Sorting Large Multifield Records on a GPU*

Shibdas Bandyopadhyay and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
shibdas@ufl.edu, sahni@cise.ufl.edu

***Abstract*—We extend the fastest comparison based (sample sort) and non-comparison based (radix sort) number sorting algorithms on a GPU to sort large multifield records. Two extensions - direct (the entire record is moved whenever its key is to be moved) and indirect ((key,index) pairs are sorted using the direct extension and then records are ordered according to the obtained index permutation) are discussed. Our results show that for the $ByField$ layout, the direct extension of the radix sort algorithm GRS [1] is the fastest for 32-bit keys when records have at least 12 fields ; otherwise, the direct extension of the radix sort algorithm SRTS [13] is the fastest. For the Hybrid layout, the indirect extension of SRTS is the fastest.**

***Index Terms*—Graphics Processing Units, sorting multifield records, radix sort, sample sort.**

I. Introduction

Graphics Processing Units (GPUs) are fast becoming an essential component of desktop computers. Cheap prices and massively parallel computation capability make them a viable choice for desktop supercomputing in addition to accelerating games and other graphics intensive tasks. From the view of general purpose computation, GPUs are manycore processors capable of running thousands of threads with very little context switching overhead. NVIDIA's Tesla GPUs come with 240 scalar processing cores (SPs) [14], organized into 30 Streaming multiprocessors (SM) each having 8 SPs. Each SM has a 16 KB fast shared memory that is shared among the threads running on that SM. There is also a vast register file comprising of 16384 32-bit registers that are used to store local variables of threads and states of numerous threads for context switching purposes. Being a graphics processor, each SM also includes texture caches to make fast texture look-up. The GPU also has a small read only constant memory. Each Tesla GPU comes with a 4GB off-chip global (or device) memory. Figure 1 gives the Tesla architecture. GPUs can now be programmed using general purpose languages such as C with Application Programming Interfaces (APIs) like OpenCL or an Nvidia specfic C extension known as Compute Unified Driver Architecture (CUDA) [23].

One of the very first GPU sorting algorithms, an adaptation of bitonic sort, was developed by Govindraju et al. [6]. Since this algorithm was developed before the advent of CUDA, the algorithm was implemented using GPU pixel shaders. Zachmann et al. [7] improved on this sort algorithm by using $BitonicTrees$ to reduce the number of comparisons while merging the bitonic sequences. Cederman et al. [5] have adapted quick sort for GPUs. Their adaptation first partitions the sequence to be sorted into subsequences, sorts these subsequences in parallel, and then merges the sorted subsequences in parallel. A hybrid sort algorithm that splits the data using bucket sort and then merges the data using a vectorized version of merge sort is proposed by Sintron et al. [18]. Satish et al. [16] have developed an even faster merge sort The fastest GPU merge sort algorithm known at this time is Warpsort [21]. Warpsort first creates sorted sequences using bitonic sort; each sorted sequence being created by a thread warp. The sorted sequences are merged in pairs until too few sequences remain. The remaining sequences are partitioned into subsequences that can be pairwise merged independently and finally this pairwise merging is done with each warp merging a pair of subsequences. Experimental results reported in [21] indicate that Warpsort is about 30% faster than the merge sort algorithm of [16]. Another comparison-based sort for GPUs–GPU sample sort–was developed by Leischner et al. [12]. Sample sort is reported to also be about 30% faster than the merge sort of [16], on average, when the keys are 32-bit integers. This would make sample sort competitive with Warpsort for 32-bit keys. For 64-bit keys, sample sort is twice as fast, on average, as the merge sort of [16].

[17], [22], [11], [16], [13] have adapted radix sort to GPUs. Radix sort accomplishes the sort in phases where each phase sorts on a digit of the key using, typically, either a count sort or a bucket sort. The counting to be done in each phase may be carried out using a prefix sum or $scan$ [4] operation that is quite efficiently done on a GPU [17]. Harris et al.'s [22] adaptation of radix sort to GPUs uses the radix 2 (i.e., each phase sorts on a bit of the key) and uses the $bitsplit$ technique of [4] in each phase of the radix sort to reorder records by the bit being considered in that phase. This implementation of radix sort is available in the CUDA Data Parallel Primitive (CUDPP) library [22]. For 32-bit keys, this implementation of radix sort requires 32 phases. In each phase, expensive scatter operations to/from the global memory are made. Le Grand et al. [11] reduce the number of phases and hence the number of expensive scatters to global memory by using a larger radix, 2^b, for $b > 0$. A radix of 16, for example, reduces the number of phases from 32 to 8. The sort in each phase is done by first computing the histogram of the 2^b possible values that a digit with radix 2^b may have. Satish et al. [16]

* This research was supported, in part, by the National Science Foundation under grants 0829916 and NETS 0963812. The authors acknowledge the University of Florida High-Performance Computing Center for providing computational resources and support that have contributed to the research results reported within this paper. URL: http://hpc.ufl.edu.

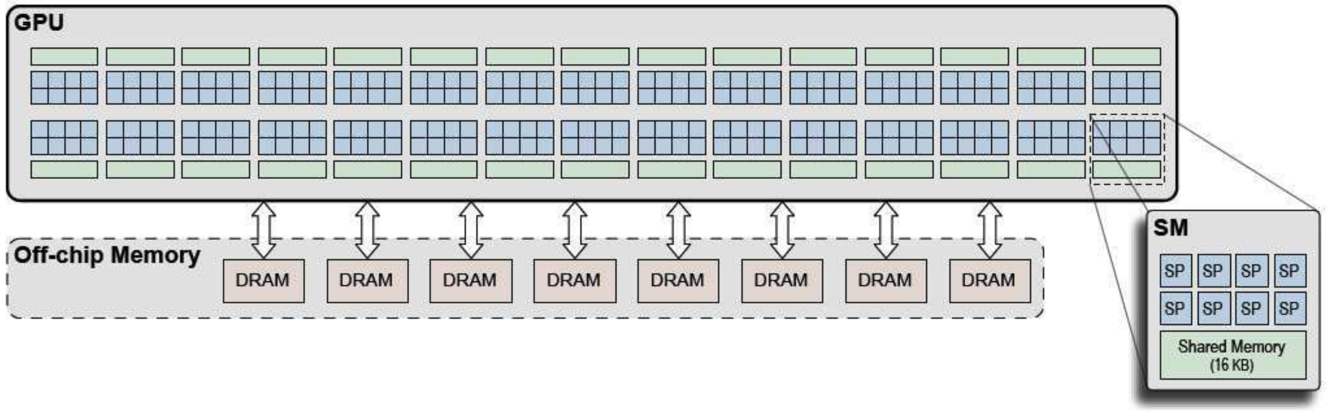

Fig. 1: NVIDIA's Tesla GPU [16]

further improve the 2^b-radix sort of Le Grand et al. [11] by sorting blocks of data in shared memory before writing to global memory. This reduces the randomness of the scatter to global memory, which, in turn, improves performance. The radix-sort implementation of Satish et al. [16] is included in NVIDIA's CUDA SDK 3.0. Merrill and Grimshaw [13] have developed an alternative radix sort, SRTS, for GPUs that is based on a highly optimized algorithm, developed by them, for the scan operation and co-mingling of several logical steps of a radix sort so as to reduce accesses to device/global memory. Presently, SRTS is the fastest GPU radix sort algorithm for integers as well as for records that have a 32-bit key and a 32-bit value field. Bandyopadhyay and Sahni [1] developed a radix sort algorithm which outperforms SDK radix sort algorithm while sorting integers and outperforms SRTS in hybrid layout while sorting records with more than one field.

Our focus, in this paper is to extend GPU number sorting algorithms to handle records with multiple fields using different layouts. Specifically, we extend the algorithms which are the fastest comparison and non-comparison based sorting algorithms. Warpsort and sample sort are both comparison based algorithms with similar reported performance. So, we selected one of these, sample sort, to extend to the sorting of multifield records. Among non-comparison based sorting methods SRTS is the fastest for sorting numbers while GRS is the fastest for sorting large records in the $Hybrid$ layout (Section III) using the direct strategy (Section III). So, we extend SRTS for sorting records. and GRS to sort records in other layouts.

The remainder of this paper is organized as follows. In Section II we describe features of the NVIDIA Tesla GPU that affect program performance. In Section III, we describe three popular layouts for records as well as two overall strategies to handle the sort of multi-field records. The next three sections discuss the extension of sample sort, SRTS and GRS to handle records in different layouts. Section VII provides extensive comparative results of sorting records in different layouts using these sorting algorithms.

II. NVIDIA Tesla Performance Characteristics

GPUs operate under the master-slave computing model (see [15] for e.g.) in which there is a host or master processor to which are attached a collection of slave processors. A possible configuration would have a GPU card attached to the bus of a PC. The PC CPU would be the host or master and the GPU processors would be the slaves. The CUDA programming model requires the user to write a program that runs on the host processor. At present, CUDA supports host programs written in C and C++ only though there are plans to expand the set of available languages [23]. The host program may invoke *kernels*, which are C functions, that run on the GPU slaves. A kernel may be instantiated in synchronous (the CPU waits for the kernel to complete before proceeding with other tasks) or asynchronous (the CPU continues with other tasks following the spawning of a kernel) mode. A kernel specifies the computation to be done by a thread. When a kernel is invoked by the host program, the host program specifies the number of threads that are to be created. Each thread is assigned a unique ID and CUDA provides C-language extensions to enable a kernel to determine which thread it is executing. The host program groups threads into blocks, by specifying a block size, at the time a kernel is invoked. Figure 2 shows the organization of threads used by CUDA.

III. Multifield Record Layout And Sorting

A record R is comprised of a key k and m other fields $f_1, f_2, \cdots, f_m$. For simplicity, we assume that the key and each other field occupies 32 bits. Let k_i be the key of record R_i and let f_{ij}, $1 \le j \le m$ be this record's other fields. With our simplifying assumption of uniform size fields, we may view the n records to be sorted as a two-dimensional array $fieldsArray[][]$ with $fieldsArray[i][0] = k_i$ and $fieldsArray[i][j] = f_{ij}$, $1 \le j \le m$, $1 \le i \le n$. When this array is mapped to memory in column-major order, we get the *ByField* layout of [2]. This layout was used also for the AA-sort algorithm developed for the Cell Broadband Engine in [9] and is essentially the same as that used by the GPU radix sort algorithm of [16]. When the fields array is mapped

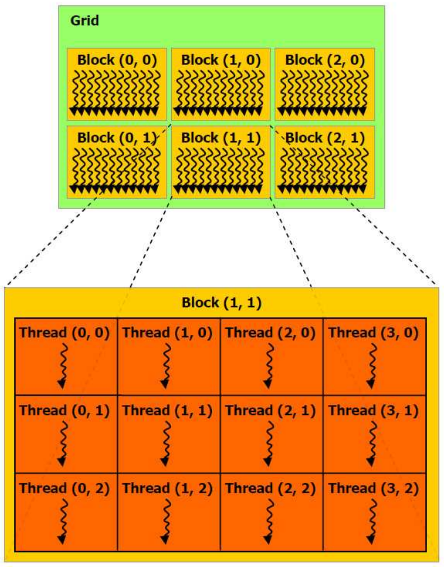

Fig. 2: Cuda programming model [23]

to memory in row-major order, we get the $ByRecord$ layout of [2]. A third layout, $Hybrid$, is employed in [13]. This is a hybrid between the $ByField$ and $ByRecord$ layouts. The keys are stored in an array and the remaining fields are stored using the $ByRecord$ layout. Essentially then, in the $Hybrid$ layout, we have two arrays. Each element of one array is a key and each element of the other array is a structure that contains all fields associated with an individual record. In this paper, we limit ourselves to the $ByField$ and $Hybrid$ layouts. We do not consider the $ByRecord$ layout as it appears that most effectively way to sort in this layout is to first extract the keys, sort (key, index) pairs and then reorder the records into the obtained sorted permutation. The last two steps are identical to the steps in an optimal sort for the $Hybrid$ layout. So, we expect that good strategies to sort in the $Hybrid$ layout will also be good for the $ByRecord$ layout. When the sort begins with data in a particular layout format, the result of the sort must also be in that layout format.

At a high level, there are two very distinct approaches to sort multifield records. In the first, we construct a set of tuples (k_i, i), where k_i is the key of the ith record. Then, these tuples are sorted by extending a number sort algorithm so that whenever the number sort algorithm moves a key, the extended version moves a tuple. Once the tuples are sorted, the original records are rearranged by copying records from the $fieldsArray$ to a new array placing the records into their sorted positions in the new array or in-place using a cycle chasing algorithm as described for a table sort in [8]. The second strategy is to extend a number sort so as to move an entire record every time its key is moved by the number sort. We call the first strategy as indirect and the second strategy as direct strategy for sorting multifield records. There are advantages and disadvantages to each strategy. Indirect strategy seems to perform much less work than the direct during sorting as the satellite data that needs to be moved with the key is only an integer index while in the direct strategy its the entire record. On the flip side, the indirect strategy has a very costly random global memory access phase at the end when records are moved to their sorted positions whereas the direct strategy does not have this phase.

IV. Sample Sort For Sorting Records

Sample sort [12] is a multi-way divide and conquer sorting algorithm which performs better when the memory bandwidth is an issue as the data transferred to and from the global memory is less than two-way approach. The serial version of sample sort works by first choosing a set of splitters randomly from the input data. The splitters are then sorted and arranged in increasing order of their values. The input data set is divided into buckets delimited by successive splitters. The elements in a particular bucket have values that are bounded by the guarding splitters. The sample sort is then called again on each of these buckets. This process continues until the size of the bucket becomes less than a certain threshold. At this point a base sorting algorithm is used to sort the small bucket. Figure 3 shows the steps of serial sample sort.

```
SampleSort($a[]$)
  if sizeof($a$) $\leq$ $M$ // a threshold
  {
    Sort($a$); // Use a sorting method to sort $a$
    return;
  }
  Select $k$ elements randomly from a and put them in $samples[]$;
  Sort($samples[]$);
  for(each element $e$ in a[])
  {
    find $i$ such that $samples[i] \leq e \leq samples[i+1]$;
    Put $e$ in bucket $b[i]$;
  }
  for(each bucket $b[i]$)
  {
    SampleSort($b[i]$);
  }
```

Fig. 3: Serial Sample Sort

To obtain an efficient parallel version of sample sort, it is necessary to balance the size of buckets assigned to thread blocks. This is done by choosing the splitters from a large randomly selected sample of keys. Once the splitters are selected, the records are partitioned into buckets by first dividing the data into equal sized tiles with each tile being assigned to a block of threads. A thread block examines its tile of data and assigns records in this tile to buckets whose boundaries are the previously chosen splitters. Finally the buckets produced for the tiles are combined to obtain global buckets. The step in the GPU sample sample sort of [12] are described below.

Phase 1: During this phase, the splitters are chosen. First, a set of random samples are taken out of the elements in the buckets. A set of splitters are then chosen from these random samples. Finally, splitters are sorted using odd-even merge sort [3] in shared memory and a Binary Search Tree of splitters is created to facilitate the process of finding the bucket for an element.

Phase 2: Each thread block is assigned a part of the input data. Threads in a block load the Binary Search Tree into shared memory and then calculate the bucket index for each element in the tile. At the end threads store the number of elements in each bucket as a k-entry histogram in the global memory.

Phase 3: The per block k-entry histograms are prefix-summed to obtain the global offset for each bucket.

Phase 4: Each thread block in this phase again calculates bucket index for all keys in its tile. They also calculate the local offset within the buckets. The local offsets are added to the global offsets from the previous phase to get the final position of the records.

When sample sorting records using the direct strategy outlined in Section III the records need to be moved only during the fourth phase as in all other phases only the keys are required to be moved. This distribution of the records from the large bucket to small buckets is repeated multiple times till the size of the bucket is below a specified threshold. Finally, quicksort is done on the records when the bucket size is small. Records are also moved during the partitioning phase of the quicksort within a small bucket. The fourth phase and the quick sort part of sample sort can be extended to handle records in $ByField$ and $ByRecord$ format. In $ByField$ layout, while moving $fieldsArray[i]$ to $outfieldsArray[j]$, threads can move the corresponding fields as shown in Figure 4

```
outKey[j] = key[i];
//Move the fields
for(f = 1; f <= m; f++) {
  outfieldsArray[j][p] = fieldsArray[i][p];
}
```

Fig. 4: Moving records in $ByField$ layout

Similarly, in $ByRecord$ ($Hybrid$) format fields can be moved by a thread while moving the keys. Figure 5 shows the code to move records assuming that $fieldsArray[i]$ and $outfieldsArray[j]$ are structures that contain the fields.

```
outKey[j] = key[i];
//Move the fields
outfieldsArray[j] = fieldsArray[i];
```

Fig. 5: Moving records in $ByRecord$ layout

We observe that in the $ByField$ layout, threads in a warp access adjacent elements in global memory resulting in coalescing of a memory access while in the $ByRecord$ layout, threads in a warp access words in the global memory that are potentially far apart generating global memory transactions of size at most 16 bytes. We employ a strategy of grouping the threads together so that we can generate larger memory transactions. Rather than a single thread reading and writing the entire record, we employ a group of threads to read and write the records into the global memory cooperatively. Then this same group of threads iterates to read and write other records co-operatively. This ensures larger global memory transactions. As an example, lets say the record is of 64 bytes in length and as each thread can read in 16 bytes of data using an $int4$ datatype, we can group 4 threads together so that they can read the entire record together. Then this thread group iterates over to read other records until all the records assigned to the thread group has been read. Let $numThreads$ denote the number of threads in a block and suppose that each thread is to read in one record and put it into proper place in the output array. Assume that records from $startOffset$ to $(startOffset + numThreads)$ are processed by this thread block. For sake of clarity of the pseudocode we assume that there is a map $mapInToOut$ which determines the proper position in the output array. In case of sample sort, it would be the Binary Search tree constructed out of the splitters which would determine the position of a particular record in the output array. Figure 6 outlines the optimized version of moving records using coalesced read and write.

```
// Determine the number of threads required to
// read the entire record
numThrdsInGrp = sizeof(Rec) / 16;
// Total number of records to be read = number of
// threads in the group
numItrs = numThrdsInGrp;
// Number of records read in a single iteration
//by all threads
nRecsPerItr = numThrds / numThrdsInGrp;
// Convert Record arrays to int4 arrays
recInt4 = (int4 *)rec; outRecInt4 = (int4 *)outRec;
// Determine the starting record and position in the
group for this thread
startRec = startOffset + threadId / numThrdsInGrp;
posInGrp = threadId % numThrdsInGrp;
for(i = 0; i < numItrs; i++)
{
  outRecInt4[mapInOut(startRec) + posInGrp] =
  recInt4[startRec + posInGrp];
  startRec += numThrdsInGrp;
}
```

Fig. 6: Optimized version of moving records in $ByRecord$ layout

V. SRTS For Sorting Records

SRTS employs a highly optimized version of the scan kernel developed by Merrill and Grimshaw[13] to perform radix sort. As with the other radix sort strategies it progressively radix sorts on 4 bits a phase. Hence, SRTS requires 8 phases to completely sort 32-bit integers. SRTS focuses on reducing the total number of reads and writes to the global memory by combining different functions done in separate kernels. The performance of most of the kernels in earlier radix sort implementation is limited by the bandwidth between SM and global memory. The technique introduced in SRTS increases the arithmetic intensity of these memory bound operations and eliminates the need for additional kernels for sorting as performed in SDK radix sort by [16]. SRTS further brings parity between computation and memory access by only having a fixed number of thread blocks in the GPU. Each thread loops over to process the data in batches and hence the amount of computation done per thread increases substantially. SRTS uses a fixed number of thread blocks enough to occupy all SMs in the GPU. The input data is divided into tiles and a set of tiles is assigned to a thread block. It uses a radix of 2^b with $b = 4$. With $b = 4$ and 32-bit keys, the radix sort runs in 8 phases with each phase sorting on 4 bits of the key. Each thread block while processing a tile finds out the number of keys with a particular radix value. The radix counters indicating number of keys with a particular radix value are accumulated over the tiles assigned to that thread block. SRTS consists of following steps [13].

Phase 1- Bottom Level Reduction: This phase consists of two sub-phases. During the first phase, each thread reads in an element from the input data and extracts the b bits being considered and increases the histogram counts correspondingly. The threads in the thread block loop over the tiles assigned to the thread block and the radix counters are accumulated in the local registers as there are only 16 different radix values for 4 bits. After the last tile of input data is processed, the threads within a block perform a local prefix sum cooperatively to prefix sum the sequence of counters and the result is written to the global memory as a set of prefix sums.

Phase 2 - Top Level Scan: In this phase a single block of threads operates over the prefix sums to compute the global prefix sum. The block-level scan is modified to handle a concatenation of partial prefix sums.

Phase 3 - Bottom Level Scan: Lastly, the threads in a block sort the elements by first reading in the prefix sum calculated during the top-level scan phase. The thread block reads in the elements again and extracts the b bits being considered. A local parallel scan is done to find the local prefix sum. These local offsets are seeded with the global prefix sums calculated earlier to get the final position of the element in the output. The input elements are next scattered in shared memory using the local offsets to put them in sorted order within the tile. Finally, those elements are read in sorted order from the shared memory and written onto the global memory. This strategy ensures better memory coherence and generates larger global memory transactions. As with the first phase, the radix counters are accumulated and are carried over to the next tile of input processed by this block of threads using local registers.

The final scatter of input elements happens during the very last phase. Only keys of the records are required during other phases. So, the fields of the record can also be moved during the third phase while scattering the keys. We can use the strategies outlined in Figures 4 and 5 to scatter the fields in $ByField$ and $ByRecord$ layouts respectively. However, due to the way SRTS is implemented using generic programming it is difficult to use the an optimized version of record moving (Figure 6) in $ByRecord$ format. The third phase of record scattering occurs only 8 times for 32-bit keys during the entire sorting process and does not depend on the number of records being sorted. This indicates there is a possibility, for SRTS, direct strategy of moving records while sorting might actually perform better than the indirect strategy of sorting $(key, index)$ pairs followed by rearrangement.

VI. GRS For Sorting Records

GRS is developed specifically for sorting records [1] along the lines of the SDK radix sort [16] but the focus is to reduce the number of times a record is read from or written into the global memory. As with the other strategies, the input data is divided into tiles and we define the $rank$ of an element as the number of elements having the same digit value before the element in the input data tile. GRS employs a additional memory to store $rank$ of the elements and eliminates the sorting phase proposed in SDK sort [16]. This helps to find the local offset of records within a input data tile and helps to sort them in the shared memory much like SRTS before writing them out to the global memory. GRS also processes 4 bits per pass and hence has 8 passes in total to sort records with 32-bit keys. The three phases in each pass of GRS are:

Phase 1: Compute the histogram for for each tile as well as the rank of each record in the tile. In this phase, a block of 64 threads operate on a input tile to cooperatively read the keys from global memory to the shared memory. The global memory reads are made coalesced by ensuring consecutive threads access the consecutive keys in global memory. The writing on the shared memory is performed with an offset to avoid bank conflicts. Threads in read the keys from the shared memory with an offset and calculate the histogram and the rank using the digit counters. The rank overwrites the keys in the shared memory as we don't need them after their ranks are calculated. Finally, the histogram and the ranks are written out to the global memory. As the rank can not exceed the size of a tile which is typically set to the 1024 records, a full integer is not required to store the rank.

Phase 2: The prefix sums of the histograms of all tiles are computed.

Phase 3: Lastly, in this phase the entire record is first read from the global memory to the shared memory. We then use the ranks, prefix-summed local histograms to find out the local offset for each record in the tile. We put the records in the shared memory according to the offset so that we get a sorted tile of records. The threads then read the records from shared memory in order and use the global prefix sum to put them in their final place in the output.

Much like SRTS, the final scatter of the records is done in the last phase. As the last phase caters to a very simple implementation, we can efficiently read and write records in this phase. This simplicity enables us to use the algorithms of Figures 4 and 6 to move records in $ByField$ and $ByRecord$ layouts respectively. As with SRTS, we only move records 8 number of times during the sorting of records with 32-bit keys. Hence, GRS also has a fair chance of outperforming the first strategy of sorting records by using (key, index) pairs.

VII. Experimental Results

We implemented and evaluated the record sorting algorithms mentioned in the previous sections using Nvidia CUDA SDK 3.2. Specifically, we evaluated two versions of sample sort, GRS and SRTS each corresponding to the direct and indirect strategies for sorting records mentioned in Section III. We evaluated the following algorithms

1) SampleSort-direct...samplesort algorithm of [12] extended for sorting records.
2) SampleSort-indirect...We form $(key, index)$ pair for each record and then sort them using samplesort. Finally a rearrangement is done to put the entire record in the sorted order.
3) SRTS-direct...SRTS algorithm [13] extended for sorting records.
4) SRTS-indirect... We form $(key, index)$ pair for each record and then sort them using SRTS. Finally a rearrangement is done to put the entire record in the sorted order.
5) GRS-direct...GRS algorithm for sorting records [1].
6) GRS-indirect... We form $(key, index)$ pair for each record and then sort them using GRS. Finally a rearrangement is done to put the entire record in the sorted order.

We implemented the above 6 multifield sorting algorithms on an Nvidia Tesla C1060 which has 240 cores and 4 GB of global memory. The algorithms are evaluated using randomly generated input sequences. In our experiments, the number of 32-bit fields per record is varied from 2 to 20 (in addition to the key field) and the number of records was 10 million. Also, the algorithms ate implemented for both $ByField$ and $Hybrid$ layout. For each combination of number of fields and layout type, the time to sort 10 random sequences was obtained. The standard deviation in the observed run times was small and we report only the average times.

A. Run Times for $ByField$ layout

Figures 7 through 8 show the comparison of SampleSort, GRS and SRTS using direct and indirect strategies for sorting 10M records with 2 to 9 fields in the $ByField$ layout. During each run, we have used the same set of records while comparing these algorithms. SampleSort-indirect runs 36% faster than SampleSort-direct when sorting 10M records with 2 fields while it runs 66% faster for records with 9 fields.

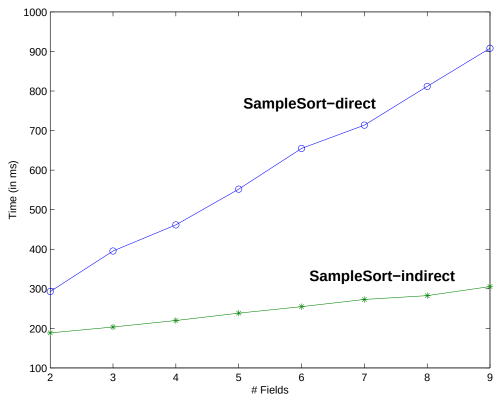

Fig. 7: SampleSort-direct and SampleSort-indirect for 10M records ($ByField$)

SRTS-indirect runs 37% slower than SRTS-direct sorting 10M records with 2 fields while it runs 27% slower for records with 9 fields.

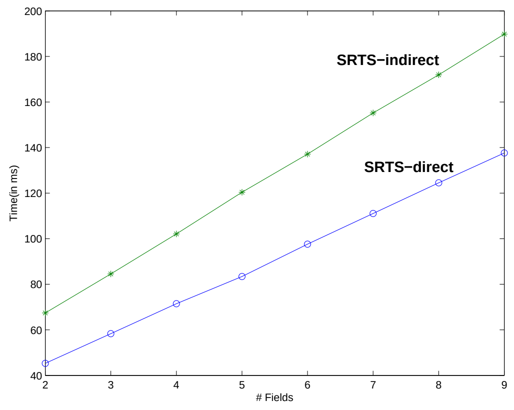

Fig. 8: SRTS-direct and SRTS-indirect for 10M records ($ByField$)

GRS-indirect runs 27% slower than GRS-direct when sorting 10M records with 2 fields while it runs 33% slower for records with 9 fields.

Figure 10 shows the comparison between the faster version of each of these three algorithms. SRTS-direct is the fastest algorithm to sort records in the $ByField$ layout when records have between 2 to 11 fields. GRS-direct is the fastest algorithm for sorting records with more than 11 fields. SRTS-direct runs 35% faster than GRS-direct when sorting 10M records with 2 fields while GRS-direct runs 38% faster than SRTS-direct when records have 20 fields. SampleSort-indirect is the slowest, running 63% slower than GRS when sorting records with 2 fields and 48% slower when sorting records with 20 fields.

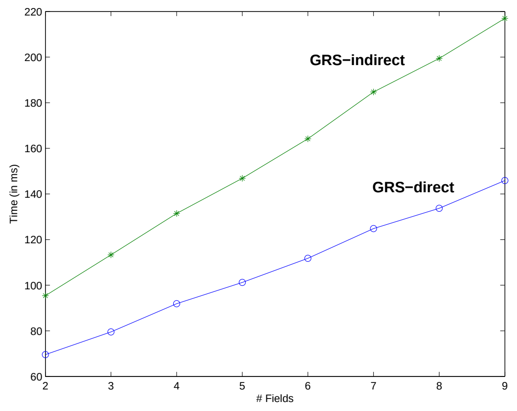

Fig. 9: GRS-direct and GRS-indirect for 10M records ($ByField$)

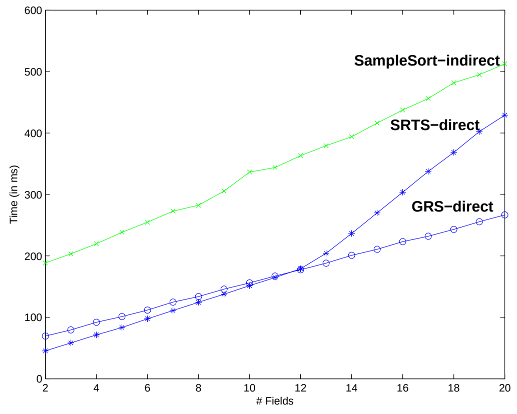

Fig. 10: Different sorting algorithms for 10M records ($ByField$)

B. Run Times for $Hybrid$ layout

Figures 11 through 12 show the comparison of SampleSort, GRS and SRTS using the direct and indirect strategies for sorting 10M records with 2 to 9 fields in the $Hybrid$ layout. SampleSort-indirect runs 12% faster than SampleSort-direct when sorting 10M records with 2 fields while it runs 72% faster for records with 9 fields.

SRTS-indirect runs 38% faster than SRTS-direct sorting 10M records with 2 fields while it runs 79% faster for records with 9 fields.

GRS-indirect runs 13% slower than GRS-direct sorting 10M records with 2 fields while it runs 41% faster for records with 9 fields.

Figure 14 shows the comparison between the faster version of each of these three algorithms. SRTS-indirect is the fastest algorithm to sort records in the $Hybrid$ layout. SRTS-indirect runs 27% faster than GRS-indirect and 71% faster than SampleSort-indirect when sorting records with 2 fields while it runs 3% faster than GRS-indirect and 16% faster than SampleSort-indirect when sorting records with 20 fields.

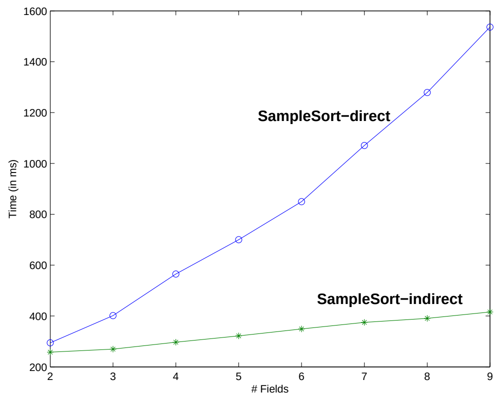

Fig. 11: SampleSort-direct and SampleSort-indirect for 10M records ($Hybrid$)

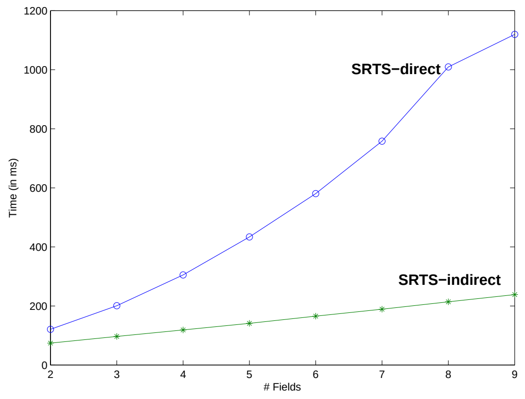

Fig. 12: SRTS-direct and SRTS-indirect for 10M records ($Hybrid$)

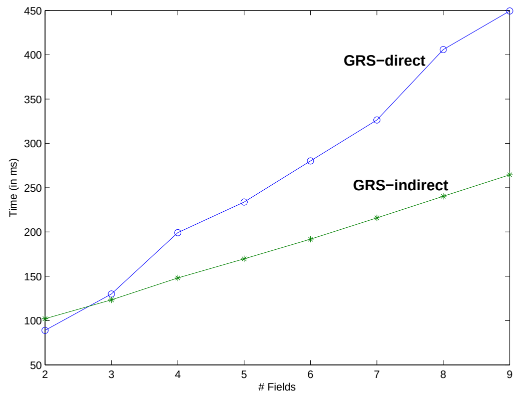

Fig. 13: GRS-direct and GRS-indirect for 10M records ($Hybrid$)

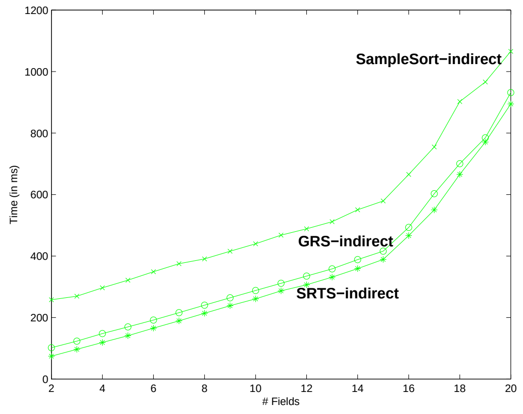

Fig. 14: Different sorting algorithms for 10M records ($Hybrid$)

VIII. Conclusion

We have considered two extensions – direct and indirect of the GPU number sort algorithms SampleSort, SRTS and GRS. We showed how each extension could be implemented optimally on a GPU maximizing device memory and SM bandwidth. Experiments conducted on the NVIDIA C1060 Tesla indicate that, for the $ByField$ layout, GRS-direct is the fastest sort algorithm for 32-bit keys when records have at least 12 fields. When records have fewer than 12 fields, SRTS-direct is the fastest. This happens due to the better memory coalescing achieved by GRS-direct during the last scatter phase compared to SRTS-direct. In the $Hybrid$ layout SRTS-indirect is the fastest algorithm although the performance gap between SRTS-indirect and GRS-indirect narrows once records have more number of fields. At this point, the last global rearrangement phase in both GRS-indirect and SRTS-indirect dominates over other phases of the algorithms and both GRS-indirect and SRTS-indirect have similar run times.

Intuitively, one would expect the indirect method to be faster than the direct method for large records. This is because the indirect method moves each record only once while the direct method moves records many times (O(log n) times on average for sample sort and 8 times for GRS and SRTS). This intuition is borne out in all cases other than when the records are in $ByField$ layout and radix sort is used. SRTS-direct and GRS-direct move each field 8 times affording some opportunity for coalescing of device memory accesses. Although SRTS-indirect and GRS-indirect move each field only once, coaleascing isn't possible as the fields of a record in the $ByField$ layout are far apart in device memory.

Acknowledgement

We are grateful to the authors of sample sort [12] and SRTS [13] for making their code available.

References

[1] Bandyopadhyay, S. and Sahni, S., GRS - GPU Radix Sort for Large Multifield Records, *International Conference on High Performance Computing* (HiPC), 2010.
[2] Bandyopadhyay, S. and Sahni, S., Sorting Large Records on a Cell Broadband Engine, *IEEE International Symposium on Computers and Communications* (ISCC), 2010.
[3] Batcher, K.E., Sorting Networks and Their Applications, *Proc. AFIPS Spring Joint Computing Conference*, vol. 32, 307-314, 1968.
[4] Blelloch, G.E., Vector models for data-parallel computing. Cambridge, MA, USA: MIT Press, 1990.
[5] Cederman, D. and Tsigas, P., GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors, *ACM Journal of Experimental Algorithmics*(JEA), 14, 4, 2009.
[6] Govindaraju, N., Gray, J., Kumar, R. and Manocha D., Gputerasort: High performance graphics coprocessor sorting for large database management, *ACM SIGMOD International Conference on Management of Data*, 2006.
[7] Greb, A. and Zachmann, G., GPU-ABiSort: optimal parallel sorting on stream architectures, *IEEE International Parallel and Distributed Processing Symposium* (IPDPS), 2006.
[8] Horowitz, E., Sahni, S., and Mehta, D., Fundamentals of data structures in C++, Second Edition, Silicon Press, 2007.
[9] Inoue, H., Moriyama, T., Komatsu, H., and Nakatani, T., AA-sort: A new parallel sorting algorithm for multi-core SIMD processors, *16th International Conference on Parallel Architecture and Compilation Techniques* (PACT), 2007.
[10] Knuth, D., *The Art of Computer Programming: Sorting and Searching*, Volume 3, Second Edition, Addison Wesley, 1998.
[11] Le Grand, S., Broad-phase collision detection with CUDA, GPU Gems 3, Addison-Wesley Professional, 2007.
[12] Leischner, N., Osipov, V. and Sanders P., GPU sample sort, IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.
[13] Merrill, D and Grimshaw A, Revisiting Sorting for GPGPU Stream Architectures, *University of Virginia, Department of Computer Science*, Technical Report CS2010-03, 2010.
[14] Lindholm, E., Nickolls, J., Oberman S. and Montrym J., NVIDIA Tesla: A unified graphics and computing architecture, *IEEE Micro*, 28, 3955, 2008.
[15] Sahni, S., Scheduling master-slave multiprocessor systems, *IEEE Trans. on Computers*, 45, 10, 1195-1199, 1996.
[16] Satish, N., Harris, M. and Garland, M., Designing Efficient Sorting Algorithms for Manycore GPUs, *IEEE International Parallel and Distributed Processing Symposium* (IPDPS), 2009.
[17] Sengupta, S., Harris, M., Zhang, Y. and Owens, J., D., Scan primitives for GPU computing, *Graphics Hardware 2007*, 97-106, 2007.
[18] Sintorn, E. and Assarsson, U., Fast parallel GPU-sorting using a hybrid algorithm, *Journal of Parallel and Distributed Computing*, 10, 1381-1388, 2008.
[19] Volkov, V. and Demmel, J.W., Benchmarking GPUs to Tune Dense Linear Algebra, *ACM/IEEE conference on Supercomputing*, 2008.
[20] Won, Y. and Sahni, S., Hypercube-to-host sorting, *Jr. of Supercomputing*, 3, 41-61, 1989.
[21] Ye, X., Fan, D., Lin, W., Yuan, N. and Ienne, P., High performance comparison-based sorting algorithm on many-core GPUs, IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.
[22] CUDPP: CUDA Data-Parallel Primitives Library, *http://www.gpgpu.org/developer/cudpp/*, 2009.
[23] NVIDIA CUDA Programming Guide, *NVIDIA Corporation*, version 3.0, Feb 2010.