Radix sort

1 Design

1.1 Decomposition of the sorting

Data decomposition The principle of radix sort is to consider, at each pass, the digit corresponding to the pass without looking at the others digits of the key.

If the set of data is composed of n keys, each key is formed by d digits, the total number of items to consider is $(d+1) \times (n \times d)$. $n \times d$ is the total number of digits and $(d+1)$ is the number of passes, the initial condition being counted as pass 0.

The granularity of this decomposition is 1 digit, the value of interest being the position of the digit in the new ordering.

This is a data decomposition approach since we consider the data independently of the transformation that are applied to it.

Functionnal decomposition In radix sort, the keys are binned based on the index of the digit considered at the pass. Each data is categorized. To go from ordering k to ordering $k+1$, there are n performed.

The total number of functions we considered is $n \times d$.

The granularity of this decomposition is 1 binning function.

> **Note 1**
> - The granularity for each decomposition is 1. This number is independent of the size of the data set.
> - The number of functions and the number of data that are considered in each decomposition are different. This difference is due to the fact that the binning of digits is done by set.

1.2 Communication

We look now at the dependency between the different element of the decomposition.

If we consider the functional decomposition, each binning of number is done independently of the others and can follow its own path. The property that is hidden in this algorithm, is the conservation of the ordering of the previous digits. This conservation is due to the sequential approach. Even though the binning of the number follow its own path, the position of the number at the end of each pass depend ont the ordering at the previous iteration.

Communication in data decomposition We will consider the data decomposition in what follows. First, digits are not all behaving independently. The digits are linked by the key they are forming. Each time a digit is binned, all other digits in the key is binned at the same time, in the same place.

At each pass, in the worst case, all digits have a new index. Hence, at each pass, there are n communications, 1 for each number. That is the cost of functional decomposition in the first part.

In the worst case, we will have a graph with $d \times n \times (d+1)$ nodes and $d \times n \times d$ edges.

1.3 Grouping

The first obvious grouping is the digits that formed the key. We have then $(d+1) \times n$ sets of data.

Radix sort operates on a set of data. The first part of a path is the histogram construction of the algorithm. Since each number is binned independently of the other, the second grouping can be of any size. The histogram construction allows to know at which position the number can be inserted.

To advanced in this grouping, we should now consider the implicit property of the radix sort : order preserving.

In parallel, that is the reason behind the grouping, this property cannot be guaranteed automatically, except if we force the processing units to behave sequentially. The position of the key in the dataset is based on the bin counter. This counters are common to all sets.

The sets are modified at each iteration.

The grouping will not depend on the number of data, only on the number of processors.

Note 2

- This approach is the first level of parallelism. It will allow for an easy implementation at the cost of synchronization and atomic operations for histogram construction.
- To improve on this approach, a hierarchial approach should be introduced.

2 GPU consideration

In the simplest case, we examine 1 bit of the keys in each pass. The computation of the rank of each element can then be performed with a single parallel prefix sum, or scan, operation.

Scans are a fundamental data-parallel primitive which can be implemented efficiently on manycore processors.

This approach to implementing radix sort is conceptually quite simple. Given scan and permute primitives, it is straightforward to implement. However, it is not particularly efficient when the arrays are in external DRAM. For 32-bit keys, it will perform 32 scatter operations that reorder the entire sequence being sorted. Transferring data to/from external memory is relatively expensive on modern processors, so we would prefer to avoid this level of data movement if possible.

```
__device__ void radix_sort(unsigned int *values)
{
    int  bit;
    for( bit = 0; bit < 32; ++bit )
    {
        partition_by_bit(values, bit);
        __syncthreads();
    }
}
template<class T>
__device__ T plus_scan(T *x)
{
    unsigned int i = threadIdx.x; // id of thread executing this instance
    unsigned int n = blockDim.x;  // total number of threads in this block
    unsigned int offset;          // distance between elements to be added

    for( offset = 1; offset < n; offset *= 2) {
        T t;

        if ( i >= offset )
            t = x[i-offset];

        __syncthreads();
```

```
24
25          if ( i >= offset )
26              x[i] = t + x[i];      // i.e., x[i] = x[i] + x[i-1]
27
28          __syncthreads();
29      }
30      return x[i];
31  }
32
33  __device__ void partition_by_bit(unsigned int *values, unsigned int bit)
34  {
35      unsigned int i = threadIdx.x;
36      unsigned int size = blockDim.x;
37      unsigned int x_i = values[i];          // value of integer at position i
38      unsigned int p_i = (x_i » bit) & 1;   // value of bit at position bit
39
40      values[i] = p_i;
41
42      __syncthreads();
43
44      unsigned int T_before = plus_scan(values);
45      unsigned int T_total  = values[size-1];
46
47      unsigned int F_total  = size - T_total;
48      __syncthreads();
49
50      if ( p_i )
51          values[T_before-1 + F_total] = x_i;
52      else
53          values[i - T_before] = x_i;
54  }
```

One natural way to reduce the number of scatter operations is to increase the radix r. Instead of considering 1 bit of the keys at a time, we can consider b bits at a time.

This requires a bucket sort with $2b$ buckets in each phase of the radix sort. To perform bucket sort in parallel, we may divide the input sequence into blocks and use separate bucket histograms for each block. Separate blocks are processed by different processors, and the per-processor histograms are stored in such a way that a single scan operation gives the offsets of each bucket in each block. This enables each block to read off its set of offsets for each bucket in the third step and compute global offsets in parallel.

While more efficient, we have found that this scheme also makes inefficient use of external memory bandwidth.

The higher radix requires fewer scatters to global memory. However, it still performs scatters where consecutive elements in the sequence may be written to very different locations in memory. This sacrifices the bandwidth improvement available due to coalesced writes, which in practice can be as high as a factor of 10.