INFO 6205 Fall 2021
Team Project-Final
Paper Description

Member: Fengyi Zhang, Peng Chen, Shibo Lu

Paper 1: A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs [1]

According to the paper, state-of-the-art approaches are heavily memory band-width-bound. Compared with their CPU-based counterparts, they require substantially more memory transfers in data process. In this paper, they propose a novel approach that almost 50% the amount of memory transfers which considerably lifts the memory bandwidth limitation and increase the highest performance of these heavily memory band-width-bound algorithm.

To solve the input that does not reside on the GPU or exceeds the available device memory, they build a pipelined heterogeneous sorting algorithm based on the efficient GPU sorting method, which reduces the overhead related to PCIe data transmission. Previous work has addressed this issue by using a least-significant-digit-first radix sort (LSD radix sort) that iterates over the keys' bits from the least-significant to the most-significant digit, considering an implementation specific number of consecutive bits at a time. With each sorting pass, a stable counting sort is used to partition the keys into buckets according to the bits being considered with the current pass [2, 3].

Compared with LSD radix sorting used by most advanced implementations (such as cub), the method of this paper does not rely on a stable sorting process [4]. Therefore, for keys falling into the same bucket, it is not limited to following the order of the previous sorting process. Lifting this restriction enables this method to use native shared memory atomic operations, which are available in the recent GPU microarchitecture to alleviate the disadvantage of considering more bits per sort [5]. their hybrid method starts from the most significant bit and advances to the least significant bit, dividing the key into smaller and smaller buckets. It avoids the situation that dividing the input into too many buckets will have a negative impact on performance. If the bucket is small enough to fit into the on-chip memory, it completes the local sorting. Because local sorting performs sorting in on-chip memory, it only needs to access the device memory twice, one is the read key and the other is the write key, no matter how many times it needs to be sorted. This further saves the necessary memory bandwidth and improves the performance of favorable distribution.

Table 1: Hybrid radix sorting example: sorting 16 keys of k=4 bits with d=2 bits and a radix of r=4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys (radix 4)	31	12	01	23	12	22	12	00	11	10	10	31	03	13	12	03
histogram	4	8	2	2												
prefix-sum	0	4	12	14												
sort (radix 4)	bucket 0				bucket 1								bucket 2		bucket 3	
	01	00	03	03	12	12	12	11	10	10	13	12	23	22	31	31
histogram	1	1	0	2	2	1	4	1					local		local	
prefix-sum	0	1	2	2	0	2	3	7					local		local	
sort (radix 4)	b_0	b_1	b_3		b_0		b_1	b_2				b_3	local		local	
	00	01	03	03	10	10	11	12	12	12	12	13	22	23	31	31

Since local sorting is to sort the keys of the storage bucket in the on-chip shared memory, it can only sort D keys at most, which is implied by the key size and available hardware resources. Buckets that exceed the threshold D of local sort are sorted into sub buckets using count sorting. Count sorting starts from the offset of the bucket in the input memory, reads the keys, divides them into sub buckets according to the specified number, and writes the sequence of sub buckets coherently into the output memory, so that the starting position of the sub bucket saving key with the minimum number value is the same as the offset of the input bucket. The implementation of

counting sorting of a single bucket follows the following steps:

(1) Calculate the histogram of the digital values of all keys in the bucket to determine the size of each sub bucket.

(2) The exclusive prefix sum on the histogram is calculated to obtain the offset of each R sub bucket.

(3) Disperse the key into the sub bucket according to the bit value of the key.

The proposed method is illustrated in Table 1, which shows the algorithm for 16 keys with a length of 4 bits. Radix sorting is performed using two digits with radix r = 4. It takes exactly two iterations to completely sort the keys. Keys are represented in Quaternary notation. In the example, they set the threshold of local sorting to blind = 3, and change to local sorting for buckets with less than three keys

This approach is successfully addressing the challenges arising from implementing an MSD radix sort on GPUs, such as load balancing issues for skewed distributions and performance degradation due to bucket handling, while still being able to max out the high memory bandwidth of GPUs. Moreover, sorting small buckets in on-chip memory rather than running them through subsequent partitioning passes enables additional performance improvements, culminating in highly speed-up over the state-of-the-art approach.

Paper 2: Engineering a Multi-core Radix Sort [6]

The naive virtual memory count sorting first generates a histogram of N keys. After calculating the prefix and to produce the starting output position of each key, each value is written to its key output position and then incremented.

Their first optimization goal is to avoid initial counting. Compared with inserting each value into the container of each key, this incurs some overhead of checking whether the current bucket is full. It is more efficient to pre-allocate space for M arrays of size n. Because items can be simply written to the next free location [7]. The algorithm only writes and reads each item once, which is a feat at the cost of $n \cdot m$ space.

To reduce this overhead and parallel communication, they use "reverse sorting" [8], in which one or more MSDS pass the data into buckets, and then sort locally through LSD. It turns out that this is more advantageous for non-uniform memory access (NUMA) systems because each processor is responsible for writing a continuous range of output, ensuring that the operating system allocates these pages from the NUMA node [9] of the processor.

Algorithm 1: Parallel Radix Sort

```
parallel foreach item do
    d := Digit(item, 3);
    buckets3[d] := buckets3[d] ∪ {item};
Barrier;
foreach i ∈ [0, 2^D) do
    bucketSizes[i] := Σ_PE |buckets3[i]|;
outputIndices := PrefixSum(bucketSizes);
parallel foreach bucket3 ∈ buckets3 do
    foreach item ∈ bucket3 ∀ PE do
        d := Digit(item, 0);
        buckets0[d] := buckets0[d] ∪ {item};
    foreach bucket0 ∈ buckets0 do
        foreach item ∈ bucket0 do
            d := Digit(item, 1);
            buckets1[d] := buckets1[d] ∪ {item};
            d := Digit(item, 2);
            histogram2[d] := histogram2[d] + 1;
    foreach bucket1 ∈ buckets1 do
        foreach item ∈ bucket1 do
            d := Digit(item, 2);
            i := outputIndices[d] + histogram2[d];
            histogram2[d] := histogram2[d] + 1;
            output[i] := item;
```

The motivation of the Algorithm 1 is easy access to each number, but it is also limited by cache and TLB size. Since many TLB entries are required, they map buckets with small pages. To increase TLB coverage, they use large pages as input. The working set consists of 2D buffer, buffer pointer, output position and 32-bit histogram counter. To avoid associativity and alias conflicts, these arrays are contiguous in memory.

In this paper, they introduced the improvement of count sorting and the new variant of radix sorting of integer key / value pairs. In bandwidth measurement, the throughput of their algorithm is within 11% of the theoretical optimal value of a given hardware.

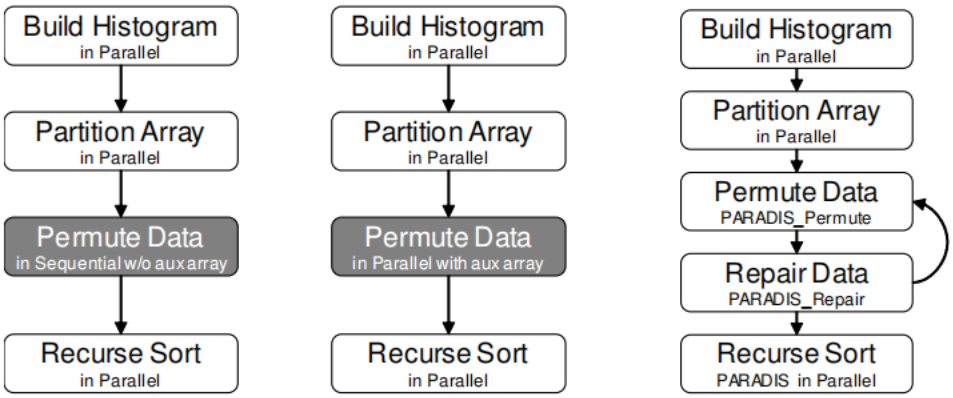
Paper 3: PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort [10]

In-place radix sorting is a popular sorting algorithm based on distribution. However, efficient parallelization of in-place radix sorting is very challenging for two reasons. First, the initial stage of arranging elements into a bucket suffers from the read-write dependency inherent in its in-place nature. Secondly, when the size of the bucket is very different, the load balancing of the recursive application of the algorithm to the result bucket is difficult, which occurs in the skewed distribution of the input data. In this paper, they propose a novel parallel in-place radix sorting algorithm PARADIS, which solves two problems:

a) "speculative permutation" solves the first problem by allocating multiple discontinuous array strips to each processor. The resulting shared-nothing scheme realizes complete parallelization. Since the speculative permutation is incomplete, the next step is the "repair" phase, which can be completed in parallel again without any data sharing between processors.

b) Distributed adaptive load balancing solves the second problem. They dynamically allocate processors in the context of radix sorting to minimize overall completion time.

Figure 1: a novel parallel in-place radix sorting algorithm PARADIS



This a fully parallelized in-place radix sort engine with two novel ideas: speculative permutation and distribution-adaptive load balancing, in details:

- A speculative permutation followed by repair which are both efficiently parallelized. By iterating these two steps, PARADIS permutes all array elements into their buckets, fully in parallel and in-place.
- A distribution-adaptive load balancing technique for recursive invocations of the algorithm on the resulting buckets. For a skewed distribution, PARADIS minimizes the elapsed run-time by adaptively allocating more processors to larger buckets.

These two novel ideas enable PARADIS to sort very efficiently large variety of benchmarks. With architectural trends towards increasing number of cores and larger memory systems, PARADIS is well suited for in-memory sorting kernels for many data management applications.

Paper 4: Theoretically-Efficient and Practical Parallel In-Place Radix Sorting [11]

This paper presents Regions Sort, a new parallel in-place radix sorting algorithm that uses a graph structure to model dependencies across elements that need to be swapped and generates independent tasks from this graph that can be executed in parallel. For sorting n integers from a range r, and a parameter K, Regions Sort requires only $O(Klogrlogn)$ auxiliary memory. This algorithm re- quires $O(nlogr)$ work and $O(\left(\frac{n}{K}+logK\right)logr)$span, making it work-efficient and highly parallel.

Algorithm 2: Pseudocode for Regions Sort

```
1: n = initial input size
2: K = initial number of blocks
3: procedure RadixSort(A, N, level)
4:     K' = ⌈K · N/n⌉
                                        ▷ Local Sorting Phase
5:     parfor k = 0 to K' − 1 do
6:         Do serial in-place distribution on A[kN/K', ..., ((k+1)N/K' − 1)]
                                        ▷ Graph Construction Phase
7:     Compute global counts array C from local counts, and per-
   bucket start and end indices
8:     Generate regions graph G
                                        ▷ Global Sorting Phase
9:     while G is not empty do
10:        Process a subset of non-conflicting edges in G in parallel
11:        Remove processed edges and add new edges to G if
   needed
                                        ▷ Recursion
12:    if level < maxLevel then
13:        parfor i = 0 to B − 1 do
14:            if C[i] > 0 then
15:                RadixSort(A_i, C[i], level + 1)
```

They implemented the 2-path variant of Regions Sort using Cilk Plus [12], which provides a provably efficient work- stealing scheduler [13]. The 2-path variant to always be faster due to avoiding the overhead of cycle finding and being able to effectively use the early recursion optimization. When processing 2-paths that are not cycles, after performing a swap, only one of the keys will be in the correct country. However, if only process a 2-cycle, then both keys will be in the correct country after swapping. Furthermore, no new edge must be created in the regions graph. Therefore, processing as many 2-cycles as possible will reduce the overall work. The algorithm first finds and processes all 2-cycles on a vertex before processing 2- paths that are not cycles. Although there is additional work needed to determine 2-cycles, the overall work decreases since a swap places two keys in the correct country instead of one, and no new edges must be formed in the regions graph.

Parallelization Across Cycles and 2-paths. In their cycle processing algorithm, they process all cycles for a vertex v before moving to the next vertex. First, finding all the cycles for vertex v serially, but allow edges to have their weight split across multiple cycles, so that all of v's outgoing edges will be contained in one or more cycles. Then execute the swaps associated with all of the cycles in parallel. Finally, remove v from the graph. While this optimization does not improve the worst-case span (cycle-finding is still done serially), it provides more parallelism during global sorting.

Regions Sort is a novel parallel in-place radix sorting algorithm that is work-efficient and highly parallel. Due to the highly parallelized code design, it achieves good scalability.

[1] Stehle, E. , & Jacobsen, H. A. . (2017). A memory bandwidth-efficient hybrid radix sort on gpus. the 2017 ACM International Conference.

[2] Ha, L., Krüger, J., & Silva, C. T. (2009, December). Fast Four-Way Parallel Radix Sorting on GPUs. In Computer Graphics Forum (Vol. 28, No. 8, pp. 2368-2378). Oxford, UK: Blackwell Publishing Ltd.

[3] Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D., & Dubey, P. (2010, June). Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (pp. 351-362).

[4] D. Merrill and NVIDIA Corporation. CUB. https://github.com/NVlabs/cub, 2016.

[5] NVIDIA GeForce GTX 980. Whitepaper. Technical report, NVIDIA, 2014.

[6] Wassenberg, J. , & Sanders, P. . (2011). Engineering a multi-core radix sort. DBLP.

[7] Wassenberg, J., Middelmann, W., & Sanders, P. (2009, September). An efficient parallel algorithm for graph-based image segmentation. In International Conference on Computer Analysis of Images and Patterns (pp. 1003-1010). Springer, Berlin, Heidelberg.

[8] Jiménez-González, D., Navarro, J. J., & Larrba-Pey, J. L. (2001, June). Fast parallel in-memory 64-bit sorting. In Proceedings of the 15th international conference on Supercomputing (pp. 114-122).

[9] an Mey, D., & Terboven, C. (2008). Affinity matters! OpenMP on multicore and ccNUMA architectures. Parallel Computing: Architectures, Algorithms and Applications, 15.

[10] Cho, M. , Brand, D. , Bordawekar, R. , Finkler, U. , & Puri, R. . (2015). Paradis: an efficient parallel algorithm for in-place radix sort. Proceedings of the VLDB Endowment, 8(12), 1518-1529.

[11] Obeya, O., Kahssay, E., Fan, E., & Shun, J. (2019, June). Theoretically-efficient and practical parallel in-place radix sorting. In The 31st ACM Symposium on Parallelism in Algorithms and Architectures (pp. 213-224).

[12] Leiserson, C. E. (2010). The Cilk++ concurrency platform. The Journal of Supercomputing, 51(3), 244-257.

[13] Blumofe, R. D., & Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), 46(5), 720-748.