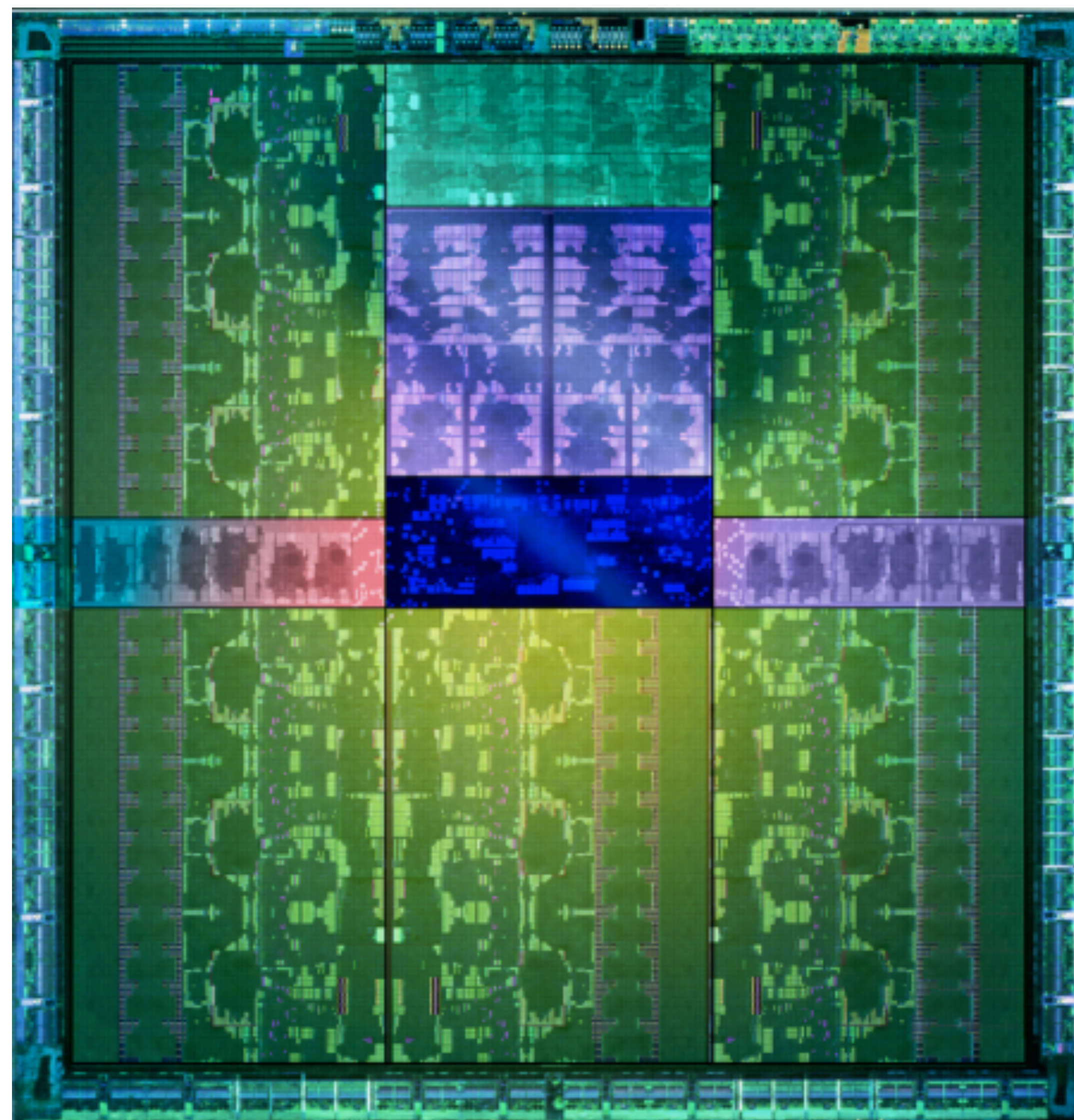
GPU COMPUTING
LECTURE 03 - BASIC ARCHITECTURE

Kazem Shekofteh
kazem.shekofteh@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

GK110 - ARCHITECTURE

Up to 15 SMX, 6 MCs, L2 cache, PCIe 3.0, CC 3.5



PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
SMX
L2 Cache

GK110 - ARCHITECTURE

192 SP units

64 DP units

32 load/store units

32 special function units

4 warp schedulers

Optimized for performance/watt

-> reduced clock frequency

Pollack's law: $S \propto \sqrt[2]{complexity}$

BULK-SYNCHRONOUS PARALLEL

REMINDER: BULK-SYNCHRONOUS PARALLEL

In 1990, Valiant already described GPU computing pretty well

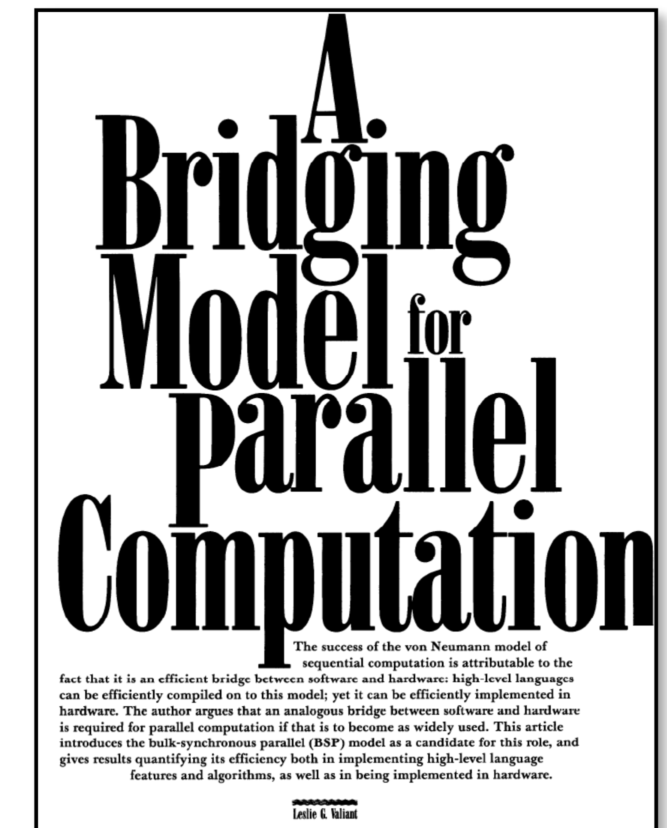
Superstep

- Compute, communicate, synchronize

Parallel slackness: # of virtual processors v, physical processors p

- v = 1: not viable
- v = p: unpromising wrt optimality
- v >> p: leverage slack to schedule and pipeline computation and communication efficiently

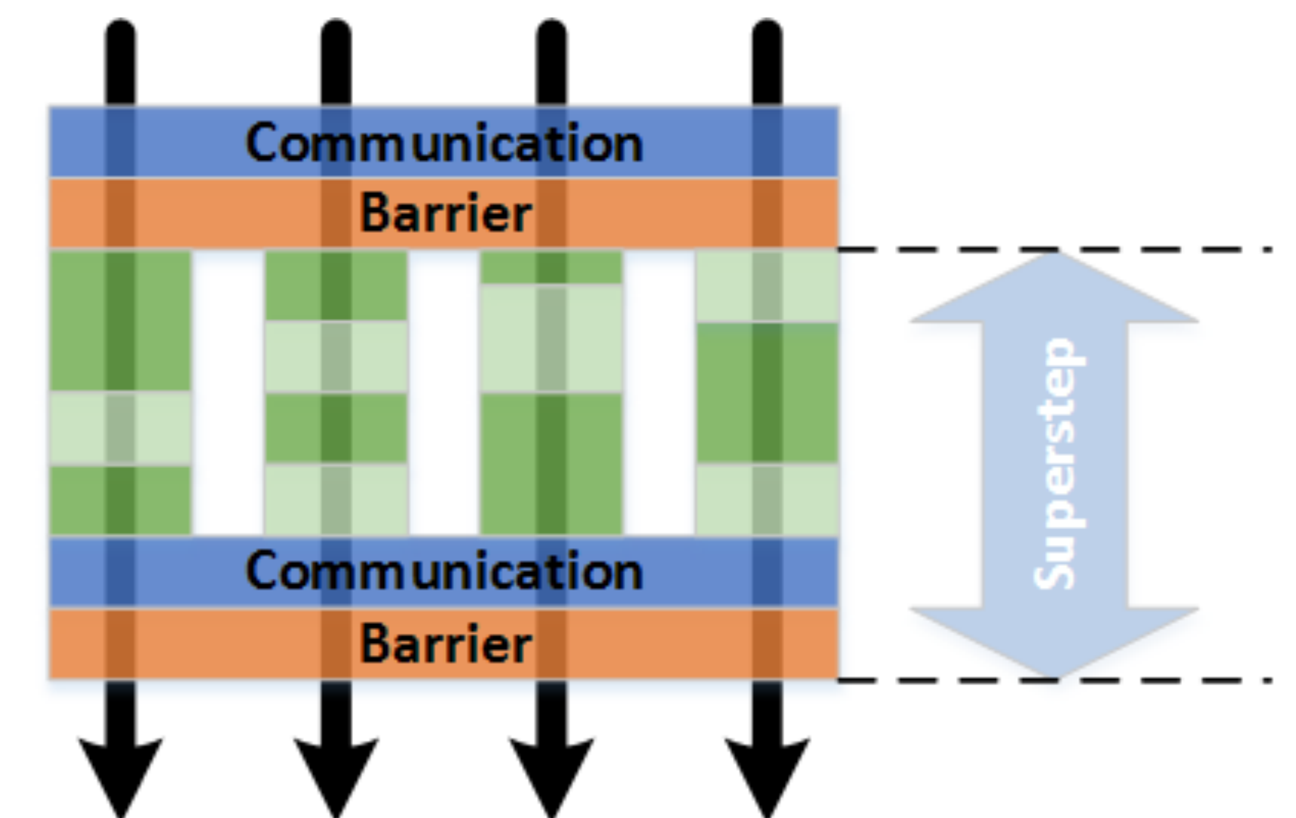
Extremely scalable, bad for unstructured parallelism

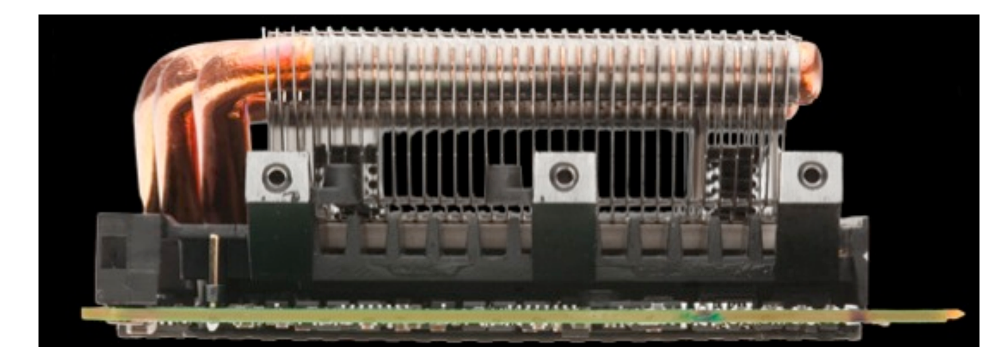

A Bridging Model for Parallel Computation

The success of the von Neumann model of sequential computation is attributable to the fact that it is an efficient bridge between software and hardware: high-level languages can be efficiently compiled on to this model; yet it can be efficiently implemented in hardware. The author argues that an analogous bridge between software and hardware is required for parallel computation if that is to become as widely used. This article introduces the bulk-synchronous parallel (BSP) model as a candidate for this role, and gives results quantifying its efficiency both in implementing high-level language features and algorithms, as well as in being implemented in hardware.

Leslie G. Valiant

Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990





REMINDER: VECTOR ISAS

Compact: single instruction defines N operations

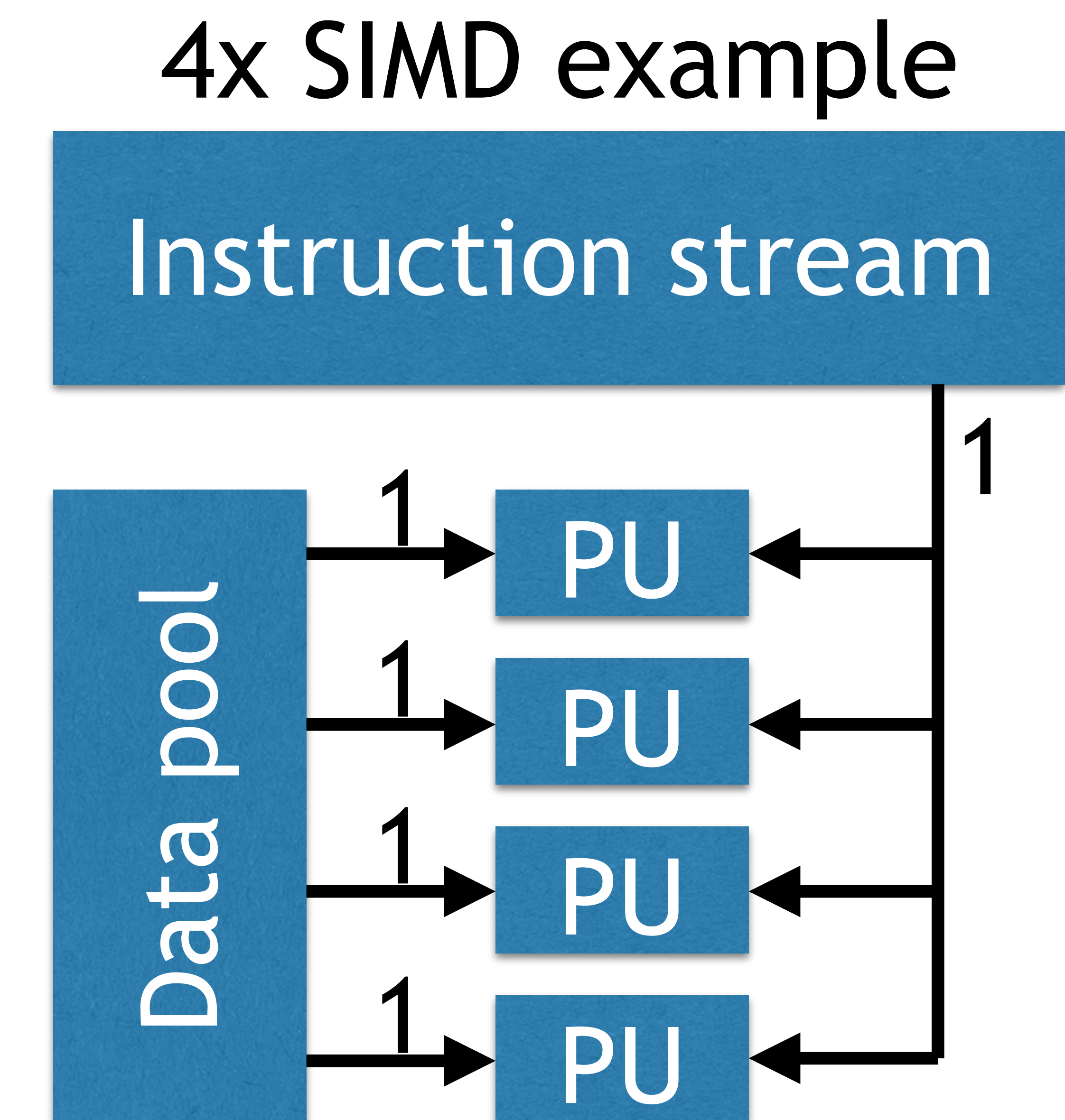
- Amortizes the cost of instruction fetch/decode/issue
- Also reduces the frequency of branches

Parallel: N operations are (data) parallel

- No dependencies
- No need for complex hardware to detect parallelism (similar to VLIW)
- Can execute in parallel assuming N parallel data paths

Expressive: memory operations describe patterns

- Continuous or regular memory access pattern
- Can prefetch or accelerate using wide/multi-banked memory
- Can amortize high latency for 1st element over large sequential pattern



OUR VIEW OF A GPU

Software view: a programmable many-core scalar architecture

- Huge amount of scalar threads to exploit parallel slackness, operates in lock-step
- SIMT: single instruction, multiple threads

<u>IT’S A (ALMOST) PERFECT INCARNATION OF THE BSP MODEL</u>

Hardware view: a programmable multi-core vector architecture

- Illusion of scalar threads: hardware packs them into compound units
- SIMD: single instruction, multiple data

<u>IT’S A VECTOR ARCHITECTURE THAT HIDES ITS VECTOR UNITS</u>

THE BEAUTY OF SIMPLICITY

GPU Computing & CUDA

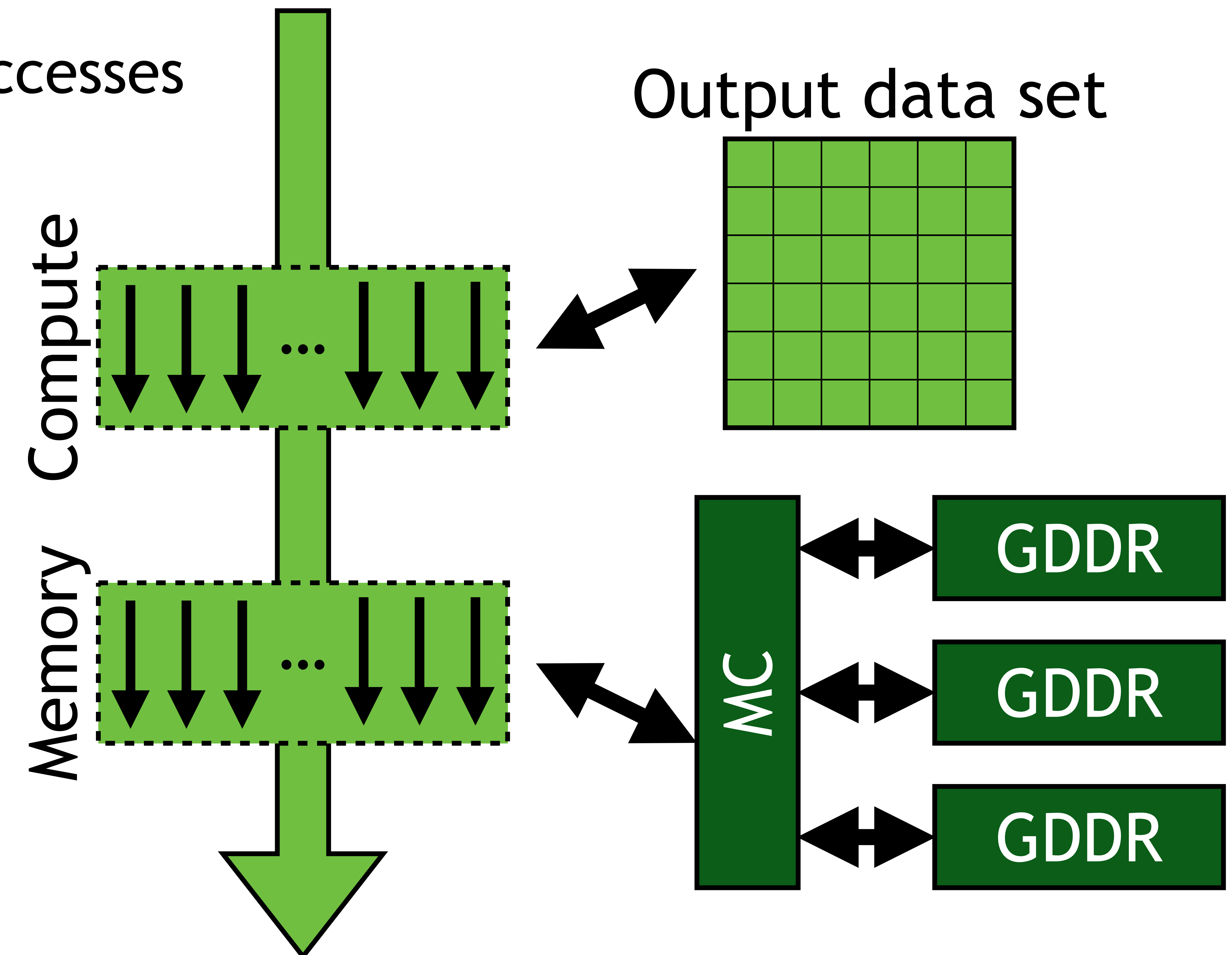
- Thread-collective computation and memory accesses
- SIMT – Single Instruction, Multiple Threads

GPU collaborative computing

- One thread per output element
- Schedulers exploit parallel slack

GPU collaborative memory access

- One thread per data element
- MCs highly optimized to exploit concurrency
- -> coalescing issues



-> If you do something on a GPU, do it collaboratively with all threads

(GLOBAL) MEMORY SUBSYSTEM

GK110 – MEMORY HIERARCHY

Registers at thread level

- Registers/thread depends on run-time configuration
- Max. 255 registers/thread

Shared memory / L1$ at block level

- Variable sizes
- L1$ can serve for register spilling
- L1$ not coherent, write-invalidate
- Compiler controlled RO L1$

L2$ / GDDR at device level

- GDDR: ~400-600 cycles access latency
- L2$ as victim cache for all upper units, write-back
- Purpose: reducing contention

Level	Memory		
Thread	**Registers** 64k/thread block		
Thread Block	Shared Memory 16-48kB	L1 Cache 16-48kB	Read-only data Cache 48kB
Multiple Kernels	**L2 Cache** 1.5MB		
	GDDR (off-chip) 6GB		
	Host memory (off-device) multiple TBs		

LOCAL MEMORY

Local memory: part of global memory, but thread-local

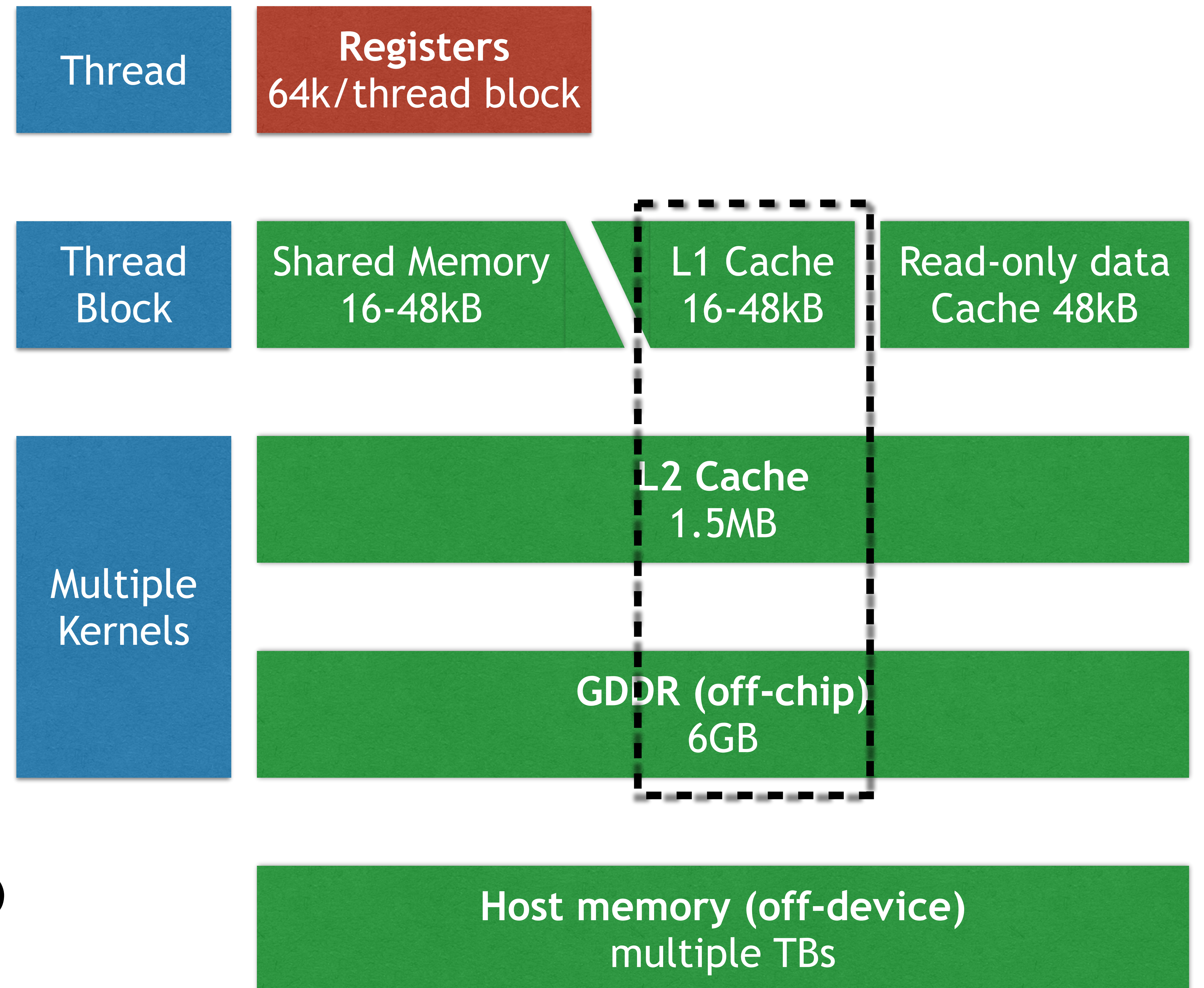
- Register spilling: when SM runs out of resources
- Limited register count per thread
- Limited total number of registers
- LM is used if the source code exceeds these limits
- Local because each thread has its private area

Differences from global memory

- Stores are cached in L1$
- Addressing is resolved by compiler

Store always happens before load

- Per thread: move data from GM to LM (stores)
- Subsequent load accesses



HOST MEMORY

Pinned/unpinned host memory

Unpinned host memory: possibility of demand paging -> staging buffers

Pinned host memory: autonomous device access possible

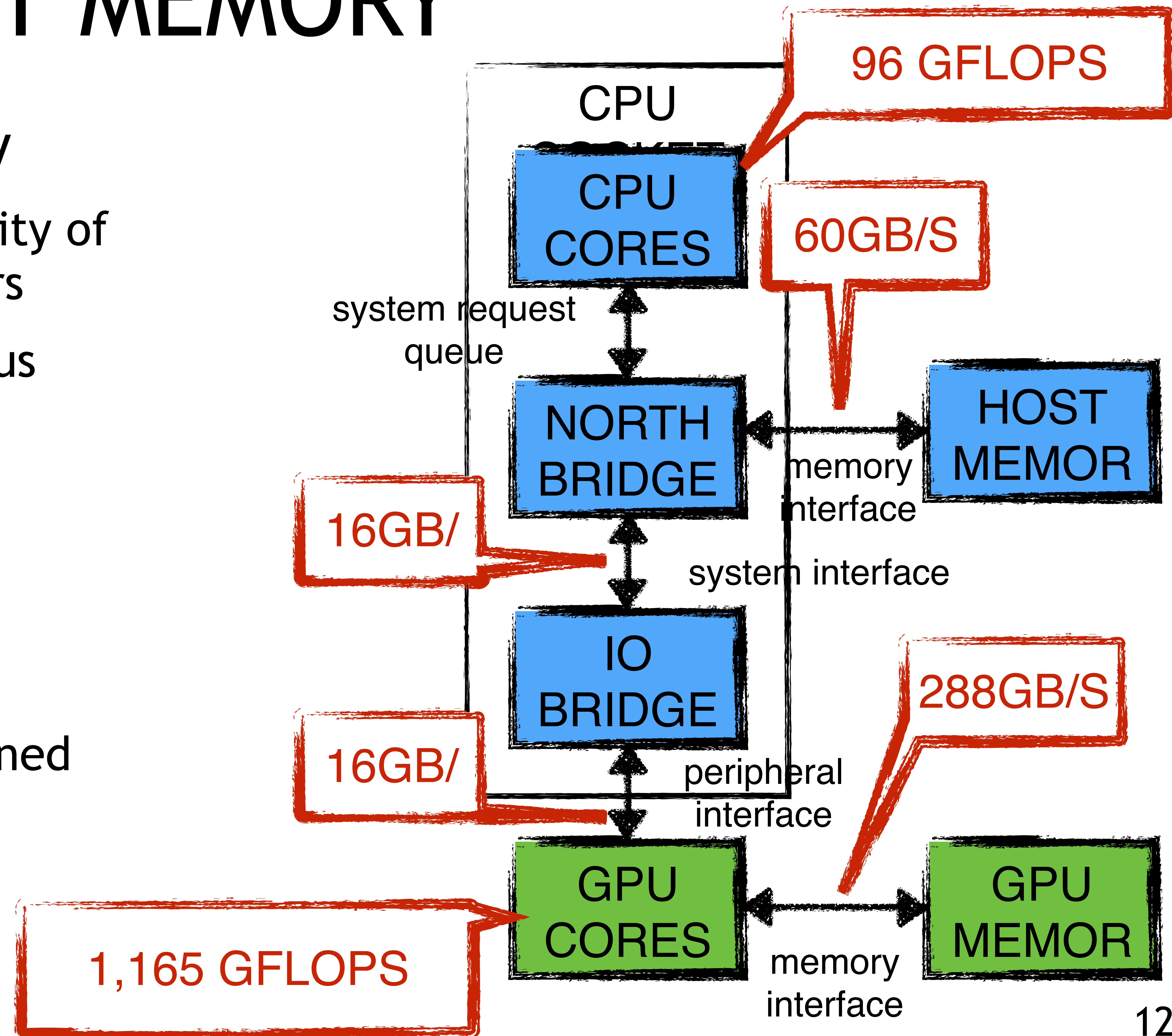
cudaMemcpy

GPU DMA engine(s)

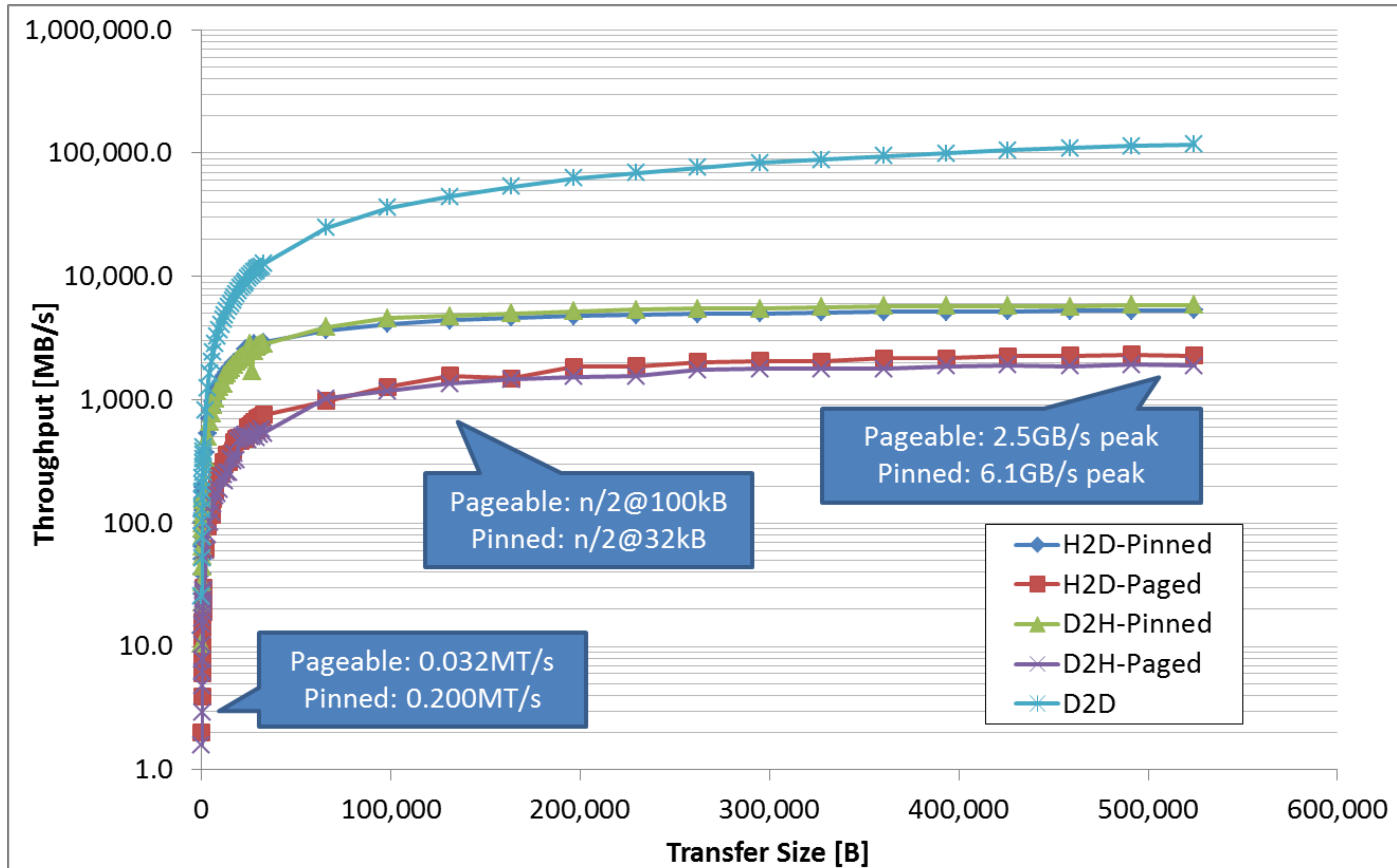
Zero copy (CC >= 2.0)

GPU threads can operate on pinned host memory

For initial shared memory fills, etc.



HOST MEMORY & CUDAMEMCPY



HOST MEMORY & STREAMS

Stream: sequence of operations performed in-order

- cudaMemcpy
- Kernel launch
- Default stream: id=0

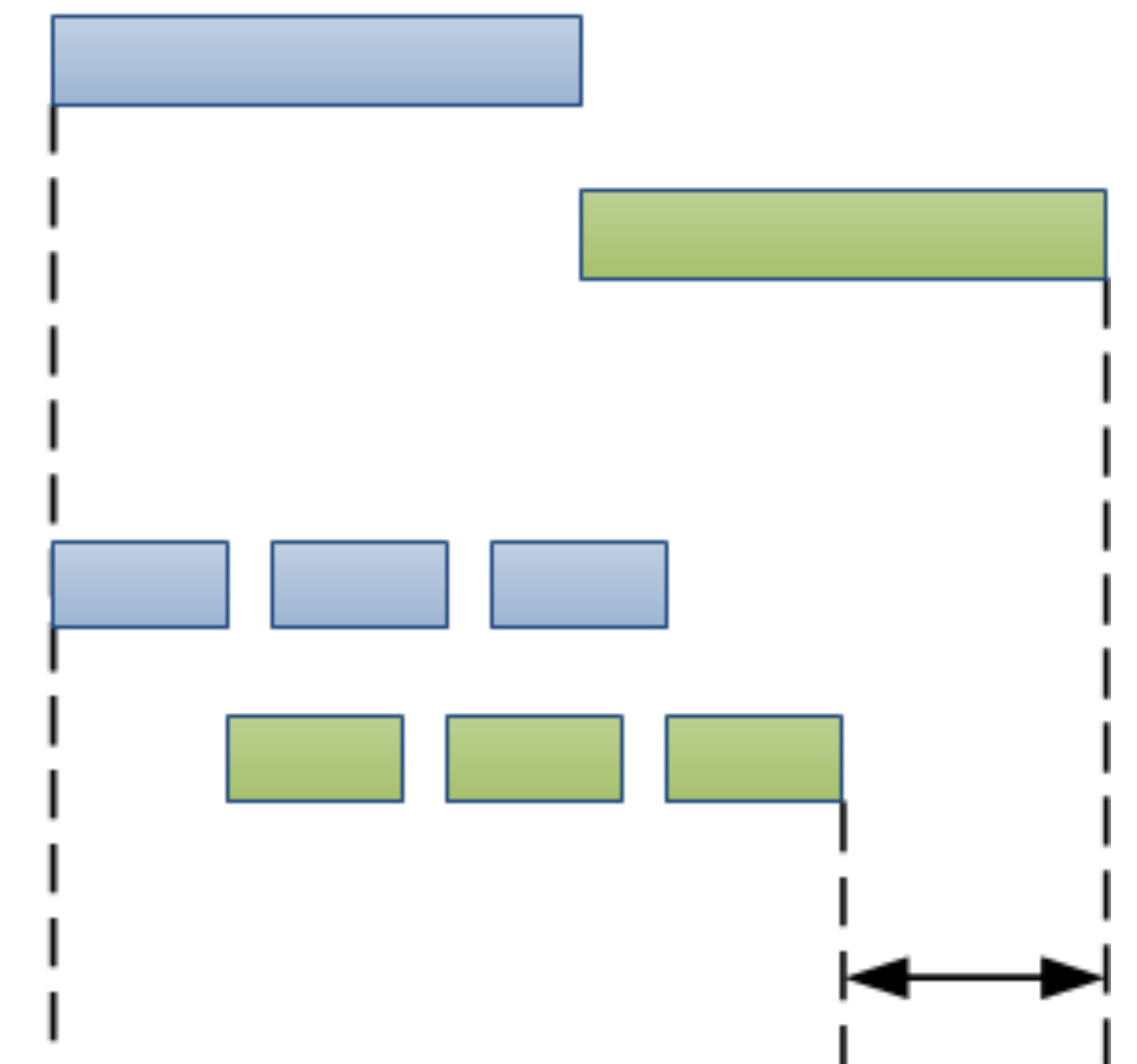
Overlap computation with data movement

- Latency hiding

Only applicable for divisible work

Most suited for compute-bound workloads

- See also zero-copy for initial data movements



GLOBAL MEMORY - COALESCING

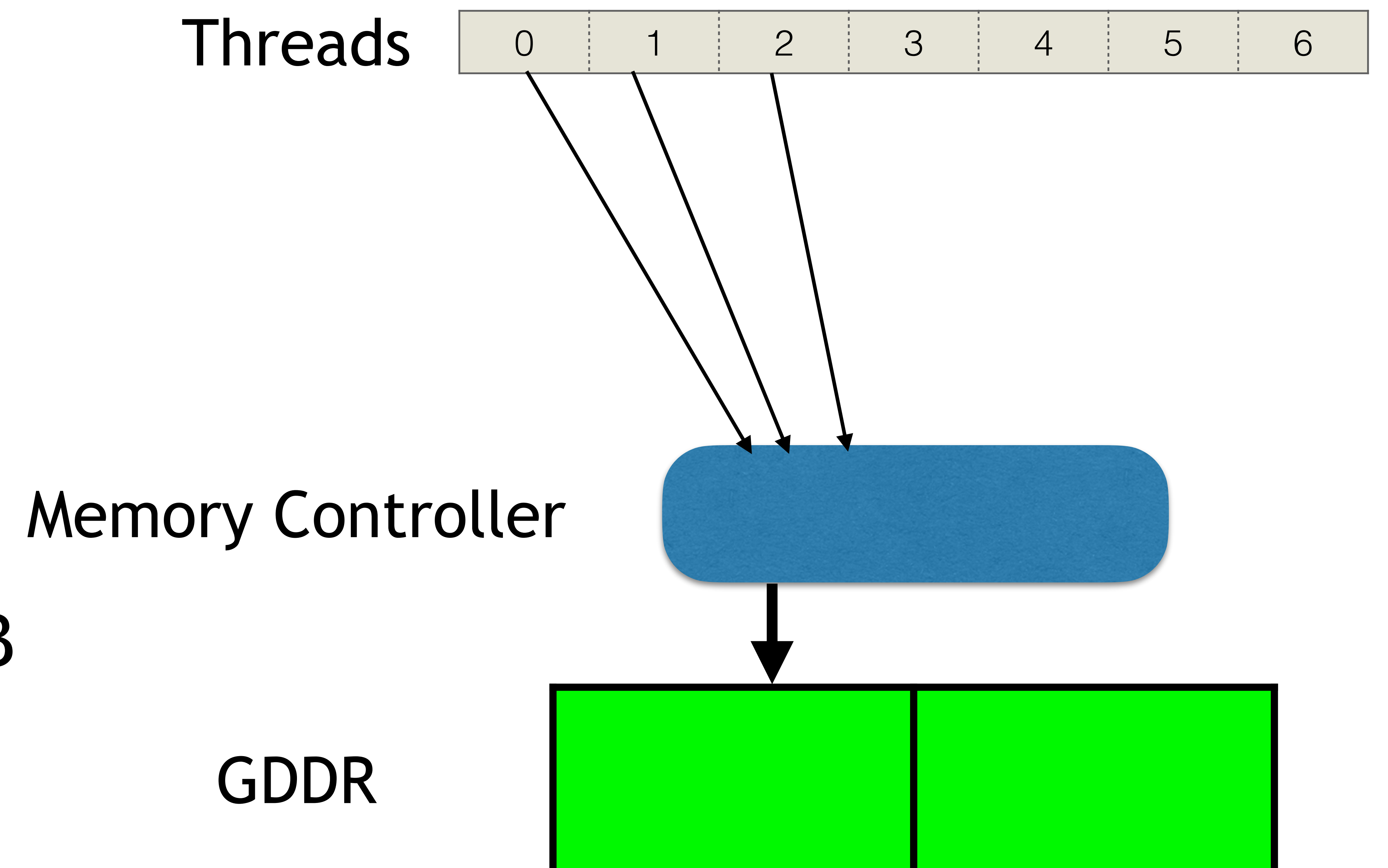
High bandwidth, high latency

Coalesced access

- Combine fine-grain accesses by multiple threads into single GDDR operations (such requests have a certain granularity)
- Coalesced thread access should match a multiple of L1/L2 cache line sizes
- For Kepler cache line sizes: L1: 128B, L2: 32B

Misaligned accesses

- One warp is scheduled, but accesses misaligned addresses
- GPUs use caches for access coalescing



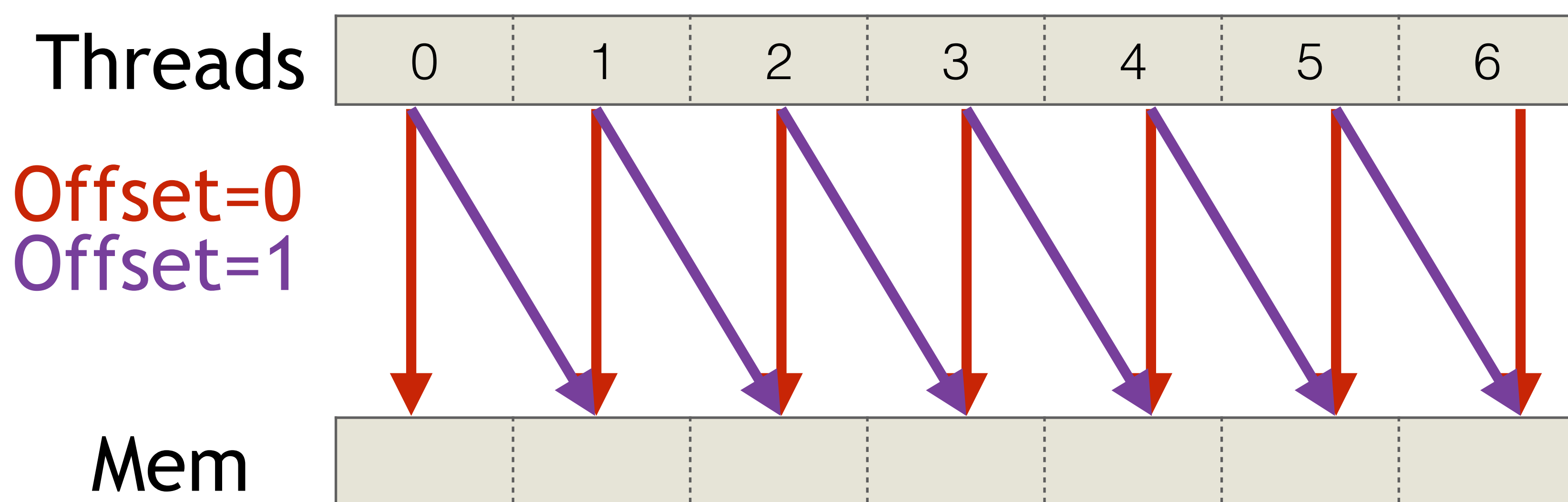
GLOBAL MEMORY – ACCESS PENALTIES

Offset: constant shift of access pattern

data[addr+offset]

Penalty: fetch 5 CLs instead of 4

4/5 of max. bandwidth



Copy with Offset (Tesla M2090 - ECC on)

Bandwidth (GB/s) vs. Offset (0–32); series: Caching, Non-caching

8 elements offset, 4B per element

NVIDIA, CUDA C Best Practices Guide

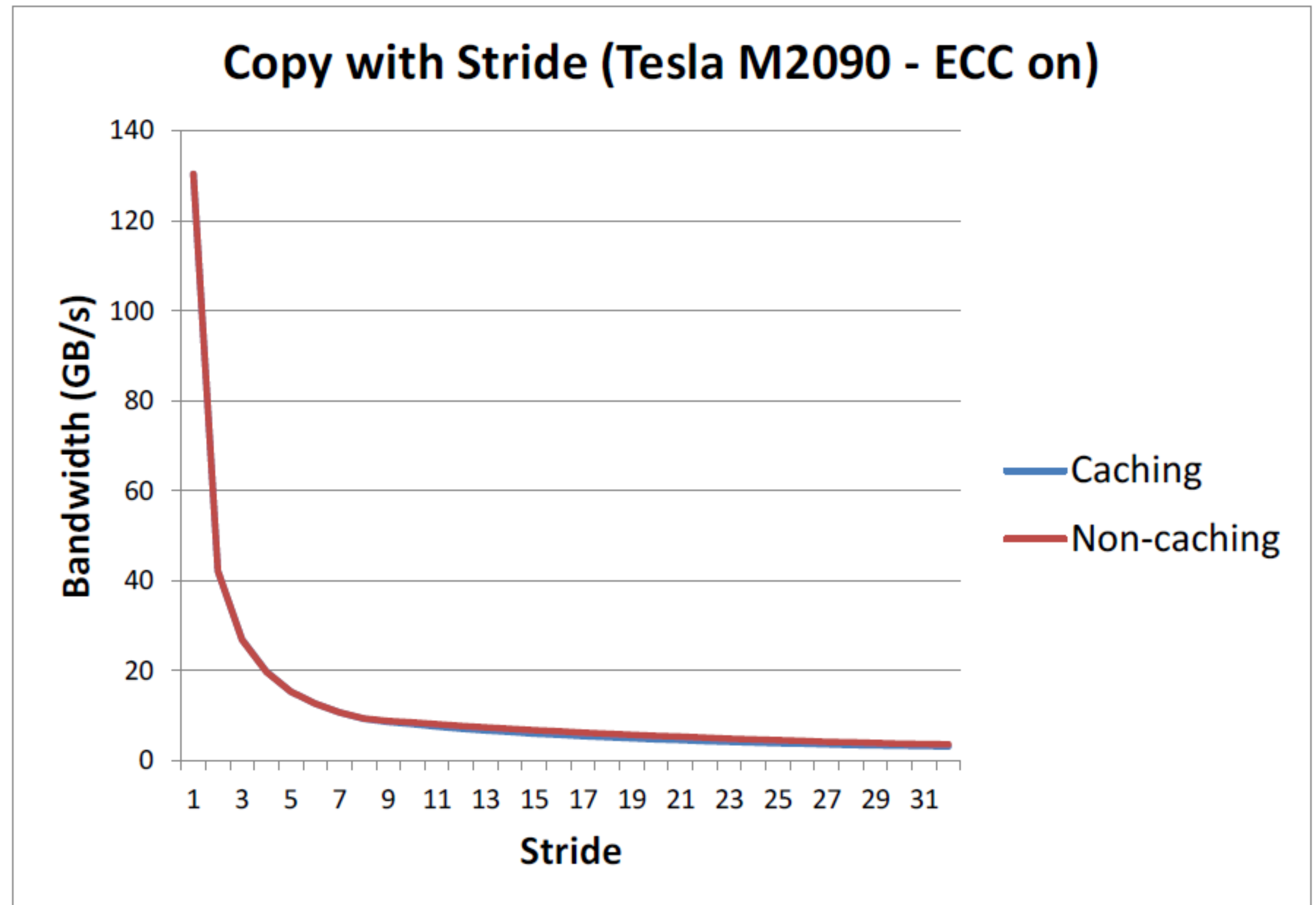
GLOBAL MEMORY – ACCESS PENALTIES

Stride: access only every nth address

- data[addr*stride]

Stride of 2

- 50% load/store efficiency
- Worsens with larger strides



NVIDIA, CUDA C Best Practices Guide

GLOBAL MEMORY – ACCESS PENALTIES

Main problem: thread scheduling does not result in coalesced accesses

Solution: manually control data movement in memory hierarchy

- Caches = transparent, implicit hierarchy
- Scratchpad (shared memory) = opaque, explicit hierarchy

Collaborative loads from global memory to shared memory

- Common case: one thread is not moving the data it requires (at least not immediately)
- One of the GPU’s main advantages is memory bandwidth: coalescing of upmost importance!

CUDA THREAD SCHEDULING

Foundation of latency tolerance

LATENCY TOLERANCE TECHNIQUES

Property	Relaxed Consistency Models	Prefetching	Multi-Threading	Block Data Transfer
Types of latency tolerated	Write (blocking read processors) Read and write (dynamically scheduled processors)	Write Read	Write Read Synchronization	Write Read
Software requirements	Labeling synchronization operations	Predictability	Explicit extra concurrency	Identifying and orchestrating block transfers
Extra hardware support	Little	Little	Substantial	Not in processor, but in memory system
Supported in commercial systems?	Yes	Yes	Yes	(Yes)

David E. Culler, Jaswinder Pal Singh, Anoop Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, 1998

THREAD SCHEDULING

Up to 1k threads per block

- One block executes on one SM
- Kepler: one SM = 192 SP + 64 DP units

Each thread block is divided in warps of 32 threads

- Implementation decision, not CUDA

Warps are the units for the scheduler

Example

- 4 blocks being executed on one SM, each block 1k threads
- How many warps?

Scheduler

1. Select one thread block to execute, allocate resources (registers, etc) as required
2. Select one out of the 32 warps of this block for instruction fetch and execution
3. Repeat until all resources are utilized
4. Upon warp stalling, select another warp for IF and EX
5. Deallocate resources after all warps have finished (non-preemptive)

THREAD SCHEDULING

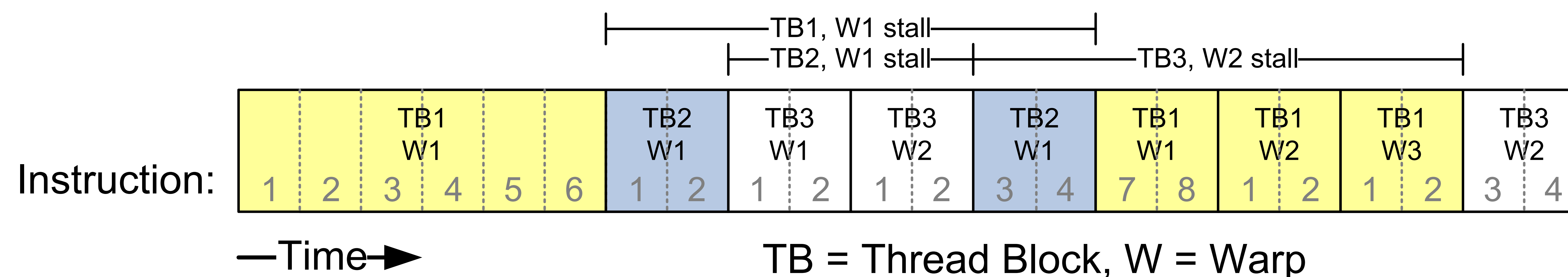
Fine-grained multi-threading (FGMT)

- Switch context (i.e., warp) every cycle
- A warp that has the operands ready for its next instruction is ready for execution
- All threads in a warp execute the same instruction

Goal of FGMT: latency hiding

- Global memory access latency: ~400-600 cycles
- Sufficient number of warps can keep all functional units busy

Warp count for maximum utilization depends on computational intensity



EXAMPLE FOR HARDWARE MULTI-THREADING (G80)

4 warp contexts, max. 1 being executed simultaneously

Explicit 32x SIMD instructions

- 32 ALUs execute a single SIMD instruction

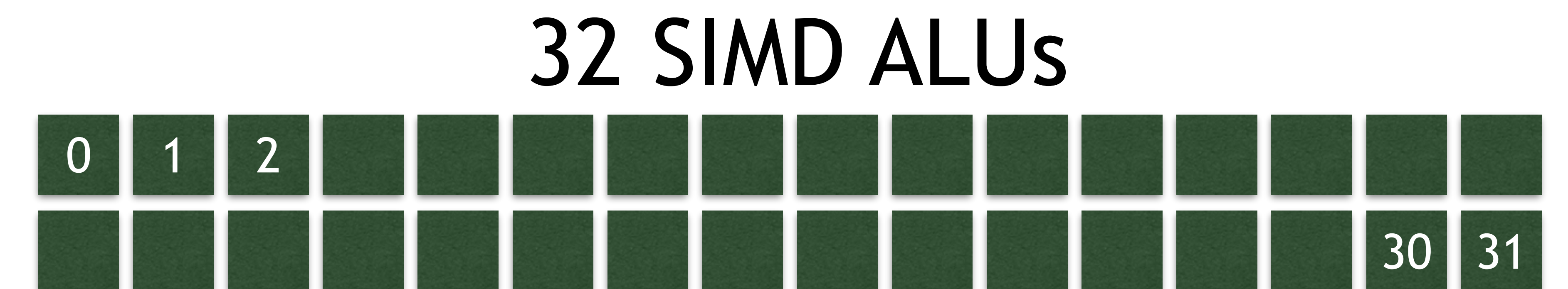
Register file (RF) is shared among contexts

- One register entry (vector) has 32 words (each 32bit)
- RF: 16 entries -> Max. of 4 registers/warp

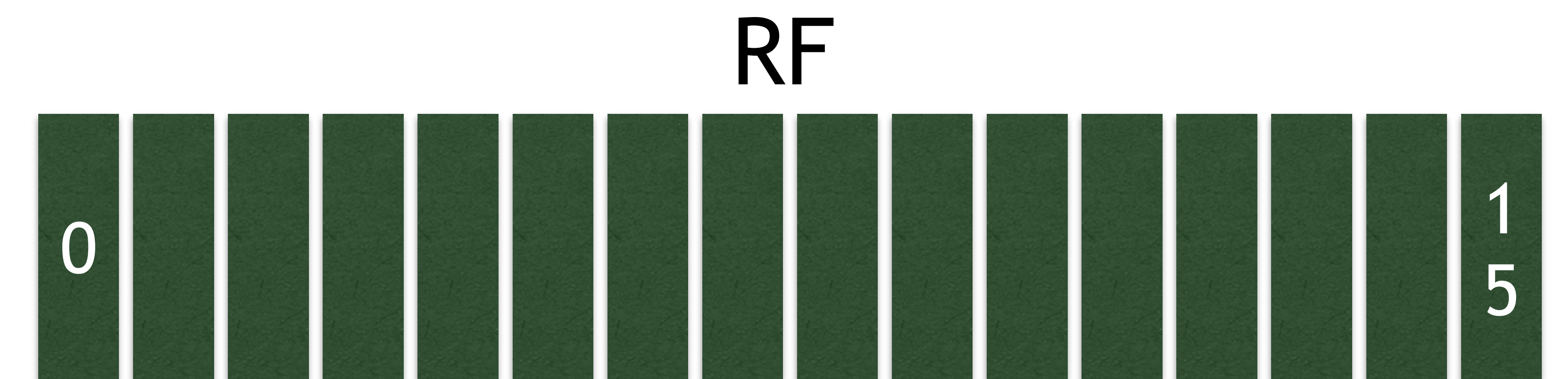
Simplifying assumptions

- Each memory access blocks execution for 50 cycles
- A memory access occurs every 20 cycles

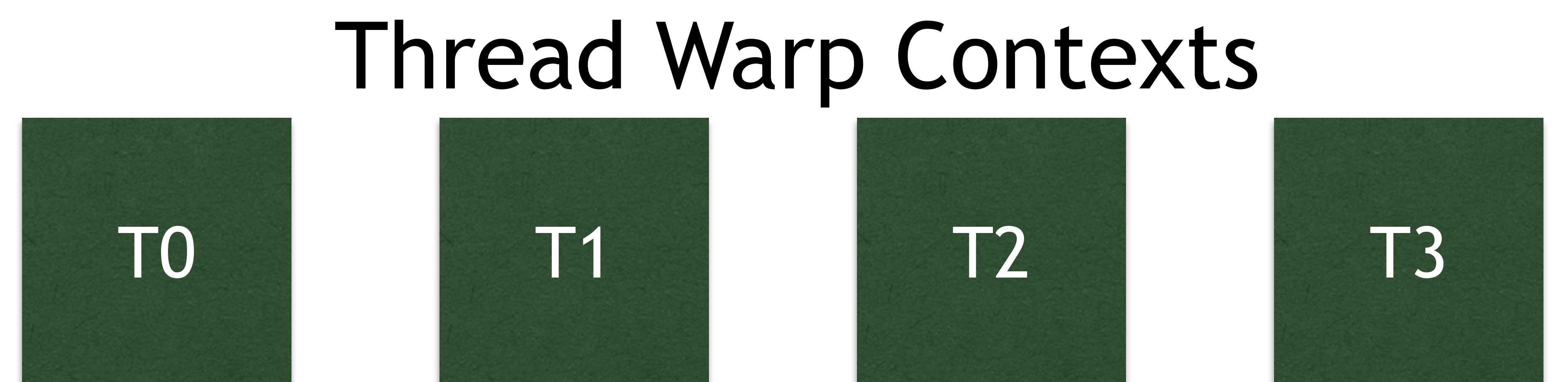
32 SIMD ALUs



RF



Thread Warp Contexts



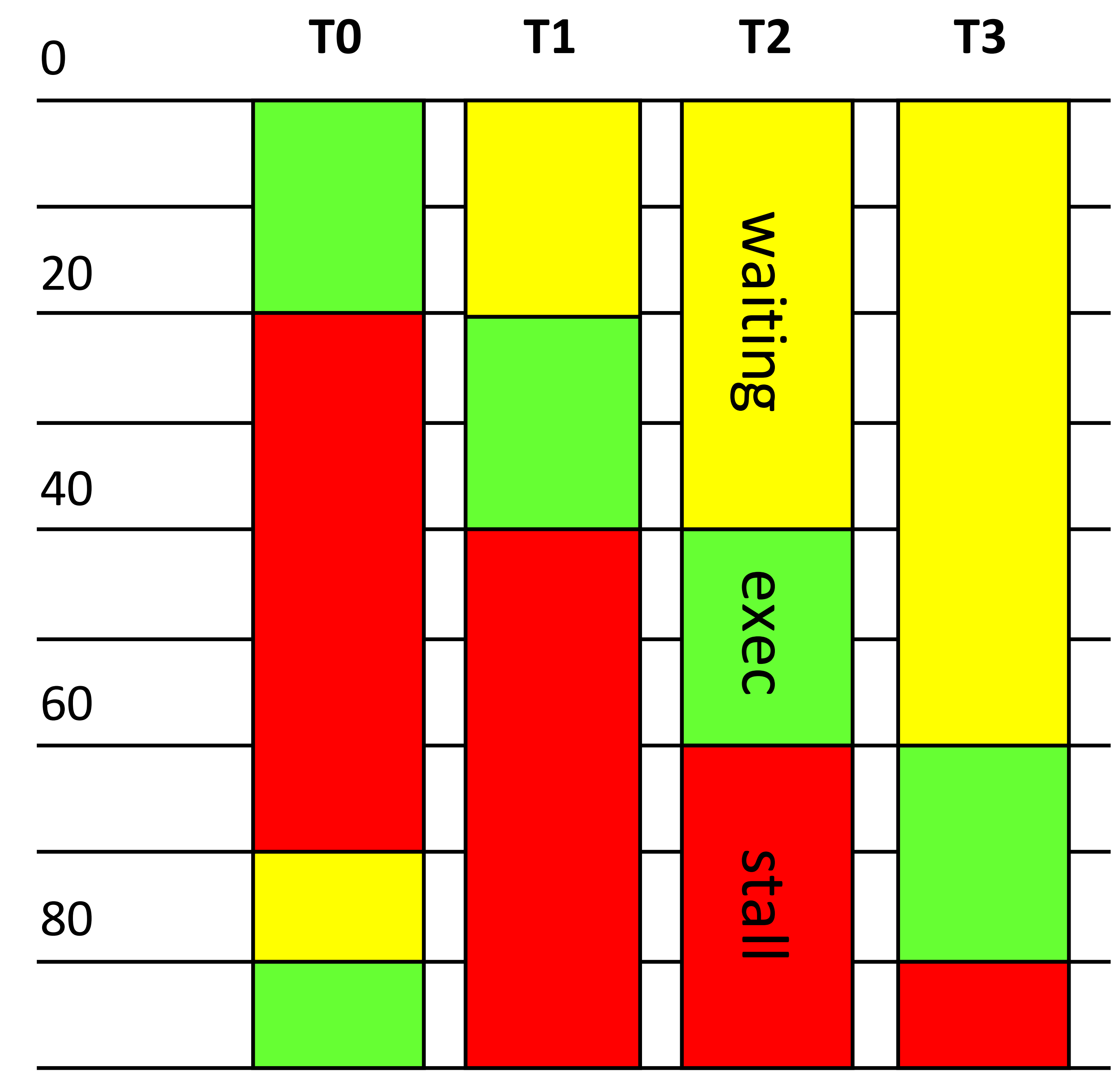
EXAMPLE FOR HARDWARE MULTI-THREADING (G80)

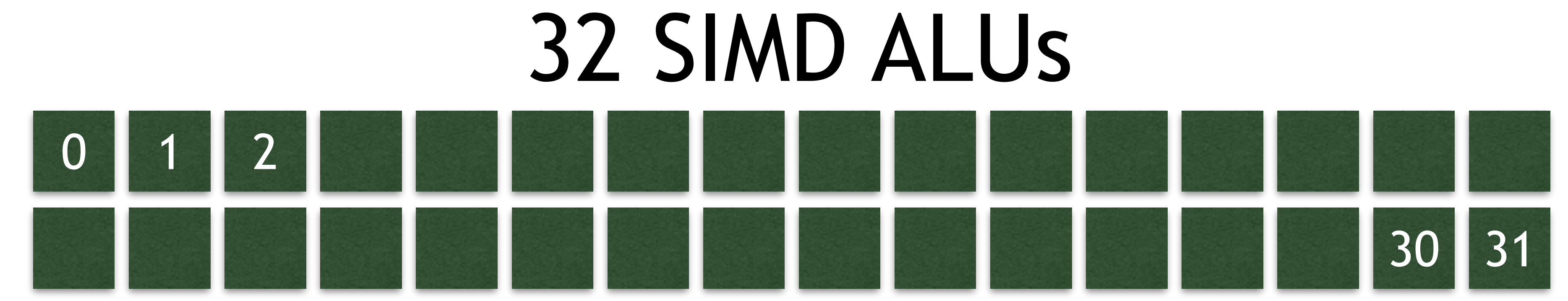
Each memory access blocks execution for 50 cycles (texture memory)

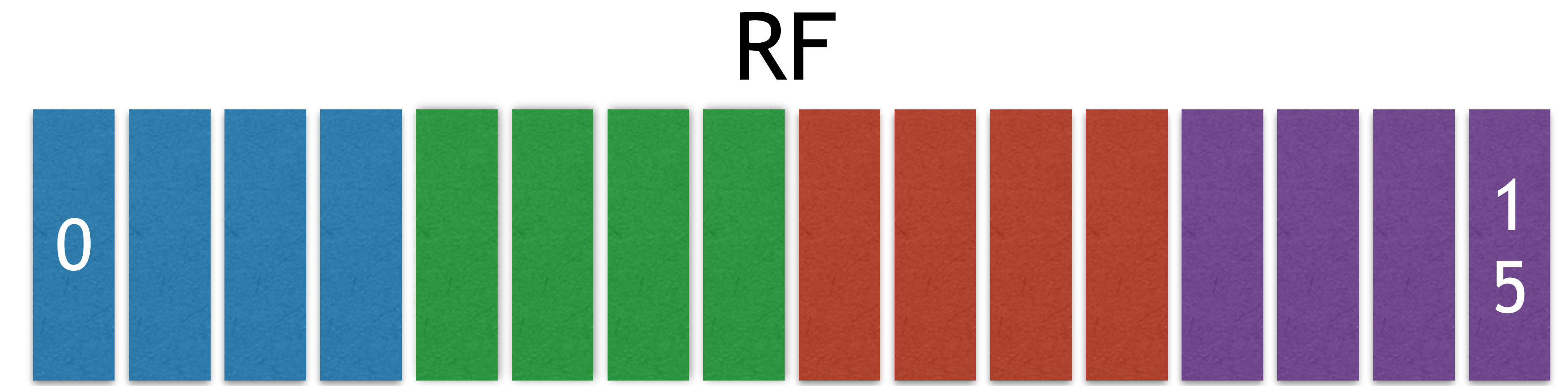
A memory access occurs every 20 cycles

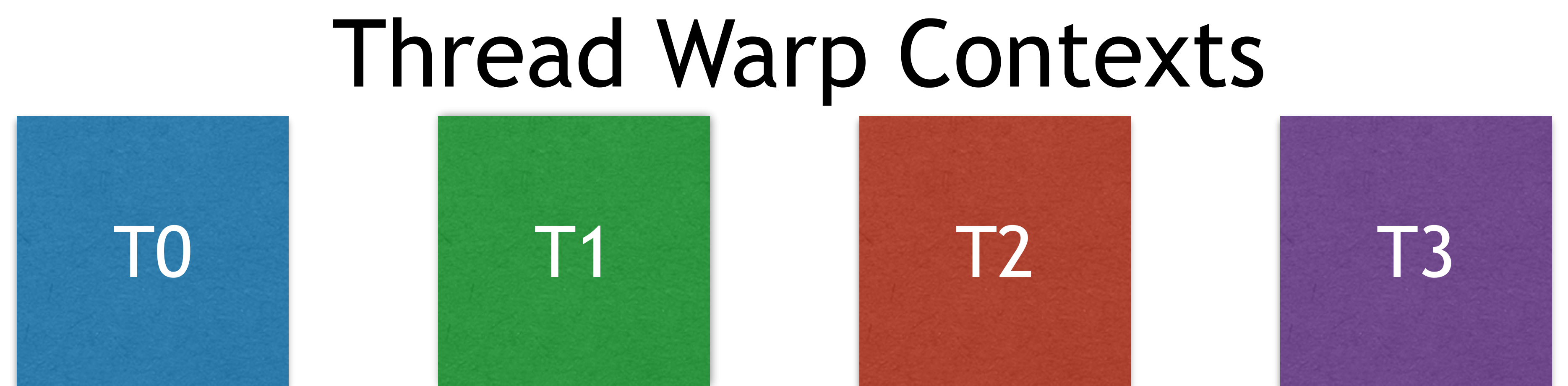
-> 4 thread warps required for full utilization

-> Per thread warp 32 entities = 128 entities









THREAD SCHEDULING (KEPLER)

Fetch one instruction per cycle (from I$)

Determine dependencies (operands)

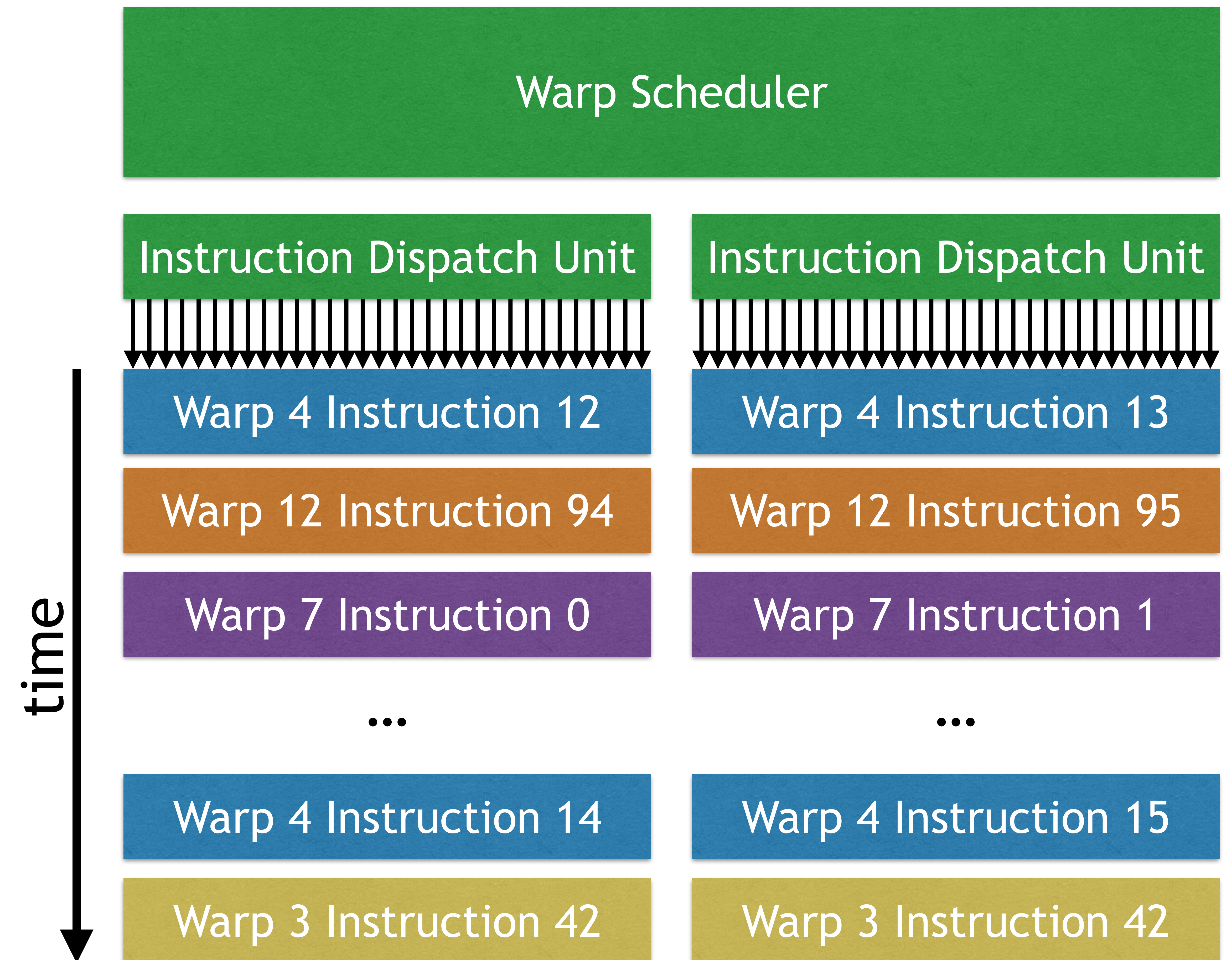
Scoreboard checks if dependencies are resolved

Prevent data hazards

Issue: select one warp based on prioritized round-robin

Priority: warp age

Scheduler broadcasts the instruction to all 32 threads in a warp



THREAD SCHEDULING - SCOREBOARD

Scoreboard: hardware table that tracks

- Instructions (fetched, issued, executed)
- Resources/Functional units (occupation)
- Dependencies (operands)
- Outputs (modified registers)

Tracks all operands of all instructions in the instruction buffer

- Any thread can proceed until scoreboard prevents issue
- OOO execution among warps
- Unfeasible without warp abstraction (32x less issue slots required)

Scoreboard: old concept from 1960s

- Separate computation and memory resources
- CDC6600 (https://en.wikipedia.org/wiki/CDC_6600)
- Enabler of OOO execution for CPUs

wikipedia.org

THREAD SCHEDULING – BRANCH DIVERGENCE

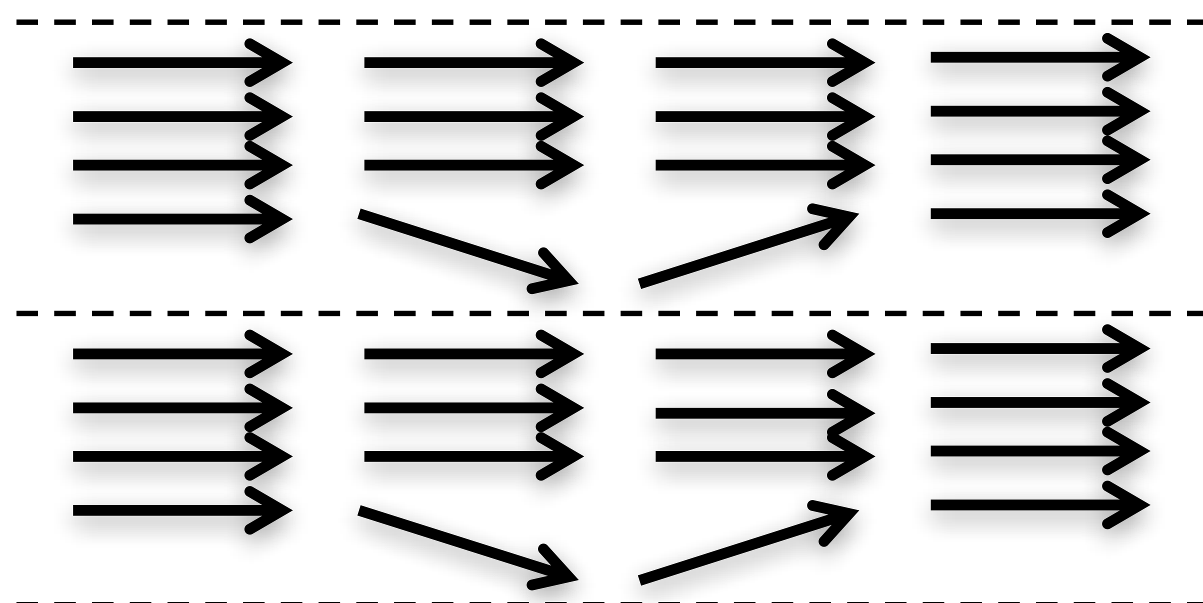
Scheduler broadcasts the instruction to all 32 threads in a warp

- Dedicated control paths
- Branch divergence problem

-> Write-masks

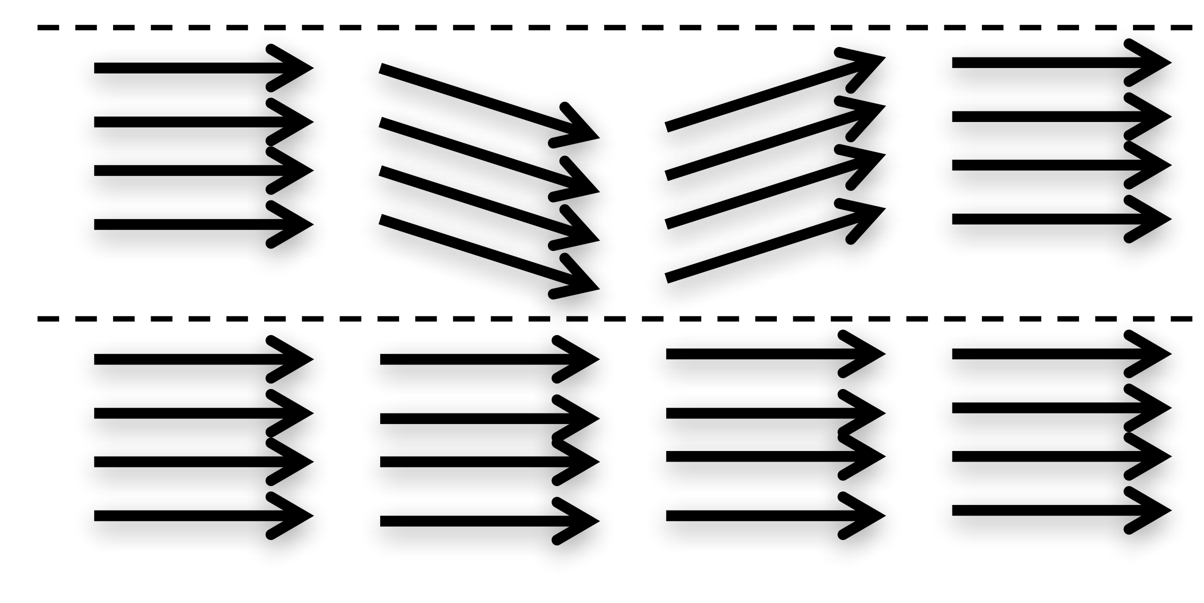
```
__global__ kernel1 (…)
{
  id = threadIdx.x;

  if ( id % 32 == 0 )
    out = complex_function_call();
  else
    out = 0;
}
```



```
__global__ kernel2 (…)
{
  id = threadIdx.x;

  if ( id < 32 )
    out = complex_function_call();
  else
    out = 0;
}
```



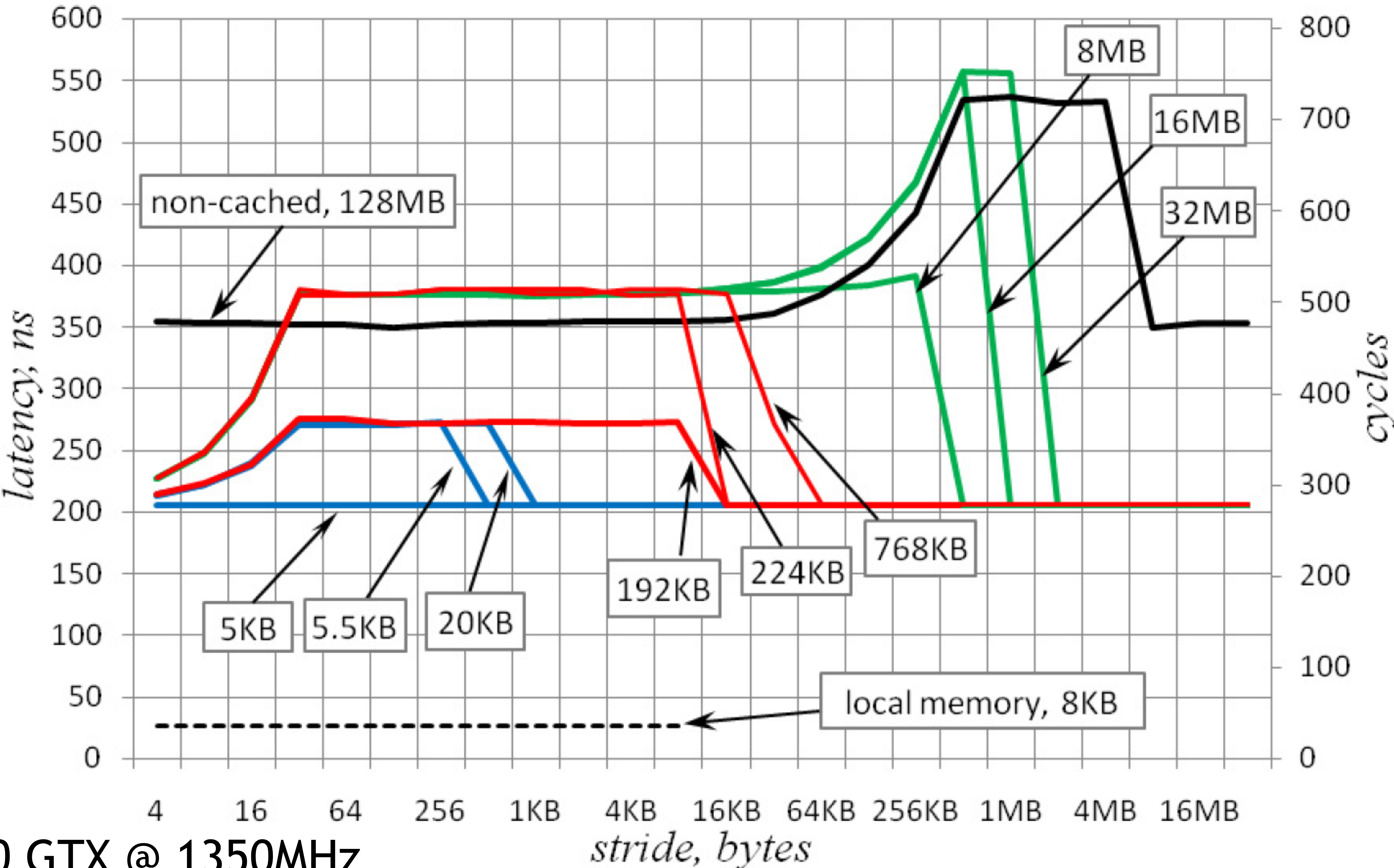
SUMMARY

SUMMARY

- GPUs have manually-controlled, rather flat memory hierarchies
  - CPUs = deep memory hierarchy
  - Caches in GPUs not used to reduce latency, but to reduce memory contention and to coalesce accesses
- Parallel slackness as in BSP
  - Latency hiding & scalability
  - Instruction stream == thread warp, != single thread (as for CPUs)
- Global memory subsystem
  - Fully featured memory subsystem, including virtual addresses, MMU and TLB
- Performance issues
  - Latency hiding: insufficient number of threads
  - Too many threads: register spilling
  - Coalescing issues (global memory): stride and offset
  - Branch divergence

BONUS: ADVANCED MEMORY ANALYSIS

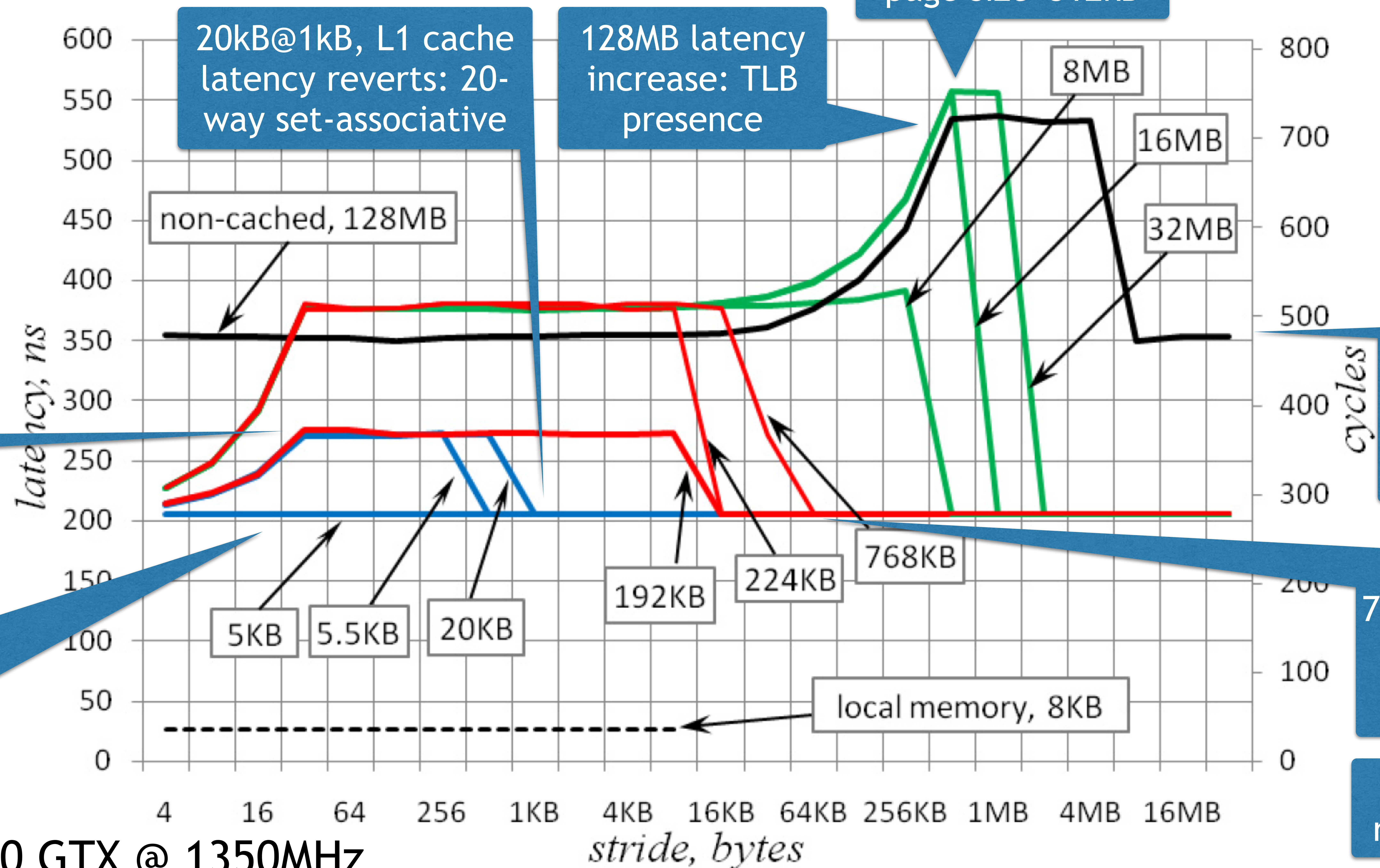
POINTER CHASING: MEMORY/CACHE ANALYSIS



GeForce 8800 GTX @ 1350MHz

Source: Vasily Volkov, James W. Demmel: LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs, LAPACK Working Note 202

POINTER CHASING: MEMOR E ANALYSIS



@32B, L1 & L2 saturation: CL size = 32B

L1 cache latency, 5kB size (latency increase for 5.5kB)

128MB@8MB stride, no overhead: 16-entry, fully-associative TLB

768kB@32kB, L2 cache latency reverts: 24-way set-associative

Ambiguous: or 6 replicated 4-way L2s

GeForce 8800 GTX @ 1350MHz

Source: Vasily Volkov, James W. Demmel: LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs, LAPACK Working Note 202