



# GPU Multisplit

Saman Ashkiani

University of California, Davis  
sashkiani@ucdavis.edu

Andrew Davidson\*

University of California, Davis  
aaldavidson@ucdavis.edu

Ulrich Meyer

Goethe-Universität Frankfurt am Main  
umeyer@cs.uni-frankfurt.de

John D. Owens

University of California, Davis  
jowens@ece.ucdavis.edu

## Abstract

Multisplit is a broadly useful parallel primitive that permutes its input data into contiguous *buckets* or *bins*, where the function that categorizes an element into a bucket is provided by the programmer. Due to the lack of an efficient multisplit on GPUs, programmers often choose to implement multisplit with a sort. However, sort does more work than necessary to implement multisplit, and is thus inefficient. In this work, we provide a parallel model and multiple implementations for the multisplit problem. Our principal focus is multisplit for a small number of buckets. In our implementations, we exploit the computational hierarchy of the GPU to perform most of the work locally, with minimal usage of global operations. We also use warp-synchronous programming models to avoid branch divergence and reduce memory usage, as well as hierarchical reordering of input elements to achieve better coalescing of global memory accesses. On an NVIDIA K40c GPU, for key-only (key-value) multisplit, we demonstrate a 3.0–6.7x (4.4–8.0x) speedup over radix sort, and achieve a peak throughput of 10.0 G keys/s.

## 1. Introduction

This paper studies the multisplit primitive for GPUs. Multisplit divides a set of items (keys or key-value pairs) into contiguous buckets, where each bucket contains items whose keys satisfy a programmer-specified criterion (such as falling into a particular range). Multisplit is broadly useful in a wide range of applications, some of which we will cite later in this introduction. But we begin our story by focusing on one particular example, the delta-stepping formulation of single-source shortest path (SSSP).

The traditional (and work-efficient) serial approach to SSSP is Dijkstra’s algorithm [12], which considers one vertex per iteration—the vertex with the lowest weight. The traditional parallel approach (Bellman-Ford-Moore [5]) considers all vertices on each iteration, but as a result incurs more work than the serial approach. On the GPU, the recent SSSP work of Davidson et al. [7] instead built upon

the delta-stepping work of Meyer and Sanders [20], which on each iteration classifies candidate vertices into *buckets* or *bins* by their weights and then processes the bucket that contains the vertices with the lowest weights. Items within a bucket are unordered and can be processed in any order.

Delta-stepping is a good fit for GPUs. It avoids the inherent serialization of Dijkstra’s approach and the extra work of the fully parallel Bellman-Ford-Moore approach. At a high level, delta-stepping divides up a large amount of work into multiple buckets and then processes all items within one bucket in parallel at the same time. How many buckets? Meyer and Sanders describe how to choose a bucket size that is “large enough to allow for sufficient parallelism and small enough to keep the algorithm work-efficient” [20]. Davidson et al. found that 10 buckets was an appropriate bucket count across their range of datasets. More broadly, for modern parallel architectures, this design pattern is a powerful one: expose just enough parallelism to fill the machine with work, then choose the most efficient algorithm to process that work. (For instance, Hou et al. use this strategy in efficient GPU-based tree traversal [15].)

Once we’ve decided the bucket count, how do we efficiently classify vertices into buckets? Davidson et al. called the necessary primitive *multisplit*. Beyond SSSP, multisplit has significant utility across a range of GPU applications. Bucketing is a key primitive in reorganizing rays into 8 direction-based buckets for better coherence in a GPU-based ray tracer [30]; as the first step in building a GPU hash table [3]; in hash-join for relational databases to group low-bit keys [11]; in string sort for singleton compaction and elimination [10]; in suffix array construction to organize the lexicographical rank of characters [9]; in a graphics voxelization pipeline for splitting tiles based on their descriptor (dominant axis) [26]; in the shallow stages of  $k$ -d tree construction [29]; in Ashari et al.’s sparse-matrix dense-vector multiplication work, which bins rows by length [4]; and in probabilistic top- $k$  selection, whose core multisplit operation is three bins around two pivots [22]. And while multisplit is a crucial part of each of these and many other GPU applications, it has received little attention to date in the literature; the work we present here remedies this with a comprehensive look at efficiently implementing multisplit as a general-purpose parallel primitive.

The approach of Davidson et al. to implementing multisplit reveals the need for this focus. If the number of buckets is 2, then a scan-based “split” primitive [13] is highly efficient on GPUs. Davidson et al. built both a 2-bucket (“Near-Far”) and 10-bucket implementation. Because they lacked an efficient multisplit, they were forced to recommend their theoretically-less-efficient 2-bucket implementation:

\* Currently an employee at Google.

The missing primitive on GPUs is a high-performance *multisplit* that separates primitives based on key value (bucket id); in our implementation, we instead use a sort; in the absence of a more efficient multisplit, we recommend utilizing our Near-Far work-saving strategy for most graphs. [7, Section 7]

Like Davidson et al., we could implement multisplit on GPUs with a sort. Recent GPU sorting implementations [17] deliver high throughput, but are overkill for the multisplit problem: unlike sort, multisplit has no need to order items within a bucket. In short, sort does more work than necessary. For Davidson et al., reorganizing items into buckets after each iteration with a sort is too expensive: “the overhead of this reorganization is significant: on average, with our bucketing implementation, the reorganizational overhead takes 82% of the runtime.” [7, Section 7]

An efficient multisplit would have enabled a significant performance improvement for Davidson et al.<sup>1</sup>, as well as the other applications cited above. In this paper we design, implement, and analyze numerous approaches to multisplit, and make the following contributions:

- On modern GPUs, “global” operations (that require global communication across the whole GPU) are more expensive than “local” operations that can exploit faster, local GPU communication mechanisms. Straightforward implementations of multisplit primarily use global operations. Instead, we propose a parallel model under which the multisplit problem can be factored into a sequence of local, global, and local operations better suited for the GPU’s memory and computational hierarchies.
- We show that reducing the cost of global operations, even by significantly increasing the cost of local operations, is critical for achieving the best performance.
- We implement local operations efficiently by using warp-synchronous schemes to avoid branch divergence, reduce shared memory usage, leverage warp-wide instructions, and minimize intra-warp communication.
- We locally reorder input elements before global operations, trading more work (the reordering) for better memory performance (greater coalescing) for an overall improvement in performance.

## 2. Related Work and Background

Many multisplit implementations, including ours, depend heavily on knowledge of the total number of elements within each bucket, i.e., histogram computation. Previous GPU histogram implementations generally take one of two approaches: 1) using atomic operations (or designed software alternatives) to count items within each bucket (e.g., Shams and Kennedy [28]), and 2) per-thread sequential histogram computations that combine their results via global reduction (e.g., Nugteren et al. [24]). The former is suitable when the number of buckets is large; otherwise atomic contention is the bottleneck. The latter avoids such conflicts by using more memory (assigning

exclusive memory units per-bucket and per-thread), then performing device-wide reductions to compute the global histogram. Either method benefits from careful optimizations that make the best use of the GPU [6], including loop unrolling, thread coarsening, and subword parallelism, as well as others.

Only a handful of papers have explored multisplit as a standalone primitive. He et al. [14] implemented multisplit by reading multiple elements with each thread, sequentially computing their histogram and orders, and then storing all results into memory. Next, they performed a device-wide scan operation over these results and scattered each item into its final position. Their main bottlenecks were the limited size of shared memory, an expensive global scan operation, and random non-coalesced memory accesses<sup>2</sup>. Patidar [27] proposed two methods with particular concentration on a large number of buckets (more than 4k): one based on heavy usage of shared-memory atomic operations (to compute block level histogram and intra-bucket orders), and the other by iterative usage of basic binary split for each bucket (or groups of buckets). Patidar used a combination of these methods in a hierarchical way to get his best results<sup>3</sup>. Both of these multisplit papers focus only on key-only scenarios, while data movements become more challenging with key-value pairs.

### 2.1 The Graphics Processing Unit (GPU)

The GPU of today is a highly parallel, throughput-focused programmable processor. GPU programs (“kernels”) launch over a *grid* of numerous *blocks*; the GPU hardware maps blocks to available parallel cores. Each block typically consists of dozens to thousands of individual *threads*, which are arranged into 32-wide *warps*. Warps run under SIMD control on the GPU hardware. While blocks cannot directly communicate with each other within a kernel, threads within a block can, via a user-programmable 48 kB *shared-memory*, and threads within a warp additionally have access to numerous warp-wide instructions. The GPU’s global memory (DRAM), accessible to all blocks during a computation, achieves its maximum bandwidth only when neighboring threads access neighboring locations in the memory; such accesses are termed *coalesced*. In this work, when we use the term “*global*”, we mean an operation of device-wide scope. Our term “*local*” refers to an operation limited to smaller scope (e.g., within a thread, a warp, a block, etc.), which we will specify accordingly. The major difference between the two is the cost of communication: global operations must communicate through global DRAM, whereas local operations can communicate through lower-latency, higher-bandwidth mechanisms like shared memory or warp-wide intrinsics. Lindholm et al. [16] and Nickolls et al. [23] provide more details on GPU hardware and the GPU programming model, respectively.

We use NVIDIA’s CUDA as our programming language in this work [25]. CUDA provides several warp-wide voting and shuffling instructions for intra-warp communication of threads. All threads within a warp can know about a certain predicate as a bitmap variable returned by `__ballot(predicate)` [25, Ch. B13]. Any set bit in this bitmap denotes the predicate being non-zero for the corresponding thread. Each thread can also access a certain register from other threads of the same warp by using `__shfl(register_name, source_thread)` [25, Ch. B14]. Other shuffling functions such as `__shfl_up()` or `__shfl_down()` use relative addresses to specify the source thread. In CUDA, threads also have access to some effi-

<sup>1</sup> Davidson et al. reported that their Near-Far approach was 1.7x faster than their radix-sort-based SSSP bucketing method. We added our own 2-bucket multisplit algorithm (described in Section 5) as a new SSSP bucketing method, resulting in a speedup of the entire application of 1.3x over the Near-Far approach and a 2.1x speedup over the radix-sort-based implementation. These speedups are a geometric mean across 4 datasets: flickr with 10M edges [8], yahoo-social with 4M edges [2], rmat with 20M edges [1], and a sparse low-diameter synthetic graph with 15.5M edges having similar characteristics to the  $G_{BF}(n, r)$  class defined by Meyer [19]. We expect that by using our multisplit methods we could implement a new SSSP algorithm with a more optimal number of buckets (e.g., 10 as suggested by Davidson et al.) featuring even more significant speedups, but such an implementation is beyond the scope of this paper.

<sup>2</sup> On an NVIDIA 8800 GTX GPU, for 64 buckets, He et al. reported 134 Mkeys/sec. As a very rough comparison, our GPU has 3.3x the memory bandwidth, and our best 64-bucket implementation runs 22.4 times faster.

<sup>3</sup> On an NVIDIA GTX280 GPU, for 32 buckets, Patidar reported 762 Mkeys/sec. As a very rough comparison, our GPU has 2x the memory bandwidth, and our best 32-bucket implementation runs 5.9 times faster.

cient integer intrinsics, e.g., `__popc()` for counting the number of set bits in a register.

## 2.2 Parallel primitive background

In this paper we leverage numerous standard parallel primitives, which we briefly describe here. A *reduction* inputs a vector of elements and applies a binary associative operator (such as addition) to reduce them to a single element; for instance, sum-reduction simply adds up its input vector. The *scan* operator takes a vector of input elements and an associative binary operator, and returns an output vector of the same size as the input vector. In exclusive (resp., inclusive) scan, output location  $i$  contains the reduction of input elements 0 to  $i - 1$  (resp., 0 to  $i$ ). Scan operations with binary addition as their operator are also known as *prefix-sum* [13]. Any reference to a multi-operator (multi-reduction, multi-scan) refers to running multiple instances of that operator in parallel on separate inputs. *Compaction* is an operation that filters a subset of its input elements into a smaller output array while preserving the order.

## 3. Multisplit and Initial Approaches

In this section, we first formally define the multisplit as a primitive algorithm. Next, we describe some initial approaches for performing the multisplit algorithm, which form a baseline for the comparison to our own methods, which we then describe in Section 4.

### 3.1 The multisplit primitive

We informally characterize multisplit as follows:

- Input: An unordered set of keys or key-value pairs. “Values” that are larger than the size of a pointer use a pointer to the value in place of the actual value.
- Input: A function, specified by the programmer, that inputs a key and outputs the bucket corresponding to that key. For example, this function might classify a key into a particular numerical range, or divide keys into prime or composite buckets.
- Output: Keys or key-value pairs separated into  $m$  buckets. Items within each output bucket must be contiguous but are otherwise unordered. Some applications may prefer output order within a bucket that preserves input order; we call these multisplit implementations “stable”.
- $m$ , the number of buckets: a modest number, say more than 2 but less than or equal to 64. For two buckets, split is traditionally the best solution, and as the number of buckets grows, the multisplit problem converges to a full sort.

More formally, let  $\mathbf{u}$  and  $\mathbf{v}$  be vectors of  $n$  key and value elements, respectively. Altogether  $m$  buckets  $B_0, B_1, \dots, B_{m-1}$  partition the entire key domain such that each key element uniquely belongs to one and only one bucket. For any input key vector, we define *multisplit* as a permutation of that input vector into an output vector. The output vector is densely packed and has two properties: (1) All output elements within the same bucket are stored contiguously in the output vector, and (2) All output elements are stored contiguously in a vector in ascending order by their bucket IDs. Optionally, the beginning index of each bucket in the output vector can also be stored in an array of size  $m$ .

This multisplit definition allows for a variety of implementations. It places no restrictions on the order of elements within each bucket before and after the multisplit (intra-bucket orders); buckets with larger indices do not necessarily have larger elements. In fact, key elements may not even be comparable entities, e.g., keys can be strings of names with buckets assigned to male names, female names, etc. We do require that buckets are assigned to consecutive IDs and

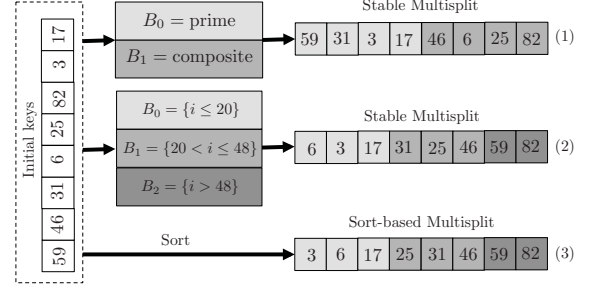


Figure 1: Multisplit examples. (1) Stable multisplit over two buckets. (2) Stable multisplit over three range-based buckets. (3) Sort can implement multisplit over ordered buckets; note that this multisplit implementation is not stable.

will produce buckets ordered in this way. Figure 1 illustrates some multisplit examples.

Throughout this paper, we focus on multisplit problems with non-trivial bucket identifications. For example, if all buckets have identity IDs (i.e.,  $B_i = \{i\}$  and all keys are in  $\{0, 1, \dots, m - 1\}$ ), then we can use any current stable sorting method and get an efficient stable multisplit. In Section 6 we refer to this case as *identity buckets*. Next, we consider some common approaches for dealing with non-trivial multisplit problems.

### 3.2 Recursive scan-based split

The first approach is based on binary split. Suppose we have two buckets. We identify buckets in a binary flag vector, and then compact keys (or key-value pairs) based on the flags. We also compact the complemented binary flags from right to left, and store the results. Compaction can be efficiently implemented by a scan operation, and in practice we can concurrently do both left-to-right and right-to-left compaction with a single scan operation. With more buckets, we can recursively perform binary splits; on each round we split key elements into two groups of buckets. We continue this process for at most  $\lceil \log m \rceil$  rounds and in the end we will have a stable multisplit. Recursive scan-based split requires multiple global operations (e.g., scan) over all elements, and may also have load-balancing issues if the distribution of keys is non-uniform.

### 3.3 Radix sort

Radix sort gradually sorts key elements based on selected groups of bits in keys. The process either starts from the least significant bits (LSB sort), or from the most significant bits (MSB sort). MSB sort is more common because, compared to LSB sort, it does less intermediate data movement when distribution of keys is not uniform. (If the distribution of keys is uniform, they should perform the same.) Today’s most efficient radix sort implementations process a group of bits on each round to gain more efficiency. As an extreme example, if in each round a single bit is processed, then radix sort is equivalent to multiple scan-based splits (in each round, splitting based on a binary bit).

In general, sorting key-value pairs does not always represent a stable multisplit. However, depending on our problem, if our buckets are defined such that buckets with larger IDs have larger elements (e.g., all elements in  $B_0$  are less than all elements in  $B_1$ , and so on), sorting our key vector implements a multisplit (Fig. 1). Radix sorts are highly efficient in modern GPUs, and hence sorting is certainly a viable option. However, radix sort performs more work than multisplit requires by not only splitting items from different buckets but also ordering elements within a bucket.



### 3.4 Reduced-bit sort

Because sorting is an efficient primitive on GPUs, we modify it to be specific to multisplit: here we introduce our *reduced-bit sort* method, which is based on sorting bucket IDs and permuting the original key-value pairs afterward. For multisplit, this method is superior to a full radix sort because we expect the number of significant bits across all bucket IDs is less than the number of significant bits across all keys.

**Key-only** In this scenario, we first make a *label* vector containing each key’s bucket ID. Then we sort (label, key) pairs based on label values. Since labels are all less than  $m$ , we can limit the number of bits in the radix sort to be  $\lceil \log m \rceil$ .

**Key-value** In this scenario, we similarly make a label vector from key elements. Then, we would like to permute (key, value) pairs by sorting labels. One approach is to sort (label, (key, value)) pairs all together and based on label. To do so, we first pack our original key-value pairs into a single 64-bit variable and then do the sort. In the end we unpack these elements to form the final results. Another way is to sort (label, index) pairs and then manually permute key-value pairs based on the permuted indices. We tried both approaches and the former seems to be more efficient. The latter requires non-coalesced global memory accesses and gets worse as  $m$  increases, while the former reorders for better coalescing internally and scales better with  $m$ .

The main problem with the reduced-bit sort method is its extra overhead (generating labels, packing original key-value pairs, unpacking the results), which makes the whole process less efficient. Today’s sort primitives do not currently provide APIs for user-specified computations (e.g., bucket identifications) to be integrated as functors directly into sort’s kernels; while this is an intriguing area of future work for the designers of sort primitives, we believe that our reduced-bit sort appears to be the best solution today for multisplit using current sort primitives.

### 3.5 Randomized insertion

The last algorithm we consider extends a traditional CPU coarse-grained multi-threaded PRAM algorithm proposed by Meyer [18]. The basic premise is to create relaxed large buffers for each bucket. We assign threads to items and then throw randomized darts<sup>4</sup> into buffers to place items. If collisions occur, darts are rethrown, and if a buffer becomes too full, the empty slots are compacted out and the buffer is emptied out into a final collection. This is repeated until all elements are processed.

In order to adapt this type of algorithm to a fine-grained bulk-synchronous GPU, we must refactor this algorithm to fit the GPU’s block-based programming model. First, we must estimate the buffer size for each bucket. We run a pre-processing global histogram method to calculate the size of each bucket. We then create a relaxed buffer size that is  $x$  times bigger than the histogram size for each bucket (e.g., for  $x = 2$ , we have a 50% chance of a collision near the end of our insertion phase).

Next, each block creates  $b$  buckets with an  $x$ -times larger relaxation buffer in shared memory. Each block then reads its input and attempts to insert each value into one of these shared memory buckets, utilizing a hash on a random number. If a collision occurs, we search for an empty adjacent slot. When a shared-memory bucket is sufficiently full, threads cooperatively write this bucket (including empty slots) to global memory. Finally, once all elements have been inserted and written to main memory, we utilize a compact [13] primitive to remove empty slots within our buckets.

<sup>4</sup> The randomized dart-throwing technique on PRAMs was originally introduced by Miller and Reif [21] in the context of generating random permutations.

**Performance Analysis** Though conceptually this method is quite simple, and allows for a fine-grained parallel implementation of multisplit, it is slower than even a traditional radix sort. One problem is memory consumption and bandwidth: we require  $x$  times as much memory as the dataset, and the compact must operate on  $x$  times more data. The larger penalty comes from warp divergence: any insertion by a thread that results in a collision stalls all other threads within that warp until that collision is resolved. Thus we have two competing performance penalties adjustable by one variable, the relaxation constant  $x$ . If we increase  $x$  to decrease the number of collisions, we increase the number of total writes and memory usage. If we decrease  $x$ , we increase the number of collisions and contention and warps frequently stall.

Initial benchmarking of this PRAM method quickly demonstrated the inefficiency introduced by these competing performance penalties. Contention-based methods on massively parallel warp-synchronous devices incur too much of a penalty for high-performance primitives. We found the best performance came from  $x = 2$ ; however, even then the performance from such a method was around 2 times slower than a radix sort. Despite the theoretical advantages of this approach, it is not likely to be the basis for the best GPU implementation of multisplit; for the remainder of this paper, we focus on deterministic approaches.

## 4. Algorithm Overview

In analyzing the performance of the deterministic methods from the previous section, we make two observations:

1. Global computations (such as a global scan) are expensive, and approaches to multisplit that require many rounds, each with a global computation, are likely to be uncompetitive. Any reduction in the cost of global computation is desirable.
2. After we derive the permutation, the cost of permuting the elements with a global scatter is also expensive, primarily because of the non-coalesced memory accesses associated with the scatter. Any increase in memory locality associated with the scatter is also desirable.

The key design insight in this paper is that we can reduce the cost of both global computation and global scatter at the cost of doing more local work, and that doing so is beneficial for overall performance. We begin by describing and analyzing a framework for the different approaches we study in this paper, then discuss the generic structure common to all our implementations.

**Our parallel model** Multisplit cannot be solved by using only local operations; i.e., we cannot divide a multisplit problem into two independent subparts and solve each part locally without any communication between the two parts. We thus assume any viable implementation must include at least a single global operation to gather necessary global information from all elements (or group of elements). We generalize the approaches we study in this paper into a series of  $N$  rounds, where each round has 3 stages: a set of local operations (which run in parallel on independent subparts of the global problem); a global operation (across all subparts); and another set of local operations. In short: {local, global, local}, repeated  $N$  times; in this paper we refer to these three stages as {prescan, scan, postscan}.

The deterministic approaches in Section 3 all fit this model. Scan-based split starts by making a flag vector (where the local level is per-thread), performing a global scan operation on all flags, and then storing the results into their final positions (thread-level local). The recursive scan-based split repeats the above approach for  $\lceil \log m \rceil$  rounds. Radix sort also requires several rounds. Each round starts by identifying a bit (or a group of bits) from its keys (local), running

a global scan operation, and then locally moving data such that all keys are now sorted based on the selected bit (or group of bits). Reduced-bit sort is derived from radix sort; the only differences are that in the first round, the label vector and the new packed values are generated locally (thread-level), and in the final round, the packed key-value pairs are locally unpacked (thread-level) to form the final results.

**Multisplit requires a global computation** Let’s explore the global and local components of stable multisplit, which together compute a unique permutation of key-value pairs into their final positions. Suppose we have  $m$  buckets  $B_0, B_1, \dots, B_{m-1}$ , each with  $h_0, h_1, \dots, h_{m-1}$  elements respectively ( $\sum_i h_i = n$ ). If  $u_i \in B_j$  is the  $i$ th key element in key vector  $\mathbf{u}$ , then its final permuted position  $p(i)$  should be

$$p(i) = \sum_{k=0}^{j-1} h_k + |\{u_r : u_r \in B_j, r < i\}|, \quad (1)$$

where  $|\cdot|$  denotes the number of elements within its set argument. The left term is the total number of key elements that belong to the preceding buckets, and the right term is the total number of preceding elements in  $u_i$ ’s bucket  $B_j$ . Computing both of these terms in this form and for all elements (for all  $i$ ) requires global operations (e.g., computing a histogram of buckets).

**Dividing multisplit into subproblems** Now, let us divide our input key vector  $\mathbf{u}$  into  $L$  subproblems:  $\mathbf{u} = [\mathbf{u}_0, \mathbf{u}_1, \dots, \mathbf{u}_{L-1}]$ . Suppose each subvector  $\mathbf{u}_\ell$  has  $h_{0,\ell}, h_{1,\ell}, \dots, h_{m-1,\ell}$  elements in buckets  $B_0, B_1, \dots, B_{m-1}$  respectively. For example, for arbitrary values of  $i, s$ , and  $j$  such that key item  $u_i \in \mathbf{u}_s$  and  $u_i$  is in bucket  $B_j$ , equation (1) can be rewritten as:

$$p(i) = \underbrace{\sum_{k=0}^{j-1} \left( \sum_{\ell=0}^{L-1} h_{k,\ell} \right)}_{\text{global offset}} + \underbrace{\sum_{\ell=0}^{s-1} h_{j,\ell}}_{\text{my bucket}} + \underbrace{|\{u_r \in \mathbf{u}_s : u_r \in B_j, r < i\}|}_{\text{local offset within my subproblem}}. \quad (2)$$

This formulation has two separate parts. The first and second terms require global computation (first: the element count of all preceding buckets across all subproblems, and second: the element count of the same bucket in all preceding subproblems). The third term can be computed locally within each subproblem. Note that equation (1) and (2)’s first terms are equivalent (total number of previous buckets), but the second term in (1) is broken into the second and third terms in (2).

The first and second terms can both be computed with a global histogram computed over  $L$  local histograms. A global histogram is generally implemented with global scan operations (here, exclusive prefix-sum). We can characterize this histogram as a scan over a 2-dimensional matrix  $\mathbf{H} = [h_{i,\ell}]_{m \times L}$ , where the “height” of the matrix is the bucket count  $m$  and the “width” of the histogram is the number of subproblems  $L$ . The second term can be computed by a scan operation of size  $L$  on each row (total of  $m$  scans for all buckets). The first term will be a single scan operation of size  $m$  over the reduction of all rows (first reduce each row horizontally to compute global histograms and then scan the results vertically). Equivalently, both terms can be computed by a single scan operation of size  $mL$  over row-vectorized  $\mathbf{H}$ . Either way, the cost of our global operation is roughly proportional to both  $m$  and  $L$ . We see no realistic way to reduce  $m$ . Thus we concentrate on reducing  $L$ .

**Local offset computation** Each element must compute its own local offset, which represents the number of elements in its subproblem that both precede it and share its bucket. We can compute the local offset in two ways: explicitly or hierarchically.

To explicitly compute local offsets of a subproblem of size  $\bar{n}$ , we make a new binary matrix  $\bar{\mathbf{H}}_{m \times \bar{n}}$ , where each row represents a bucket and each column represents a key element. Each entry of this new matrix is one if the corresponding key element belongs to that bucket, and zero otherwise. Then by performing an exclusive scan on each row, we can compute local offsets for all elements belonging to that row (bucket). So each subproblem requires the following computations:

1. Mark all elements in each bucket (making local  $\bar{\mathbf{H}}$ )
2.  $m$  local reductions over the rows of  $\bar{\mathbf{H}}$  to compute local histograms (a column in  $\mathbf{H}$ )
3.  $m$  local exclusive scans on rows of  $\bar{\mathbf{H}}$  (local offsets)

For clarity, we separate steps 2 and 3 above, but we can achieve both with a single local scan operation.

If we decide to take the hierarchical approach, each time that we divide our subproblems into  $L_2$  smaller sub-subproblems, we similarly break the local offset term into two new terms, both among all items within our own bucket: the total number of items in previous sub-subproblems, and our new local offset within our own sub-subproblem (equivalent to the second and third terms in (2)). So, assuming that we divide each subproblem into  $L_2$  smaller parts, we would simply apply the explicit steps above to each smaller part.

**Our multisplit algorithm** Now that we’ve outlined the different computations required for the multisplit, we can present a high-level view of the algorithmic skeleton we use in this paper. We require three steps:

1. *Local*. For each subproblem, for each bucket, count the number of items in the subproblem that fall into that bucket.
2. *Global*. Scan the bucket counts for each bucket across all subproblems, then scan the bucket totals. Each subproblem now knows both the total count for each bucket across the whole input vector (term 1 in equation 2) as well as the total count for each buckets in the previous subproblems (term 2 in equation 2).
3. *Local*. For each subproblem, for each item, compute the local offset for that item’s bucket (term 3 in equation 2). We can now write each item in parallel into its location in the output vector.

**Reordering elements for better locality** After computing equation (2) for each key element, we move key-value pairs to their final positions in global memory accordingly. However, in general, consecutive key elements in the original input do not belong to the same bucket, and thus their final destination might be far away from each other. Thus, when we write them back to memory, our memory writes are poorly coalesced, and our achieved memory bandwidth during this global scatter is similarly poor. This results in a huge performance bottleneck. How can we increase our coalescing and thus the memory bandwidth of our final global scatter?

Our solution is to *reorder* our elements within a subproblem before they are scattered back to memory. Within a subproblem, we attempt to place elements from the same bucket next to each other, while preserving order within a bucket (and thus the stable property of our multisplit implementation). We do this reordering at the same time we compute local offsets in equation (2). How do we group elements from the same bucket together? A local multisplit within the subproblem!

We have already computed histogram and local offsets for each element in each subproblem. We only need to perform another local exclusive scan on local histogram results to compute new positions for each element in its subproblem (computing equation (1) for each subproblem). We emphasize that performing this additional stable multisplit on each subproblem does *not* change its histogram and local offsets, and hence does not affect any of our computations

Granularity	size of $\mathbf{H}$	global operation on
thread-level	$m \times \text{No. of threads}$	$mn$
warp-level	$m \times \text{No. of warps}$	$mn/N_T$
block-level	$m \times \text{No. of blocks}$	$mn/(N_T N_W)$

Table 1: Size of global operations for each local granularity.  $N_T$  denotes the number of threads per warp, and  $N_W$  is the number of warps per block.

described previously from a global perspective; the final multisplit result is identical. But, it has a significant positive impact on the locality of our final data writes to global memory.

**Choosing the size of subproblems** The traditional approach to both histograms and multisplit (as in He et al. [14]) is to assign the  $n$  elements to  $n$  threads<sup>5</sup> and treat them as  $n$  separate subproblems. This approach is simple: with it, “local” work is minimal, even trivial, and the scan implementation at the core of the global operation will typically transparently leverage the GPU’s computation hierarchy. Nonetheless, a scan of size  $n$  is still more expensive than we would like.

GPUs offer many potential natural subproblem sizes that correspond to the levels of their computational hierarchies: problems the size of threads, warps, blocks, and the entire device (global). We have shown above that choosing larger subproblems is possible, and in the next section how we can implement their operations efficiently. We thus face a performance tradeoff. With a large  $L$ , local computations are easier because each subproblem has fewer elements, but global operations will cost more because  $\mathbf{H}$  is larger. On the other hand, a smaller  $L$  (fewer subproblems) leads to cheaper global operations, but also larger subproblems and hence more expensive local computation.

## 5. Implementation Details

So far we have seen that we can reduce the size and cost of our global operation (size of  $\mathbf{H}$ , Table 1) by doing more local work (increasing the size of  $\mathbf{H}$  instead). This is a complex tradeoff. In this section we describe three novel and efficient multisplit implementations that explore different points in this implementation space:

**Direct Multisplit** Rather than split the problem into subproblems across threads, as in traditional approaches [14], Direct MS splits the problem across warps, leveraging efficient warp-wide intrinsics to perform the local computation. The major advantage of warp-sized subproblems is the reduction of the cost of the global step by a factor of  $N_T = 32$ , the number of threads per warp.

**Warp-level Multisplit** Warp-level MS also uses warp-sized subproblems, but additionally reorders elements within a warp for better locality.

**Block-level Multisplit** Block-level MS modifies Warp-level MS to use thread-block-sized subproblems and reordering, offering a further reduction in the cost of the global step (but considerably more complex local computation).

Table 2 shows an overview comparison between different stages of each approach. We now discuss the most interesting aspects of our implementations of these three approaches, separately describing how we compute histograms and local offsets for larger subproblem sizes and how we reorder final results before writing them to global memory to increase coalescing.

<sup>5</sup> We might apply thread coarsening—multiple items per thread—and divide  $n$  by the number of items per thread.

### 5.1 Computing Histograms and Local Offsets

Direct Multisplit and Warp-level MS reduce the size and cost of the global scan by a factor of  $N_T = 32$  at the cost of more complex local offset computations. These computations now require cooperation between the threads inside a warp. Fortunately, NVIDIA’s relatively new warp-wide intrinsics [25] enable efficient computations within a warp. Direct MS follows the skeleton we summarized in the last section:

**Pre-scan (local)** Each warp reads a section of key elements, generates a local matrix  $\bar{\mathbf{H}}$ , and computes its histogram (reducing each row). Each warp thus computes a single column of  $\mathbf{H}$  and stores its results into global memory.

**Scan (global)** We perform an exclusive scan operation over the row-vectorized  $\mathbf{H}$  and store the result back into global memory (e.g., matrix  $\mathbf{G} = [g_{i,\ell}]_{m \times L}$ ).

**Post-scan (local)** Each warp reads a section of key-value pairs, generates its local matrix  $\bar{\mathbf{H}}$  again<sup>6</sup>, and computes local offsets (with a local exclusive scan on each row). We then compute final positions by using the base addresses from  $\mathbf{G}$ , then write key-value pairs directly to their storage locations in the output vector. For example, if key  $u \in B_i$  is read in warp  $\ell$  and its local offset is equal to  $k$ , its final position will be  $g_{i,\ell} + k$ .

A simplified pseudo-code of the Direct MS is shown in Algorithm 1. Here, we can identify each key’s bucket by using a `whatBucket()` function. We compute warp histogram and local offsets with `warp_histogram()` and `warp_offsets()` procedures, which we describe in detail later in this section (Alg. 2 and 3).

#### Algorithm 1 The Direct Multisplit algorithm

---

**Input:** `key[]`, `value[]`, `whatBucket()`: keys, values and a bucket identifier fn.  
**Output:** `key_ms[]`, `value_ms[]`: keys and values after multisplit  
// `key[]`, `value[]`, `key_ms[]`, `value_ms[]`,  $\mathbf{H}$ , and  $\mathbf{G}$  are all in global memory  
//  $L$  is the number of subproblems (here total number of warps)  
// ===== Pre-scan stage:  
**for** each warp  $i=0:L-1$  **parallel device do**  
    `bucket_id[0:31] = whatBucket(key[32*i + (0:31)])`;  
    `histo[0:m-1] = warp_histogram(bucket_id[0:31])`;  
     $\mathbf{H}[0:m-1][i] = \text{histo}[0:m-1]$ ;  
**end for**  
// ===== Scan stage:  
 $\mathbf{H\_row} = [\mathbf{H}[0][0:L-1], \mathbf{H}[1][0:L-1], \dots, \mathbf{H}[m-1][0:L-1]]$ ;  
 $\mathbf{G\_row} = \text{exclusive\_scan}(\mathbf{H\_row})$ ;  
 $[\mathbf{G}[0][0:L-1], \mathbf{G}[1][0:L-1], \dots, \mathbf{G}[m-1][0:L-1]] = \mathbf{G\_row}$ ;  
// ===== Post-scan stage:  
**for** each warp  $i=0:L-1$  **parallel device do**  
    `bucket_id[0:31] = whatBucket(key[32*i + (0:31)])`;  
    `histo[0:m-1] = warp_histogram(bucket_id[0:31])`;  
    `offsets[0:31] = warp_offsets(bucket_id[0:31])`;  
    **for** each thread  $k=0:31$  **parallel warp do**  
        `final_position[k] = G[bucket[k]][i] + offsets[k]`;  
        `key_ms[final_position[k]] = key[32*i + k]`;  
        `value_ms[final_position[k]] = value[32*i + k]`;  
    **end for**  
**end for**

---

In computing warp-level histogram and local offsets, we aim to use only warp-level voting schemes (ballots) without any shared memory usage, and to use a minimum number of rounds ( $\log m$ ). So, instead of explicitly forming the binary matrix  $\bar{\mathbf{H}}$ , each thread generates its own version of the rows of this matrix and stores it in its local registers as a binary bitmap. Then per-row reduction is equivalent to a warp-wide population count operation (`...popc`), and exclusive scan equates to first masking corresponding bits and then reducing the result. We now describe both in more detail.

<sup>6</sup> Note that we compute  $\bar{\mathbf{H}}$  a second time rather than store and reload the results from the computation in the first step. This is deliberate. We find that the recomputation is cheaper than the cost of global store and load.



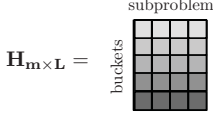

	Operation	Direct MS	Warp-level MS	Block-level MS
Pre-scan	Reading from memory	reading <b>keys</b> from global memory ( $n$ )		
	Computing histograms in each subproblem	warp histogram	warp histogram	1. warp histogram 2. block histogram by reduction over warp results per bucket
	Storing histogram results into global memory  $\mathbf{H}_{m \times L}$	$H = [h_{ij}]_{m \times L}$ $h_{ij}$ : number of elements in $i$ -th bucket and $j$ -th warp		$H = [h_{ij}]_{m \times L}$ $h_{ij}$ : number of elements in $i$ -th bucket and $j$ -th block
Scan	Device-wide exclusive scan on rows of $\mathbf{H}_{m \times L}$ and generate $\mathbf{G}_{m \times L}$	 $\mathbf{G}_{m \times L}$		
Post-scan	Reading from memory	reading <b>keys, values, and global offsets</b> from global memory ( $2n + mL$ )		
	1. Recomputing histograms in subproblems 2. Computing local offsets in subproblems	1. warp histograms 2. warp local offsets		1. warp histograms 2. block histogram by exclusive scan over step 1's results per bucket 3. warp local offsets 4. block local offsets by using steps 2 and 3
	Reordering in subproblems	no reordering	warp-level reordering	block-level reordering
	Computing new local offsets <sup>†</sup>	no changes	lane ID $-\sum_{k=0}^{j-1} h_{k,\ell}$	thread index $-\sum_{k=0}^{j-1} h_{k,\ell}$
	Computing final positions	Computed based on Equation (2): by using $\mathbf{G}$ and new local offsets		
	Final data movements	Moving key-value pairs into final positions in memory		

Table 2: Detailed overview of our three proposed algorithms. Merged columns denote that the discussed operation is identical between the corresponding approaches. <sup>†</sup>: In order to compute new local indexes after the reordering, we assume the (arbitrary) considered key is in the  $j$ -th bucket and in the  $\ell$ -th subproblem. Lane ID denotes a thread's position within a warp, and thread index denotes a thread's position within a block.

**Warp-level histograms** To compute warp-level histograms, we assign each bucket (each row of  $\mathbf{H}$ ) to a thread. That thread is responsible for counting the elements of the warp that fall into that bucket. For cases where there are more buckets than the warp width ( $N_T = 32$ ), we assign  $\lceil m/32 \rceil$  buckets to each thread. First, we focus on  $m \leq 32$ ; Algorithm 2 shows the detailed code.

Each thread  $i$  is in charge of the bucket with an index equal to its lane ID (0–31). Thread  $i$  reads a key, computes that key's bucket ID (0–31), and initializes a warp-sized bitvector (32 bits) to all ones. This bitvector corresponds to threads (keys) in the warp that might have a bucket ID equal to this thread's assigned bucket. Then each thread broadcasts the least significant bit (LSB) of its observed bucket ID, using the warp-wide ballot instruction. Thread  $i$  then zeroes out the bits in its local bitmap that correspond to threads that are broadcasting a LSB that is incompatible with  $i$ 's assigned bucket. This process continues with all other bits of the observed bucket IDs (for  $m$  buckets, that's  $\log m$  rounds). When all rounds are complete, each thread has a bitmap that indicates which threads in the warp have a bucket ID corresponding to its assigned bucket. The histogram result is then a reduction over these set bits, which is computed with a single population count (`__popc`) instruction (line 14). For  $m \geq 32$ , we do the updates in lines 8 and 10 multiple times (for multiple `histo_bmp` registers per thread).

#### Algorithm 2 Warp-level histogram computation

```

1: procedure WARP-HISTOGRAM(bucket_id[0:31])
   Input: bucket_id[0:31]  $\triangleright$  a warp-wide array of bucket IDs
   Output: histo[0:m-1]  $\triangleright$  number of elements within each  $m$  buckets
2:   for each thread  $i = 0:31$  parallel warp do  $\triangleright$  all threads within a warp
3:     histo_bmp[i] = 0xFFFFFFFF;
4:     for (int  $k = 0$ ;  $k < \text{ceil}(\log_2(m))$ ;  $k++$ ) do
5:       temp_buffer = __ballot(bucket_id[i] & 0x01);
6:       if (( $i \gg k$ ) & 0x01) then  $\triangleright$  histogram computation
7:         histo_bmp[i] &= temp_buffer;
8:       else
9:         histo_bmp[i] &= XOR(0xFFFFFFFF, temp_buffer);
10:      end if
11:      bucket_id[i] >>= 1;
12:    end for
13:    histo[i] = __popc(histo_bmp[i]);  $\triangleright$  counting number of set bits
14:  end for
15:  return histo[0:m-1];
16: end procedure

```

**Warp-level local offset computation** Local offset computations follow a similar structure to histograms (Algorithm 3). In local offset computations, however, each thread is only interested in keeping track of ballot results that match its item's *observed* bucket ID, rather than the bucket ID to which it has been assigned. Thus we compute a bitmap that corresponds to threads whose items share

our same bucket, mask away all threads from later buckets, and use the population count instruction to compute the local offset (line 14). Histogram and local offset computations are shown in two separate procedures (Alg. 2 and 3), but since they share many common operations they can be merged into a single procedure if necessary. For example, in Direct MS, we only need histogram computation in the pre-scan stage, but we need both histogram and local offsets in the post-scan stage.

#### Algorithm 3 Warp-level local offset computation

```

1: procedure WARP_OFFSET(bucket_id[0:31])
   Input: bucket_id[0:31]  $\triangleright$  a warp-wide array of bucket IDs
   Output: offset[0:31]  $\triangleright$  for each element, number of preceding elements
       within the same bucket
2:   for each thread  $i = 0:31$  parallel warp do  $\triangleright$  all threads within a warp
3:     offset_bmp[i] = 0xFFFFFFFF;
4:     for (int  $k = 0$ ;  $k < \text{ceil}(\log_2(m))$ ;  $k++$ ) do
5:       temp_buffer = _ballot(bucket_id[i] & 0x01);
6:       if (bucket_id[i] & 0x01) then  $\triangleright$  local offset computation
7:         offset_bmp[i] &= temp_buffer;
8:       else
9:         offset_bmp[i] &= XOR(0xFFFFFFFF, temp_buffer);
10:      end if
11:      bucket_id[i] >>= 1;
12:    end for
13:    offset[i] = _popc(offset_bmp[i] & (0xFFFFFFFF >> (31-i)));
    $\triangleright$  counting number of preceding set bits
14:  end for
15:  return offset[0:31];
16: end procedure

```

**Block-level histograms** For our Block-level MS, we perform the identical computation as Direct MS and Warp-level MS, but on a block granularity. If we chose explicit local computations (described in Section 4) to compute histograms and local offsets, the binary matrix  $\bar{\mathbf{H}}$  would be large, and we would have to reduce it over rows (for histograms) and scan it over rows (for local offsets). Because of this complexity, and because our warp-level histogram computation is quite efficient, we pursue the second option: the hierarchical approach. We first compute histograms for each warp, storing results in shared memory and forming a matrix  $\mathbf{H}_2$  with  $m$  rows and  $N_W$  columns (similar to  $\mathbf{H}$  but among warps within each block). We then reduce block-level histograms per row, which we implement with our own multi-reduction operation in shared memory (in  $\log N_W$  rounds of coalesced shared memory accesses).

**Block-level local offsets** For warp-level local offsets, we begin with a similar scheme to Algorithm 3. Then, the block-level local offset for each element is its warp-level local offset plus the sum of all elements in previous warps and in the same bucket (similar to the second term in equation (2)). So, we must perform another scan operation on block histograms. Finally, the warp that already possesses the final reduction result from the histogram computation performs the scan operation by using warp-wide shuffles.

## 5.2 Reordering for better locality

As described in Section 4, one of the main bottlenecks in a permutation like multisplit is the random scatter in its final data movement. Figure 2 shows an example of such a case. As we suggested previously, we can improve scatter performance by reordering elements locally in each subproblem such that in the final scatter, we get better coalescing behavior (i.e., consecutive elements are written to consecutive locations in global memory).

However, while a higher achieved write memory bandwidth will improve our runtime, it comes at the cost of more local work to reorder elements. Warp-level reordering requires the fewest extra computations, but it may not be able to give us enough locality as the number of buckets increases (Fig. 2). We can achieve better

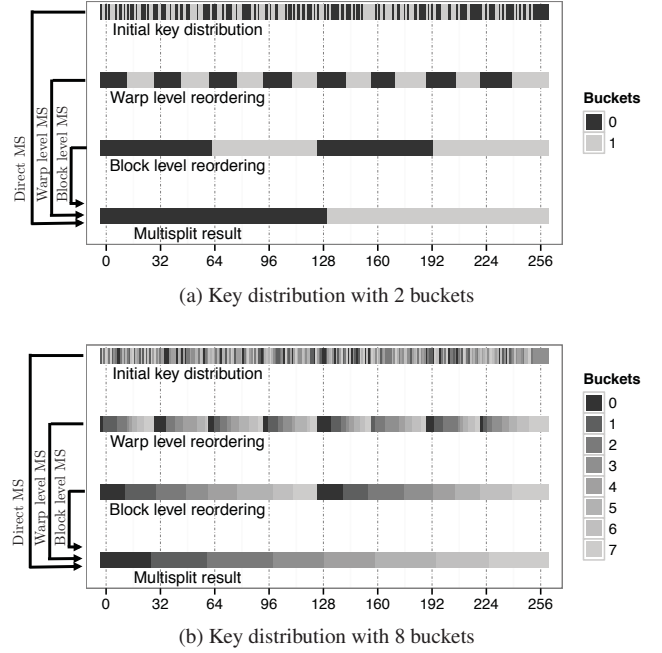


Figure 2: Key distributions for different multisplit methods and different number of buckets. Key elements are initially uniformly distributed among different buckets. This window shows an input key vector of length 256; each warp is 32 threads wide and each block has 128 threads.

locality, again at the cost of more computation, by reordering across warps within a block.

### 5.2.1 Warp-level Reordering

Warp-level MS extends Direct MS by reordering warps before the final write for better memory coalescing behavior. Our first question was whether we prefer to perform the reordering in our pre-scan stage or our post-scan stage. We know that in order to compute the new index for each element in the warp, we need to know about its histogram and we need to perform a local (warp-level) exclusive scan over the results. We have already computed the warp level histogram in the pre-scan stage, but we do not have it in the post-scan stage and thus would either have to reload it or recompute it.

However, if we reorder key-value pairs in the pre-scan stage, we must perform two coalesced global reads (reading key-value pairs) and two coalesced global writes (storing the reordered key-value pairs before our global operation) per thread. Recall that in Direct MS, we only required one global read (just the key) per thread in its pre-scan stage.

In the end, the potential cost of the additional global reads was significantly more expensive than the much smaller cost of recomputing our efficient warp-level histograms. As a result, we reorder in the post-scan stage and require fewer global memory accesses overall.

The only difference between Direct MS and Warp-level MS is in post-scan, where we compute both warp-level histogram and local offsets (Algorithm 2 and 3). Then, in order to find the new index for each key element, we need to know the total number of keys in previous buckets (as in equation (1)) and thus we need an exclusive scan operation over the already-computed histograms. This requires only warp-wide shuffle instructions (`_shfl_up`) in  $\log N_T$  rounds.



Now, each thread has observed a key element and has identified its bucket ID. It also knows which thread was in charge of that bucket ID and can ask for the result of the exclusive scan using a single `__shfl` instruction. Note that local offsets for keys do not change as we reorder them, but we do have to recompute them as well, based on equation (1): a reordered element in shared memory corresponding to lane ID  $i$  that belongs to bucket  $j$  has a local offset equal to  $i - \sum_{k=0}^{j-1} h_{k,\ell}$ . Once again, we know who has the scan result (right term) and can ask for it by `__shfl`. Using this result, we reorder the key-value pairs in shared memory to correspond to the more coalesced order we just computed and write all key-value pairs back to global memory in a coalesced way.

### 5.2.2 Block-level Reordering

The benefit from warp-level reordering is rather modest, particularly as the number of buckets grows, because we only see a small number of elements per warp that belong to the same bucket. For potentially larger gains in coalescing, our block-level MS reorders entire blocks.

Just as with warp-level reordering, we must evaluate equation (2) for all elements within a block. Block-level histograms and our scan operation for local offset computation gives us the first term in equation (2). We have already computed local offsets (the third term) as well. The additional information we need is the total number of elements in the same bucket but in previous warps within that block (the second term in equation (2)). To achieve that, in our second histogram computation in the post-scan stage, instead of row reductions we perform an exclusive scan operation on each row.

We implement our own multi-scan operation to do this with  $2 \log N_T$  coalesced shared memory accesses. Another option was to rearrange elements into shared memory in a row-vectorized fashion and then run a single block-wide scan operation on  $mN_W$  elements. We found this unappealing as it requires random shared memory scatters. Also, because of the way we assign buckets to threads in warps, each warp has a column of  $\mathbf{H}_2$ , which makes it easier to perform element-wise row scan (or reduction) on columns.

**Final data movement** In a similar way to Warp-level MS, since key-value pairs are already reordered in shared memory, we can easily recompute their block-wide local offsets. For example, in block  $\ell$ , if the key element corresponding to thread index  $i$  (i.e., `threadIdx.x`) belongs to bucket  $j$ , then the block-level local offset is  $i - \sum_{k=0}^{j-1} h_{k,\ell}$ , where the second term is already computed by our initial local offset computations. Final positions are then added to the global offsets computed in  $\mathbf{G}$  and key-value pairs are stored in global memory with coalesced writes.

### 5.3 More buckets than the warp width

So far, we have only considered cases where the number of buckets is less than or equal to the warp width  $N_T$ . These cases map nicely to our implementation because we made each thread responsible for a single bucket, and hence all histogram-related data movement could be performed with a single warp access. Adding more buckets presents no theoretical concerns, but will degrade performance.

For more than  $N_T$  buckets, some threads must be responsible for multiple buckets. For example, the  $k$ -th thread in each warp is responsible for counting elements in  $B_k, B_{k+32}, B_{k+64}$ , and so on. Histogram and local offset computations in the pre-scan stage do not change. However, all other histogram-related memory reads/writes are now linearized by a factor of  $\lceil m/32 \rceil$  as well. Section 6.4 shows more details about the performance degradation of such linearization.

## 6. Performance Evaluation

In this section we evaluate our multisplit methods and analyze their performance. All experiments are run on a NVIDIA K40c GPU

Method	Avg. running time	Processing rate
Radix sort (key-only)	22.36 ms	1.50 Gkeys/sec
Radix sort (key-value)	37.36 ms	0.90 Gkeys/sec
Scan-based split (key-only)	5.55 ms	6.05 Gkeys/sec
Scan-based split (key-value)	6.96 ms	4.82 Gkeys/sec

Table 3: Average running time and processing rate, over  $2^{25}$  randomly generated inputs, uniformly distributed over two buckets.

with 12 GB DRAM. All programs are compiled with NVIDIA’s nvcc compiler (version 6.5.12). The authors have implemented all codes except for device-wide scan operations and radix sort, which are from CUB (version 1.4.1). All experiments are run over 50 independent trials. Since the main focus of this paper is on multisplit as a GPU primitive within the context of a larger GPU application, we assume that all required data is already in the GPU’s memory and hence no transfer time is included. Throughout this section, unless otherwise stated, we have used  $N_W = 8$  warps per block (256 threads per block). Choosing an appropriate  $N_W$  is important because it moderately influences the amount of parallelism in each block (for instance, for Warp-level MS,  $N_W = 2$  is 1.4x slower). Its influence is more important for Block-level MS as it affects both the potential locality to be extracted as well as the amount of inter-warp communications (for Block-level MS,  $N_W = 2$  is 2x slower). All our experiments (except in Section 6.5) are over 32-bit random integers uniformly distributed over  $m$  buckets, and buckets are defined to equally divide the 32-bit domain. All our multisplit methods have similar performance for any other 32-bit data (e.g., floating-point numbers).

### 6.1 Common approaches and references

In Section 3, we introduced some of the common approaches to implement multisplit. Table 3 shows performance results for radix sort and the scan-based split methods. With uniform distribution of keys, radix sort’s performance is independent of the number of buckets; instead, it only depends on the number of significant bits. Iterative scan-based split can be used on any number of buckets. For this method, we ideally have a completely balanced distribution of keys, which means in each round we run twice the number of splits as the previous round over half-sized subproblems. So, we can assume that in the best-case scenario, iterative scan-based split’s average running time is lower-bounded by  $\log(m)$  times the runtime of a single scan-based split method. This ideal lower bound is not competitive for any of our scenarios, and thus we have not implemented the iterative part of this method.

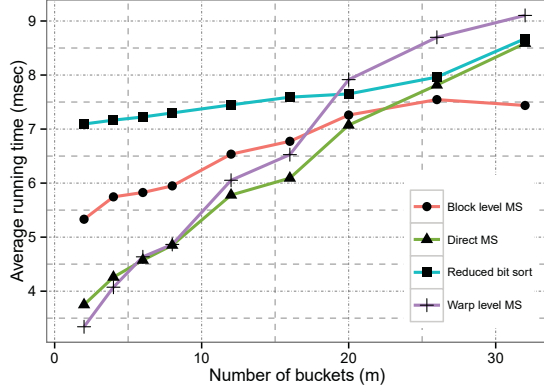
### 6.2 Performance versus number of buckets: $m \leq 32$

In this section we analyze our performance as a function of the number of buckets ( $m \leq 32$ ). Our methods differ in three principal ways: 1) how expensive are our local computations, 2) how expensive are our memory accesses, and 3) how much locality can be extracted by reordering.

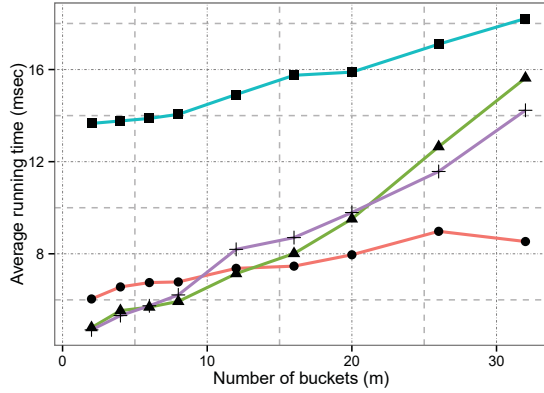
#### 6.2.1 Average running time

Table 4 shows the average running time of different stages in each of our three approaches, the reduced bit sort method, and a lower bound for the recursive scan-based split. All of our proposed methods have the same basic computational core, warp-wide local histogram and local offset computations. However, there are three major reasons behind the performance differences among our methods as the number of buckets increases (each can be verified in Table 4):

**Reordering process** Reordering keys (key-values) requires extra computation and shared memory accesses. Reordering is always



(a) Key-only,  $m \leq 32$



(b) Key-value,  $m \leq 32$

Figure 3: Average running time (ms) versus number of buckets for all multisplit methods: (a) key-only, 32 M elements (b) key-value, 32 M elements.

more expensive for Block-level MS as it also requires inter-warp communications. But both of these negative costs are relatively constant with respect to  $m$  (mostly depend on  $N_W$  and  $N_T$ ).

**Increased locality from reordering** Since block level subproblems have more elements than warp level subproblems, Block-level MS is always superior to Warp-level MS in terms of locality. Both factors decrease by  $\frac{1}{m}$  as  $m$  increases.

**Global operations** As described before, by increasing  $m$ , the height of matrix  $\mathbf{H}$  increases. However, Block-level MS performs most of its computations locally and only stores block histograms in global memory (compared to other methods that store warp histograms). As a result, scan operations for both the Direct MS and Warp-level MS get significantly more expensive by a factor of  $mN_W$ , rather than just  $m$  as in Block-level MS.

Figures 3a and 3b show the average running time of our multisplit algorithms vs. the number of buckets. For small  $m$ , Block-level MS has the best locality (at the cost of substantial local work), but Warp-level MS achieves fairly good locality coupled with simple local computation; it is the fastest choice for small  $m$  ( $\leq 6$  (key-only),  $\leq 5$  (key-value)). For large  $m$  ( $\geq 22$  (key-only),  $\geq 16$  (key-value)), the superior memory locality of Block-level MS coupled with a minimized global scan cost make it the best method overall. In between these extremes, all three methods are roughly similar

Algorithm	Stage	Key-only			Key-value		
		Number of buckets (m)					
		2	8	32	2	8	32
Direct MS	Pre-scan	1.32	1.49	2.19	1.32	1.49	2.19
	Scan	0.12	0.39	1.48	0.12	0.39	1.48
	Post-scan	2.31	2.98	4.92	3.36	4.06	11.97
	Total	3.75	<b>4.85</b>	8.59	4.79	<b>5.93</b>	15.63
Warp level MS	Pre-scan	1.32	1.49	2.19	1.32	1.49	2.19
	Scan	0.12	0.39	1.47	0.12	0.40	1.47
	Post-scan	1.91	2.99	5.44	3.27	4.34	10.56
	Total	<b>3.34</b>	4.86	9.11	<b>4.70</b>	6.22	14.23
Block level MS	Pre-scan	1.59	1.58	1.88	1.59	1.58	1.88
	Scan	0.03	0.07	0.21	0.03	0.07	0.21
	Post-scan	3.70	4.30	5.35	4.41	5.13	6.44
	Total	5.33	5.95	<b>7.44</b>	6.04	6.78	<b>8.53</b>
Reduced bit sort	Labeling	2.07	2.07	2.07	2.07	2.07	2.07
	Sorting	5.01	5.22	6.60	5.94	6.33	10.49
	(un)Packing	—	—	—	5.66	5.66	5.66
	Total	7.09	7.29	8.67	13.67	14.06	18.22
Recursive split	Labeling	1.54	4.62	7.70	1.54	4.62	7.70
	Scan	1.47	4.41	7.35	1.47	4.41	7.35
	Splitting	2.54	7.62	12.7	3.95	11.85	19.75
	Total	5.55	16.65	27.75	6.96	20.88	34.8
Sort on identity buckets <sup>†</sup>		2.62	2.68	4.20	5.01	5.22	6.60

Table 4: Average running time (ms) for different stages of our multisplit approaches, reduced bit sort, and recursive split, with  $n = 2^{25}$  and a varying number of buckets. Because we did not fully implement recursive scan-based split, its runtimes are expressed as a lower (ideal) bound equal to the scan-based split time multiplied by  $\log m$  rounds. <sup>†</sup>: Radix sort results for the trivial case with identity buckets ( $B_i = \{i\}$ ) (Section 3.1).

in performance, staking out different points in the tradeoff space between complexity and global scan/scatter cost.

As described in Section 3.1, in this paper we have focused on non-trivial bucket identification scenarios, without trivial cases such as identity buckets. For example, if our original key elements are all from  $0, 1, \dots, m-1$ , and all buckets are defined as  $B_i = \{i\}$ , then each key element is equal to its bucket ID and thus we can directly do sorting rather than multisplit. As shown in Table 4, in such a scenario radix sort is almost always more efficient than our multisplit methods (the exception is 2-bucket key-value pairs). We should note that such a key-only case has little to no practical use, but we included it for the sake of clear comparison. Part of this difference is caused by kernel overheads in our methods from launching multiple kernels.

For other non-trivial cases and with uniform distribution of keys among all 32-bit integers, by comparing Table 4 and Table 3 it becomes clear that our multisplit method outperforms radix sort by a big margin. Our multisplit methods are also always superior to the reduced bit sort method that we introduced in Section 3.4. This is partly because of the extra overheads that we introduced for bucket identification and creating the label vector. Even if we ignore this overhead, since reduced bit sort performs its operations and permutations over the label vector as well as original key (key-value) elements, its data movements are more expensive compared to all our multisplit methods that only process and permute original key (key-value) elements.

## 6.2.2 Processing rate, and multisplit speed of light

It is instructive to compare any implementation to its “speed of light”: a processing rate that could not be exceeded. For multisplit’s speed of light, we consider that computations take no time and all memory accesses are fully coalesced. Our parallel model requires

Scenario	Algorithm	Number of buckets (m)				
		2	4	8	16	32
Key-only	Direct MS	8.95	7.88	<b>6.92</b>	<b>5.51</b>	3.91
	Warp level MS	<b>10.04</b>	<b>8.23</b>	6.90	5.14	3.69
	Block level MS	6.29	5.84	5.64	4.95	<b>4.51</b>
	Reduced bit sort	4.64	4.60	4.51	4.34	3.85
Key-value	Direct MS	7.00	6.06	<b>5.66</b>	4.19	2.15
	Warp level MS	<b>7.14</b>	<b>6.31</b>	5.40	3.86	2.36
	Block level MS	5.56	5.11	4.95	<b>4.50</b>	<b>3.93</b>
	Reduced bit sort	2.46	2.44	2.39	2.13	1.84

Table 5: Processing rate (G keys/sec) for our multisplit methods on  $n = 2^{25}$  keys with a uniform distribution across buckets.

one single global read of all elements before our global operation barrier to compute histograms. We assume the global operation is free. Then after the global operation, we must read all keys (or key-value pairs) and then store them into their final positions. For multisplit on keys, we thus require 3 global memory accesses per key; 5 for key-value pairs. Our GPU has a peak memory bandwidth of 288 GB/s, so the speed of light for keys, given the many favorable assumptions we have made for it, is 24 Gkeys/s, and for key-value pairs is 14.4 G keys/s.

Table 5 shows our processing rates for 32M keys and key-value pairs, uniformly distributed among buckets. Warp-level MS has the highest peak throughput (on 2 buckets), 10.04 G key/s. Our achieved rates significantly outperform radix sort and scan-based split (Table 3).

### 6.3 Performance on different GPU microarchitectures

As discussed in the beginning of this section, we have used NVIDIA’s Tesla K40c GPU for all our experiments. This GPU, designed primarily for servers, is NVIDIA’s most powerful computational device. The K40c is based on NVIDIA’s “Kepler” microarchitecture. In this part we want to briefly explore performance results of our multisplit methods on other NVIDIA microarchitectures as well. In our design we have not used any (micro)architecture-dependent optimizations and hence we do not expect radical behavior change on any other machine, other than possible speedup differences based on the device’s capability. We have repeated our experiments on a GeForce GTX 750 Ti GPU, a mid-level consumer GPU based on NVIDIA’s newer “Maxwell” microarchitecture. On the GTX 750 Ti, radix sort over 32 M key elements and key-value pairs achieves 0.80 and 0.48 G keys/sec processing rate respectively (compared to the K40c’s 1.50 and 0.90 G keys/sec from Table 3). Table 6 shows the speedup of our multisplit methods and the reduced-bit sort against radix sort on both devices.

As we expected, the general behavior is similar on both microarchitectures. The major difference for the Maxwell device is the superiority of reordering-based methods (Warp-level MS and Block-level MS) compared to the Direct MS method. We believe this indicates that the Tesla-based K40c does a better job in hiding latencies imposed by non-coalesced memory accesses than the one in our Maxwell device. In other words, the reordering process that we implement to avoid non-coalesced memory accesses appears to be even more valuable in the newer Maxwell microarchitecture.

### 6.4 Performance for more than 32 buckets

In this section we consider scenarios with more than 32 buckets ( $m > N_T$ ). As discussed in Section 5.3, our algorithms were specifically designed to exploit warp-level primitives for their histogram computations. With more buckets, our histogram computations require  $\log m$  rounds; all intermediate registers and data movements must be scaled by  $\lceil m/32 \rceil$ . We concentrate on Block-level

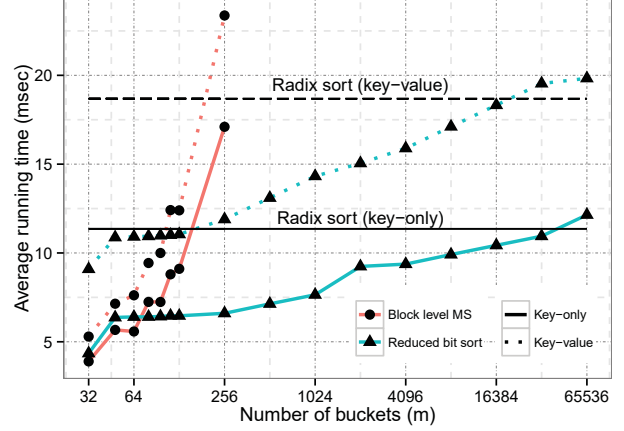


Figure 4: Average running time (ms) versus  $m \geq 32$  for Block-level MS and reduced bit sort, 16 M elements.

MS, which had our best performance as the number of buckets approached (and exceeded) 32 buckets.

Because all our previous intermediate data movements to shared memory are now scaled, our previous multi-scan and multi-reduction procedures (Section 5.1) are not as efficient any more (the height of  $\bar{\mathbf{H}}$  is now more than the warp-width). So instead, we do what we did with our global scan operations: we store a row-vectorized  $\bar{\mathbf{H}}$  in shared memory and then run a single block-wide scan operation of size  $mN_W$  (we use CUB’s scan implementation for this). The rest of our procedure is similar to the  $m < 32$  case.

As the number of buckets grows, radix sort is increasingly attractive: it has no performance dependence on the number of buckets. Since the reduced bit sort uses a regular radix sort in its logarithmic number of stages, its performance should scale with the number of buckets only logarithmically, as opposed to our methods, which in some parts scale linearly. So, our expectation is that as the bucket count grows, reduced bit sort will eventually become the best method. Figure 4 shows the average running time of our Block-level MS and reduced-bit sort vs. the number of buckets until they converge to radix sort’s performance limit. In a key-only (key-value) scenario, Block-level MS remains superior to reduced-bit sort until almost 64 (96) buckets, and finally converges to radix sort for 192 (224) buckets. Reduced-bit sort scales much better with the number of buckets: it converges to radix sort for almost 32k (key) and 16k (key-value) buckets. One of the important bottlenecks for our methods with a high number of buckets is our heavy usage of shared memory within a block. In our implementations, we assign an exclusive part of shared memory for each column of  $\bar{\mathbf{H}}$  of length  $m$  per warp. As the number of buckets increases, we are still observing the same 32 key elements per warp, which means on average most of our elements in  $\bar{\mathbf{H}}$  are zero ( $\bar{\mathbf{H}}$  becomes very sparse). Future work may choose a different approach to address the sparsity of  $\bar{\mathbf{H}}$  as bucket count becomes large.

### 6.5 Initial key distribution over buckets

So far we have only considered scenarios in which initial key elements were uniformly distributed over buckets (i.e., a uniform histogram). In our implementations we have considered small subproblems (warps and blocks) compared to the total size of our initial key vector. Since these subproblems are relatively small, having a non-uniform distribution of keys means that it would be more likely to see empty buckets in some of our subproblems; in practice, our methods would behave as if there were fewer buckets



Scenario	Algorithm	Tesla K40c (Kepler)					GeForce GTX 750 Ti (Maxwell)				
		Number of buckets (m)					Number of buckets (m)				
		2	4	8	16	32	2	4	8	16	32
Key-only	Direct MS	5.97x	5.25x	<b>4.61x</b>	<b>3.67x</b>	2.60x	4.67x	3.73x	2.80x	2.52x	1.52x
	Warp level MS	<b>6.69x</b>	<b>5.49x</b>	4.60x	3.43x	2.46x	<b>5.61x</b>	<b>4.26x</b>	<b>3.39x</b>	2.63x	1.70x
	Block level MS	4.20x	3.89x	3.76x	3.30x	<b>3.01x</b>	3.32x	3.14x	2.96x	<b>2.88x</b>	<b>2.73x</b>
	Reduced bit sort	3.15x	3.12x	3.06x	2.95x	2.58x	2.90x	2.82x	2.76x	2.72x	2.65x
Key-value	Direct MS	7.80x	6.75x	<b>6.30x</b>	4.66x	2.39x	5.65x	3.86x	2.83x	2.41x	1.45x
	Warp level MS	<b>7.95x</b>	<b>7.03x</b>	6.01x	4.29x	2.62x	<b>6.35x</b>	<b>5.32x</b>	4.00x	3.03x	1.66x
	Block level MS	6.19x	5.69x	5.51x	<b>5.01x</b>	<b>4.38x</b>	4.47x	4.36x	<b>4.23x</b>	<b>4.06x</b>	<b>3.40x</b>
	Reduced bit sort	2.73x	2.71x	2.66x	2.37x	2.05x	2.12x	2.12x	2.11x	2.08x	2.06x

Table 6: Speedup versus radix sort for a varying number of buckets and two GPU architectures. The initial key vector of length  $n = 2^{25}$  is uniformly distributed over buckets. Each speedup is computed against the radix sort performance on that specific device.

for those subproblems. All of our *computations* (e.g., warp-level histograms) are data-independent and, given a fixed bucket count, would have the same performance for any distribution. However, our *data movement*, especially after reordering, would benefit from having more elements within fewer buckets and none for some others (resulting in better locality for coalesced global writes). Consequently, the uniform distribution is the worst-case scenario for our methods.

In this section we consider a binomial distribution as an example of a non-uniform distribution. In general  $B(m-1, p)$  denotes a binomial distribution over  $m$  buckets with a probability of success  $p$ . For example, the probability that a key element belongs to bucket  $0 \leq k < m$  is  $\binom{m-1}{k} p^k (1-p)^{m-k-1}$ . This distribution forces an unbalanced histogram as opposed to the uniform distribution. Note that by choosing  $p = 0.5$ , the expected number of keys within the  $k$ th bucket will be  $n \binom{m-1}{k} 2^{1-m}$ . We also consider a milder distribution where 25% of keys are uniformly distributed among buckets and the rest are within one bucket. Figure 5 shows the average running time of Block-level MS and reduced-bit sort for different distributions. Both methods perform better as keys become less uniformly distributed, because on average there is less intermediate data movement. The reduced-bit sort is more sensitive, because the internal hierarchical approach in radix sort helps it to put most of its effort on more common bits. With a uniform distribution, all bits are equally populated, so all bits get the same attention.

## 7. Conclusion

The careful design and analysis of our GPU multisplit implementations allow us to provide significant performance speedups for multisplit operations over traditional sort-based methods. Beyond simply demonstrating the design and implementation of a family of fast and efficient multisplit primitives, we offer three main lessons that are broadly useful for parallel algorithm design and implementation: Minimize global (device-wide) operations, even at the cost of increased local computation; the benefit of more coalesced memory accesses outweighs the cost of local reordering; and leveraging warp-wide hardware intrinsics and warp-synchronous programming, where applicable, is both highly beneficial and superior to shared-memory-based communication.

## Acknowledgments

The authors would like to thank Michael Garland and Sean Baxter for their valuable comments on paper drafts. Also, thanks to NVIDIA for providing the GPUs that made this research possible. We appreciate the funding support from UC Lab Fees Research Program Award 12-LR-238449, DFG grant ME 2088/3-1, MADALGO (Center for Massive Data Algorithmics), NSF awards CCF-1017399 and OCI-1032859, and Sandia LDRD award #130144.

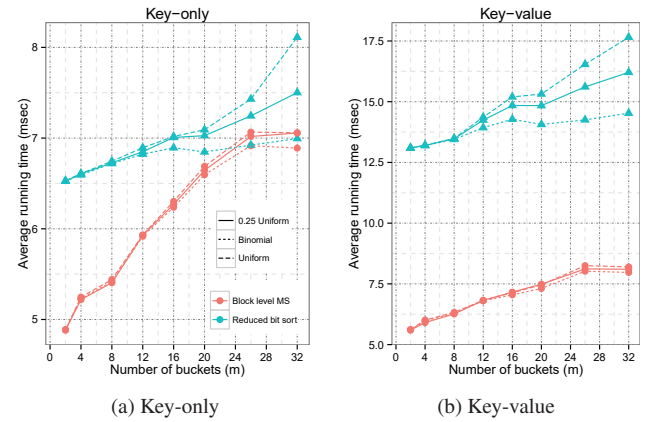


Figure 5: Average running time vs. number of buckets for three different initial key distributions: a uniform distribution, a binomial distribution  $B(m-1, 0.5)$ , and a distribution with 25% of keys uniformly distributed and the rest in just one bucket.

## References

- [1] The Graph 500 list. <http://www.graph500.org/>, July 2013.
- [2] Yahoo labs dataset selections. <http://webscope.sandbox.yahoo.com/>, July 2013.
- [3] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics*, 28(5):154:1–154:9, Dec. 2009. doi: 10.1145/1661412.1618500.
- [4] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14*, pages 781–792, Nov. 2014. doi: 10.1109/SC.2014.69.
- [5] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*, chapter 3.3.4: The Bellman-Ford-Moore Algorithm, pages 97–99. Springer-Verlag London, 2009. doi: 10.1007/978-1-84800-998-1.
- [6] S. Brown and J. Snoeyink. Modestly faster histogram computations on GPUs. In *Proceedings of Innovative Parallel Computing, InPar '12*, May 2012. doi: 10.1109/InPar.2012.6339589.
- [7] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single source shortest paths. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2014*, pages 349–359, May 2014. doi: 10.1109/IPDPS.2014.45.

- [8] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011. doi: 10.1145/2049662.2049663.
- [9] M. Deo and S. Keely. Parallel suffix array and least common prefix for the GPU. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 197–206, Feb. 2013. doi: 10.1145/2442516.2442536.
- [10] A. Deshpande and P. J. Narayanan. Can GPUs sort strings efficiently? In *20th International Conference on High Performance Computing*, HiPC 2013, pages 305–313, Dec. 2013. doi: 10.1109/HiPC.2013.6799129.
- [11] G. F. Damos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. Technical Report GIT-CERCS-12-01, Georgia Institute of Technology Center for Experimental Research in Computer Systems, Feb. 2012. URL <http://www.cercs.gatech.edu/tech-reports/tr2012/git-cercs-12-01.pdf>.
- [12] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390.
- [13] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, Aug. 2007.
- [14] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 511–524, June 2008. doi: 10.1145/1376616.1376670.
- [15] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474, Apr. 2011. doi: 10.1109/TVCG.2010.88.
- [16] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2): 39–55, Mar./Apr. 2008. doi: 10.1109/MM.2008.31.
- [17] D. Merrill and A. Grimshaw. Revisiting sorting for GPGPU stream architectures. Technical Report CS2010-03, Department of Computer Science, University of Virginia, Feb. 2010. URL <https://sites.google.com/site/duanemerrill/RadixSortTR.pdf>.
- [18] U. Meyer. Buckets strike back: Improved parallel shortest paths. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS 2002, Apr. 2002. doi: 10.1109/IPDPS.2002.1015582.
- [19] U. Meyer. Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds. *Journal of Algorithms*, 48(1): 91–134, Aug. 2003. doi: 10.1016/S0196-6774(03)00046-4.
- [20] U. Meyer and P. Sanders.  $\Delta$ -stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, Oct. 2003. doi: 10.1016/S0196-6774(03)00076-2. 1998 European Symposium on Algorithms.
- [21] G. L. Miller and J. H. Reif. Parallel tree contraction—Part 1: Fundamentals. In S. Micali, editor, *Randomness and Computation*, volume 5 of *Advances in Computing Research*, pages 47–72. JAI Press Inc., 1989. ISBN 9780892328963.
- [22] L. Monroe, J. Wendelberger, and S. Michalak. Randomized selection on the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 89–98, Aug. 2011. doi: 10.1145/2018323.2018338.
- [23] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, Mar./Apr. 2008. doi: 10.1145/1365490.1365500.
- [24] C. Nugteren, G.-J. van den Braak, H. Corporaal, and B. Mesman. High performance predictable histogramming on GPUs: Exploring and evaluating algorithm trade-offs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 1. ACM, 2011.
- [25] NVIDIA Corporation. NVIDIA CUDA C programming guide. PG-02829-001\_v6.5, Aug. 2014.
- [26] J. Pantaleoni. VoxelPipe: A programmable pipeline for 3D voxelization. In *Proceedings of High Performance Graphics*, HPG '11, pages 99–106, Aug. 2011. ISBN 978-1-4503-0896-0. doi: 10.1145/2018323.2018339.
- [27] S. Patidar. Scalable primitives for data mapping and movement on the GPU. Master's thesis, International Institute of Information Technology, Hyderabad, India, June 2009.
- [28] R. Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proceedings of the International Conference on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, Gold Coast, Australia, Dec. 2007.
- [29] Z. Wu, F. Zhao, and X. Liu. SAH KD-tree construction on GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 71–78, Aug. 2011. doi: 10.1145/2018323.2018335.
- [30] X. Yang, D. Xu, and L. Zhao. Efficient data management for incoherent ray tracing. *Applied Soft Computing*, 13(1):1–8, Jan. 2013. doi: 10.1016/j.asoc.2012.07.002.