PMPH group project: Sorting on GPUs

Mathias Heldbo (tzc182), Rasmus Gabelgaard Nielsen (ncg106)
and Hans Peter Lyngsøe (pvr448)

November 2024

1 The radix sort algorithm

In order to gain an understanding of radix sort, the sequential implementation will be discussed, and then a discussion of parallelization using basic blocks.

- Radix sort The basic idea of radix sort is to sort an integer into digits consisting of r bits. Then the list is sorted for each digit until all digits have been traversed and the list is fully sorted. The main assumption of radix sort is that all integers in the input list can be split into digits
- Radix sort sequential: The sequential implementation iterates through the digits, r bits at a time, a standard implementation is 1 bit at a time, but more is more efficient. Each digit m, has r bits, such that $m = 2^r$. Radix sort works by performing a counting sort on each set of digits, until all digits have been traversed. Counting sort follows the following syntax:

```
COUNTING-SORT
  HISTOGRAM-KEYS
    do i ← 0 to 2^r − 1
      Bucket[i] ← 0
    do j ← 0 to N − 1
      Bucket[D[j]] ← Bucket[D[j]] + 1
  SCAN-BUCKETS
    Sum ← 0
    do i ← 0 to 2^r − 1
      Val ← Bucket[i]
      Bucket[i] ← Sum
      Sum ← Sum + Val
  RANK-AND-PERMUTE
    do j ← 0 to N − 1
      A ← Bucket[D[j]]
      R[A] ← K[j]
      Bucket[D[j]] ← A + 1
```

Figure 1: Counting Sort [Zagha and Blelloch, 1991]

.

Where Histogram-Keys counts the number of elements in a bucket for each unique digit. Scan-Buckets, scan the buckets and returns the position at which the first element of a subset having a certain digit is to be placed, that is, the offset. Finally, Rank-and-permute matches the digit value with the bucket offset and iterates through each "sub bucket" to order the elements by their digit. After this, counting sort is repeated on each subset of digits until the whole list is sorted.

- For parallelization of radix sort, similar ideas are used, however, there a few key differences. One such method is discussed in [Zagha and Blelloch, 1991]. With p processors, the input list is split into p sub-lists for each processor to work on. Each processor performs Histogram-keys locally to create local histograms, then scan buckets is run globally to calculate the global offset, Rank and Permute can then be performed locally and written to global memory as Rank and Permute takes in the offset, and therefore doesn't overwrite the work done by other processors [Zagha and Blelloch, 1991].

  Let us now consider a very simple parallel radix sort implementation going through 1 bit at a time, for 32 bit integers using basic blocks, and let us consider the example of

$$[1, 4, 2, 3]$$

  with a bit representation of:

$$[001, 100, 010, 011]$$

  Please bear in mind that this implementation is a very simplified parallel radix sort implementation in Futhark using basic blocks such as map, scan, reduce and scatter. Due to its syntax, Futhark automatically distributes the processes depending on available processors, however, better approaches are discussed in part 2.

```
-- xs = [1,4,2,3] in integer format
-- xs = [001, 100, 010, 011] in bit format
def radix_bit [n] 't (f: t -> u32) (xs: [n]t) (b: i32): [n]t =
(1)    let bits = map(\x -> (i32.u32 (f x >> u32.i32 b )) & 1) xs -- [1, 0,
    0, 1], D = O(1), W = O(n)
(2)    let bits_neg = map (1-) bits                               -- [0, 1,
    1, 0], D = O(1), W = O(n)
(3)    let offs = reduce (+) 0 bits_neg                           -- 2,
               D = O(log n), W = O(n)
(4)    let idxs0 = map2 (*) bits_neg (scan (+) 0 bits_neg)        -- [0, 1,
    2, 0], D = O(log n), W = O(n)
(5)    let idxs1 = map2 (*) bits (map (+offs) (scan (+) 0 bits))  -- [3, 0,
    0, 4], D = O(log n), W = O(n)
(6)    let idxs2 = map2 (+) idxs0 idxs1                           -- [3, 1,
    2, 4], D = O(1), W = O(n)
(7)    let idxs = map (\x -> x-1) idxs2                           -- [2, 0,
    1, 3], D = O(1), W = O(n)
```

```
(8)     let xs' = scatter (copy xs) idxs xs                          -- [4, 2,
    1, 3], D = O(1), W = O(n)
        -- in terms of the bit in question, it becomes        [100, 010, 001,
            011]
(9)     in xs'
                                                                        -- D =
    O(log n), W = O(n)

def radix [n] 't (f: t -> u32) (xs: [n]t): [n]t =
(10)     loop xs for i < 32 do radix_bit f xs i                       -- D = O(
    log n), W = O(n)
```

This implementation is from radix-sort-key (futhark-lang.org), and applied to each bit of 32 bit unsigned integers, as radix sort requires each digit to be in the range of 0 to m-1. The radix_bit part of the algorithm takes in a bit. The bits are stored in bits (1), bits_neg (2) contains the bits converted to their compliment, offs (3), get the offsets of the bits by calculating how many 0 bits there are, which is what bits_neg is used for. Idxs0 (4) finds the indices of elements with a bit of 0, while idxs1 (5) finds the indices of elements with a bit of 1, and idxs2 (6) concatenate those two lists. Indexes are then stored in idxs (7) assuming zero indexing, by subtracting 1 from every index. Then scatter (8) is used to distribute the elements to their corresponding location.
Let us now consider the work depth complexity, for a list of length n, the work and depth complexity is written next to the code, where scan and reduce both have a depth complexity of O(log n), and all lines have O(n) work complexity, as they traverse n elements. While this work depth complexity might seem good on paper, this is a best case scenario, in which we always have n processors. More advanced methods deal with the distribution of the workload in a fashion more suited to the GPU, and this improving the overall speed and efficiency of the algorithm.

2 Fast parallel radix sort

Although [Zagha and Blelloch, 1991] was great at the time, our literature search provided several higher performing parallel radix sorts. The first implementation by Satish et. al. provides a significant speed up from previous methods using the following ground principles and distributing 4 bit digits per thread,

1) Each block loads and sorts its tile in on-chip memory using b iterations of 1-bit split.
2) Each block writes its 2^b-entry digit histogram and the sorted data tile to global memory.
3) Perform a prefix sum over the $p \times 2^b$ histogram table, stored in column-major order, to compute global digit offsets [19], [21] (see Figure 2).
4) Using prefix sum results, each block copies its elements to their correct output position.

Figure 2: Satish steps [Satish et al., 2009]

. They also proposed parallel local radix counting and pre-sorting as well as global radix ranking and coalesced global shuffling[Satish et al., 2009]Building off these concepts, Ha. et. al proposed two improvements, implicit counting and a mixed-data structure with 4 elements, that is 2-bit digits.

- Implicit counting
  Uses bit shifting to create a 32-bit register for radix value 0, 1 and 2. With the count of 0 assigned to bits from 0 to 9, the count of 1 assigned to bit values from 10-19, and the count of 2 assigned to bits from 20-29. The implicit count is:

  $$impl_{cnt} = cnt_0 + (cnt_1 \ll 10) + (cnt_2 \ll 20)$$

  The implicit value is:

  $$impl_{val} = (val3) \ll (10 \cdot val)$$

  $impl_{cnt}$ is calculated by incrementing by the $impl_{val}$:

  $$impl_{cnt} = impl_{cnt} + impl_{val}$$

  The count is retrieved by:

  $$cnt\left[val\right] = impl_{cnt} \gg (10 \cdot val)$$

  and the fourth counting value is retrieved by:

  $$cnt\left[3\right] = idx - cnt\left[0\right] - cnt\left[1\right] - cnt\left[2\right]$$

  Thus implicit counting reduces the count operations from 4 in Satish. et. al to 2.

- Mixed-data structure

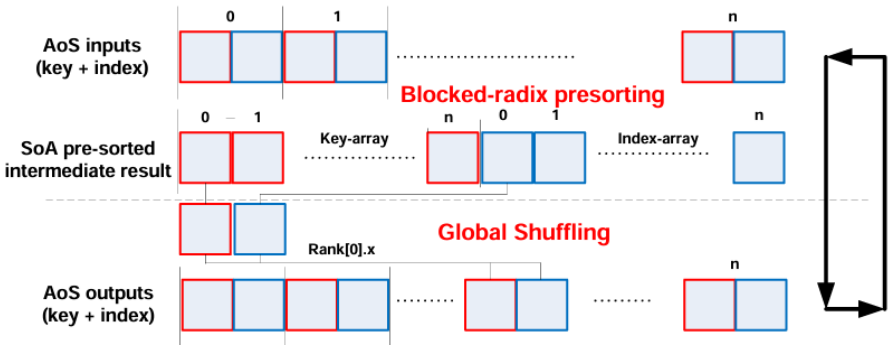

Figure 3: Mixed data method [Ha et al., 2010]

. Ha. et. al's hybrid data structure uses AoS and SoA, which reduces suboptimal coalesced scattering effects, and requires only one shuffle pass on the pre sorting data due to the structure [Ha et al., 2010]
We have decided to attempt to implement the Ha. et. al algorithm

- Pseudocode

```
1. Initialize data structures:
   - Load input keys and values.
     * Work: O(n), Depth: O(1)
   - Assuming the input is integers we can map from 32 to 24 bit by:
   - Determine [a,b] using reduce
     * Work: O(n), Depth: O(log n)
   - Map input values from a, b to [0, b-a]
     * Work: O(n), Depth: O(1)

2. Iterate through bit shifts:
   For each bit shift (e.g., 4 bits per pass):
     - Map the current bit shift to all elements.
       * Work: O(n), Depth: O(1)
     - Create an array of structures ‘AoS‘ by associating each element’s
         index and key.
       * Work: O(n), Depth: O(1)

3. Local counting and presorting (within each block):
   - For each block of block size B * 4 elements:
     - Compute implicit counts:
       * Work: O(B*4)  per block, total O(n), Depth: O(log B*4)

     - Calculate the local rank for each element in the block using **
         exclusive scan** on the histogram:
       * Work: O(B*4) per block, total O(n), Depth: O(log B*4)
```

```
     - Scatter locally (pre sort):
       * Work: O(B) per block, total O(n), Depth: O(1)

4. Global ranking and offset calculation:
   - Compute global offsets using **prefix sum** (scan) across all local
     histograms:
     * Work: O(P) = O(n/(B))
     * Depth: O(log P) = O(log n/(B))

   - Adjust local ranks by adding block offsets:
     * Work: O(n), Depth: O(1)

5. Global shuffling and final scatter:
   - Scatter elements globally using computed global offsets:
     * Work: O(n), Depth: O(log B)

6. Repeat until all bits are processed.
   - Number of passes depends on bit width (e.g., for 24-bit keys (mapping
     from 32 to 24) with 2 bits (4 elements) per pass, we need 16 passes).
   - Total:
     * Work: O(n * #passes), Depth: O(#passes * (log B + log P) = O(#passes *
         (log B + log (n/B)))
```

Additionally, the algorithm has bank conflict-free access, storing in different arrays to avoid concurrent access, Mapping of integers from [a,b to 0, b-a] for integers, and floats by mapping from [a,b] to [0.5, 1] to change from 32 bit to 24 bit. 30% performance increase. It is an approximation but it is precise enough.

3 In depth cuda implementation

4 Performance evaluation

References

Linh Ha, Jens Krüger, and Cláudio T Silva. Implicit radix sorting on gpus. *arXiv preprint arXiv:1010.2016*, 2010.

Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. *Proc. IEEE International Symposium on Parallel Distributed Processing*, 2009.

Marco Zagha and Guy. E Blelloch. Radix sort for vector multiprocessors. *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 712 – 721, 1991.