# Scalable Split and Gather Primitives for the GPU

Suryakant Patidar
skp@research.iiit.ac.in
CVIT, IIIT Hyderabad

P. J. Narayanan
pjn@iiit.ac.in
CVIT, IIIT Hyderabad

## Abstract

We present efficient implementations of two primitives for data mapping and distribution on the massively multithreaded architecture of the GPUs in this paper. The *split* primitive distributes elements of a list according to their category. Split is an important operation for data mapping and is used to build data structures, distribute work load, etc., in a massively parallel environment. The *gather/scatter* primitive performs fast, distributed data movement. Efficient data movement is critical to high performance on the GPUs as suboptimal memory accesses can pay heavy penalties. The split we implement is a generalization of the binary split [Blelloch 1990] and is implemented using the shared memory and the atomic operations available on them. The split performance scales logarithmically with the number of categories, linearly with the list length, and linearly with the number of cores on the GPU. This makes it useful for applications that deal with large data sets. We also present a variant of split that partitions the indexes of records. This facilitates the use of the GPU as a coprocessor for split or sort, with the actual data movement handled separately. We can compute the split indexes for a list of 32 million records in 180 milliseconds for a 32-bit key and in 800 ms for a 96-bit key. The instantaneous locality of memory references play a critical role in data movement on the current GPU memory architectures. For scatter and gather involving large records, we use collective data movement in which multiple threads cooperate on individual records to improve the instantaneous locality. The split, gather, and their combinations find many applications and expect our primitives will be used by future GPU programmers. We show sorting of 16 million 128-byte records in 379 milliseconds with 4-byte keys and in 556 ms with 8-byte keys.

## 1 Introduction

The Graphics Processor Unit (GPU) has been increasingly used for a wide range of problems involving heavy computations in graphics, computer vision, science, etc. The main attraction is the high computation power per unit cost; today's off-the-shelf GPUs deliver 1 TFLOPs of single precision power for under $400. The programming model available on them have also become general purpose with the advent of CUDA [Nvidia 2008] and the newly-adopted standard of OpenCL [Khronos 2009]. However, extracting the best performance from the GPU requires a deep knowledge of its internal architecture including the core layout, memory, scheduling, etc. A lot of this information is not publicly available; the architecture also changes much more frequently than the architecture of multicore microprocessors.

One way to exploit the GPU's computing power effectively is through high level primitives upon which other computations are built. All architecture specific optimizations can be incorporated into the primitives by designing and implementing them carefully. An application can gain high performance if its computationally intensive parts can be broken down into such primitives. The map-reduce primitive has recently been very effective in distributing compute intensive applications to a cluster of processors [Dean and Ghemawat 2004]. GPUs essentially follow the data parallel model in which kernels of code are applied on many data elements in parallel. Blelloch defined several data parallel primitives including scan, reduce, and binary split [Blelloch 1989; Blelloch 1990]. Sengupta et al. implemented these primitives on the GPU under CUDA [Sengupta et al. 2007] which has been made available under the CUDPP libarary [Harris et al. 2007]. Several applications have been built using these primitives including kd-trees [Zhou et al. 2008], octrees [Zhou et al. April, 2008], BVH trees [Lauterbach et al. 2009], etc.

In this paper, we present efficient and scalable implementations of two data mapping primitives with wide applications on the GPU. Data mapping and distribution are used in distributed applications for data structure building, load balancing, etc. We present the *split* primitive that distributes data elements based on a category each belongs to. This is a generalization of the binary split and has been found to be critical for all throughput computing [Siggraph Asia Courses 2008]. We also define index-variants of split that defer the actual data movement, which is useful to handle bulky data records. We also present efficient implementations of the *scatter* and *gather* primitives on the GPU that work in conjunction with the index-variants of split. The separation of index computation from data movement enables the use of the GPU as a fast co-processor for split and sort, with data movement handled separately. The GPU performance is highly sensitive to memory access patterns; suboptimal implementation can pay heavy penalty. Optimized primitives are basic building blocks using which many regular and irregular applications can be built.

The main contributions of this paper are the following:

1. We present efficient and scalable implementations of the split and split-index primitives for the GPU. These can be used to split a list to its category or to sort a list of numbers. We can sort 128 million 32-bit numbers in about 650 milliseconds and 128 million 128-bit numbers in about 4 seconds. We can compute the split-index for 64 million records with 32-bit keys in 350 milliseconds.

2. We improve and exploit the instantaneous locality of memory references to improve the data movement performance on the GPU. The coherence in memory access between different compute elements is critical to memory performance on the GPUs, like caching on the CPUs.

3. We present efficient implementaiton of the gather and scatter primitives for fast data movement within the GPU, taking advantage of the instantaneous locality of memory references. We can randomly scatter 16 million 128-byte records in about 290 milliseconds and 8 million 256-byte records in under 150 ms.

4. We show applications of the split and gather primitives for several operations. Sorting large records maps to a split-index followed by a gather. We show sorting of 16 million 128-byte records in 379 milliseconds with 4-byte keys and in 556 ms with 8-byte keys. We also discuss how these primitives can be used to build distributed data structures for applications like ray tracing and surface reconstruction.

## 2 Related Work

Split has been implemented recently on the GPU by He et al. [He et al. 2008]. They built a per-thread histogram on the GPU to overcome the problem of concurrent writes by multiple threads. They used split as a primitive for implementing a variety of relational join operations for databases on the GPU. Their approach is limited by the available shared memory and limits the number of bins per pass to as low as 64 on current GPUs. The CUDPP library by Harris et al. [Harris et al. 2007] implements the split primitive for two categories. A list of elements tagged with *true* or *false* are split into the two categories. They define and implement *compact* as an operation which reduces the above kind of list to a smaller list of elements which are tagged true. CUDPP sort is based on this and splits the data 1 bit (2 categories) at a time. Lauterbach et al. [Lauterbach et al. 2009] use multiple independent splits similar to CUDPP split, for building a bounding volume hierarchy for ray tracing.

Several sorting algorithms have been developed for the GPUs. Bitonic sort, a parallel algorithm for sorting was first implemented by Purcell et al. [Purcell et al. 2005]. Govindaraju et al. [Govindaraju et al. 2006] demonstrated improved sorting performance for external sorting algorithm using graphics processors on large databases. Their implementation of bitonic sort used programmable pixel shaders with OpenGL API. With the introduction of C like interface (Nvidia CUDA) for GPGPU work, sorting algorithms like radix sort and merge sort have been implemented which tend to be faster than GPUSort due to absence of graphics API overhead. Harris et al. [Harris et al. 2007] propose a bit-wise radix sort approach using CUDA. They partition the data based on a bit starting from least significant bit and moving towards most significant bit. Satish et al. [Satish et al. 2009] propose a method for bit-wise sorting of data similar to CUDPP. They implement parallel radix sort by improving the number of bits handled per pass by CUDPP approach. They also describe a compare based merge sort algorithm. Cederman et al. [Cederman and Tsigas 2008] describe a quick sort based approach for sorting large data on the GPU. Katz et al. [Katz 2008] implement compare-based sort using bitonic/merge approach. An earlier unpublished report from our group implemented split and sort using ordered atomic operations on the GPU. An anonymous version of the report is attached as additional material.

## 3 Split and Split-Index Primitives

Split is a popular operation for database retrieval, data structure building, etc. Split can be defined as appending each input record $x$ to a list of the category (or bin) it belongs to, or performing $\mathbf{append}(x, List[\mathbf{category}(x)])$. *List[i]* holds all records of category $i$. Split is a function that partitions an input relation into a number of categories. The length-preserving split appends together the individual lists into a single list *List* in the order of the category number. A record could also be part of multiple categories, resulting in non-disjoint partitions and an increased size of the output relation. Such multi-splits can be handled by placing records in multiple lists. We consider the problem of splitting an $N$-element list to $M$ categories or bins. The basic split algorithm is given below:

Steps 1 and 3 can have clashes when records are processed in parallel. Data parallel computing models (such as the GPU's) examine multiple records simultaneously using multiple threads and can clash while counting (Step 1) and while computing own index (Step 3). Efficient primitives for prefix sum (Step 2) avoid clashes [Blelloch 1990; Sengupta et al. 2007]. The global memory atomic operations available on the GPU can be used to keep the counts in the global memory. This needs $O(N)$ locations for the list and $O(M)$ for the counts in the global memory. This, however, incurs severe performance penalty due to the clashes as well as the slower access times of the global memory. He et al. use a private copy of the count for each thread in the shared memory [He et al. 2008]. The memory requirements is $O(BTM)$ in the global memory and $O(TM)$ in the shared memory, where $B$ is the number of blocks used and $T$ is the number of threads per block. The shared memory requirement limit the number of categories they can handle.

**Algorithm 1** Split Operation

1: Read the records and compute the counts for each category.
2: Compute the prefix sum of the counts to get the starting index of each category in the output list.
3: Read each record, compute its index within its category and write the record to the output using the index.

### 3.1 Split Using Shared Memory Atomics

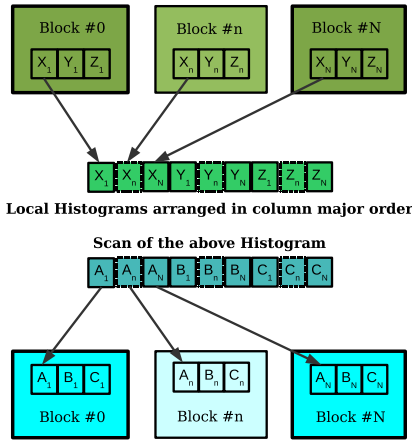



**Figure 1:** *Each CUDA block processes a segment of input records. The counts for each category is in the shared memory and is written back in column-major order to global memory. A prefix sum of it is loaded back to each block as the starting index of each category for its share.*

The later generation of GPUs support atomic operations on the shared memory which are very fast. Our split implementation uses them for counting and index computations. We can also split to many more categories as we store only a single copy of the count per CUDA block on the GPU. Each block processes a segment of $K$ input records using its threads. The threads work in parallel and each handles several records sequentially. The counts of each block are written to the global memory in a column-major order into an array of size $BM$ as done by He et al. [He et al. 2008]. The prefix sum is computed over them in Step 2. The prefix sums are read into the block in column-major order (Figure 1). Each thread goes over its segment, computes its index using shared memory atomic operations and writes it to the final location. Global memory required by the algorithm is $O(BM)$ for working and an $O(N)$ to keep the output. In place split is not possible due to the unpredictable scatter in writing (Step 3).

The writing step involves a general scatter of the records. The writing step takes about 90% of the total time if implemented as de-

scribed above. Writing to widely separated global memory locations is very inefficient on the GPUs. The GPU performs coalesced memory operations well. Coalescing is a dual concept of caching on uniprocessors. Caching improves the performance in the presence of temporal locality in memory accesses by the same thread. Coalescing improves the performance when there is *instantaneous locality* in the memory references by a block of consecutive threads, as the accesses are combined into a minimum number of expensive memory transactions. Completely coalesced reads can be a factor 50-100 times faster than a totally random read on current GPUs.

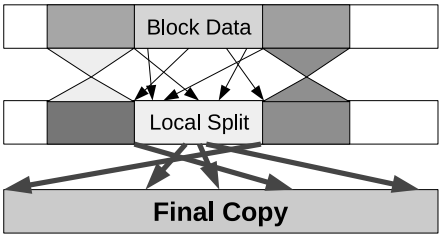



**Figure 2:** *The records of a CUDA block are first locally split, followed by a copy to the final location. The instantaneous locality is better for local split than the global split. The final copy has data moving in groups and has high instantaneous locality.*

The range of the destination index of each record is the length $N$ of the list in general. The expected value of the instantaneous locality is clearly inversely proportional to the range. We improve the writing speed by performing a two-step scatter operation as given in Algorithm 2. Step 3 splits each record within the segment of records handled by the corresponding CUDA block (Figure 2). For this, the local index of each record within its segment is calculated by each thread and the record is written to a temporary list in the global memory at that index. This is a scatter with a range of $K$. The range is shorter as $K$ is much smaller than $N$, resulting in a slightly better instantaneous locality than global scatter. In Step 4, threads of a CUDA block read the records from this temporary list, calculates their final indexes, and copies the data to the final location. On the average, $\frac{K}{M}$ records map to the same category. They will be in consecutive positions in the temporary list as well as in the final list (Figure 2). High degree of instantaneous locality is ensured in Step 4 if consecutive threads handle consecutive records of the temporary list, if $K$ is significantly smaller than $M$. Our split algorithm is given below.

**Algorithm 2** BasicSplit

1: Load the elements of the segment sequentially in each thread. Compute the count for each category per CUDA block using hardware atomic operations on the shared memory.
   – Store them in column major order in blockCount
   – Scan of the count in shared memory and store in localScan
2: Scan the blockCount array, giving the starting index of each bin for each block in globalScan
3: Split the segment locally using ordered atomics and store in localSplit.
4: Scatter localSplit to full range of output array by computing the global scatter index using globalScan and localScan

For $N = 16$ million, $M = 256$ and $K = 8192$, the 2-step scatter takes 14 milliseconds, with Step 3 taking 12 ms and Step 4 the rest. The single step scatter on the same data takes 24 milliseconds. The significant speed up is due to the improved instantaneous locality in the local split operation and the high instantaneous locality in the final copy operation.

### 3.2 Scaling in Number of Categories

Algorithm 2 can only split to a small number of bins due to the small amount of shared memory available on today's GPUs. The counts need $4 * M$ bytes of shared memory. A total of 16 KB of shared memory is available per Streaming Multiprocessor today, to be shared by all CUDA blocks that time-share it. As a result, 256 categories (using an 8-bit category valu) is optimal to exploit sufficient parallelism through high occupancy, which relates to the number of threads simultaneously in flight on the multiprocessor. The maximum number of categories is 2048, but that restricts the number of threads in flight to 512 resulting in an underutilizaiton of the GPU (Figure 4).

We split to larger number of bins by handling the category numbers 8-bits at a time iteratively. We split using the right-most 8 bits as the category in the first step, followed by a split using the next 8 bits as the category, etc., until all bits of the category number are used up. This is similar to radix sort using the least-significant digit. The general algorithm to split to $2^m$ categories is given below:

**Algorithm 3** Split ($m$)

1: For $j = 0$ to $\lceil m/8 \rceil$ do
2: Invoke BasicSplit using bits $8j{:}(8j + 7)$ as the category

The above algorithm works correctly only if the BasicSplit algorithm is *stable*, i.e., maintains the original ordering of records that have the same key value. Since the order of the segments in BasicSplit is same as that of the blocks, the original ordering of records with the same key in multiple segments will be preserved automatically. The ordering of records within the same segment (and hence handled in parallel by threads of a CUDA block) depends on the index given to it in Step 3 of Algorithm 2. The index depends on how clashes are resolved in the shared memory atomic operations. As records within a segment are assigned monotonically as the thread IDs within the block, we need the clashes resolved in favour of the lowest numbered clashing thread. Such operations can be called *ordered atomic* operations. The shared memory atomic operation on present GPUs cannot ensure any specific ordering as implemented.

We use a simulated ordered-atomic increment operation to calculate the index of each record in Step 3 of Algorithm 2. A thread-serial realization of the atomic operation is used as given in Appendix A. Correct results are obtained only when a CUDA block has exactly 32 threads and consequently, the simulated ordered atomics are 8-10 times slower than hardware atomics.

### 3.3 Splitting Index Values

Split is often performed on database records that are large in size. Reading and copying of the bulky records can be inefficient and wasteful, especially if split is performed in multiple steps. We can split the *indexes* of the records instead of the records themselves in such situations. The index values are less than the length $N$ of the list. A 32-bit number can store the indexes of a list of 4 billion records, which is sufficient for most problems today. Splitting of the indexes reduces to splitting a new record consisting of the original key value and the 32-bit index value. After the split, the index part of the records will contain the index in the original list for each position. A gather applied to the original list using these indexes will split the input list. The actual data movement may not be needed in many cases as only some records of the split list are needed. For instance, only records of a few select categories may be needed after a split on a database table. Costly data movement can

be avoided by accessing only the required records using the index values.

### 3.4 Split Primitives on the GPU

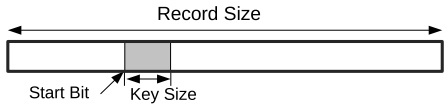



**Figure 3:** *Record, key, and start bit for split primitives*

Split is very useful in distributed data mapping. Efficient split can be a basic building block to many applications and has a role as a fundamental primitive, like scan and reduce [Harris et al. 2007]. We provide efficient implementations of the following split primitives.

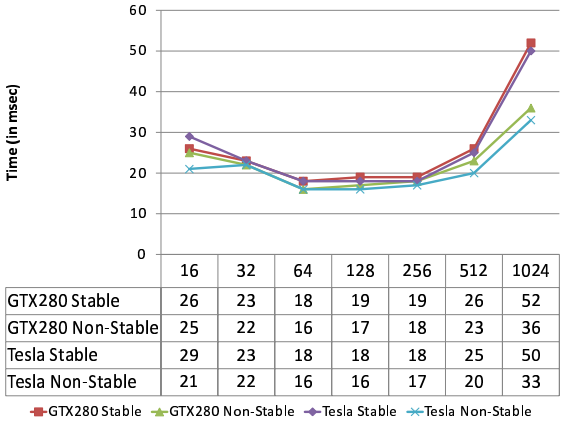
1. **split8(list[], rSz, sBt)**: The basic function to split the *list* of records of size *rSz* bytes to 256 bins starting with bit number *sBt* from the left of the start of the record. The output is returned in the same list.
2. **split8ns(list[], rSz, sBt)**: A non-stable version of split8 that is also slightly faster.
3. **split(list[], rSz, kSz, sBt)**: Split the list of *rSz*-byte records using a *kSz*-bit key starting at bit *sBt* of the record (Figure 3). This primitive uses split8 iteratively.
4. **splitGatherIndex(list[], rSz, kSz, sBt)**: Split the index values instead of the records. The function returns a list *gindex* of index values for subsequent gathering.
5. **splitScatterIndex(list[], rSz, kSz, sBt)**: Similar, but returns a list *sindex* of index values for scattering. That is, sindex[i] gives the index in the split list for the input record list[i].

The GPU is a coprocessor to the CPU which can perform compute intensive operations very fast. GPU is not good at data movement involving irregular patterns; the bandwidth available to move data between the CPU and the GPU is highly limited. The split-index primitives explicitly enable the use of the GPU as a coprocessor that performs the compute intensive part of the split, leaving the data movement to the CPU or another device that is good at that.

### 3.5 Performance of Split

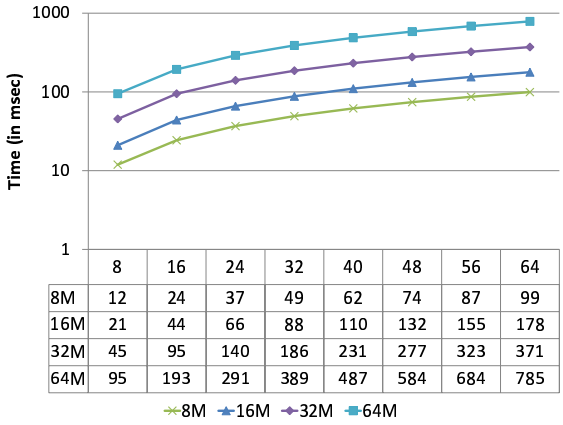
Split can operate on a maximum of 1K bins in a single pass due to shared memory limitations. Figure 4 shows times for a single pass of split for different numbers of bins. Split performs best in the central region, where the number of bins is large enough for minimal atomic clashes and small enough for efficient use of shared memory. We use 256 bins or 8-bits of key size as the basic split operation for maximum efficiency. Figure 5 gives the times for splitting 64-bit records to different key sizes. All timings in this paper are taken on a single GPU of a Tesla S1070 server, unless otherwise indicated. The figure shows that the split time increases linearly with the key size (or logarithmically with the number of categories). We can also see linear increase in split time as the number of records increases. All key values in all our experiments are generated using a system random generator.

Figure 6 gives split results for many combinations of keys and values, with the record size varying from 32 bits to 128 bits. Figure 7 gives the times for splitting indexes for less than 4 billion records. The index is then a 32-bit number which is used as the value with different key sizes. The dependence on the key size can be observed




| | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|
| GTX280 Stable | 26 | 23 | 18 | 19 | 19 | 26 | 52 |
| GTX280 Non-Stable | 25 | 22 | 16 | 17 | 18 | 23 | 36 |
| Tesla Stable | 29 | 23 | 18 | 18 | 18 | 25 | 50 |
| Tesla Non-Stable | 21 | 22 | 16 | 16 | 17 | 20 | 33 |

**Figure 4:** *Timings for the basic split step for 16 million elements for different number of categories/bins.*




| | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 |
|---|---|---|---|---|---|---|---|---|
| 8M | 12 | 24 | 37 | 49 | 62 | 74 | 87 | 99 |
| 16M | 21 | 44 | 66 | 88 | 110 | 132 | 155 | 178 |
| 32M | 45 | 95 | 140 | 186 | 231 | 277 | 323 | 371 |
| 64M | 95 | 193 | 291 | 389 | 487 | 584 | 684 | 785 |

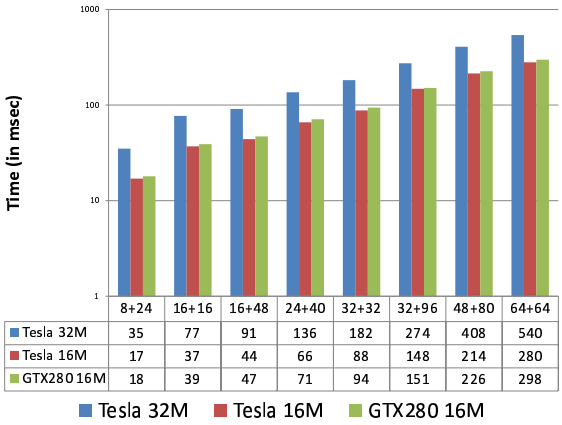
**Figure 5:** *Time to split 64-bit records using key sizes from 8 to 64 bits for lists of lengths from 8 to 64 million. Split is scalable in the key size and list length.*

to be linear when record size is fixed. The dependence on the record size is sublinear in this range as larger records are read using higher access widths.

## 4 Gather and Scatter Primitives

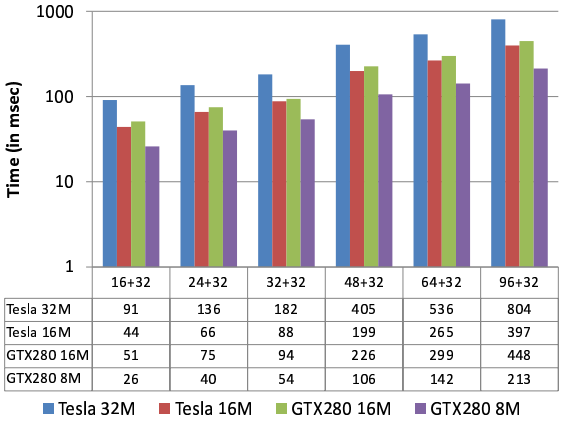
The data movement performance of the GPU depends heavily on the memory access patterns. Optimal accesses can be several folds faster than suboptimal ones. Optimal access patterns require deep understanding of the architecture and may not available to every user. We present two primitives for the common data mapping operations on the GPU, namely, gather and scatter, using an index list.

Gather is a simple operation that can be implemented using:
`outList[threadID]` ← `inList[gindex[threadID]]`,
where *threadID* is a sequence number of each thread. The reading of gindex and the writing of outList are perfectly coalesced with very high instantaneous locality on current GPUs as consecutive threads access consecutive records. The reading of inList follows irregular access pattern and can be very inefficient due to low instantaneous locality. The GPU can handle 4-byte, 8-byte, and 16-byte entities in a single memory access. The above instruction




| | 8+24 | 16+16 | 16+48 | 24+40 | 32+32 | 32+96 | 48+80 | 64+64 |
|---|---|---|---|---|---|---|---|---|
| Tesla 32M | 35 | 77 | 91 | 136 | 182 | 274 | 408 | 540 |
| Tesla 16M | 17 | 37 | 44 | 66 | 88 | 148 | 214 | 280 |
| GTX280 16M | 18 | 39 | 47 | 71 | 94 | 151 | 226 | 298 |

**Figure 6:** *Split timings for different record sizes and key sizes.*




| | 16+32 | 24+32 | 32+32 | 48+32 | 64+32 | 96+32 |
|---|---|---|---|---|---|---|
| Tesla 32M | 91 | 136 | 182 | 405 | 536 | 804 |
| Tesla 16M | 44 | 66 | 88 | 199 | 265 | 397 |
| GTX280 16M | 51 | 75 | 94 | 226 | 299 | 448 |
| GTX280 8M | 26 | 40 | 54 | 106 | 142 | 213 |

**Figure 7:** *Time to split a 32 bit index value for different key sizes.*

completes the gather for these record sizes. Similar observations hold for the scatter operation.

### 4.1 Collective Data Movement

Gather and scatter of large records need to loop over elements of the same record. Since a thread moves a record, the inner loop goes over its elements. This, however, reduces the instantaneous locality of writes of gather by a factor equal to the number of data elements in the record, as the memory accessed by consecutive threads will have gaps between them. The instantaneous locality can be improved by multiple threads copying each record collectively, with consecutive threads reading and writing adjacent data elements. Figure 8 demonstrates the approach.

Current GPUs achieve the highest instantaneous locality if 16 consecutive threads (called a half-warp) access adjacent data elements. Thus, best performance is obtained when maximum number of threads cooperate on a single record, if a thread cannot load a record in a single read. The data access width should be set to 4, 8, or 16 bytes accordingly. For example, 8 threads should cooperate using 4-byte accesses on 32-byte records, 16 threads using 4-byte acess on 64-byte records, and 16 threads using 8-byte access on 128-byte records, etc. The number of threads that cooperate on a record of size *recSz* bytes and the data-access width are:

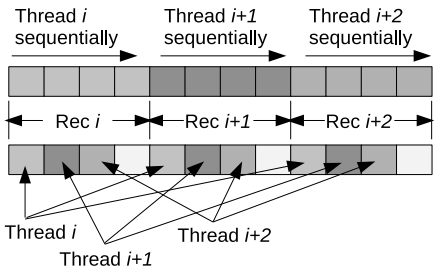



**Figure 8:** *Instantaneous locality is low when each thread copies one record element by element (top). Collective copying of records by multiple threads improves the locality (bottom)*

| | |
|---|---|
| $recSz \leq 64$: | (*recSz* / 4) threads and 4-byte accesses record |
| $recSz \leq 256$: | 16 threads and 4, 8 or 16 byte accesses, depending on (*recSz* / 16) |
| $recSz > 256$: | (*recSz* / 16) threads and 16-byte accesses |

### 4.2 Data Movement Primitives

We provide implementations of the following data movement primitives on the GPU:

1. **gather(list[], rSz, gindex[])**: Returns a list that is a permutation of the input *list* of records of size *rSz* bytes, with the list *gindex* providing the index to gather from.
2. **scatter(list[], rSz, sindex[])**: Similar, with *sindex* providing the index to scatter each record to.

### 4.3 Performance of Gather and Scatter

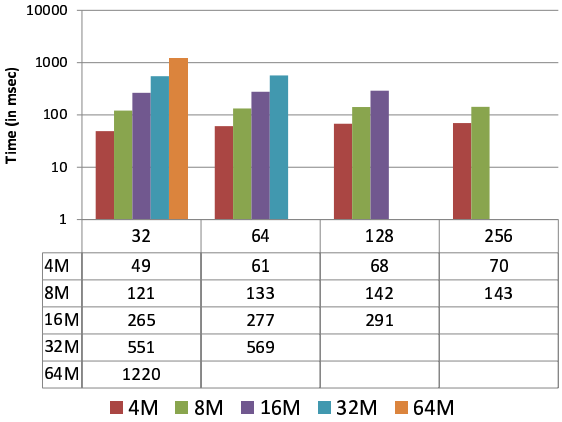
Figure 9 shows the times for gather and scatter for combinations of number of records ranging from 4 to 64 million and record size ranging from 32 to 256 bytes, using the collective data movement scheme described above. The dependence on the number of records can be seen to be linear, especially for larger records. The dependence on the record size is highly sublinear as all memory operations become completely coalesced with high instantaneous locality when moving larger records. From the table, the time to move 16 million 128-byte records is twice the time needed to move 8 million 256-byte records, though the total data moved is 2 gigabytes. This is because a record is collectively moved by 16 threads, each accessing 8-byte elements, in the former case whereas the latter case uses 16 threads and 16-byte accesses.

## 5 Applications of Split and Gather Primitives

The primitives we presented find wide applications in different applications. We give results of a few here.

### 5.1 Sort

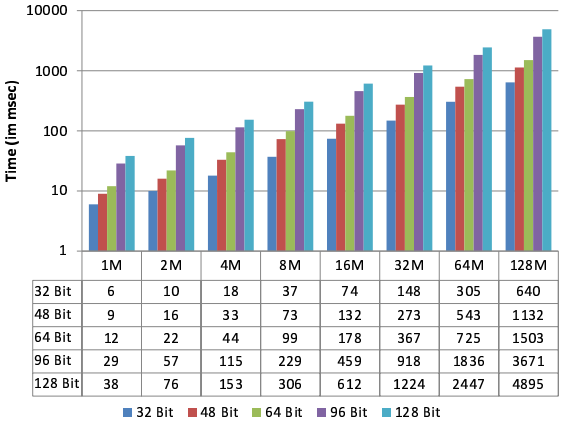
Sorting is a special case of split where the key has ordinal values and is itself interpreted as the category number. Our split scales linearly in the key size by applying the basic 8-bit split procedure iteratively using ordered atomic operations. Our *SplitSort* algorithm performs sorting using repeated splits, which is akin to LSD radix sort applied to a radix of 256. The scalability of the basic split




| | 32 | 64 | 128 | 256 |
|---|---|---|---|---|
| 4M | 49 | 61 | 68 | 70 |
| 8M | 121 | 133 | 142 | 143 |
| 16M | 265 | 277 | 291 | |
| 32M | 551 | 569 | | |
| 64M | 1220 | | | |

**Figure 9:** *Results for random scatter of 4 to 64 million records of sizes 32 bytes to 256 bytes on Tesla S1070.*

makes sorting also highly scalable. Figure 10 gives the sorting performance on one GPU of a Tesla S1070. We can clearly see the linear behaviour in the number of records as well as on the key size.



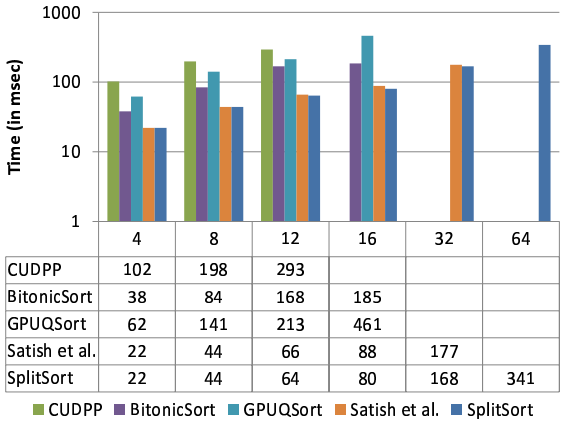

| | 1M | 2M | 4M | 8M | 16M | 32M | 64M | 128M |
|---|---|---|---|---|---|---|---|---|
| 32 Bit | 6 | 10 | 18 | 37 | 74 | 148 | 305 | 640 |
| 48 Bit | 9 | 16 | 33 | 73 | 132 | 273 | 543 | 1132 |
| 64 Bit | 12 | 22 | 44 | 99 | 178 | 367 | 725 | 1503 |
| 96 Bit | 29 | 57 | 115 | 229 | 459 | 918 | 1836 | 3671 |
| 128 Bit | 38 | 76 | 153 | 306 | 612 | 1224 | 2447 | 4895 |

**Figure 10:** *Sorting times for list lengths from 1 to 128 million for 32 to 128 bit numbers.*

Figure 11 compares our SplitSort with other reported approaches on an Nvidia GTX280 GPU for 32-bit keys. We perform about 5-10% better than the method by Satish et al. [Satish et al. 2009][1], the best previous effort. Our is, however, the first work to demonstrate scalable sorts beyond 32-bit numbers. Satish et al. report sorting a record consisting of a 32-bit key and a 32-bit value. They sort 16 million such records in about 110 ms (as seen in the graph). Our algorithm sorts a similar list in about 94 ms and a 32 million list in 195 ms on the same hardware.

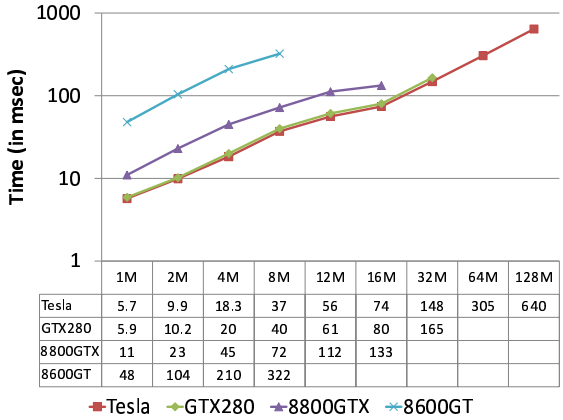
Figure 12 shows another aspect of scalability of our approach. The figure shows sorting times for lists of different sizes for 32-bit numbers. The sorting time scales linearly with the list length as was observed before. The sorting time also grows approximately linearly with the number of cores available on the GPU. The 8800GTX and 8600GT do not support atomic operations on the shared memory. We simulate it by serializing the clashes.

[1]The timing for sorting 32-bit numbers were obtained from the authors as the paper only reported sorting key-value pairs.




| | 4 | 8 | 12 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|
| CUDPP | 102 | 198 | 293 | | | |
| BitonicSort | 38 | 84 | 168 | 185 | | |
| GPUQSort | 62 | 141 | 213 | 461 | | |
| Satish et al. | 22 | 44 | 66 | 88 | 177 | |
| SplitSort | 22 | 44 | 64 | 80 | 168 | 341 |

**Figure 11:** *Comparison of our SplitSort with other reported sorts on a GTX280.*




| | 1M | 2M | 4M | 8M | 12M | 16M | 32M | 64M | 128M |
|---|---|---|---|---|---|---|---|---|---|
| Tesla | 5.7 | 9.9 | 18.3 | 37 | 56 | 74 | 148 | 305 | 640 |
| GTX280 | 5.9 | 10.2 | 20 | 40 | 61 | 80 | 165 | | |
| 8800GTX | 11 | 23 | 45 | 72 | 112 | 133 | | | |
| 8600GT | 48 | 104 | 210 | 322 | | | | | |

**Figure 12:** *Comparison of sort times on different GPUs. A roughly linear performance growth can be seen with increase in cores. The Tesla and the GTX280 have 240 cores each. The 8800GTX has 128 cores and teh 8600GT has 32 cores.*

Sorting 48-bit and 64-bit numbers finds use in data distribution applications, especially, data structure building. The octree built on the GPU by Zhou et al. [Zhou et al. April, 2008] was limited to 9 levels as greater than 32-bit sorting wasn't available. With such scalable sort, the GPUs can be used to sort lists of large records encountered in large databases, etc. The GPU can act as a sorting coprocessor if the *splitGatherIndex* primitive is used for index mapping. The data movement is separated by the use of the *gather/scatter* primitives. Sort of such records can be decomposed into a two-step process as shown below.

**Algorithm 4** SortLargeRecords

1: gindex[] ← splitGatherIndex(list[], rSz, kSz, sBt)
2: outList[] ← gather(list[], rSz, gindex[])

Table 1 gives the time to sort 8 million to 64 million records of size 32 bytes to 256 bytes, for key sizes of 32 to 64 bits. We can sort 16 million 128-byte records in 379 milliseconds with 4-byte keys and in 556 ms with 8-byte keys. We can also sort 64 million 32-byte records in 1490 ms with 4-byte keys and in 3126 ms with 8-byte keys. These cases use up all of the 4 GB available on a single GPU

of the Tesla S1070. Scalability of our approach clearly makes it possible to handle such challenging cases.

| Record Size | List Length | | | | |
|---|---|---|---|---|---|
| | 8M | 16M | 32M | 48M | 64M |
| Key size: **4 bytes** | | | | | |
| 32B | 270 | 352 | 733 | 1102 | 1488 |
| 64B | 182 | 367 | 752 | - | - |
| 128B | 194 | 373 | - | - | - |
| 256B | 196 | - | - | - | - |
| Key size: **6 bytes** | | | | | |
| 32B | 210 | 460 | 955 | 1720 | 2650 |
| 64B | 230 | 473 | 972 | - | - |
| 128B | 244 | 489 | - | - | - |
| 256B | 248 | - | - | - | - |
| Key size: **8 bytes** | | | | | |
| 32B | 251 | 530 | 1080 | 1980 | 3124 |
| 64B | 263 | 540 | 1190 | - | - |
| 128B | 278 | 555 | - | - | - |
| 256B | 281 | - | - | - | - |

**Table 1:** *Sorting large records. Times are shown in milliseconds to sort lists of length 8 to 64 million using key sizes of 4 to 8 bytes.*

### 5.2 Data Structure Building

One approach to handle dynamic and deformable objects for ray tracing is to build an appropriate data structure in each frame. Bounding Volume Hierarchy [Lauterbach et al. 2009], kd-trees [Zhou et al. 2008] and camera-space voxels [Patidar and Narayanan 2008] have been used by different authors. Lauterbach et al. use a series of 1-bit splits to build the hierarcy, which can be speeded up using the general split primitive we presented. Patidar and Narayanan divide the view frustum to $128 \times 128 \times 16$ voxels and map each triangle to multiple voxels using a multilevel split operation. This can be performed using a split after mapping each triangle to a record with an 18-bit key (to address the 256K voxels) and 20 bit value (to address 1 million triangles). The data structure for 16 million voxels can be built in the same time. The octree built by Zhou et al. [Zhou et al. April, 2008] will not be limited to 9 levels if our scalable sorting is used. Applications like particle simulation and N-body problem map naturally to splits to a 2D or 3D grid. A concurrent submission from our group uses these split primitives to implement a fast minimum spanning tree algorithm. It uses splits on lists of length $|V|$ or $|E|$ of 32-bit and 64-bit data to manipulate the graph. An anonymous version of the submission is given as additional material.

## 6 Conclusions and Future Work

We presented a few data mapping primitives and their scalable implementations on the GPU using CUDA in this paper. The split primitive and its variants scale linearly in the number of records, the key size, and the number of available cores. The gather primitive scales linearly with the number of records and sublinearly with the record size due to memory coalescing effects. We demonstrated high performance on these primitives and used them to build applications like sorting large numbers and sorting of large records, such as sorting 64 million 32-byte records using 8-byte keys in about 3 seconds.

We will release optimized implementations of these primitives for general use. These primitives can find a lot of applications in processing irregular data such as graphs on massively multithreaded architectures like the GPU. We outlined how their use can accelerate applications like data structure building for ray tracing, sorting data for fluid simulation, etc.

## References


BLELLOCH, G. 1989. Scan primitives as parallel operations. *IEEE Transactions on Computers 38*, 11, 1526–1538.

BLELLOCH, G. 1990. *Vector Models for Data-Parallel Computing*. MIT Press.

CEDERMAN, D., AND TSIGAS, P. 2008. A practical quicksort algorithm for graphics processors. In *Proceedings of the 16th annual European symposium on Algorithms*.

DEAN, J., AND GHEMAWAT, S. 2004. Mapreduce: Simplified data processing on large clusters. *OSDI '04* (December), 137–150.

GOVINDARAJU, N., GRAY, J., KUMAR, R., AND MANOCHA, D. 2006. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of ACM SIGMOD International Conference on Management of data*.

HARRIS, M., OWENS, J. D., SENGUPTA, S., ZHANG, Y., AND DAVIDSON, A. 2007. Cuda data parallel primitives library. Tech. rep., Nvidia Corporation.

HE, B., YANG, K., FANG, R., LU, M., GOVINDARAJU, N., LUO, Q., AND SANDER, P. 2008. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*.

KATZ, A., 2008. An implementation of bitonic/merge sort. `http://courses.ece.uiuc.edu/ece498/al1/halloffame.html`.

KHRONOS. 2009. Opencl : Open compute library.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast bvh construction on gpus. In *Proceedings of Eurographics*.

NVIDIA. 2008. Nvidia CUDA : Compute Unified Device Architecture.

PATIDAR, S., AND NARAYANAN, P. J. 2008. Ray casting deformable models on the gpu. In *Indian Conference on Computer Vision, Graphics and Image Processing*, IEEE Press.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2005. Photon mapping on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, ACM, New York, NY, USA, 258.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of International Parallel and Distributed Processing Symposium*.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 97–106.

SIGGRAPH ASIA COURSES. 2008. Beyond programmable shading.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5, 1–11.

ZHOU, K., GONG, M., HUANG, X., AND GUO, B. April, 2008. Highly parallel surface reconstruction. Tech. Rep. MSR-TR-2008-53, Microsoft Research.

# A Ordered Atomic Operations

An *atomic operation* is a set of actions that can be combined so that they appear to the rest of the system to be a single operation that succeeds or fails. An atomic operation on the shared location can be implemented by serializing it in some order such that the final results are correct. That is, for an operation $\mathcal{O}$ performed concurrently on a location $M$ by a set $S$ of processes, the resultant value in $M$ is:

$$M \leftarrow \mathcal{O}_t(\mathcal{O}_s(\cdots \mathcal{O}_r(\mathcal{O}_q(\mathcal{O}_p(M))))),$$

where $\{p, q, r, \cdots, s, t\}$ is a permutation $\mathcal{P}(S)$ of the processes in $S$. All permutations will give correct results for associative operations. Atomic operations of the *test-and-set* class are needed to build coherent data structures on distributed systems. These operations return to each process the value of the location $M$ immediately prior to its own operation, in one indivisible step. The effect of the atomic operation is equivalent to the following steps.

1. Compute a permutation $\mathcal{P}$ of $S$ as $p_1, p_2, p_3, \cdots, p_{|S|}$. The exact permutation used is unspecified and is usually implementation-dependent.
2. Each process $p_i$ gets the following partial result $m_i$ as its return value

   $$m_i \leftarrow \mathcal{O}_{i-1}(\mathcal{O}_{i-2}(\cdots \mathcal{O}_1(M))),$$

   where $\mathcal{O}_j$ is the operation of the $j$'th process in $\mathcal{P}$. The operation of the first process of the permutation is applied first.
3. The final result in $M$ is given by

   $$M \leftarrow \mathcal{O}_{|S|}(\mathcal{O}_{|S|-1}(\cdots(\mathcal{O}_2(\mathcal{O}_1(M))))).$$

An *ordered atomic* invocation of a concurrent operation $\mathcal{O}$ on a shared location $M$ is equivalent to its serialization within the set $S$ of processes that contend for $M$ in the order of a given priority value $\pi$. The steps involved are the same as above, with the the first one replaced by:

1. Compute a permutation $\mathcal{P}$ of $S$ as $p_1, p_2, \cdots, p_{|S|}$ such that

   $$\pi(p_1) \leq \pi(p_2) \leq \cdots \leq \pi(p_{|S|}).$$

**Example:** Assume $N$ processes hold a bit $b$ and a data element *data*. Assume $n$ processes have $b = 1$, where $n < N$ is not known ahead of time. The $n$ data elements are to be packed into a shared array of length $n$. Atomic increment applied to a common count value will give each process a unique index to which its data can be written. The order of packing will be unspecified. Assume a different scenario in which the packing has to follow the order of the unique processor ID ranging from 1 to $N$. That is, the data from a process $i$ should appear earlier than the data from all processes $j$ if $i < j$. Atomic operation cannot gurantee the required results if an arbitrary (implementation-dependent) permutation is applied to serialize the computations of processes contending to increment the same location. This problem can be solved easily using an *ordered atomic increment* with the process ID as the priority value. The increment operation returns the current value of the location and post-increments it. The priority ensures that processes with lower IDs get lower values of *index* than those with higher IDs, resulting in the desired ordering of the data elements.

### Ordered Atomics on CUDA

The atomic operations on the global memory and shared memory perform an implementation-dependent serialization and cannot guarantee any ordering. Our experiments on the GPU hardware verified this fact; the ordering is not maintained and split done using them produce wrong results. We can, however, simulate ordered atomic operations by serializing the threads of a warp. Each thread will wait for its turn to write. This will incur a fixed overhead proportional to the warp size, but does not perform extra shared memory writes. The kernel code outline for ordered atomic increment of `index` is given below.

```
for i = 0 to WARPSIZE-1 sequentially
   if (threadIdx.x == i)
       ownIndex = index++
```

For each of the 32 cycles, only one thread performs the write on the shared memory thus ensuring the atomicity. This approach is slow, but can control the order in which clashing threads are serialized. The thread ID is used as the priority value but other priority values can also be used. Our experiments on the GPU confirms this fact and the ordering can be preserved for clashes within the same warp of threads. The current GPU hardware does not guarantee ordering of warp scheduling for the sake of greater ability to hide memory latencies. Thus, ordered atomic operations are not guaranteed to work across multiple warps by serializing. The serializing incurs a speed penalty factor of 5 to 10 times over the shared memory atomic operations implemented in hardware.