Efficient Parallel Merge Sort for Fixed and Variable Length Keys

Andrew Davidson
University of California, Davis
aaldavidson@ucdavis.edu

David Tarjan
NVIDIA Corporation
dtarjan@nvidia.com

Michael Garland
NVIDIA Corporation
mgarland@nvidia.com

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

ABSTRACT

We design a high-performance parallel merge sort for highly parallel systems. Our merge sort is designed to use more register communication (not shared memory), and does not suffer from over-segmentation as opposed to previous comparison based sorts. Using these techniques we are able to achieve a sorting rate of 250 MKeys/sec, which is about 2.5 times faster than Thrust merge sort performance, and 70% faster than non-stable state-of-the-art GPU merge sorts.

Building on this sorting algorithm, we develop a scheme for sorting variable-length key/value pairs, with a special emphasis on string keys. Sorting non-uniform, unaligned data such as strings is a fundamental step in a variety of algorithms, yet it has received comparatively little attention. To our knowledge, our system is the first published description of an efficient string sort for GPUs. We are able to sort strings at a rate of 70 MStrings/s on one dataset and up to 1.25 GB/s on another dataset using a GTX 580.

1. INTRODUCTION

Sorting is a widely-studied fundamental computing primitive that is found in a plethora of algorithms and methods. Sort is useful for organizing data structures in applications such as sparse matrix-vector multiplication [3], the Burrows-Wheeler transform [1, 15], and Bounding Volume Hierarchies (LBVH) [11]. While CPU-based algorithms for sort have been thoroughly studied, with the shift in modern computing to highly parallel systems in recent years, there has been a resurgence of interest in mapping sorting algorithms onto these architectures.

For fixed key lengths where direct manipulation of keys is allowed, radix sort on the GPU has proven to be very efficient, with recent implementations achieving over 1 GKeys/sec [13]. However, for long or variable-length keys (such as strings), radix sort is not as appealing an approach: the cost of radix sort scales with key length. Rather, comparison-based sorts such as merge sort are more appealing since one can modify the comparison operator to handle variable-length keys. The current state of the art in comparison sorts on the GPU include a bitonic sort by Peters et al. [16], a bitonic-based merge sort (named Warpsort) by Ye et al. [26] a Quicksort by Cederman and Tsigas [5] and sample sorts by Leischner et al. [12] and Dehne and Zaboli [8].

In this work we implement a merge-sort-based comparison sort that is well-suited for massively parallel architectures like the GPU. Since a GPU requires hundreds or thousands of threads to reach bandwidth saturation, an efficient GPU comparison sort must select a sorting implementation that has ample independent work at every stage. Merge sort is therefore well-suited for the GPU as any two pre-sorted blocks can be merged independently. We focus on designing an efficient stable merge sort (order preserved on ties) that reduces warp divergence, avoids over-segmenting blocks of data, and increases register utilization when possible. We extend our techniques to also implement an efficient variable-key sort (string-sort). Our two major contributions are a fast stable merge sort that is the fastest current comparison sort on GPUs, and the first GPU-based string-sort of which we are aware.

2. RELATED WORK

Sorting has been widely studied on a broad range of architectures. Here we concentrate on GPU sorts, which can generally be classified as radix or comparison sorts.

Radix sorts rely on a binary representation of the sort key. Each iteration of a radix sort processes b bits of the key, partitioning its output into 2^b parts. The complexity of the sort is proportional to b, the number of bits, and n, the size of the input ($O(bn)$), and fast scan-based split routines that efficiently perform these partitions have made the radix sort the sort of choice for key types that are suitable for the radix approach, such as integers and floating-point numbers. Merrill and Grimshaw's radix sort [13] is integrated into the Thrust library and is representative of the fastest GPU-based radix sorts today. However, as keys become longer, radix sort becomes proportionally more expensive from a computational perspective, and radix sort is not suitable for all key types/comparisons (consider sorting integers in Morton order [14], for instance).

Comparison sorts can sort any sequence using only a user-specified comparison function between two elements and can thus sort sequences that are unsuitable for a radix sort. Sorting networks stipulate a set of comparisons between elements that result in a sorted sequence, traditionally with $O(n \log^2 n)$ complexity. Because those comparisons have ample parallelism and are oblivious to the input, they have been used for sorting since the earliest days of GPU computing [17]. Recent sorting-network successes include an implementation of Batcher's bitonic sort [16].

The classic Quicksort is also a comparison sort that lacks the

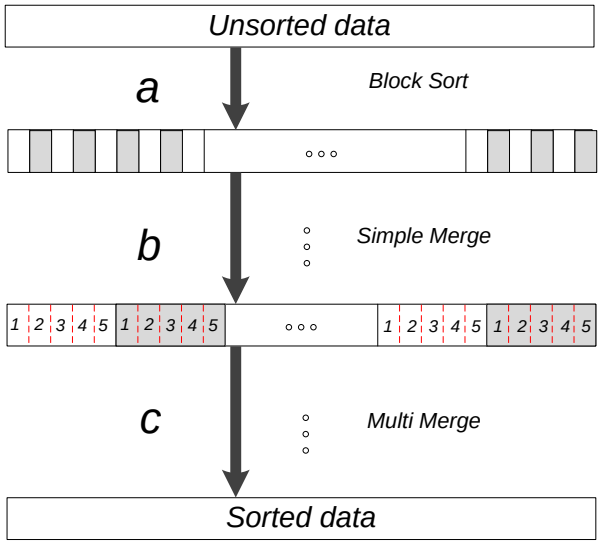


Figure 1: *A high level overview of our hierarchical merge sort. In (a) each CUDA block performs an independent sort on a set of elements. The second stage (b) CUDA blocks work independently to merge two sequences of data. In the final step (c), CUDA blocks cooperate to merge sequences.*

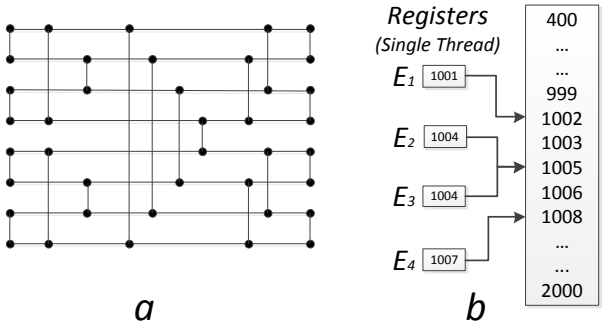


Figure 2: *Our block sort consists of two steps. First each thread performs an eight-way bitonic sort in registers (as seen in* a*). Then each thread is responsible for merging those eight elements in the log(numElements/8) stages.*

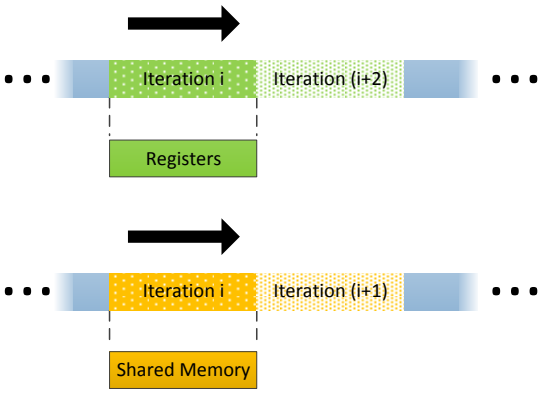


Figure 3: *In order to merge partitions much larger than a GPU's shared memory and register capacity, we must manage two moving memory windows. From two sorted sequences we load a portion of one set into registers (A), and a portion of the second into shared memory (B). Then we update each memory window according to the largest elements, as illustrated above.*

obliviousness of a sorting network, instead relying on a divide-and-conquer approach. It has superior complexity ($O(n \log n)$) to sorting networks, but is more complex from a parallel implementation perspective. Quicksort was first demonstrated on the GPU by Sengupta et al. [19] and was further addressed by Cederman and Tsigas [5]. Sample sorts generalize quicksorts; rather than splitting the input in 2 or 3 parts as in quicksort, they choose representative or random *splitter* elements to divide the input elements into many buckets, typically computing histograms to derive element offsets, then sort each bucket independently. Leischner et al. [12] use random splitters and Dehne and Zaboli [8] deterministic splitters in their GPU implementations.

Merge sorts are $O(n \log n)$ comparison sorts that recursively merge multiple sorted subsequences into a single sorted sequence. Typical mergesort approaches on GPUs use any sorting method to sort small chunks of data within a single GPU core, then recursively merge the resulting sorted subsequences [18].

Hybrid approaches to sorting, typically used to best exploit different levels of the GPU's computational or memory hierarchies during computation, are common, such as the highly successful radix-bitonic sort of Govindaraju et al. [9] or the quicksort-mergesort of Sintorn and Assarsson [21].

The GPU methods above do not focus on complex key types like strings; in fact most of the above comparison-based work simply uses fixed-length integers and do not address complex comparison functions like string-compare. The above work also does not address the complexities of storing variable-length keys like strings. In the CPU world, string keys merit special attention; a representative example (among many) is the burstsort-based string-sorting technique of Sinha et al. [20] that optimizes cache management of string keys to achieve better locality and performance.

3. MERGE SORT

In this section we will describe our merge sort implementation, and compare our design to the previous state-of-the-art GPU merge sort. We organize our sort into a hierarchical three-stage system.

We design a merge sort implementation that attempts to avoid shared memory communication when possible(favoring registers), uses a persistent-thread model for merging, and reduces the number of binary searches. Due to these choices, we utilize more register communication and handle larger partitions at a time to avoid load imbalance. Our merge sort consists of three stages, designed to handle different amounts of parallelism. We will next highlight our design choices for each stage in our merge sort.

Block Sort.

Since local register communication has much higher throughput than standard shared memory communication on GPUs (such as NVIDIA CUDA capable cards), using registers for sorting when possible is preferred. A common strategy to achieve higher register utilization on GPUs is to turn fine-grained threads (that handle one element) into fatter, coarse threads (that handle multiple elements).

Work in GPU linear algebra kernels by Volkov et al. [22–24] has shown that quite frequently sacrificing occupancy(the ratio of active warps to possible warps) for the sake of better register utilization leads to higher throughput. Though in these linear algebra kernels, the communication pattern is quite different from our merge sorts downsweep pattern, more recent work by Davidson et al. has also shown that similar divide-and-conquer methods can benefit from fewer threads with higher register utilization (*register packing*) [6, 7]. In order to achieve more register communication, we split our blocksort step into two stages. We decompose our block sort into two stages. First each *thread* loads eight concurrent elements in registers, and sorts them locally using a bitonic sort as illustrated in Figure 2. Though bitonic sorters are in general non-stable, since we are only sorting eight elements we can carefully construct our sorting network to maintain stability.

Now that we have a set of sorted sequences (eight elements each), we can begin merging them. A common strategy to merge two sorted sequences A and B is to search for the *intersection index* of each element in A into B, and vice versa. The final index for an element A_i after being merged is $i + j$ where $B_j < A_j \leq B_{j+1}$. After each elements output index (the sum of your own index and the intersecting index) has been calculated the resulting list $C = \text{merge}(A, B)$ will be a new sorted sequence. This merged sequence C will have size $\text{sizeof}(A) + \text{sizeof}(B)$. Since each thread operates independently, in order to locate the correct *intersection index*, previous parallel merge sorts have used binary search for each element in both partitions.

Since, in our case, each block is sorting m elements (m for us is heuristically selected to be 1024), we now have $\frac{m}{8}$ threads per block, each with eight sorted elements. Our second stage involves $\log(\frac{m}{8})$ stages to merge all these sorted sequences into a final sorted block. In each merge stage, every thread (still responsible for eight elements) changes its *search space* in order to calculate the correct indices for its elements. In the first merge stage, the *search space* is the neighboring eight elements, in the second stage it becomes sixteen, and so on. Once a thread has calculated the correct index to insert its element, it dumps the sorted values into shared memory. After each merge step a thread then synchronizes (as intermediate values are dumped into shared memory), and then loads eight new concurrent elements for the next merge stage.

Instead of each thread performing eight binary searches to find each *insertion index*, we perform a binary search for the first element and use a linear search for the other seven elements, using the results of our initial binary search as a starting point. Since each thread is handling a sorted subsequence, a search using the first element will most likely be close in relation to a search using the next element. This is illustrated in Figure 2. Though it is possible that in the worst case these linear searches will require more trips to memory ($O(n)$ vs $O(\log(n))$), we find in our experiments that in the average case these secondary linear searches require only a fraction of the accesses that a binary search would need.

Merge Sort—Simple.

After our block sort stage, we have t blocks of m size. In our next stage we want each CUDA block to work independently in merging two of these sorted sequences together. At every step we halve the number of blocks and double the size of each sequence we are merging. Our design goal is to create a merge which utilizes shared memory at every stage for our binary and linear searches even though our hardware shared memory size remains fixed. In order for our algorithm to be able to handle sequences of arbitrary size and still use shared memory effectively, we design two moving

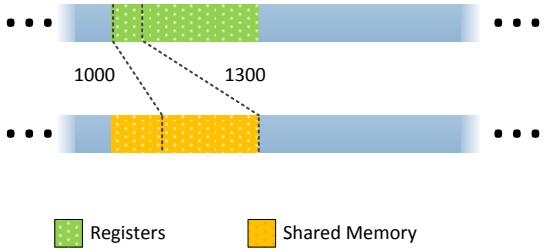


Figure 4: *Example of the state of two memory windows of A and B.*

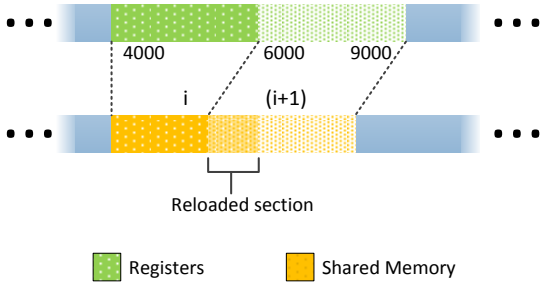


Figure 5: *Optimization to reduce load imbalance. We can reload elements for the next binary search in order to get more overlap (6000–9000 range in this example).*

memory windows: one in registers, and one in shared memory.

Our moving memory windows work as follows: Each thread loads a set of k values it will merge into registers (again using our *register packing* strategy), starting at the beginning of one sequence (sequence A). We now have k elements and q threads. The product of k and q gives us the number of elements in registers that a block can handle at a time. This then sets the size of the moving window in partition A. Next we load into shared memory a set of values from the other sequence (sequence B). The size of this moving window is set by hardware limitations on shared memory size allowed per block. We will refer to the size of the register window as a_s and the shared memory window as b_s. By keeping track of the largest and smallest values in each moving window (register and shared memory), after each thread finishes updating its current values, a block will decide whether to load new values into registers, new values into shared memory or both. This process is illustrated in Figure 3. We find that performance is improved if we have our shared memory windows be larger than our register windows.

Now we can step through and merge partitions of arbitrary size. However, implementing our merge as just described has one major disadvantage. Since we are updating blocks based on the status of two moving windows, a block cannot update its register window until *all* other threads are merged. Similarly, a block cannot update shared memory values until all register values in A have an opportunity to check if their *insertion index* is within the range of the maximum and minimum values in this window. Therefore the

amount of work a block will do at any given time is the union of elements that share the same range in the register and shared memory windows as illustrated in Figure 4. This leads to some load imbalance between blocks, since threads are responsible for concurrent values. However, if all values a warp is responsible for lies outside this union, no useless work will be done. Therefore for every sequence a block handles there can only be at most one divergent warp (portion of the threads searching while others stand idle) at any given time.

However we still have a possible load-balance issue. If the overlap in our valid window ranges is very small, some blocks will have a lot of work while others may have little to no work. We can help address this issue by modifying our window update step in the following way. Before we update our register window, first check the *insertion index* associated with the last value within that window (the last value in the last thread). We can broadcast that index to all threads, and reload the entire shared memory window with this additional offset.

Since all threads need to calculate their insertion index regardless, this step adds no extra computation. An example of this is illustrated in Figure 5. A disadvantage from this modification is that we will have to reload a portion of values from B into shared memory ($b_s - a_i$ values must be reloaded). Therefore we heuristically choose a switching threshold k such that if $b_s - a_i \geq k$ we perform this shift load.

Unfortunately, applying this same optimization to the register window (A) is not as easy. We would require all threads to vote in shared memory whether they lie inside or outside of their current shared memory window, then the sum of these votes would be the offset for our shift load. However, this requires extra work, atomic operations, and the cost of loading a whole new set of register values. Therefore, in general, this shift load optimization only makes sense for our shared memory moving window.

As in our block sort, each thread will be responsible for multiple values. Every thread checks to see if its value is in the search window's valid range, performs a binary search to find the correct index for its first value, and then performs linear searches for the consecutive elements in that thread.

Merge Sort—Multiple.

Though we can now merge very large sequences within a single block, the final steps of our merge sort will only have a few independent sequences to sort(and the last merge only two independent blocks). Modern GPUs require dozens of blocks to maintain optimal throughput. Therefore we require a method that allows CUDA blocks to cooperate in merging two sequences. Therefore the final stage in our hierarchical sort splits sequences in A into multiple partitions, that are then mapped to appropriate partitions in B.

However, unlike previous GPU merge sorts, we do not bound the number of elements in a partition. Given a set of sequences l we define a number of needed blocks p to fill our GPU. We then select the number of partitions per sequence s such that $s \cdot l = p$. Therefore, we still do not suffer from over-segmentation, yet keep the machine as occupied as possible. Davidson et al. used a similar strategy for solving large tridiagonal systems, where multiple CUDA blocks cooperate on a single large system [6].

Otherwise, we utilize the same principles as our previous step. Each thread is responsible for multiple elements, a subset from a partition in sequence A and B are loaded into register and shared memory windows respectively, and then we perform a binary search for the first element, followed by linear searches.

Now we have a modular hierarchical merge sort that can handle an arbitrary number of partitions and sequence sizes. An overview of our merge sort hierarchy can be seen in Figure 1. We will now compare our sort with the previous state-of-the-art merge sort on the GPU.

3.1 Previous Merge Sorts

Satish et al. developed a comparison-based merge sort which is implemented in a divide and conquer approach [18]. The sorts they created outperformed carefully optimized multicore CPU sorts, which helped dispel the myth that GPUs were ill-suited for sorting.

Their algorithm is designed in the following way. First they divide and locally sort a set of t tiles using Batcher's odd-even merge sort [2]. Next, each block merges two adjacent sequences at a time ($\log_2 t$ stages) until the entire sequence is sorted. In order to maintain parallelism, and ensure they are able to use shared memory at every stage, their method splits each sub-partition to make sure each sequence being merged is of a relatively small size (256 elements in their example). In a merge stage, each element in A is assigned to a thread; this thread then performs a binary search into the B partition. Since each of these partitions are sorted, the sum of its index with the binary search index gives the output index in C. This merge process is repeated until the entire dataset is sorted.

This is only a brief summary of the sort implemented by Satish et al. Though the merge sort they created was quite impressive and revolutionary, we must highlight some disadvantages to their approach which our method attempts to address.

Block Sort Disadvantages.

In Satish et al.'s implementation, each thread handles one element, performing an odd-even merge sort to calculate the correct index at any given stage. Though this results in a great deal of parallelism, unfortunately it also results in limited register work (all values are stored in shared memory). In this case, parallelism is being distributed at too fine-grained a level. Each thread is responsible for only one element, and each element must perform a binary search.

The block sort they implement is also unstable. A stable sort requires all elements which have equal keys to preserve ordering (i.e., if $a_i = a_j$ and $i < j$, a_i will appear before a_j after the sort is complete). Due to its predictability, stability is often a desired trait for algorithms that use sorts.

Over-Segmentation.

Each sorted sequence is divided into small chunks (256 elements in their case) and each thread handles only one element; this again ensures a high level of parallelism. However, this also leads to over-segmentation. The number of chunks being processed is a factor of the size of the input, not of the machine. We prefer instead to have p chunks, where p gives the machine enough parallelism to saturate the machine. Given the limited shared memory in a machine, achieving this requires extra bookkeeping and software challenges.

Limited Register Work.

Within the merge stage, each element locates its new index through a binary search. This means there are n binary searches needed at *every* merge stage. This requires $n \log_2 t$ queries into shared memory at each stage. Since each thread is only handling one element, the amount of storage needed in registers (ones value and key) is minimal, and most of the needed information to progress resides in shared or global memory. This leads to kernels that are bounded

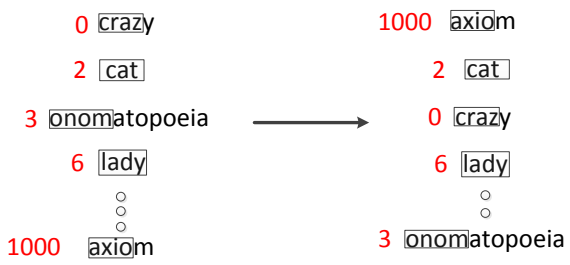


Figure 6: *Basic setup of our string sort representation. We can store the first four characters within an unsigned integer as the Key (boxed values), and use the starting character address as the Value (red values). Then we can perform our sort as described in the previous section by modifying the compare operators to handle ties in our Keys.*

heavily by shared memory usage.

Load Imbalance.

In a similar vein as our over-segmentation argument, if we limit the maximum size of a chunk to be some standard size t, if t is small there can be cases where the associative chunk it must merge with is either much larger than t (requiring a re-partitioning to meet the minimum chunk size criterion), or so small the merge stage is trivial. This causes both load-imbalance and extra repartitioning work that we want to avoid. Satish et al. attempt to mitigate this by using splitters to help normalize chunk sizes. However, since we are able to handle larger sequences, we suffer much less from general load-balancing problems.

4. VARIABLE LENGTH KEY SORTS

Now that we have an operational stable key-value merge sort, we can start building a string sort. Our strings in this case are of arbitrary size, with a null-terminating character. We build our sort by building off of our key-value pair sort. We begin by storing the start of each string (up to the first four characters) as the key in our key-value sort, and the address of each strings origin as the value. Now when performing our merge sort, if two of our shared memory keys are different, we can perform a compare and swap just as we would with our key-value sort. However, when a tie occurs in our keys, we must use the addresses (stored in values) to go out into global memory and resolve the tie. This allows us to break ties on the fly, rather than performing a key-value sort and then search for and process consecutive sequences of ties as a post-processing step.

We must also be able to process ties when deciding how to move our shared memory and register windows. Though this now allows us to sort variable-length strings, dealing with this type of data leads to three performance issues. First, if a thread runs across a tie that must be broken, all threads within this warp must wait for this tie to resolve (divergence issues). Also, as we merge larger partitions, the ratio of our memory windows to the size of a partition becomes quite small. Therefore, the probability that an element will have to resolve a tie becomes higher as we get into later merge steps. Finally, since the variance in our memory windows decreases as our partition sizes increase, long sets of equivalent keys (chains of ties) become more probable, making a worst-case linear search more likely.

We will discuss these drawbacks, and possible ways to mitigate some of them in Section 6. First, though, we will present the results of our fixed-length sorts and variable-length sorts.

5. RESULTS

We will now demonstrate our experimental performance on three variations of our merge sort: 32-bit key-only sort, key-value sort, and variable-length string sort. In our test cases for fixed-length keys, our data consists of uniformly distributed integers. We test our string sort on two types of datasets. First we have a dataset gathered from Wikipedia including over a million words (12 MB) [25]. Our second dataset is a list of sentences, gathered from around 20 books and novels from Project Gutenberg [10]. Sentences from novels have a number of different attributes which affect performance. First, each sentence is much longer than the Wikipedia word list. Second, authors often begin sentences in the same way (e.g., Then, And, etc), which will lead to a larger number of at *least* one tie-break. Finally, since authors often use repetition in their literature (e.g., poetry) there will also be cases when two strings will have a long series of the same characters. Using both of these sets we can try to quantify how much of an effect these characteristics have on performance. We also chose both datasets due to their accessibility, range in data, and authenticity (not synthetic). A histogram showing the data distribution of both dataset can be seen in Figure 9a.

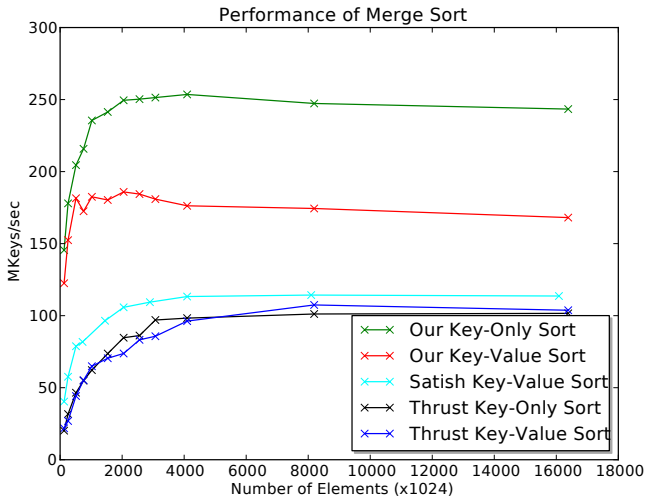
We report the sorting rates for our merge sorts and string sorts as a function of size. Since our strings are variable length, we also report the throughput in MB/s of sorted string data. We do not include transfer time as sorting is often a single step in a much more involved algorithm. Therefore we expect users to re-use the sorted data on the GPU. Since our string sort is a specialization of our key-value sort, we report the ratio in performance between the two for both datasets (lower is better). This gives us an idea of the extra cost of load-imbalance and divergence caused by global memory tie-breaking for each dataset.

We compare our key-only and key-value sorts with the Thrust library comparison sort [4]. Thrust bases its comparison sort on a variant of Satish et al.'s merge sort. In order to make the sort stable, the library uses a different block level sort, and is coded for generality. Though this reduces the performance of Thrust's merge sort, it is still widely used due to its accessibility and ease of use. It is therefore the de-facto standard for comparing GPU based sorts [5, 12, 16, 26].

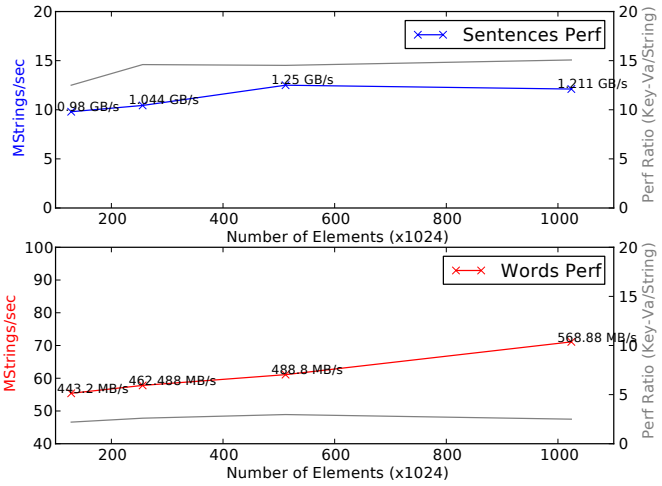
We also compare our sort with a non-stable optimized key-value version of Satish et al.'s sort sort. Though this sort uses the same merge techniques, it utilizes a different (but faster) block sort that isn't stable [2].

In comparison to the Thrust sort, we are about 3x faster than their key-only implementation and about 2x faster than their key-value implementation. Figure 7a shows that Thrust's merge sort performance for key-only and key-value are nearly identical. We believe this is due to their key-only sort implementation being a variant of key-value (with the values being a duplicate of the keys). We are also about 70% faster than the non-stable optimized key-value sort.

The performance for our string sort can be seen in Figure 7b. Since our string sort is a variation on our key-value sort, we compare the performance ratio between the two in terms of key-value pairs vs. strings sorted. This gives us an idea of the performance impact when dealing with global-memory ties (on average a 2.5x performance impact for words, and 14-15x worse for sentences). We also report the rate of our string sort performance in terms of MB/s for both datasets. In Section 6 we will analyze the causes of



(a) *Fixed-Key Sort Performance*



(b) *String Sort Performance*

Figure 7: *Performance in MKeys/sec of our merge sort for string, key and key-value sorts on a GTX 580. We also compare the performance ratio between our key-value sort and string sort. Though our string-sort has a slower stings/sec sorting rate for our sentences database, since each string is much longer the overall MB/s sorting rate is higher.*

this performance degradation, and discuss future optimizations.

Though merge sort is an $n \log n$ method, we require a certain number of elements to reach peak performance. We expect our sorting rate to degrade due to the asymptotic $O(\frac{1}{\log n})$ factor, which begins to take effect after about 4 million elements. The thrust sort pays a much higher initial overhead, leading to a flatter degradation over time.

We test our implementation on a GTX 580 which has up to 1.58 TFlop/s and a theoretical bandwidth of 192.4 GB/s. Since sorts are traditionally not computationally intensive, the theoretical bandwidth bounds our performance. Our next section will analyze our performance, give a rough estimate and compare our performance to the theoretical bounds of a merge sort algorithm, and discuss factors limiting our performance. We will also discuss the performance and limiting factors of our string sort implementation, as well as future work that may improve performance.

6. PERFORMANCE ANALYSIS

In this section we will analyze our performance and determine (1) how well our implementation compares to a theoretical bound of our key and key-value sorts; (2) where our time is going; and (3) where efforts for future improvements lie.

Theoretical Upper Bound.

First we will derive a loose upper bound on the possible sorting rate for a key-only and key-value merge sort (blocks of size p) using our strategy. We do this to show that our method is reasonably efficient when compared to this bound, and provide a comparison for future merge sort implementations. We will use as a limiting factor our global memory bandwidth B. If we assume that for our blocksort stage and each merge stage we must read elements in at least twice (once for register window, and once for shared memory window) and write out elements at least once, we have $2 \cdot (1 + \log(\frac{n}{p}))$ global memory accesses, for a key-only sort.

As an example, if we assume our blocksize p is 1024, and we are sorting four million elements, we will have a blocksort stage and twelve merge stages, each requiring at minimum of two global reads (each element is placed once in registers to search, and once in a shared memory search space) and one write (totaling $38n$). Under these conditions (four million elements) our theoretical sorting rate cap is at 1.26 GKeys/s, which is about 5x faster than what we are able to achieve. Similarly, we can show that our cap for key-value pairs is 941 Mpairs/s, which is also about 5x faster than our achieved rate.

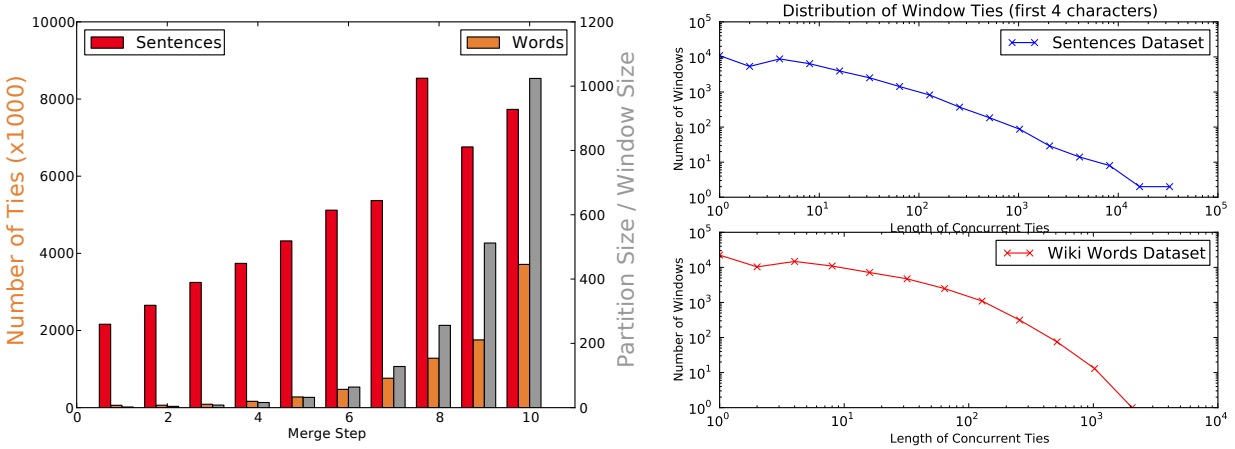
We can attempt to show a tighter theoretical bound by including the minimum shared memory communication required at every stage. Under our conditions:

- Each thread is responsible for k elements;
- Each thread performs a binary search into a shared memory window of size p;
- For $k - 1$ stages we perform a linear search; and
- The sum of all search spaces loaded into shared memory is at least n.

Therefore we can get a lower bound on the minimum shared memory communication needed by calculating the lower bound per thread. Each thread requires $\log(p) + (k - 1)$ shared memory reads to search, and all threads combined will load the entire input set. Since there are again $\log(\frac{n}{p})$ merge stages, the amount of shared elements loads necessary are at least $n \log(\frac{n}{p})(1 + \frac{(log(p)+k-1)}{k}))$.

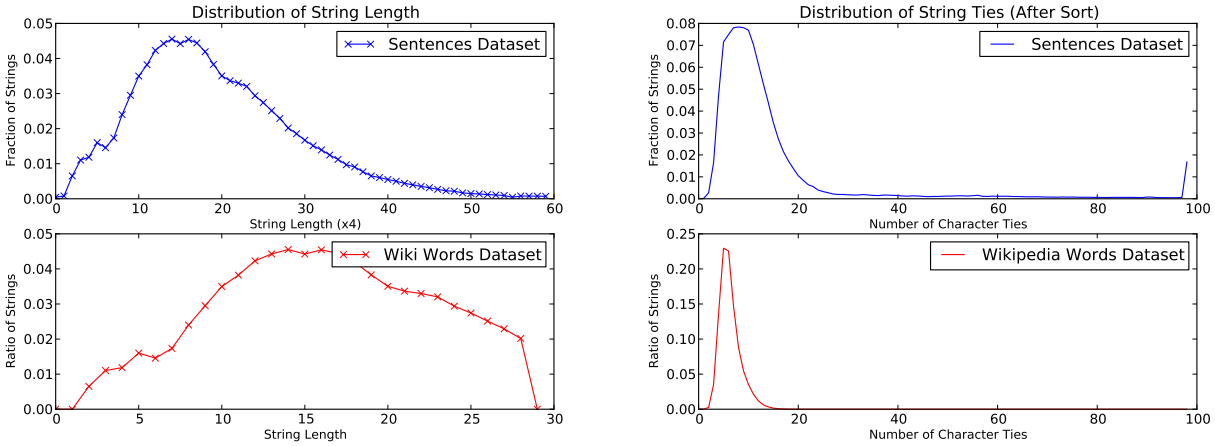
Since the theoretical maximum bandwidth of shared memory is about 1.2 TB/s, we can plug in the same p, n, and choosing k as four we add an extra 0.885 ms to sort four million elements on an NVIDIA GTX 580. This reduces the theoretical sorting rate to 997.4 MKeys/s for key-only sort and 785.45 MPairs/s for key-value sort. Therefore our sort performance is about 4x and 4.2x away from the theoretical bound for key-only and key-value pairs respectively.

Though it is unlikely for an algorithm that requires both synchronization and moving memory windows to be close to the global



(a) *Ties Per Merge Step* **(b)** *Post-Sort Window Ties*

Figure 8: *Figure 8a shows the total number of global memory accesses needed to resolve ties per merge stage for a million strings for both of our datasets. Figure 8b measures the number of shared memory windows with duplicate keys, and the length of these windows after our sort is complete. As the size of our partition grows (while our window size remains fixed), the variance between the largest and smallest key in our window shrinks. This leads to ties becoming more probable, forcing global memory reads to resolve the ties, and degrading performance. For our dataset involving book sentences, this variance is even smaller leading to more global memory lookups and a lower Strings/sec sorting rate. We also test the number of key ties in a row once the sort is finished, and report the number of ties. Since our shared window size is relatively small (we select as a heuristic 1024 elements), performing a binary or linear search within long blocks with the same key will be relatively useless, and require a large number of global memory tie breaks.*



(a) *String Length Distribution* **(b)** *Sorted Ties Distribution*

Figure 9: *Statistics regarding our two string datasets. All strings are stored concurrently with null-termination signifying the end of a string and the beginning of a new string. Our* Words *dataset has on average strings of length 8 characters long, while our* Sentences *dataset has strings on average 98 characters long. As strings from our sentences are much longer on average, they will run into more lengthy tie-break scenarios as we perform our merge-sort. Our sentences dataset, has many ties ranging from 10–20 characters long, and quite a number that are even greater than 100 (we clipped the range).*

memory theoretical cap, this bound gives us an idea how much time is being spent on the actual sorting itself (instead of global memory data movement). We now must analyze what else may be causing this performance difference. From this analysis, we may be able to learn where we should focus our efforts to optimize merge sort in future work. Next we will discuss factors that effect our sorting performance.

6.1 Fixed-Length Keys Performance

The two factors that have the largest effect on our merge sort performance are divergence and shared memory bank conflicts. Though we can streamline our windowed loads to be free of divergence or bank conflicts, it is difficult to do so for both the binary search stage and linear search stage.

Bank Conflicts.

Divergence occurs most frequently in our binary search stage. To illustrate, consider a SIMD warp with each thread performing a binary search in a shared memory window. Each thread will query the middle value; this results in an efficient broadcast read. However, given two evenly distributed sequences, in the next query half of the threads will search to the left and the other half will search to the right. This will create a number of 2-way bank conflicts. Similarly, subsequent searches will double the number of possible shared memory bank conflicts. We attempted to resolve this problem by implementing an address translator that would avoid bank conflicts. However, this modification did not improve performance.

Divergence.

Though our linear search stage does not suffer heavily from bank conflicts, it does suffer from divergence problems. When each thread performs its linear search, it begins independently in a shared memory location. It will then incrementally check the subsequent values until it finds the first value larger than its key. Since a SIMD warp must operate in lockstep, a thread cannot progress past this linear search stage until all threads are finished. Therefore a warp will have to wait for the thread which has the largest *gap* to search through.

6.2 Variable-Length Keys Performance

Since our string sort is built on top of our fixed-length key merge sort, it suffers from the same divergence and bank conflicts mentioned previously. However, it is also affected by other causes of divergence as well as load imbalance.

Divergence.

Since warps operate in lockstep, any threads that break ties will stall all other threads within a warp. Since these divergent forks must be resolved in high-latency global memory, they are inherently expensive for SIMD machines. This isn't an issue that can easily be rectified either. However, since we keep our original string locations static and in-order, if there are multiple ties when comparing two strings, the majority of consecutive tie-breaks should be cached reads. Although this doesn't directly address the divergence cost, it helps to mitigate the effect.

When comparing our two data sets it becomes apparent that the average number of global memory ties that must be resolved begins to dominate performance. Not only are our sentences much longer on average than our words, but require much more work to resolve ties. Figure 9b compares the number of shared concurrent characters between two concurrent strings *after being sorted* between our two datasets. After every step of our merge sort, comparisons will be between *more* similar strings(as illustrated in Figure 9b and Figure 8a), this gives us an idea of how many worst case comparisons will be required.

For authors, it is very common to begin sentences in similar ways (e.g., Then, And, But, etc.), which results in many string ties of about 10–20 of the same characters in a row. In Figure 9b we even see a set of very similar strings greater than 100 characters long (we capped our histogram). Since all threads in a warp must wait for a tie to resolve before continuing, such occurrences are very costly.

We could expect a database of names and addresses to have somewhat similar behavior, where ties among sets of common names must be broken. On the other hand, our Wikipedia word list dataset has much fewer ties and none that exceed 20 characters in a row. As we can see from Figure 7b, our sentences dataset is over 5x slower (lower MStrings/sec) sorting rate than our words dataset. However since each sentence is much longer (about 10x), we achieve a higher GB/s sorting rate with sentences.

Long Sets of Similar Strings.

As sequences become very large in comparison to the memory windows we can handle, the distribution of the values (variance) decreases. Since our shared memory and register sizes are limited, we cannot directly scale our windows to keep pace. Therefore, some threads in our linear search stage are more likely to run into long sets of ties before calculating their correct indexes, while others resolve their location immediately. Figure 8a illustrates this effect. As we begin to merge larger and larger blocks, the number of total ties within a merge step grows. Figure 8b shows the number of keys that share the same value after our string is sorted. This effect is a data-dependent load imbalance. Though it was more efficient in our uniform-length key sort to perform linear searches in every merge stage (after an initial binary search) as described in Section 3 this change in distribution makes the worst-case for linear searches more probable. Therefore, we limit the number of linear searches, and have threads perform more binary searches (worse average case, but better worst case) when locating their insertion indexes. When comparing our two datasets, the effect is much more pronounced in our sentences database (again since authors have common ways of beginning sentences).

We could also attempt to mitigate the amount of variance within a window using the following strategy: Since each thread knows the largest and smallest value in a memory window, a simple *and* operation can determine the set of most significant bits shared by all values within that window. Then a block can decide whether it is worth it to shift those shared bits out and load new bits to increase the variance within that block. We think this can help reduce the number of ties, and we plan to implement it in future work.

7. CONCLUSION

We have presented an efficient hierarchical approach to merge sort. We harness more register communication, handle arbitrarily large partitions, create just enough work to fill the machine and limit unnecessary binary searches.

Our merge sort attains best-of-class performance through four main techniques: (1) in our initial step, sorting 8 elements within each thread, which leverages register bandwidth; (2) a novel binary-then-linear searching approach in our merge within a thread block; (3) avoiding over-segmentation with a moving shared memory and register windows; and (4) a modular, three-step formulation of merge sort that is well-suited to the GPU computational and memory hi-

erarchy (and possibly suitable for tuning to other parallel architectures).

From this merge sort we are able to specialize a string sort, which we believe is the first general string sort on GPUs. The performance of the string sort highlights the large cost of thread divergence when comparisons between strings must break ties with non-coalesced, long-latency global memory reads. We see this as the most critical issue for optimizing string sort on future GPU-like architectures.

There are a number of possible directions we would like to take future work in both comparison sorts, and string sorts in general. We have focused on implementing an efficient merge sort. However, we would like to explore comparison-based techniques for handling very large sequences across multiple GPUs. For example, hybrid techniques that combine merge sort with sample sort appear promising for handling hundreds of gigabytes worth of key-value pairs.

We would also like to develop methods for avoiding thread divergence and global memory tie breaks in our current string sort, and explore hybrid string sorting techniques that might combine radix-sorts with comparison sorts (such as our merge sort).

Acknowledgments

We appreciate the support of the National Science Foundation (grants OCI-1032859 and CCF-1017399). We would like to thank Anjul Patney for some assistance in creating figures, and Ritesh Patel for testing our string sort and reporting bugs. We would also like to thank the reviewers for their valuable comments and feedback.

8. REFERENCES

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[2] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, Apr. 1968.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, pages 18:1–18:11, Nov. 2009.

[4] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W. W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 4, pages 359–372. Morgan Kaufmann, Oct. 2011.

[5] D. Cederman and P. Tsigas. GPU-Quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14:4:1.4–4:1.24, Jan. 2010.

[6] A. Davidson and J. D. Owens. Register packing for cyclic reduction: A case study. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 4:1–4:6, Mar. 2011.

[7] A. Davidson, Y. Zhang, and J. D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, pages 956–965, May 2011.

[8] F. Dehne and H. Zaboli. Deterministic sample sort for GPUs. *CoRR*, 2010. `http://arxiv.org/abs/1002.4464`.

[9] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 325–336, June 2006.

[10] P. Gutenberg. Free ebooks by project guttenberg, 2010. `http://www.gutenberg.org/`.

[11] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[12] N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing*, Apr. 2010.

[13] D. Merrill and A. Grimshaw. Revisiting sorting for GPGPU stream architectures. Technical Report CS2010-03, Department of Computer Science, University of Virginia, Feb. 2010.

[14] G. Morton. *A Computer Oriented Geodetic Data Base and A New Technique In File Sequencing*. International Business Machines Co., 1966.

[15] R. A. Patel, Y. Zhang, J. Mak, and J. D. Owens. Parallel lossless data compression on the GPU. In *Proceedings of Innovative Parallel Computing (InPar '12)*, May 2012.

[16] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. Fast in-place, comparison-based sorting with CUDA: a study with bitonic sort. *Concurrency and Computation: Practice and Experience*, 23(7):681–693, 2011.

[17] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003*, pages 41–50, July 2003.

[18] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[19] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106, Aug. 2007.

[20] R. Sinha, J. Zobel, and D. Ring. Cache-efficient string sorting using copying. *Journal of Experimental Algorithmics*, 11, Feb. 2007.

[21] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.

[22] V. Volkov and J. Demmel. LU, QR, and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, Electrical Engineering and Computer Sciences, University of California at Berkeley, 13 May 2008.

[23] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 31:1–31:11, Nov. 2008.

[24] V. Volkov and J. W. Demmel. Using GPUs to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices. LAPACK Working Note 197, Department of Computer Science, University of Tennessee, Knoxville, Jan. 2008.

[25] Wikipedia. Wikimedia downloads, 2010. `http://dumps.wikimedia.org`.

[26] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne. High performance comparison-based sorting algorithm on many-core GPUs. *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –10, Apr.

2010.