Intro

- lessons related to my graduation project guided by xww
- bilibili links: https://www.bilibili.com/video/BV1yt411w7h8
- cuda c is used
- **thinking in a parallel way is important**

1 GPU Programming Model

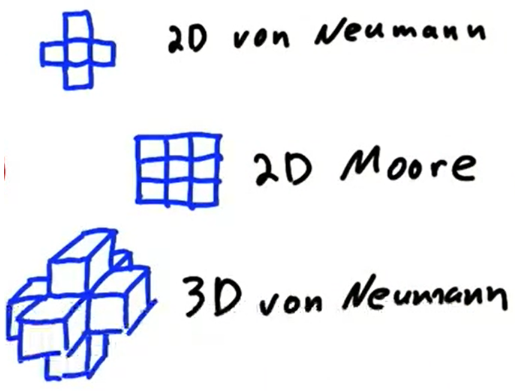
- How to be faster?
  - faster clock
  - more work per clock cycle
  - parallel computing
- 2 kinds of PROCESSORS
  - CPU - complex control hardware
    - flexibility & performance
    - expensive in terms of power
    - more for control
  - GPU - simpler control hardware
    - more hardware for computation
    - potentially more power efficient
    - more restrictive programming model
- what can the programming model do
  - send data from CPU memory to GPU memory
  - send data from GPU memory to CPU memory
  - allocate GPU memory space
  - CPU launches a kernel on GPU
  - dynamic parallelism (may not be widely supported): GPU launches a kernel on GPU
  - a typical GPU program
    - `cudaMalloc`: CPU allocates storage on GPU
    - `cudaMemcpy`: CPU copies input data from CPU to GPU
    - `launchKernel`: CPU launches kernels on GPU to process the data
    - `cudaMemcpy`: CPU copies results back to CPU from GPU
    - ...
  - **BIG IDEA**
    - kernels look like serial programs
    - write the program as if it will run on one single thread, but the GPU will run it on many threads according to the number specified
- Square example: `square.cu`
  - compile: `nvcc -o square square.cu`
  - convention:

        - variables for host start with `h_`
        - variables for device start with `d_`
    - launch a kernel: `square<<<blocks, threads per block[, shared memory]>>>`
        - GPU can run many blocks at once
        - maximum number of threads per block
            - older: 512
            - newer: 1024
        - optional: blocks and threads per block can be dimensional: `dim3(w, 1, 1) == dim3(w) == w`
        - optional: stream
        - how to set arguments

2 GPU Hardware and Parallel Communication Patterns

- communication takes place through memory

Parallel Communication Patterns

- map: tasks read from and write to specific data elements - **one input for one output**
    - square
- gather: each calculation gathers input data from different places together - **many input for one output**
    - pooling
- scatter: tasks compute where to write output - **one input for many output**
    - map & scatter
- stencil: tasks read input from a fixed neighborhood in an array and much data reuse - **several input for one output**
    - a special kind of gather
    - 

- transpose: reorder the data in memory - **one input for one output**
    - a special kind of scatter

GPU Hardware

- GPU

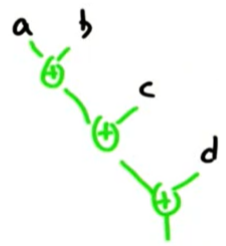
    - multiple SMs: streaming multiprocessor and one block is allocated to one SM, but one SM may deal with several blocks, which depends on the available resources
    - a GPU may have multiple SMs
        - a SM has multiple streaming processors, which is responsible for a thread
        - memory
            - local memory: accessible to a specific thread
            - shared memory: accessible to a thread block
            - global memory: accessible to all kernels
            - host/CPU memory
    - different thread blocks should not cooperate
- synchronization
    - barrier: point in the program where threads stop and wait. When all threads have reached the barrier, they can proceed
- CUDA makes few guarantees about when and where thread blocks will run
    - pros
        - hard ware can run things efficiently
        - no waiting on slowpokes
        - scalability
    - cons
        - no assumptions for block execution sequence
        - no communication between blocks - dead lock
- CUDA guarantees
    - all threads in a block run on the same SM at the same time
    - all blocks in a kernel finish before any blocks from the next kernel run
- How to write efficient program from high level
    - maximize arithmetic intensity $\frac{math}{memory}$
        - reduce the time waiting for operands
        - maximize compute ops per thread
        - minimize time spent on memory per thread
            - speed: local memory > shared memory >> global memory >> CPU memory
            - coalesce global memory accessed
                - speed: coalesced > strided > random
    - avoid thread divergence - `if`, `switch`
- atomic operation
    - Problem: multiple threads read/write the same memory location
    - cons
        - only certain operations and data types are supported
            - can work around using atomic comparison and swap
        - still no ordering constraints
            - floating-point arithmetic is non-associative
        - serializes access to memory
            - slow

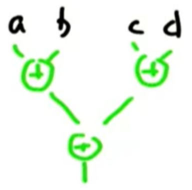
3 Fundamental GPU Algorithms

Analyze

- step complexity
  - the depth of the workflow/tree
- work complexity
  - total number of nodes of the workflow/tree

Reduce

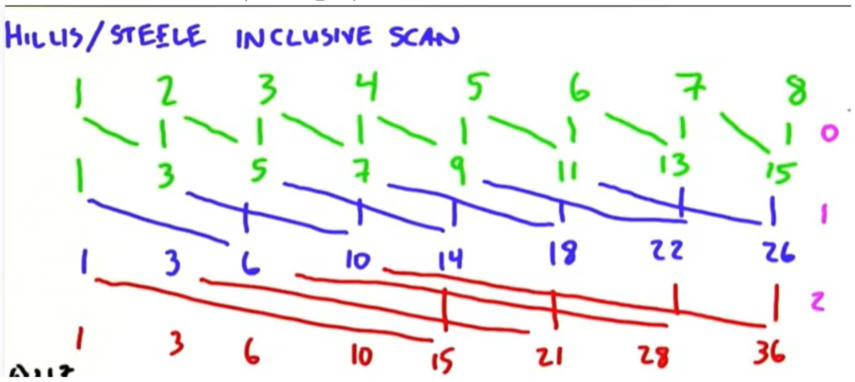
- inputs
  - set of elements
  - binary & associative reduction operator
- serial implementation
  - each iteration is dependent on the previous iteration
  - 

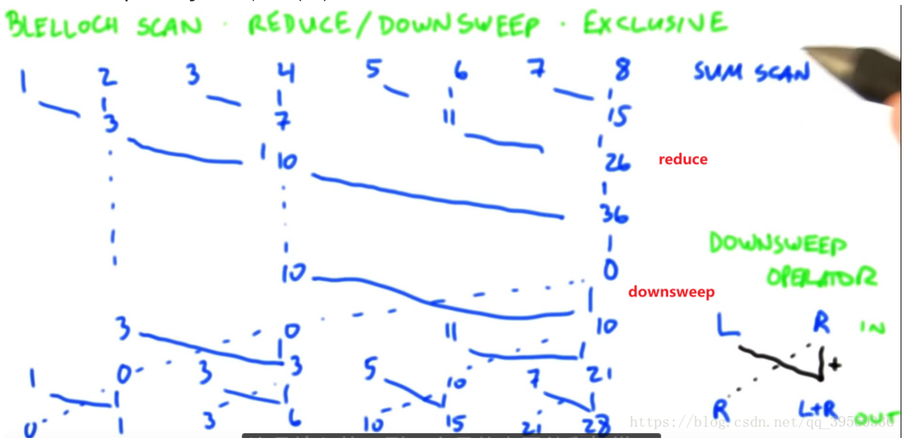
  - step complexity: $n - 1/O(n)$
  - work complexity: $n - 1/O(n)$
- parallel reduce
  - 

  - step complexity: $O(\log_2 n)$
  - work complexity: $n - 1/O(n)$
  - [Brent Theorem]
- see `reduce.cu`

Scan

- address set of problems otherwise difficult to parallelize
- not useful in serial world but very useful in parallel world
- inputs
  - set of elements
  - binary & associative scan operator
  - identity element I (for a certain operator) with property [I op a = a]
- 2 kinds:
  - exclusive scan: all elements before without the current element

        - result starts with identity element
        - the last element is ignored
    - inclusive scan: all elements before with the current element
        - result starts with the first element
        - the last element is used
- serial implementation
    - step complexity: $n/O(n)$
    - work complexity: $n/O(n)$
- *Hillis & Steele's scan*
    - inclusive scan
    - more step-efficient
    - step complexity: $\log n/O(\log_2 n)$
    - work complexity: $O(n\log_2 n)$
    - 

- *Blelloch scan*
    - exclusive scan
    - more work-efficient
    - step complexity: $2\log n/O(\log_2 n)$
    - work complexity: $2n/O(n)$
    - 

- which to choose
    - more work than processors: processors limited the performance, so prefer the work-efficient one
    - more processors than work: steps limited the performance, so prefer the step-efficient one

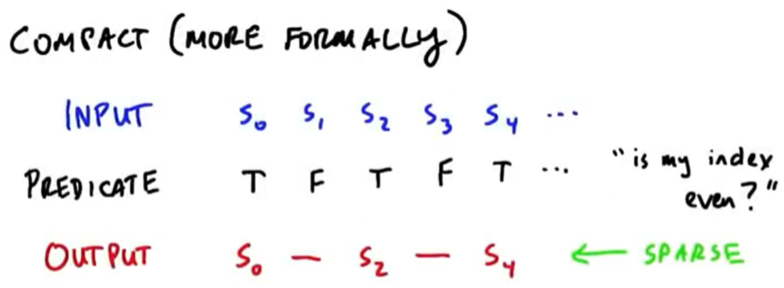
Histogram

- divide the array into several bins for statistics
- cumulative distribution function
    - input: histogram

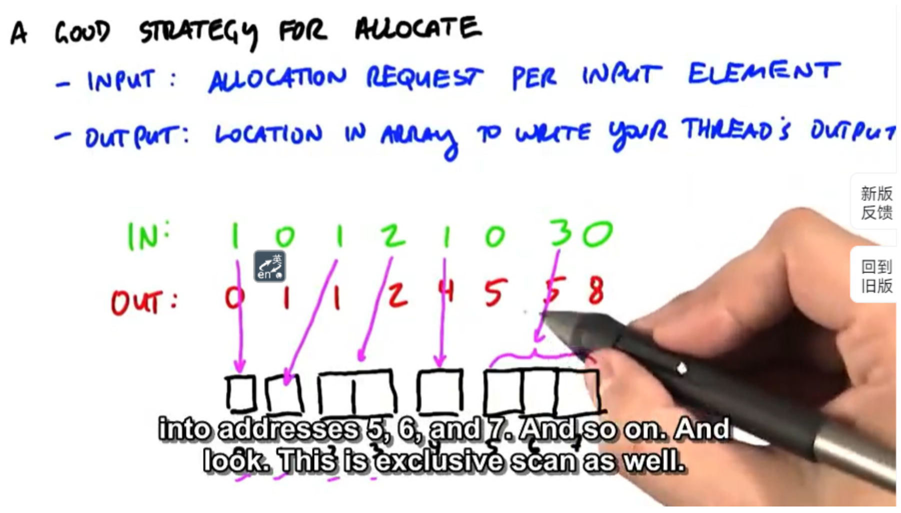
- output: cumulative distribution function
- operation: exclusive scan

- problems: synchronization
  - atomic operation(addition)
    - slow
  - produce per-thread local histograms and then reduce
    - no need for atomic operations
  - sort the data and then reduce by key

Compact / Filter



- dense computation (usually with `map()`) is better than sparse computation (usually with `if`), since less threads are required and less idle threads are generated
  - clipping in graphics pipeline
  - matrix multiplication
- compact algorithm
  - predicate
  - scan: produce 1 for TRUE and produce 0 for FALSE
  - exclusive-sum-scan: produce addresses
  - scatter input into output using addresses
- a good strategy for allocation (in step scatter)
  - input: allocation request per input element
  - output: location in array to write the thread's output
  - example of clipping


A GOOD STRATEGY FOR ALLOCATE
- INPUT: ALLOCATION REQUEST PER INPUT ELEMENT
- OUTPUT: LOCATION IN ARRAY TO WRITE YOUR THREAD'S OUTPUT
IN: 1 0 1 2 1 0 3 0
OUT: 0 1 1 2 4 5 5 8
into addresses 5, 6, and 7. And so on. And look. This is exclusive scan as well.
新版 反馈
回到 旧版

- segmented scan
  - group the large scan into many small scans
    - each is launched independently

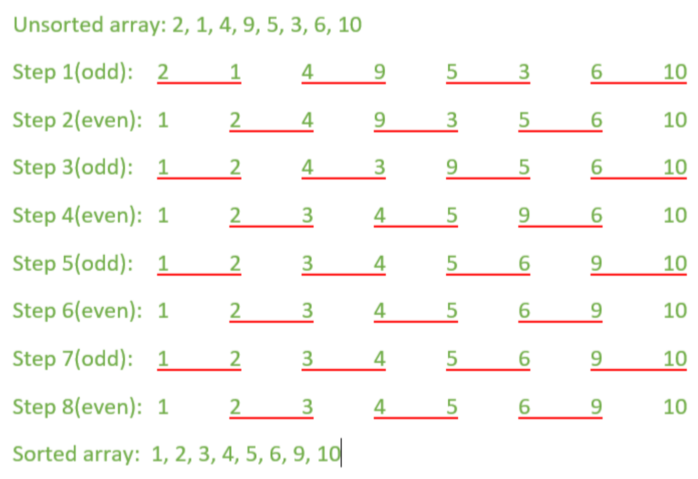
        - combined as segments
    - application: multiplication of matrix and vector
        - CSR (Compressed Sparse Row) representation of sparse matrix
            - value vector
            - column vector
            - rowptr vector (index of column vector)

            > not all the elements in a row are 0?

        - steps:
            - create segmented representation of value vector from rowptr vector
            - gather vector values using column
            - pairwise multiply the above vector
            - (backward) inclusive segmented sum scan

Sort

brick sort & merge sort

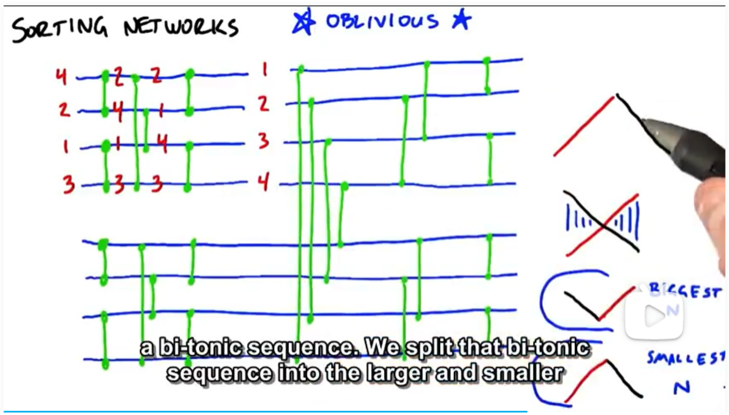
- odd-even sort (brick sort)
    - 

      Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10

      ```
      Step 1(odd):   2  1  4  9  5  3  6  10
      Step 2(even):  1  2  4  9  3  5  6  10
      Step 3(odd):   1  2  4  3  9  5  6  10
      Step 4(even):  1  2  3  4  5  9  6  10
      Step 5(odd):   1  2  3  4  5  6  9  10
      Step 6(even):  1  2  3  4  5  6  9  10
      Step 7(odd):   1  2  3  4  5  6  9  10
      Step 8(even):  1  2  3  4  5  6  9  10
      ```

      Sorted array: 1, 2, 3, 4, 5, 6, 9, 10
    - step complexity: $O(n)$
    - work complexity: $O(n^2)$
- parallel merge sort
    - parallel merge
        - each thread deals with one element
        - use the index in the list and binary search result in the other list to obtain the final index
    - break up the last big merge into smaller ones
        - otherwise many SMs are idle!
        - choose splitters and merge the small list between
    - step complexity: $O(n \log n)$
    - work complexity: $O(n^2)$

sorting networks

- a kind of data-independent sort
- bitonic sequence:
  - only one extreme point & left side is non-strictly monotonically decreasing & right side is non-strictly monotonically increasing
  - sequence that can be rotated to satisfy the above requirement
- bitonic sort
  - basic operation: `min` + `max` / `swap`
  - one of the fastest and relatively simple
  - steps:
    - split the bitonic sequence into 2 halves
    - cross the two halves and do pairwise comparisons, so that the biggest elements form a new bitonic sequence and the smallest elements form a new bitonic sequence
    - iterate
  - 

    a bi-tonic sequence. We split that bi-tonic sequence into the larger and smaller
  - particularly efficient when the data can fit into the share memory, so sorting network is often used to sort the data in a block together with other sorting algorithms sorting the blocks
  - step complexity: $O(\log^2 n)$
  - work complexity: $O(n \log^2 n)$

radix sort

- GPU performance leader
- only for integer
- implemented by scans
- steps
  - start with LSB
  - split the input into 2 sets based on bit. Otherwise preserve order (compact & scatter)
  - move to the next LSB, repeat
  - to reduce the passes: testing multiple bits in a pass
- work complexity: $O(kn)$
  - k: bits in representation

others

- quick sort
    - one of the fastest in serial world
    - steps
        - choose pivot element
        - compare all elements vs. pivot
        - split into 3 arrays: <P, =P and >P
        - recurse on each array
    - implement
        - comparison & split: compact
        - recurse: segment
        - there is no recursion in GPU
        - hard to implement
- key value sort
    - sort the pointer rather than the complex data structure
    - move the key and the value together
    - most GPUs handle 64-bit data, it’s good to use a 32-bit key and a 32-bit value

4 Optimizing GPU Algorithms

Levels of optimization:

1. picking good algorithms
    - 3-10x speedup
2. basic principles for efficiency
    - 3-10x speedup
3. architecture-specific detailed optimization
    - 30%-80% speedup
4. μ-optimization of instruction level

APOD - systematic optimization

- analyze
- parallelize
- optimize
- deploy

analyze

- understanding hotspots
    - do not rely on intuition
    - run a profiler to know which to parallelize
    - *Amdahl’s law* to know the speedup limit: $S = \frac{1}{(1-P)+\frac{P}{n}}$
- understanding strong scaling vs. weak scaling

    - weak scaling: run a bigger problem or more small problems in parallel
    - strong scaling: run a problem faster
- time limits
    - memory access
        - use `CUDA Device Query` to check the information of the device
        - 40-60 usage: okay
        - 60-75 usage: pretty well
        - >75 usage: excellent
        - `GB/s` is usually used
    - compute performance

parallelize

> see `transpose.cu`

- maximum parallelism does not always bring the best performance -> granularity coarsening
- v1: thread per matrix
- v2: thread per row
- v3: thread per element
- v4: thread per element with tiling
    - consider memory access limit and set `K=32`
- v5: thread per element with tiling and `K=16`
    - consider occupancy
- v6: consider shared memory bank conflicts & padding

optimize

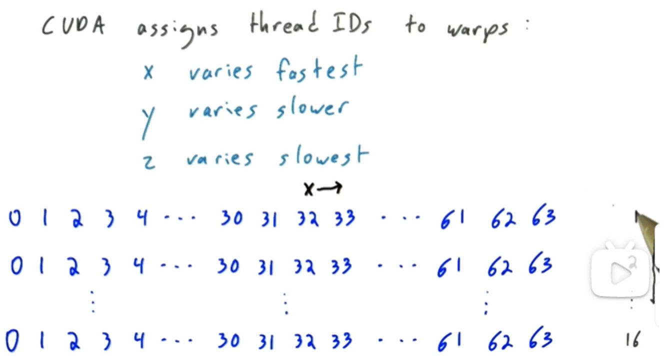
> see `transpose.cu`

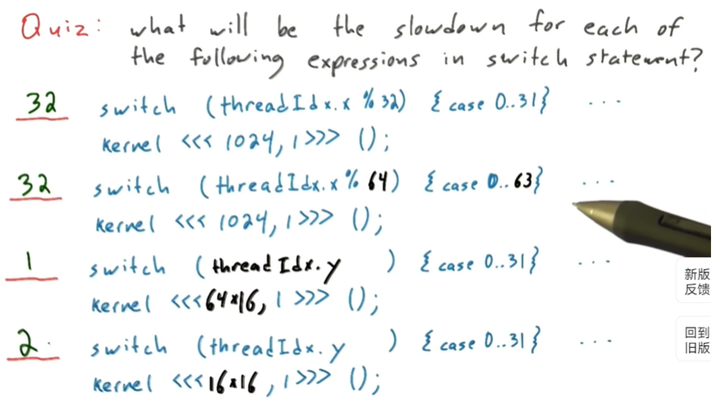
For code:

- most GPU codes are memory limited
    - tools to visualize performance: nSight / NVVP
    - parallel per element version only achieves 31.1% peak memory bandwidth, so it needs to make use of coalescing
        - writing `out` in `out[j + i*N] = in[i + j*N];` is bad
        - method: each tile is transposed in shared memory to achieve coalescing writes
    - *Little's Law*: the number of bytes delivered = the average latency of each memory transaction $\times$ the bandwidth
        - increase the number of bytes delivered or decrease the latency to increase the bandwidth
        - the synchronization limits the latency, so reduce `K` to 16 helps to increase efficiency
    - occupancy: usage of the resources on a SM
        - increasing occupancy usually helps to a point
        - methods:
            - control amount of shared memory can affect occupancy

            - change the blocks and threads in `<<< , >>>`
            - compilation options to control register usage
    - further optimization: [shared memory bank conflicts](#) & padding
- optimize compute performance
    - minimize time waiting at barriers
    - minimize thread divergence
        - [warp](#): set of threads (32 threads) that execute the same instruction at the same time

            > warp is the basic unit of scheduling and operation and not all the threads are executed at the same time

        - CPU - SIMD: SSE/AVX
        - GPU - [SIMT](#)
            - be aware of branch divergence
                - avoid branchy code
                - beware large imbalance in thread workloads
            - switch
                - 
                  CUDA assigns thread IDs to warps:
                  x varies fastest
                  y varies slower
                  z varies slowest

                - 
                  Quiz: what will be the slowdown for each of the following expressions in switch statement?
                  ```
                  32  switch (threadIdx.x % 32) { case 0..31} ...
                      kernel <<<1024,1>>> ();
                  32  switch (threadIdx.x % 64) { case 0..63} ...
                      kernel <<<1024,1>>> ();
                  1   switch (threadIdx.y    ) { case 0..31} ...
                      kernel <<<64x16, 1>>> ();
                  2   switch (threadIdx.y    ) { case 0..31} ...
                      kernel <<<16x16, 1>>> ();
                  ```

                > the threads in a warp should belong to the same block?

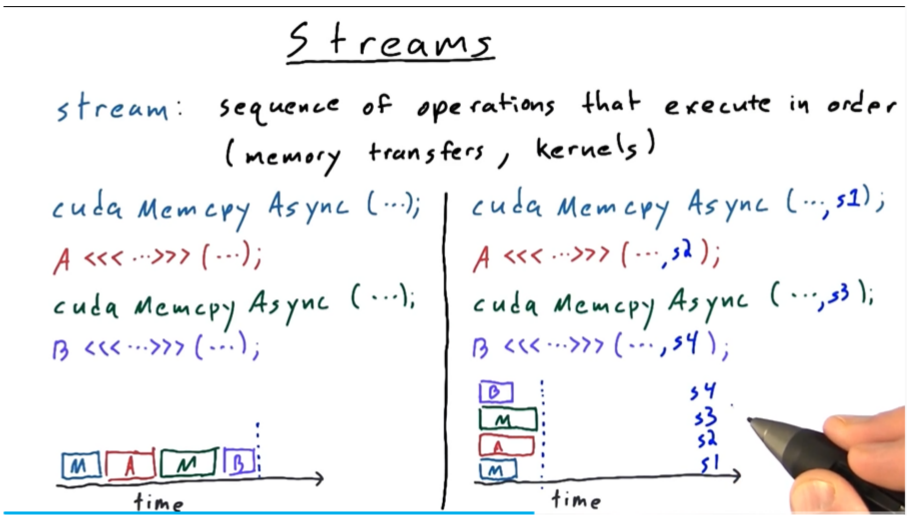
            - loops
    - assorted math optimizations
        - use double precision only when you mean it
        - use intrinsics

For system:

- host and device communicate through PCI/PCIe

- use `cudaHostMalloc()` or `cudaHostRegister()` to avoid copy to staging area of PCI (PCIe can only transfer data in staging area)
    - enables asynchronization `cudaMemcpyAsync()`
- streams: sequence of operations that execute in order
    - each operation/instruction can be assigned a stream

      ```
      cudaStream_t s1;
      cudaStreamCreate(&s1);
      cudaStreamDestroy(s1);
      ```

    - one fashion to use:
        - break the data that fits the GPU memory into 2 halves
        - transfer one half and begin the next transfer while beginning processing the ready half
    - overlap memory transfers & compute
    - help to fill GPU with smaller kernels
    - 

5 Parallel Computing Patterns

N-body Computation

Problems:

- all-pairs N-body
    - simplest
    - brute-force: $O(N^2)$
    - tree methods (*Barnes-Hut*): $O(N \log N)$
    - fast multipole method: $O(N)$
- target: to obtain the force on each object, and move each object accordingly to simulate

Solutions:

> see `nbody.cu`

- NxN matrix

    - x-axis: source object
    - y-axis: destination object
- better: tiling
    - PxP tiles
    - calculate each tile and sum up the horizontal tiles
    - one thread for one destination object in a tile
        - still must share source object parameters across all threads
        - no sharing necessity for destination objects
        - no communication necessity to sum up results
        - reduce parallelism actually
    - to obtain best efficiency, the size of the tile should be chosen carefully
    - **why to make tradeoff between more parallelism and more work per thread?**
        - the communication cost with a thread is much smaller than the cost between threads
        - more parallelism may bring more communication and more synchronization

Sparse Matrix-Vector Multiply

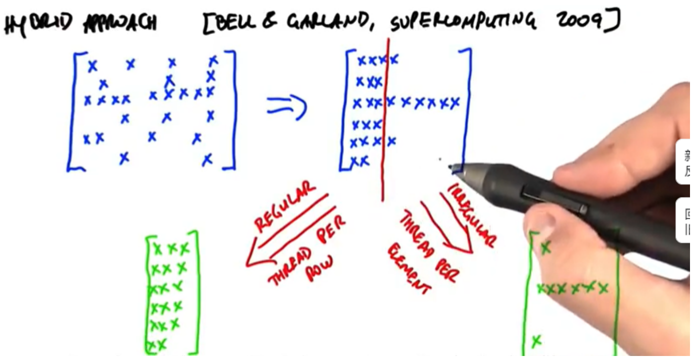
Problems:

- multiply a sparse matrix by a dense vector
    - to use less memory
    - to use less computation
- what is the role of a thread?
    - thread per row
        - calculate one entry of the result
        - more work per thread
    - thread per element
        - calculate one partial product of one entry
        - more threads
        - communication between threads
    - **again to make tradeoff between more parallelism and more work per thread**

Solutions:

> see `spmv.cu`

- thread per row
    - the lengths of rows may be different and the run time depends on the longest row
    - better when the lengths of rows are similar
- thread per element
    - run time depends only on the number of non-zero elements
    - better when the lengths are varying
- hybrid approach
    -



- how to draw the line dividing the matrix?
    - keep threads busy
    - closer communication is cheaper

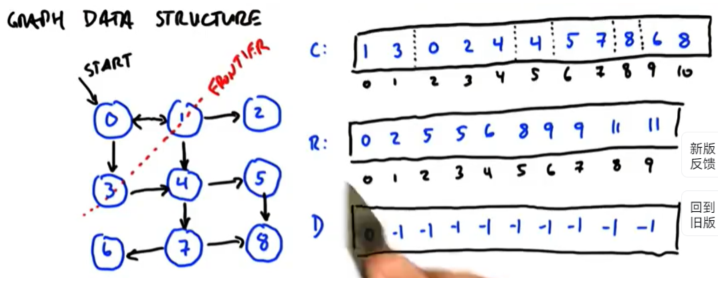
BFS

Problems:

- implement BFS on graphs in parallel
- calculating distance from root for each node

Solutions:

> see `BFS.cu`

- simple version with no frontier
    - steps:
        - in each iteration, visit all the edges
        - if a vertex is found unvisited, set its depth
        - iterate V times at most
    - what if a vertex is set multiple times in one iteration: it's OK since the value to be set is the same
    - when to terminate: when no vertex is set in one iteration
    - work complexity: $O(VE)$
        - consider a linear chain
        - usually in a graph edges are far more than vertices, so the work complexity is at least $O(V^2)$
        - weak scalability
- with a frontier
    - 

        - `C` storing the neighbors, whose length is $O(E)$
        - `R` storing the start offset of neighbors of each vertex, whose length is $O(V)$
        - `D` storing the depth
    - steps:

        - in parallel, for each vertex on the frontier, find the starting point of its neighbors
        - for each vertex on the frontier, calculate the number of its neighbors by `R[v+1] - R[v]`
        - allocate space to store the new frontier
        - copy each active edge list to the allocated array and the storing location is calculated by an exclusive scan
        - call the list, removing/compact items that have been visited, which can be determined by `D`

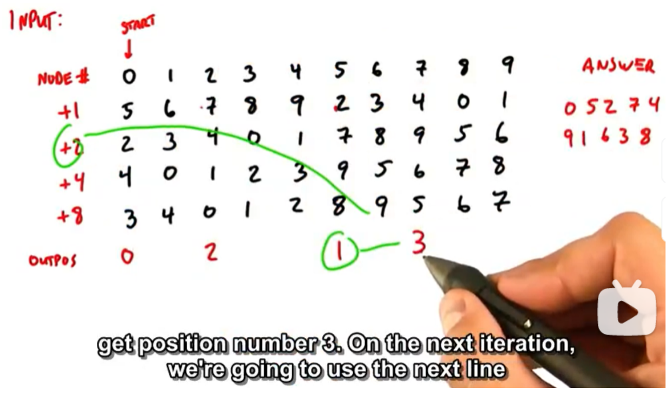
List Ranking

> In a link list, how to set `chum` (a pointer pointing to the last node) ?
>
> in each iteration, use `chum[k] = chum[chum[k]]` instead of `chum[k] = next[k] + 1`

Problems:

- output: all nodes in order
- input
    - each node knowing its successor
    - starting node
    - organized as a link list

Solutions:

- construct a table recording the node k-hop away from each node
    - work complexity: $O(N \log N)$ (more than serial algorithm)
    - step complexity: $O(\log N)$ (less than serial algorithm)
- steps:
    - construct the table
    - scatter and recording output positions
        - in n-th iteration, check the 2^n-hop-away row to activate threads from currently active threads
    - 

NODE #	0	1	2	3	4	5	6	7	8	9
+1	5	6	7	8	9	2	3	4	0	1
+2	2	3	4	0	1	7	8	9	5	6
+4	4	0	1	2	3	9	5	6	7	8
+8	3	4	0	1	2	8	9	5	6	7
OUTPOS	0		2			1		3		

get position number 3. On the next iteration, we're going to use the next line

Hash

Problems:

- chaining is bad for parallelism

    - load imbalance: the lengths may be varying, so some threads may be waiting for a long time
    - contention in construction
- implement parallel hashing

Solutions:

- cuckoo hashing
    - multiple hash tables and one hash function per table
    - steps:
        - all items use h_1 and write into t_1; kick out the existed items if necessary
        - for those failed ones, use h_2
        - repeat
        - if there are too many iterations, choose new hash functions and start over
    - implementation
        - constant time lookup
        - construction is simple & fast
            - kick out should be atomic
        - alternative algorithm
            - construction: sort
            - lookup: binary search

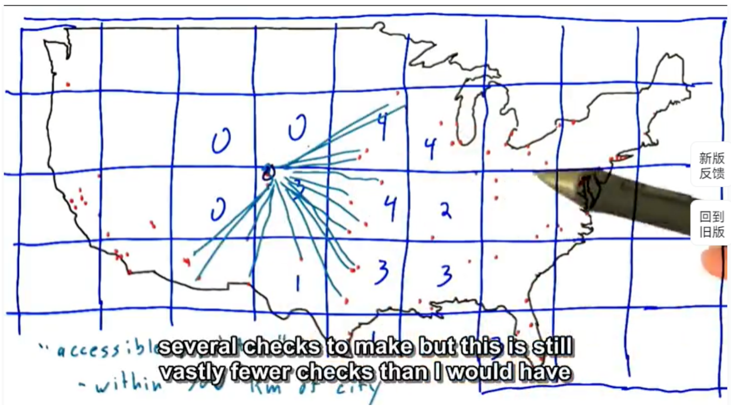
6 Additional Parallel Computing Topics

- reference

Parallel Optimization Patterns

7 basic techniques from Stratton's taxonomy:

1. data layout transformation
    - burst utilization like AoS -> SoA
    - partition camping (bank conflict?)
    - arrays of structures of tiled arrays (ASTA)
2. scatter-to-gather transformation
    - gather: many overlapping reads
    - scatter: many potentially conflicting writes
3. tiling
    - buffering date into fast on-chip storage for repeated access/reuses
4. privatization
    - avoid different threads/blocks writing to the same output location
5. binning/spatial data structure
    - binning: build data structure that maps output locations to (a small subset of) the relevant input data
    - group the input into bins, and only consider relevant bins when calculating

- 

  several checks to make but this is still vastly fewer checks than I would have

6. compaction
    - when most input elements require no computation
    - consider the structure of warps, the speedup is usually at most 32x
7. regularization
    - reorganizing input data to reduce load imbalance

Libraries

- `cuBLAS`
    - cuda Basic Linear Algebra Subroutines
- `cuFFT`
    - 1D, 2D, 3D
    - cuda Fast Fourier Transform
- `cuSPARSE`
    - BLAS-like routines for sparse matrix formats
- `cuRAND`
    - pseudo-random and quasi-random generation routines
- `NPP`
    - low-level image processing primitives
- `Magma`
    - GPU and multiple CPU implementations of LAPACK routines
- `CULA`
    - implementations of Eigensolvers, matrix factorization, and matrix solvers for dense matrices similar to the LAPACK API
- `ArrayFire`
    - framework for data-parallel array manipulation

How to use:

- substitute library calls
- manage data locality
- rebuild and link

CUDA C++ Programming Power Tools

- `Thrust`
    - analogous to STL

    - automatically installed when installing CUDA toolkits
    - CSDN and documents
- `CUB`
    - CUDA Unbound
    - accessing memory can be complicated and this tool helps to manage global memory, shared memory and local variables of threads
- `CUDA DMA`
    - make it easier to use shared memory
    - specifically focus on shared memory and global memory
    - high performance
    - explicit transfer patterns: sequential, strided and indirect

Platforms

Other languages:

- wrap CUDA C++
    - PyCUDA
- target CUDA directly
    - copperhead - data-parallel subset of Python
    - CUDA Fortran
    - Halide - image processing DSL
    - Matlab

        > see `whitebalance.m`

Cross-platform:

- OpenCL
- OpenGL compute
- OpenACC

Dynamic Parallelism

> see `dynamic.cu`

- bulk parallelism
    - every thread is executing in parallel without any dependency
- nested parallelism
    - a kernel launches another kernel
    - watch out
        - multiple launches caused by every thread executing the same launch call
        - each block is independent
        - a block's private/shared data is private and cannot be inherited
    - recursive parallelisms
- task parallelism
    - use multiple tasks to fill the whole GPU

    - multiple tasks operate their own data

> composability: every kernel is independent and does not care about what other kernels are doing