# Synchronization

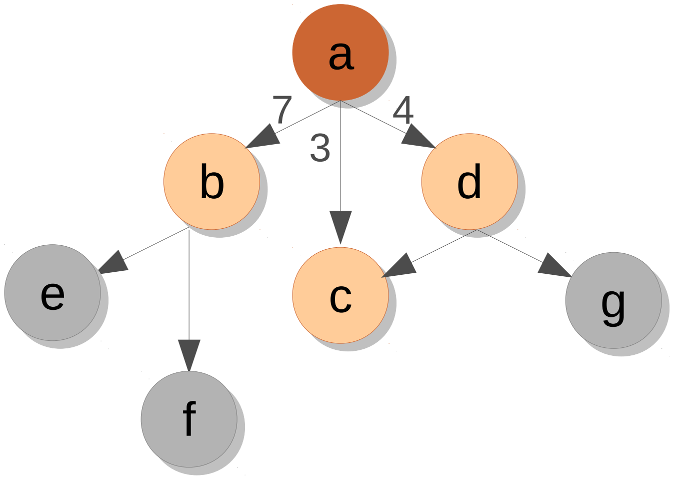
# Recap - atomics

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

- Ensure all-or-none behavior.
  - e.g., atomicInc(&x[0], ...);
- **dkernel**<<<K1, K2>>> would ensure x[0] to be incremented by exactly K1*K2 – irrespective of the thread execution order.
  - When would this effect be visible?

# Let's Compute the Shortest Paths

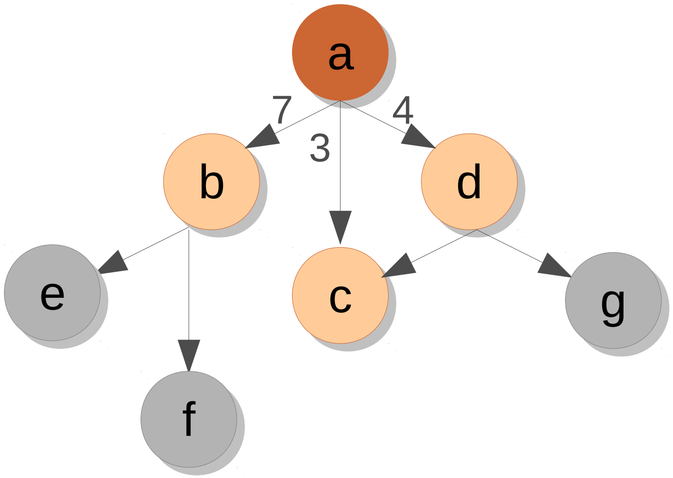
- You are given an input graph of India, and you want to compute the shortest path from Nagpur to every other city.
- Assume that you are given a GPU graph library and the associated routines.



```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = ...
    for each n in g.allneighbors(id) {      // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            dist[n] = altdist;
}    }    }
```

# Let's Compute the Shortest Paths

- You are given an input graph of India, and you want to compute the shortest path from Nagpur to every other city.
- Assume that you are given a GPU graph library and the associated routines.



```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = ...
    for each n in g.allneighbors(id) {      // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        ~~if (altdist < dist[n]) {~~
            ~~dist[n] = altdist;~~   atomicMin(&dist[n], altdist);
~~}~~   }   }
```

# AtomicCAS

- Syntax: oldval = atomicCAS(&var, x, y);

- **Typical usecases:**
  - *Locks*: critical section processing
  - *Single*: Only one arbitrary thread executes the block.

**Classwork: Implement *lock* with *atomicCAS*.**

# Lock using atomicCAS

Does this work?

```
    atomicCAS(&lockvar, 0, 1);
```

Does not ensure mutual exclusion.

Then how about

```
if (atomicCAS(&lockvar, 0, 1) == 0)
    // critical section
```

Does not block other threads.

Make the above code blocking.

```
do {
    old = atomicCAS(&lockvar, 0, 1);
} while (old != 0);
```

Correct code?

# Lock using atomicCAS

- The code works on CPU.
- It also works on GPU across warps.
- But it hangs for threads belonging to the same warp.
  - When one warp-thread acquires the lock, it waits for other warp-threads to reach the instruction just after the do-while.
  - Other warp-threads await this successful thread in the do-while.

Correct code?

```
do {
   old = atomicCAS(&lockvar, 0, 1);
} while (old != 0);
```

# Lock using atomicCAS

```
do {
    old = atomicCAS(&lockvar, 0, 1);
} while (old != 0);

// critical section

lockvar =
```

On CPU

```
do {
    old = atomicCAS(&lockvar, 0, 1);
    if (old == 0) {
        // critical section
        lockvar = 0;     // unlock
    }
} while (o
```

On GPU

**Classwork: Implement *single* with *atomicCAS*.**

# Single using atomicCAS

```
if (atomicCAS(&lockvar, 0, 1) == 0)
    // single section
```

Important not to set lockvar to 0 at the end of the single section.

# What is the output?

```
#include <stdio.h>
#include <cuda.h>

__global__ void k1(int *gg) {
        int old = atomicCAS(gg, 0, threadIdx.x + 1);
        if (old == 0) {
                printf("Thread %d succeeded 1.\n", threadIdx.x);
        }
        old = atomicCAS(gg, 0, threadIdx.x + 1);
        if (old == 0) {
                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;
}
```

- Some thread out of 64 updates gg to its threadid+1.
- Warp threads do not execute atomics together! That is also done sequentially.
- Irrespective of which thread executes the first atomicCAS, no thread would see gg to be 0. Hence second printf is not executed at all.
- If gg was updated by some thread 0..30, then the corresponding thread with id 1..31 from either of the blocks would update gg to -1, and execute the third printf.
- Otherwise, no one would update gg to -1, and no one would execute the third printf.

- On most executions, you would see the output to be that thread 0 would execute the first printf, and thread 1 would execute the third printf.

# Synchronization

- Control + data flow
- Atomics
- Barriers
- ...

# Barriers

- A barrier is a program point where all threads need to reach before any thread can proceed.
- End of kernel is an implicit barrier for all GPU threads (global barrier).
- ~~There is no explicit global barrier supported in CUDA~~. *grid*.sync() is now supported (from CUDA 9).
- Threads in a thread-block can synchronize using __syncthreads().
- How about barrier within warp-threads?