CUDA OPTIMIZATION, PART 2

NVIDIA Corporation

OUTLINE

- Architecture:
  - Kepler/Maxwell/Pascal/Volta
- Kernel optimizations
  - Launch configuration
- Part 2 (this session):
  - Global memory throughput
  - Shared memory access

Most concepts in this presentation apply to *any* language or API on NVIDIA GPUs

GLOBAL MEMORY THROUGHPUT

MEMORY HIERARCHY REVIEW

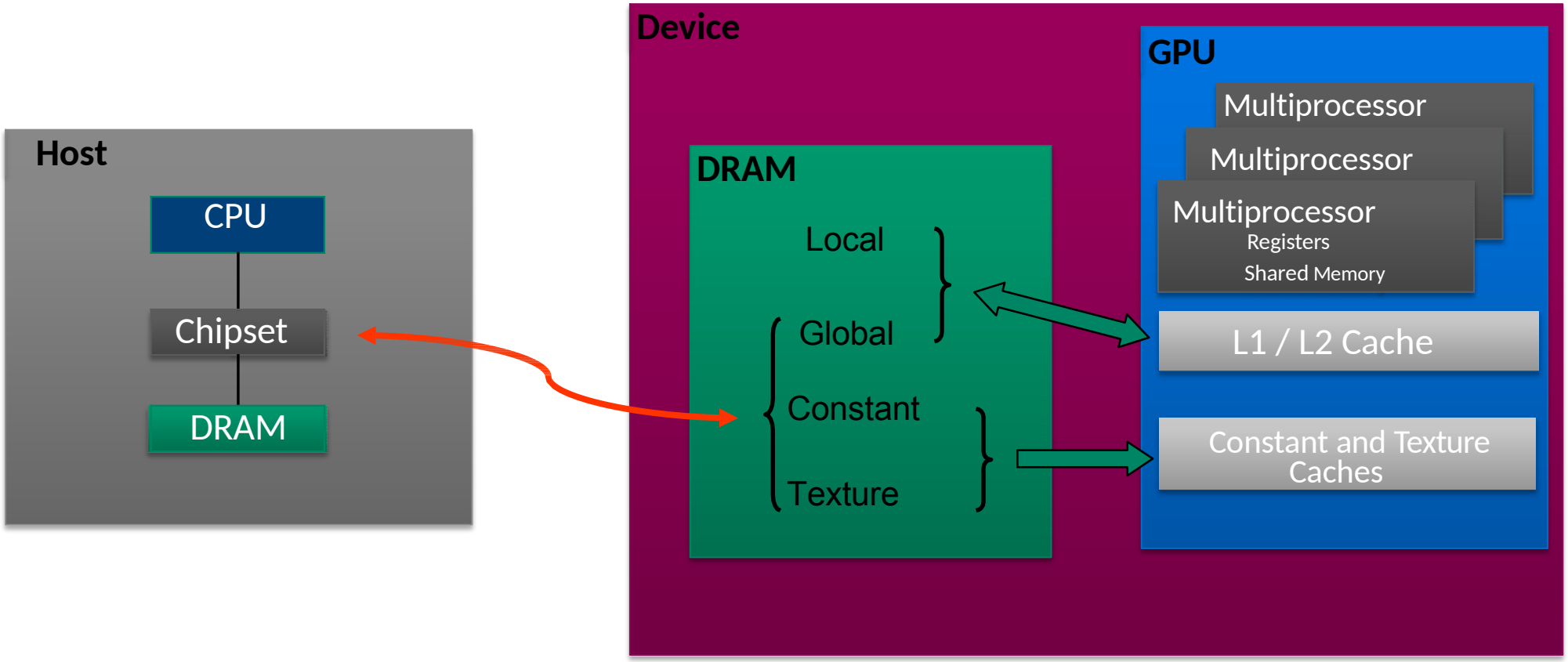
- Local storage
  - Each thread has own local storage
  - Typically off-node DRAM and registers (managed by the compiler)
- Shared memory / L1
  - Program configurable: typically up to 48KB shared (or 64KB, or 96KB…)
  - Shared memory is accessible by threads in the same threadblock
  - Very low latency
  - Very high throughput: >1 TB/s aggregate

MEMORY HIERARCHY REVIEW

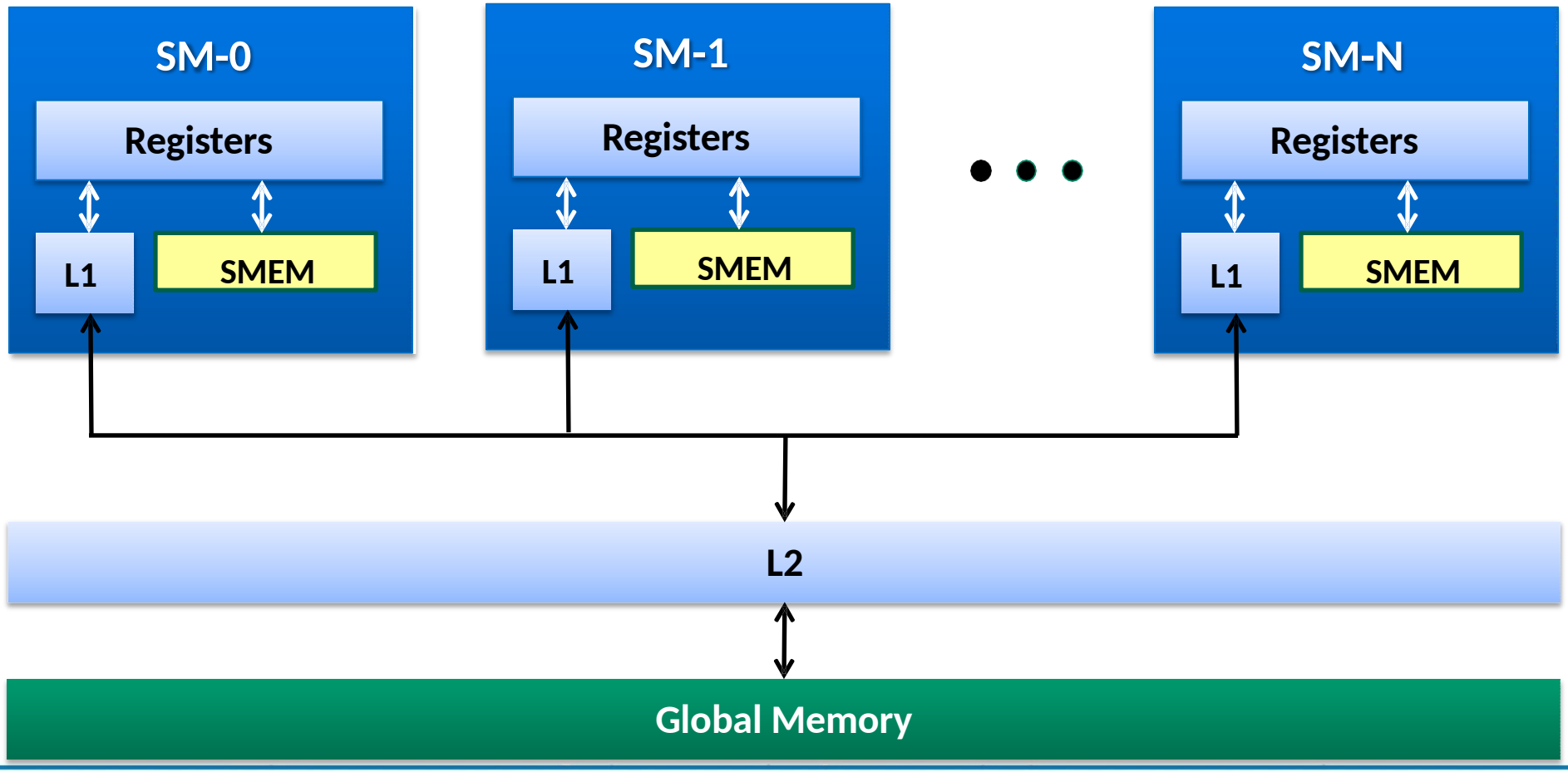
- L2
  - All accesses to global memory go through L2, including copies to/from CPU host Global

- Global Memory
  - Accessible by all threads as well as host (CPU)
  - High latency (hundreds of cycles)
  - Throughput: up to ~900 GB/s (Volta V100)

MEMORY ARCHITECTURE



MEMORY HIERARCHY REVIEW



GPU MEM OPERATIONS

- Loads:
  - Caching
    - Default mode
    - Attempts to hit in L1, then L2, then GMEM
    - Load granularity is 128-byte line
- Stores:
  - Invalidate L1, write-back for L2

GPU MEM OPERATIONS

- Loads:
  - Non-caching
    - Compile with –Xptxas –dlcm=cg option to nvcc
    - Attempts to hit in L2, then GMEM

      Do not hit in L1, invalidate the line if it's in L1 already

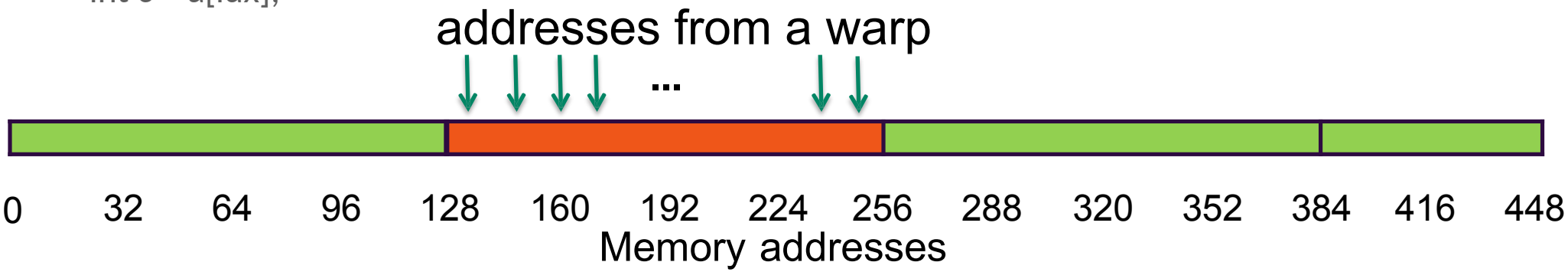
      The L1 can be bypassed with a non-caching load.
    - Load granularity is 32-bytes (segment)

LOAD OPERATION

- Memory operations are issued per warp (32 CUDA threads, a SIMD thread)
  - Just like all other instructions
- Operation:
  - CUDA Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments

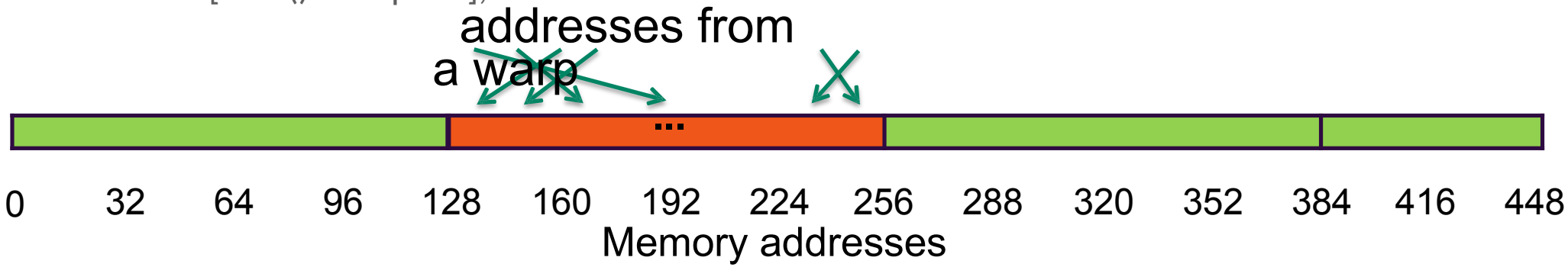
CACHING LOAD

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss (data load granularity)
  - Bus utilization: 100%
  - int c = a[idx];



CACHING LOAD

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - int c = a[rand()%warpSize];

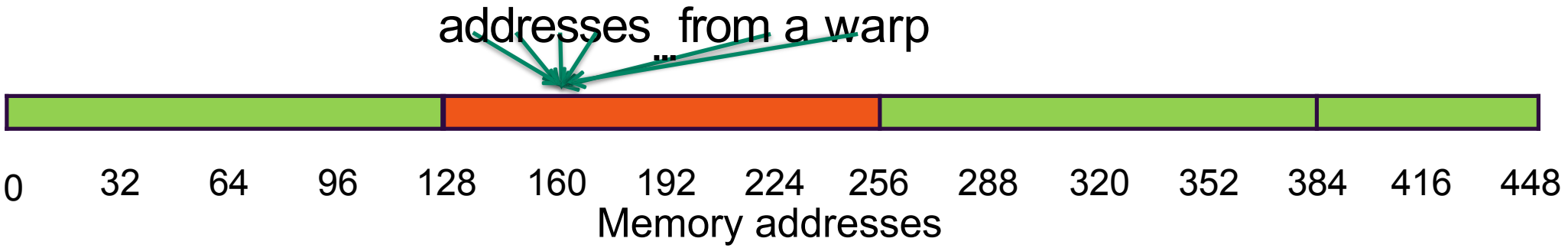


CACHING LOAD

- Warp requests 32 misaligned, consecutive 4-byte words, e.g. bytes[126, 253]
- Addresses fall within 2 cache-lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%
  - int c = a[idx-2];

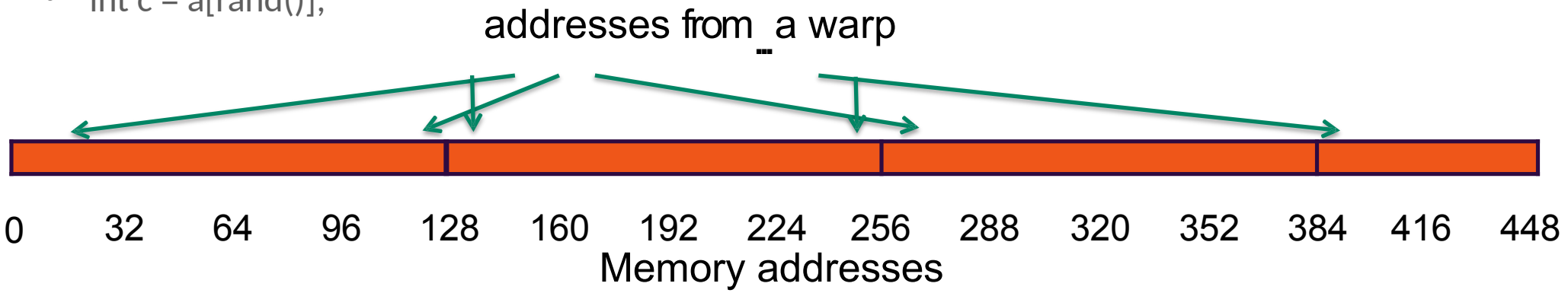
CACHING LOAD

- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125% (4/128), load 128 bytes, only using 4 bytes
  - int c = a[40];



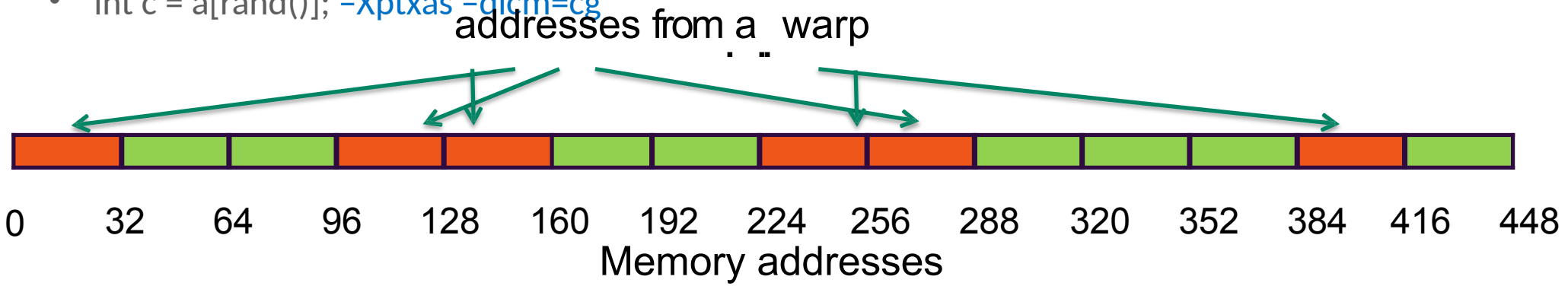
CACHING LOAD

- Warp requests 32 scattered 4-byte words
- Addresses fall within N cache-lines
  - Warp needs 128 bytes
  - N*128 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*128) (3.125% worst case N=32)
  - int c = a[rand()];



NON-CACHING LOAD

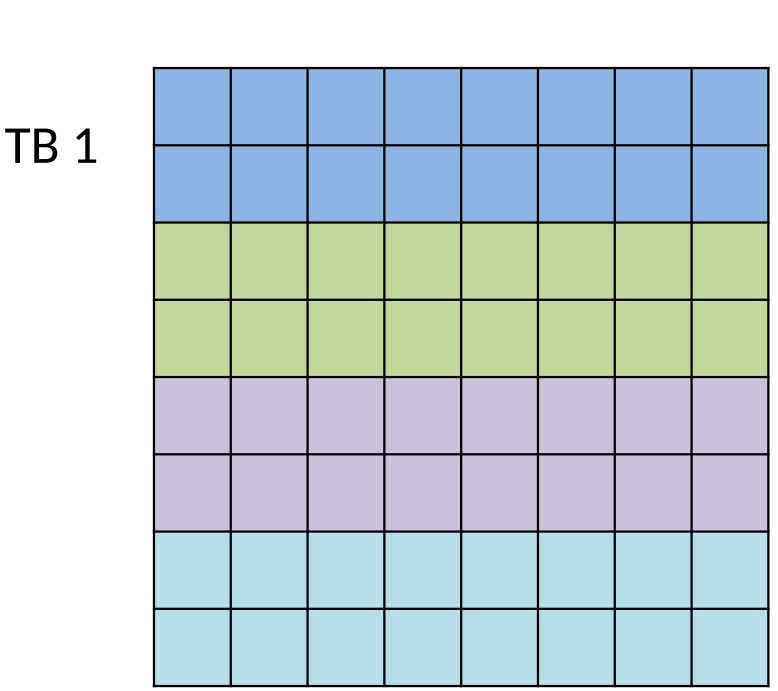
- Warp requests 32 scattered 4-byte words
- Addresses fall within N segments
  - Warp needs 128 bytes
  - N*32 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*32) (12.5% worst case N = 32)
  - int c = a[rand()]; –Xptxas –dlcm=cg



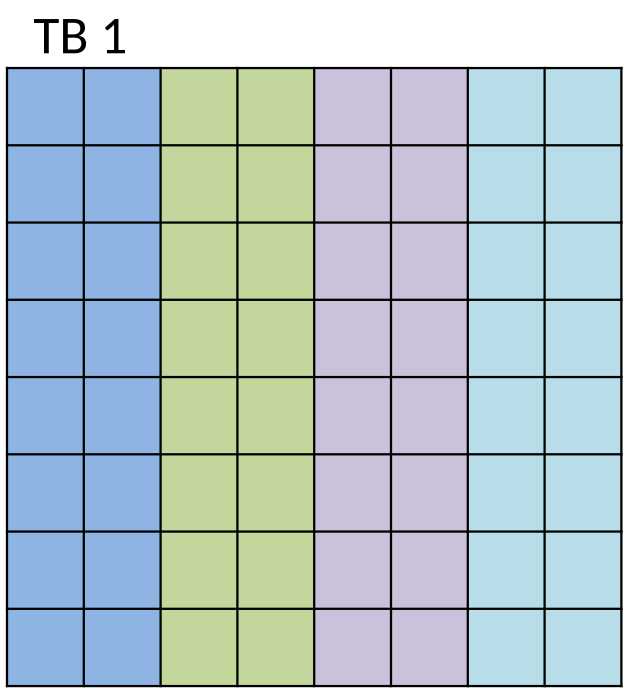
GPU MEM OPTIMIZATION GUIDELINES

- Strive for perfect coalescing
  - (Align starting address - may require padding)
  - A warp should access within a contiguous region
- Have enough concurrent accesses to saturate the bus
  - Process several elements per thread
    - Multiple loads get pipelined
    - Indexing calculations can often be reused
  - Launch enough warps to maximize throughput
    - Latency is hidden by switching warps
- Use all the caches!

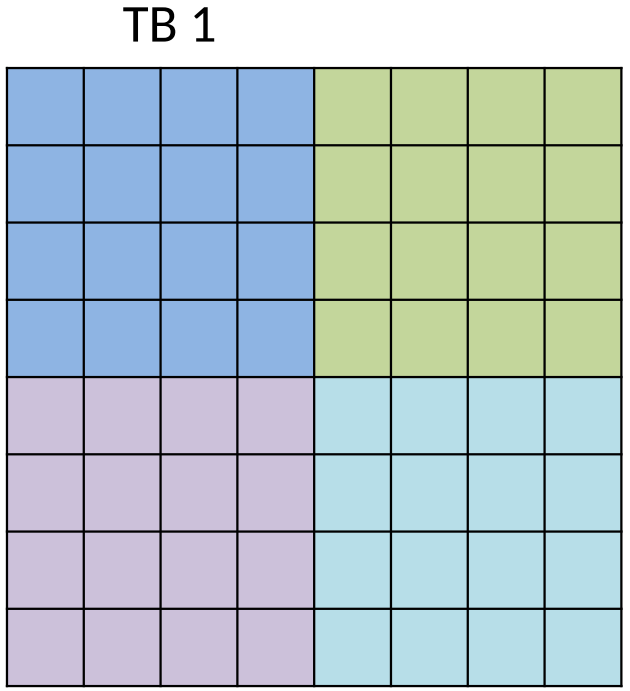
A x B = C, split C into thread blocks



TB 1: A[row 0-1] B[col 0-7]



TB 1: A[row 0-7] B[col 0-1]



TB 1: A[row 0-3] B[col 0-3]

SHARED MEMORY

SHARED MEMORY

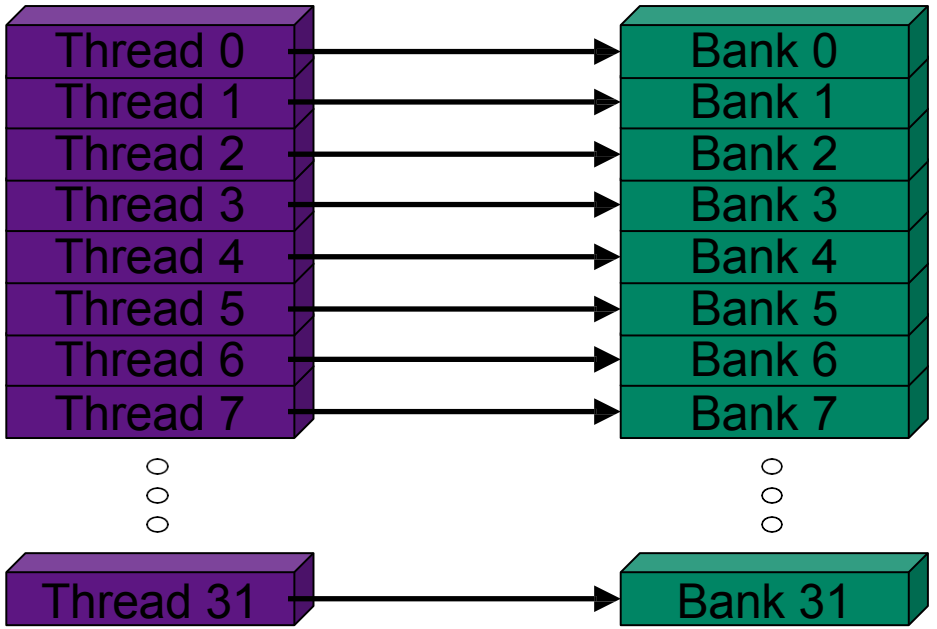
- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses , like CPU cache
  - Use it to improve global memory access patterns
  - The shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache)
  - Can be managed (allocate and free) via programming API
- Organization:
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks

SHARED MEMORY

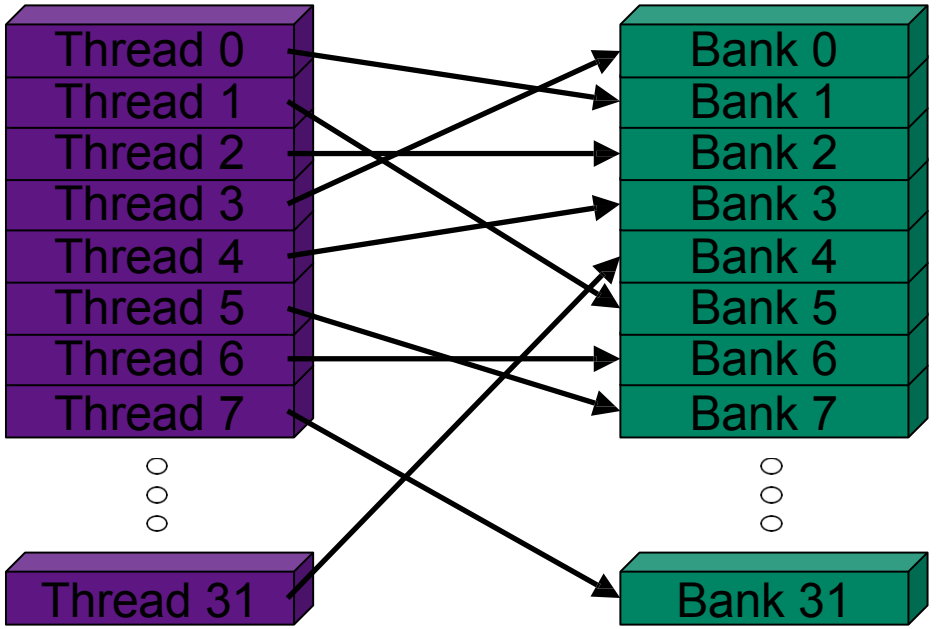
- Performance:
  - Typically: 4 bytes per bank per 1 or 2 clocks per multiprocessor
  - shared accesses are issued per 32 threads (warp)
  - serialization: if *N* threads of 32 access different 4-byte words in the same bank (bank conflict), *N* accesses are executed serially

BANK ADDRESSING EXAMPLES

No Bank Conflicts

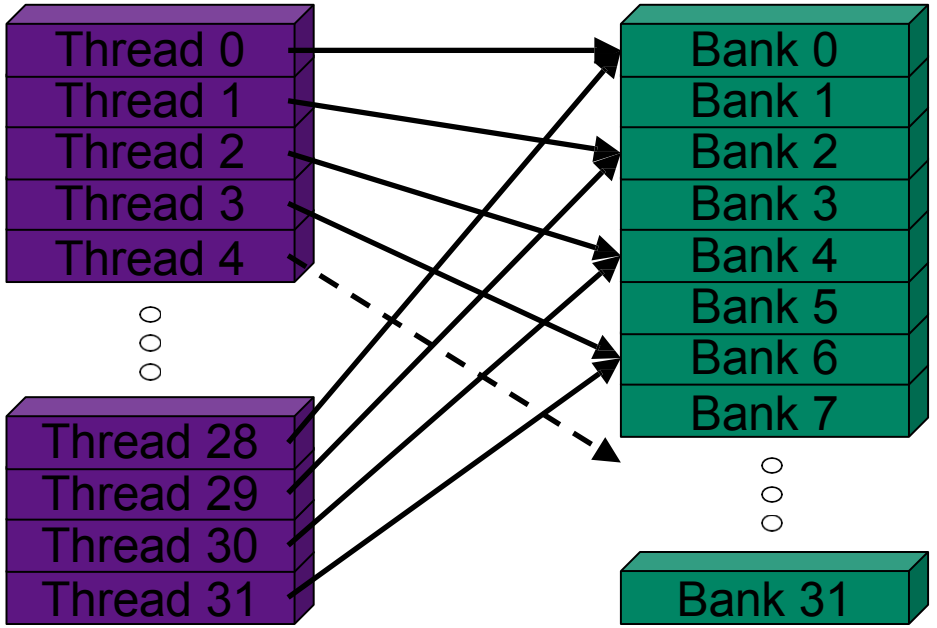


No Bank Conflicts

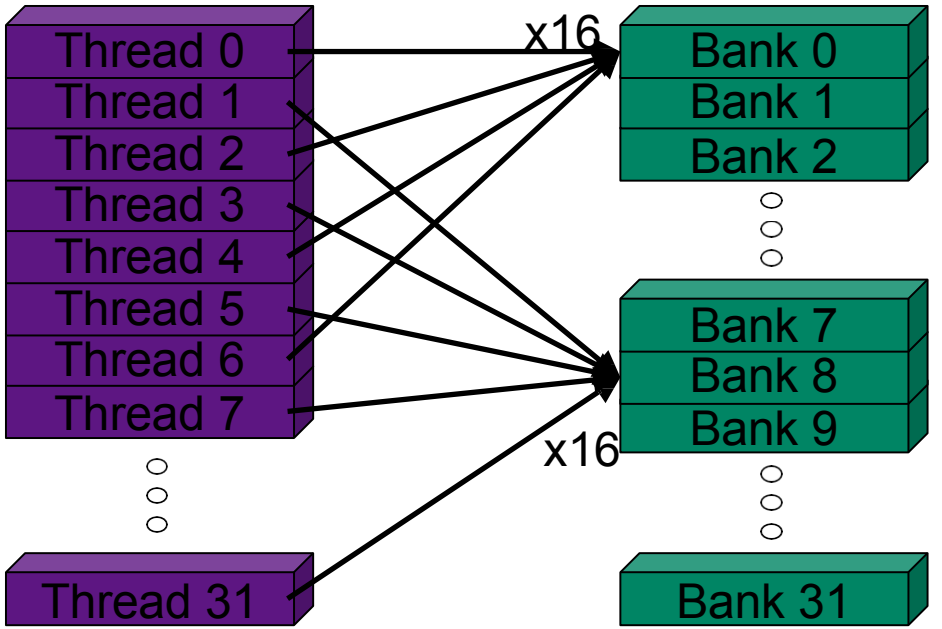


BANK ADDRESSING EXAMPLES

2-way Bank Conflicts



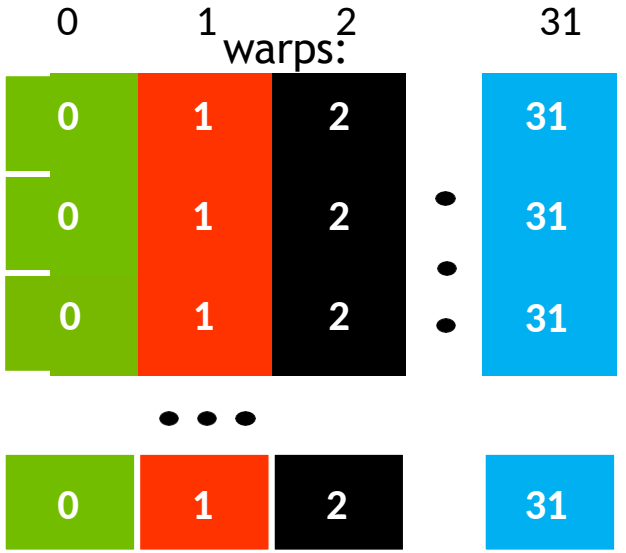
16-way Bank Conflicts



SHARED MEMORY: AVOIDING BANK CONFLICTS

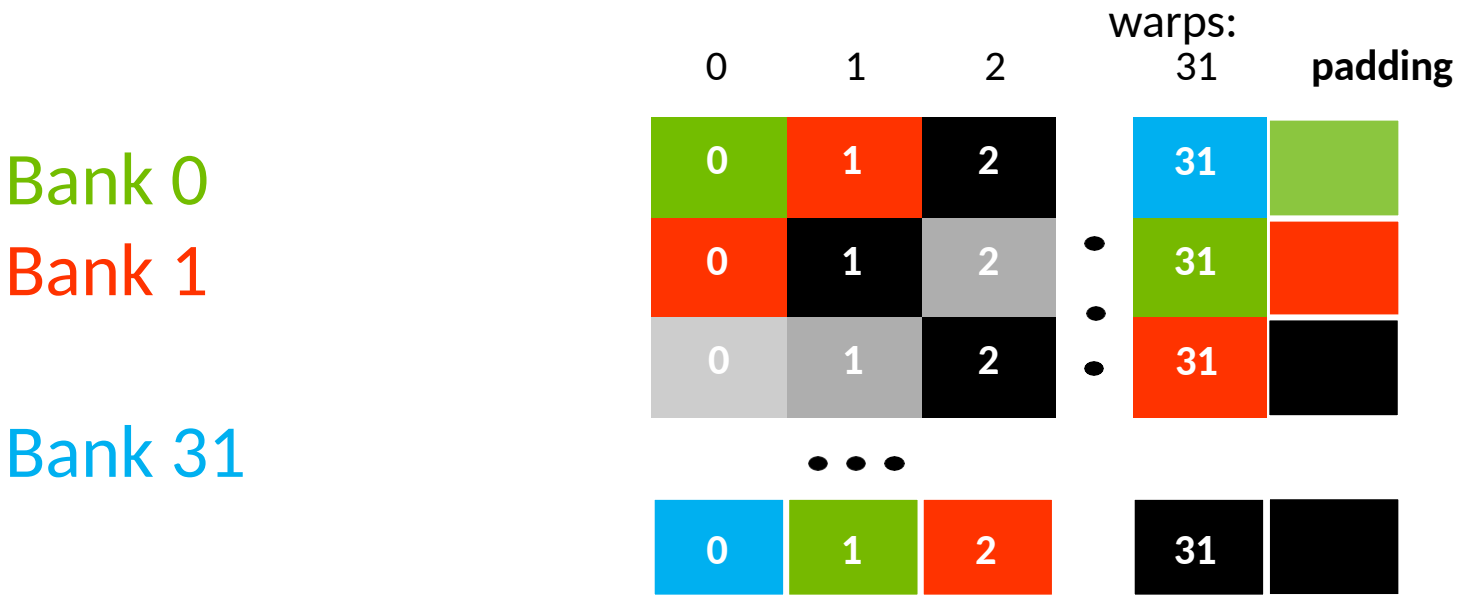
- 32x32 Shared MEM array
- Warp accesses a **column of 32x32 matrix (stored in row-major)**:
  - 32-way bank conflicts (threads in a warp access the same bank)

Bank 0
Bank 1
…
Bank 31



SHARED MEMORY: AVOIDING BANK CONFLICTS

- Add a column for padding:
- 32x33 SMEM array
  - Warp accesses a **column of 32x33 matrix with 1 padding column (stored in row-major)**:
    - 32 different banks, no bank conflicts



SUMMARY

- Kernel Launch Configuration:
  - Launch enough threads per SIMD Processor to hide latency
  - Launch enough threadblocks to load the GPU
- Global memory:
  - Maximize throughput (GPU has lots of bandwidth, use it effectively)
- Use shared memory when applicable (over 1 TB/s bandwidth)
- Use analysis/profiling when optimizing:
  - “Analysis-driven Optimization” (future session)

FUTURE SESSIONS

- Atomics, Reductions, Warp Shuffle
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)
- Analysis Driven Optimization
- Cooperative Groups

FURTHER STUDY

- Optimization in-depth:
  - http://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf
- Analysis-Driven Optimization:
  - http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf
- CUDA Best Practices Guide:
  - https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
- CUDA Tuning Guides:
  - https://docs.nvidia.com/cuda/index.html#programming-guides (Kepler/Maxwell/Pascal/Volta)