Shared memory.

Within a thread block - sharing

Programmable L1 cache / scratchpad memory

Useful for repeated small data / synchronization.

```
__shared__ float a[N];      // inside kernel
__shared__ unsigned s;

a[id] = id;
if (id == 0) s = 1;
```

what is this id? — threadIdx only

no blockId ... as it is within a thread block.

5/2/2020.

Classwork.

You are given a 1024 x 1024 integer matrix M. Each row is assigned to a thread block. Each thread is assigned a matrix element M[i][j]. It changes M[i][j] to M[i][j+1] (where possible). Exploit shared memory.

```
__global__ void stencil(int* M, int N){
  __shared__ int mrow[1024];
  // copy from M to mrow using blockIdx.x
  // if(threadIdx.x < 1022){
  //   mrow[threadIdx.x] = M[blockIdx.x * blockDim.x + threadIdx.x];
  // }
  // Compute if(threadIdx.x <= 1022){
  //   M[blockIdx.x * blockDim.x + threadIdx.x] = mrow[threadIdx.x] +
  //                                              mrow[threadIdx.x + 1];
  // }
}
```

* Need to ensure that all the copying from M to mrow gets done before the compute step starts.

// For all the threads within a warp, no issues.

0 - 30 ✓ 31st ← can be a problem

next warp 0th thread - done/not ??

Precise dependency: last thread of 1 warp waits for first thread of next warp.

Over approx: All writes to mrow finish before the compute part starts.

⇒ **Barrier**. after copy step, before compute step.

__syncthreads(); // ensures all the threads within a thread block reach that point before proceeding any further.

Within a warp, implicit barrier.

```
if (threadIdx.x == 0)   s = 0;
if (threadIdx.x == 1)   s += 1;
if (threadIdx.x == 100) s += 2;
if (threadIdx.x == 0)   printf("s = %d\n", s);
```

} within a warp, one after another

→ another warp

6/2/2020.

CW. What is the o/p ?

```
#include <stdio.h>
#include <cuda.h>
#define BLOCKSIZE 26
__global__ void dkernel() {
  __shared__ char str[BLOCKSIZE+1];
  str[threadIdx.x] = 'A' + (threadIdx.x + blockIdx.x) % BLOCKSIZE;
  if (threadIdx.x == 0) {
      str[BLOCKSIZE] = '\0';
  }
  if (threadIdx.x == 0) {
    printf("%d : %s\n", blockIdx.x, str);
  }
}
int main() {
   dkernel<<<10, BLOCKSIZE>>>();
   cudaDeviceSynchronize();
}
```

< 32 so, within a warp, so it is executed in SIMD fashion

Q) What is the potential bug in this code?
→ If BLOCKSIZE > 32, then the multiple warps may run in asynchronous manner---

Put syncthreads() (1) b/w the 2 if's ? both are okay.
(2) before first if ? ∵ 2 ifs are going to be within a thread, so seq

Where can data race happen?

CW: Convert the following C codes to optimized CUDA:

```
for (i=0; i<N; i++) {
    p += a[i];
    q *= g * a[i];
    r = q / a[i];
}
```

__global__ void dkernel (int *a, int N, int *p, int *q, int *r) {

p | // 1st option - sequential
// 2nd option - atomic instruction
atomicAdd (*p, a[tid.x]);
can be slower than for loop on GPU.
// 3rd option → divide & conquer / reduction.
↳ likely to be faster than atomic add

q | same funda as p.

r ← final q / a[final]. (if only last one is reqd)

Q) Do we need barrier b/w p, q, r assignments?
→ within a thread, within a warp
warps within a thread block / across thblocks
⟹ atomic instruction & divide & conquer ← okay

```
for (i=0; i<N; ++i) {
    for (j=0; j<M; ++j) {
        p = a[i*M + j];
        q = q * a[i*M + j];
        r = q / a[i*M + j];
    }
}
```

10|2|2020.

L1 v/s shared.

cudaFuncSetCacheConfig (dkernel, cudaFuncCachePreferNone);
" " " " L1
" " " " shared

Dynamic shared memory.

When the amount of shared memory reqd is unknown at compile time, dynamic shared memory can be used. -- dkernel() {

dkernel<<< --, --, L >>>( );

extern __shared__ int s[];

} ↑ amt of dynamic shared memory we require

similar to main (int argc, char* argv[], char* env[])
↑
environ vars.

most of the time, we would not require dynamic shared memory. Only when the amount of work done by a thread depends on the input, we would need dynamic shared memory.

Q) ~~How to~~ Can we specify multiple shared variables? ... s[]
... s1[]

Bank conflicts. Try catching cuda errors

- Shared memory is organized into banks
- Accesses to a bank are sequential.

↑ queue of requests to one bank.

If all the threads ~~are~~ require data from the same bank, then the requests will be serviced sequentially. ✗ bank conflict ✓ (see below)

4 bytes = 1 word ⇒ characters will be stored 4 at a time; Consecutive words are stored in adjacent banks → useful for coalesced access.

integers/floats ⇒ one in each bank.

We cannot change the way banks store data. But what we can change is our access pattern.

EXCEPTION: Warp accesses to the same word are not sequentialized.

Warp accesses to different words of same bank ⇒ bank conflict

```
__global__ void bankNoConflict() {
    __shared__ unsigned s[1024];
    s[threadIdx.x] = threadIdx.x;
                    // stride of 1
                    // normal access.
}
```

```
__global__ void bankConflict() {
    __shared__ unsigned s[1024];
    s[32*threadIdx.x] = threadIdx.x;   // stride of 32
}
```

Suppose there are 32 banks — DIAGRAM

Accesses to diff words in same bank ⟶ Bank Conflict

* Recall False Sharing.

Multicore programming ⟵

write to an array element. — spatial locality is a liability here
invalidate again & again.

HW: create programs that create bank conflicts
measure time taken

Texture Memory

- Fast read only memory cache
- Optimized for 2d spatial access.
-

```
texture<float, 2, cudaReadModeElementType> tex
```

(2 = # dims)

cudaMallocArray

in main(): cudaBindTextureToArray(tex, cuArray, ...);

in kernel: ... = tex2D(tex, ...);

→ example from cuda SDK

Cannot write to texture inside a kernel.
But can do multiple dimensions

Constant Memory.

- Read only memory
- 64 kb per SM
- __constant__ unsigned meta,
- main(): cudaMemcpyToSymbol(meta, &hmeta, sizeof(—)),
- kernel: data[threadIdx.x] = meta[0];

Compute capability

Version no. major.minor eg 3.5

→ features supported by GPU hardware. Backwards Compatibility.

→ used by application at runtime.

- arch = sm_35.

Compatibility. —

1 - Tesla
2 - Fermi
3 - Kepler
and so on.

~~CUDA_ARCH <= defined in device code~~

Cuda version ⇒ Software part

CUDA_ARCH ← device code

>350 for simulated thing — atomics on double simulated using float.

CW.

Write CUDA code for the foll. functionality; use shared mem

Assume foll. data type, filled with some values.

struct Point { int x, y; } arr[N];

Each thread operates on 4 elements.

Find the average AVG of the 'x' values.

If a thread sees 'y' value above the average, it replaces all 4 y values to AVG.

Else, it adds y values to a global sum.

Host prints # elements set to AVG.

```
__global__ void k() {
    extern __shared__ int x[];
    extern __shared__ int y[]
}
    <<< x, y, zero >>>();
```

11-2-2020.

SYNCHRONIZATION.

- **Mutual Exclusion** No two processes are accessing the same shared resource at the same time.

  * **Critical section** - arbitrary piece of code, such that, if all threads follow the same ~~semantics~~ discipline / protocol for dealing with the critical section, they'll all see a semantically consistent value; a sensitive piece of code. (usually small). ↓ Even if one doesn't follow, we'll be in trouble.

  * **Locks** -

- **Atomic section** - Effect of access is, that section is fully executed, or not executed at all.

- **Deadlock**.

  ~~read write~~

  * Parallel / concurrent execution.
  * cyclic dependency.
  * Hold and wait.
  * No preemption.
  * At least 2 resources (shared) reqd at same time by more than 1 thread. But each thread has some resources & needs / waits for some others.

Locks are usually blocking

B) How to avoid deadlocks?

① Deadlock prevention → Single lock for all resources, but - Performance will suffer a hit. ∵ not all resources may be used at a time

② Allow preemption. → Removes one of the necessary conditions for deadlock

③ Enforce an order (total order) on the acquisition of resources.
→ breaks the cyclic dependency.

* avoids deadlock but may cause **starvation**.

- How to avoid starvation? Time-based priorities assigned to the locks / resources.

  ↓ wait

  eg. priority ↑ with time, so a process waiting for long time gets more priority.

(process with lowest priority would starve otherwise—)

Linux → CFS: Completely Fair Scheduling

multi level feedback queues).

* Another way is to prove that ~~through~~ deadlocks can happen, they are very rare. Many practical systems follow this—

→ T_1 locks R_1 : it doesn't have R_2, T_1 releases R_1 & goes away

T_2 locks R_2 : it doesn't have R_1, T_2 releases R_2 & goes away.

This keeps happening again & again

⇒ Not deadlock, but ~~Deadlock~~ livelock (Pehle aap)

Not blocked altogether, but not making progress either.

Data Race. 4 necessary conditions.

1. Multiple threads
2. Common memory location
3. Atleast one write
4. Concurrent execution

Data races are usually not good news.

But for implementation of locks, data races are useful.

8) How to get rid of data race?

→ avoid one of the 4 necessary conditions.

1. Execute sequentially
2. Privatization / data replication (foll. by some kind of merging)
3. Separating reads & writes by a barrier.
4. Mutual exclusion.

Classwork.

T1

```
1  flag = 1;
2  while (flag) {
       ⋮
   }
3  S1;
```

T2

```
4  while (! flag) {
       ⋮
   }
5  S2;
6  flag = 0;
```

1 2 stack

~~1 2 stack~~

1 4 5 6 2 3 — S_2 S_1

1 4 2 3 5 6 — S_1 S_2

4 1 5 6 2 3 — S_2 S_1

4 1 2 5 3 6 — S_2 S_1

4 5 6 1 2 3 — S_2 S_1.

4 stack.

12/2/2020.

Classwork

Given roll numbers & marks of 80 students in GPU programming, assign grades.

S = 90, A = 80, B = 70, C = 60, D = 50, E = 40, and U.

Use input arrays and output arrays.

→ Naive: one thread for each student, with one thread block.

```
__global__ void divmod(int *marks, char *roll_no)
    __shared__ char grades[N];
    int uid = (blockDim.x * blockIdx.x) + threadIdx.x;
    if (uid < N) {
        if (marks[uid] >= 90) {
            grades[uid] = 'S';
        }
        else if (marks[uid] >= 80 && marks[uid] < 90) {
            grades[uid] = 'A';
        }
        ...
        else {
            grades[uid] = 'U';
        }
    }
}
```

roll_no: our roll no has the format of CS16B011, 8 chars long

No real synchronization required

Suppose we also want the total counts of no. of students with each grade → histGrades[7]

#S, #A, #B, #C, #D, #E, #U

Now, we need to maintain counts of each grade.
There can be 2 threads trying to access same counter. ⇒ synchronizati
reqd.

```
__global__ void kernel(int *marks, int N) {
  __shared__ char grades[N];
  __shared__ int histGrades[7];
  int wid = ...;
  if (wid < N) {
    if (marks[wid] >= 90) {
      grades[wid] = 'S';
      histGrades[0]++; // problem
      atomicInc(&histGrades[0], INTMAX);
```

↓

guarantees that, if 2 threads are trying to access at same time, they will go one after another - resulting in correct results

```
    }
  }
}
```

Q) Do we need __syncthreads() ? → No. It doesn't matter or rate in which order the diff threads are computing. (in this case).

Eg: If we want the max of all elements in histoGrades we'll have to wait for all elements of histGrades to be computed, then only we ~~need to~~ can do atomicMax.

So, __syncthreads() would be reqd here.

Blocks of statements are separated by barriers into "phases"

We need to check for data race only within a phase, not across phases.

Shortest Path in a Graph.

Given an input graph of India, you want to compute the shortest path from Nagpur to every other city. Assume you are given a GPU graph library and the associated routines. Each thread operates on a node and settles distances of the neighbours (Bellman Ford).

↓

nicely parallel.

contrary to Dijkstra's (inherently sequential, uses priority queue)

```
__global__ void dsssp (Graph g, unsigned *dist) {
    unsigned id = ...;   // one Thread – one node
    for each n in g.allneighbours(id) {
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            dist[n] = altdist;   // should be an atomic update
        }
    }
}
```

→ incorrect

Kernel should be called a number of times from CPU

→ max (n−1) times, where n = no. of nodes

(*) Persistent kernel
- (*) if some temp. computation result is stored in shared memory
- (*) great load imbalance → eg. in a graph/social network

Synchronization — Control + data flow
Atomics
Barriers

CW · Implement mutual exclusion for two threads - using only control & data flow methods

T_1	T_2	
{	{	S_1, S_2 both are writing to same
S_1	S_2	shared variable.
:		

How to prevent this from happening?

bool flag = false; // Shared

T_1	T_2
≡	≡
while (flag);	while(!flag);
S_1;	S_2;
flag = true;	flag = false;
≡	≡

Issues

// Performance drop - spinning on while - unnecessarily wastes CPU cycles

// Codes for diff threads are different.

// If we launch less number of threads (1 in this case), then it gets blocked.

14/2/2020

Debugging and Profiling.

Debugging.

* Non-determinism due to thread-scheduling - diff outputs possible
* Depending on the application semantics, intermediate steps may also need to be deterministic.
* No. of correct intermediate values - may be large.
* cuda-gdb - for debugging CUDA programs on real hardware.
* There are GPU simulators also...
* cuda-gdb allows break points, single step, read/write memory contents.

* stable sorting - duplicate elements appear in same order as in input

(*) cuda-gdb → Design decision → restricts parallelism to a single thread block.

Prev errors - probably overflow / etc may still continue.

If there are multiple errors, we'll get info about last one only.

in main():

```
cudaError_t err = cudaGetLastError();
printf("error = %d, %s, %s\n", err, cudaGetErrorName(err),
                               cudaGetErrorString(err));
```

eg. error = 77, cudaErrorIllegalAddress, an illegal memory access was encountered.

cuda-gdb.

```
nvcc -g -G file.cu
```

- `-g`: adds symbol info into a.out (CPU code)
- `-G`: adds symbol info into a.out (GPU code) - kernels.

// also, disables optimizations.
∴ we would be single-stepping through the source code, so we should be able to see deterministic execution for our source code.

cuda-gdb a.out // loads a.out symbol info and stops.

> //prompt

> run // to execute the program

* may have to stop windows manager — GUI part... : graphics application

* Due to lots of threads, cuda-gdb works with a focus (current thread).
  ↓
  probably doesn't perform SIMD execution... confirm this...

(cuda-gdb) run

:

CUDA exception : Device Illegal address

starts with 1 rather than 0.

[Switching focus to CUDA kernel 0, grid 1, block (1,0,0), thread (0,0,0), device 0, SM 13, warp 0, lane 0]

↑ thread within warp

Error: shows printf(...); ← prev stmt is executed, error occurs & the cursor is here now.

Kernel <<< (1,2,3), (4,5,6) >>> ();

3 6

C - semantics:

(only 1 in this case)

(cuda-gdb) info cuda kernels. ← information about kernels.

(cuda-gdb) info threads ← information about each thread.

→ shows only 3 threads, but we launched 20. (2×10)

includes both CPU & GPU threads

2 1 → erroneous one

(cuda-gdb) info cuda threads ← information about GPU threads only

(cuda-gdb) cuda kernel block thread ← current focus

(cuda-gdb) cuda block 1 thread 0 ← switches focus to another block, thread.

Breakpoints:

(cuda-gdb) break main ← or any function name / line # file.cu : 223.

(cuda-gdb) set cuda break-on-launch application

Conditional breakpoint

break file.cu : 29 if threadIdx.x == 0 ...

Single step.: s, step.

Info about SMs: info cuda sms.

Info about warp: info cuda lanes. ⇒ shows that warp execution is actually in SIMD fashion for this execution : only 10 threads in a block

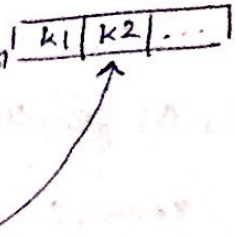
Profiling → next week

2 kernels → by default. same stream (queue type) ← seq

⇒ no need to have cudaDeviceSynchronize() b/w 2 GPU kernels.

but is a good practice to have b/w GPU & CPU instr

17/2/2020.



S1
K1 <<< ... >>> launch
x = 5
K2 <<< ... >>> launch
S2

S_2 can be executing in parallel with K1 or K2.
Only the launches happen seq.

Synchronization:

- control + data flow (initially, flag = false)

while (! flag)
S1;

S2;
flag = True.

Mutual Exclusion b/w 2 threads:

(V1) lock(), unlock()
Methods should be same.

lock();
CS
unlock();

Say, tids => 0, 1

lock()

```
L1 me = tid
L2 other = 1 - me;
L3 flag [me] = true;
L4 while (flag [other]);
```

unlock()

```
U1 flag [tid] = false;
```

T1 - L1 L2 L3
T2 - L1 L2 L3

Q1) Can the threads wait indefinitely? Yes - both wait for each (एक नहीं आया) other. (Otherwise, if 1 thr doesn't execute at all, other also can. If one goes into CS, other cannot go into CS.

Q2) Should flags be initialized to false? → Yes.

Q3) Does the code ensure mutual exclusion? → Yes

Deadlock can occur (Livelock → threads release lock, again get lock, again have to release --)

* Mutual exclusion guaranteed
* May lead to deadlock
* If one thread runs before the other, all goes well.

(*) Primitive assignments must be atomic

flag[me] = true / false // Inherent assignments

Rest assured that these will be atomic, provided by hardware.

(V2) volatile int victim;

keyword in C → disables compiler optimizations for that variable.
→ to convey to the compiler.

can remove the loop or CS : it seems like it'll never happen

Also. if var is in cpu's private cache, the var stays in shared memory - coherency issues avoided.

lock()
me = tid;
victim = me;
while (victim == me);

unlock()
;

Cooperative protocol - threads are not selfish
Give importance to "other".
Involve coordination b/w threads ⇒ spend more energy
individual threads have to

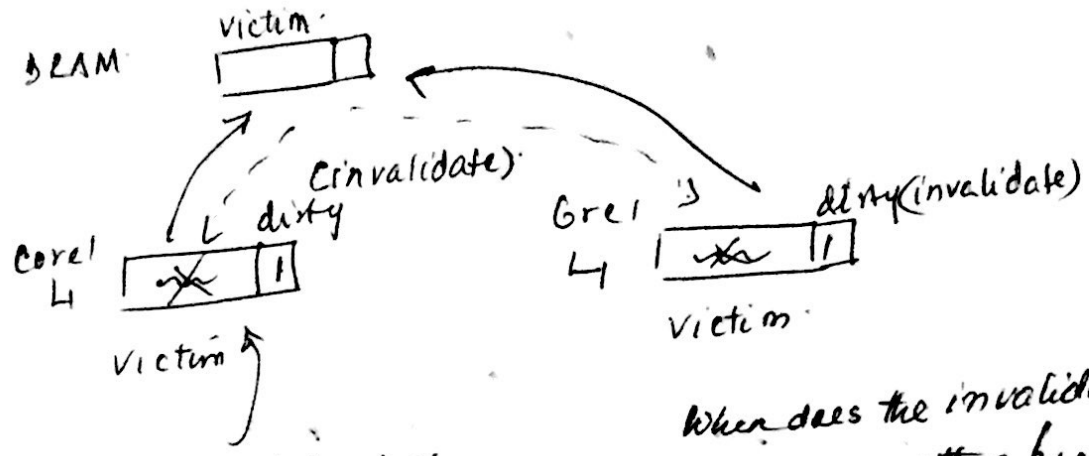
① If only single thread runs, leads to deadlock
② Does it ensure mutual exclusion? Yes.

works well if threads repeatedly take locks

Tid 0 → lock sets victim = 0
while (victim == 0);
Now, tid = 1 ⇒ lock sets victim = 1
Now this can get out. Tid 1 → waits on while loop

Both threads see same value for victim. ← to mutual exclusion

Mutex violated only if Tid 1 sees victim = 0 and Tid 0 sees victim = 1



1 read local copy.

When does the invalidation reach the other private L1?

By marking volatile, it ensures on CPU that:

a) writing → all the way back to shared mem

b) reading → all the way from DRAM/shared mem

On GPU:

Reading → all the way from DRAM ✓

But writing → ~~not~~ all the way ~~from~~ upto DRAM. shared mem

"Memory fence" - whatever mem ops are done for a thread are "pushed" to memory

Need to explicitly call memory fence in GPU.

threadfence(); //after every write to shared var

read ← mark var as volatile.

* ALSO, | data race is possible |.

However, when you implement a synchronization protocol such as lock(), you need to exploit data race.

(V1) → worked when threads were "away" from each other

(V2) → " " " " "close" to each other

(V3) → Combine both.

Volatile bool flag[2]; → must be initialized to false. → crucial to avoid a deadlock if other thread isn't there

Volatile int victim; } to ensure progress, to prevent Starvation.

lock()

me = tid;

other = 1 - me;

flag[me] = true;

victim = me;

while (flag[other] && victim == me);

unlock()

flag[tid] = false;

Peterson's protocol

↓

a) Does it ensure mutex? ENSURES MUTUAL EXCLUSION.

b) Can deadlocks occur? NO.

c) Can starvation occur? NO.

d) Is the algorithm "fair"? Yes... check ✓

↳ Both threads get opportunity to enter CS.

[Mutex, Progress, Bounded wait]

18/2/2020.

Peterson's lock for N threads.

while(flag[other] && victim == me)

↑

need to change to 'or' of all the flags....

Bakery algorithm.

- Devised by Lamport
- works with N threads
- Maintain FCFS using ever-increasing numbers.

Q) can we replace max operation by one variable, which uses atomic instruction?
→ YES.

bool flag[N]; // false } initially. should be volatile.
int label[N]; // 0

Assumption → everything happens in main mem; if caching is involved, problem may arise
CHECK: writing to label, reading from label → ??

lock()

me = tid;
flag[me] = true;
label[me] = 1 + max(label); → not atomic. within a warp, will all threads get same label? to break ties.

while(∃k != me : flag[k] && (label[k], k) < (label[me], me));

inherent loop - costly operation.

unlock()

flag[tid] = false;

Q) Does fairness ⇒ no starvation?
CHECK.

① Mutual Exclusion → preserved ✓

② Deadlock → No; Strictly '<' condition ensures total order
⇒ No cyclic dependency could arise

③ Starvation → No; flag[k] would be false for uninterested threads ✓

④ fairness → Yes ✓ FCFS will ensure fairness..

Bakery Algorithm, as we saw, is suited for CPU.

What about GPU?

Q) Does warp-based execution pose any trouble?

Across warps - shouldn't be an issue. Works on CPU also.

Blocking ---- ?

As long as you don't use atomic instr, the same code works on GPU. But performance is slow.

On GPUs, locks are usually prohibited.

* No. of threads Enormous — High spinning cost @ large scale.

Each thread will have to wait for long time.

Avoid locks as much as possible.

If really unavoidable & the conflicts ~~should be~~ are sparse, then could consider using locks.

Locks can be implemented using atomics.

Atomics

- primitive operations whose effects are visible either none or fully (never partially).
- needs hardware support
- eg. atomic CAS, atomic Min, atomic Add etc

Compare & Swap.

1 mem loc involved — being read & written

* works with both global mem (across thread blocks) & shared mem (within thread block)

```
__global__ void dkernel (int *x) {
    ++x[0];
}
...
dkernel <<<1, 2>>> (x);
```

5) what is the value of x at the end of this?

x = 1 or 2

Thread 0	Thread 1
load x[0], R1	load x[0], R2
inc R1	inc R2
store R1, x[0].	store R2, x[0].

interleaving ⟸ Could lead to 1/2

a) If x[0] itself req. multiple instructions --- problems.

→ Ensure all-or-none behaviour.

eg. atomicInc (&x[0], ...); // deterministic outcome (2)

(*) dkernel <<< K1, K2 >>> would ensure x[0] to be incremented by exactly K1 * K2, irrespective of the thread execution order.

→ This effect would be visible after kernel execution is over.

→ Could be any order of threads

t1 t2 t3 . . .

x:

1

2

3

→ A queue is formed.

PTO No

check

atomicInc --;

print ( ); → all threads within a warp see single final value of x

Across warps → Could be diff.

8) does atomicInc (&x[0], K); get split into multiple instn - non atomically?

Computation of x[k] mem address - not part of atomic instruction.

Actual load, add, store is atomic

Recall. dsssp — atomic Min (tdist[n], altdist);

lost update problem solved by atomic min ✓

Every thread has to follow atomic Min ✓ otherwise chaos.

1 Atomic CAS.

oldval = atomicCAS(&var, x, y);

Compare var, x.

If var == x, change var to y; Return x

otherwise don't do anything; Return x

z = atomicCAS (&var, x, y)

atomic { load var, R1
if R1 == x
store y, var
return R1 }

[CAS] generic atomic instr

usually available in Hardware

others like min etc built using CAS

19-2-2020.

Typical use cases of CAS:

- Locks: critical section processing
- Single: only one arbitrary thread executes the block.

(Parallel → seq → Parallel ...)

↑ done by a single.

- Other atomic variants. - atomic min etc can be simulated using atomic CAS

Ex. Implement lock with atomic CAS.

curr = 0 ; // some initialization.

lock()

mycurr = atomicCAS (&curr, 0 , tid);

0: curr == 0, then curr gets replaced by tid, 0 is returned
⇒ currently tid has acquired lock
& can go into CS.

!0: curr ≠ 0, then whatever that value is, is returned

lock()

do {
    mycurr = atomicCAS (&curr, 0, tid); // busy trying to acquire lock.
} while (mycurr != 0)
// comes here only if I am able to change curr from 0 to tid.
CS

---

unlock()

mycurr = atomicCAS (&curr, tid, 0); // replace curr value with 0, so that some other thread can acquire

↑ not reqd.

curr = 0 is enough

cw. Implement single with atomic CAS.

```
S1
single {
   CS
}
S2
```

All threads execute S_1, S_2. Exactly one of the threads executes CS. while CS is being executed, other threads could be executing either S_1 or S_2.

```
lockVar = 0; // initially
S1  // all threads execute S1, no problem
if ( atomicCAS(&lockVar, 0, 2) == 0) { // Doesn't block other threads from going ahead to S2.
   CS
}
S2
```

Also, since we are not unlocking, no other thread can go into the if

If single is inside for loop.
Reset the lockVar only after all the threads have finished S_2. // barrier reqd

Remember.
Atomic instructions do not behave like a barrier.

Multiple 'singles' → multiple lock variables.

Qu. What's the output? not deterministic.

```
#include <stdio.h>
#include <cuda.h>
__global__ void k1(int *gg) {
    int old = atomicCAS(gg, 0, threadIdx.x + 1);
    if(old == 0) {                    → executed by exactly one thread.
        printf("Thread %d succeeded 1.\n", threadIdx.x);
    }
    old = atomicCAS(gg, 0, threadIdx.x + 1);
    if(old == 0) {                    → will not be executed by any thread
        printf("Thread %d succeeded 2.\n", threadIdx.x);
    }
    old = atomicCAS(gg, threadIdx.x, -1);
    if (old == threadIdx.x) {
        printf("Thread %d succeeded 3.\n", threadIdx.x);
    }
}
int main() {
    int *gg;
    cudaMalloc(&gg, sizeof(int));
    cudaMemset(&gg, 0, sizeof(int));
    k1<<<2,32>>>(gg);
    cudaDeviceSynchronize();
    return 0;
}
```

2 threads (out) see this

if 32, none; 1 --- 31 tids

But: atomic, only one of them will be able to execute.

Even if warp threads are executing an atomic instruction, only one of the threads is going to do it at a time.

2 warps.