GPU

Parallel processor
- Many cores
- Small memory

memory transfer overhead

-------------------------

Sorting:
Input: Large array $A = \langle a_1, a_2, \ldots, a_n \rangle$
Output $B = \langle b_1, b_2, \ldots, b_n \rangle$
- $\mu(a_i) = b_j$ exists
- $b_j \leq b_{j+1}$

----------------

Data driven sorting?
- insertion sort?
  $O(n^2)$
  (choose one and place in correct spot)
- quick sort?
  $O(n \log n)$
  (need splitter: median hard, otherwise varies size...)
- heap sort?
  $O(n \log n)$
  (need to maintain heap data structure, hard on GPU)
- radix sort?
  $O(nk)$ (for $k$ digit w/ constant bits)
  lengths of each digit category uncontrollable length.

<hard to make highly parallel>

Data Independent sorting
- bubble sort?
  $O(n^2)$
  (compare all neighbors)
  very parallelizable, but takes $n$ rounds to move point from 1 to $n$
- merge sort?
  $O(n \log n)$
  (divide + conquer + join)
  join step very sequential :( 
- bitonic sort
  (divide + conquer + join)
  join step parallel !!!

<will also hybridize merge+bubble...>
Bitonic Sort:

Bitonic sequence:
- increasing, 1 2 4 6 8 11
- decreasing, 9 7 4 3 2 1
- increasing then decreasing, or 1 4 6 9 3 2
- decreasing then increasing. 9 5 2 3 4 6
(at most one local maxima/minima)

BitonicSplit(A):
Input: 1 bitonic sequence A size n
Output: 1 increasing (sorted) sequence B size n

for h = log n to 1
  for i = 1 to n/2^h PARDO
    for j = 0 to 2^{h-1} PARDO
      min(A[i + (2j)*(n/2^h)], A[i + (2j+1)(n/2^h)]) -> B[i + (2j)*(n/2^h)]
      max(A[i + (2j)*(n/2^h)], A[i + (2j+1)(n/2^h)]) -> B[i + (2j+1)(n/2^h)]

Example:
24 20 15 9 4 2 5 8|10 11 12 13 22 30 32 45
10 11 12 9|4 2 5 8|24 20 15 13|22 30 32 45
4 2|5 8|10 11|12 9|22 20|15 13 24 30 32 45
4 2|5 8|10 9|12 11|15 13 22 20|24 30 32 45
2 4 5 8 9 10 11 12 13 15 20 22 24 30 32 45

How to get a bitonic sequence?

for h = 1 to log n
  for i = 1 to n/2^h PARDO
    for j = 0 to 2^{h-1} PARDO
      BitonicSplit(A[i + (2j)(n/2^h), i + (2j+2)(n/2^h) - 1]) //(reverse second half)

- sets of size 2 are bitonic
- let S be an ascending sorted set
  let T be a descending sorted set
  S cat T is bitonic
- run bitonic sort of sets of doubled size for log n rounds

-----
BitonicSplit on all pairs -> sort all pairs
BitonicSplit on all quads (reverse second pair) -> sort all quads
...
BitonicSplit on list (reverse second half) -> sorted list
0(log n) rounds of Bitonic split
Each Bitonic split takes O(log n) rounds

O(log^2 n) parallel time
O(n log^2 n) work

Fine-grain parallelism:
- core of each operation is a compare/swap.
- data independent

For several years, this was fastest GPU sort!
What are the weak points of this?
How can it be improved?

---------------------------------------------

Hybrid (bucket/quick + merge sort)

Sintorn + Assarsson 08
(beats bitonic by factor 2-3)
takes advantage of advanced architecture of GPU (GeForce 8800)

1. Create L sub-lists using L-1 {l_1,l_2,...l_{L-1}} pivots
   so p in L_i has l_i < p <= l_{i+1}
2. Move each L_i to separate processor group
3. Merge Sort on each list L_i

details:
(1) three proposed methods:
   (a) bucket sort (two-rounds)
      i : choose L-1 pivots by linear interpolation [min,max]
      (random sample may work better, distribution independent)
      ii : build histogram w/ AtomicInc on buckets
      iii: re-linear interpolate based on histogram
      (again I think random sample may work better, more general)
   (b) Use NVidia histogram functionality to help w/ splits.
   (c) Run log(L) rounds of quick sort by choosing random pivots

   (d) other option: run multi-selection sort we discussed in class
        or just log(L) median operations in O(N) time each

Note: assigning a point p to a pivot can be done in parallel, but
takes O(log L) (binary search on {l_i}_i). Perhaps can be done
quicker with clever bit-shifting....

(2) Use local hierarchy of GPU to move to sub-hierarchies on GPU each
L of roughly the same size.
Importance of same size, otherwise, when last is running, others will
be idle.
(3)
1. break to sets of size 4
2. run special "kernel" to sort sets of size 4
3. merge pairs of sets
   (for most of run, many more sets than processors, so highly parallel)
4. eventually p processors in group, and < p lists left to merge
   (lose some parallelism, but oh, well, did pretty well).

Work = O(n \log n)
PTime:
(1) = O(\log L)
(a) 2 rounds of O(\log L) time to assign
(b) \log L rounds of finding median (and counting)
   * O(\log n \log \log n) to find median
   but heuristic (random split) only takes O(1)/round
(2) = O(\log L) (each list of size roughly N/L) (but could be N !)
(3) = O(n/L) since last round one 1 processor needs to run a merge on two lists.
   = O(n/L + \log L) optimal for L = n --> (\log n)
   but that requires (1) to complete sort! ...L restricted by num processors

Odd-Even Transition Merge Sort:

-----
Odd-Even Transition Sort:
for h = 1 to n/2
   for i=1 to n/2 PARDO
      min(A[2i-1],A[2i]) -> A[2i-1]
   for i=1 to n/2-1 PARDO

O(n) Ptime, O(n^2) Work

Way to make this
- O(\log^2 n) Ptime
- O(n \log^2 n) Work
- fine-grained
- data independent

1. Grow sorted sub-pieces
2. Join takes O(\log m) for sorted sets of size m

"sorting network"