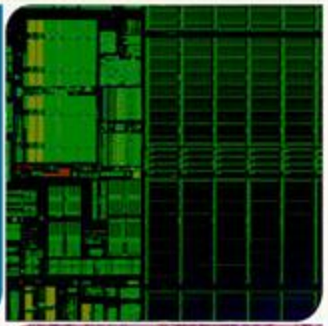
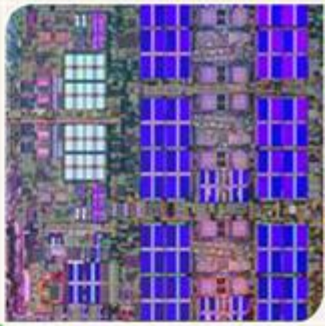
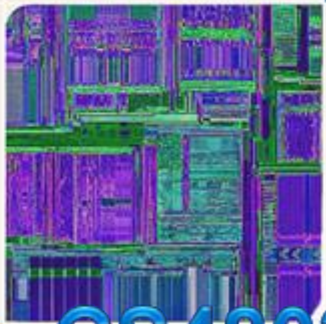




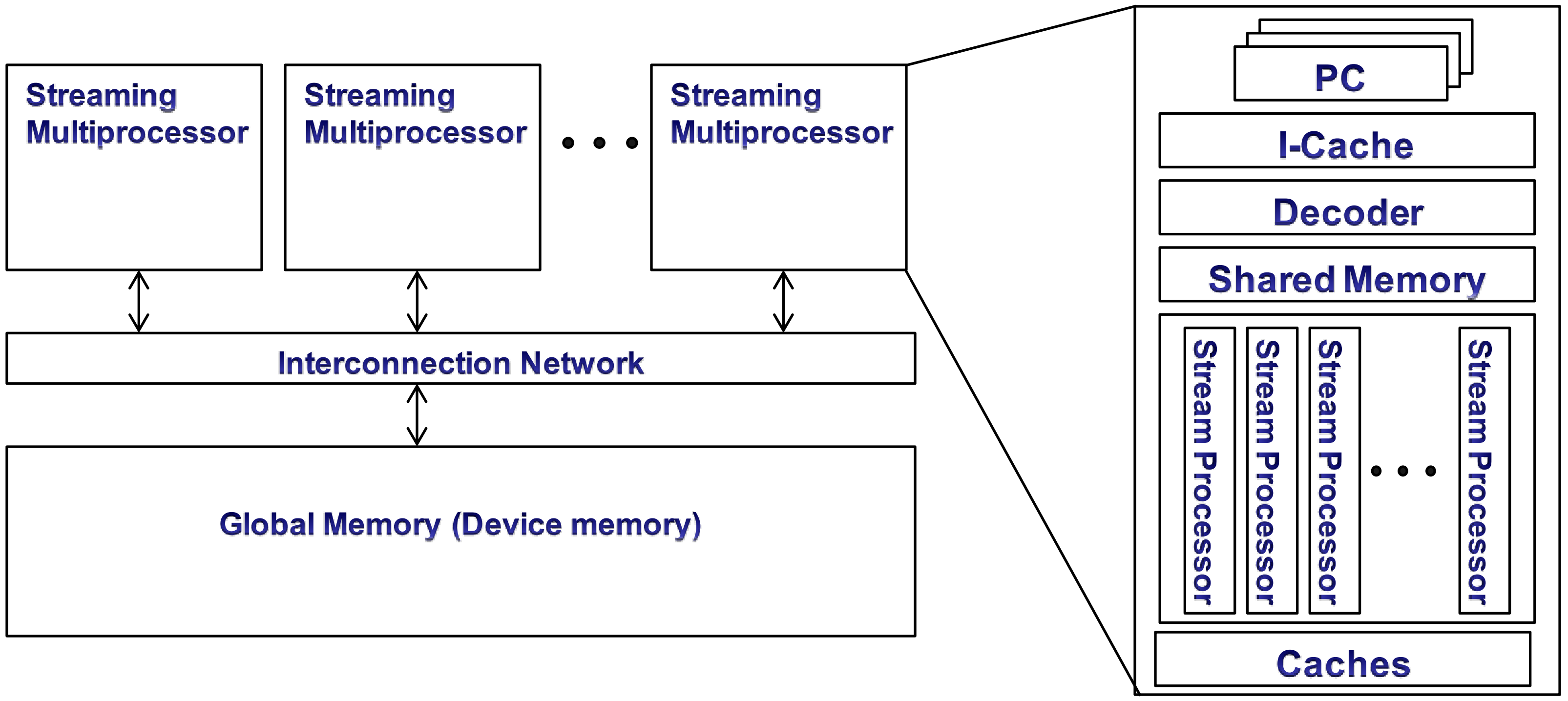
CS4803DGC Design and Programming of Game Consoles

Spring 2011

Prof. Hyesoon Kim

Georgia Tech | College of Computing

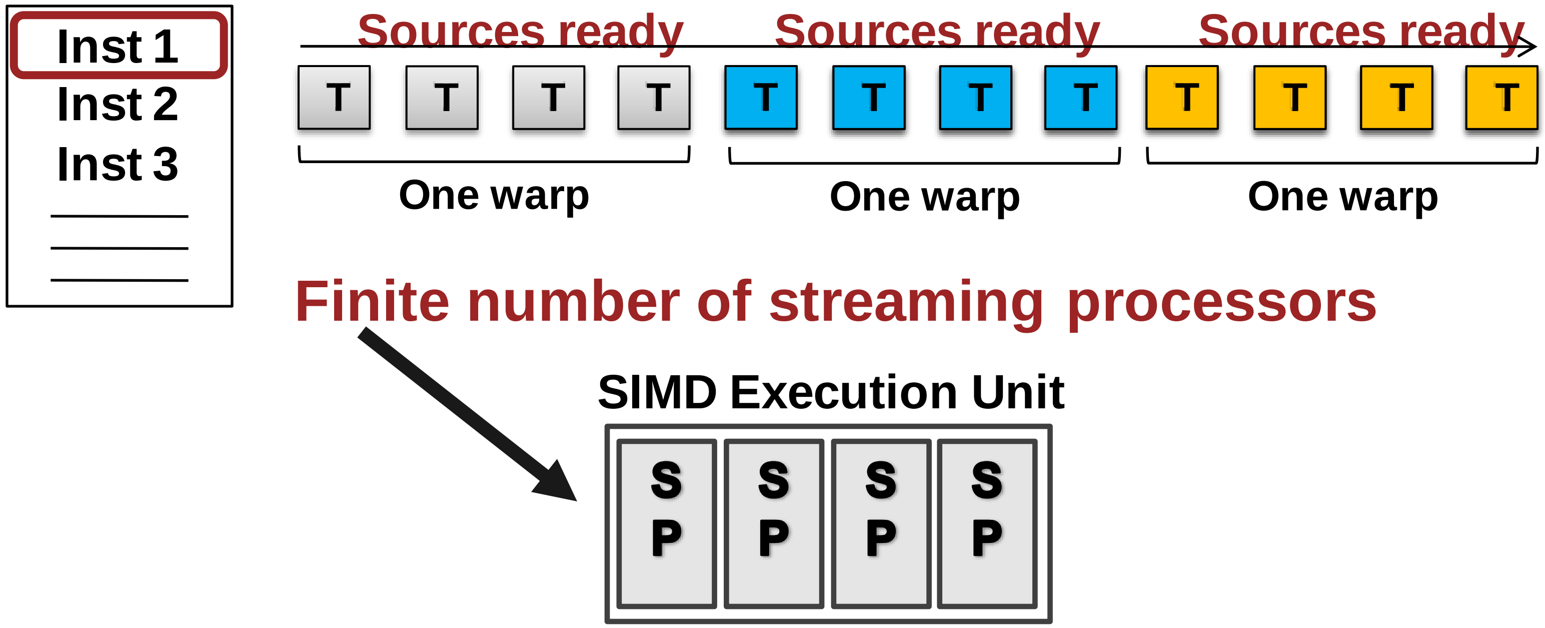
Overview of GPU (Tesla) Architecture



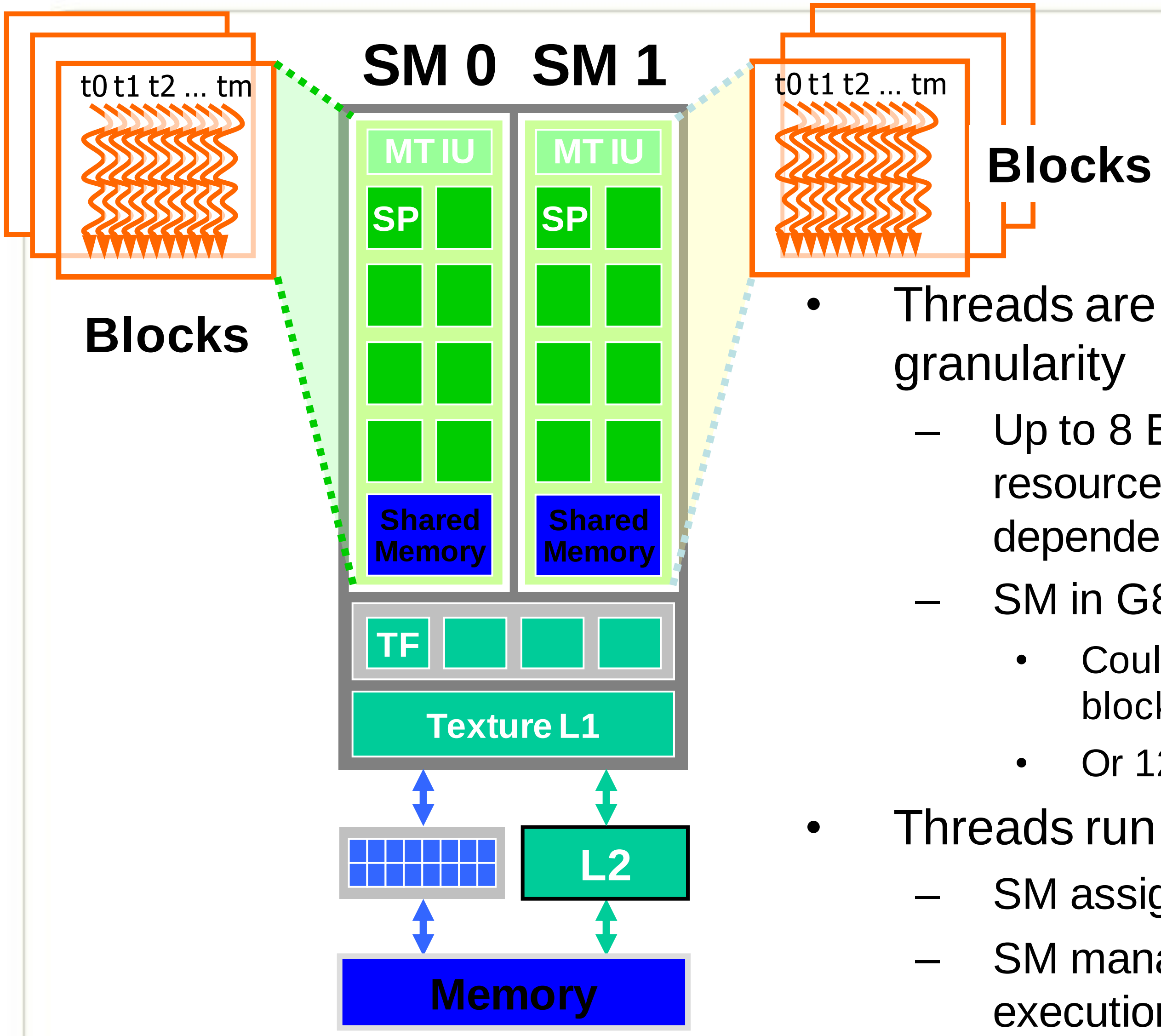
Execution Unit: Warp

- ❑ **Warp is the basic unit of execution**
  - A group of threads (e.g. 32 threads for the Tesla GPU architecture)

Warp Execution

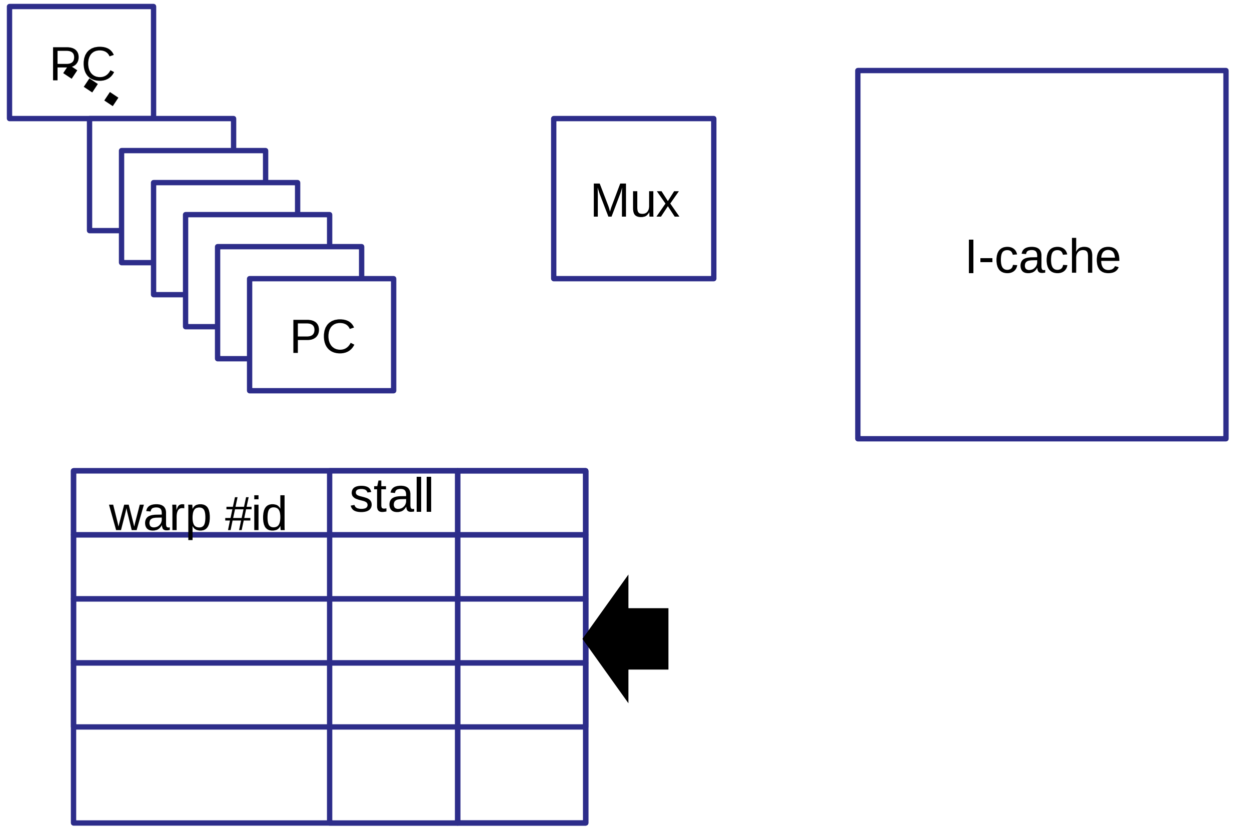


SM Executes Blocks

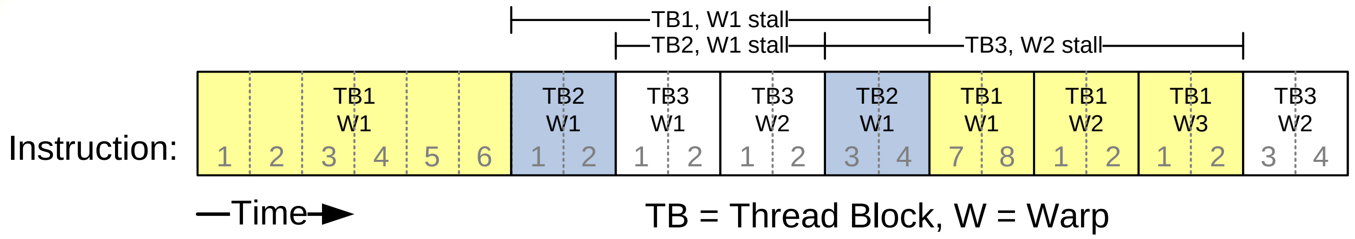


- Threads are assigned to SMs in Block granularity
  - Up to 8 Blocks to each SM as resource allows (# of blocks is dependent on the architecture)
  - SM in G80 can take up to 768 threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently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

Warp Maintaing Unit



Pipeline



Kirk & Hwu

- Fetch
  - One instruction for each warp (could be further optimizations)
  - Round Robin, Greedy-fetch (switch when stall events such as branch, I-cache misses, buffer full)
- Thread scheduling polices
  - Execute when all sources are ready
  - In-order execution within warps
  - Scheduling polices: Greedy-execution, round-robin

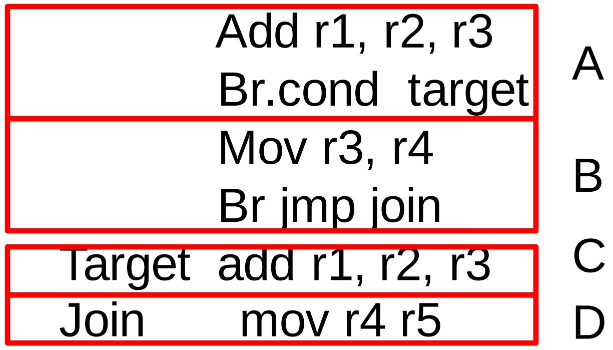
No Branch Prediction. Why?

- 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

Georgia Tech College of Computing

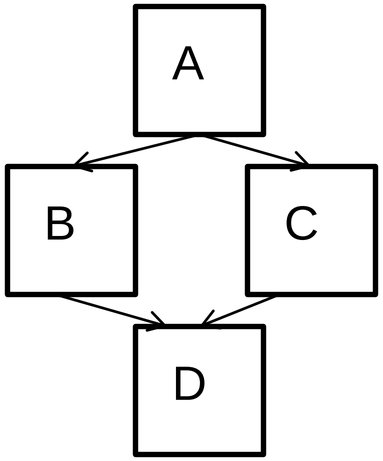
Background: CFG (Control Flow Graph)

- 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
```

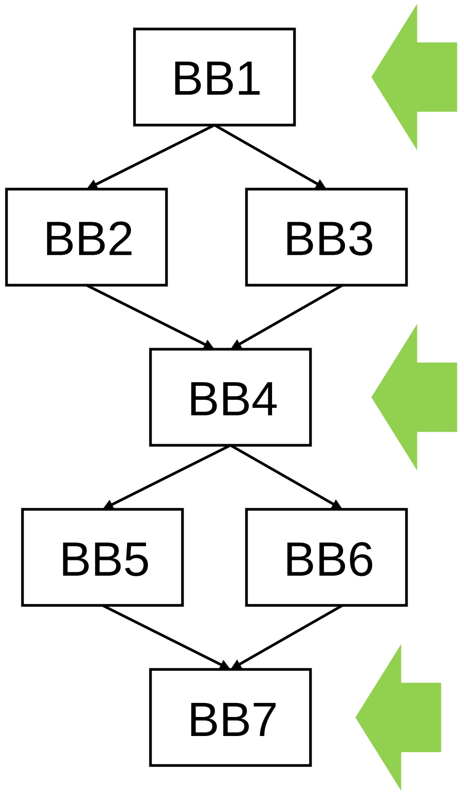
Control-flow graph



$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
B \arrow[dr] & & C \arrow[dl] \\
 & D &
\end{tikzcd}$$

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

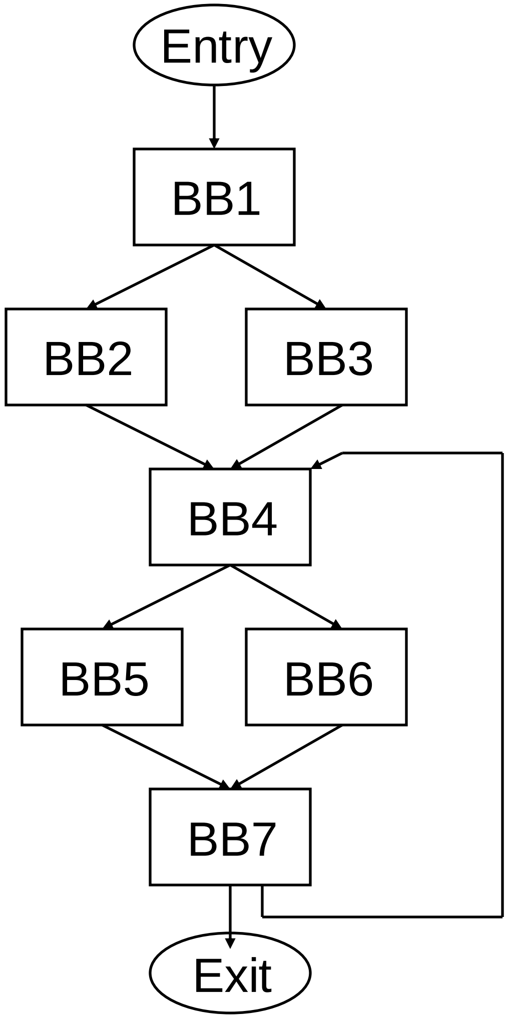

$$\begin{tikzcd}
 & BB1 \arrow[dl] \arrow[dr] & \\
BB2 \arrow[dr] & & BB3 \arrow[dl] \\
 & BB4 \arrow[dl] \arrow[dr] & \\
BB5 \arrow[dr] & & BB6 \arrow[dl] \\
 & BB7 &
\end{tikzcd}$$

Georgia Tech College of Computing

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***

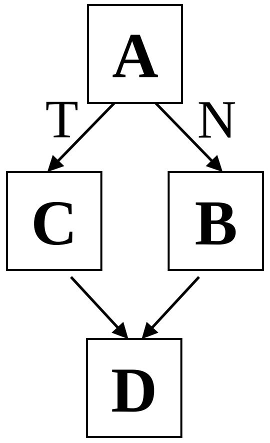
Eliminating Branches

- Predication
- Loop unrolling

Predication

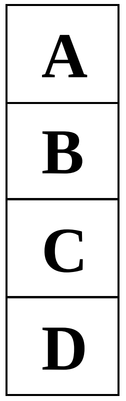
```
if (cond) {
      b = 0;
}
else {
      b = 1;
}
```

(normal branch code)



```
A    p1 = (cond)
     branch p1, TARGET
B    mov b, 1
     jmp JOIN
C  TARGET:
     mov b, 0
```

(predicated code)



```
A          p1 = (cond)
B   (!p1) mov  b, 1
C    (p1) mov  b, 0
```

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;
```

$\Longrightarrow$

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

Reduction Example

- Sum { 1- 100}, How to calculate?

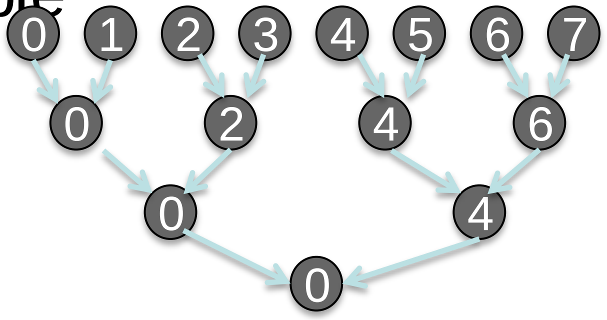
Handling Branch Instructions

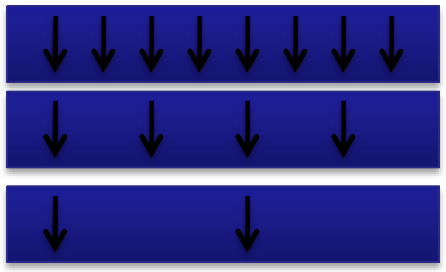
- Reduction example

If (threadId.x%==2)

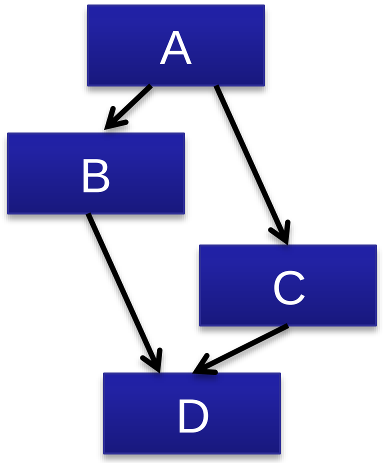
If (threadId.x%==4)

If (threadId.x%==8)

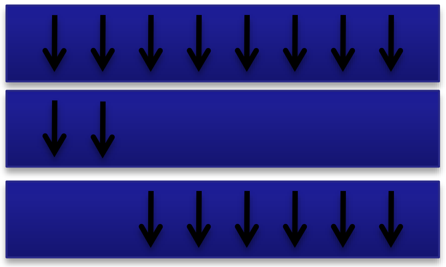




- What about other threads?
- What about different paths?



```
If (threadid.x>2) {
            do work B}
else {
            do work C
}
```



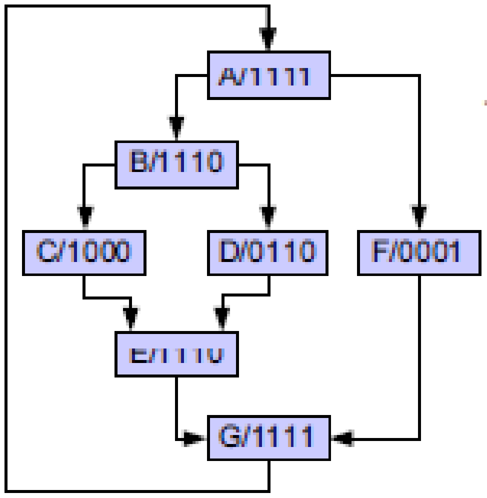
Divergent branch!

From Fung et al. MICRO '07

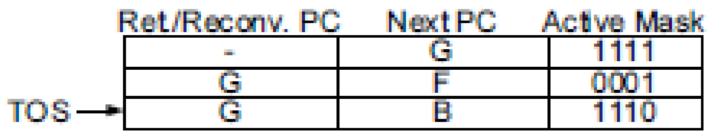
Divergent Branches

- All branch conditions are serialized and will be executed
  - Parallel code → sequential code
- Divergence occurs within a warp granularity.
- It's a performance issue
  - Degree of nested branches
- Depending on memory instructions, (cache hits or misses), divergent warps can occur
  - Dynamic warp subdivision [Meng'10]
- Hardware solutions to reduce divergent branches
  - Dynamic warp formation [Fung'07]

Stack Based Divergent Branch Execution

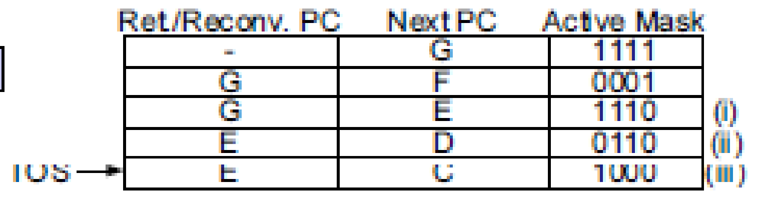


(a) Example Program



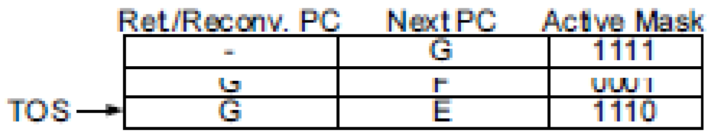
	Ret./Reconv. PC	Next PC	Active Mask
	-	G	1111
	G	F	0001
TOS →	G	B	1110

(c) Initial State



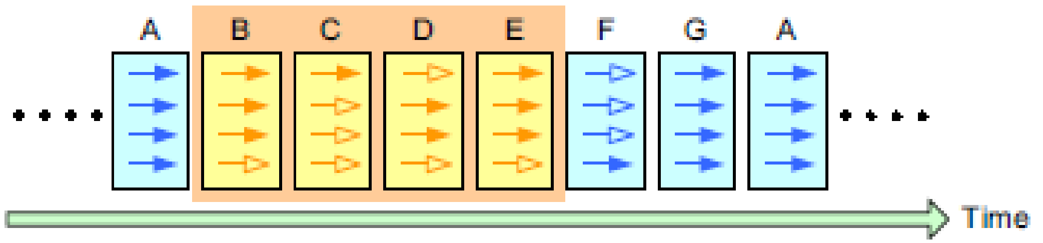
	Ret./Reconv. PC	Next PC	Active Mask	
	-	G	1111	
	G	F	0001	
	G	E	1110	(i)
	E	D	0110	(ii)
TOS →	E	C	1000	(iii)

(d) After Divergent Branch



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

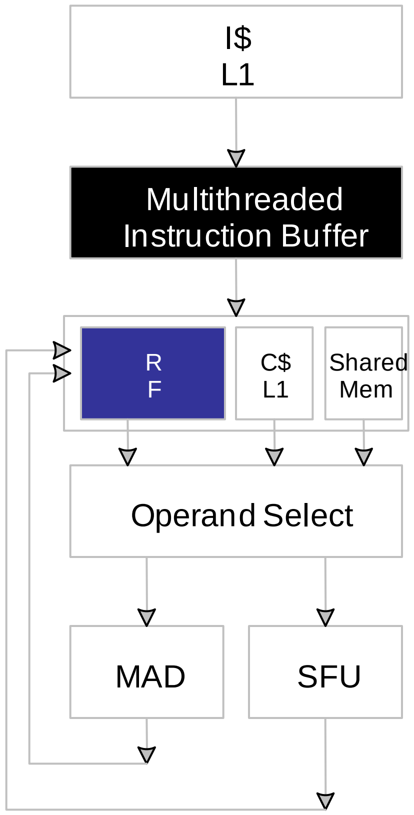
(e) After Reconvergence



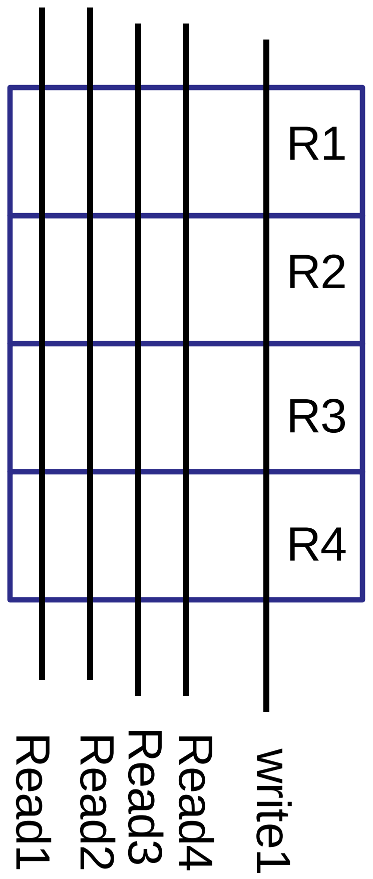
(b) Re-convergence at Immediate Post-Dominator of B

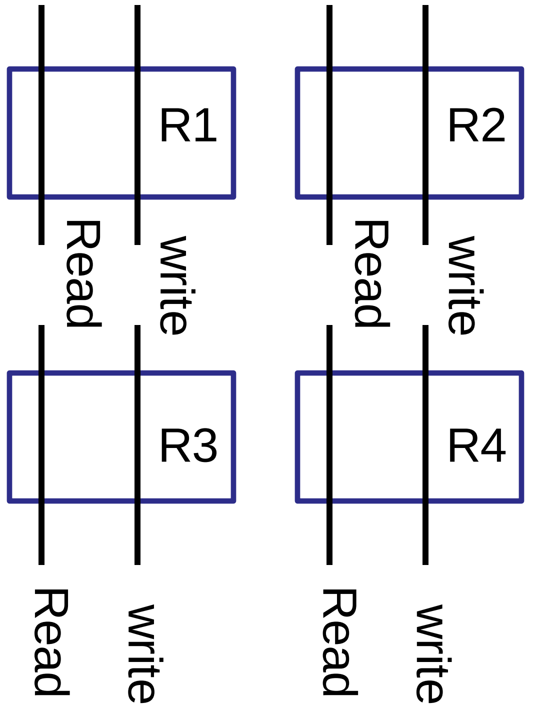
SM Register File

- 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



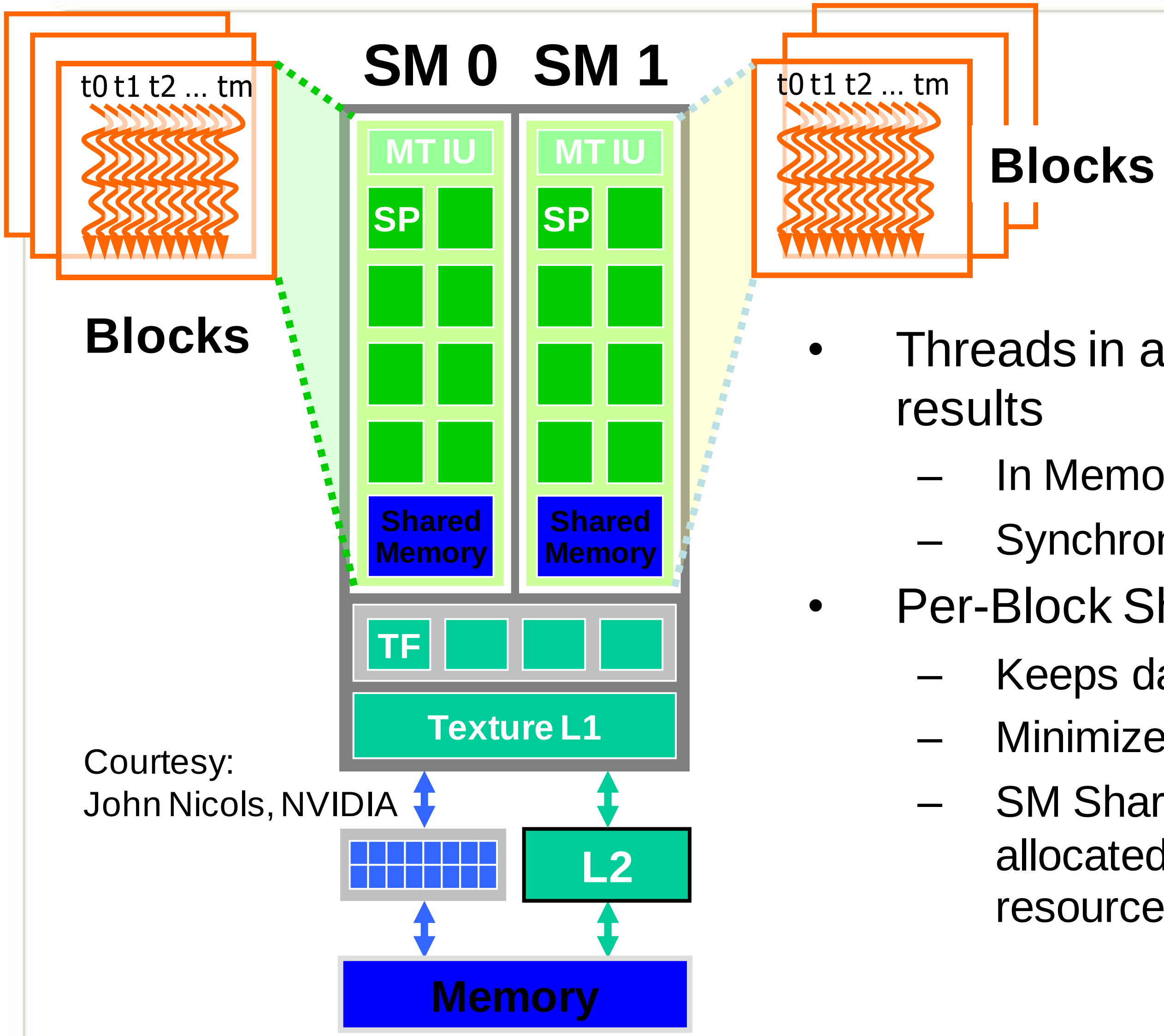
Ports vs. Banks





- Multiple read ports
- Banks

SM Memory Architecture



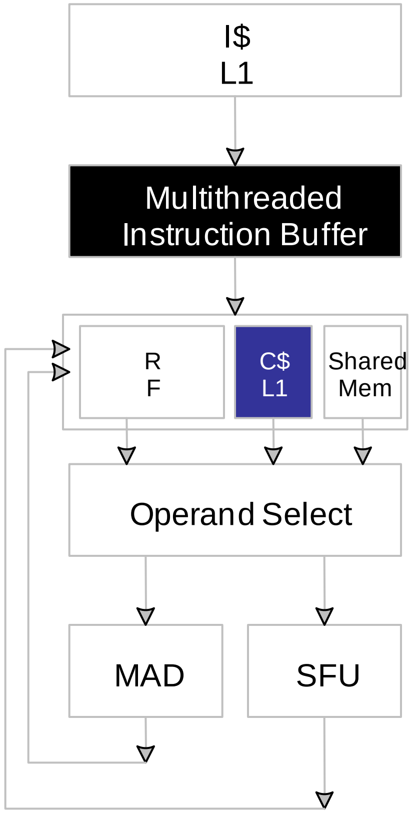
Courtesy:
John Nicols, NVIDIA

- 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

Georgia Tech College of Computing

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

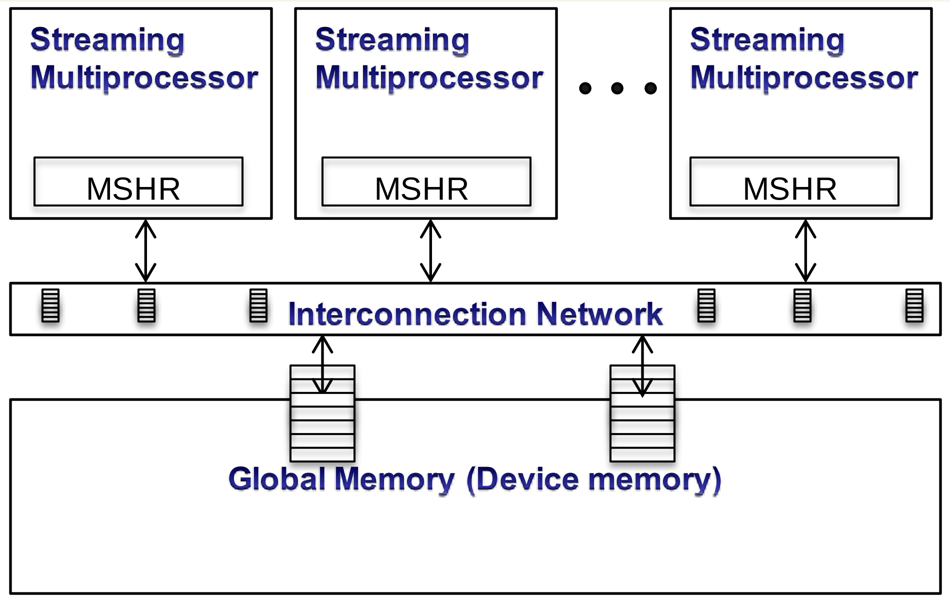
- Textures are 1D,2D, 3D arrays of values stored in global DRAM
- Textures are cached in L1 and L2
- Read-only access
- Caches are optimized for 2D access:
  - Threads in a warp that follow 2D locality will achieve better memory performance
- Texels: elements of the arrays, texture elements

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

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

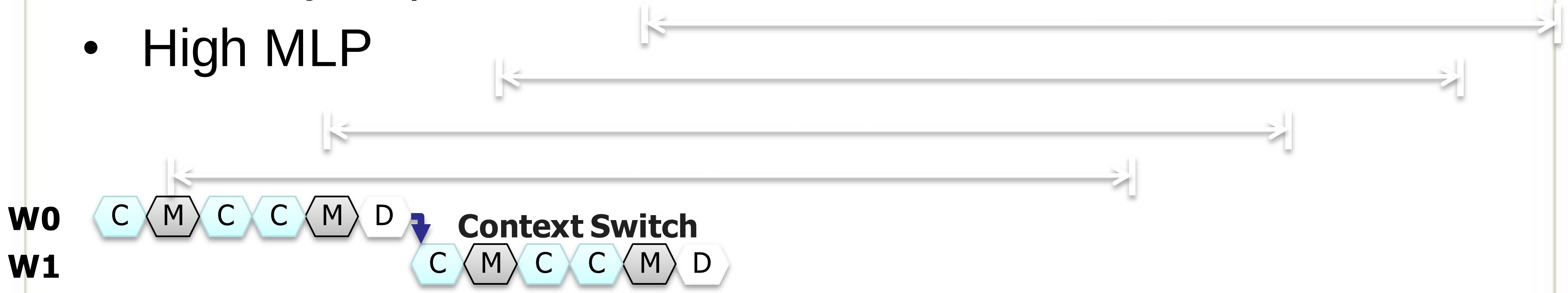
GPU Memory System



- Many levels of queues
- Large size of queues
- High number of DRAM banks
- Sensitive to memory scheduling algorithms
  - FRFCFS >> FCFS
- Interconnection network algorithm to get FRFCFS Effects
  - Yan'09,

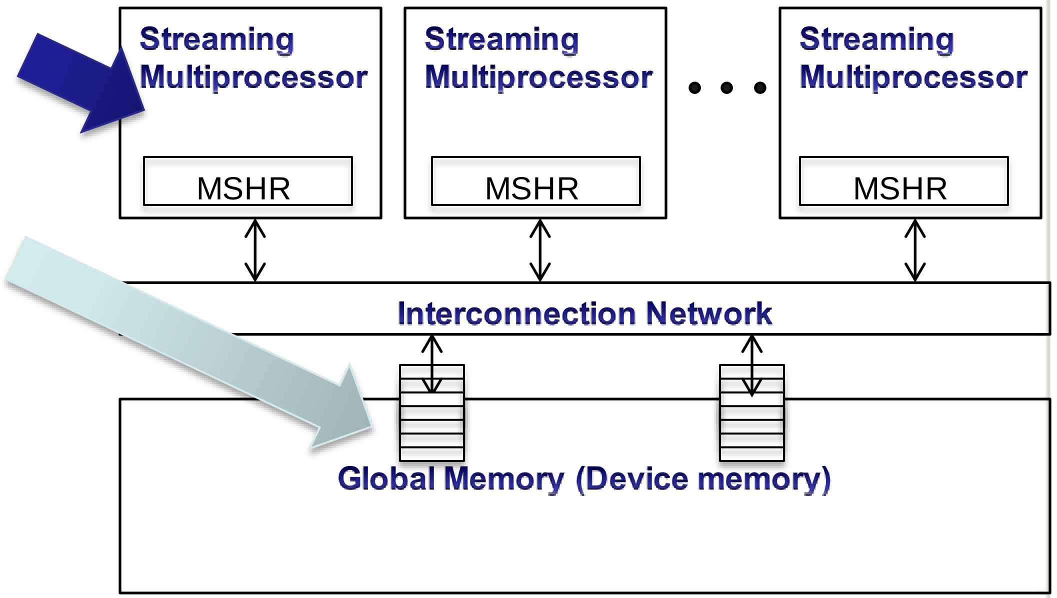
Multiple In-flight Memory Requests

- In-order execution but
- Warp cannot execute an instruction when sources are dependent on memory instructions, not when it generates memory requests
- High MLP



Same Data from Multiple Threads (SDMT)

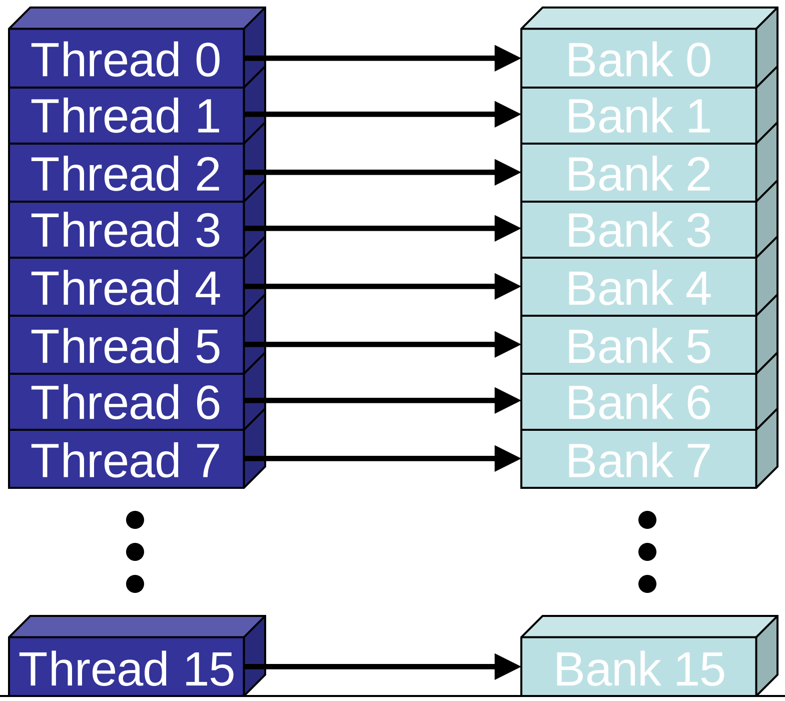
- High Merging Effects
  - Inter-core merging
  - Intra-core merging



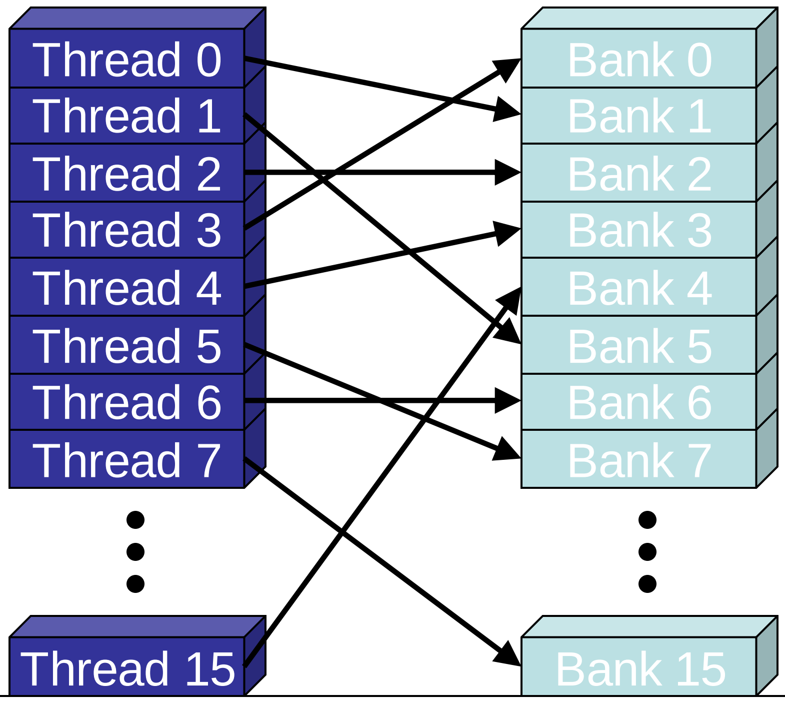
- Techniques to take advantages of this SDMT
  - Compiler optimization[Yang'10]: increase memory reuse
  - Cache coherence [Tarjan'10]
  - Cache increase reuses

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:

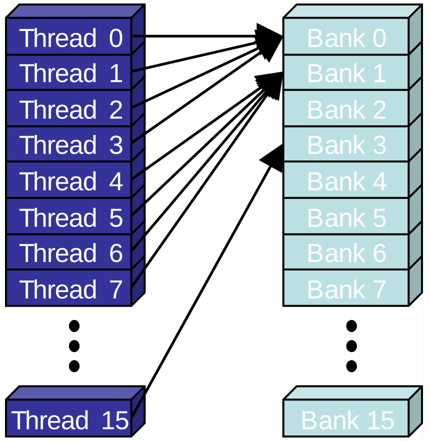
```
foo = shared[baseIndex + threadIdx.x]
```

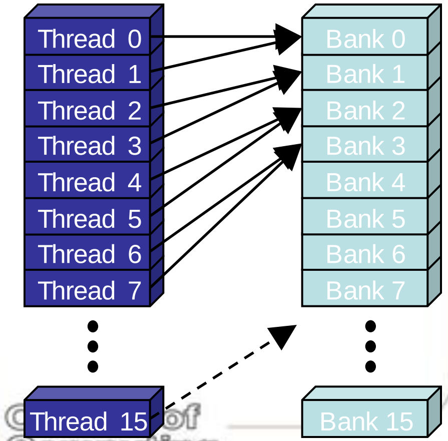
- But not if the data type is smaller
  - 4-way bank conflicts:

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

  - 2-way bank conflicts:

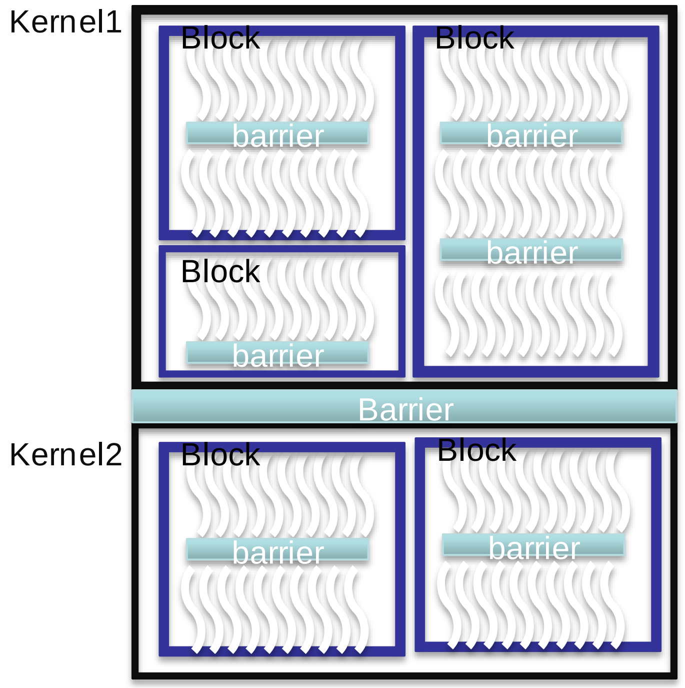
```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```





Synchronization Model

- Bulk-Synchronous Parallel (BSP) program (Valiant [90])
- Synchronization within blocks using explicit barrier
- Implicit barrier across kernels
  - Kernel 1 → Kernel 2
  - C.f.) Cuda 3.x



Global Communications

- Use multiple kernels
- Write to same memory addresses
  - Behavior is not guaranteed
  - Data race
- Atomic operation
  - No other threads can write to the same location
  - Memory Write order is still arbitrary
  - `Keep being updated: atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`
- Performance degradation
  - Fermi increases atomic performance by 5x to 20x (M. Shebanow)

FERMI ARCHITECTURE

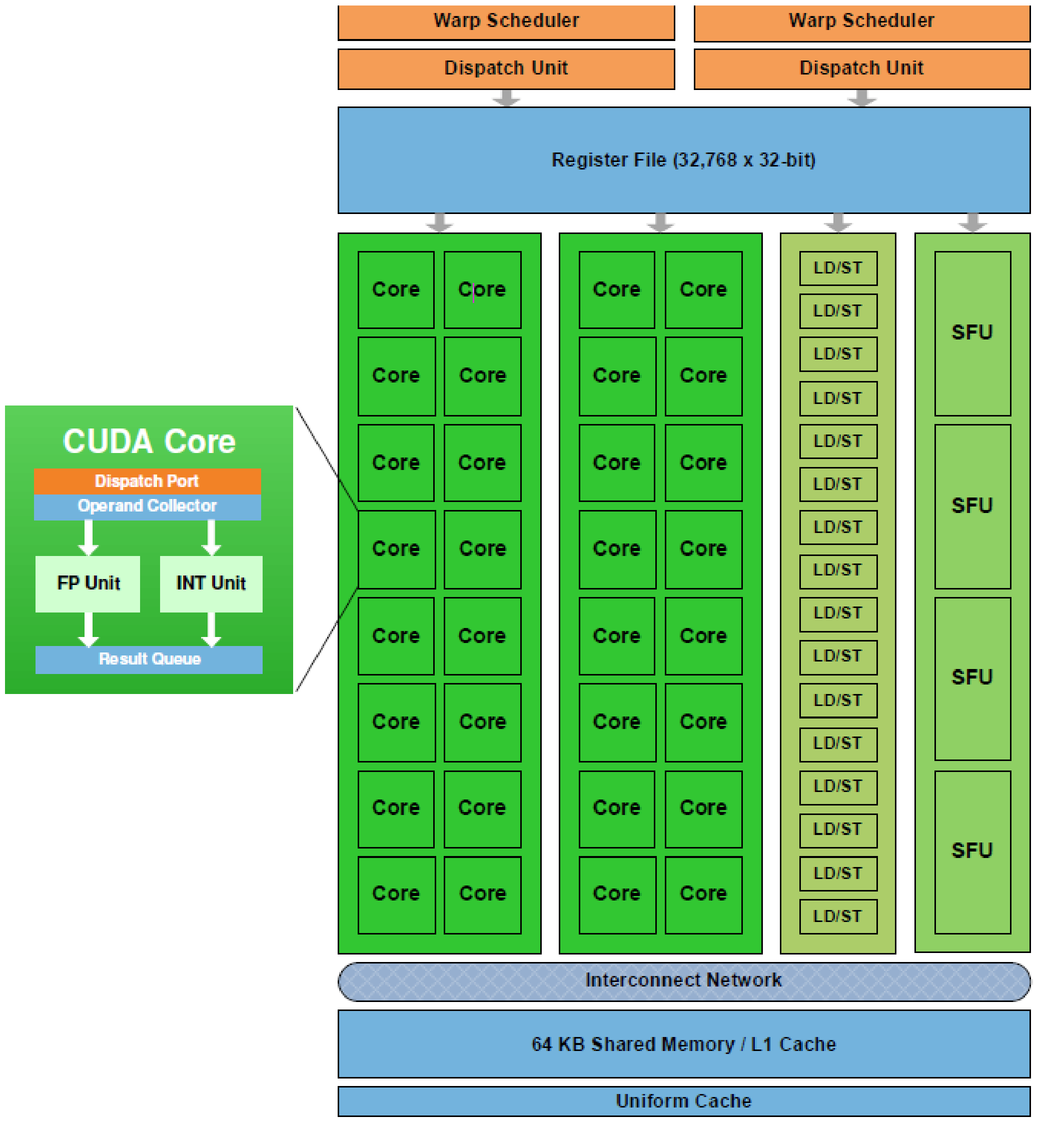
White paper, NVIDIA's Next Generation, CUDA Compute Architecture Fermi

White paper: World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism

Major Architecture Changes in Fermi

- SM
  - 32 CUDA cores per SM (fully 32-lane)
  - Dual Warp Scheduler and dispatches from two independent warps
  - 64KB of RAM with a configurable partitioning of shared memory and L1 cache
- Programming support
  - Unified Address Space with Full C++ Support
  - Full 32-bit integer path with 64-bit extensions
  - Memory access instructions to support transition to 64-bit addressing
- Memory system
  - Data cache , ECC support , Atomic memory operations
- Concurrent kernel execution

- Better integer suppor
- 4 SFU
- Dual warp scheduler
- More TLP
- 16 cores are execute together



Fermi Streaming Multiprocessor (SM)

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit