Fast Segmented Sort on GPUs

Kaixi Hou†, Weifeng Liu‡§, Hao Wang†, Wu-chun Feng†
†Department of Computer Science, Virginia Tech, Blacksburg, VA, USA, {kaixihou, hwang121, wfeng}@vt.edu
‡ Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark, weifeng.liu@nbi.ku.dk
§ Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway.

ABSTRACT

Segmented sort, as a generalization of classical sort, orders a batch of independent segments in a whole array. Along with the wider adoption of manycore processors for HPC and big data applications, segmented sort plays an increasingly important role than sort. In this paper, we present an adaptive segmented sort mechanism on GPUs. Our mechanisms include two core techniques: (1) a differentiated method for different segment lengths to eliminate the irregularity caused by various workloads and thread divergence; and (2) a register-based sort method to support N-to-M data-thread binding and in-register data communication. We also implement a shared memory-based merge method to support non-uniform length chunk merge via multiple warps. Our segmented sort mechanism shows great improvements over the methods from CUB, CUSP and ModernGPU on NVIDIA K80-Kepler and TitanX-Pascal GPUs. Furthermore, we apply our mechanism on two applications, i.e., suffix array construction and sparse matrix-matrix multiplication, and obtain obvious gains over state-of-the-art implementations.

CCS CONCEPTS

•Theory of computation → Parallel algorithms; •Computing methodologies → Massively parallel algorithms;

KEYWORDS

segmented sort, SIMD, vector, registers, shuffle, NVIDIA, GPU, Pascal, skewed data

ACM Reference format:
Kaixi Hou, Weifeng Liu, Hao Wang, Wu-chun Feng. 2017. Fast Segmented Sort on GPUs. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079105

1 INTRODUCTION

Sort is one of the most fundamental operations in computer science. A sorting algorithm orders entries of an array by their ranks. Even though sorting algorithms have been extensively studied on various parallel platforms [26, 33, 38, 40], two recent trends necessitate revisiting them on throughput-oriented processors. The first trend is that manycore processors such as GPUs are more and more used both for traditional HPC applications and for big data processing. In these cases, a large amount of independent arrays often need to be sorted as a whole, either because of algorithm characteristics (e.g., suffix array construction in prefix doubling algorithms from bioinformatics [15, 44]), or dataset properties (e.g., sparse matrices in linear algebra [4, 28–31, 42]), or real-time requests from web users (e.g., queries in data warehouse [45, 49, 51]). The second trend is that with the rapidly increased computational power of new processors, sorting a single array at a time usually cannot fully utilize the devices, thus grouping multiple independent arrays and sorting them simultaneously are crucial for high utilization.

As a result, the *segmented sort* that involves sorting a batch of segments of non-uniform length concatenated in a single array becomes an important computational kernel. Although directly sorting each segment in parallel could work well on multicore CPUs with dynamic scheduling [39], applying similar methods such as "dynamic parallelism" on manycore GPUs may cause degraded performance due to high overhead for context switch [14, 43, 46, 48]. On the other hand, the distribution of segment lengths often exhibits the skewed characteristics, where a dominant number of segments are relatively short but the rest of them can be much longer. In this context, the existing approaches, such as the "one-size-fits-all" philosophy [37] (i.e., treating different segments equally) and some variants of global sort [3, 10] (i.e., traditional sort methods plus segment boundary check at runtime), may not give best performance due to load imbalance and low on-chip resource utilization.

We in this work propose a fast segmented sort mechanism on GPUs. To improve load balance and increase resource utilization, our method first constructs basic work units composed of adaptively defined elements from multiple short segments of various sizes or part of long segments, and then uses appropriate parallel strategies for different work units. We further propose a register-based sort method to support N-to-M data-thread binding and in-register data communication. We also design a shared memory-based merge method to support variable-length chunks merge via multiple warps. For the grouped short and medium work units, our mechanism does the segmented sort in the registers and shared memory; and for those long segments, our mechanism can also exploit on-chip memories as much as possible.

Using segments of uniform and synthetic power-law length on NVIDIA K80-Kepler and TitanX-Pascal GPUs, our segmented sort can exceed state-of-the-art methods in three vendor libraries CUB [37], CUSP [10] and ModernGPU [3] by up to 86.1x, 16.5x, and 3.8x, respectively. Furthermore, we integrate our mechanism with two real-world applications to confirm their efficiency. For the suffix array construction (SAC) in bioinformatics, our mechanism results in a factor of 2.3–2.6 speedup over the latest skew-SA

ICS '17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3079079.3079105

method [44] with CUDPP [19]. For the sparse matrix-matrix multiplication (SpGEMM) in linear algebra, our method delivers a factor of 1.4–86.5, 1.5–2.3, and 1.4–2.3 speedups over approaches from cuSPARSE [35], CUSP [10], and bhSPARSE [30], respectively. The contributions of this paper are listed as follows:

- We identify the importance of segmented sort on various applications by exploring segment length distribution in real-world datasets and uncovering performance issues of existing tools.
- We propose an adaptive segmented sort mechanism for GPUs, whose key techniques contain: (1) a differentiated method for different segment lengths to eliminate load imbalance, thread divergence, and irregular memory access; and (2) an algorithm that extends sorting networks to support N-to-M data-thread binding and thread communication at GPU register level.
- We carry out a comprehensive evaluation on both kernel level and application level to demonstrate the efficacy and generality of our mechanism on two NVIDIA GPU platforms.

2 BACKGROUND AND MOTIVATION

2.1 Segmented Sort

Segmented sort (SegSort) performs a segment-by-segment sort on a given array composed of multiple segments. If there is only one segment, the operation converts into the classical sort problem that gains much attention in the past decades. Thus sort can be seen as a special case of segmented sort. The complexity of segmented sort can be $\sum_{i=1}^{p} n_i \log n_i$[1], where p is the number of segments in the problem and n_i is length of each segment. Fig. 1 shows an example of segmented sort, where an array stores a list of keys (integer in this case) plus an additional array *seg_ptr* used for storing head pointers of each segment.

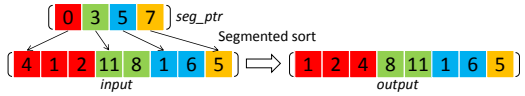


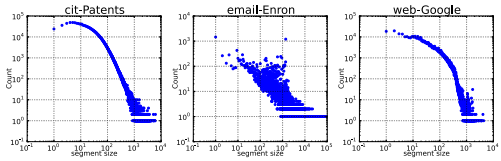
Figure 1: An example of segmented sort. It sorts four integer segments of various lengths pointed by *seg_ptr*.

2.2 Skewed Segment Length Distribution

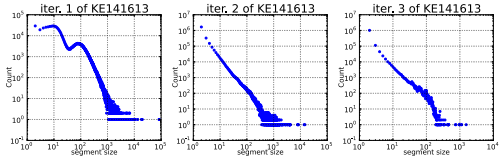
We use real-world datasets from two applications to analyze the characteristics of data distribution in segmented sort. The first application is the suffix array construction (SAC) from Bioinformatics, where the prefix doubling algorithm [15, 44] is used. This algorithm calls SegSort to sort each segment in one iteration step, and the duplicated elements will form another sets of segments for the next iteration. This procedure continues until no duplicated element exists. The second one is the sparse matrix-matrix multiplication (SpGEMM). In this algorithm, SegSort is used for reordering entries in each row by their column indices.

As shown in Fig. 2, the statistics of segments derived from these two algorithms shares one feature that the small/medium segments dominate the distribution, where around 96% segments in SpGEMM and 99% segments in SAC have less than 2000 elements, and the rest can be much longer but contributes less to the number of entries in the whole problem. Such highly skewed data stems from either

[1]For generality, we only focus on comparison-based sort in this work.



(a) Segments from squaring three different matrices in SpGEMM



(b) Segments from the first three iterations in SAC

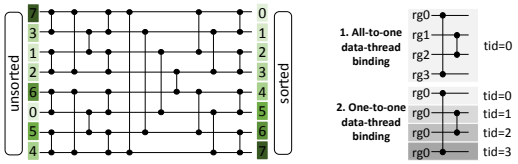
Figure 2: Histogram of segment length changes in SpGEMM and SAC

the input data or the intermediate data generated by the algorithm at runtime. As a result, the segments of various lengths require differentiate processing methods for high efficiency. Later on, we will present an adaptive mechanism that constructs basic work units composed of multiple short segments of various sizes or part of long segments, and processes them in a very fine grained way for achieving load balance on manycore GPUs.

2.3 Sorting Networks and Their Limitations

A sorting network consisting of a sequence of independent comparisons is usually used as a basic primitive to build the corresponding sorting algorithm. Fig. 3a provides an example of bitonic sorting network that accepts 8 random integers as input. Each vertical line in the sorting network represents a comparison of input elements. Through theses comparisons, the 8 integers are sorted.

Although a sorting network provides a view of how to compare input data, it does not give a parallel solution of how the data are distributed into multiple threads. A straightforward solution is directly mapping one element of input data to one GPU thread. However, this method will waste the computational power of GPU, because every comparison represented by a vertical line is conducted by two threads. This method also leads to poor instruction-level parallelism (ILP), since the insufficient operations per thread cannot fully take advantage of instruction pipelining. Therefore, it is important to investigate how many elements in a sorting network processed by a GPU thread can lead to best performance on GPU memory hierarchy. In this paper, we call it the *data-thread binding* on GPUs.



(a) A 8-way bitonic sorting network **(b) Data-thread bindings**

Figure 3: One sorting network and existing strategies using registers on GPUs

Fig. 3b presents two examples of using data-thread binding to realize a part of sorting network shown in Fig. 3a. Fig. 3b-1 shows

the most straightforward method of simply conducting all the computation within a single thread [3]. This option can exhibit better ILP but at the expense of requesting too many register resources. On the contrary, Fig. 3b-2 shows the example to bind one element to one thread [13]. Unfortunately, this method wastes computing resources, i.e., the first two threads perform the comparison on the same operands as the last two threads do. Therefore, we will investigate a more sophisticated solution that allows N-to-M data-thread binding (N elements binding to M threads) and evaluate the performance after applying this solution to different lengths of segments on different GPU architectures.

Another related but distinct issue is that even if we know the best N-to-M data-thread binding for a given segment on a GPU architecture, how to efficiently exchange data within/between threads is still challenging, especially at GPU register level. Different with the communications via the global and shared memory of GPU that have been studied extensively, data sharing through registers may require more research.

3 METHODOLOGY

3.1 Adaptive GPU SegSort Mechanism

The key idea of our SegSort mechanism is to construct relatively balanced work units to be consumed by a large amount of warps (i.e., a group of 32 threads in NVIDIA CUDA semantics) running on GPUs. Such work units can be a combination of multiple segments of small sizes, or part of a long segment split by a certain interval. To prepare the construction, we first group different lengths of segments into different bins, then combine or split segments for making the balanced work units, finally apply differentiated sort approaches in appropriate memory levels to those units.

Specifically, we categorize segments into four types of bins as shown in Fig. 4: (1) A unit bin, which segments only contain 1 or 0 element. For these segments, we simply copy them into the output in the global memory. (2) Several warp bins, which segments are short enough. In some warp bins, a segment will be processed by a single thread, while in others, a segment will be handled by several threads, but a warp of threads at most. That way, we only use GPU registers to sort these segments. This register-based sort is called **reg-sort** (Sec. 3.2), which allows the N-to-M data-thread binding and data communication between threads. Once these segments are sorted in registers, we write data from the registers to the global memory, and bypass the shared memory. To achieve the coalesced memory access, the sorted results may be written to the global memory in a striped manner after an in-register transpose stage (Sec. 3.4). (3) Several block bins, which consist of medium size segments. In these bins, multiple warps in a thread block cooperate to sort a segment. Besides of using the reg-sort method in GPU registers, a shared memory based merge method, called **smem-merge**, is designed to merge multiple sorted chunks from *reg-sort* in a segment (Sec. 3.3). After that, the merged results will be written into the output array from the shared memory to the global memory. As shown in the figure, the number of warps in the thread block is configurable. (4) A grid bin, designed for the sufficient long segments. For these segments, multiple blocks work together to sort and merge data. Different with the block bins, multiple rounds of **smem-merge** have to move data back and forth between the shared memory and the global memory to process these extremely long segments. In each round of calling **smem-merge**, the synchronization between multiple thread blocks is necessary: after the execution of *reg-sort* and *smem-merge* in each block, intermediate results need to be synchronized across all cooperative blocks via the global memory [47]. After that, the partially sorted data will be repartitioned and assigned to each block by using a similar partitioning method in *smem-merge*, and utilizing inter-block locality will in general further improve overall performance [27].

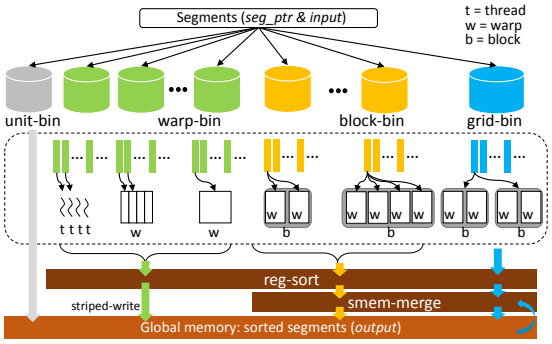


Figure 4: Overview of our GPU Segsort design

As shown in Fig. 4, the binning is the first step of our mechanism and a specific requirement of segmented sort compared to the standard sort. It is crucial to design an efficient binning approach on GPUs. Such an approach usually needs carefully designed histogram and scan kernels, such as those proposed in previous research [25, 47]. We adopt a simple and efficient "histogram-scan-bin" strategy generating the bins across GPU memory hierarchy. We launch a GPU kernel which number of threads is equal to the number of segments. Each thread will process one element in the *segs_ptr* array. *Histogram*: each thread has a boolean array as the predicates and the array length is equal to the number of total bins. Then, each thread calculates its segment length from *segs_ptr* and sets the corresponding predicate to *true* if the segment length is in the current bin. After that, we use the warp vote function `__ballot()` and the bit counting function `__popc()` to accumulate the predicates for all threads in a warp for the warp-level histogram. Finally, the first thread of each warp atomically accumulates the bin sizes in the shared memory to produce the block-level histogram, and after that the first thread in each block does the same accumulation in the global memory to generate the final histogram. *Scan*: We use an exclusive scan on the histogram bin sizes to get starting positions for each bin. *Binning*: all threads put their corresponding segment IDs to positions atomically obtained from the scanned histogram. With such highly efficient designs, the overhead of grouping is limited and will be evaluated in Sec. 4.2.

3.2 Reg-sort: Register-based Sort

Our *reg-sort* algorithm is designed to sort data in GPU registers for all bins. As a result, our method needs to support N-to-M data-thread binding, meaning M threads cooperate on sorting N elements. In order to leverage GPU registers to implement fast data exchange for sorting networks, M is set up to 32, which is the warp

size. For the medium and long segments where multiple warps are involved, we still use *reg-sort* to sort each chunk of a segment and use *smem-merge* to merge these sorted chunks. On the other hand, although our method theoretically supports any value of N, N is bound to the number of registers: if N is too large, the occupancy is degraded significantly because too many registers are used.

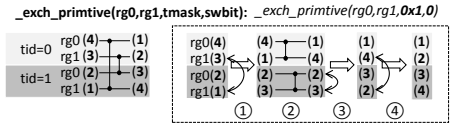


Figure 5: Primitive communication pattern and its implementation

Fig. 5 shows the data exchange primitive in the bitonic sorting network 3a with the details of implementation on GPU registers by using shuffle instructions. In this example, each thread holds two elements as the input: the thread 0 has 4 and 3 in its register rg0 and rg1; and the thread 1 has 2 and 1 accordingly. This situation corresponds to a 4-to-2 data-thread binding. The primitive is implemented in four steps. First, each thread needs to know the communicating thread by the parameter *tmask*. The line 12 of Alg. 1 shows how to calculate its value, which is equal to the current cooperative thread group size minus 1. In this example, M is 2 because there are two threads in a cooperative group, and *tmask* is 1 (represented as 0x1 in the figure). This step uses the shuffle instruction `__shfl_xor(rg1, 0x1)`[2] to shuffle data in rg1: the thread 0 gets data from the rg1 of thread 1, and similar to the thread 1. This step makes the data changed from "4, 3, 2, 1" to "4, 1, 2, 3". Second, each thread will compare and swap data in rg0 and rg1 locally to change the data to "1, 4, 2, 3". After the second step, the rg0 in each thread has the smaller data and the rg1 has the larger one. Thus, the third step is necessary to exchange the data in thread 1 to make the smaller data in its rg1 and the larger data in it rg0 for the following shuffle, because the shuffle instruction can only exchange data in registers having the same variable name, i.e., rg1. In a more general case where there are more threads are involved, we use the parameter *swbit* to control which threads need to execute this local swap operation. The line 13 of Alg. 1 shows how to calculate the value of *swbit*, which is equal to log coop_thrd_size − 1. In this case, the current cooperative thread group size is 2, the *swbit* is 0, and the thread 1 will do the swap when `bfe(1, 0)` returns 1. After the third step, we get "1, 4, 3, 2". Fourth, we shuffle again on rg1 to move the larger data of thread 0 to thread 1 and the smaller data of thread 1 to thread 0, and then get the output "1, 2, 3, 4".

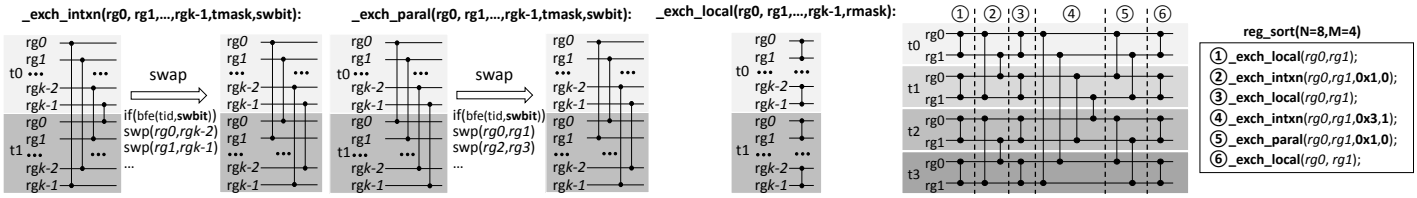
Based on the primitive, we extend two patterns of the N-to-M binding to implement the bitonic sorting network, which are (1) *exch_intxn*: The differences of corresponding data indices for comparisons decrease as the register indices increase in the first thread. (2) *exch_paral*: The differences of corresponding data indices for comparisons keep consistent as the register indices increase in the first thread. The left hand sides of Fig. 6a and Fig. 6b show these two patterns. In these figures, each thread handling k elements, where $k = N/M$ of the N-to-M data-thread binding. These two patterns can be easily constructed from the primitive *exch_primitive* by simply swapping corresponding registers in each thread locally as shown in the right hand side of Fig. 6a and Fig. 6b. Different with these two patterns involving the inter-thread communication, a third pattern in the N-to-M data-thread binding only has the intra-thread communication. As shown in Fig. 6d, we can directly use the compare and swap operation for the implementation without any shuffle-based inter-thread communication. We call this pattern *exch_local*. This pattern can be an extreme case of N-to-M binding, i.e., N-to-1, where the whole sorting network is processed by one thread. All of these three patterns are used in Alg. 1 to implement a general N-to-M binding and the corresponding data communication for the bitonic sorting network.

[2]The *shuffle* operation `__shfl_xor(v, mask)` lets the calling thread x obtain register data v from thread x ^ mask, and the operation `__shfl(v, y)` lets the calling thread x fetch data v from thread y.

Algorithm 1: Reg-sort: N-to-M data-thread binding and communication for bitonic sort

```
   /* segment size N, thread number M, workloads per thread wpt = N/M,
      regList is a group of wpt registers.                              */
1  int p = (int) log N;
2  int pt = (int) log M;
3  for l ←p; l >= 1; l-- do
4      int coop_elem_num = (int)pow(2, l − 1);
5      int coop_thrd_num = (int)pow(2, min(pt, l − 1));
6      int coop_elem_size = (int)pow(2, p − l + 1);
7      int coop_thrd_size = (int)pow(2, pt − min(pt, l − 1));
8      if coop_thrd_size == 1 then
9          int rmask = coop_elem_size − 1;
10         _exch_local(regList, rmask);
11     else
12         int tmask = coop_thrd_size − 1;
13         int swbit = (int) log coop_thrd_size − 1;
14         _exch_intxn(regList, tmask, swbit);
15     for k ← l + 1; k <=p; k + + do
16         int coop_elem_num = (int)pow(2, k − 1);
17         int coop_thrd_num = (int)pow(2, min(pt, k − 1));
18         int coop_elem_size = (int)pow(2, p − k + 1);
19         int coop_thrd_size = (int)pow(2, pt − min(pt, k − 1));
20         if coop_thrd_size == 1 then
21             int rmask = coop_elem_num − 1;
22             rmask = rmask − (rmask >> 1);
23             _exch_local(regList, rmask);
24         else
25             int tmask = coop_thrd_num − 1;
26             tmask = tmask − (tmask >> 1);
27             int swbit = (int)log coop_thrd_size − 1;
28             _exch_paral(regList, tmask, swbit);
```

The pseudo-codes are shown in Alg. 1, which essentially combine *exch_intxn* and *exch_paral* patterns (ln. 14 and ln. 28) in each iteration, and use *exch_local* when no inter-thread communication is needed. In each step where all comparisons can be performed in parallel, we group data elements as *coop_elem_num*, representing the maximum number of groups that the comparisons can be conducted without any interaction with elements in other group (ln. 4 and ln. 16); and the *coop_elem_size* represents how many elements in each group. For example, in the step 1 of Fig. 6d, the data is put into 4 groups, each of which has 2 elements. Thus, *coop_elem_num* is 4 and *coop_elem_size* is 2. In the step 4 of this figure, there is only 1 group with the size of 8 elements. Thus, *coop_elem_num* is 1 and *coop_elem_size*is 8. Similarly, the algorithm groups threads into *coop_thrd_num* and *coop_thrd_size* for each step to represent the maximum number of cooperative thread groups and the number of threads in each group. For example, the step 1 of Fig. 6d has 4 cooperative thread groups and each group has 1 thread. Thus,



(a) *exch_intxn* pattern (b) *exch_paral* pattern (c) *exch_local* pattern (d) An example of bitonic sorting where $N = 8$ and $M = 4$

Figure 6: Generic *exch_intxn*, *exch_paral* and *exch_local* patterns, shown in (a,b,c). The transformed patterns can be easily implemented by using primitives. A code example of Alg. 1 for is shown in (d)

coop_thrd_num is 4 and *coop_thrd_size* is 1. In contrast, the step 4, *coop_thrd_num* is 1 and *coop_thrd_size* is 4 that means the 4 threads need to communicate with each other to get required data for the comparisons in this step.

If there is only one thread in a cooperative thread group (ln. 8 and ln. 20), the algorithm will switch to the local mode *exch_local* because the thread already has all comparison operands. Once there are more than one thread in a cooperative thread group, the algorithm uses *exch_intxn* and *exch_paral* patterns and calculates corresponding `tmask` and `swbit` to determine the thread for the communication pair and the thread that needs the local data rearrange aforementioned in the previous paragraph. In the step 1 of this figure, where *coop_thrd_size* is 1 and *exch_local* is executed, the algorithm calculates `rmask`, which controls the local comparison on registers (ln. 9). In the step 4, where *coop_elem_size* is 8 and *coop_thrd_size* is 4, all 8 elements will be compared across all 4 threads. In this case, the `tmask` is 0x3 (ln. 12) and `swbit` is 1 (ln. 13). In the step 5 where the *exch_paral* pattern is used, the algorithm calculates *coop_elem_size* is 4 and *coop_thrd_size* is 2. Thus, the `tmask` is 0x1 and `swbit` is 0. Note that although our design in Alg. 1 is for bitonic sorter, our ideas are also applicable to other sorting networks by swapping and padding registers based on the primitive pattern.

3.3 Smem-merge: Shared Memory-based Merge

As shown in Fig. 4, for medium and large sized segments in the block bins and grid bin, multiple warps are launched to handle one segment and the sorted intermediate data from each warp need to merge. The *smem-merge* algorithm is designed to merge such data in the shared memory.

Our *smem-merge* method enables multiple warps to merge chunks having different numbers of elements. We assign first m warps with *x-warp_size* and the last m' warps with *y-warp_size* to keep load balance between warps as possible as we can. Inside each warp, we also try to keep balance among cooperative threads in the merge. We design a searching algorithm based on the MergePath algorithm [17] to search the splitting points to divide the target segments into balanced partitions for each threads.

Fig. 7 shows an example of *smem-merge*. In this case, there are 4 sorted chunks in the shared memory belonging to a segment. They have 4, 4, 4, and 5 elements, respectively. We assume each warp has 2 threads. As a result, the first 3 warps will merge 4 elements, and each thread in these warps will merge two elements; while the last warp will work on 5 elements, and the first thread will merge 2 elements and second thread will merge 3 elements. In the

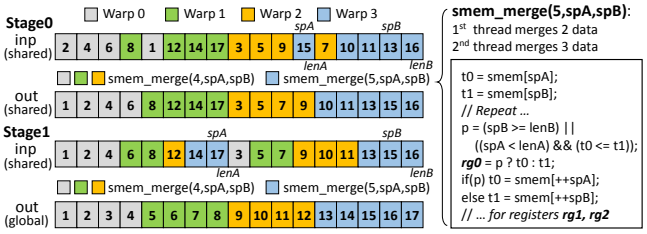


Figure 7: An example of warp-based merge using shared memory

figure, the splitting points `spA` and `spB` for the second thread of warp 3 (in blue color) are computed by the MergePath algorithm. The right part of the figure shows the merge codes executed by this thread that process 3 elements. Our merge method first loads data from `spA` and `spB` to two temporary registers `t0` and `t1`. By checking if `spA` and `spB` is out-of-bound and comparing `t0` and `t1`, the merge algorithm selects the smaller element to fill first result register `rg0`. The algorithm continues loading the next element to fill `t0` or `t1` from corresponding chunks pointed by `spA` or `spB`, until assigned number of elements is encountered. After that, the merged data in registers, e.g., *rg0, rg1* for first thread, and *rg0, rg1, rg2* for the second thread, will be stored back to shared memory for another iteration of merge. As shown in the figure, two iterations of *smem-merge* are used to merge four chunks belonging to one segment.

3.4 Other Optimizations

Load data from global memory to registers: we decouple the data load from global memory to registers and the actual segmented sort in registers. Our data load kernel uses successive threads to load contiguous data for coalesced memory access. Although we don't keep the original global indices of input data in registers, that doesn't matter because the input data is unsorted and the global indices are not critical for the following sort routine.

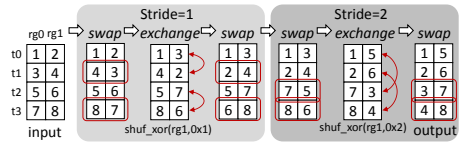
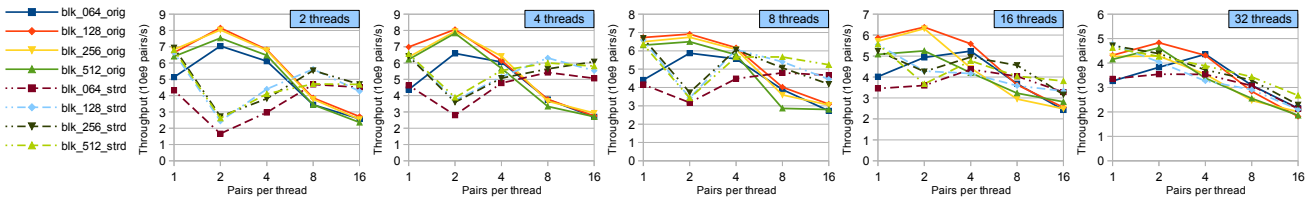
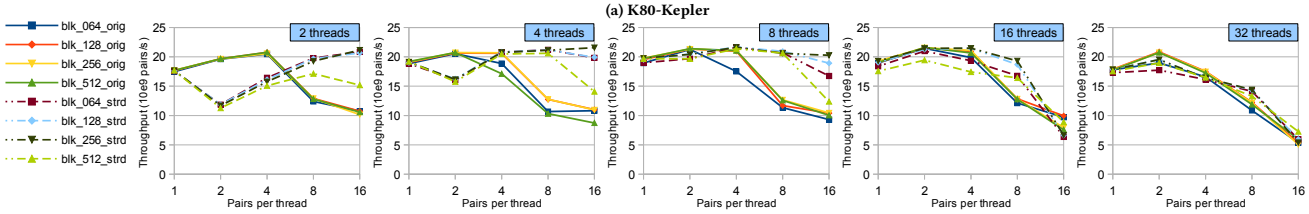


Figure 8: An example of in-register transpose

Store data from registers to global memory: when the segments in the warp bins are sorted, we directly store them back to global memory without the merge stage. However, because the sorted data may distributed into different registers of threads in a



(a) K80-Kepler



(b) TitanX-Pascal

Figure 9: Performance of reg-sort routines with different combinations of data-thread binding policies, write methods, and block sizes

warp, directly storing them back will lead to uncoalesced memory access. When the number of elements per thread grows, the situation will become even worse. Therefore, as well as keeping the direct store back method, which is labeled as *orig* in our evaluation, we design the **striped write** method, which is labeled as *strd* in the evaluation. We implement an in-register transpose method to scatter the sorted data in registers of threads. The transpose method starts from shuffling registers by the stride of 1, then doubles the stride to 2, and finishes the shuffles until $\log 32 = 5$ iterations, where 32 is the warp size. After that, the successive threads can write data to global memory in a cyclic manner for coalesced memory access. Fig. 8 shows an example of using 4 threads to transpose data in their two registers. This example shows that the number of iterations for the in-register transpose depends on the number of threads but not on the number of elements each thread has[3]. As a result, after $\log 4 = 2$ iterations, the successive elements are scattered to these threads. In the evaluation, we will investigate the best scenarios for *orig* and *strd*, considering the *orig* method has the uncoalesced memory access problem, while the *strid* method has the in-register transpose overhead.

4 PERFORMANCE RESULTS

We conduct the experiments on two generations of NVIDIA GPUs K80-Kepler and TitanX-Pascal. Tab. 1 lists the specifications of the two platforms. The input dataset is two arrays holding key and value pairs separately. The total dataset size is fixed at 2^{28} and segment numbers are varied accordingly to the target segment sizes. We report the throughput in the experiments equal to $2^{28}/t$ pairs/s, where t is the execution time.

Table 1: Experiment Testbeds

	Tesla K80 (Kepler-GK210)	TitanX (Pascal-GP102)
Cores	2496 @ 824 MHz	3584 @ 1531 MHz
Register/L1/LDS per core	256/16/48 KB	256/16/48 KB
Global memory	12 GB @ 240 GB/s	12 GB @ 480 GB/s
Software	CUDA 7.5	CUDA 8.0

[3]The number of elements each thread has will determine how many shuffle instructions are needed in each iteration.

4.1 Kernel Performance

For the reg-sort kernels, we alternate the thread group size M in {2, 4, 8, 16, 32}. At the same time, each thread varies its bound data N/M in {1, 2, 4, 8, 16} (N/M is labeled as pairs per thread *ppt* in figures). We use the maximum bound data of 16 because we observe the performance deteriorates obviously for higher numbers than 16. Thus, the target segment size N is a variable number ranging from 2 (i.e., 2 threads each bind 1 pair) to 512 (i.e., 32 threads each bind 16 pairs). Since reg-sort might exhaust register resources, we also vary the block sizes as 64, 128, 256, and 512 for maximizing occupancy. Fig. 9 shows the diversified performance numbers of reg-sort kernels, which actually demonstrates that choosing a single solution for all segments would lead to suboptimal performance even among GPUs from the same vendor.

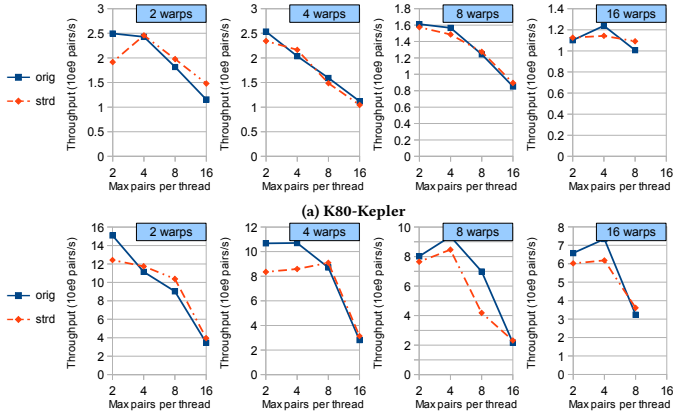
In Fig. 9, we notice that the impact of data-thread binding policies varies widely depending on the GPU devices. For example, when the segment size N equals to 16, the possible candidates are 2(threads):8(ppt), 4:4, 8:2, and 16:1. On K80-Kepler GPU, the highest performance is given by 8:2, achieving 30% speedups over the slowest policy of 2:8. In contrast, the TitanX-Pascal GPU shows very similar performance for these policies. This insensitivity to register resources exhibited on Pascal architecture can contribute to its larger available register files per thread. Note, the policies of each thread binding only 1 pair is equivalent to the method proposed in [13]. This method actually wastes computing resources, because each comparison over two operands is conducted twice by two threads, resulting in suboptimal performance numbers.

The striped write method (*strd*) in reg-sort kernels is particularly effective when each thread binds more than 4 pairs. This is because when the ppt is small (<4), consecutive threads can still access almost consecutive memory locations for coalesced memory transaction. However, larger ppt indicates the thread access locations are highly scattered, causing inefficiently memory transaction instead. In this case, the striped write method is of great necessity. On the other hand, the block size also has the effect on the performance and the optimal one is usually achieved by using 128 or 256 threads.

For the smem-merge kernels, Fig. 10 shows the performance numbers of changing the number of cooperative warps and ppt. The

warp numbers are in {2, 4, 8, 16}, while the ppt varies among {2, 4, 8, 16}. In this scenario, the target segment size N ranges from 128 (i.e., 2 warps each merge 32x2 pairs) to 4096 (i.e., 16 warps each merge 32x8 pairs). Differed from reg-sort kernels, the best data-thread binding policies are more consistent for smem-merge on the two devices. For example, to merge 1024-length segment, both devices prefer to use 8 warps with 4 ppt. Considering memory access, the striped method provides similar performance with original method, which directly exploits random access of shared memory in order to achieve coalesced transaction on global memory. Note, in our implementation, we carefully select shared memory dimensions and sizes to avoid band conflicts.

Now, we can select best kernels for different segments with length of the power of two. Other segments can be handled by directly padding them to the nearest upper power of two in reg-sort kernels. For smem-merge kernels, we use different ppt for different warps to minimize padding values. Since our method is based on the pre-defined sorting networks, the characteristics of input datasets will cause negligible to the selection of best kernels. Moreover, for each GPU, we only need to conduct the offline selection once. The results show that we only need 13 bins to handle all the segments. The first 8 and 9 bins use reg-sort to handle up to 256 pairs on Kepler GPU and 512 on Pascal respectively. Then, other segments less than 2048 can be handled by smem-merge kernels using 3 and 2 bins on the two platforms. Finally, our grid-bin kernels can efficiently sort the segments longer than 2048, which are all assigned to the last bin.



(a) K80-Kepler

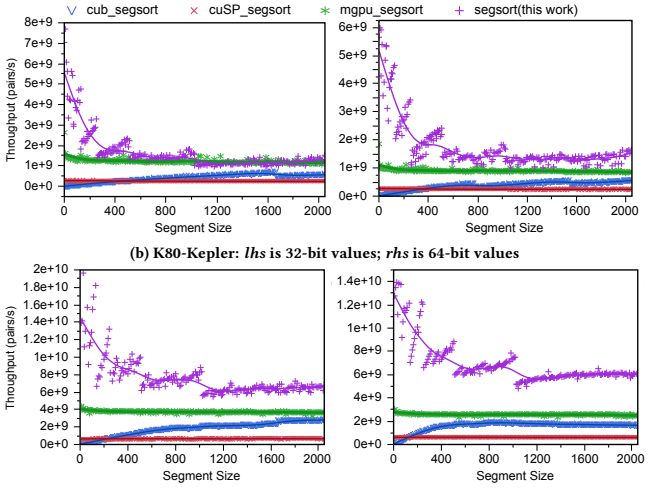
(b) TitanX-Pascal

Figure 10: Performance of smem-merge routines with different combinations of data-thread binding policies and write methods. The results for 16 warps with 16 ppt are not available due to exhaustion of shared memory resources

4.2 Segmented Sort Performance

We conduct performance comparison of our best kernels over three existing tools: (1) *cusp-segsort* from CUSP [10] library that extends the segment pointer *seg_ptr* to form an another layer of primary keys, attaches it to the input keys and values, and performs the global sort from Thrust library [20] on the new data structure; (2) *cub-segsort* [37] that assigns each block to order a segment and uses radix sort scheme; and (3) *mgpu-segsort* [3] that evolves from the global merge sort with runtime segment delimiter checking, thus can ensure the sort only occurs within segment.

Datasets of uniform distribution: We test a batch of uniform segments to evaluate the performance of our segsort kernels, which are plotted in Fig. 11. Since the cusp-segsort and mgpu-segsort are designed from the global sort, their performance is determined by the total input size. This "one-size-fit-all" philosophy ignores the characteristics of the segments and thus shows the plateau performance. For the short segments (<256 on Kepler and <512 on Pascal), our segsort can achieve an average of 13.4x and 3.1x speedups over cusp-segsort and mgpu-segsort respectively on Kepler. These speedups rise to 15.5x and 3.2x on Pascal. For the other segments, our segsort provides an average of 5.6x and 1.2x improvements on Kepler (10.0x and 2.1x on Pascal). The performance improvements are mainly from our reg-sort and smem-merge, which minimize shared/global memory transactions.



(b) K80-Kepler: *lhs* is 32-bit values; *rhs* is 64-bit values

(c) TitanX-Pascal: *lhs* is 32-bit values; *rhs* is 64-bit values

Figure 11: Performance of different segsort over segments with uniform distribution

In contrast, although cub-segsort conducts a more "real" segmented sort with each block working on one segment, this strategy falls short when the segments are of great amount and of short lengths. The maximum number of segments can be processed in parallel in cub-segment is only 65535 (limitation of gridDim.x), which requires multiple rounds of calling if the segment number is too large. Furthermore, assigning a block to handle one segment may waste computing resources severely, especially when the segments are very short. Thus, our segsort can achieve an average of 211.2x and 30.4x speedups on Kepler and Pascal devices respectively. As the segment size increases, we can still keep 3.7x and 3.2x average improvements on the two platforms. Note, the staircase-like performance of our segsort is caused by the padding used in our method.

Datasets of power-law distribution: In this test, we use a collection of synthetic power-low data. Since segment lengths are non-uniform, we include binning overhead and use overall wall time for our segsort. The data generation tool is from PowerGraph [16] and its generated samples follow a Zipf distribution [1]. The equation of $P(l) \propto l^{-\alpha}$ shows the probability of segments with length l is

proportional to $l^{-\alpha}$, where α is a positive number. This implies that the higher α will result in high skewness to shorter segments. For different segment bounds, we vary the skewness to test our segsort. We vary α from 0.1 to 1.6 (with stride of 0.1) and limit maximum segment size from 50 to 2000 (with stride of 50). Therefore, each method is tested with 640 sampling points of different parameter configurations.

Fig. 12 plots the speedups of our segsort over the existing tools on both 32-bit and 64-bit values. For cusp-segsort and mgpu-segsort, we fix the total key-value pairs as 2^{28}. However, for the cub-segsort, we set the segment number to 65535. Otherwise, multiple rounds of calling are required. Fig. 12(a,b) show the speedups of our segsort over the cub-segsort. Because of high cost for radix sort on short segments, our segsort can provide significant speedups, reaching up to 63.3x and 86.1x (top-left corner). For longer segment lengths with power-law distribution, our method can keep 1.9x speedups on both GPU devices due to better load balancing strategy.

Compared to cusp-segsort in Fig. 12(c,d), our segsort achieves up to 12.5x and 16.5x improvements on Kepler and Pascal, and compared with mgpu-segsort in Fig. 12(e,f), our method gets up to 3.0x and 3.8x performance gains. In both situations, we can notice that the top-left corners are blue, indicating less speedups (vs. cusp-segsort) or similar performance (vs. mgpu-segsort). This is because that the condition of $\alpha = 1.6$ and segment bound of 50 make almost all the segments to be 1. Thus, the segmented sorts become a copying procedure and exhibit the similar performance. On the other hand, we observe that the Pascal shows higher performance benefits over Kepler. The reasons are two-fold: faster atomic operations for binning and larger register files for sorting. Moreover, for the 64-bit values, we usually get higher performance gains compared to 32-bit values. This is mainly because we can handle the permutation indices more efficiently in the registers rather than the shared memory or global memory. To evaluate the binning overhead, we calculate the arithmetic mean for the ratio of binning to overall kernel time, which are only 5.9% for Kepler and 3.4% for Pascal.

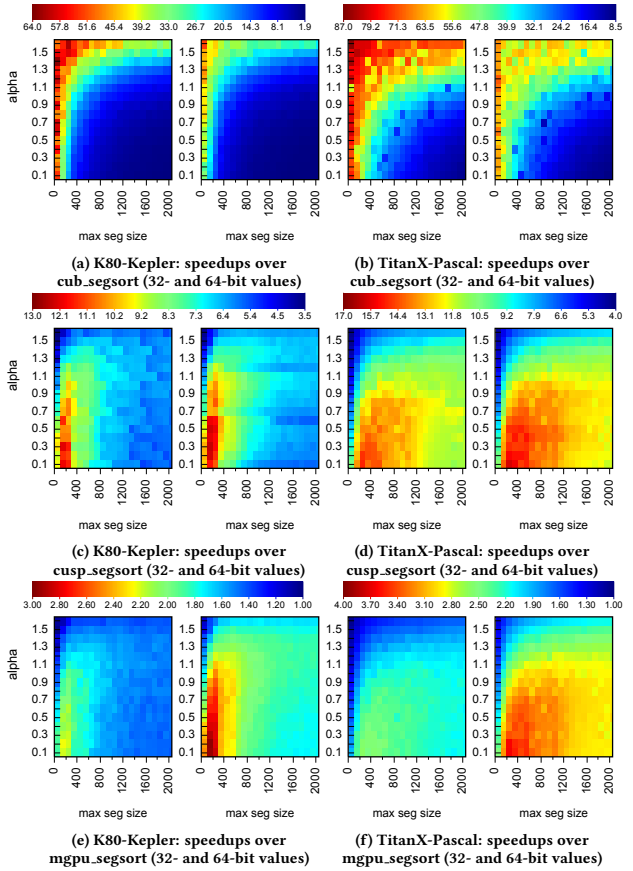


Figure 12: Segmented sort v.s. existing tools over segments of power-law distribution

5 SEGSORT IN REAL-WORLD APPLICATIONS

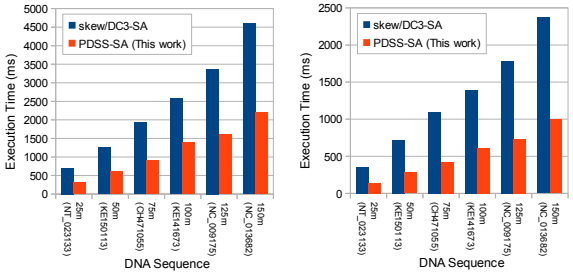
In order to further evaluate our approach, we choose two real-world applications, characterized by skewed segment distribution: (i) The suffix array construction to solve problems of pattern matching, data compression, etc. in text processing and bioinformatics [15, 44]. (ii) The sparse general matrix-matrix multiplication (SpGEMM) to solve graph and linear solver problems, e.g., sub-graphs, shortest paths, and algebraic multigrid [4, 8, 41]. We use our approach to optimize the applications and compare the results with state-of-the-art tools, i.e., skew/DC3-SA [19, 44], ESC(CUSP) [4, 10], cuSPARSE [35], and bhSPARSE [30].

5.1 Suffix Array Construction

The suffix array stores lexicographically sorted indices of all suffixes of a given sequence. Our approach for the suffix array construction is based on the *prefix doubling* algorithm [32] with the computational complexity of $O(Nlog(N))$, where N is the length of input sequence. The main idea is that we can deduce the orders of two same $2h$-size strings S_i and S_j, if the orders of all h-size strings are known, which is stored in an unfinished suffix array h-SA. For example, we treat S_i as two concatenated h-size prefixes S_{ia} and S_{ib}. Similarly, S_j is split into two S_{ja} and S_{jb}. Then, by looking up the known h-SA, the comparison rule for S_i and S_j becomes: if the prefix S_{ia} differs from S_{ja}, we can directly determine the order of S_i and S_j accordingly; otherwise, we need to check the order of S_{ib} and S_{jb}. That way, if they are different, the order of S_i and S_j can also be induced. However, if they are same, we have to mark S_i and S_j unsolved and put them under the same category (i.e., updating h-SA to $2h$-SA with position i and j storing the same value). The ordering will proceed for $O(log(N))$ iterations by doubling the prefix length. This is a segmented sort with the customized comparison, and a segment corresponds to a category that contains a group of suffixes whose orders are not determined.

We use our method to optimize the prefix doubling, i.e., *PDSS-SA*, and use the baseline from the cuDPP library [19], which is based on the DC3/skew algorithm on GPUs [44]. Fig. 13 presents the performance comparison over six DNA sequences of different lengths from NCBI NR datasets [34]. Our PDSS-SA can provide up to 2.2x and 2.6x speedups over the baseline on the K80-Kepler and TitanX-Pascal platforms, respectively. The improvement can be explained in two folds. First, although the baseline takes a linear time algorithm, the prefix doubling exhibits more ease of performing parallelization, e.g., global radix sort, prefix sum, etc.

Second, for the dominant kernels (taking up 30% to 60% of total time), we use efficient segmented sort to handle the considerable amounts of short segments iteratively, while the baseline uses more expensive sort and merge kernels recursively.



(a) K80-Kepler (b) TitanX-Pascal

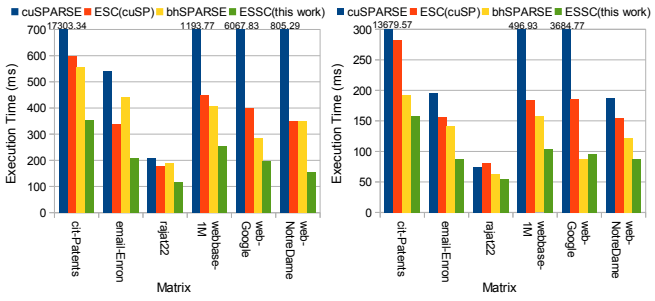
Figure 13: Performance of suffix array construction using our segmented sort

5.2 Sparse Matrix-Matrix Multiplication

The SpGEMM operation multiplies a sparse matrix A with another sparse matrix B and obtains a resulting sparse matrix C. This operation may be the most complex routine in sparse basic linear algebra subprograms because all the three involved matrices are sparse. The *expansion*, *sorting* and *compression* (ESC) algorithm developed by Bell et al. [4] is one of several representative methods that aim to utilize wide SIMD units for accelerating SpGEMM on GPUs [4, 30, 35]. The ESC method includes three stages: (1) expanding all candidate nonzero entries generated by the necessary arithmetic operations into an intermediate sparse matrix $\widehat{C}$, (2) sorting $\widehat{C}$ by its indices of rows and columns, and (3) compressing $\widehat{C}$ into the resulting matrix C by fusing entries with duplicate column indices in each row.

We use the ESC SpGEMM in the CUSP library [10] as the baseline, and replace its original sort in the second stage of its ESC implementation with our segmented sort by mapping row-column information of the intermediate matrix $\widehat{C}$ to segment-element data in the context of segmented sort. Since we have segmented sort instead of sort in the ESC method, we call our method ESSC (*expansion*, *segmented sorting* and *compression*). Note that to better understand the effectiveness of segmented sort for SpGEMM, all other stages of the SpGEMM code remain unchanged. We select six widely used sparse matrices *cit-Patents*, *email-Enron*, *rajat22*, *webbase-1M*, *web-Google* and *web-NotreDame* from the University of Florida Sparse Matrix Collection [12] as our benchmark suite. Squaring those matrices will generate a $\widehat{C}$ including rows in power-law distribution. We also include two state-of-the-art SpGEMM methods in cuSPARSE [35] and bhSPARSE [30] into our comparison.

Fig. 14 plots the performance of the four participating methods. It can be seen that our ESSC method achieves up to 2.3x and 1.9x speedups over the ESC method on the Kepler and Pascal architectures, respectively. The performance gain is from the fact that the sorting stage takes up to 85% (from 45%) overall cost of ESC SpGEMM, and our segmented sort is up to 8x (from 3.2x) faster than the sort method used in CUSP. Compared with the latest libraries cuSPARSE and bhSPARSE, our ESSC method brings performance improvement of up to 86.5x and 2.3x, respectively. The performance gain is mainly from the irregularity of the row distribution of $\widehat{C}$.



(a) K80-Kepler (b) TitanX-Pascal

Figure 14: Performance of SpGEMM using our segmented sort

6 RELATED WORK

The sorting kernel has received much attention due to the pervasive need to order data in a plethora of applications. It has been parallelized and optimized on x86-based architectures [9, 22] and GPUs [26, 33, 38, 38, 40]. Several optimized sort implementations have been included in vendor supplied libraries, e.g., cuDPP [19], Thrust [20], ModernGPU [3], and CUB [37]. These methods mainly focus on the global sort over a complete input array, and extend the global sort for segmented sort with sub-optimal performance.

Specifically, current GPU segmented sort solutions fail to take advantage of both data distribution and architecture, because most of them [10, 15, 37] adopt a "one-size-fits-all" philosophy that treats different segments equally. The mechanisms in [10, 15] sort the whole array after extending input with segment IDs as primary keys, which will consume extra memory space and result in an increased computational complexity. Another mechanism [37] uses one thread block to handle a segment, no matter how different the segments are. It will give rise to some deficiencies when processing a numerable batch of short segments, due to the resource under-utilization. Many GPU applications [30, 44, 50] reformulate the segmented sort problem in terms of global sort and call APIs supported by libraries, sacrificing the benefits of segmentation.

There are many research efforts aimed at designing parallel solutions for a large amount of small independent problems via segmented data structure, e.g., segmented scan [6], segmented sum [7, 31], and batched BLAS [18] in the MAGMA library [2]. Due to their high cost for global operations and "one-size-fits-all" execution pattern, we believe our methods can be applied on these segmented problems to improve their performance.

Many studies investigate the data-level parallelism on x86-based systems [21, 23, 36, 42]. Correspondingly, several studies have illustrated the benefits of using registers to improve performance on GPUs. Demouth [13] has presented a set of shuffle based kernels, e.g., the bitonic sort; while this method maps an element to a thread, leading to the waste of computing resource and the poor ILP. Ben-Sasson et al. [5] proposes a fast multiplication in binary field by using registers as cache. Hou et al. [24] use registers for stencil applications on both AMD and NVIDIA GPUs. Davidson and Owens [11] use registers to speedup a downsweep patterned computation. Distinguished from the above work, our method is more general to support the N-to-M data-tread binding and can be extended to support other kernels using GPU registers.

7 CONCLUSION

In this paper, we have presented an efficient segmented sort mechanism that adaptively combine or split segments of different sizes for load balanced processing on GPUs, and have proposed a register-based sort algorithm with N-to-M data-thread binding and in-register communication for fast sorting networks on multiple memory hierarchies. The experimental results illustrate that our mechanism is greatly faster than existing segmented sort methods in vendor support libraries on two generations of GPUs. Furthermore, our approach improves overall performance of applications SAC from bioinformatics and SpGEMM from linear algebra.

8 ACKNOWLEDGEMENTS

The work has been supported in part by the NSF-BIGDATA program via IIS-1247693. The authors also acknowledge Advanced Research Computing at Virginia Tech for providing computational resources. This work also has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 752321.

REFERENCES

[1] L. A. Adamic and B. A. Huberman. 2002. Zipf's Law and the Internet. *Glottometrics 3* (2002).
[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. 2009. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA projects. *J. of Physics: Conf. Series* (2009).
[3] S. Baxter. *ModernGPU2.0: A Productivity Library for General-purpose Computing on GPUs.* https://github.com/moderngpu/moderngpu.
[4] N. Bell, S. Dalton, and L. N. Olson. 2012. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM J. on Scientific Computing* (2012).
[5] E. Ben-Sasson, M. Hamilis, M. Silberstein, and E. Tromer. 2016. Fast Multiplication in Binary Fields on GPUs via Register Cache. In *ACM Int'l Conf. on Supercomputing (ICS).*
[6] G. E. Blelloch. 1989. Scans As Primitive Parallel Operations. *IEEE Trans. Comput.* (1989).
[7] G. E. Blelloch, M. A. Heroux, and M. Zagha. 1993. *Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors.* Technical Report.
[8] T. M. Chan. 2007. More Algorithms for All-pairs Shortest Paths in Weighted Graphs. In *ACM Symp. on Theory of Computing (STOC).*
[9] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. 2008. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *Proc. of the VLDB Endow. (PVLDB)* (2008).
[10] S. Dalton, N. Bell, L. Olson, and M. Garland. 2014. CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. (2014). http://cusplibrary.github.io/ V0.5.0.
[11] A. Davidson and J. D Owens. 2011. Register Packing for Cyclic Reduction: A Case Study. In *ACM Workshop on General Purpose Processing on GPUs (GPGPU).*
[12] T. A. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* (2011).
[13] J. Demouth. 2013. Shuffle: Tips and Tricks. (2013). GTC'13 Presentation.
[14] I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu. 2016. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *IEEE/ACM Int'l Symp. on Microarchitecture (MICRO).*
[15] P. Flick and S. Aluru. 2015. Parallel Distributed Memory Construction of Suffix and Longest Common Prefix Arrays. In *ACM Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC).* ACM.
[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI).*
[17] O. Green, R. McColl, and D. A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *ACM Int'l Conf. of Supercomputing (ICS).*
[18] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. 2015. Batched Matrix Computations on Hardware Accelerators Based on GPUs. *Int. J. High Perform. Comput. Appl. (IJHPCA)* (2015).
[19] M. Harris, J. D. Owens, S. Sengupta, S. Tzeng, Y. Zhang, A. Davidson, R. Patel, L. Wang, and S. Ashkiani. 2016. CUDPP: CUDA Data-Parallel Primitives Library. (2016). http://cudpp.github.io/ V2.3.
[20] J. Hoberock and N. Bell. 2015. *Thrust: A Parallel Algorithms Library.* https://thrust.github.io/.
[21] K. Hou, H. Wang, and W.-c. Feng. 2014. Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study. In *Int'l Conf. on Parallel Processing Workshops (ICPPW).*
[22] K. Hou, H. Wang, and W.-c. Feng. 2015. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In *ACM Int'l Conf. on Supercomputing (ICS).*
[23] K. Hou, H. Wang, and W.-c. Feng. 2016. AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS).*
[24] K. Hou, H. Wang, and W.-c. Feng. 2017. GPU-UniCache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs. In *ACM Int'l Conf. on Computing Frontiers (CF).*
[25] W. Jung, J. Park, and J. Lee. 2014. Versatile and Scalable Parallel Histogram Construction. In *ACM Int'l Conf. on Parallel Architectures and Compilation (PACT).*
[26] N. Leischner, V. Osipov, and P. Sanders. 2010. GPU Sample Sort. In *IEEE Int'l Symp. on Parallel Distributed Processing (IPDPS).*
[27] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*
[28] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter. 2016. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *Parallel Processing: Int'l Conf. on Parallel and Distributed Computing (Euro-Par).* Springer Berlin Heidelberg.
[29] W. Liu and B. Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ACM Int'l Conf. on Supercomputing (ICS).*
[30] W. Liu and B. Vinter. 2015. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. *J. Parallel Distrib. Comput. (JPDC)* (2015).
[31] W. Liu and B. Vinter. 2015. Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. *Parallel Comput. (ParCo)* (2015).
[32] U. Manber and G. Myers. 1993. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. on Computing* (1993).
[33] D. Merrill and A. Grimshaw. 2011. High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing. *Parallel Processing Letters* (2011).
[34] NCBI. *Genbank.* ftp://ftp.ncbi.nlm.nih.gov/genbank
[35] NVIDIA. 2016. cuSPARSE library. (2016). https://developer.nvidia.com/cuSPARSE
[36] B. Ren, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni. 2017. Exploiting Vector and Multicore Parallelism for Recursive, Data- and Task-Parallel Programs. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP).*
[37] NVIDIA Research. 2016. *CUB 1.6.4.* http://nvlabs.github.io/cub/.
[38] N. Satish, M. Harris, and M. Garland. 2009. Designing Efficient Sorting Algorithms for Manycore GPUs. In *IEEE Int'l Symp. on Parallel Distributed Processing (IPDPS).*
[39] J. Shun, G. E Blelloch, J. T Fineman, P. B Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA).*
[40] E. Sintorn and U. Assarsson. 2008. Fast Parallel GPU-sorting Using a Hybrid Algorithm. *J. Parallel Distrib. Comput. (JPDC)* (2008).
[41] V. Vassilevska, R. Williams, and R. Yuster. 2010. Finding Heaviest H-subgraphs in Real Weighted Graphs, with Applications. *ACM Trans. Algorithms* (2010).
[42] H. Wang, W. Liu, K. Hou, and W.-c. Feng. 2016. Parallel Transposition of Sparse Data Structures. In *ACM Int'l Conf. on Supercomputing (ICS).*
[43] J. Wang and S. Yalamanchili. 2014. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *IEEE Int'l Symp. on Workload Characterization (IISWC).*
[44] L. Wang, S. Baxter, and J. D. Owens. 2015. Fast Parallel Suffix Array on the GPU. In *Parallel Processing: Int'l European Conf. on Parallel and Distributed Computing (Euro-Par).* Springer Berlin Heidelberg.
[45] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO).*
[46] H. Wu, D. Li, and M. Becchi. 2016. Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS).*
[47] S. Yan, G. Long, and Y. Zhang. 2013. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP).*
[48] Y. Yang and H. Zhou. 2014. CUDA-NP: Realizing Nested Thread-level Parallelism in GPGPU Applications. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP).*
[49] Y. Yuan, R. Lee, and X. Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. of the VLDB Endow. (PVLDB)* (2013).
[50] J. Zhang, H. Wang, and W.-c. Feng. 2016. cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on CPU+GPU. *IEEE/ACM Trans. on Computational Biology and Bioinformatics (TCBB)* (2016).
[51] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of In-memory Key-value Stores. *Proc. of the VLDB Endow. (PVLDB)* (2015).