General Notes:

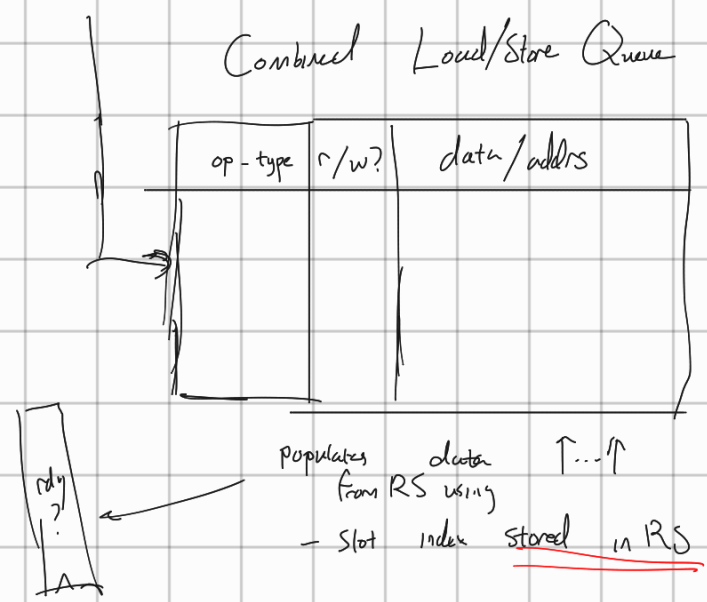
- don't support unaligned accesses
  (i.e. for N-byte load, address % N must be 0)
  this is not a serious limitation → CUDA has same
- Memory ordering is quite relaxed → loads and stores by one thread can appear in any order to another
- Coherence is also very relaxed: use write through L1; L2 is coherent. However L1 can contain stale data

Memory Pipe
Reservation Station

↓ ↓ ↓ ... ↓ x16 memory requests
(possibly some masked/predicated away)

Design 1: keep loads/stores in program order

from decode



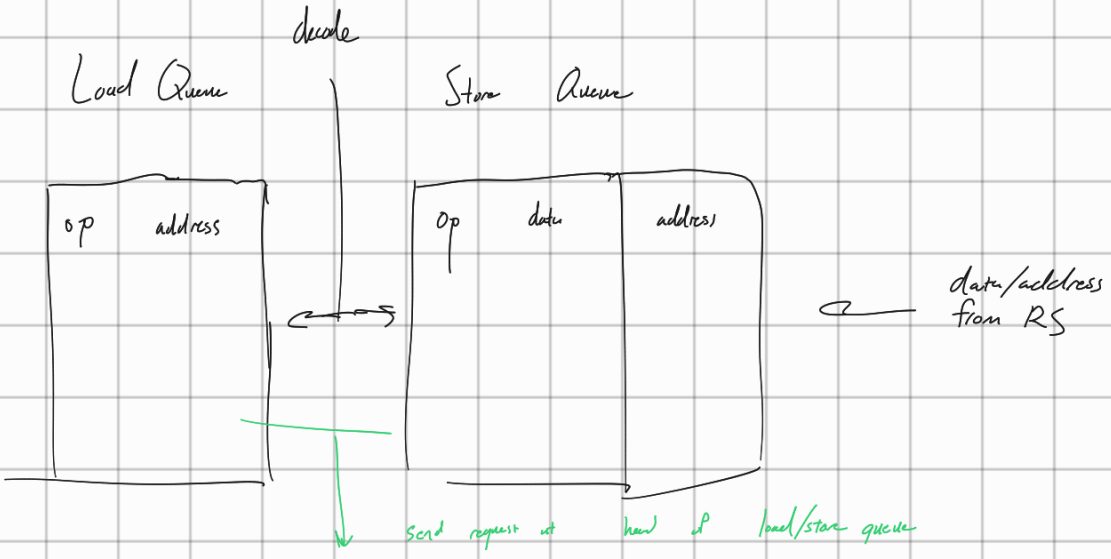
- Slots allocated in-order from decode stage
- only 1 outstanding memory transaction in flight per warp (too limiting?)

Cost is essentially 1 SRAM to each as queue

1RW would be optimal but probably not feasible

width dominated by $16 \times 32 = 512$ needed for store data
$+ 16 \times 32 = 512$ and addresses
(a lot is wasted for reads ...)

Design 2: keep loads/stores in program order, but separate them physically



- as before, slots allocated in program order from decode, slot index comes from RS

- when allocating a Load Queue entry, write tail pointer of Store Queue
- Load not allowed to issue until head of store queue reaches that stored value (i.e. all older stores retired)

- Store not allowed until it's at head of queue and load queue head > stored tail ptr

- Also forces program order but allows sizing

the queues differently

Design 2.5: keep stores in program order, allow reordering consecutive loads

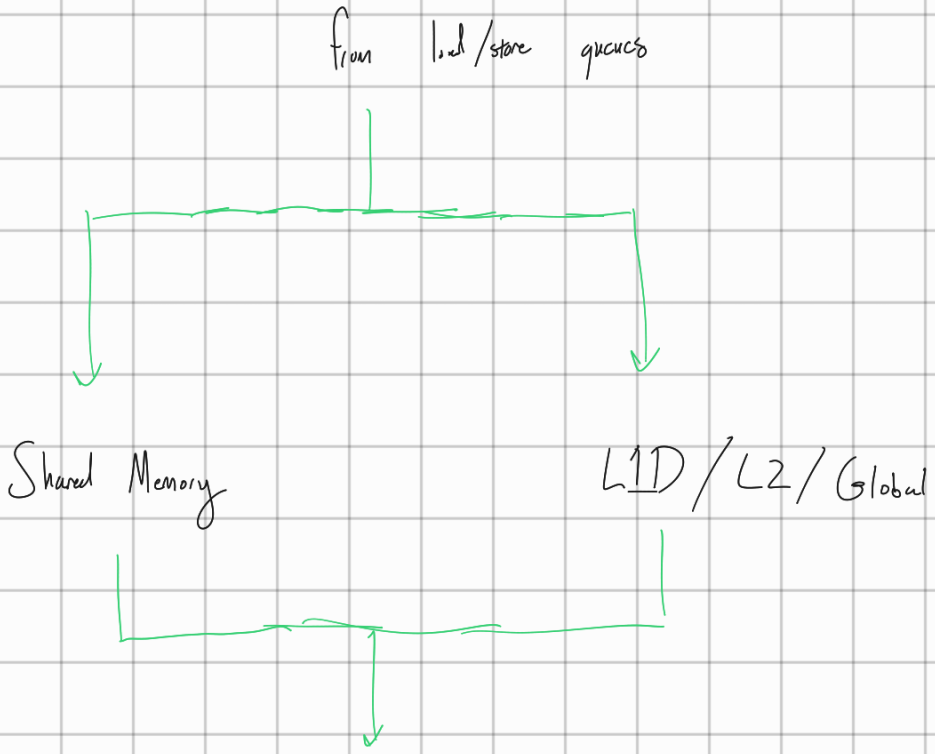
- Same as above, but instead of only issuing from head, issue from anything which is ready
- actually this seems much more sensible since it's really not much additional overhead over Design 2

Design 3: allows loads/stores to happen out of program order (full reordering)

- essentially a CPU load/store queue setup
  - loads can be reordered w.r.t any other load, but not older stores to same address
    (before starting a load, ensure no pending older stores to same address)
  - Stores cannot be reordered w.r.t older stores to same address or
    (before starting a store, ensure no pending store to same address / load to same address)

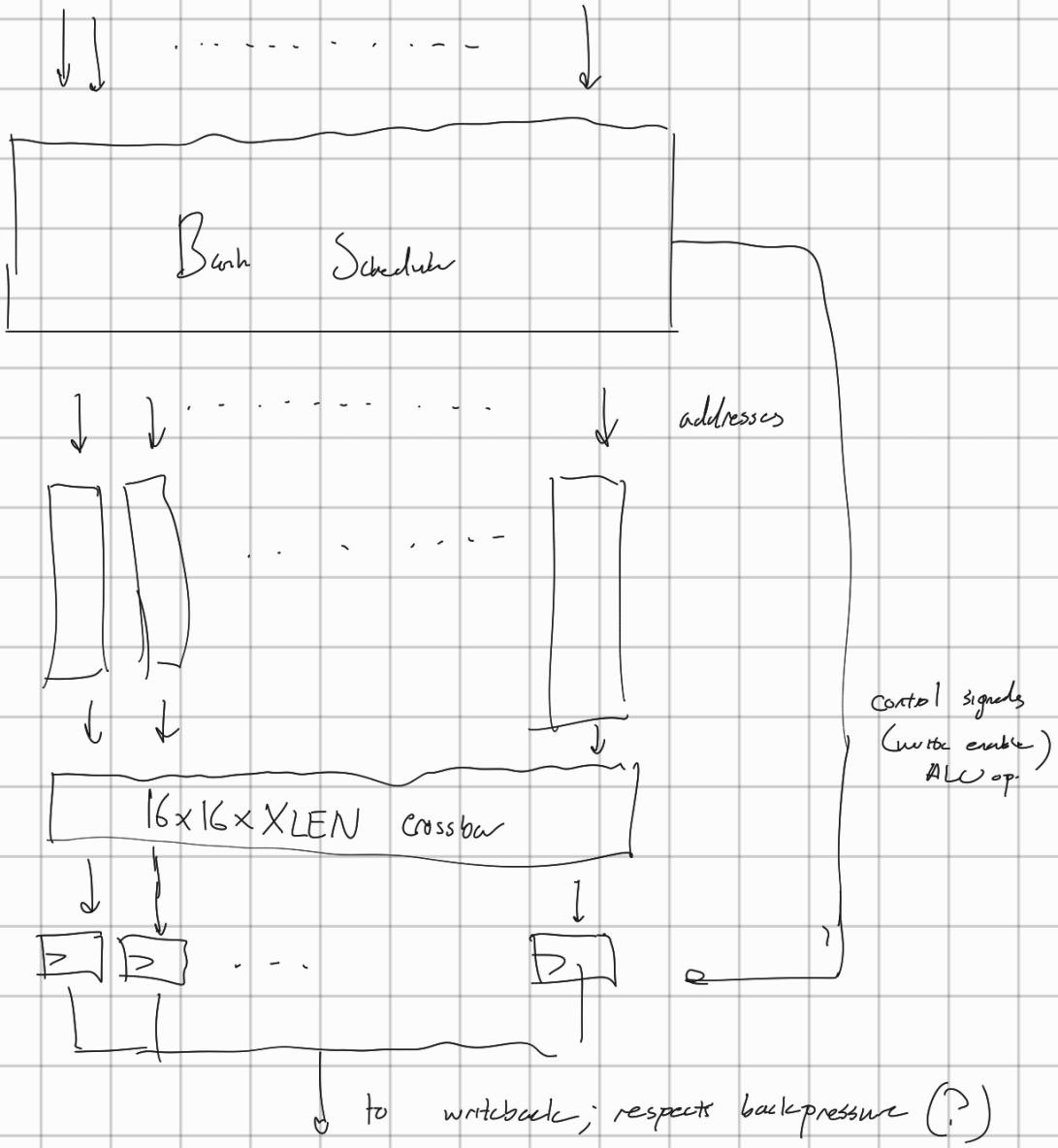
★ but we only enforce ordering within a single lane - if multiple threads non-atomically access same memory, it is very undefined ...

- i honestly think this is too expensive to do alias detection for 16 threads or speculation/replay both seem super expensive ...



- Note: not possible to do a mixed shared mem / global mem operation
  - ↳ they will be part of the same address space due to ISA limitation (can we change this?)
  - fault or undefined behavior if you try something weird
- Shared mem lines in certain address range — switch between SMEM/GLOBAL w/ 2 comparators
- Both can apply backpressure (e.g. serializing bank conflicts in SMEM) to issue, but busy shared MEM shouldn't affect global memory accesses

Shared Memory



16 banks, word interleaved

(word 1 ∈ bank 1, word 2 ∈ bank 2,
word 16 ∈ bank 16, word 17 ∈ bank 1, ...)

- Bank scheduler responsible for serializing bank conflicts

Stage 1: Decode + Broadcast Detection + Conflict Detection

LSBs = bank index (word-interleaving)

MSBs = address within bank (bank address)

- For reads, check if it is a broadcast

  (address$_1$ = address$_2$ = ... = address$_{16}$)

  - we can special case this to avoid very bad serialization

- Conflict detection / serialization

each bank uses some arbitration policy to pick 1 lane to service, chosen used as select into XBAR

  - Serviced lanes tracked, continue serializing until compare equal to thread mask

Step 2. Read/Write Data

- index into SRAMs, read/write. very easy

Atomics Support

TODO: how to efficiently support lr, sc?

way too many lanes to track (# warps x # threads)

do we actually need lr, sc?

- AMO* (+, −, min, max)

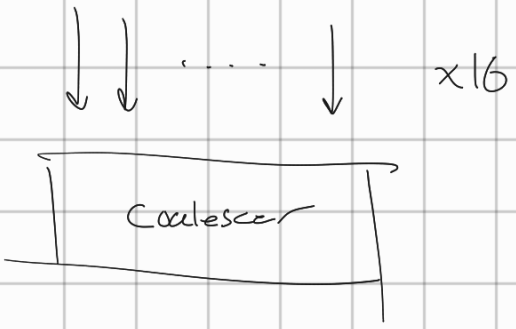
is easy on the other hand - just put in a

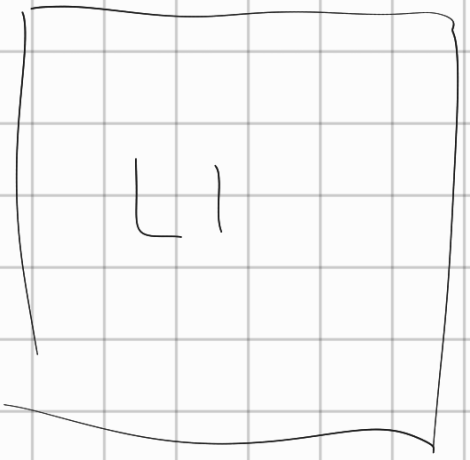
per-bank ALU and

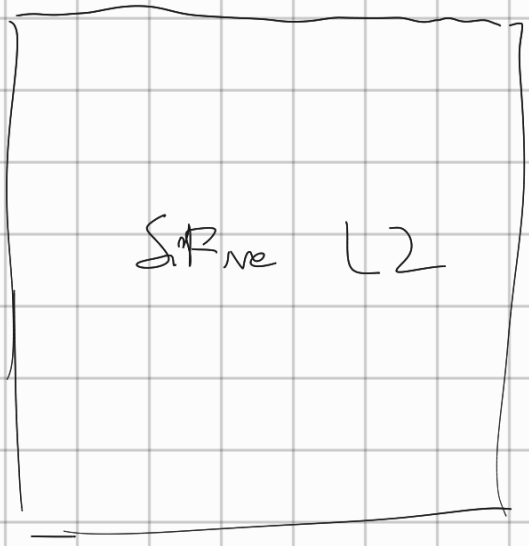
have bank scheduler take some

extra cycles to read, modify, then write.

LLD / L2 / Global







coalescer: can safely do spatial and temporal coalescing (?), but probably spatial only for this design

TODO: Can we reuse Rocket Chip's L1? Writing cache is hard

- probably want write-through cache for coherence reasons?

- need flush mechanism for kernel boundaries