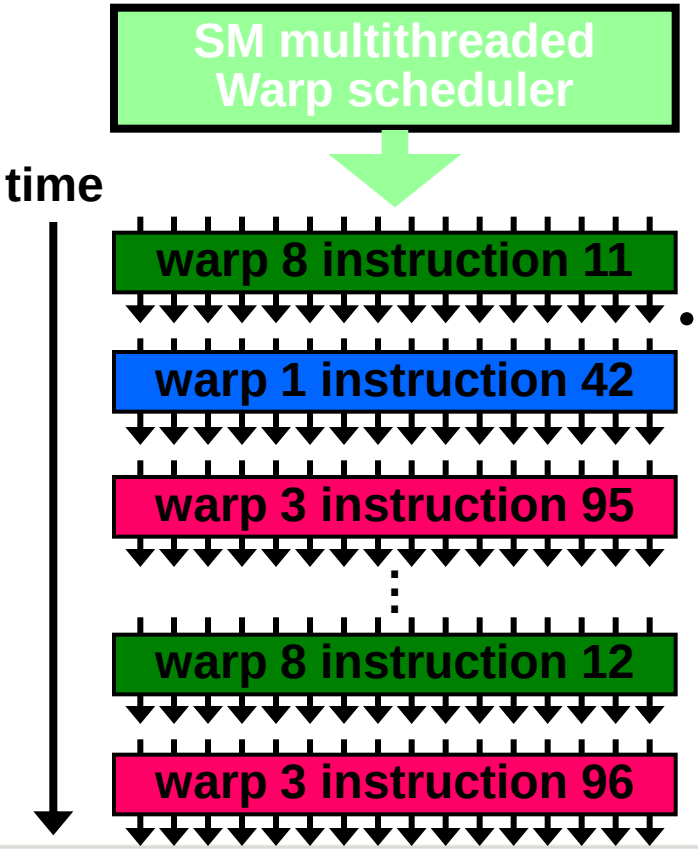
- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks to each SM as resource allows
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM assigns/maintains thread id #s
  -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



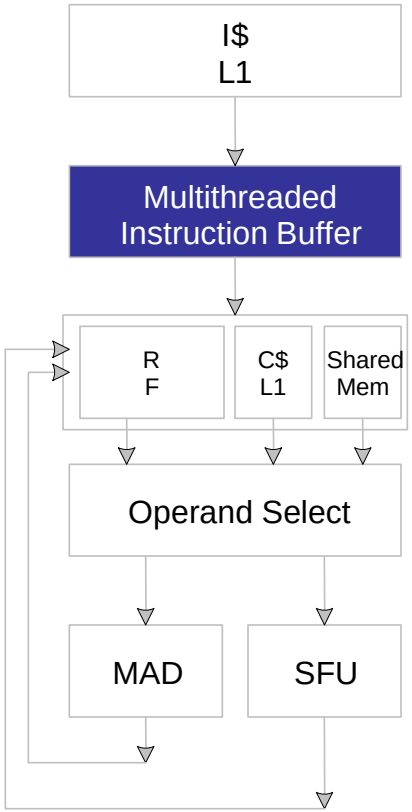
SM Warp Scheduling



- 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

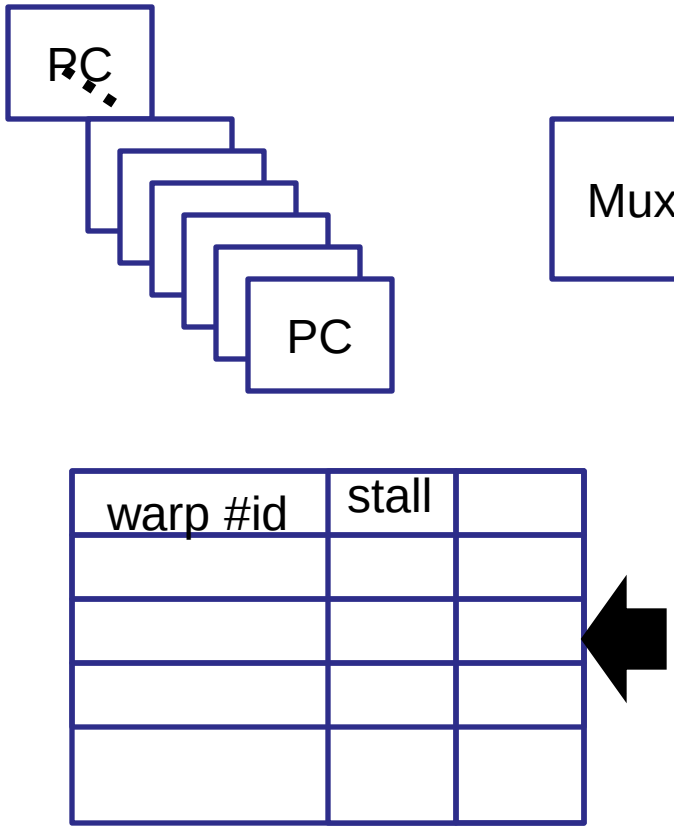




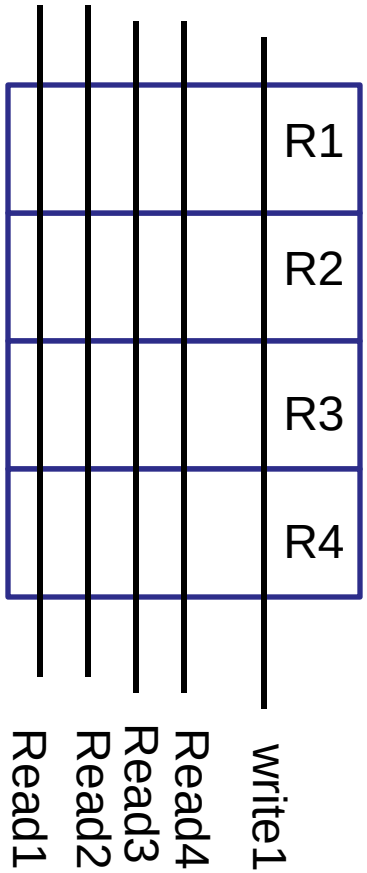
Fetch Polices

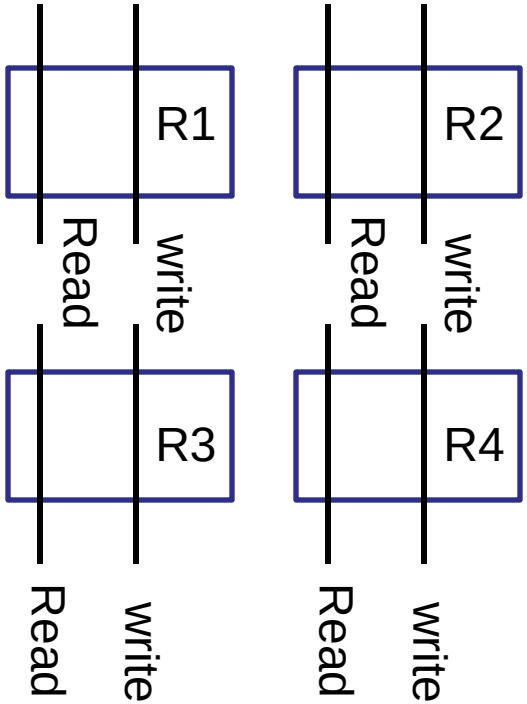
- Strict round robin
- Utilization based policy
- Switch when it fetches a branch

Warp Maintaing Unit



Ports vs. Banks

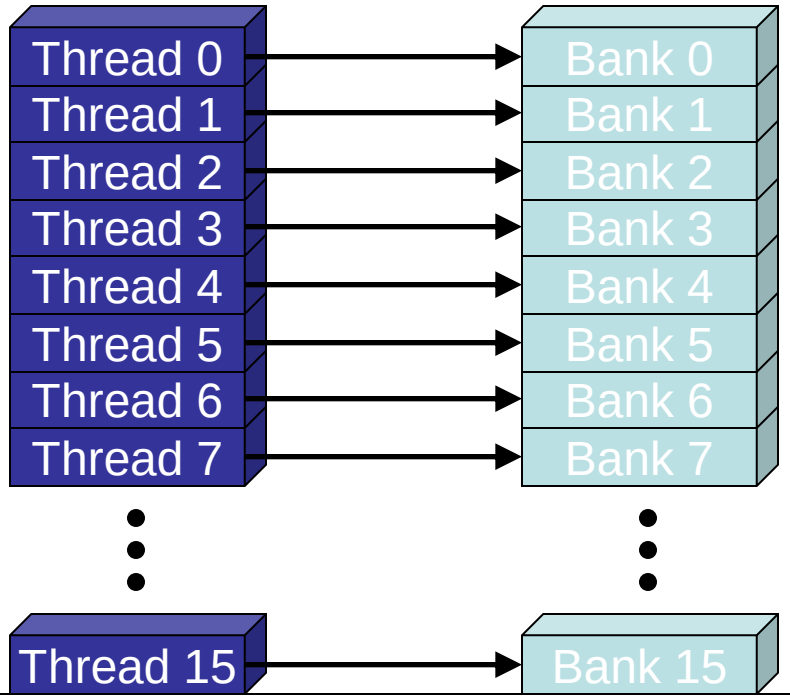




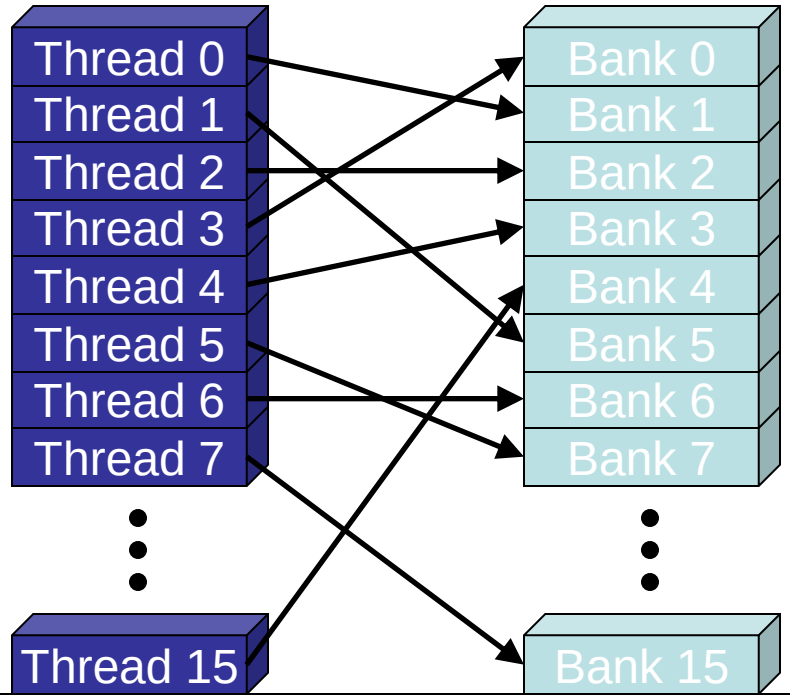
- Multiple read ports
- Banks

Shared Memory: Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1



- No Bank Conflicts
  - Random 1:1 Permutation



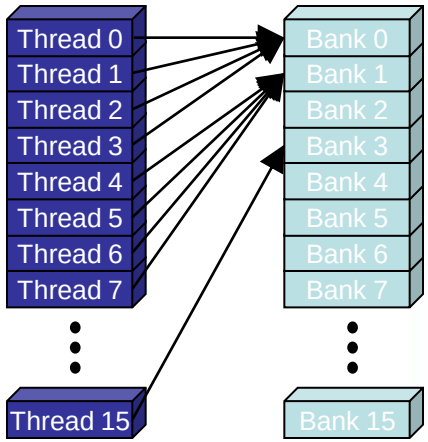
Data types and bank conflicts

- This has no conflicts if type of `shared` is 32-bits:

```
foo = shared[baseIndex + threadIdx.x]
```

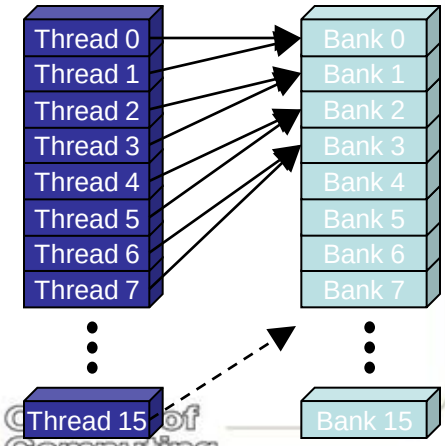
- But not if the data type is smaller
  - 4-way bank conflicts:

```
__shared__ char shared[];
foo = shared[baseIndex + threadIdx.x];
```



  - 2-way bank conflicts:

```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```



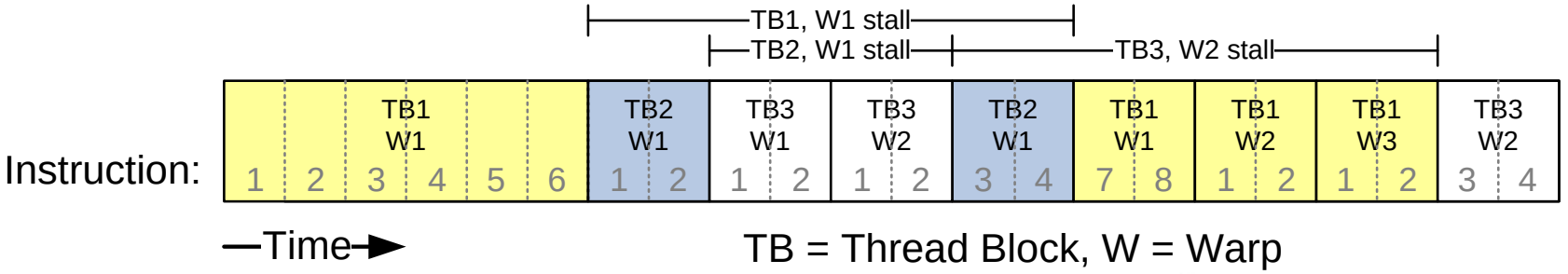


No Branch Prediction. Why?

- Enough parallelism
  - Switch to another thread
  - Speculative execution is
- Branch predictor could be expensive
  - Per thread predictor
- Branch elimination techniques
- Pipeline flush is too costly

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



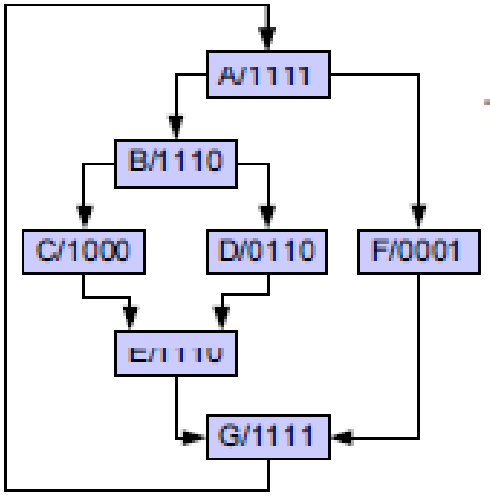
Control

- 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) {…}

Stack Based Divergent Branch Execution



(a) Example Program

	Ret/Reconv. PC	Next PC	Active Mask
	-	G	1111
	G	F	0001
TOS →	G	B	1110

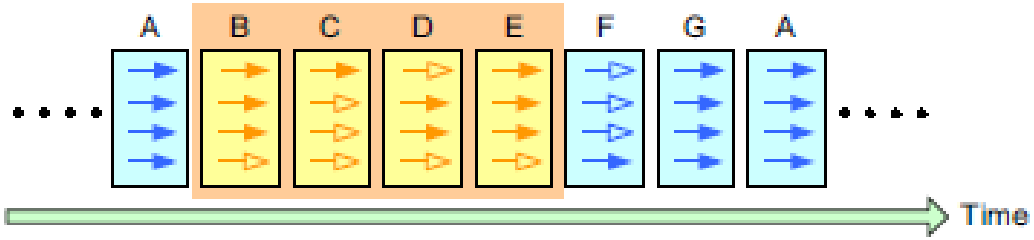
(c) Initial State

	Ret/Reconv. PC	Next PC	Active Mask	
	-	G	1111	
	G	F	0001	
	G	E	1110	(i)
	E	D	0110	(ii)
TOS →	E	C	1000	(iii)

(d) After Divergent Branch

	Ret/Reconv. PC	Next PC	Active Mask
	-	G	1111
	G	F	0001
TOS →	G	E	1110

(e) After Reconvergence



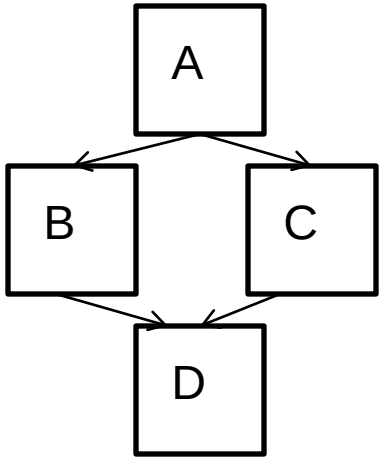
(b) Re-convergence at Immediate Post-Dominator of B

Background: CFG (Control Flow Graph)

- Basic Block
  - Def: a sequence of consecutive operations in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end
  - Single entry, single exit

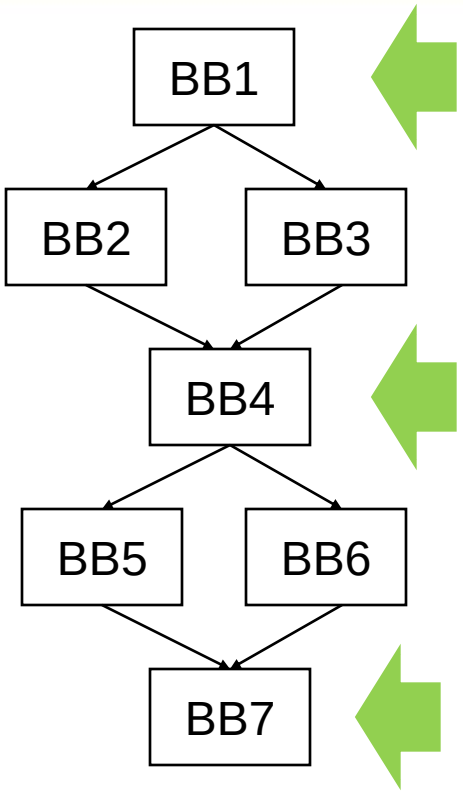
	Add r1, r2, r3 Br.cond target	A
	Mov r3, r4 Br jmp join	B
Target	add r1, r2, r3	C
Join	mov r4 r5	D

Control-flow graph



Dominator/Postdominator

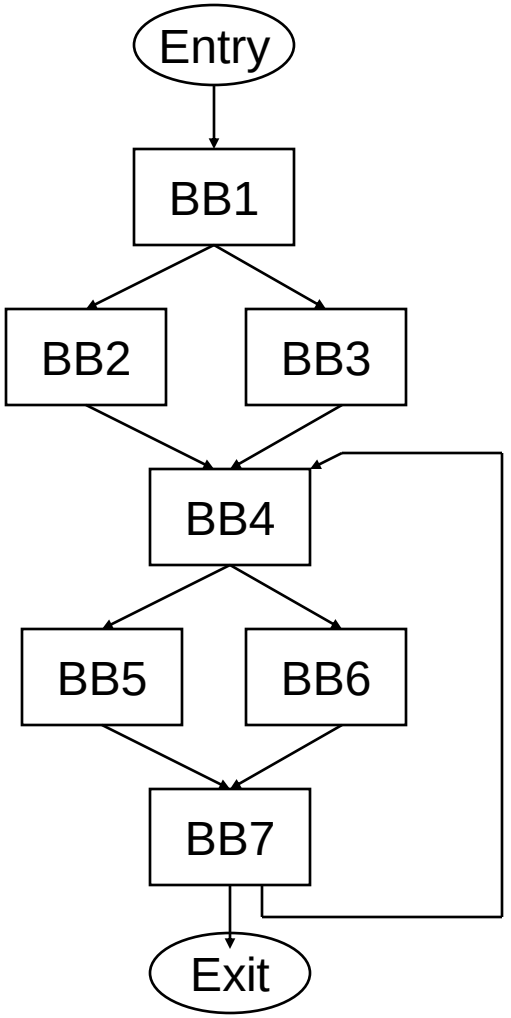
- **Defn: Dominator** – Given a CFG, a node x dominates a node y, if every path from the Entry block to y contains x
  - Given some BB, which blocks are guaranteed to have executed prior to executing the BB
- **Defn: Post dominator**: Given a CFG, a node x post dominates a node y, if every path from y to the Exit contains x
  - Given some BB, which blocks are guaranteed to have executed after executing the BB
  - reverse of dominator



Immediate Post Domiantor

- Defn: Immediate post dominator (ipdom) – Each node n has a unique immediate post dominator m that is the first post dominator of n on any path from n to the Exit
  - Closest node that post dominates
  - First breadth-first successor that post dominates a node

- Immediate post dominator is the reconvergence point of divergent branch



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***

https://users.ece.utexas.edu/~merez/new/pmwiki.php/EE382VFa07/Schedule?action=download&upname=EE382V_Fa07_Lect11_S09Control.pdf



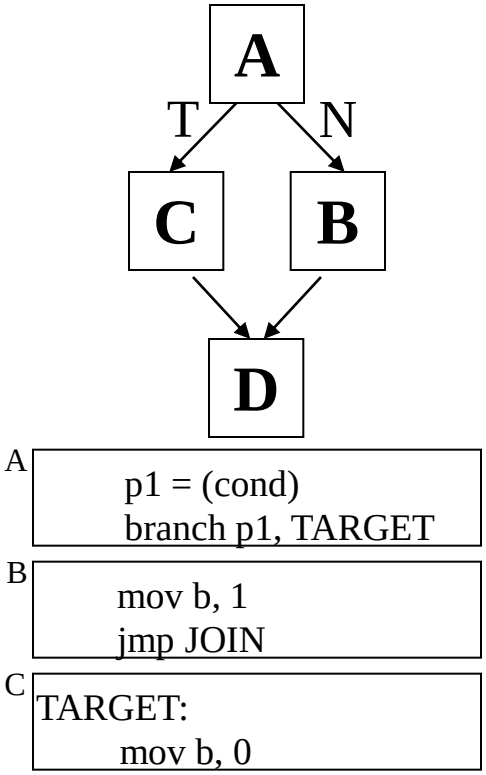
Eliminating Branches

- Predication
- Loop unrolling

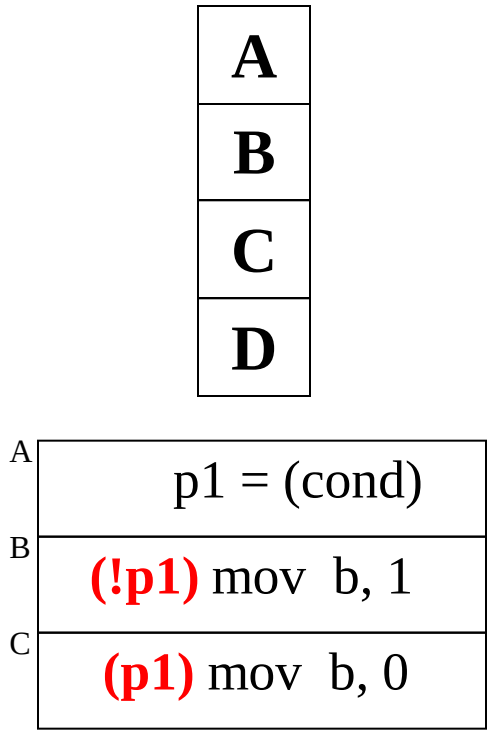
Predication

```
if (cond) {
      b = 0;
}
else {
      b = 1;
}
```

(normal branch code)



(predicated code)



Convert control flow dependency to data dependency

Pro: Eliminate hard-to-predict branches (in traditional architecture)

Eliminate branch divergence (in CUDA)

Cons: Extra instructions

Instruction Predication in G80

- Comparison instructions set condition codes (CC)
- Instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)

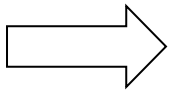
- Compiler tries to predict if a branch condition is likely to produce many divergent warps
  - If guaranteed not to diverge: only predicates if < 4 instructions
  - If not guaranteed: only predicates if < 7 instructions
- May replace branches with instruction predication

- ALL predicated instructions take execution cycles
  - Those with false conditions don't write their output
    - Or invoke memory loads and stores
  - Saves branch instructions, so can be cheaper than serializing divergent paths

Loop Unrolling

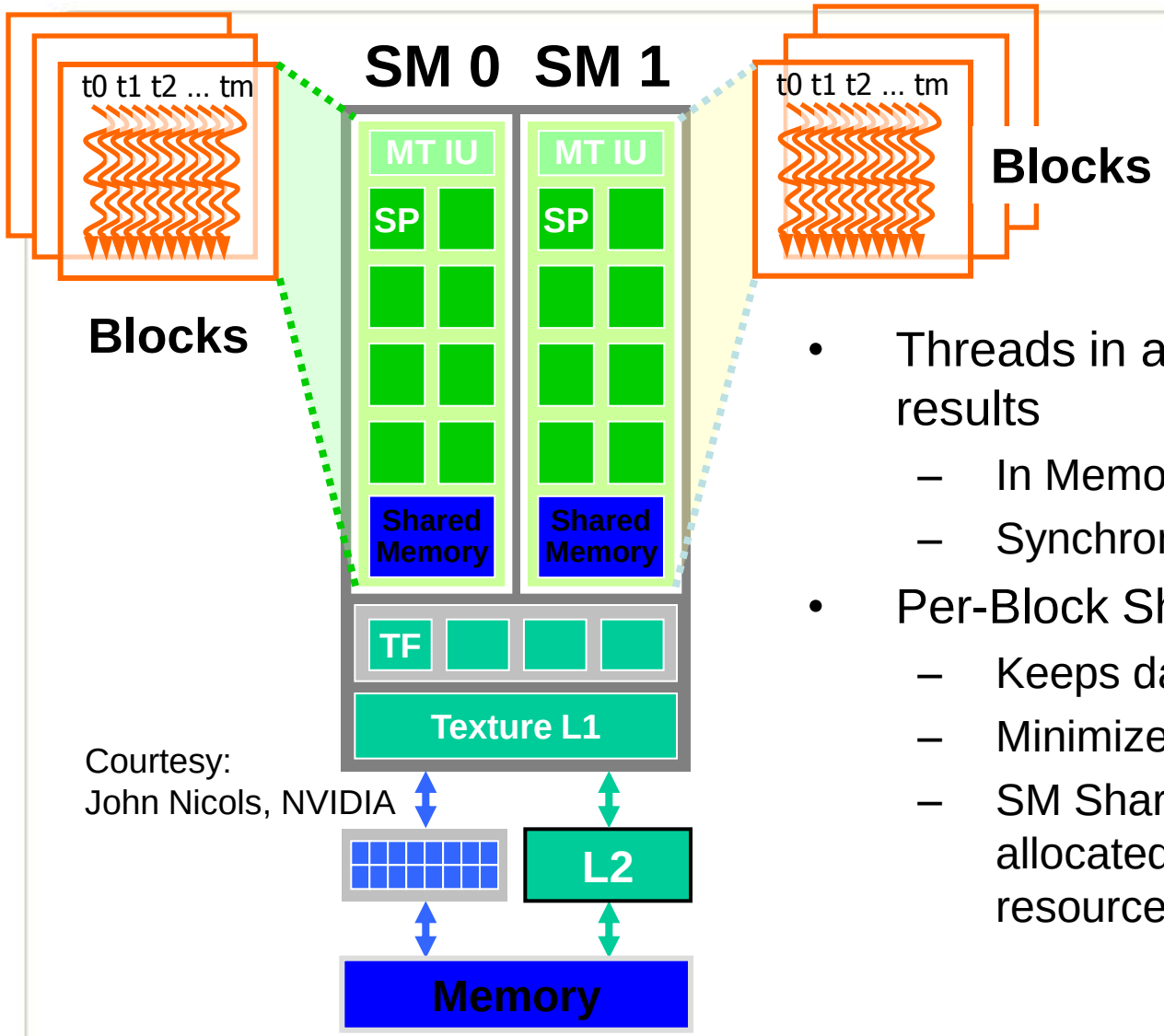
- Transforms an M-iteration loop into a loop with M/N iterations
  - We say that the loop has been unrolled N times

```
for(i=0;i<100;i++)
  a[i]*=2;
```



```
for(i=0;i<100;i+=4){
  a[i]*=2;
  a[i+1]*=2;
  a[i+2]*=2;
  a[i+3]*=2;
}
```

SM Memory Architecture

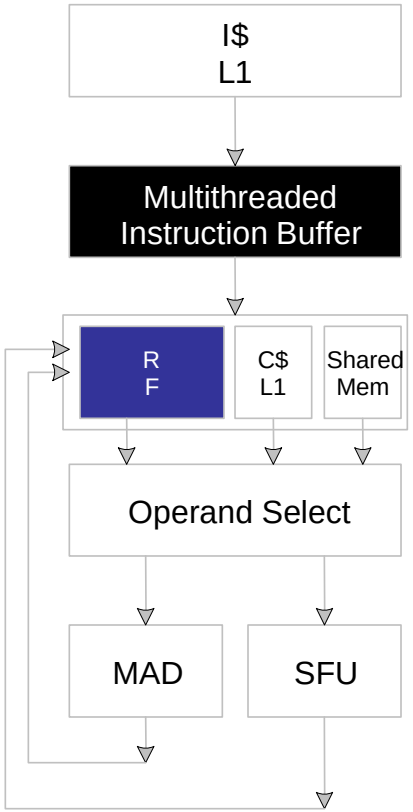


Courtesy:
John Nicols, NVIDIA

- 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

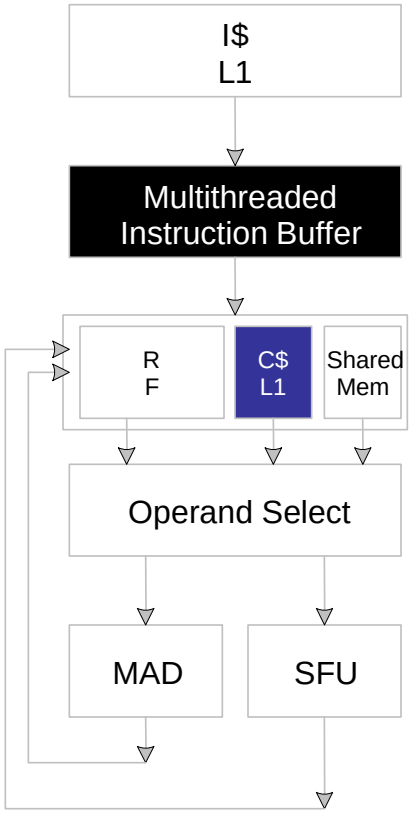
SM Register File

- 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



Constants

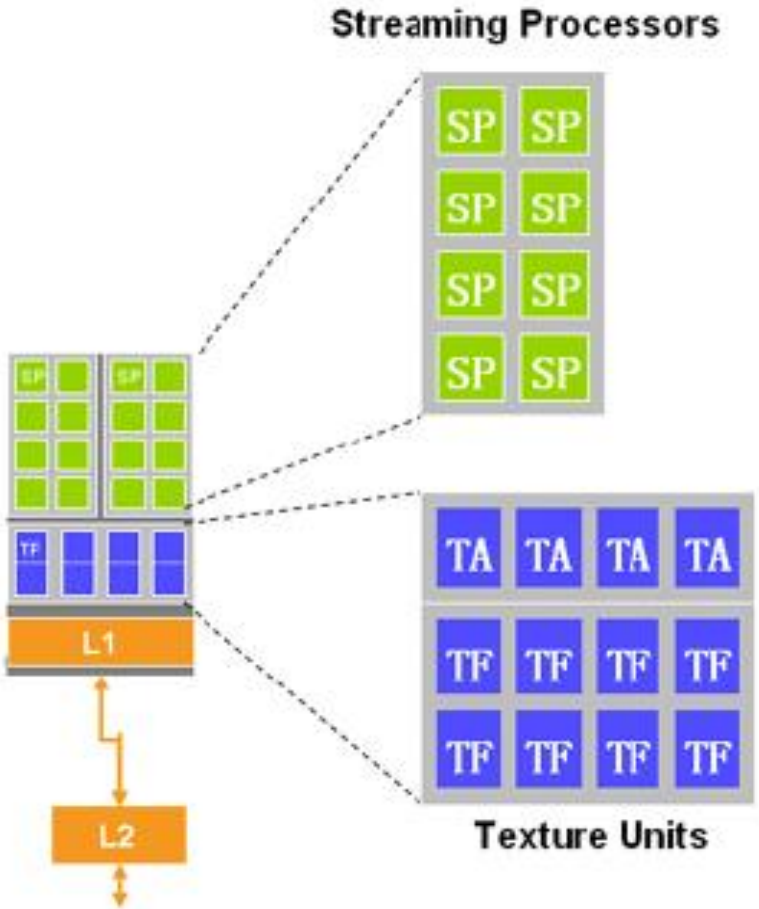
- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!
- Can reduce the number of registers.



Textures

- Textures are 1D,2D, 3D 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
- Texels: elements of the arrays, texture elements

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 ☺