Improving Scheduling for Irregular Applications with Logarithmic Radix Binning

James Fox[1], Alok Tripathy[1], and Oded Green[1,2]

[1]Computational Science and Engineering, Georgia Institute of Technology - USA
[2]NVIDIA Corporation

***Abstract*— Effective scheduling and load balancing of applications on massively multi-threading systems remains challenging despite decades of research, especially for irregular and data dependent problems where the execution control path is unknown until run-time. One of the most widely used load-balancing schemes used for data dependent problems is a parallel prefix sum (PPS) array over the expected amount of work per task, followed by a partitioning of tasks to threads. While sufficient for many systems, it is not ideal for massively multi-threaded systems with SIMD/SIMT execution, such as GPUs. More fine-grained load-balancing is needed to effectively utilize SIMD/SIMT units. In this paper we introduce Logarithmic Radix Binning (LRB) as a more suitable alternative to parallel prefix summation for load-balancing on such systems. We show that LRB has better scalability than PPS for high thread counts on Intel's Knight's Landing processor and comparable scalability on NVIDIA Volta GPUs. On the application side, we show how LRB improves the performance of PageRank up to $1.75X$ using the branch-avoiding model. We also show how to better load-balance segmented sort and improve performance on the GPU.**

I. Introduction

As the number of cores and threads in accelerators continues to grow, so does the challenge of effectively load-balancing and fully utilizing the system. For regular applications such as dense matrix multiplication, where control flow is easily determined prior to execution, it can be very beneficial to find an optimal schedule and data arrangement prior to execution. Irregular applications such as graph algorithms and sparse matrix multiplication are more challenging, as control flow is typically highly data-dependent. Thus, cost-effective runtime-time scheduling and load-balancing is necessary. In this paper we present an efficient load-balancing mechanism that makes the execution of irregular and data dependent problems closer to regular problems where system utilization is typically high.

Scaling data dependent problems to systems with thousands of cores typically requires breaking the work down to its smallest units. As these tasks are typically very short, it is also necessary that the load-balancing mechanism add little overhead. It is not surprising that mechanisms such as prefix summations arrays are used to partition the workload across the threads [2], [14] due to its linear time complexity. Using prefix sum arrays on massively multi-threaded systems such as NVIDIA GPUs typically requires additional load-balancing [22], [10], [6] to ensure that all threads are properly utilized. Work stealing has also been used for graph applications [19] and sorting [8]. While effective for smaller thread counts, these approaches do not necessarily scale to systems with many thousands of threads, such as accelerators or processors with SIMD/SIMT execution models.

In two recent papers [13] (for the CPU with SIMD programming) and [9] (for the GPU and SIMT programming) we introduced a load-balancing technique for triangle counting called Logarithmic Radix Binning (LRB). Using LRB, we achieved near linear scaling on Intel's Knight's Landing (KNL) processor with its AVX-512 instruction set and showed improved performance on the GPU. Originally, LRB was designed as a load-balancing mechanism for triangle counting and counting common neighbors in adjacency list intersections. In this paper we show that LRB is not limited to triangle counting and can be used to load-balance a broader range of problems, including PageRank and segmented sorting. Segmented sorting can be used to sort rows of a sparse matrix or graph. We explore these new formulations and show that LRB improves their performance while being cost effective.

Algorithmic Contributions

- We extend LRB from its original formulation, used in triangle counting, to a broader form that can be applied to a wider range of problems.
- We show that LRB is an effective method for load-balancing irregular applications in both SIMD and SIMT programming models. Unlike with prefix sum, LRB is designed to give predictable load-balancing at the vector lane and thread granularity.
- We show that the overhead introduced by LRB load-balancing is relatively low. Its time complexity is similar to that of prefix summation. Empirically, computing the LRB reordering is competitive with and sometimes faster than computing a simple prefix sum array. Additionally, LRB does not incur the overhead of fully sorting a work array prior to PPS, an load-balancing alternative.

Performance Contributions

- On the GPU, LRB reordering is anywhere from 10% slower and up to $4.5\times$ faster than computing a prefix sum array. On the CPU, for smaller threads counts LRB is roughly $2-5\times$ slower than prefix summation. However, LRB scales better to large thread counts and is up to $2\times$ faster.

• We show how to apply LRB and vector instructions to improve the performance of PageRank up to $1.75X$ using the branch-avoiding model [11].
• We show how to apply LRB to the problem of segmented sorting. We reorder the segments based on their length and achieve better performance using existing sorting implementations. Specifically, our LRB based segmented sort is anywhere from $50 - 250\times$ faster than NVIDIA's CUB's segmented sort, even though we use CUB's sorting functionality without changing it.

II. Related Work

Scheduling and load-balancing are heavily researched topics in the last half century. Some scheduling algorithms focus on optimizing execution time in an off-line setting, whereas other algorithms are required to make decision online (sometimes referred to as in real-time). The latter case is typically true for data dependent applications where the work per task is not known apriori and only discovered at execution time. This is the case for graph algorithms, SpMV, and other irregular problems such as segmented sorting.

In many cases, these data dependent tasks have relatively low computational intensity. Therefore, whatever on-line scheduling or load-balancing method that is used should not add a lot of overhead to the execution. It is not surprising that prefix summation arrays are often used for partitioning work across the threads, as prefix summation arrays can be computed efficiently.

Parallel Prefix Summation

Prefix summation arrays are used in a wide range of applications, including as the offset pointer arrays for the compressed sparse row (CSR) data structure used for sparse graph and matrix problems, cumulative distribution function, lexicographical sorting, and as a building block for radix sort. Some additional use cases of prefix sum arrays can be found in algorithm optimizations: breadth-first search [2], triangle counting [14], and the Gunrock [22] and Hornet [6] graph frameworks.

Computing the prefix sum array is a high parallel operation and work efficient algorithms have been designed for CPU, GPUs, and other accelerators. This operation is sometimes referred to as a *scan*. We use these terms interchangeably in this paper. One of the first papers to discuss parallelization of the prefix summation is [5]. One of the first work efficient prefix summation operations for NVIDIA GPUs can be found in [15].

The time complexity of a sequential scan is $O(N)$ given an input array with N elements. The parallel version of this algorithm has a work-depth (equal to the time complexity) of $O(log(N))$. The actual work depth is $2 \cdot log(N)$ for both the upwards and downwards sweep. Between each computational phase is a synchronization, which is required to ensure that the correct values are propagated to the next phase. For a system with T threads, the work complexity is $O(N+T \cdot log(T))$ and the time complexity is $O(\frac{N}{T}+log(T))$. Specifically, each thread is given N/T elements for which they are responsible for doing a local scan. Synchronization on the GPU is significantly less expensive than it is on the CPU allowing for improved scalability. The number of writes required by these algorithms is roughly $2 \cdot N$.

III. Logarithm Radix Binning

In this section, we introduce Logarithm Radix Binning (LRB for short), a method for effectively load balancing small tasks for many threaded systems. LRB can also be extended for vector ISAs and is appropriate for SIMD and SIMT programming models. In this paper we modify the formulation used in [13] which is specific for triangle counting and make LRB a more generic load-balancing mechanism that can be applied to a wider range of problems.

LRB works by placing tasks with a similar amount of work within the same "work-bins"[1]. These work-bins allow placing tasks with similar computational properties within the same spatial and temporal vicinity such that the tasks can be executed by concurrent threads. While the workload estimation function is application-dependent, the binning process we outline is application independent.

A. Logarithm Radix Binning

Formally, given a set of N tasks $T = \{t_1, t_2, .., t_N\}$, we denote the work for each of these tasks, $t \in T$, as $work(t)$ or w_t (for short). Work is measured by some unit of execution, such as the number of instructions, and as such w_t is an integer. Representing w_t requires $\log(|w_t|)$ bits, this is typically 32 or 64 bits. We denote the width of this representation as B. While B scales logarithmically with the size of the largest task, in practice we can set $B \in \{32, 64\}$. It is not expected that a larger representation for B will be needed as this would represent an extremely large task (e.g. 2^{64} operations).

As a first step, we create an array of counters of size $B + 1$ entries and initialize all counters to be zero. These counters will be used to bin tasks with a similar amount of work into the same "work-bins". Specifically, for counter $b \in \{0, 1, .., B\}$, whenever a task with work $2^b <= w_t < 2^{b+1}$ is encountered that counter is incremented by one. For each task we compute the following:

$$bin_t = (\lceil log_2(w_t) \rceil). \tag{1}$$

The tasks are placed in bins based on the logarithmic work estimate, hence the name **"Logarithm Radix Binning"**. Note the ceiling operation in Eq. 1 ensures that the indices are integer values. In practice, finding these values can be done using a less expensive approach: $bin_t = (B - cntLZ(w_t v))$ where $cntLZ$ is a function for counting the number of leading zeros. By subtracting the leading zeros from the word width, we can identify the ceiling of the logarithm value.

Fig. 1 depicts the ordering of tasks for several different load-balancing approaches: (a) our new LRB reordering, (b)

[1]Bins are selected by the logarithmic value of the estimated amount of work for that task.

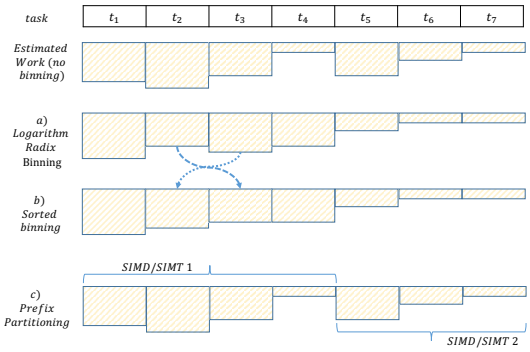


Fig. 1. Example of binning methodologies for a given input. The top array depicts the input with an expected amount of work per task: a) tasks reorganized using LRB, b) tasks sorted from largest to smallest, and c) partitioning using a prefix sum array for SIMD and SIMT execution where these tasks will be sent to a single vector unit. We highlight the difference between LRB a) and the sorted task b). Also note, the purpose of c) is to highlight that a single execution unit might receive tasks of different lengths leading to poor performance.

Algorithm 1: LRB Pseudo Code

for $i = 0 : 1 : B$ **do**
  $Bins[B] \leftarrow 0$
// Loop 2
for $i = 0 : 1 : N - 1$ **do**
  $b_i \leftarrow (\lceil log_2(w(T[i])) \rceil)$
  $atomicAdd(Bins[b_i], 1)$
// Loop 3
$prefixB[0] \leftarrow 0$
for $i = 1 : 1 : B$ **do**
  $prefixB[i] \leftarrow prefixB[i-1] + Bins[i-1]$
// Loop 4
for $i = 0 : 1 : N - 1$ **do**
  $b_i \leftarrow (\lceil log_2(w(T[i])) \rceil)$
  $pos_i = atomicAdd(prefixB[b_i], 1)$
  $T_{reordered}[pos_i] = T[i]$

sorting, and (c) using a prefix sum array for partitioning the work (without sorting).

LRB reordering can be considered an approximation to sorting, although that is not the end goal. While a full sorting followed by PPS could achieve similar goals to LRB, we believe the overhead with this approach would be unnecessarily high, and not competitive with LRB. In later experiments we show that LRB overhead is competitive with that of parallel prefix sum.

Fig. 1 (c) shows a scheduling where the input was partitioned using a prefix sum operation. While each thread (or multiprocessor) might get an equal amount of work, the vector lanes within the SIMD/SIMT units are imbalanced. This can lead to system under-utilization.

LRB does ensure that threads within the same SIMD or SIMT units will get similar amount of work. This is due to LRB reordering that places tasks in the same bin such that no task differs by more than twice the amount of work. Specifically, if two tasks are in bin b, that means that both of the tasks require at most 2^b and no fewer than 2^{b-1} operations, otherwise they would have been in different bins (Eq. 1). This is an important feature of LRB.

B. LRB Reordering

Alg. 1 presents sequential pseudo-code for doing the LRB reordering of the tasks. LRB is made up of four loops. The first loop initializes the bin sizes to be of size zero and is parallelizable (even though B is a small constant). The second for loop iterates over the tasks, array T, and estimates the amount of work per task. Based on the size of the task, the bin that will contain that task is incremented. At the end of this loop, the number of tasks within each bin is known. Using the third loop a small prefix summation operation is computed for the reordered output from the largest task bin to the smallest task bin. This loop is small since $B \in 32, 64$. The elements of bin 0 will be followed by the elements of bin 1, and so forth. Through the prefix sum operation, we have the starting points of the bins in the reorder output. Finally, in the fourth and last for loop, the elements are placed in the reordered output based on their bin sizes.

While we describe a sequential version of LRB for brevity, LRB is easily parallelizable and we do so in practice. Implementation differs between CPU and GPU, but in general we run local versions of LRB per thread before combining results globally. Further complexity analysis of the parallel LRB and comparison with parallel prefix is given in Sec. III-D.

C. LRB Task Partitioning

While there are different ways to assign tasks to threads given the LRB-reordered task list, and depending on the platform, we present one simple assignment scheme here. Specifically, we use the reordered tasks' index set modulo with respect to P. For example if task 1234 is the largest task and is the first task to be placed in the first bin, its index in the reordering is 0.

The scheduling will be as follows: thread 0 will receive the edges $(0, P, 2 \cdot P, ...)$, thread 1 will receive the edges $(1, P+1, 2 \cdot P+1, ...)$ and so forth. If each thread or core can executed upto w concurrent sub-threads[2], then the workload partitioning will be at a wider granularity of w elements at a time, for example thread 0 will receive $((0, ..., w-1), P \cdot w + (0, ..., w-1), ...)$.

By using the above task partitioning scheme we ensure that tasks within a given bin are distributed evenly amongst the threads. Further, if w tasks are dispatched per thread, these tasks are also well balanced and get executed efficiently using SIMD/SIMT instructions.

D. Complexity Analysis

Given N elements, the array is scanned once to obtain the number of tasks in each bin—$O(N)$. This is followed up by a $O(B)$ prefix sum operation. Lastly, the LRB reorganization takes another $O(N)$ operations. Since B is a small constant fixed before computation, the sequential complexity is $O(N)$.

Parallel LRB on the CPU achieves a time complexity of $O(\frac{N}{P} + B \cdot \log P)$. The $O(B \cdot \log P)$ comes from running

[2]In the branch-avoiding model [13], [23] these are referred to as control flows and each data lane in a vector unit executes a different thread. In SIMT (GPU), these are the threads within a thread-block.

parallel prefix sum over each column of a P bin arrays of size B. For results in this paper, we implemented each prefix sum sequentially, which likely left some performance on the table.

Once new positions are determined, the reordering phase is embarrassingly parallel. From a storage complexity, two additional arrays are required in parallel LRB: $O(P \times B)$ for prefix arrays and $O(N)$ for the final reordered data.

IV. Experimental Setup

a) Experiment System: The experiments presented in this paper are executed on two distinct systems: an Intel Xeon Phi 7250 processor and an NVIDIA GV100 GPU.

The Intel Xeon Phi 7230, has 96GB of DRAM memory (102.4 GB/s peak bandwidth). This processor is part of the Xeon Knights Landing (KNL) series of processors, and features an additional 16GB of MCDRAM high bandwidth memory (400 GB/s peak bandwidth). We target the faster of these two memories via NUMA control and ensure that the data and all allocated memory fit into MCDRAM. As such the lower bandwidth DRAM memory isn't utilized. The KNL processor has 64 cores with 256 threads (4-way SMP). The 4 threads of each core share 2 vector processing units supporting AVX-512 instructions. To utilize these instructions for irregular applications, we use the vectorized branch avoiding model introduced in [13], [23] (which extend the branch avoiding model introduced in [12]). In the vectorized branch avoiding model, each data lane in the AVX-512 unit executes a different data control flow. As such each vector unit can execute for up to 16 data elements simultaneously, assuming 32 bit data. Given these system parameters, we are able to execute up to 256 concurrent threads of execution, each issuing 16-word wide instructions. Thus, load-balancing is important.

The GV100 is a Volta micro-architecture GPU with 80 SMs and 64 SPs per SM, for a total of 5120 SPs (lightweight hardware threads). In practice, roughly 40K software threads are required for fully utilizing the GPU. The GV100 has a total of 32GB of HBM2 memory. The GV100 also has 640 tensor cores, but these are not used in our experiments. Our GV100 is the PCI-E variant that has a peak power consumption of 250 watts.

b) Problem Set: In the following section we will look at three different problems that allow us to test the performance of LRB. First of all, we compare the performance of LRB reordering with that of doing a prefix sum computation. Even though prefix sum alone does not ensure good load-balancing for SIMD/SIMT, it is a good performance point-of-reference. In the second experiment, we show that LRB can be used for improving the performance of PageRank for SIMD and SIMT models by using the vectorized branch-avoiding model on the KNL processor. Lastly, we show how to improve the performance of a segmented sort algorithm by re-organizing the segments based on their lengths using LRB. Segmented sort can be used for sorting the rows of a matrix or graph. Our new and improved segmented-sort does not require a new sorting functionality to be implemented. Rather, it uses existing sorting functionality but in a different sequence.

c) Inputs: For testing the aforementioned problems we use real world data taken from a wide range of graphs and matrices: SNAP[17], HPEC Graph Challenge [20], DIMACS [1], and Florida Matrix [7]. Table I captures a subset of the inputs used more extensively in this paper.

V. Empirical Performance Analysis

A. LRB vs. Prefix Summation

In this subsection we compare the cost of executing the LRB reordering with that of computing a PPS on the same array. Specifically, for inputs we use matrices (graphs in Table I) where the elements in each array is the number of non-zero elements in each row. Thus, the result of the PPS operation is equal to the creation of the offset array of a CSR matrix. The result of the LRB is rearranging each element by bins. For prefix sum on the GPU, we used the `ExclusiveSum` function from the CUB library [18]. For the CPU, we used a parallel prefix sum implementation in GAP [4].

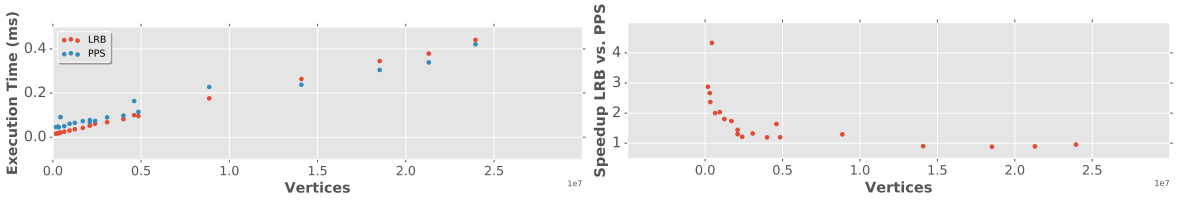
a) GPU Results: Fig. 2 (a) depicts the time it takes to create the LRB reordering and to compute the PPS array as a function of the number of input size. For both computations, it is clear the execution time is linear with the input size. Fig. 2 (b) depicts the speedup of the LRB reordering over the PPS operation. Both these subplots are for the GPU. Creating the LRB reordering can be up to $4\times$ faster than PPS operation for smaller inputs. As the inputs become large the times become roughly the same and the prefix sum operation can be upto 10% faster.

b) CPU Results: Fig. 3 (a) depicts the speedup of LRB over PPS on the Intel KNL processor. Fig. 3 (b) and (c) depict the scaling for LRB and PPS, respectively. For smaller thread counts, LRB is slower than PPS by as much as $5\times$. However, for large threads it is almost $2\times$ faster. The improvement with the increased thread counts is in large part due to the improved scalability of LRB. LRB scales to roughly $20\times$ where as PPS scalability is very limited and peaks at $4\times$. This could be due to the larger number of synchronizations used in PPS.

B. Vectorized PageRank with LRB

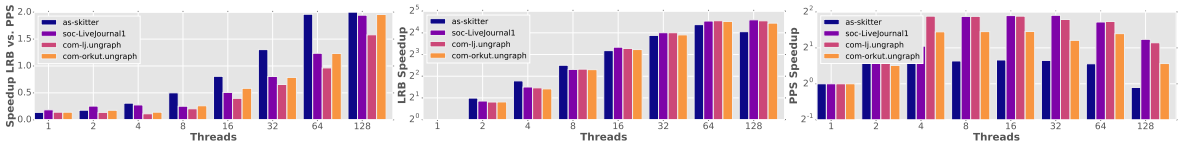
We look to improve the performance of PageRank by using the vectorized branch avoiding model. Specifically, we target Intel's AVX-512 ISA and show that we can parallelize the PageRank computation across the vector unit's multiple data lanes (as was done in [13], [23]). We present our vectorized PageRank in Alg. 2. At a high level, each thread computes the next PageRank values for 16 vertices at a time, by processing each vertice's in-neighbor list in lockstep via AVX-512 instructions. To improve load-balancing and SIMD utilization, we use LRB to reorder vertices by degree prior to running any PageRank iterations.

For experiments we use the optimized PageRank implementation from GAP [4] as our baseline. Both the baseline and our implementation were compiled using Intel's *icpc*



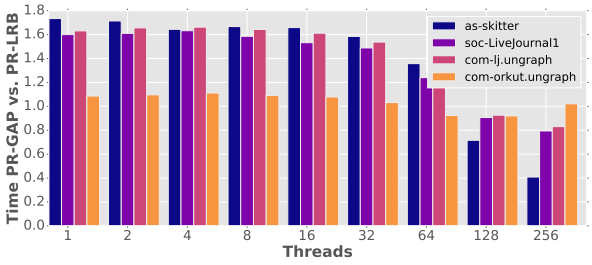
(a) Execution time (b) Speedup of LRB creation over PPS

Fig. 2. Performance analysis of LRB vs PPS on NVIDIA V100 GPU.

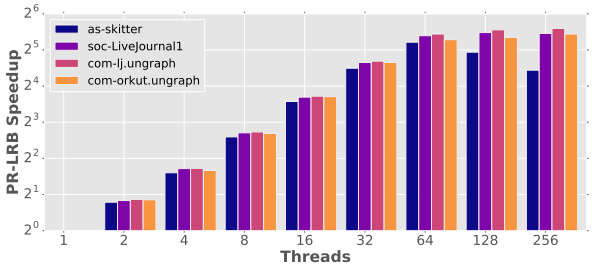


(a) Speedup of LRB creation over PPS (b) Strong scaling for LRB (c) Strong scaling for PPS

Fig. 3. Performance analysis of LRB vs PPS on Intel KNL processor.



(a) Ratio of GAP's PR execution time vs. PR-LRB. X-axis is number of threads used on the KNL system, and numbers for four graphs are reported per thread configuration.



(b) PR-LRB speedup vs. number of threads. Single-threaded PR-LRB is baseline.

Fig. 4. PR-LRB performance and scaling

compiler. While the baseline also benefits to some extent from compiler-driven vectorization, our implementation is designed with vectorization in mind. Only the MCDRAM (high-bandwidth) memory is used in our experiments.

Fig. 4 (a) compares the execution time T_{PR} of the GAP baseline to our implementation T_{PR-LRB}'s via the ratio $\frac{T_{PR}}{T_{PR-LRB}}$. The PR-LRB numbers used in Fig. 4 includes both LRB reordering and subsequent PR execution time. With the exception of *com-orkut*, PR-LRB consistently outperforms the baseline by $1.5\times$ to $1.75\times$ up to 32 threads, before a decline from 64 threads onwards to being slightly slower than baseline PR in performance at 128 and 256 threads.

To better understand performance trends, we present

Algorithm 2: Vectorized Pagerank Iteration

for $v \in V$ **do in parallel**
  $PR_{curr}[v] \leftarrow 0$
  $C[v] \leftarrow Pr_{prev}[v]/deg[v]$
for $i = 0; i < |V|; i \mathrel{+}= 16$ **do in parallel**
  $\vec{u} \leftarrow [u_i, u_{i+1}, ..., u_{i+15}]$
  $L \leftarrow \max\{deg(u_i), deg(u_{i+1}), ..., deg(u_{i+15})\}$
  $t \leftarrow 0$
  for $j = 0; j < L; j \mathrel{+}= 1$ **do**
    // $n_j(u_i)$ denotes j-th neighbor of u_i
    $\vec{m} \leftarrow [n_j(u_i), n_j(u_{i+1}), ..., n_j(u_{i+15})]$
    $\vec{t} \mathrel{+}= Gather(\vec{m}, C)$
  $\vec{t} \leftarrow \vec{t} * damp + \frac{1-damp}{|V|}$
  $Scatter(\vec{t}, PR_{curr})$

strong-scaling results for PR-LRB in Fig. 4 b). Our implementation has linear scalability up to 64 threads (which also represents the number of physical cores in the system). Beyond 64 threads, hyper-threading is used and performance stagnation is a known side-effect [21]. PR-LRB is also more data-intensive than regular PR as each thread can be responsible for up to 16 memory requests. Our scalability is therefore likely to become memory-bound, and at an earlier thread count than in a non-vectorized implementation. We also did not utilize DDR memory, which could be a factor if our application is more than simply bandwidth-bound.

C. Segmented Sorting

Segmented sorting is where the input data is split into multiple partitions and each partition needs to be sorted. One very important use-case for segmented sorted is the sorting of the rows for a matrix, or the adjacency arrays of a graphs. One key benefit of segmented sort is that each segment can be sorted in parallel, and each sorting itself can also be parallelized. Numerous applications require sorting segments rather than doing a full sort across the data.

For example, a sparse matrix can be sorted in multiple manners. If the matrix is stored as a COO, the the sorting

TABLE I

INPUTS USED FOR SEGMENTED SORTING, EQUIVALENT OF CSR SORTING OF EACH ROW. LRB-CUB IS COMPARED AGAINST SEVERAL SEGMENTED SORT ALGORITHMS AS WELL AS TWO-PHASE SORT PAIRS ALGORITHM. WE SHOW BOTH EXECUTION TIMES AND SPEEDUP OF LRB IN COMPARISON TO CUB AND MODERNGPU (MGPU). BB-SORT WAS TYPICALLY FASTER DUE TO ITS OPTIMIZED MERGING AND SORTING KERNELS.

				Segmented Sorting Algorithms					Speedup		
name	nv	ne	CUB-SortPair	MGPU	CUB	LRB-CUB	BB-Sort		SortPairs	CUB	MGPU
soc-LiveJournal1	4847571	68993773	31.47	13.128	1923.32	13.066	11.003		2.40	179.38	1.004
As-Skitter	1696415	22190596	10.054	3.23	786.09	3.819	5.1		2.63	268.29	0.844
com-lj.ungraph	3997962	69362378	29.447	10.864	1803.58	13.09	12.864		2.24	167.86	0.829
com-orkut.ungraph	3072441	234370166	98.12	36.184	1390.39	31.558	25.456		3.10	52.15	1.146
Soc-twitter-2010	21297772	530051618	218.28	64.05	9446.72	74.384			2.93	199.28	0.861

can be done trivially by sorting the (row, col) pairs in array of length nnz or $|E|$ using two iterations, the first on the columns and the second on the rows. Thus, the time complexity is $O(nnz)$ or $O(nnz \cdot log(nnz))$ for a radix-sort and merge-sort, respectively. In contrast, the time complexity of the segmented rows, especially for merge-sort, can be smaller as the segments are smaller and the multiplier within the $log(nnz)$ will be smaller for the various segments. The different computational requirements for each segmented sort that makes this problem workload imbalanced, which LRB is able to resolve.

In this section, we will compare the performance of two different segmented-sorts in two widely used GPU frameworks: ModernGPU [3] and CUB [18]. ModernGPU's segmented sort is based on merge-sort, whereas CUB's is based on radix-sort. ModernGPU's segmented sort is orders of magnitude faster than CUB's (Table I), while their standard array sorting algorithms have comparable performance (and in many cases CUB is faster). Thus the huge discrepancy in their segmented sort performance is unexpected. We also compare our new LRB-CUB segmented sort to the recent segment sort, ESSC, in [16]. Unlike ModernGPU and CUB, ESSC is a standalone sorting algorithm and not part of a framework, though it is the best performing. ESSC uses highly optimized merging and sorting kernels as part of its implementation. These kernels are several thousands lines of code; comparatively, our approach requires fewer than a hundred lines of code.

Table I depicts the execution times for the aforementioned sorting algorithms. At first glance, CUB's segmented sort is orders of magnitude slower than the other solutions. We show that the problem is with its load-balancing. CUB has several different sorting functions in its frameworks: 1) sort an entire array using the whole GPU for this operation, 2) segment-sorting which sorts multiple segments using the entire device, and 3) a sorting algorithm at the thread block granularity. The last of these, 3), targets small arrays that can typically fit in shared-memory. Our LRB based segmented-sort utilizes the existing sorting functionality. Specifically we target 3) by reordering the segments into convenient chunks that fit in the L1 caches and can use CUB's thread-block sorted with optimized run-time parameters.

a) LRB Based Segmented Sort: Alg. 3 depicts our LRB based segmented sort. Our LRB sorting algorithm is simple and that does not require any new sorting kernels,

Algorithm 3: LRB-CUB - Segmented Sort Using LRB. Input is a CSR graph.

```
(V_RE, Bins) ← LRB_ReOrdering(offset, Edges);
startVertex ← Bins[0]; endVertex ← Bins[20];
// Sort all vertices that have an a degree between 2^12 to 2^32
SegmentedSort(offSet, Edges, V_RE, Bins, startVertex,
  endVertex, sortedEdges)
for i = 21 : 1 : 32 do in parallel
    startVertex ← Bins[b − 1]; endVertex ← Bins[b];
    // All segments are of length 2^(b−1) to 2^b
    for u ∈ V_RE[startVertex], ..., V_RE[endVertex] do in
      parallel
        // GPU Kernel that use CUB's thread-block sort
        sortSingleSegment(u, offset, Edges, sortedEdges)
```

in contrast to the load-balanced ESSC [16]. Instead, our LRB-CUB based segmented sort uses the already existing sorting functionality in CUB without modification. Note, for the bins with smaller rows, bins 21 to 32 (which contain rows with upto 2^{12} entries), each bin is processed by a different GPU kernel that use's CUB's thread-block sort with different parameters. Furthermore, these GPU kernels can be launched concurrently.

b) Performance Analysis: Using LRB and CUB's various sorting functioning we are able to significantly improve the performance of CUB's original sorting algorithm by as much as $268\times$. LRB-CUB's segmented sort performance is now comparable with that of ModernGPU's. LRB-CUB is also roughly $2.5 \times -3\times$ faster than running two iterations of CUB's Radix sort. In a deeper look, roughly $15\% - 35\%$ of LRB-CUB's execution time is spent on sorting the larger rows (which roughly 10% of the rows). The LRB re-ordering itself accounts for less than 1% of the total execution time. Using the fast thread-block sorting functionality in CUB is dependent on the LRB reordering, specifically as using these functions requires compile time configuration.

While ESSC [16] is faster than LRB, it also uses highly optimized merging and sorting kernels that we did not use. These could also help LRB-CUB. Furthermore, these kernels are a few hundred lines of code in contrast to our approach which requires less than a hundred lines of code, most of which are on the CPU and responsible for managing the GPU.

Lastly, we also checked the performance of ModernGPU's segmented sort with the reorganized input and saw that it too had a $2\times$ performance improvement. While ModernGPU

sorted the segments correctly, the segments were not ordered correctly and required an additional reordering post-sort. Our key takeaway from this last experiment with ModernGPU is that LRB can also help with segmented sort that is merge-sort based.

VI. Conclusions

In this paper we show how to extend LRB load-balancing to a wider range of irregular applications. We also show that computing the LRB reordering scales well and sometimes better than computing the prefix sum array in parallel. We analyze the time complexity of parallel LRB and show that it is comparable to that of a parallel prefix sum. Furthermore, while a prefix sum alone does not ensure good load-balancing for SIMD/SIMT models, LRB can ensure that the work is distributed in a manner such that threads get work units that are never more than twice as large as other threads' at a time.

Empirically, we showed that LRB works efficiently on two massively multi-threaded systems, NVIDIA's V100 GPU and Intel's KNL processor. LRB improves performance on the KNL for PageRank by allowing use of the vector units in an efficient and well load-balanced manner. On the GPU, LRB improves the performance of segmented sort by up-to $268\times$ using existing sorting functionality.

In conjunction with previous work, we have shown that several different irregular applications fit well with our LRB load-balancing mechanism, and that LRB is simple enough to generalize. LRB converts execution of irregular applications into more regular execution (with better system utilization). In future work, we will explore how LRB can be applied to other irregular problems, as well as if LRB can be applied in distributed environment.

Acknowledgments

Funding was provided in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop," *Contemporary Mathematics*, vol. 588, 2013.

[2] B. W. Barrett, J. W. Berry, R. C. Murphy, and K. B. Wheeler, "Implementing a portable multi-threaded graph library: The MTGL on Qthreads," in *IEEE Int'l Symp. on Parallel & Distributed Processing (IPDPS)*, 2009, pp. 1–8.

[3] S. Baxter, "Modern GPU," 2013.

[4] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv preprint arXiv:1508.03619*, 2015.

[5] G. E. Blelloch, "Prefix sums and their applications," 1990.

[6] F. Busato, O. Green, N. Bombieri, and D. Bader, "Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[7] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," http://www.cise.ufl.edu/research/sparse/matrices.

[8] P. Flick, P. Sanders, and J. Speck, "Malleable sorting," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 418–426.

[9] J. Fox, O. Green, K. Gabert, X. An, and D. Bader, "Fast and Adaptive List Intersections on the GPU," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[10] O. Green and D. Bader, "cuSTINGER: Supporting Dynamic Graph Algorithms for GPUS," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2016.

[11] O. Green, M. Dukhan, and R. Vuduc, "Branch-Avoiding Graph Algorithms," *arXiv preprint arXiv:1411.1460*, 2014.

[12] O. Green, M. Dukhan, and R. Vuduc, "Branch-Avoiding Graph Algorithms," in *27th ACM on Symposium on Parallelism in Algorithms and Architectures*, 2015, pp. 212–223.

[13] O. Green, J. Fox, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. Bader, "Logarithmic Radix Binning and Vectorized Triangle Counting," in *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

[14] O. Green, L. Munguia, and D. Bader, "Load Balanced Clustering Coefficients," in *ACM Workshop on Parallel Programming for Analytics Applications (PPAA)*, Feb. 2014.

[15] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[16] K. Hou, W. Liu, H. Wang, and W.-c. Feng, "Fast segmented sort on gpus," in *Proceedings of the International Conference on Supercomputing*. ACM, 2017, p. 12.

[17] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data.

[18] D. Merrill and NVIDIA-Labs, "Cuda Unbound (cub) Library," *NVIDIA-Labs*, 2015.

[19] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *18th ACM SIGPLAN symposium on Principles and practice of Parallel Programming*, 2013, pp. 135–146.

[20] S. Siddharth, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static graph challenge: Subgraph isomorphism," in *IEEE Proc. High Performance Embedded Computing Workshop (HPEC)*, Waltham, MA, 2017.

[21] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

[22] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 265–266.

[23] J. Watkins and O. Green, "A Fast and Simple Approach to Merge and Merge Sorting using Wide Vector Instructions," in *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, Dallas, TX, 2018.