GPU as a Parallel Machine: Sorting on the GPU

CIS 700/010: 3/17/05
Scribed by Joseph T. Kider Jr.

1 .Background

Sorting is a fundamental algorithmic building block. One of the most studied problems in computer science is ordering a list of items efficiently. Buck and Purcell showed how the parallel bitonic merge sort algorithm, could exploit many of the parallel features of the SIMD architecture of the GPU. Efficient sorting has practical importance to optimizing algorithms that require sorted lists to work correctly. Moreover, the cost of reading data back from the GPU to the CPU is incredibly inefficient; sorting the data directly on the GPU is preferable for many graphics algorithms

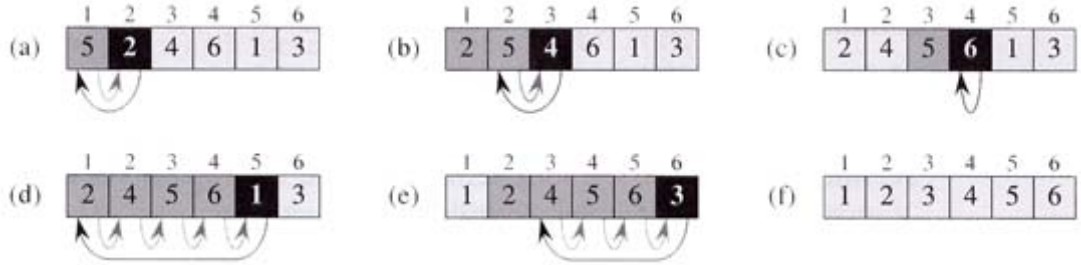
There is a wide variety of needs for sorting algorithms in computer graphics. For example, a program that renders objects according to their depth value so that it can be drawn in the correct order on the screen. The so-called painter's algorithm consists of sorting the object or polygons from back to front and rasterizing them in that order. In Lecture 3 we studied "depth peeling": a way of extracting layers from a scene in depth sorted order. Additionally, particle systems need to be sorted according to viewer distance. The sorting data textures contain the particle-viewer distance and the index of the particle which correctly rendered. In physical simulation, sorting is necessary for inserting the participating objecting into spatial structures for collision detection.

2. Traditional Sorting Algorithms

Sorting algorithms can be divided into two categories: data driven and data-independent ones. First we will briefly review the standard data driven algorithms:

2.1 Insertion Sort

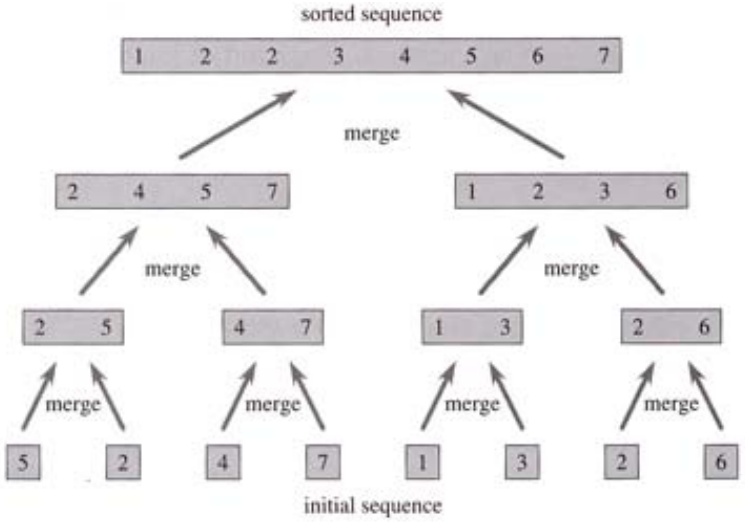
Insertion Sort works the way many people sort a hand of playing cards. The insertion sort works just as the name suggests – it inserts each item into its proper place in the final list. The insertion sort has a running time of $O(n^2)$.



```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

2.2 Merge Sort

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. This algorithm follows the *divide-and-conquer* approach they *divide* the problem into several subproblems that are similar to the original problem but smaller size, *solve* the subproblems recursively, and then *combine* these solutions to create a solution to the original problem.



We can derive the following recrance for the worst case running time T(n) of merge sort:

$T(n) = \theta(1)$ if n=1
$= 2T(n/2) + \theta(n)$ if n>1

Using the second case of the Master theorem we derive the merge sort has an algorithmic complexity of $O(n \log n)$. A major drawback to this approach is at least twice the memory requirements of the other sorts. Merge sort does not sort 'in place', like insertion or quick sort.

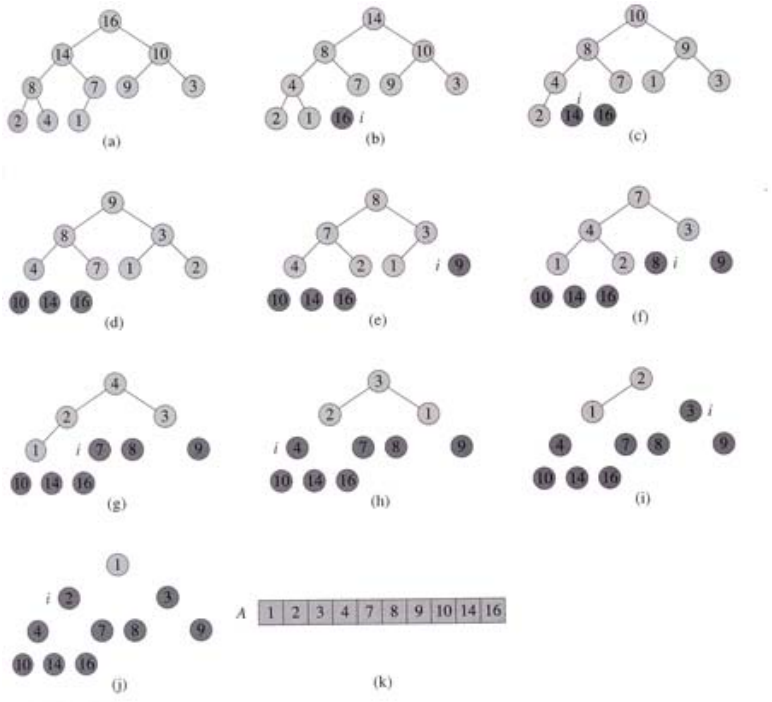
2.3 Bubble Sort

The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. General-case is bubble sort has a running time of an abysmal $O(n^2)$.

2.4 Heap Sort

The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

1. construct a heap,
2. add each item to it (maintaining the heap property!),
3. when all items have been added, remove them one by one (restoring the heap property as each one is removed).

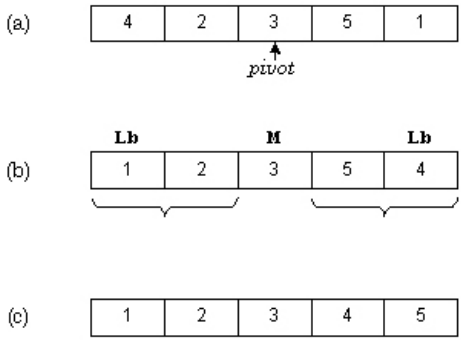


In-place and non-recursive, making it a good choice for extremely large data sets. Addition and deletion are both O(log n) operations. There are n additions and deletions leading to an O(n log n) worst case running time. The heap sort is the slowest of the $O(n \log n)$ sorting algorithms, but unlike the merge and quick sorts it doesn't require massive recursion or multiple arrays to work. (Quick sort worst case time is $O(n^2)$ which is worse then O(n log n), but when optimized quick sort outperforms heap sort).

2.5 Quick Sort

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sublists.

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it. After this partitioning, the pivot is in its final position.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.



The base case of the recursion are lists of size one, which are always sorted. The algorithm always terminates because it puts at least one element in its final place on each iteration. The most crucial concern of a quicksort implementation is the choosing of a good pivot element. A naïve implementation of quicksort, like the ones below, will be terribly inefficient for certain inputs. For example, if the input is already sorted, a common practical case, this implementation of quicksort degenerates into a selection sort with $O(n^2)$ running time. A common choice is to randomly choose a pivot index, typically using a pseudorandom number generator. If the numbers are truly random, it can be proven that the resulting algorithm, called *randomized quicksort*, runs in an expected time of $O(n \log n)$.

2.6 Radix Sort

Radix sort is a fast stable sorting algorithm which can be used to sort items that are identified by unique keys. Every key is a string or number, and radix sort sorts these keys in some particular lexicographic-like order. The algorithm operates in $O(n \cdot k)$ time, where n is the number of items, and k is the average key length.

1. take the least significant digit (or group of bits) of each key.
2. sort the list of elements based on that digit, but keep the order of elements with the same digit (this is the definition of a stable sort).
3. repeat the sort with each more significant digit.

Example (170, 45, 75, 90, 2, 24, 802, 66) :
sorting by least significant digit (1s place) gives: 170, 90, 2, 802, 24, 45, 75, 66
sorting by next digit (10s place) gives: 2, 802, 24, 45, 66, 170, 75, 90
sorting by most significant digit (100s) gives: 2, 24, 45, 66, 75, 90, 170, 802

3. Sorting Networks: Parallel methods

Data independent methods do exhibit the discrepancies of time variation due to input like data dependent methods do. Data independent methods can be represented as a sorting network.

3.1 Bitonic sequence

A 0-1-sequence is called bitonic, if it contains at most two changes between 0 and 1, i.e. if there exist subsequence lengths $k, m \in \{1, ..., n\}$ such that

$a_0, ..., a_{k-1} = 0$, $a_k, ..., a_{m-1} = 1$, $a_m, ..., a_{n-1} = 0$ or

$a_0, ..., a_{k-1} = 1$, $a_k, ..., a_{m-1} = 0$, $a_m, ..., a_{n-1} = 1$

Examples: 00000 , 111111 , 0001110000, 111000111

More generally, a sequence of numbers is bitonic sequence if it has at most one local maximum or one local minimum.

Examples: 1,2,3,4,5 10,6,5,3,1 3,7,9,8,6,5,4,1 10,8,6,9,12,15,20

3.2 Bitonic sort

The bitonic sort takes a bitonic sequence as its input. Then it forms a binary split of its elements, compares the two partner elements, and then exchange the values as necessary. It continues this process recursively, and combines the elements at the end.

- BINARY-SPLIT: divide the list equally into two. Each item on the first half of the list has a "partner" which is the process in the same relative position from the second half of the list. Each pair of partners compare and exchange their values.

- Start with a bitonic sequence of length N (a power of 2 for simplicity)
and apply BINARY-SPLIT to it:

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

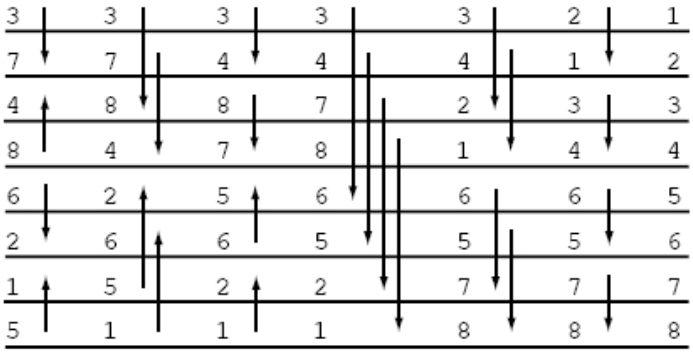
Result after Binary-split:
10 11 12 9 4 2 5 8 24 20 15 13 22 30 32 45

Notice that:
a) Each element in the first half is smaller than each element in the second half
b) Each half is a bitonic list of length n/2.

If you keep applying the BINARY-SPLIT to each half repeatedly:

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

Analysis:
In order to form a sorted sequence of length n from two sorted sequences of length $n/2$, there are $\log(n)$ comparator stages required (e.g. the $3 = \log(8)$ comparator stages to form sequence i from d and d'). The number of comparator stages $T(n)$ of the entire sorting network is given by: $T(n) = \log(n) + T(n/2)$. The solution of this recurrence equation is $T(n) = \log(n) + \log(n)\text{-}1 + \log(n)\text{-}2 + ... + 1 = \log(n) \cdot (\log(n)+1) / 2$.
Each stage of the sorting network consists of $n/2$ comparators. On the whole, these are $O(n \cdot \log(n)^2)$ comparators. So it would take n processors $O(\log(n)^2)$ to sort this.



Purcell Figure 2: Stages in a bitonic sort of eight elements. The unsorted input sequence is the left column.

3.1 Odd-even transition Sort

Odd-Even Transposition Sort is a parallel sorting algorithm. It is based on the Bubble Sort technique of comparing two numbers and switching them if the first is greater than the second, to achieve a left to right ascending ordering. Each number would look to its right neighbor and if it were greater and switch as necessary.

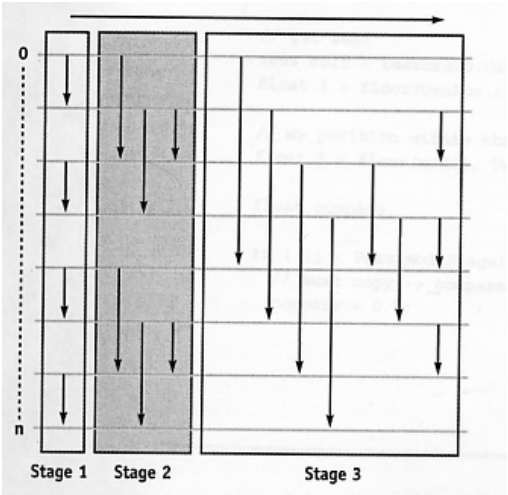


Figure 46-1 (GPU Gems) the odd even transition sorting network for n keys arrows indicate the compare operation 'less-than'

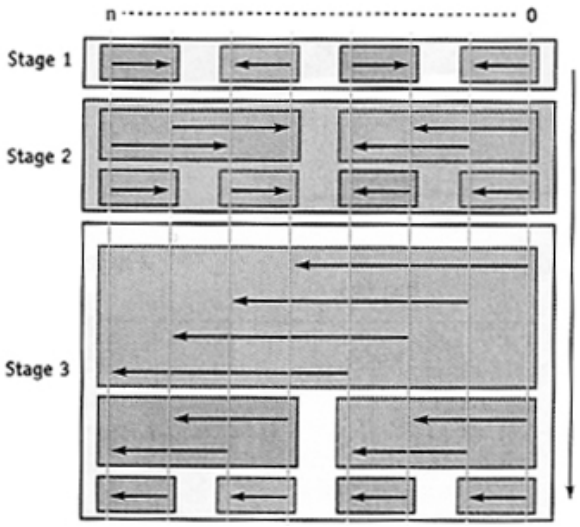
Example:

Iteration	Phase		PE1	PE2	PE3	PE4	PE5
1	1		4	5	2	1	3
1	2		4	5	1	2	3
2	1		4	1	5	2	3
2	2		1	4	2	5	3
3	1		1	2	4	3	5
3	2		**1**	**2**	**3**	**4**	**5**

Analysis: Odd-Even Transposition Sort only requires n/2 iterations of the dual phase sort, compared to n passes through the array with Bubble Sort. It has a running time of $O(\log^2 n)$ which asymptotically compares to Bitonic sort, but is slower by a constant factor.

4. Mapping Bitonic Sort to GPU

By rotating the normal bitonic sort 90 degrees to the right, we obtain the figure to the right for sorting a 1D array of n keys. Observe that in each pass, there are always groups of items that are treated alike. There is a strong relationship between neighboring groups: they sometimes have parameters with opposite values. We now draw several quads per sorting pass that exactly fit pairs of groups on the right figure and together cover the entire buffer. On the right side of each quad, you perform the operation opposite in the vertex program are linearly interpolated by the rasterizer over the fragments. If we want to sort many items, it is efficient to store them in a 2D texture. The remaining two actions that have to be computed are to decide which compare operation to use and to locate the partner item to compare.



GPU Gems Figure 46-4: Grouping keys for the Bitonic merge sort

The final optimization that is made is to generalize the sorter to work on key/index pairs. Because the GPU processes four-vectors, we can pack two key/index pairs into one fragment. This optimization cuts the row width in half and thus cuts the number of fragments in half as well.

`std::sort`: 16-Bit Data, Pentium 4 3.0 GHz

N	Full Sorts/Sec	Sorted Keys/Sec
256^2	82.5	5.4 M
512^2	20.6	5.4 M
1024^2	4.7	5.0 M

Odd-Even Merge Sort: 16-Bit Data, NVIDIA GeForce 6800 Ultra

N	Passes	Full Sorts/Sec	Sorted Keys/Sec
256^2	136	36.0	2.4 M
512^2	171	7.8	2.0 M
1024^2	210	1.25	1.3 M

Bitonic Merge Sort: 16-Bit Float Data, NVIDIA GeForce 6800 Ultra

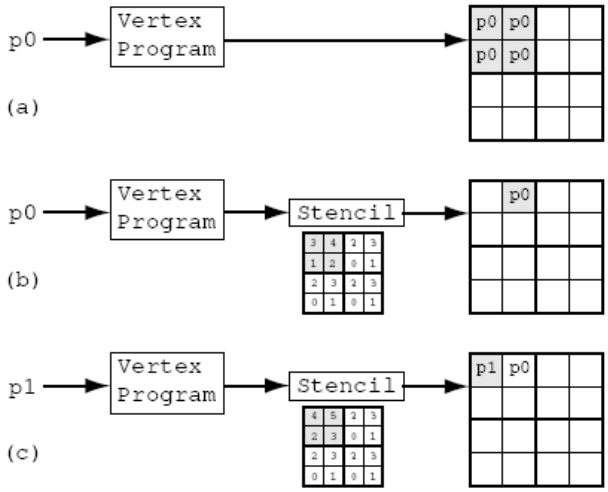
N	Passes	Full Sorts/Sec	Sorted Keys/Sec
256^2	120	90.07	6.1 M
512^2	153	18.3	4.8 M
1024^2	190	3.6	3.8 M

GPU Gems Table 46-1 : Performance of the CPU and GPU sorting Algorithms

5. Hashing on the GPU: Stencil Routing

In Photon Mapping on Programmable Graphics hardware, Purcell et al. devised a method of organizing photons into grid cells. Normally this can be perceived as a many-to-one routing problem, as there may be multiple photons to store in each cell. However, if there is a limit on the maximum number of photons that will be stored per cell, we can preallocate the storage for each cell. By knowing this .texture footprint of each cell in advance, we reduce the problem to a variant of one-to-one routing.

Purcell's main idea was to draw the photons as a large fat point using glPoint over the entire footprint of its destination cell. The the stencil buffer would route the appropriate photons to a unique destination within that footprint. Each set of mxm pixels contain at most mxm photons. They set the stencil buffer to control the location each photon renders to within each grid cell by allowing at most one fragment of the mxm fragments to pass for each drawn photon. The stencil buffer is initialized such that each grid cell region contains the increasing pattern. The stencil test is set to write on equal to m^2-1, and to always increment. Each time a photon is drawn, the stencil buffer allows only one fragment to pass through, the region of the stencil buffer covering the grid cell all increment, and the next photon will draw to a different region of the grid cell. This allows effcient routing of up to the first m^2 photons to each grid cell. This method can be completed in a single pass. [4]



Purcell Figure 5: Building the photon map with stencil routing. For this example, grid cells can hold up to four photons, and photons are rendered as 2 _ 2 points. Photons are transformed by a vertex program to the proper grid cell. In (a), a photon is rendered to a grid cell, but because there is no stencil masking the fragment write, it is stored in all entries in the grid cell. In (b) and (c), the stencil buffer controls the destination written to by each photon.

6. Resources

(note that a lot of the diagrams and specific algorithms above come from GPU Gems and CLRS)

1. Thomas Cormen, Charles Leiserson, Ron Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press, Sept. 2001

2. Wikipedia, Sorting Algorthim Overviews; http://en.wikipedia.org/wiki/Sorting_algorithm

3. Kipfer, Westermann, Improved GPU Sorting Chapter 46, GPU Gems 2, Addison-Wesley Professional, March 2005

4. Purcell, Donner, Cammarano, Jensen, and Hanrahan; *Photon Mapping on Programmable Graphics Hardware* Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware pp. 41-50, 2003.

5. UberFlow: A GPU-Based Particle Engine, by Westermann, Segal M and Kipfer (Graphics Hardware 2004)