GRS - GPU Radix Sort For Multifield Records*

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 number sorting algorithms on the GPU to sort large multi-field records. We notice that traditional way of sorting the records by first sorting a (key, index) pair to obtain the sorted permutation of the records followed by actually rearranging the entire records to their final position might not actually be the most efficient way to sort them depending on the type of sorting algorithm used and the layouts of the records in the memory.**

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

I. Introduction

Graphics Processing Units (GPUs) are fast becoming an essential component of the desktop computers. Cheap prices and massively parallel computation capability make them a viable choice for supercomputing on desktop 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 at once with a very little context switching overhead. NVIDIA's Tesla GPUs come with 240 scalar processing cores (SPs) [13], 8 of them are grouped into a Streaming multiprocessor (SM). So, there are a total of 30 SMs. Each SM has a 16 KB fast shared memory which is shared among the threads running on that SM. There is also a vast register file comprising of 16384 32-bit registers which 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. It also has a small read only constant memory. Each Tesla GPU comes with a 4GB off-chip global (or device) memory. Figure 1 shows a brief outline of the Tesla architecture. With the recent Fermi series of GPUs, many features including L1 and L2 cache hierarchy are introduced to make them more suitable toward general purpose computing. GPUs can now be programmed using general purpose languages such as C with Application Programming Interfaces (APIs) like OpenCL or Nvidia specfic C extension known as Compute Unified Driver Architecture (CUDA) [22]. During recent years, there has been a explosion of research directed toward expanding the applicability of GPUs to the fundamental high performance computing problems such as sorting.

* This research was supported, in part, by the National Science Foundation under grants 0829916 and 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.

One of the very first GPU sorting algorithms, an adaptation of bitonic sort, was developed by Govindraju et al. [5]. Since this algorithm was developed before the advent of CUDA, the algorithm was implemented using GPU pixel shaders. Zachmann et al. [6] improved on this sort algorithm by using $BitonicTrees$ to reduce the number of comparisons while merging the bitonic sequences. Cederman et al. [4] 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. [17]. Satish et al. [15] have developed an even faster merge sort The fastest GPU merge sort algorithm known at this time is Warpsort [20]. 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 [20] indicate that Warpsort is about 30% faster than the merge sort algorithm of [15]. Another comparison-based sort for GPUs–GPU sample sort–was developed by Leischner et al. [11]. Sample sort is reported to be about 30% faster than the merge sort of [15], 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 [15].

[16], [21], [10], [15], [12] 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 $scan$ [3] operation that is quite efficiently done on a GPU [16]. Harris et al.'s [21] 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 [3] 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 [21]. 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. [10] reduce the number of phases and hence the number of expensive scatters to global memory by using a

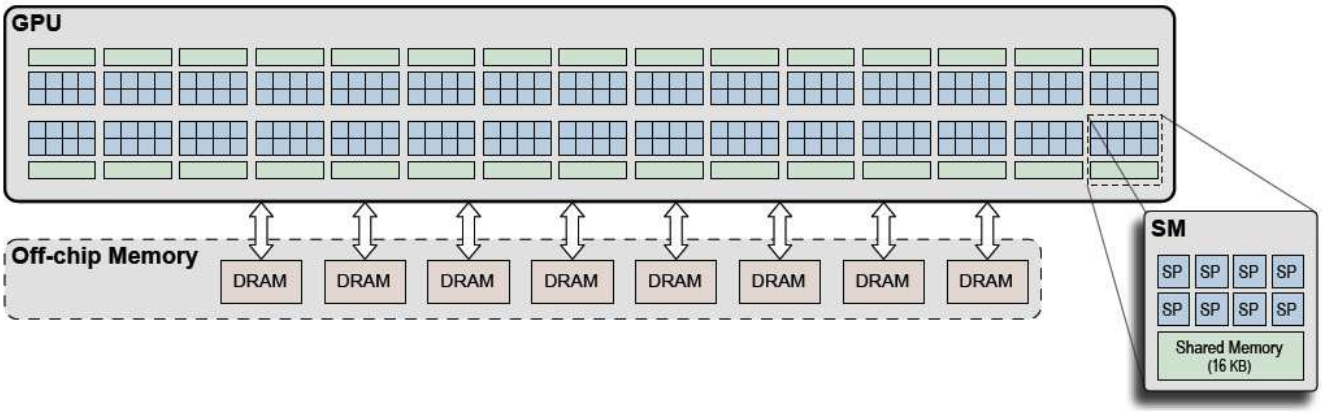

Fig. 1: NVIDIA's Tesla GPU [15]

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. [15] further improve the 2^b-radix sort of Le Grand et al. [10] 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. [15] is included in NVIDIA's CUDA SDK 3.0. Merrill and Grimshaw [12] 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 the sorting algorithms to handle records with multiple fields and arranged in different layouts. To this end, we have chosen the algorithms which are the fastest comparison and non-comparison based sorting algorithms namely, sample sort and SRTS respectively. We also include the GRS into the mix as it is already extended to sort the records with multiple fields. We extend these algorithms to handle records efficiently in $ByField$ and $ByRecord$ layouts outlined in [2].

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. Next three sections **??**, V and VI discuss the extension of sampleSort, 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 [14] 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 [22]. 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.

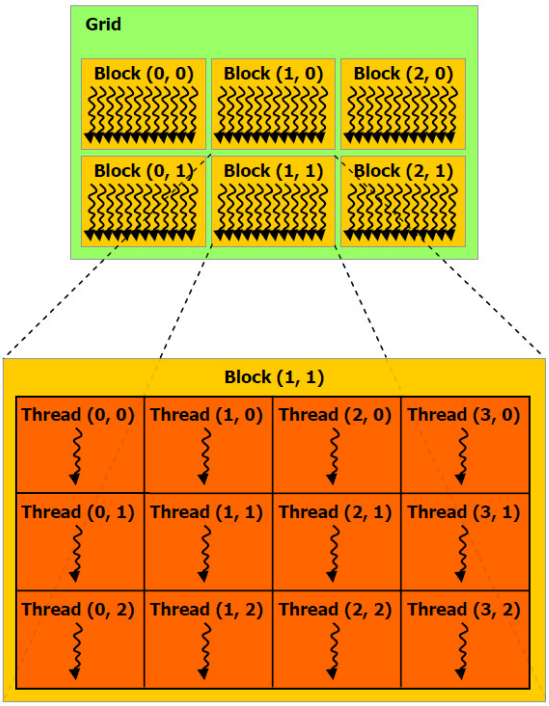

Fig. 2: Cuda programming model [22]

The GPU schedules the threads so that a block of threads runs on the cores of an SM. At any given time, an SM executes the threads of a single block and the threads of a block can execute only on a single SM. Once a block begins to execute on an SM, that SM executes the block to completion. Each SM schedules the threads in its assigned block in groups of 32 threads called a *warp*. The partitioning into warps is fairly intuitive with the first 32 threads forming the first warp, the next 32 threads form the next warp, and so on. A *half warp* is a group of 16 threads. The first 16 threads in a warp form the first half warp for the warp and the remaining 32 threads form the second half warp. When an SM is ready to execute the next instruction, it selects a warp that is ready (i.e., its threads are not waiting for a memory transaction to complete) and executes the next instruction of every thread in the selected warp. Common instructions are executed in parallel using the 8 SPs in the SM. Non-common instructions are serialized. So, it is important, for performance, to avoid thread divergence within a warp. Some of the other factors important for performance are:

1) Since access to global memory is about an order of magnitude more expensive than access to registers and shared memory, data that are to be used several times should be read once from global memory and stored in registers or shared memory for reuse.
2) When the threads of a half-warp access global memory, this access is accomplished via a series of memory transactions. The number of memory transactions equals the number of different 32-byte (64-byte, 128-byte, 128-byte) memory segments that the words to be accessed lie in when each thread accesses an 8-bit (16-bit, 32-bit, 64-bit) word. Given the cost of a global memory transaction, it pays to organize the computation so that the number of global memory transactions made by each half warp is minimized.
3) Shared memory is divided into 16 banks in round robin fashion using words of size 32 bits. When the threads of a half warp access shared memory, the access is accomplished as a series of 1 or more memory transactions. Let S denoted the set of addresses to be accessed. Each transaction is built by selecting one of the addresses in S to define the broadcast word. All addresses in S that are included in the broadcast word are removed from S. Next, upto one address from each of the remaining banks is removed from S. The set of removed addresses is serviced by a single memory transaction. Since the user has no way to specify the broadcast word, for maximum parallelism, the computation should be organized so that, at any given time, the threads in a half warp access either words in different banks of shared memory or they access the same word of shared memory.
4) Volkov et al. [18] have observed greater throughput using operands in registers than operands in shared memory. So, data that is to be used often should be stored in registers rather than in shared memory.

5) Loop unrolling often improves performance. However, the #pragma unroll statement unrolls loops only under certain restrictive conditions. Manual loop unrolling by replicating code and changing the loop stride can be employed to overcome these limitations.
6) Arrays declared as register arrays get assigned to global memory when the CUDA compiler is unable to determine at compile time what the value of an array index is. This is typically the case when an array is indexed using a loop variable. Manually unrolling the loop so that all references to the array use indexes known at compile time ensures that the register array is, in fact, stored in registers.

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}, $1 \leq j \leq m$ 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 $fieldsArray[i][j] = f_{ij}$, $1 \leq j \leq m$, $1 \leq i \leq n$. 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 [8] and is essentially the same as that used by the GPU radix sort algorithm of [15]. When the fields array is mapped to memory in row-major order, we get the $ByRecord$ layout of [2]. A third layout, $Hybrid$, is employed in [12]. 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. We observe that when $m < 2$, the $ByField$ and $Hybrid$ layouts are identical. When the sort begins with data in a particular layout format, the result of the sort must also be in that layout format. Our primary focus in this paper is the $ByField$ layout. However, to make an apples to apples comparison with SRTS [12], which is the fastest known radix sort for integers and records with a single 32-bit field, we make a simple extension of our $ByField$ algorithm to sort when the $Hybrid$ layout is used.

At a high level, there are two very distinct approaches to sort multifield records. In one, 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 [7]. 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. There are advantages and disadvantages to which strategy should be employed to sort records with multiple fields. First strategy seems to perform much less work than the second during sorting as the satellite data that needs to be moved with the key is only an integer index while in the second strategy its the entire record. On the flip side, the first strategy has a very costly random global memory access phase at the end when the records are moved to their sorted positions whereas the second strategy does not have this phase.

IV. Sample Sort [11] For Sorting Records

Sample sort 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 a two-way approach. 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 buckets have values which 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 sorting algorithm is used to sort the small bucket. Sample sort might be difficult to parallelize efficiently as the size of the buckets assigned to the thread blocks can vary greatly depending on the splitters chosen. However, this problem is mitigated by choosing the splitters from a large sample of random numbers.

The parallelized version of sample sort on the GPU is done in multiple iterations. In each iteration we look at the elements assigned and distribute them into k buckets. The distribution consists of 4 kernel launches.

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 out from those random samples. Finally, splitters are sorted using odd-even merge sort in shared memory [**?**] and a Binary Search Tree 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 indices 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: Per block k-entry histograms are prefix-summed to obtain the global offset for each bucket.

Phase 4: Each thread in this phase again calculates the local indices and local offsets. Local offsets are added to the global offsets from the previous phase to get the final position of the element.

While sorting records following the second strategy outlined in SectionIII the records need to be moved during the fourth phase as in all other previous phases only the keys are required.

This distribution of the records from the large bucket to small buckets is repeated multiple times till the size of the bucket is below certain threshold. Records are correspondingly moved during these iterations. Finally, quicksort is done on the records when the bucket size if small. Records are also moved during partitioning phase of the quicksort within a small bucket. As there are potentially many times during the execution of the sample sort records are moved first strategy of doing a (key, index) pairs is better than the second strategy of moving the entire record every time a key is being moved. 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 $rec[i]$ to $outRec[j]$, threads can move the corresponding fields as shown in Figure 3

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

Fig. 3: moving records in $ByField$ layout

Similarly, in $ByRecord$ ($Hybrid$) format fields can be moved similarly by a thread while moving the keys. Figure 4 shows the code to move records assuming the records $rec[i]$ and $outRec[j]$ are structures containing the values.

```
outKey[j] = key[i];
//Move the fields
outRec[j] = rec[i];
```

Fig. 4: moving records in $ByRecord$ layout

We observe that in the $ByField$ layout, each thread accesses the consecutive elements in the fields array which results in generating a large global memory transaction while moving records in $ByRecord$ layout each threads reads a record and writes the entire record into the global memory. This reading and writing of records will be broken into global memory transactions of 16 bytes given the compiler optimizes the code. As the records are separated in memory this would generate a set of smaller global memory transaction. We employ a strategy of grouping the threads together so that we can generate a larger memory transaction. 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 threads iterate to read and write other records assigned to them co-operatively. This ensures more 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 these thread group iterate over to read other records until they are finished reading all the records assigned to them. Lets say $numThreads$ denotes the number of threads in a block and each thread is supposed to read in one record and put it into proper place in the output array. Hence, we can 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 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 5 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. 5: optimized version of moving records in $ByRecord$ layout

V. SRTS [12] For Sorting Records

SRTS employs an highly optimized version of the scan kernel developed by Merrill and Grimshaw[**?**] to perform the entire radix sort. As with the other radix sort strategies it progressively radix sorts on 4 bits during the each phase. Hence, SRTS requires 8 of these iterations to completely sort a set of 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. Most of the kernels in radix sort implementation are memory bound. The technique introduced in SRTS increases the arithmetic intensity of these memory bound operations and eliminates the need for additional kernel for sorting as indicated in SDK radix sort by citesat. 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. It consists of three phases.

Phase 1: Bottom Level Reduction This phase consists of two sub-phases. During the first phases, each thread reads in its element from the input data and extracts the value at the current digit place and increases the digit counts correspondingly. The digit counts are accumulated in local registers as there are a possible of 16 different digit values for 4 bits. The threads in the thread block loops over all tiles assigned to the thread block and the digit counters are accumulated in the local registers. After the last tile of input data is processed, the threads within a block perform a local reduction cooperatively to reduce the sequence of counters and the result is written to the global memory as a set of partial reductions.

Phase 2: Top Level Scan In this phase a single block of threads operate over the partial reductions to find out the global prefix sum. The scan is modified to handle a concatenation of already partially reduced sums.

Phase 3: Bottom Level Scan Lastly, the thread in a block enact sorting of the elements by first reading in the prefix sum calculated during the top-level scan phase. It reads in the elements again and extracts the bits corresponding to the current digit place. 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 first scattered in shared memory using the local offsets to put them in sorted order within the tile. Finally, those elements are read in order from the shared memory and written onto the global memory to ensure better memory coherence generating larger global memory transactions. The aggregate count of the digits 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 Figure 3 and Figure 4 to scatter the fields in $ByField$ and $ByRecord$ layouts respectively. However, due to the way SRTS is implemented using generic programming its hard to use the an optimized version of record moving (Figure 5 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, second strategy of moving records while sorting might actually perform better than the first strategy of indirect sorting followed by rearrangement.

VI. GRS [1] For Sorting Records

GRS is developed specifically for sorting records [1] along the lines of the SDK radix sort [15] but the focus is to reduce the number of times a record is read or write into the global memory. It employs an additional storage to store $rank$ of the elements and gets rid of the sorting phase proposed in SDK sort [15]. The $rank$ of an element is defined as the number of elements having the same digit value before the element in the input data tile. 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. It 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 entitles us to use the optimized version for moving records $ByField$ and $ByRecord$ layouts detailed in Figures 3 and 5 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

VIII. Conclusion

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] Blelloch, G.E., Vector models for data-parallel computing. Cambridge, MA, USA: MIT Press, 1990.

[4] Cederman, D. and Tsigas, P., GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors, *ACM Journal of Experimental Algorithmics*(JEA), 14, 4, 2009.

[5] 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.

[6] Greb, A. and Zachmann, G., GPU-ABiSort: optimal parallel sorting on stream architectures, *IEEE International Parallel and Distributed Processing Symposium* (IPDPS), 2006.

[7] Horowitz, E., Sahni, S., and Mehta, D., Fundamentals of data structures in C++, Second Edition, Silicon Press, 2007.

[8] 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.

[9] Knuth, D., *The Art of Computer Programming: Sorting and Searching*, Volume 3, Second Edition, Addison Wesley, 1998.

[10] Le Grand, S., Broad-phase collision detection with CUDA, GPU Gems 3, Addison-Wesley Professional, 2007.

[11] Leischner, N., Osipov, V. and Sanders P., GPU sample sort, IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.

[12] Merrill, D and Grimshaw A, Revisiting Sorting for GPGPU Stream Architectures, *University of Virginia, Department of Computer Science*, Technical Report CS2010-03, 2010.

[13] Lindholm, E., Nickolls, J., Oberman S. and Montrym J., NVIDIA Tesla: A unified graphics and computing architecture, *IEEE Micro*, 28, 3955, 2008.

[14] Sahni, S., Scheduling master-slave multiprocessor systems, *IEEE Trans. on Computers*, 45, 10, 1195-1199, 1996.

[15] Satish, N., Harris, M. and Garland, M., Designing Efficient Sorting Algorithms for Manycore GPUs, *IEEE International Parallel and Distributed Processing Symposium* (IPDPS), 2009.

[16] Sengupta, S., Harris, M., Zhang, Y. and Owens, J., D., Scan primitives for GPU computing, *Graphics Hardware 2007*, 97-106, 2007.

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

[18] Volkov, V. and Demmel, J.W., Benchmarking GPUs to Tune Dense Linear Algebra, *ACM/IEEE conference on Supercomputing*, 2008.

[19] Won, Y. and Sahni, S., Hypercube-to-host sorting, *Jr. of Supercomputing*, 3, 41-61, 1989.

[20] 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.

[21] CUDPP: CUDA Data-Parallel Primitives Library, *http://www.gpgpu.org/developer/cudpp/*, 2009.

[22] NVIDIA CUDA Programming Guide, *NVIDIA Corporation*, version 3.0, Feb 2010.