Sorting on GPUs
Characteristics of GPU

• Manycore
  – Array of SM multiprocessors(1 ~ 30), each supporting 1024 threads

• Zero overhead SIMT(single instruction multiple thread) architecture

• Threads are executed in warps(32)

• Need a large number of live threads (5000-10000) to efficiently utilize the entire chip.
Radix Sort

• Easy to parallelize
  – Counting sort can be reduced to a parallel prefix sum or scan operation
GPU Radix sort

- Use radix sort to sort individual chunks of the input array. Chunks are sorted in parallel by multiple thread blocks. Chunks are as large as can fit into the shared memory of a single multiprocessor on the GPU.
- After sorting the chunks, we use a parallel bitonic merge to combine pairs of chunks into one. This merge is repeated until a single sorted array is produced.
Step 1: Radix sort chunks

- Radix sort is particularly well suited for small sort keys, such as small integers, that can be expressed with a small number of bits. At a high level, radix sort works as follows.

- We begin by considering one bit from each key, starting with the least-significant bit. Using this bit, we partition the keys so that all keys with a 0 in that bit are placed before all keys with a 1 in that bit, otherwise keeping the keys in their original order.

- We then move to the next least-significant bit and repeat the process.
Radix sort (1)
Radix sort (2)

Input

Split based on least significant bit b

\[ e = \text{Set a “1” in each “0” input} \]

\[ f = \text{Scan the 1s} \]

\[ \text{totalFalses} = e[n-1] + f[n-1] \]

\[ t = i - f + \text{totalFalses} \]

\[ d = b \oplus t : f \]

Scatter input using \( d \) as scatter address
Radix sort

- Thus for $k$-bit keys, radix sort requires $k$ steps. Our implementation requires one scan per step.
- The fundamental primitive we use to implement each step of radix sort is the split primitive. The input to split is a list of sort keys and their bit value $b$ of interest on this step, either a true or false. The output is a new list of sort keys, with all false sort keys packed before all true sort keys.
Radix sort

• The last element in the scan's output now contains the total number of false sort keys. We write this value to a shared variable, totalFalses.

• Now we compute the destination address for the true sort keys. For a sort key at index \(i\), this address is \(t = i - f + \text{totalFalses}\). We then select between \(t\) and \(f\) depending on the value of \(b\) to get the destination address \(d\) of each fragment.

• Finally, we scatter the original sort keys to destination address \(d\). The scatter pattern is a perfect permutation of the input, so we see no write conflicts with this scatter.
Radix sort

• With split, we can easily implement radix sort. We begin by loading a block-size chunk of input from global memory into shared memory. We then initialize our current bit to the least-significant bit of the key, split based on the key, check if the output is sorted, and if not shift the current bit left by one and iterate again. When we are done, we copy the sorted data back to global memory. With large inputs, each chunk is mapped to a thread block and runs in parallel with the other chunks.
Step 2: Merge Sorted Chunks

- After each block-size chunk is sorted, we use a recursive merge sort to combine two sorted chunks into one sorted chunk.
- If we have $b$ sorted chunks of size $n$, we require $\log_2 b$ steps of merge to get one final sorted output at the end.
- On the first step, we perform $b/2$ merges in parallel, each on two $n$-element sorted streams of input and producing $2n$ sorted elements of output.
- On the next step, we do $b/4$ merges in parallel, each on two $2n$-element sorted streams of input and producing $4n$ sorted elements of output, and so on.
Merge

- Sorted Input A
- Sorted Input B

Flip B, pairwise compare to A

Smallest element in each comparison yields smallest p elements overall in a bitonic sequence
Merge

• Parallel Bitonic Sort
• 2 inputs of arbitrary length located in GPU main memory.
• 2 buffers in shared memory, one for each input.
• Merge the smallest elements from each buffer.
• It then refills the buffers from main memory if necessary, and repeats until both inputs are exhausted.
• All reads from global memory into shared memory and all writes to global memory are coherent and blocked.
• Each input element is read only once from global memory and each output element is written only once.
Merge

• In our merge kernel, we run $p$ threads in parallel. The most interesting part of our implementation is the computation and sorting of the $p$ smallest elements from two sorted sequences in the input buffers.

• For $p$ elements, the output of the pairwise parallel comparison between the two sorted sequences is bitonic and can thus be efficiently sorted with $\log_2 p$ parallel operations.
__global__ static void bitonicSort(int * values) {
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;// Copy input to shared mem.
    shared[tid] = values[tid];
    __syncthreads();// Parallel bitonic sort.
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        // Bitonic merge:
        for (unsigned int j = k / 2; j>0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0) {
                    if (shared[tid] > shared[ixj])
                        swap(shared[tid], shared[ixj]);
                } else {
                    if (shared[tid] < shared[ixj])
                        swap(shared[tid], shared[ixj]);
                }
            }
            __syncthreads();
        }
    }
    // Write result.
    values[tid] = shared[tid];
}
New Sorting Algorithm (Sintorn and Assarsson)

- Hybrid of
  - Bucketsort:
  - Vector-Mergesort:

- Results:
  - 20% improvement over radix sort, best GPU algorithm
  - 6-14 times faster than quicksort on CPU
MergeSort

• $O(n \log n)$

• Can be easily parallelized on a GPU with scattered writing support
  – Highly bandwidth-limited!
    (bitonic sorting of 8-bit values are nearly four times faster than for 32-bit values) (same # of compare/swap)

• Very inefficient when $L<p$
Vector-MergeSort

Works on four 32-bit floats simultaneously, resulting in a nearly 4 times speed improvement.
New Sorting Algorithm (Sintorn and Assarsson)

• Three parts:
  – Histogramming: to split input list into L independent sublists for Pivot Points
  – Bucketsort: to split into lists than can be sorted using next step
  – Vector-Mergesort:
    • Elements are grouped into 4-float vectors and a kernel sorts each vector internally
    • Repeat until sublist is sorted

• Results:
  – 20% improvement over radix sort, best GPU algorithm
  – 6-14 times faster than quicksort on CPU
Terasort

- Definition: Sorting 1TB data
  - 100 bytes records with 10 bytes keys
  - 10,000,000,000 records

- Key metric: Minimize the time to sort

- Overall Execution Steps
  - Partitioning
  - Local sort
Terasort
- General Overview

Binary Record 100 bytes

| 10-byte Key | 90-byte Value |

Stage 1:
Hash based on the first 10 bits

Stage 2:
Sort each bucket on local node

Bucket-0
Bucket-1
Bucket-1023
Terasort
- *Terasort on MapReduce*

1. Building a partition table

2. Just reading input data

3. Partitioning based on the pre-built partition table

4. Sorting each partition

**Input**

**Map Phase**

**Reducer**

**Reduce Phase**

**Output 1**

**Output 2**

**Output 3**
Terasort
- *Partition Function (Sampling)*

This is not a MapReduce application.
Terasort
- Results

• Yahoo’s Hadoop
  – 209 seconds (1TB)
  – 910 nodes
  – 4 disks per node (total 3,640 disks)

• Google’s MapReduce
  – 68 seconds (1TB), 6h 2m (1PB)
  – 1000 nodes
  – 12 disks per node (total 12,000 disks)