

EE382V (17325): Principles in Computer Architecture
Parallelism and Locality

Fall 2007

Lecture 13 – GPU Architecture (NVIDIA G80)

Mattan Erez



The University of Texas at Austin

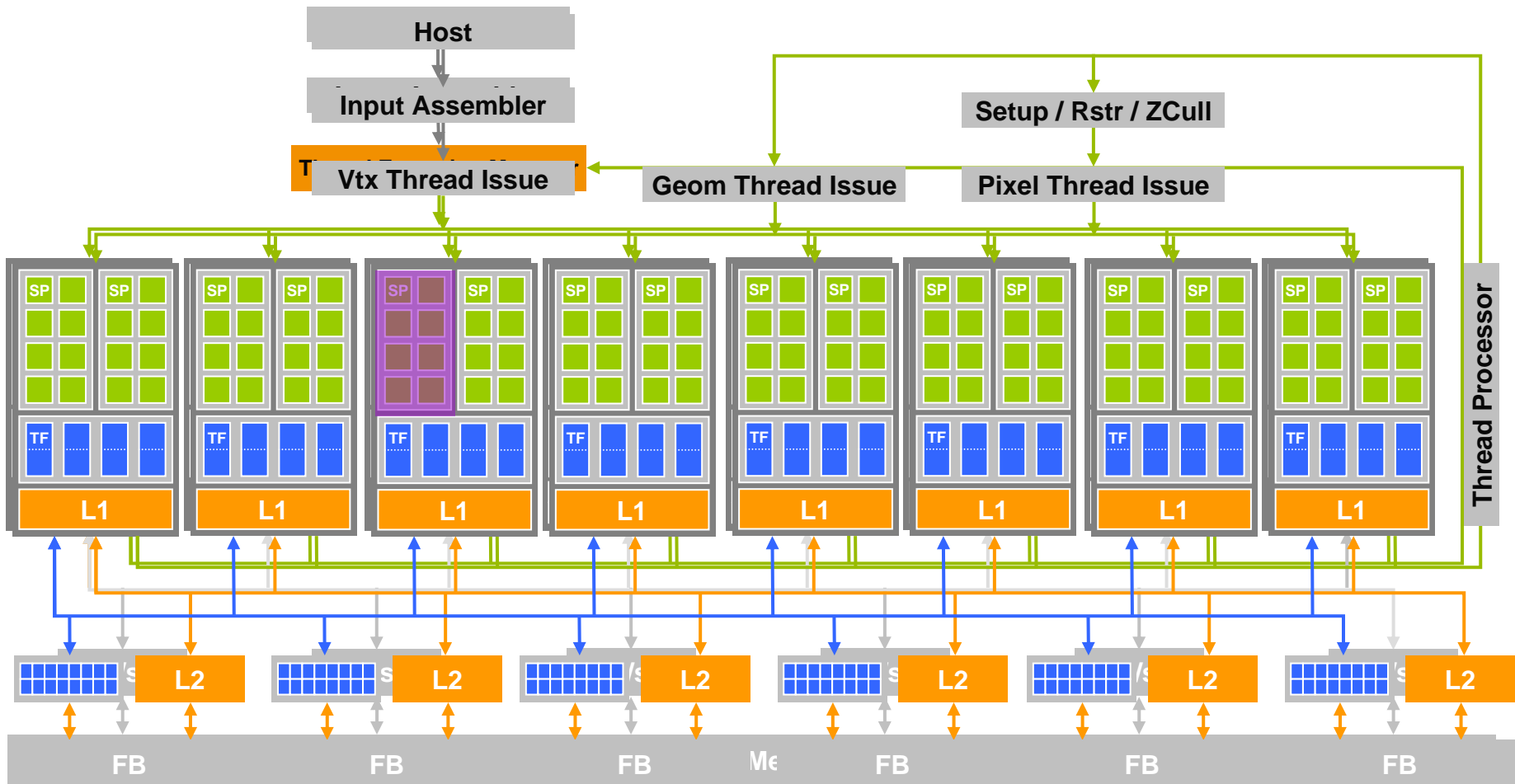
Outline

- Quick recap of execution architecture
- Memory architecture
 - Registers, SM shared memory, global memory, and cached memory
- Synchronization, Communication, and Control flow
- CUDA (moved to Lecture 14)

- Most slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
 - From The University of Illinois ECE 498AI class
- Some slides courtesy Massimiliano Fatica (NVIDIA)

Make the Compute Core The Focus of the Architecture

- The future of GPUs is programming the processors
- And build the operating mode as a special processor

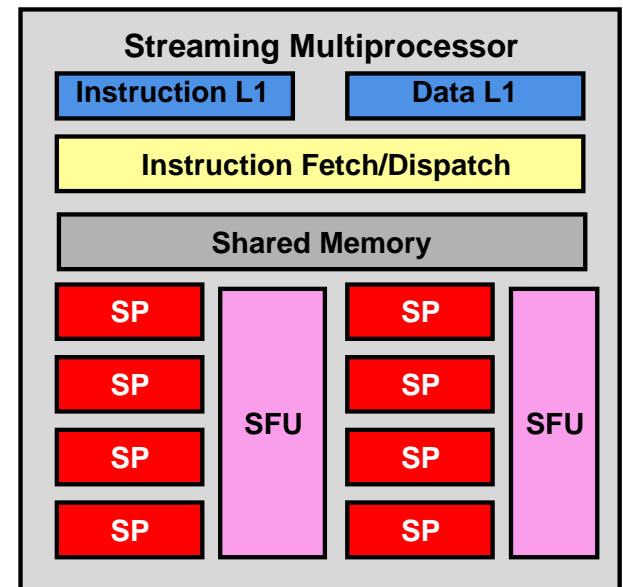


Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
 - 8 Streaming Processors (SP)
 - 2 Super Function Units (SFU)

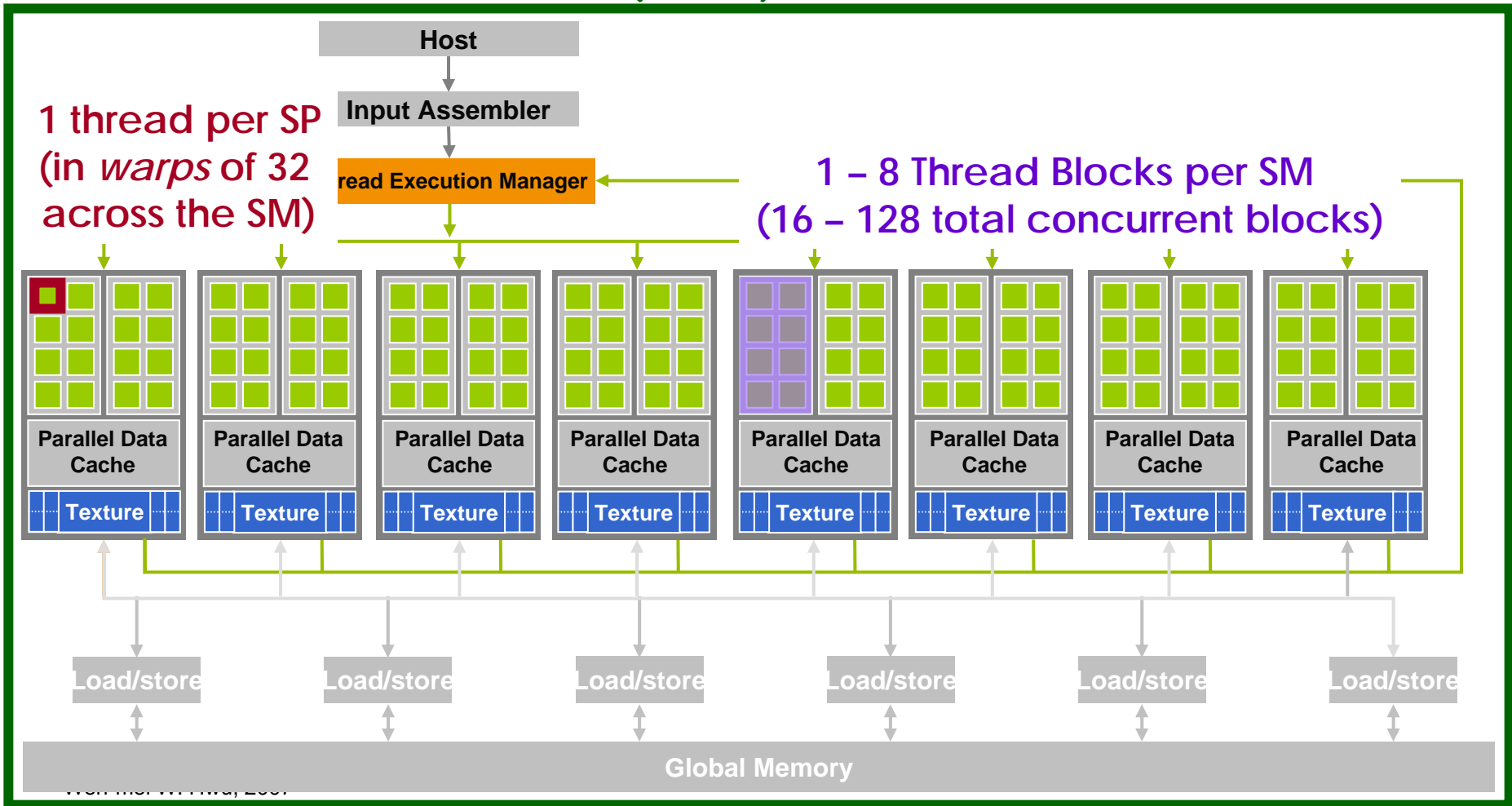
- Multi-threaded instruction dispatch
 - Vectors of 32 threads (*warps*)
 - Up to 16 warps per thread block
 - HW masking of inactive threads in a warp
 - Threads cover latency of texture/memory loads

- 20+ GFLOPS
- 16 KB shared memory
- 32 KB in registers
- DRAM texture and memory access



Make the Compute Core The Focus of the Architecture

1 Grid (kernel) at a time



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

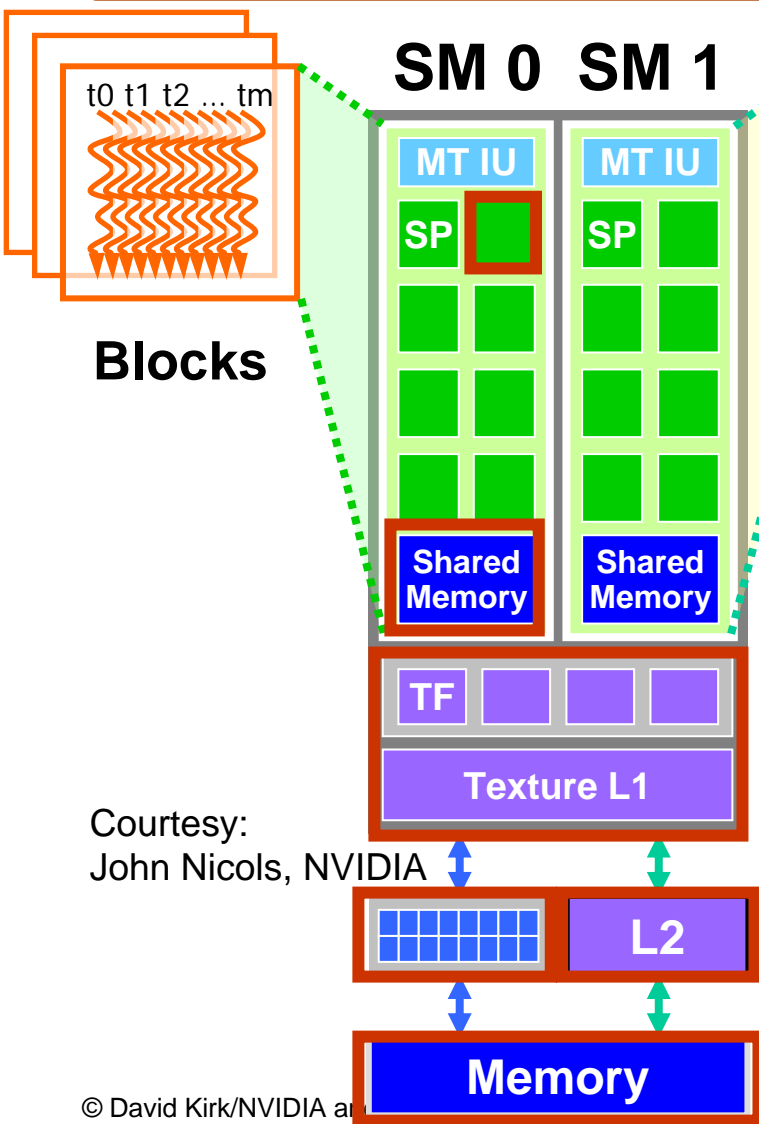
1 - 8 Thread Blocks per SM
(16 - 128 total concurrent blocks)

Outline

- Quick recap of execution architecture
- Memory architecture
 - Registers, SM shared memory, global memory, and cached memory
- Synchronization, Communication, and Control flow
- CUDA

- Most slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
 - From The University of Illinois ECE 498AI class
- A few slides courtesy David Luebke (NVIDIA)

SM Memory Architecture

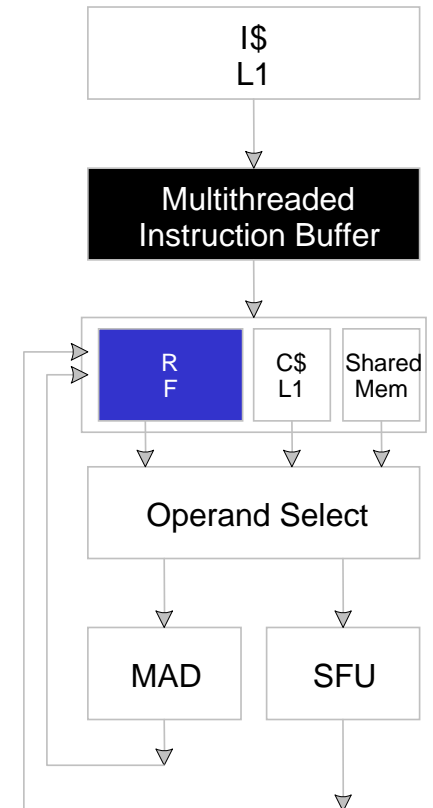


- Registers in SP
 - 1K total per SP
 - shared between thread
 - same per thread in a block)
- Shared memory in SM
 - 16KB total per SM
 - shared between blocks
- Global memory
 - Managed by Texture Units
 - Cache - read only
 - Managed by LD/ST ROP units
 - Uncached - read/Write

Courtesy:
John Nicols, NVIDIA

SM Register File

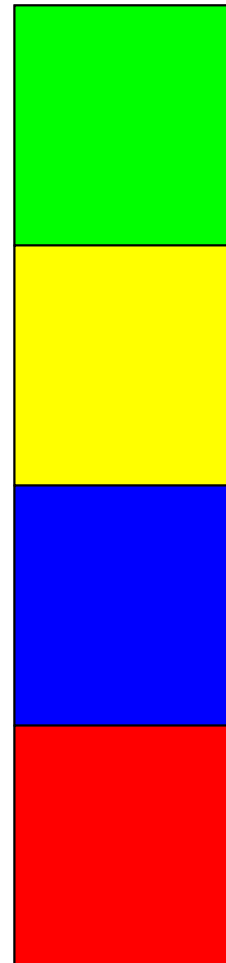
- Register File (RF)
 - 32 KB (1 Kword per SP)
 - Provides 4 operands/clock
- TEX pipe can also read/write RF
 - 2 SMs share 1 TEX
- Load/Store pipe can also read/write RF



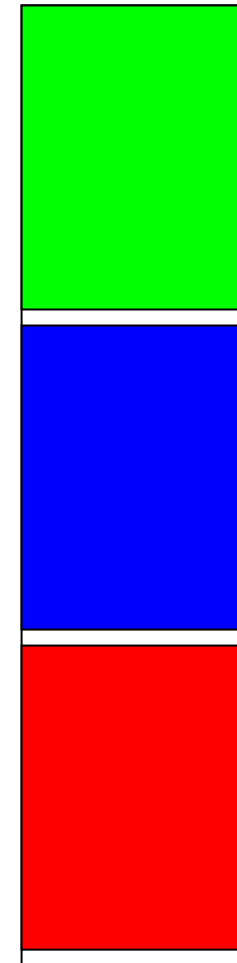
Programmer View of Register File

- 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 Block only access registers assigned to itself

4 blocks



3 blocks



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 $10 \times 256 = 2560$ registers
 - $8192 = 3 \times 2560 + \text{change}$
 - 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 $11 \times 256 = 2816$ registers
 - $8192 < 2816 \times 3$
 - Only two Blocks can run on an SM, **1/3 reduction of parallelism!!!**

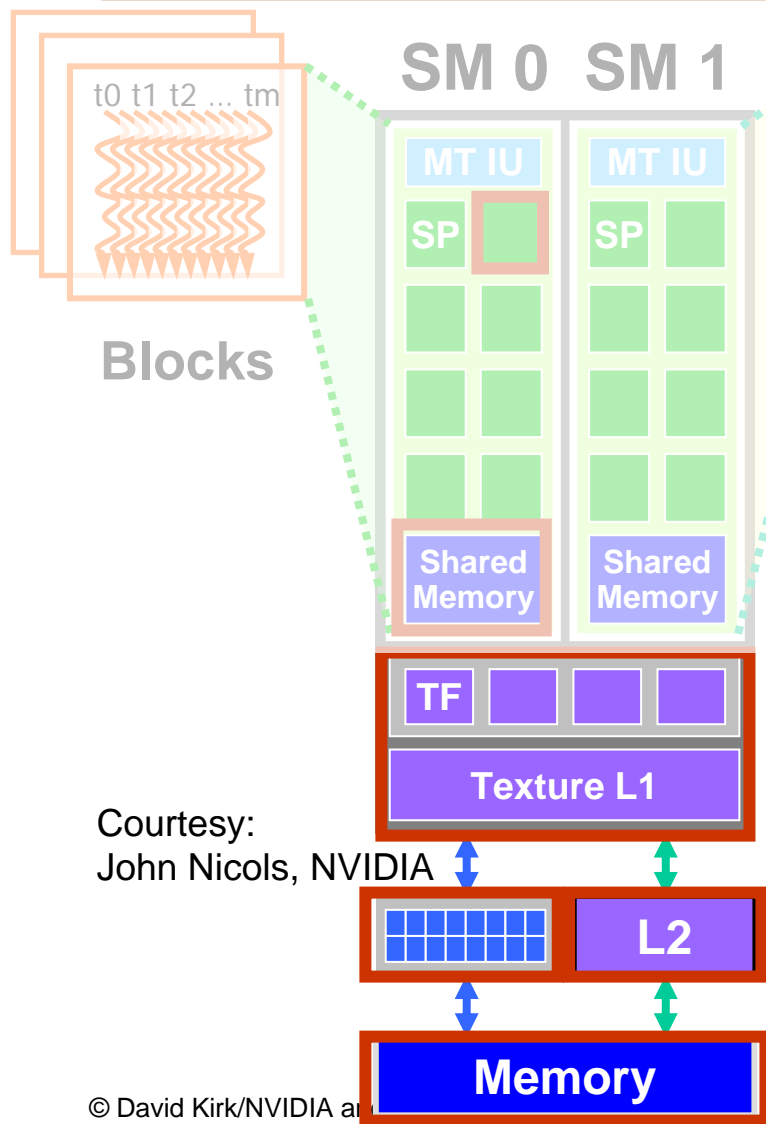
More on Dynamic Partitioning

- 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

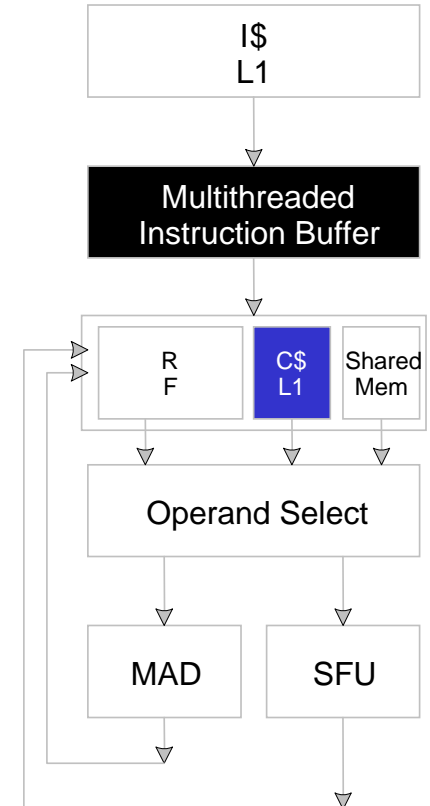


- Registers in SP
 - 1K total per SP
 - shared between thread
 - same per thread in a block)
- Shared memory in SM
 - 16KB total per SM
 - shared between blocks
- Global memory
 - Managed by Texture Units
 - Cache - read only
 - Managed by LD/ST ROP units
 - Uncached - read/Write

Courtesy:
John Nicols, NVIDIA

Constants

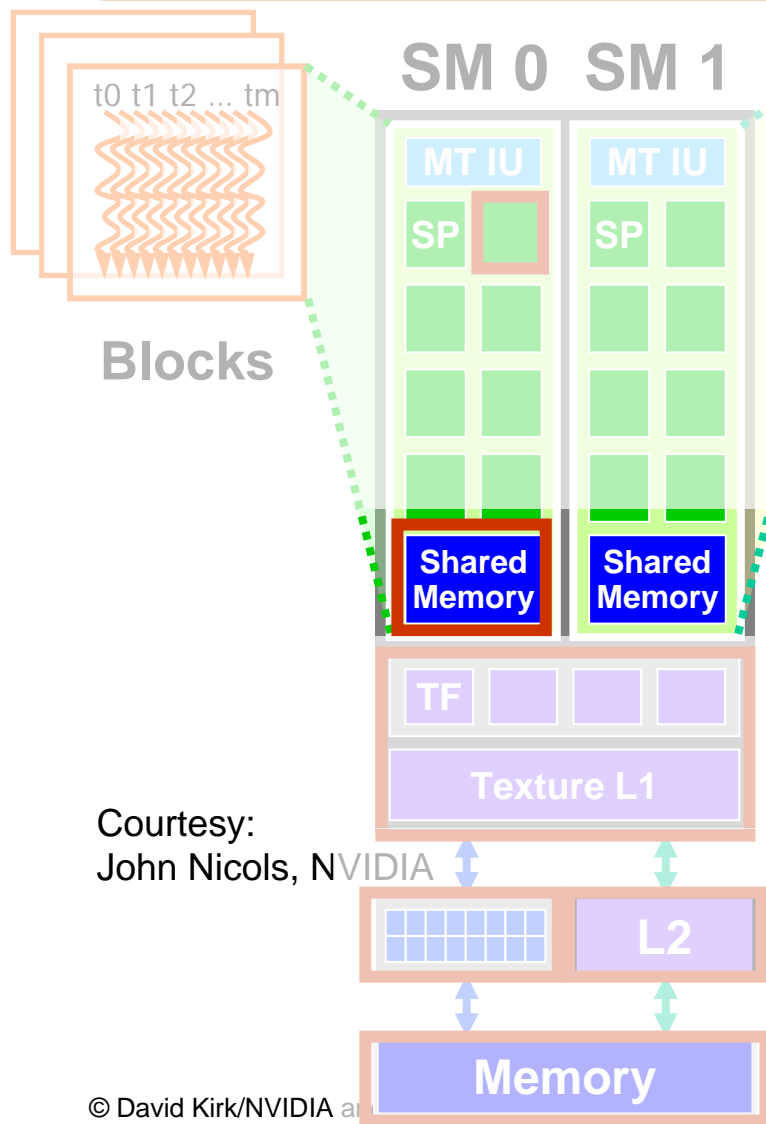
- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
 - L1 per SM
- A constant value can be broadcast to all threads in a Warp
 - Extremely efficient way of accessing a value that is common for all threads in a Block!



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

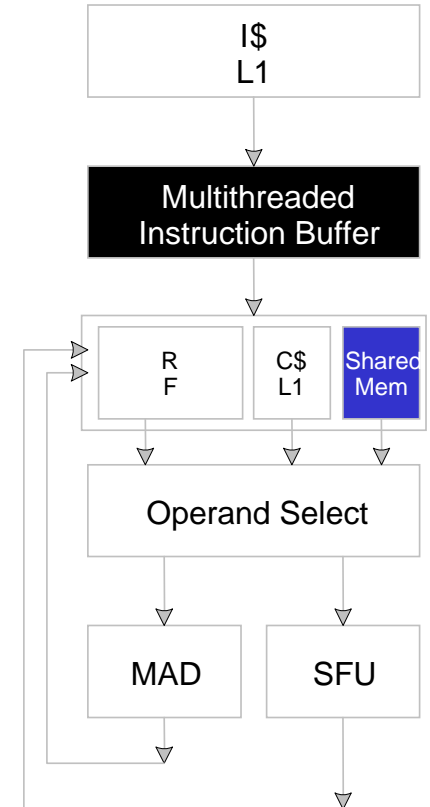


- Registers in SP
 - 1K total per SP
 - shared between thread
 - same per thread in a block)
- Shared memory in SM
 - 16KB total per SM
 - shared between blocks
- Global memory
 - Managed by Texture Units
 - Cache - read only
 - Managed by LD/ST ROP units
 - Uncached - read/Write

Courtesy:
John Nicols, NVIDIA

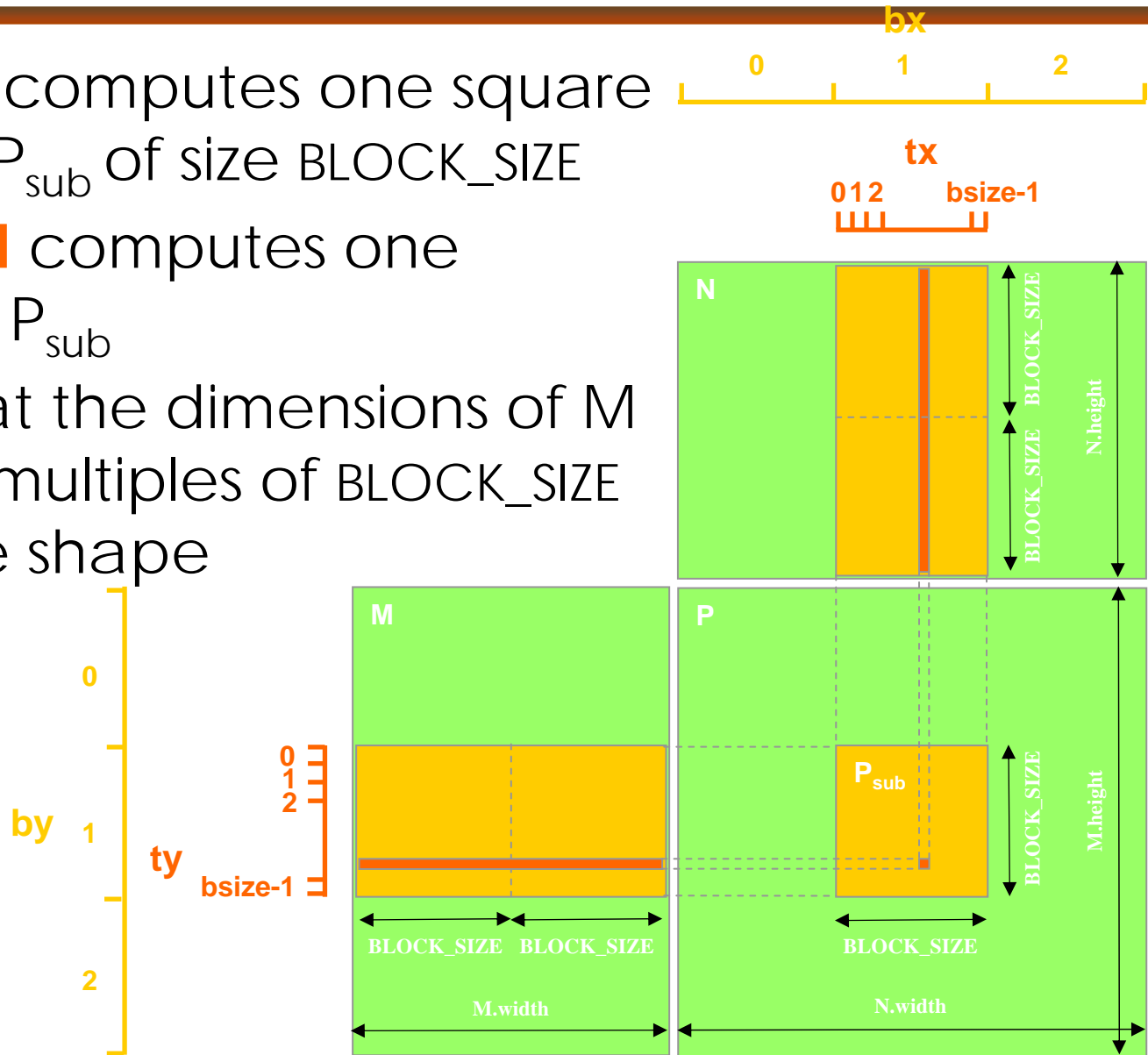
Shared Memory

- 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 ☺



Multiply Using Several Blocks

- One **block** computes one square sub-matrix P_{sub} of size `BLOCK_SIZE`
- One **thread** computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of `BLOCK_SIZE and square shape`



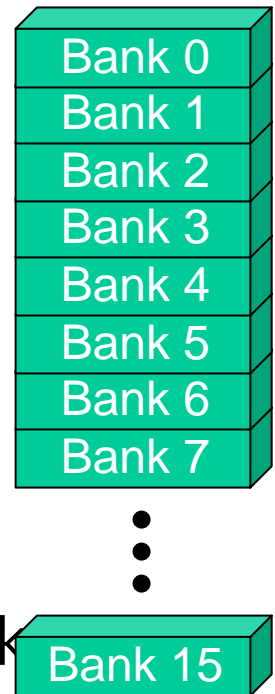


Matrix Multiplication Shared Memory Usage

- Each Block requires $2 * WIDTH^2 * 4$ bytes of shared memory storage
 - For $WIDTH = 16$, each BLOCK requires 2KB, up to 8 Blocks can fit into the Shared Memory of an SM
 - Since each SM can only take 768 threads, each SM can only take 3 Blocks of 256 threads each
 - Shared memory size is not a limitation for Matrix Multiplication of

Parallel Memory Architecture

- In a parallel machine, many threads access memory
 - Therefore, memory is divided into banks
 - Essential to achieve high bandwidth
- Each bank can service one address per cycle
 - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a **bank conflict**

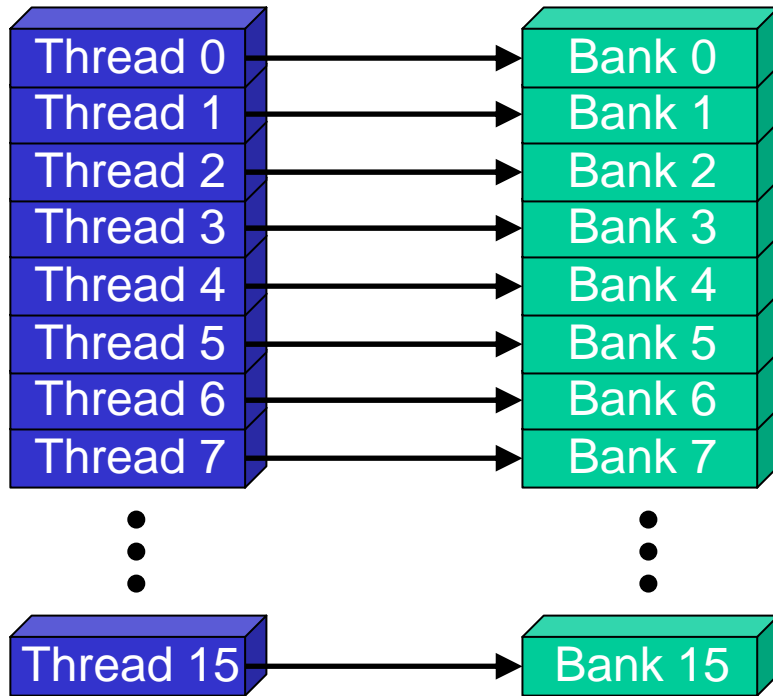


licting accesses are serialized

Bank Addressing Examples

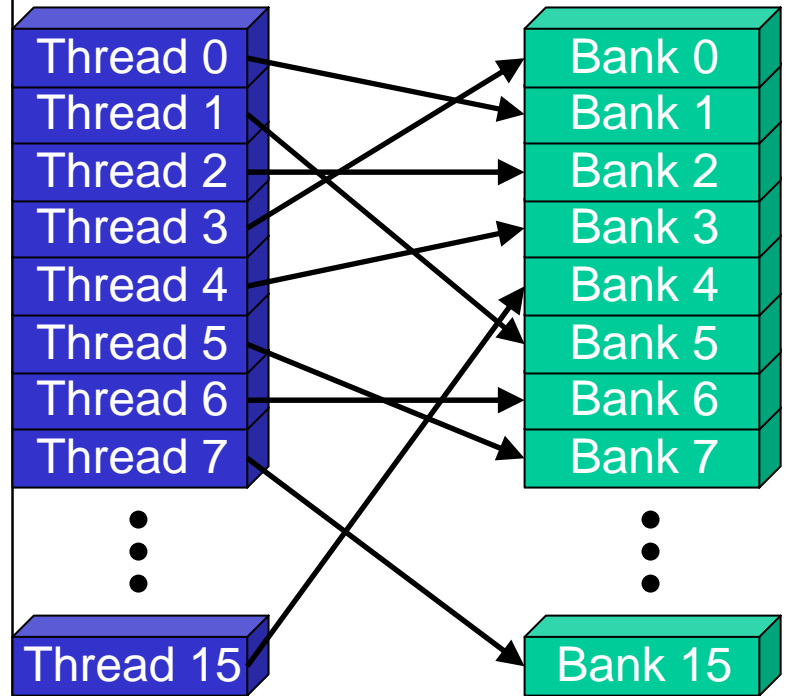
- No Bank Conflicts

- Linear addressing
stride == 1



- No Bank Conflicts

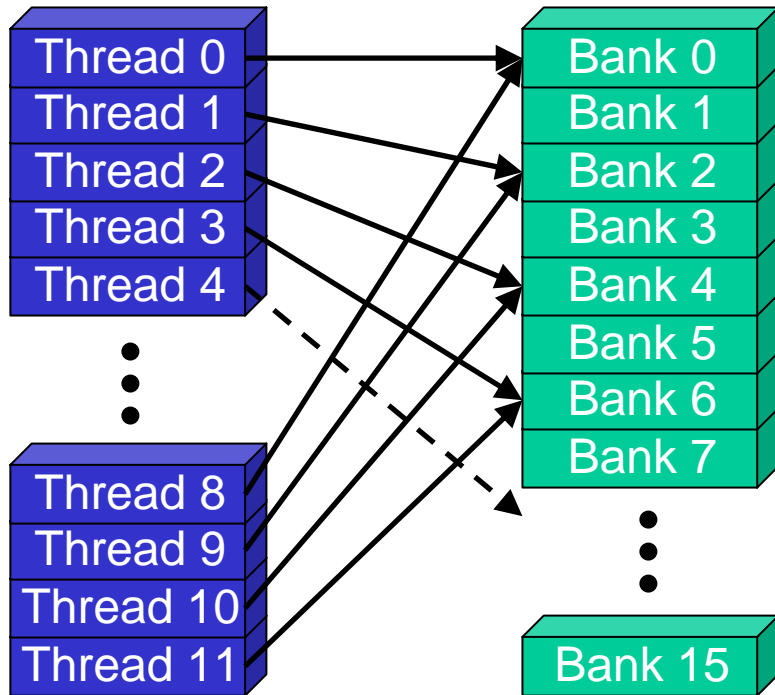
- Random 1:1 Permutation



Bank Addressing Examples

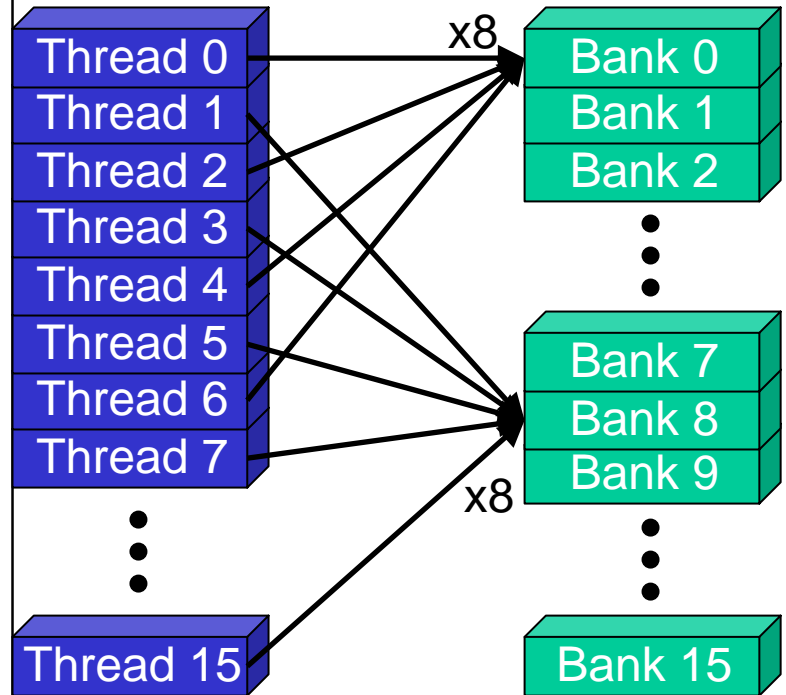
• 2-way Bank Conflicts

- Linear addressing
stride == 2



• 8-way Bank Conflicts

- Linear addressing
stride == 8



How addresses map to banks on G80

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
 - So bank = address % 16
 - Same as the size of a half-warp
 - No bank conflicts between different half-warps, only within a single half-warp

Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
 - If all threads of a half-warp access different banks, there is no bank conflict
 - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- The slow case:
 - Bank Conflict: multiple threads in the same half-warp access the same bank
 - Must serialize the accesses
 - Cost = max # of simultaneous accesses to a single bank

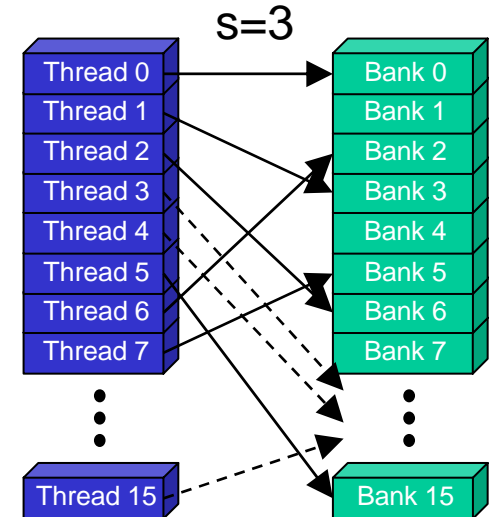
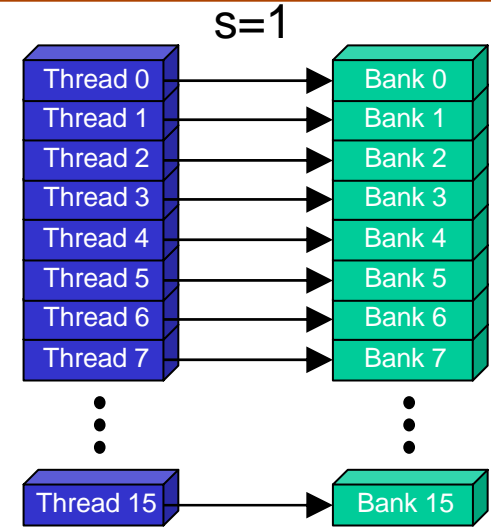
Linear Addressing

- Given:

```

__shared__ float shared[256];
float foo =
    shared[baseIndex + s *
           threadIdx.x];
    
```

- This is only bank-conflict-free if s shares no common factors with the number of banks
 - 16 on G80, so s must be odd



Data types and bank conflicts

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

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

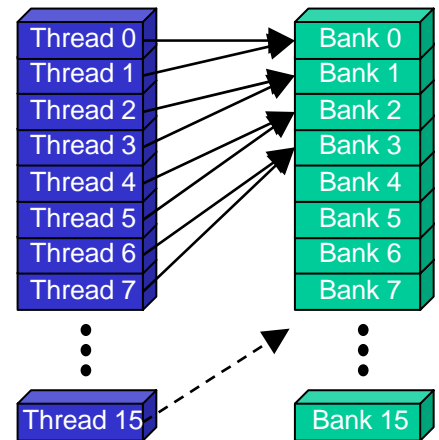
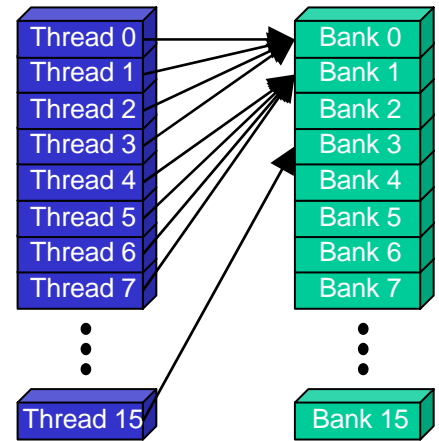
- But not if the data type is smaller

- 4-way bank conflicts:

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

- 2-way bank conflicts:

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

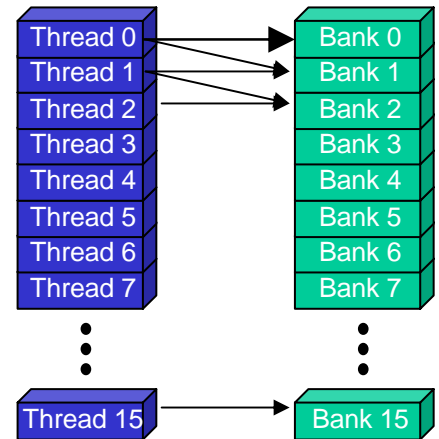


Structs and Bank Conflicts

- Struct assignments compile into as many memory accesses as there are struct members:

```

struct vector { float x, y, z; };
struct myType {
    float f;
    int c;
};
__shared__ struct vector vectors[64];
__shared__ struct myType myTypes[64];
    
```



- This has no bank conflicts for vector; struct size is 3 words
 - 3 accesses per thread, contiguous banks (no common factor with 16)

```

struct vector v = vectors[baseIndex + threadIdx.x];
    
```

- This has 2-way bank conflicts for my Type; (2 accesses per thread)

```

struct myType m = myTypes[baseIndex + threadIdx.x];
    
```

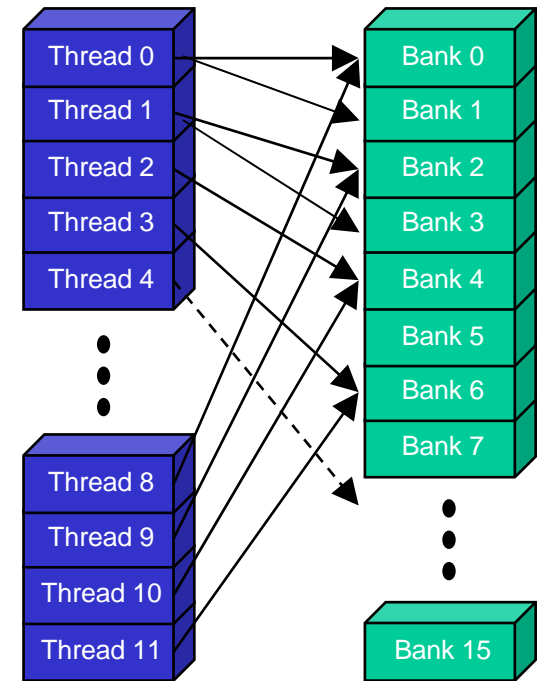
Common Array Bank Conflict Patterns

1D

- Each thread loads 2 elements into shared mem:
 - 2-way-interleaved loads result in 2-way bank conflicts:

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```

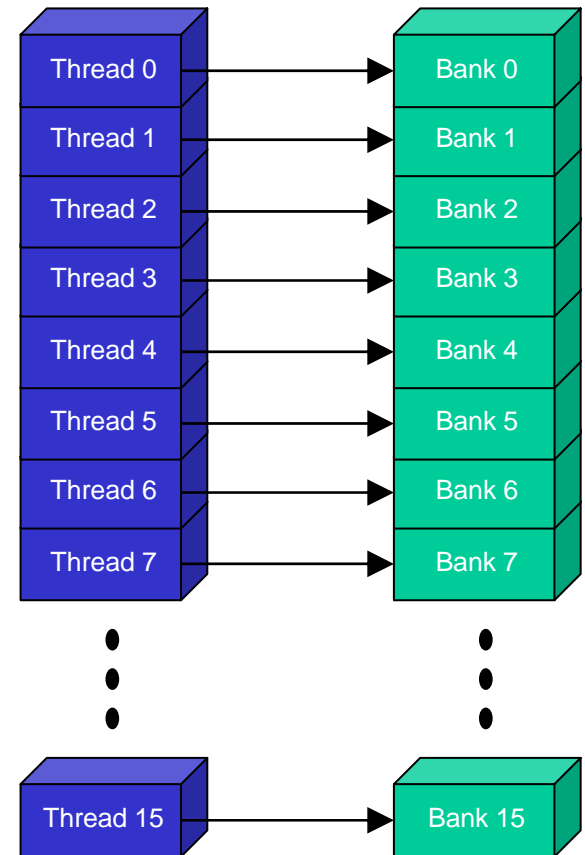
- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffice.
 - Not in shared memory usage where there is no cache line effects but banking effects



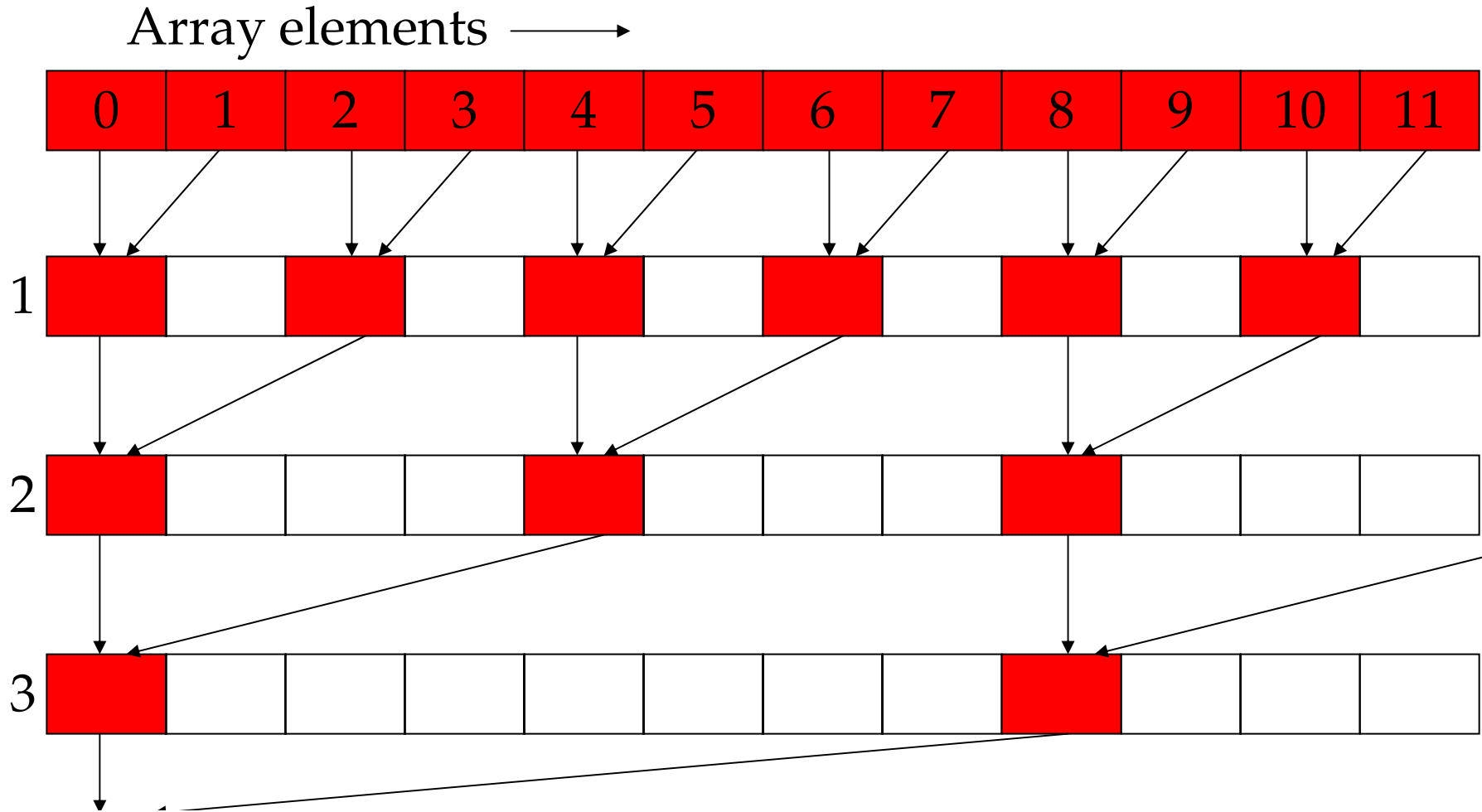
A Better Array Access Pattern

- Each thread loads one element in every consecutive group of `blockDim` elements.

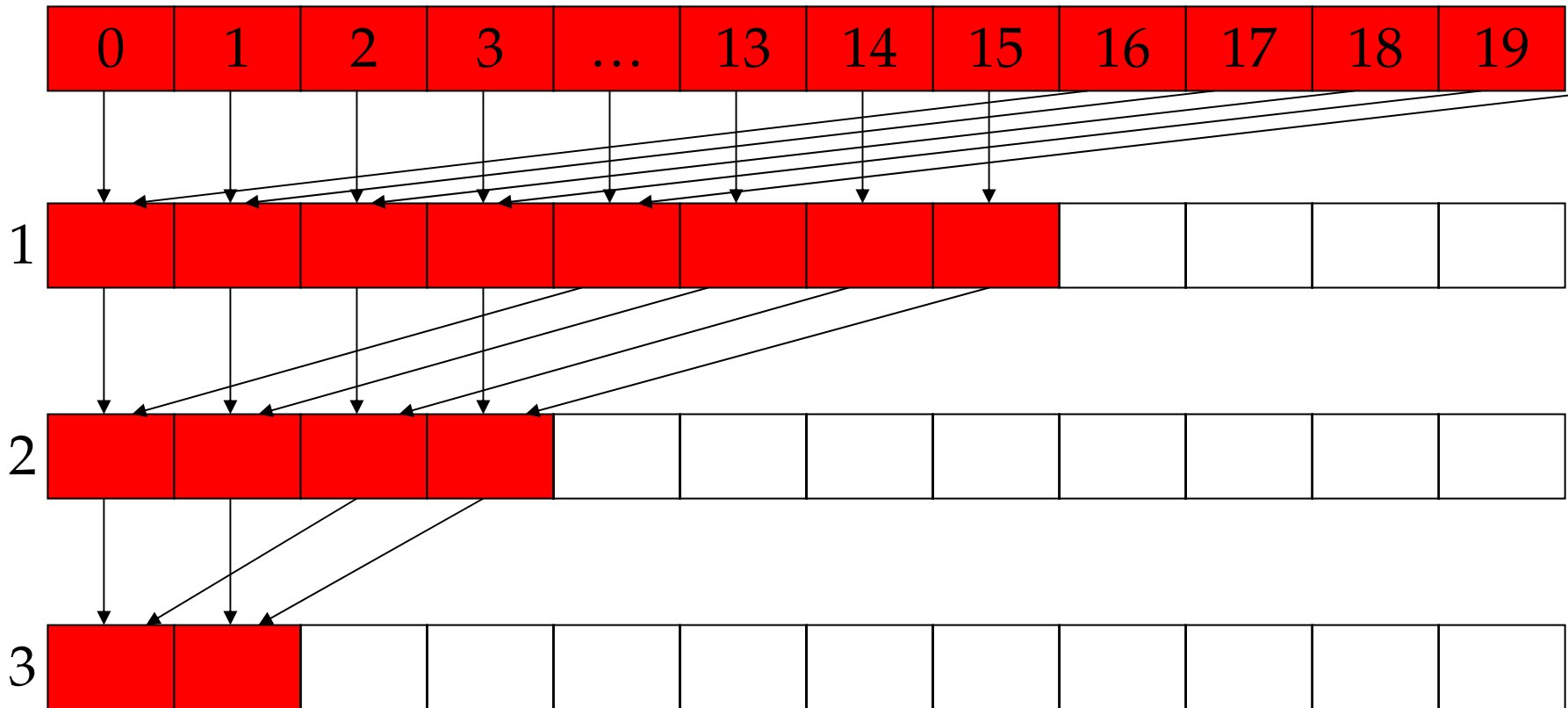
```
shared[tid] = global[tid];
shared[tid + blockDim.x] =
    global[tid + blockDim.x];
```



Vector Reduction with Bank Conflicts



No Bank Conflicts



Common Bank Conflict Patterns (2D)

- Operating on 2D array of floats in shared memory
 - e.g. image processing
- Example: 16x16 block
 - Each thread processes a row
 - So threads in a block access the elements in each column simultaneously (example: row 1 in purple)
 - 16-way bank conflicts: rows all start at bank 0
- Solution 1) pad the rows
 - Add one float to the end of each row
- Solution 2) transpose **before processing**
 - Suffer bank conflicts during transpose
 - But possibly save them later

Bank Indices without Padding

0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
0	1	2	3	4	5	6	7	...	15
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	1	2	3	4	5	6	7	...	15

Bank Indices with Padding

0	1	2	3	4	5	6	7	...	15	0
1	2	3	4	5	6	7	8	...	0	1
2	3	4	5	6	7	8	9	...	1	2
3	4	5	6	7	8	9	10	...	2	3
4	5	6	7	8	9	10	11	...	3	4
5	6	7	8	9	10	11	12	...	4	5
6	7	8	9	10	11	12	13	...	5	6
7	8	9	10	11	12	13	14	...	7	8
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
15	0	1	2	3	4	5	6	...	14	15

Load/Store (Memory read/write) Clustering/Batching

- Use LD to hide LD latency (non-dependent LD ops only)
 - Use same thread to help hide own latency
- Instead of:
 - LD 0 (long latency)
 - Dependent MATH 0
 - LD 1 (long latency)
 - Dependent MATH 1
- Do:
 - LD 0 (long latency)
 - LD 1 (long latency - hidden)
 - MATH 0
 - MATH 1
- Compiler handles this!
 - But, you must have enough non-dependent LDs and Math

Outline

- Quick recap of execution architecture
- Memory architecture
 - Registers, SM shared memory, global memory, and cached memory
- Synchronization, Communication, and Control flow
- CUDA

- Most slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
 - From The University of Illinois ECE 498AI class
- A few slides courtesy David Luebke (NVIDIA)

Communication

- How do threads communicate?
- Remember the execution model:
 - Data parallel streams that represent independent vertices, triangles, fragments, and pixels in the graphics world
 - These *never* communicate
- Some communication allowed in compute mode:
 - Shared memory for threads in a thread block
 - No special communication within warp or using registers
 - No communication between thread blocks
 - Kernels communicate through global device memory
- **Mechanisms designed to ensure portability**

Synchronization

- Do threads need to synchronize?
 - Basically no communication allowed
- Threads in a block share memory – need sync
 - Warps scheduled OoO, can't rely on warp order
 - Barrier command for all threads in a block
 - `__syncthreads()`
- Blocks cannot synchronize
 - Implicit synchronization at end of kernel

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
 - $\text{If (TID} > 2) \{ \dots \} \rightarrow \text{If (TID / warp_size} > 2) \{ \dots \}$

Outline

- Quick recap of execution architecture
- Memory architecture
 - Registers, SM shared memory, global memory, and cached memory
- Synchronization, Communication, and Control flow
- CUDA
 - System
 - Programming Model
 - API
- Most slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
 - From The University of Illinois ECE 498AI class
- A few slides courtesy David Luebke (NVIDIA)

Compute Unified Device Architecture

- CUDA is a programming system for utilizing the G80 processor for compute
 - CUDA follows the architecture very closely

- General purpose programming model
 - User kicks off batches of threads on the GPU
 - GPU = dedicated super-threaded, massively data parallel co-processor

Matches architecture features

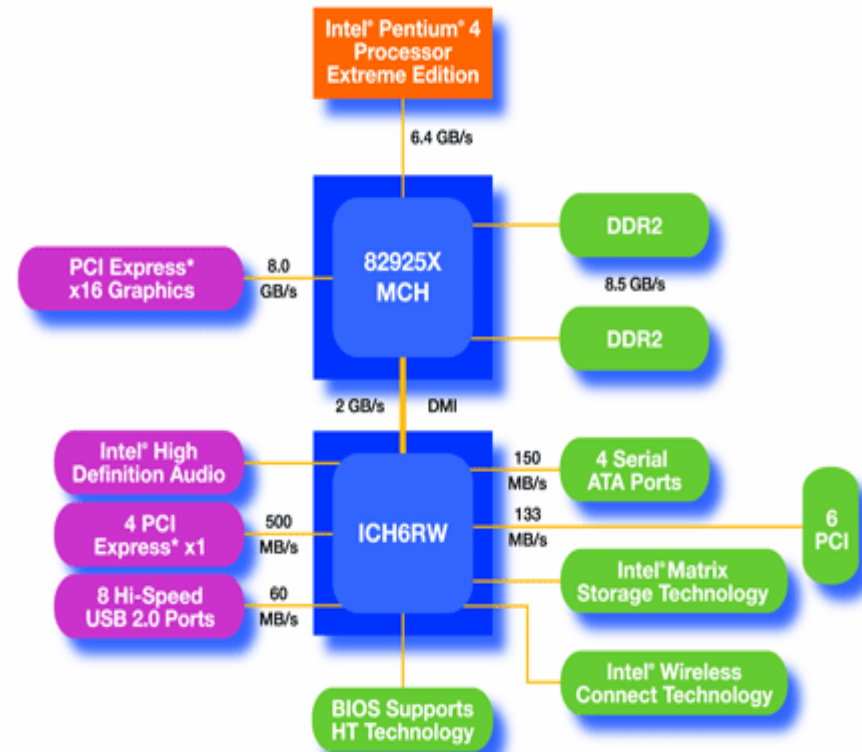
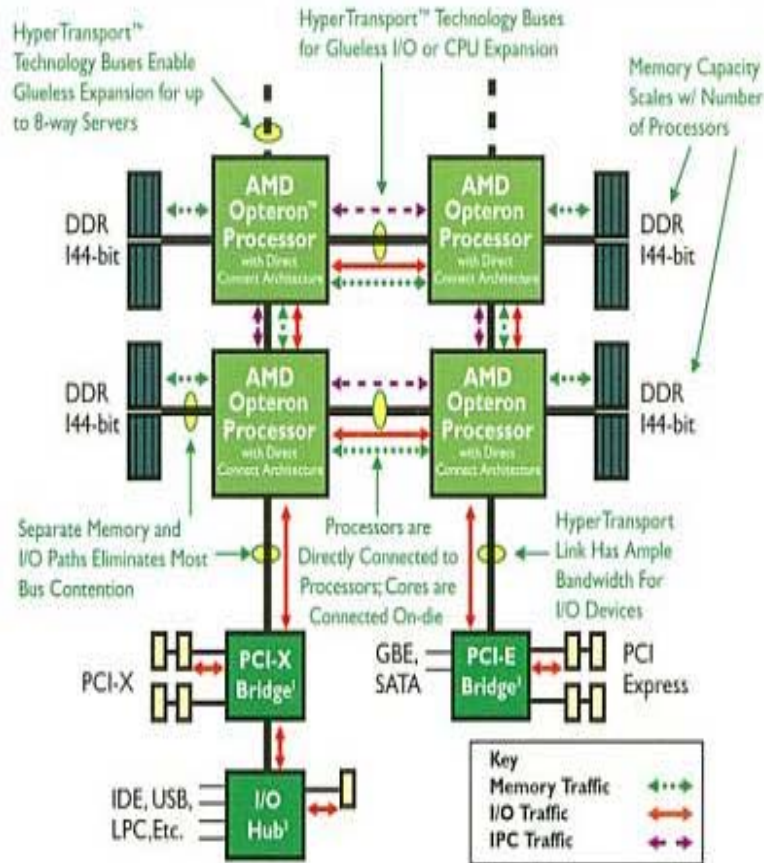
Specific parameters not exposed

The CUDA Platform

- The G80 GPU is not integrated into the CPU
 - Connects through a PCI Express bus
 - Communicates through OS (drivers)

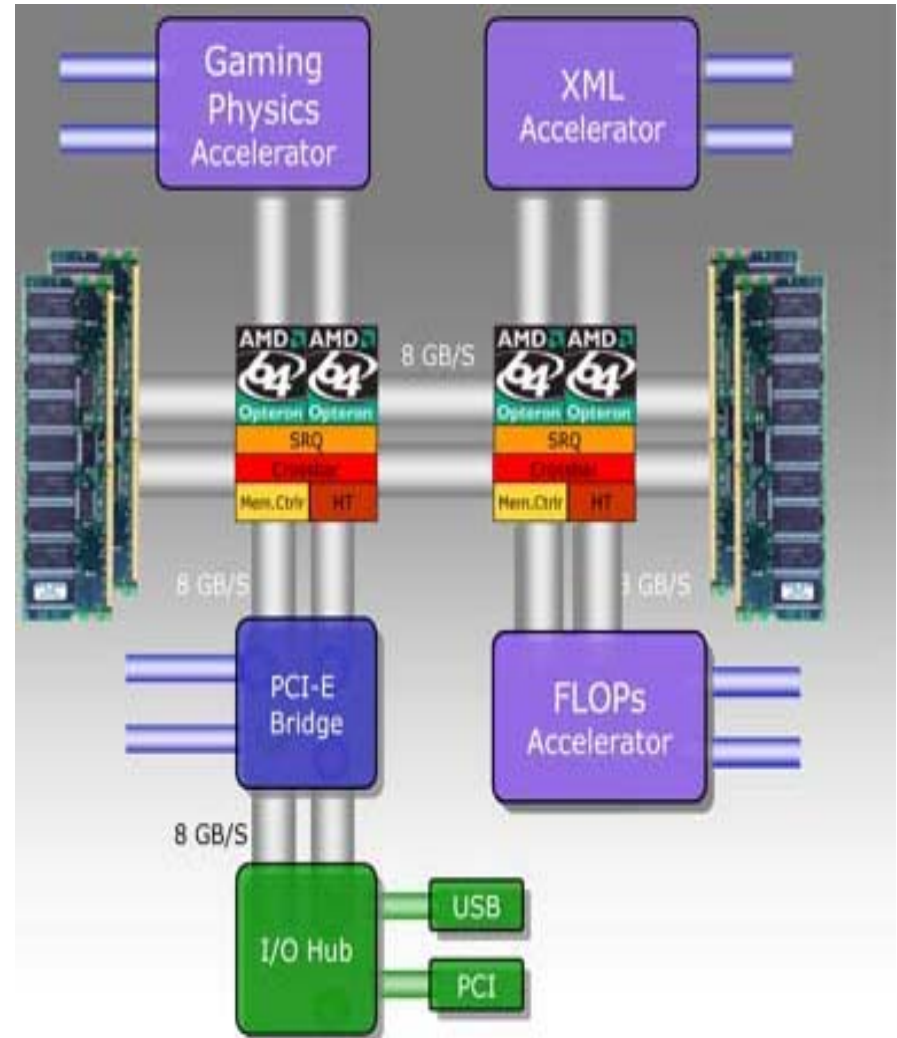


AMD Opteron™ Processor-based 4P Server



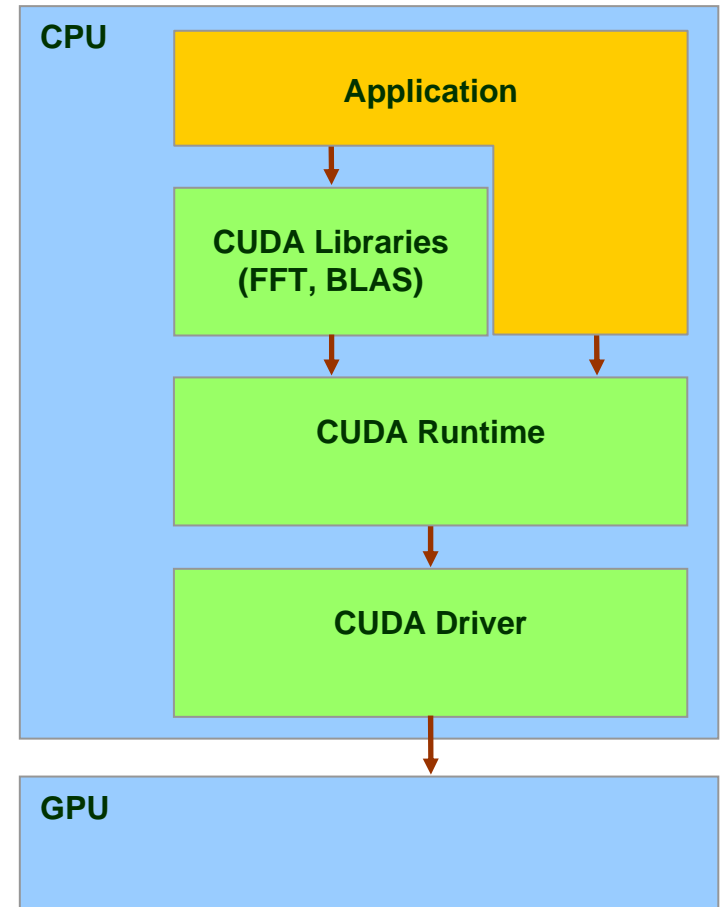
Better in the Near Future

- AMD Torrenza technology
 - Allows licensing of coherent HyperTransport™ to 3rd party manufacturers to make socket-compatible accelerators/co-processors
 - Allows 3rd party PPU (Physics Processing Unit), GPUs, and co-processors to access main system memory directly and coherently
 - Eliminate the “driver model”?



CUDA Programming System

- Targeted software stack
 - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
 - Standalone Driver - Optimized for computation
 - Interface designed for compute - graphics free API
 - Data sharing with OpenGL buffer objects
 - Guaranteed maximum download & readback speeds
 - Explicit GPU memory management



CUDA API and Language: Easy and Lightweight

- The API is an extension to the ANSI C programming language
 - Low learning curve
- The hardware is designed to enable lightweight runtime and driver
 - High performance