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, 1) == 0) { // Doesn't block other threads from going ahead to S2
      CS
}
S2
```

A'so. 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);
    }                                          → if 32, none
                             → 2 threads (0/1) all this
    old = atomicCAS (gg, threadIdx.x, -1);       1 --- 31 tids
    if (old == threadIdx.x) {       But ∵ atomic, only one of
                                    them will be able to execute.
        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);   → 2 warps.
    cudaDeviceSynchronize ();
    return 0;
}
```

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

21-2-2020.

Profiling:

- Measuring "indicators" of performance

  * cache hits - timing.
  * extent of parallelism
    - can solve the same problem with multiple algorithms
    - each algorithm can be implemented in diff ways
      eg. recursive, iterative, diff data structures.

1. time taken by diff kernels.
2. Memory utilization -
3. No of cache misses.
4. Degree of divergence.
5. Degree of coalescing.

and so on.

* Intrusive, Non-intrusive profiling

↑ change the program itself to measure some timing/memory-indicator/ performance indicator.

↑ reqd for application-specific indicators

↓ without changing program - hardware counters etc

cuda profiler.

- nvprof - command line - non intrusive ; also includes some API calls that can be added to code (intrusive).
- nvvp, nsight - visual profilers.

An event is a measurable activity on a device. It corresponds to a hardware counter value.

~140 events in cuda

nvprof.

- Non intrusive by default.
- cuda profiler start(), stop() - can be added to profile a part of program.

Q) Which kernel should you optimize?

Kernel calls. ⇒ need to optimize K2.

K2 - atomic instr. more time. → all in μs.

K3 - accessing shared mem ; sync threads is very fast. also, within single warp in each block

K1 - very simple memory accesses, takes less time

time taken decreases

API calls.:

cudaLaunch takes ~ 96% of the time. — few (ms) per kernel launch (1023)

rest of them ⇒ very small fraction of the time.

So our kernel launch is taking far more time than kernel Computation ⇒ kernel should do enough work so that it can compensate launch overhead.

- One optimization is to merge the loops. – "LOOP FUSION"
(though this was not really the bottleneck)
↑ kernel launch

With this: Profiling results ⇒.

$K_2 > K_3 > K_1$.

cudaLaunch also takes similar time.

- One way we can improve the performance is to parallelize the running of the kernels also. → need to learn about streams.

(- Alternate idea

Kernel launch is bottleneck ⇒ reduce # kernel launches
⇒ "KERNEL FUSION".

Single kernel K with all 3 kernels ka functionality.

Profiling results :

Earlier ~ 522 μs for kernels
Now ~ 200 μs for fused kernel.

1
203
177
142
522 μs

But kernel launch overhead ⇒ still the same.
∵ for loop.

→ Change kernel launch config. – no for loop.
~~block size~~. ↑ # blocks.

Again cudaLaunch takes almost same time as before (slightly lesser).

Ahmdahl's law
Sequential bottleneck

1st kernel Launch takes that much time

But. kernel time ⇒ ~ 3 μs.

* cudaDeviceSynchronize ??

→ If we ↑ # threads instead of blocks, Many args:
* No. of warps larger . * less thread blks ⇒ less parallelization etc.

nvprof

- supports device-specific profiling
  remote profiling
- o/ps can be dumped to files a.csv (∵ table)

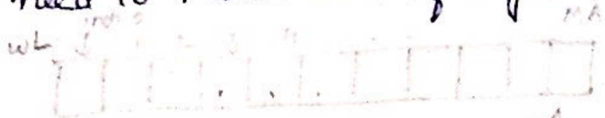
24/2/2020.

eg. set of next nodes to be processed in SSSP/BFS etc
→ nodes where work needs to be done.

CW. Each thread adds elements to a worklist (implemented as an array).
Initially, assume that each thread adds exactly K elements.
Later relax the constraint.

Since threads are adding elements into worklist simultaneously, we need to take care of synchronization.

worklist must be large enough to accommodate max no. of elements



It is possible that some node gets added to worklist twice. ✓

thread (wl) ← contains the k elements.

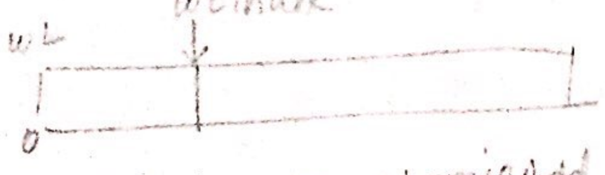
for (i=0; i<k; i++){
oldval = atomicInc(ind);
wl[oldval] = ele[i];
}

[* distance update, worklist update would have to be done atomically (in SSSP algo)]

→ extra calls to atomic insts when compared to atomicAdd.

→ for loop id*k ... (id+1)*k indices of wait list reserved for thread 'id'.

But not all threads may want to add elements.

→ 

oldWLIndex = atomicAdd (&wLindex, K);

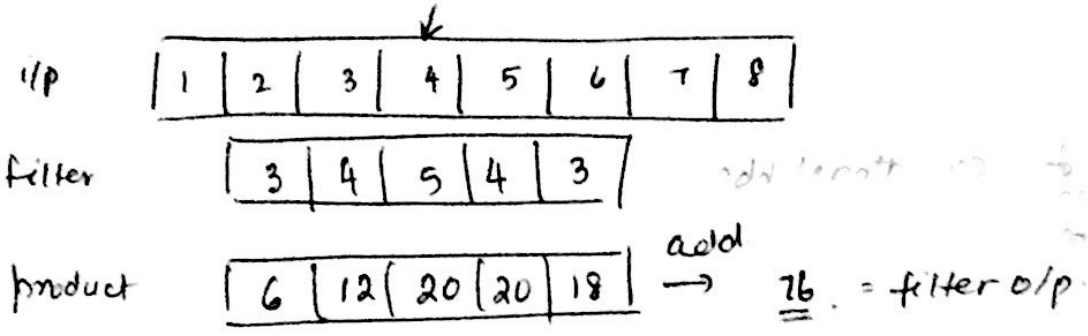
for(i= oldWLIndex ... i= oldWLIndex + k)

wL[i] = ...

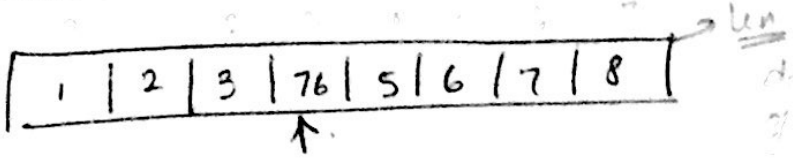
If diff threads want to add diff no. of elements, how do we modify the program?

Convolution filter (Prof Marco Bertini's slides) — used in many graphics filter/processing applications, DSP...

- Each output cell contains weighted sum of input data element & its neighbours.
- The weights are specified as a filter (array).
- The idea can be applied in multiple dimensions.
- We'll work with 1D convolution & odd filter size.



replace element under consideration with filter o/p



29.2.2020.

Calculating sum → using atomic instruction (sequential)
Other parts ⇒ nicely parallel.

can parallelly compute the products, and then sequentially compute the sum.

→ this parallel ~~call to~~ kernel launch can be done from main(), or from inside the kernel itself.

BARRIERS.

- A barrier is a program point where all threads need to reach before any thread proceeds.
- End of kernel is an implicit barrier for all GPU threads (global barrier).
- There is no explicit global barrier supported in CUDA, so far.
- Threads in a thread-block can synchronize using __syncthreads().

If we want to simulate something like this, we need to use 2 different kernels & after 1st kernel, need to store the transient data in global memory before calling 2nd kernel.

- Warp threads → natural barrier blw each instruction.

Note __syncthreads() does not synchronize blw threads of different thread blocks.

- __syncthreads() is not only about control synchronization, but also data synchronization.

~~[illegible] threads~~

- <u>Memory fence operation</u> – ensures that writes from a thread are made visible to other threads.

- __syncthreads() executes a fence for all block-threads.

- There is a separate <u>__threadfence_block()</u> ~~[illegible]~~ instruction also. Then, there is a <u>__threadfence()</u>.

→ <u>'volatile'</u> works with both reads & writes : as per documentation.

- In general, a fence does not ensure that other threads will read the updated value.
  - this can happen due to caching.
  - the other thread needs to use <u>volatile</u> data.

- In CUDA,

<u>eg</u>. Write a CUDA kernel to find <u>maximum</u> over a set of elements and then let thread 0 print the value in the same kernel. // or – use 2 kernels, if # threads > 1024 more than 1 thread blk?

```
__global__ void dkernel (int* arr, int N) {
    __shared__ int max;
    int id = . . . ;
    atomicMax (&max, arr[id]); // has an implicit memory fence.
    __syncthreads(); // assuming single-thread block
    if (id == 0) printf("max = %d\n", max);
```

* Need to have a threadfence() only if we are using direct read/write to/from a variable (∵ caching optimizations are possible).
  → In that case, volatile has to be used.
* Atomic instructions → no need for explicit threadfence().
  They anyway follow memory fence semantics.
* Thread fence is a memory fence operation. It does not ensure that whatever update you are doing is atomic.
  But atomic instructions involve thread fence.

CW. Each thread is given work[id] amount of work. Find average amount of work per thread. If a thread's work is above (average + K), push the extra work to a worklist (array of sufficiently large size). Then, threads that have less work, & finish earlier, will take up some work from the work list → "Work Donation". – useful for load balancing

```
__global__ void dkernel(        )
    __shared__ ... sum;
    __shared__ float avg;
    sum = 0;
    int uid = ....
    atomicAdd(&sum, work[uid]);
    __syncthreads(); // assuming single thread block
    avg = sum/N;
    if (work[uid] > avg + K) {
        // add to work list → as we done before.
    }
    // process my items.
    // →
```

→ // until donation list is empty:
- remove item from list
- process that item

```
__shared__ min;
min = 0;
__threadfence_block();
... = min; // read min
```

If one thread writes to min, will the min value be reflected when another thread reads ?
No.

tf_block: only assures memory fence.
no barrier; no syncthreads
__syncthreads() :: has tf_block internally

On GPU: working with linked list based data structures (like stack / queue based on linked list) is not easy.
Copying GPU ↔ GPU costly.
It's best to work with arrays only.

If we use a data structure that uses offset-based indexing, then it is okay.

Global x

One of the threads: x = blockIdx.x; (not atomic) — can be overwritten.
__syncthreads();

Some other thread reads x:

- within same thread block, x is reflected.
- across thread blocks, x is not reflected

26/2/2020.

Donation List example

Assuming only 1 thread block.

Pseudocode.

// calculate avg

// if my work > avg + k
    donate.

// process my items.

// until donation box is empty
    remove item from box
    process that item.

int dboxind → init to zero.

donation box ⇒ dbox [large enough array]

```
__shared__ int sum;
__shared__ float avg;
sum = 0;
__syncthreads();
int uid = ...
atomicAdd( &sum, work[uid] );
__syncthreads();
if (uid == 0) {
    avg = sum / blockDim.x;
}
__syncthreads();
if (work[uid] > avg + k) {
    oldInd = atomicAdd
        (&dboxind, work[uid]
            - avg - k)
    k = 0;
    for (i = oldInd; i < work[uid] -
            avg - k; i++) {      // sequential processing
        dbox[i] = items[uid][k];
        k++;
    }
}
// no need for barrier here
// Purpose of donating work is defeated if
   we put __syncthreads() here
process (items[uid]);
// If I reach here & see dbox not empty,
   I can help. - no need of syncthreads()
while (dboxind < N * ELEPERTHREAD)
    oldval = atomicDEC(&dboxind, N * ELEPERTHREAD);
    process ( dbox[oldval] );
}
```

—— end of kernel ——

* Put syncthreads() outside if blocks etc – possibility of deadlocks.

Small caveat: But it may happen that fast threads may reach here, and no item has been added to dbox yet. So the fast threads would just finish.

④ atomicInc(&x, N): 

⑤ atomicDec(&x, N): 

Two threads see dbox_ind = 1 at same time and do atomicDec,

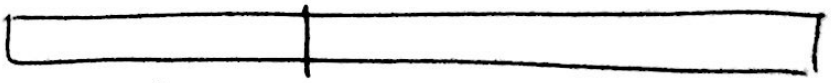
one gets 0, other gets N-1.

↓ this thread comes out of loop.

↘ continues to loop through (will come out hopefully after some time)

How to take care of this problem?

→ valid range + invalid range.



#threads * EPT. blockDim.x threads.

⑦ It could happen that dbox index has been incremented, but the elements have not been added yet. ⇒ data race ...?

How to fix this? ① syncthreads : but purpose of work donation is defeated

Taxonomy of Synchronization Primitives.

Primitive	Control Synchronization (→ whether threads are going to wait)	Data Synchroniz (→ visibility across threa
1) --syncthreads()	block	block.
2) atomic (internally)	--	block for share all for global
3) --threadfence_block()	--	~~block~~ all
4) --threadfence()	--	all (could be more - but not practically useful)
5) global barries (simulated)	all	-- (but not useful with data synchronization)
6) while loop	customizable (using warpid/ threadid etc)	all.
7) volotile → similar to --threadfence(), but for a specific variable.	--	

Control flow	Data flow.
x = 5; ↓ y = 6;	a = 5, → y = x;

29.2.2020

Reductions.

- Converting a set of values to few values (typically 1).
- Computation must be reducible.
  - * Must satisfy associativity property.

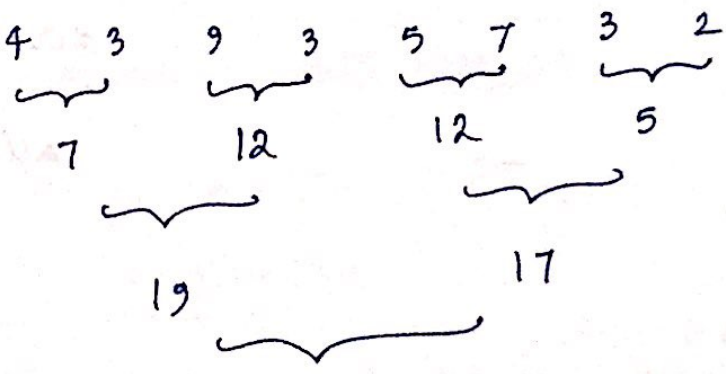
$$(a \cdot (b \cdot c)) = ((a \cdot b) \cdot c)$$

eg. min, max, sum, xor, etc.

→ need not be restricted to primitive types

eg. sum over strings - Concatenate

- complexity measures.



n numbers
→ $n/2$ threads.
operate parallelly
O(1) ops.

$n/4$ threads.

$n/8$ threads.

$\log_2 n$ "steps" ← thread synchronization.
barriers.

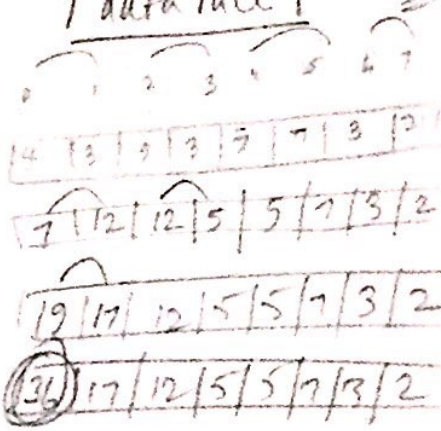
We'll look at within a thread block mostly.

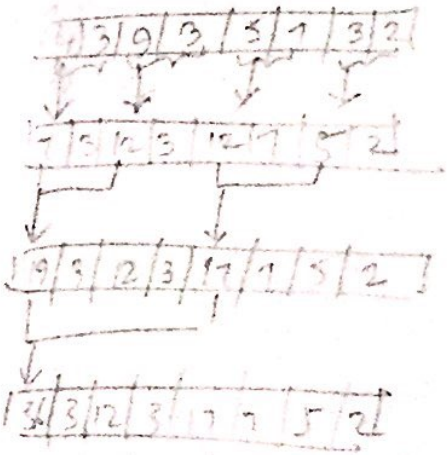
⇒ barriers implemented using __syncthreads().

CH. Implement this reduction code, assume 1 thread block, assume # input elements is a power of 2

assume

```
__global__ void reduceSum(int N, int *arr) {
// assume that we perform in-situ ops on input
int tid = threadIdx.x ...;
int i, numSteps = ...
                  ↑
                  log2 N

for (i = 2; i <= N; i *= 2) {        2, 4, 8 → ②
    arrUpperLimit = N/i;

    if (threadId < arrUpperLimit) {
        atomicAdd(&arr[threadId], arr[threadId + ...]);
        can be avoided as well.
        not read.
    }

    __syncthreads();
}

// sum is in arr[0]
}
```





```
for (int off = N/2; off > 0; off /= 2) {
    if (threadId < off) {
        a[threadIdx] += a[threadIdx + off];
    }
    __syncthreads();
}
```

→ avoids atomic

→ 2 syncthreads.
separate reads & writes.

cw.

Assuming that each a[i] is a character, find a concatenated strings using reduction.

* String concatenation cannot be done using a[i] and a[i + n/2]. ∵ not commutative
(sum - commutative as well)

Assume that string Concat of 2 strings is implemented using a for loop.