2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

GPU Performance Enhancement via Communication Cost Reduction: Case Studies of Radix Sort and WSN Relay Node Placement Problem

Che-Rung Lee, Shih-Hsiang Lo, Nan-Hsi Chen, Yeh-Ching Chung
Department of Computer Science, National TsingHua University
HsinChu, Taiwan
cherung@gmail.com, albert@sslab.cs.nthu.edu.tw, senkrad1101@hotmail.com, ychung@cs.nthu.edu.tw

I-Hsin Chung
IBM T.J.Watson Research
Yorktown, NY
ihchung@us.ibm.com

Abstract—**As the computational power of Graphics Processing Unit (GPU) increases, data transmission becomes the major performance bottleneck. In this study, we investigate two techniques, data streaming and data compression, to reduce the communication cost on GPU. Data streaming enables overlap of communication and computation, whereas data compression reduces the data size transferred among different memory spaces. Although both techniques increase computation cost, overall performance can still be enhanced by reducing communication cost. We demonstrate the effectiveness of the two techniques via two case studies: radix sort and 3-star, a deployment algorithm in wireless sensor networks. For radix sort, a new algorithm, which mixes MSD and LSD algorithms and employs data streaming, is presented. Its performance is 25% faster than the fastest GPU radix sort implementation currently available in the public domain. For the 3-star algorithm, the speed increases several hundreds of times faster than that obtained by the CPU code. The data streaming and data compression, which is a hybrid CPU-GPU algorithm, provide an additional 54% performance improvement to the GPU implementation. Data compression not only reduces communication cost, but also improves the computation time, by which further performance enhancement can be achieved.**

Keywords- GPU, data compression, data streaming, radix sort, wireless sensor networks.

I. INTRODUCTION

With the advance of graphics hardware technology, programming and executing general applications on Graphics Processing Units (GPUs) is more feasible. Nowadays, a single GPU with hundreds or even thousands of processing elements has great potential for improving the performance of various computational intensive applications. To harness the massive computational power of GPUs, programmers must explore the parallelism of applications and must utilize hardware resources efficiently. A continuing challenge is reducing the communication cost among different levels of memory spaces.

Many strategies for reducing the overhead of memory access in GPUs have been investigated. Vectorization and memory coalescing are two programming techniques commonly used to reduce the cost of memory access on GPUs [1][2][3][4][5]. Another common approach is massive fine-grained threading, which improves processor utilization and hides the communication latency [6]. This approach is also effective for computing-bound applications, such as matrix-matrix multiplication. Algorithmically, communication avoiding methods [7][8] and cache-oblivious algorithms [9] are also designed to minimize communication cost. The underlying concepts of these algorithms have also been used to enhance performance in GPU programming. In the application level, various communication reduction techniques that use the domain knowledge of data properties have been reported in the literature [10][11]. In [12], a CUDA API called Dymaxion is provided to optimize memory mapping for programmers.

The data streaming and compression techniques introduced in [13] reduces the communication overhead for output data. Here, the techniques are generalized and applied to other problems. The principle of data streaming in input and output is to enable overlapping communication and computation. Doing so requires effective data partitioning and processing. For applications run on the GPU, this is usually not problematic since tiled or block algorithms are designed to partition and process data to enable efficient use of the limited shared memory. Once a block of computation is complete, the transmission of its output and the computation of the next block can be executed simultaneously.

However, overlapping communication and computation may be insufficient for hiding the latency of communication because of the large performance gap between communication and computation. Therefore, another common rescue technique is data compression. Although data compression is widely used in data management to reduce storage space and network bandwidth requirements, applying compression to enhance GPU performance raises different concerns. First, although complex compression methods may improve the compression ratio, they may not be effective when implemented in GPUs. The selected compression method should optimize overall performance even if does not achieve the best compression ratio. Second, the data sender and receiver are CPU and GPU, or vice versa, which have different architectural characters.

Designing asymmetric compression schemes that enable both processor types to perform different computation simultaneously is a continuing challenge in the GPU and CPU computation.

Nevertheless, the processor-coprocessor relationship between CPU and GPU does give advantages to compress certain data, such as index. Since the data have been partitioned in the streaming process, the output can be re-indexed based on the data in the same partition. Although re-indexing reduces index size, it requires extra computation by both the GPU and CPU. However, because of the extreme simplicity of this coding/decoding approach for compression/decompression, both computations can be hidden in the streaming process. As a result, the compression still improves overall performance.

This study demonstrates how data streaming and compression can be performed in two case studies: the radix sort and a sensor deployment algorithm in wireless sensor networks, called 3-star. The radix sort [14][15][16] outputs a permutation of N input data (finite bit integers) that arranges them in descending or ascending order. The 3-star algorithm [17] is designed to solve a sensor deployment problem in wireless sensor networks (WSNs). Given N sensors on a plane, a 3-star is 3 sensor nodes whose pairwise distance is larger than a given constant R, but the radius of their circumscribed circle is less than R. The center of the circumscribed circles of the 3-star is a potential location to place a special sensor, called a relay node.

The rest of this paper is organized as follows. Section II briefly introduces the GPU architecture, the background of radix sort and the 3-star algorithm. Section III presents how the data streaming and compression techniques are carried out to the two case studies. Section IV gives the experimental results of the studied cases. The last section concludes the study and proposes future works.

II. Background

This section briefly introduces the background of various topics, including the GPU architecture, the radix sort algorithm, and the 3-star algorithm in WSNs.

A. GPU Architecture

The GPU used in this study is a CUDA-enabled device [18] with an array of multiprocessors and various memory spaces. The hardware execution unit in the CUDA device is called a warp, i.e., a group of threads. All threads in a warp execute the same instructions synchronously. Each multiprocessor can execute one or more warps concurrently. Processed data could be placed in different levels of memory hierarchy in the device, including registers, shared memory, cache, constant memory, texture memory and global memory.

A typical CUDA program is executed as follows. After data are copied from the host (CPU) memory to the device (GPU) memory, the host invokes a kernel function with the thread configuration, such as the thread block size, *etc*. When a kernel function is executed, all thread blocks are distributed to multiprocessors, which then arrange the thread blocks into warps and schedules them for execution. When the kernel function is completed, the host copies data from the device memory to the host memory.

Several performance concerns arise when programming in CUDA. First, threads should access data in the low latency memory (*e.g*., shared memory or cache) rather than in the high latency memory (*e.g*., global memory). Even if threads require data access from the global memory, coalesced memory access can reduce the number of memory transactions. Second, threads in a warp should avoid different execution paths, which would cause warp divergence, *i.e*., threads within a warp must perform different execution paths sequentially.

The latest CUDA architecture, Fermi, allows concurrent kernel execution [19], which means multiple kernel functions can be executed simultaneously on the different multi-processors. In this study, we will not consider this ability.

B. Radix sort

Radix sort, a non-comparative sorting algorithm for integers or finite bits data, requires only O(kN) time to sort N data, where k is the number of bits of data [14]. Radix sort functions as follows. First, it represents the data in a specific radix. This conversion can be done implicitly by dividing the finite bits of data into fixed length groups if the radix is of a power of 2. The data are then hashed repeatedly into buckets according to their digits, starting from the most significant digit (MSD) or from the least significant digit (LSD), until all digits are processed. During each hashing, conflicting data are stored in a queue; after hashing is complete, those queues are concatenated into a long list, from the beginning of which the next hashing begins.

The major challenges of parallelizing radix sort queuing hashed data and concatenating all queues into a list. The prefix sum or scan technique is typically performed to solve both problems [20]. Given an array L of numbers, the prefix sum returns a list M of same size, where $M[i]$ equals the sum of $L[1]$ to $L[i\text{-}1]$. Notably, parallel prefix sum of N elements can be computed by using N processors in O (log N) time.

Radix sort implemented on GPUs has been studied intensively because of its technical importance. Radix sort is the fastest (non-comparative) sorting algorithms on GPU. In [20], an efficient parallel scan method on GPU was proposed, and one of its applications is radix sort. In [15], the authors presented fast sorting algorithms, including radix sort and merge sort. In [16], Merrill and Grimshaw presented a tuned radix sort GPU implementation, which is integrated into the Thrust package. The three-step process they proposed for each digit is: upward reduce, spline scan, and downward scatter. Their algorithm has been integrated to the state-of-the-art package, Thrust [21].

C. Relay Node Placement Problem

A WSN consisting of a set of sensors is used for sensing desired data and reporting them to a base station. The sensors transmit data through the embedded wireless device with limited effective transmission range *R*. Advanced sensors not only sense and send data, they also forward data for neighboring sensors. This study considers the sensors capable of sensing, sending data, and forwarding data. Given a set of sensor nodes on a plane, which may or may not be fully connected, the problem of interest in this study is how to deploy additional sensors (relay nodes) so that all sensors form a connected network. Each sensor is modeled as a point.

Under this setting, the problem of deploying the minimum number of relay nodes to connect all sensors is called the Single Tier Relay Node Placement problem (STRNP), which turns out to be NP-hard [17][22][23]. A simple approximation algorithm is to construct the minimum spanning tree (MST) of the sensor nodes first, and then to place relay nodes on the edges of the MST such that no resulted edge is longer than *R*. This algorithm has an approximation ratio of 5 [23]. A better algorithm in terms of approximation ratio is called the 3-star algorithm [17]. A 3-star is a set of 3 sensor nodes, whose mutual distance is larger than *R*, but the radius of their circumscribed circle is less than *R*. Therefore, the three sensors can be connected by placing a relay node at the center of their circumscribed circle, instead of putting two relay nodes on the edge of a triangle. The 3-star algorithm places the relay nodes in the circumscribed center of potential 3-stars, and then runs the MST algorithm again to connect unconnected components. The approximation ratio of the 3-star algorithm can be proven to be 3 [23].

The most time consuming task of the 3-star algorithm is finding candidate of 3-stars, which takes $O(N^3)$ time to check all 3-tuple of sensor nodes. This study only considers the problem of how to find all potential 3-stars.

A literature search shows that no studies propose a GPU implementation of the 3-star algorithm.

III. Data Steaming and Compression Techniques

A. General strategy

A GPU program generally consists of three steps:

1. Import data from CPU memory to GPU memory,
2. Invoke GPU kernel function to compute, and
3. Export result from GPU memory to CPU memory.

The data streaming under this framework is to overlap those three steps. To overlap the first and the second steps, one requires the computation can be preceded on partial data and regardless their content, *e.g.*, matrix-matrix multiplication. Preprocessing raw data is necessary if the computation is content-dependent. Overlapping of the second and the third step is much simpler and requires a much simpler condition, *i.e.*, data can be partitioned and processed separately. To simplify this discussion, only the latter problem is discussed.

Suppose the data is partitioned into D_1, D_2, …, D_n. The data streaming of the second and the third step overlaps the computation of D_{i+1} with the output process of D_i. The algorithm of output streaming is outlined in Figure 1.

In Figure 1, the pair of clauses `Parallel Do/End Parallel Do` indicates the tasks between them can be processed at the same time. The terms `CPU` or `GPU` are added to the front of each statement to specify which device is responsible for the task. In CUDA, the `Output` command corresponds to the function call,

```
cudaMemcpy(,,,cudaMemcpyDeviceToHost)
```

and the `Compute` command corresponds to the invocation of kernel functions. Overlapping of communication and computation requires the asynchronous function, `cudaMemcpyAsync(,,,)`.

```
GPU: Compute D[1];
For i = 2,..., n
   Parallel Do
      CPU: Output D[i-1];
      GPU: Compute D[i];
   End Parallel Do
End For
CPU: Output D[n];
```

Figure 1. Pseudo code of output data streaming

As stated in the document of NVIDIA CUDA library [24], the `cudaMemcpyAsync()` function is asynchronous with respect to the host, so the call may return before the copy is completed. It requires page-locked host memory and returns an error if a pointer to pageable memory is passed as input. The copy can optionally be associated to a stream by passing a non-zero stream argument. Therefore, for the operation

`cudaMemcpyHostToDevice`
or `cudaMemcpyDeviceToHost`,

the copy of the specific stream can be overlapped with operations in other streams.

The second technique for reducing the communication cost is data compression, which enhances performance by reducing the transferred data size. The pseudo code of output data streaming and compression is given in Figure 2.

```
GPU: Compute and compress D[1];
For i = 2,..., n
  Parallel Do
    CPU: Output and decompress D[i-1];
    GPU: Compute and compress D[i];
  End Parallel Do
End For
CPU: Output and decompress D[n];
```

Figure 2. Pseudo code of output data streaming and compression

B. Radix sort

The state-or-art radix sort implementation on GPU is based on the LSD (least significant digit first) algorithm, which keeps all data integral during the entire computation.

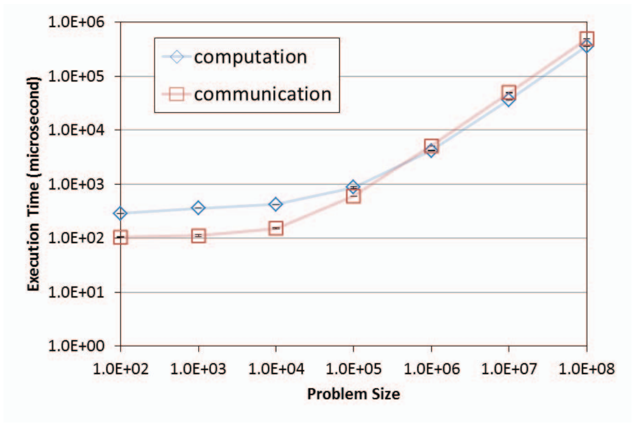

Figure 3. Computation and communication time in the Thrust radix sort implementation.

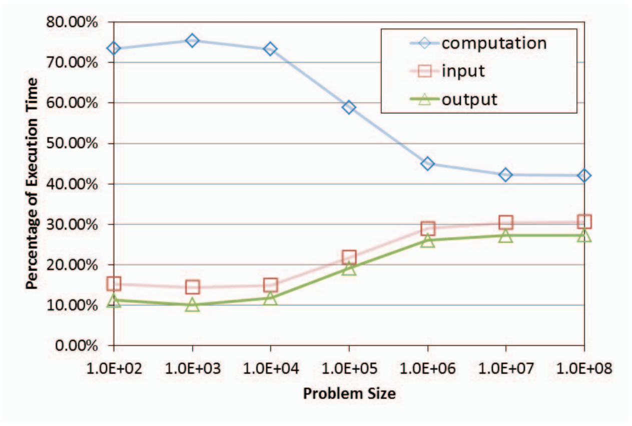

Figure 4. Percentage of time for computation, input, and output in the Thrust radix sort implementation.

However, partitioning data requires the MSD (most significant digit first) algorithm, which functions like the bucket sorting algorithm or the sample sort algorithm. The MSD-based algorithm first hashes data into buckets according to their most significant digits and then sorts data in each bucket recursively. However, the MSD algorithm may slow down when the bucket size is too small or when the bucket number is too many.

The proposed algorithm combines MSD and LSD. After the MSD algorithm partitions the data to a sufficiently small size, the LSD algorithm sorts each partition. Figure 3 displays the profiling results of the LSD algorithm reported in [16]. The computation and communication time for data size from 10^2 to 10^8 is given, where communication time include the transmission of key-value pairs. As can be seen, the computation time levels up after $N=10^6$, which indicates the stream microprocessors in GPU are not fully utilized until $N \geq 10^6$. Therefore, subdividing the data is unnecessary when N is smaller than 10^6.

The size of the output data for radix sort is not massive. However, when the data size scales up, the time of communication still exceeds the time of computation, as shown in Figure 3. Figure 4 shows the percentage of time required for input and output. The figure shows that, although the total communication time consumes over 50% of the total time, the time of output alone is still shorter than that of computation. Therefore, the output data need not be compressed for radix sort.

The major problem of applying data streaming is the load balance of each data partition. For uniformly distributed data, all the buckets are of approximately equal size after applying the MSD algorithm. Generally, however, the size of buckets tends to be highly unbalanced. The proposed solution is to make the MSD algorithm be a separated kernel function and to allow CPU to perform load balancing scheduling based on the returning results of the MSD algorithm. For a large bucket, the MSD algorithm is again used to split the bucket into smaller ones. For small buckets, if they are in a sequence, then we can merge them together and sort them as a bigger bucket. In Figure 5, for example, bucket 2 is too large and buckets 3 to 8 are too small. After calling the MSD algorithm, the CPU calls it again to split bucket 2, to merge the data in bucket 3, 4, 5 into one LSD call, and to merge the data in bucket 6, 7, 8 into another LSD call.

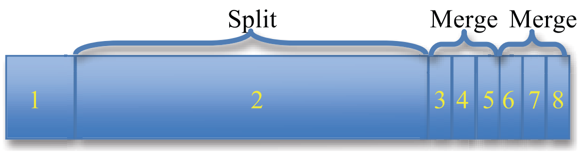

Figure 5. Example of procedure for balancing bucket sizes after calling the MSD algorithm.

C. The 3-star algorithm

The problem of finding 3-stars provides a natural partition method. Recall that a 3-star is defined as 3 vertices of a triangle where the edges of the vertices are all larger than a constant R, but the radius of their circumscribed circle is smaller than or equals to R. Figure 6 shows the possible search space (the donut shaped area) for a given sensor node. Suppose a center node is located at the center of the circle. The donut-shaped shading area represents the possible search spaces for the other two vertices of a 3-star. Therefore, if the space is partitioned by a fixed-size grid, where the grid size is $2R$, only the sensor nodes in nine neighboring grid cells need to be checked.

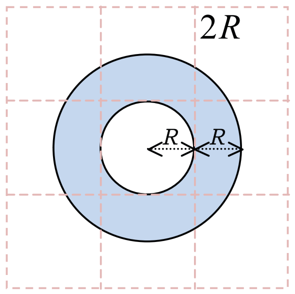

Figure 6. Possible searching space (shaded area) for the 3-stars with a vertex located in the center of circle.

Although this computation can be performed in parallel on GPU in many different ways, the method used here is to let each thread block to take of one grid cell and to check the 3-stars that include at least care of one vertex in the grid cell. Each thread is responsible to a sensor node and searches all possible combinations of other two sensors nodes in the nine neighboring grid cells. To avoid repeated checking, each grid cell is indexed by the pair of its row index and column index as shown below.

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

When a thread block checks the grid cell of index (r, c), the only grid cells it needs to check are (r, c), (r +1, c), (r -1, c +1), (r, c +1), and (r, c +1). For instance, if a thread block is assigned to the grid (2, 2), the grid cells it needs to check are (1, 3), (2, 2), (2, 3), (3, 2), and (3, 3).

This approaches reduces the number of possible combinations of grids that need to be checked to 5×5=25 since only one of the sensor nodes in (2, 2) is required, and the other two can be in any of the neighboring 5 cells. The number of grids that need to be checked can be reduced by eliminating impossible combinations. For instance, three sensor nodes in cell (2, 2), (1, 3), and (3,2) cannot form a 3-star because the distance between (1, 3) and (3, 2) is larger then 2R. Doing so can reduce the number of combinations to 13.

Checking a 3-star for three given sensor nodes only requires comparison of the radius of their circumscribed circle to R. Let a, b, c be the length of the triangle formed by three sensor nodes, and let s = (a+b+c)/2. The radius of their circumscribed circle can be computed by

$$r = \frac{abc}{2\sqrt{s(s-a)(s-b)(s-c)}}. \tag{1}$$

However, use of this formula requires computation of a, b, c before computation of r. The distance d between two points (x_1, y_1) and (x_2, y_2) can be computed by

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{2}$$

Both computations require numerous invocations of floating point operations such as square and square root, which are computationally expensive. To avoid floating point operations, a lookup table is used to retrieve a, b, and c from the distance pair ($|x_1{-}x_2|$, $|y_1{-}y_2|$), since all coordinates are all integers and their differences are bounded.

After obtaining the distance a, b, and c, one can use another lookup table to check the feasibility of 3-star. Since the ranges of a and b are finite, they can be arranged into a 2-dimentional table. The upper bound and the lower bound of c that can make three sensors a 3-star is given in this table. Therefore, checking the range of c and the actual length of the third edge reveals whether the given sensors are 3-star. To speed up the computation, all table entries are pre-computed and loaded to the texture memory.

When outputting 3-stars, the IDs of three sensor nodes must be reported, which requires 12 bytes for three integers. The used technique for reducing sizes of IDs is to compress the output, which divides the IDs into three parts: Cell ID (CID), Segment ID (SID), and Local ID (LID).

$$\text{(CID, SID, LID)}. \tag{3}$$

Since the report of 3-stars in the same cell and the same segment is aggregated together, only the local IDs are required in most cases. Additionally, since there are only 13 possible cases, the CPU can know the CID of all three sensor nodes. Figure 7 illustrates the format of the compressed indices for 3-stars in a thread block.

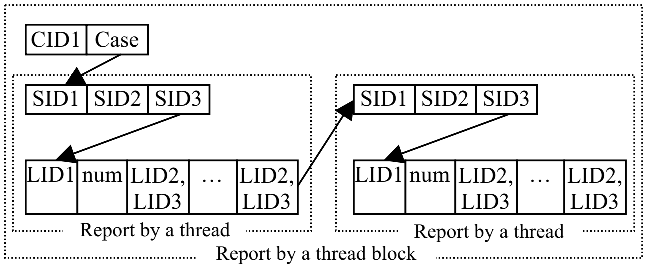

Figure 7. List of 3-stars in the compression form.

IV. EXPERIMENTS AND RESULTS

A. Experimental environment

The experimental platform has one Intel i7 920 processor 2.67 GHz with 6 GB DRAM. The OS is 64-bit Linux, kernel version 2.6.27. The GPU used in the experiments is a Tesla C2050, which contains 448 CUDA cores and 3GB GDDR5 memory. The host programs use GCC 4.3 and enable optimizations, –O3 and –DNDEBUG. The device programs use CUDA compiler driver 4.0. The execution time recorded for each test case was the average obtained in 10 runs.

B. Radix Sort

After modifying the radix sort program in the Thrust package, its performance was compared with that of the original code. The radix sort programs were tested using two different data distributions: uniform and normal. The radix in use is 16.

For uniformly distributed data, tests were run for program size of 10^4, 10^5, 10^6, 10^7 and 10^8. Table I compares the performance of the original radix sort with that of the modified program for different program sizes. The unit of time is microseconds. The comparison shows that the performance improvement obtained by the modified program increases with program size. The time required to output data can be reduced to 6% (speedup is 16). Overall, the modified program decreases computation time by 20% and 25% for program sizes 10^7 and 10^8, respectively, but it increases computation time for other program sizes due to the overhead of performing LSD calls many times.

TABLE I. PERFORMANCE COMPARISON OF THE ORIGNAL AND THE MODIFIED RADIX SORT PROGRAM FOR 10^4, 10^5, 10^6, 10^7, AND 10^8 UNIFORMLY DISTRIBUTED DATA (MICROSECOND)

Program Size	Original	Modified
10^4	674	3852
10^5	1564	4380
10^6	9505	11403
10^7	85948	70769
10^8	850778	644498

TABLE II. PROFILE OF RADIX SORT PROGRAMS FOR 10^8 UNIFORMLY DISTRIBUTED DATA (MILLISECOND)

Operation	Original	Modified
Reduction	34.91	38.35
Spline-scan	0.19	2.77
Scatter	316.75	318.05
MemcpyHtoD	261.18	262.11
MemcpyDtoH	238.49	241.62
Bucketing and alignment	0.00	0.08
Overlap of output data	0.00	222.30
Total time	851.52	640.67

Table II compares the profiles of the original radix sort and the modified radix sort programs for 10^8 uniformly distributed integer data. The unit of time is milliseconds. The first three operations, *reduction*, *spline-scan*, and *scatter*, are major operations in the radix sort algorithm. The next two operations, *MemcpyHtoD* and *MemcpyDtoH*, are data movements from the host to the device and from the device to the host, respectively. As can be seen, except for spline-scan, all operations of the modified program are as fast as those of the original program. The purpose of the spline-scan is to calculate the offsets of the numbers within a block. Because the MSB algorithm is invoked once and the LSB algorithm is invoked 16 times, the time spent on spline-scan in the modified program is about 15 times of the original program, which is roughly equal to the ratio of the number of calls made by both programs (17 times. Although the spline-scan is much slower in the modified program, it occupies only a small fraction of the entire computation. Therefore, the additional overhead is negligible.

The next two operations are performed only by the modified program. After invoking the MSB algorithm, Bucketing and alignment operation divides the data into smaller chunks (buckets) and aligns the data in each bucket to 4 integers. A comparison shows that the cost of the operation is low compared to other operations. The last operation is the overlapping of the output data, which is performed by data streaming. Since computation time (reduction+scan+scatter) is longer than the time required for MemcpyHtoD, most of the time spent on data output can be hidden by computation. Additionally, data output consumes about one third of the entire time, the overall improvement is significant.

As explained in Section III.B, normal distributed data requires an additional load balance operation to ensure that the buckets are of approximately equal size. Essentially, the load balance operation subdivides the large buckets into smaller ones by calling the MSB algorithm again. Table III lists the profiling results of the original program and the modified program for 10^8 normally distributed data. The table shows that the time spent on the load balance operation is negligible. However, the increased number of buckets and the smaller bucket sizes result in a 50-fold increase in spline-scan time. The time required for the reduction operation also increases substantially. Fortunately, overlapping of data output enables a 21% reduction in overall time.

TABLE III. PROFILE FOR RADIX SORT PROGRAMS FOR 10^8 NORMALLY DISTRIBUTED DATA (MILLISECOND)

Operation	Original	Modified
Reduction	34.86	47.32
Spine scan	0.19	10.51
Scatter	291.04	296.96
MemcpyHtoD	261.17	262.09
MemcpyDtoH	238.49	242.27
Bucketing and alignment	0.00	0.37
Load balance	0.00	0.02
Overlap of data output	0.00	213.89
Total time	825.76	645.64

C. *The 3-star Algorithm*

The computation of the 3-star algorithm is similar to that required to solve the rectangle intersection problem [13]. Even when the input size is not large, the potential output can be enormous. Figure 8 illustrates a sensor placement scenario that have O(N^3) 3-stars, in which sensors are located at one of three clusters evenly. Each sensor node in a cluster can match two other sensor nodes at two other clusters.

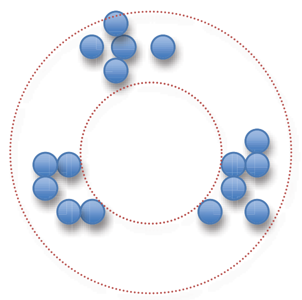

Figure 8. Sensor placement scenario that have O(N^3) 3-stars.

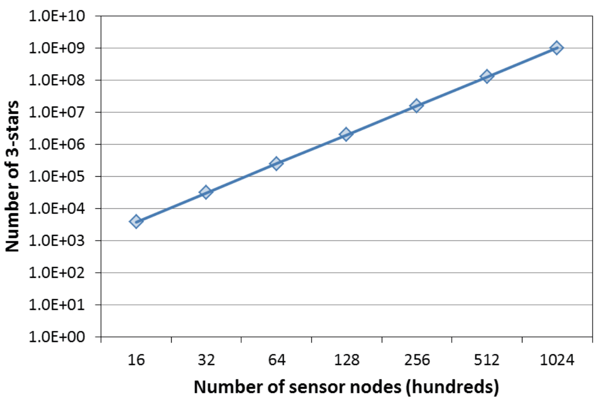

Figure 9. Relation between the number of 3-stars and the number of sensor nodes on a 2000×2000 plane.

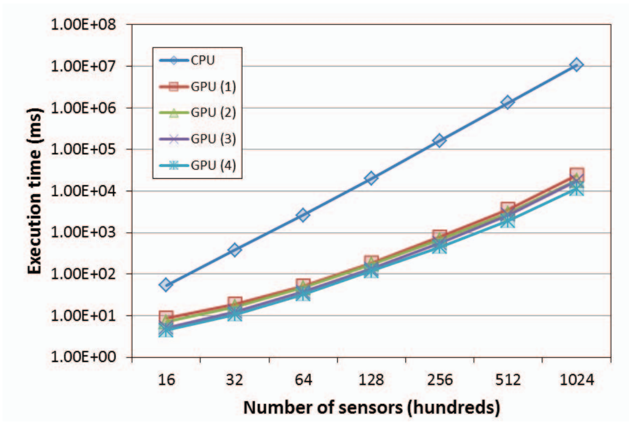

Figure 10. Performance comparison of CPU and GPU implementations on a 2000×2000 plane.

In practice, however, sensors are rarely deployed in this manner. Even if they were placed in this manner, relay nodes are easily located at the center of three clusters. Therefore, the experiments in this study did not consider extreme cases to demonstrate the power of data streaming and compression, even though an excellent performance improvement was ensured. Instead, this study considers the more likely scenario in which sensors are uniform-randomly placed on a 2000×2000 plane where the sensor communication radius is set to 50. The number of sensor nodes was varied from 1,600 to 102,400. Figure 9 displays the relation between the number of sensor nodes and the number of 3-stars in logarithmic scales. The figure shows that the number of sensor nodes grows cubically with the number of sensor nodes.

The following four GPU implementations are presented:

(1) Baseline
(2) Baseline + streaming
(3) Baseline + compression
(4) Baseline + compression + streaming.

Figure 10 compares the performance of a sequential CPU code and four GPU implementations for various data sizes. The sequential code is executed on one Intel i7 920 core with frequency 2.67 GHz, and the GPU code is executed on 448 CUDA cores with frequency 1.15GHz. The CPU implementation is also optimized by performing data partitioning and cell reduction, as mentioned in III.C, but not by using the table lookup. The performance test shows that the GPU (4) implementation obtains a nearly 1000-fold speed increase for N=102,400. If the frequency of processing core and the number of cores are taken into account, the GPU implementation is about 5 times more efficient than the CPU implementation.

Figure 11 shows the profiling results for four GPU implementations for N=102,400, in which the time of five major tasks, LOAD, COMP, REPORT, DataOut, and DECODE, are compared. From the figure, one can see that the performance bottleneck is computation, COMP. Comparison of the second GPU implementation (baseline+streaming) to the baseline implementation shows no data streaming advantage. In fact, REPORT time is increased in the second implementation. Although DataOut performance improves, the improvement is insignificant after compensating for the loss in REPORT. The overall performance improvement achieved by data streaming approximates 26%.

Data compression obtained interesting results. The data compression was expected to substantially reduce the cost of data output. Although OUTPUT and DataOut time decrease, LOAD and COMP time also decrease at a certain ratio, which is unexpected since the output data is compressed after LOAD and COMP, not before.

This phenomenon actually results from the different size of declared local variables. Without data compression, 1024 integers must be temporarily allocated for temporary storage of output data. When the data compression is applied, only 256 integers are required to hold the data. This small difference affects the performance since scheduling the thread blocks in the NVIDIA GPU depends on the allocated resources. If substantial resources such as registers per thread block or per thread block are requested, the number of thread blocks that can be executed simultaneously in one streaming multiprocessor is limited.

In summary, when only the data compression is applied, a 33% performance improvement over the baseline implementation can be achieved. When data compression is cooperated with data streaming, the overall performance improvement can reach as high as 54%.

V. Conclusion and Future Work

This study investigated the effects of two communication reduction techniques, data streaming and data compression, on GPU performance. The two techniques are demonstrated in two applications: radix sort and the 3-star algorithm. The proposed concept is to partition the data, and then use asynchronous memory copy to overlap the computation and communication. Data compression reduces data size during data movement.

This study only focused on the two applications. However, data streaming and compression can be applied to

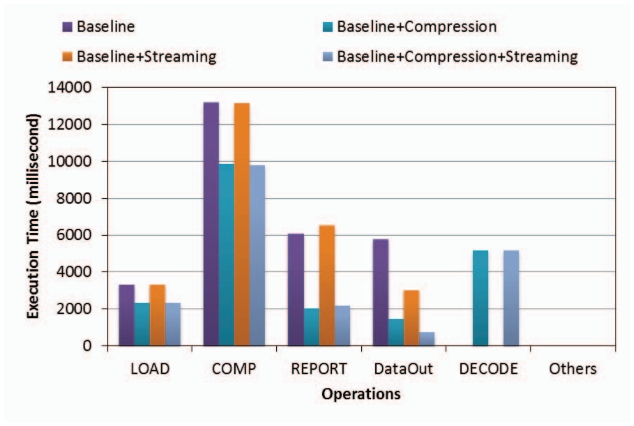

Figure 11. Profiles of four implementations of the 3-star algorithm for N=102,400 on a 2000×2000 plane.

enhance performance in other GPU tasks. We expect the two techniques can be applied to more applications, and have more sophisticated development and usages.

There are several directions in our future work. First, using separate but concurrent kernel functions can enable the use of more complex data compression techniques and improves the compression ratio. Second, by the concurrent kernel execution on GPUs, the computations could also be streamed for enhancing performance. Third, many details, such as load balancing and the side effects of data compression, require further study and analysis.

Acknowledgments

The authors would like to thank the National Science Council of Taiwan for financially/partially supporting this research under Contract No. 100-2218-E-007-013-, and thank the anonymous reviewers for their valuable comments and suggestions. Ted Knoy is appreciated for his editorial assistance.

References

[1] B. Jang, S. Do, H. Pien, and D. Kaeli, "Architecture-aware Optimization Targeting Multithreaded Stream Computing," Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, D.C., 2009.

[2] J. Byunghyun, D. Kaeli, D. Synho, and H. Pien, "Multi-GPU Implementation of Iterative Tomographic Reconstruction Algorithms," in Biomedical Imaging: From Nano to Macro, 2009. ISBI '09. IEEE International Symposium on, 2009, pp. 185-188.

[3] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens, "Efficient Computation of Sum-products on GPUs through Software-managed Cache," Proceedings of the 22nd annual international conference on Supercomputing, Island of Kos, Greece, 2008.

[4] B. Jang, P. Mistry, D. Schaa, R. Dominguez, and D. Kaeli, "Data Transformations Enabling Loop Vectorization on Multithreaded Data Parallel Architectures," Proceedings of the 15th ACM SIGPLAN, 2010.

[5] K. G. Naga, L. Scott, G. Jim, and M. Dinesh, "A Memory Model for Scientific Algorithms on Graphics Processors," IEEE/ACM SC 2006, pp. 6-6.

[6] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," IEEE SC 2008. pp. 1-11.

[7] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-Avoiding QR Decomposition for GPUs," IPDPS 2011.

[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing Communication in Numerical Linear Algebra," UC Berkeley Tech Report EECS-2011-15.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," Proceedings of the 40th Annual Symposium on Foundations of Computer Science, pp. 285-297.

[10] Cole Trapnell, Michael C. Schatz, "Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment", Parallel Computing, Vol. 35, No. 8-9. 2009, pp. 429-440.

[11] Gharaibeh, Abdullah and Ripeanu, Matei, "Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance", SC 2010, pp. 1-12.

[12] Shuai Che, Jeremy W. Sheaffer and Kevin Skadron, "Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems", SC 2011.

[13] Shih-Hsiang Lo, Che-Rung Lee, Yeh-Ching Chung, and I-Hsin Chung, "A Parallel Rectangle Intersection Algorithm on GPU+CPU", CCGRID 2011, pp. 43-52.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, "Introduction to Algorithms", 2nd ed.: McGraw-Hill, 2001.

[15] Nadathur Satish, Mark Harris, and Michael Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," IPDPS 2009

[16] D. Merrill and A. Grimshaw, "High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing," Parallel Processing Letters, vol. 21, no. 2, pp. 245-272, 2011.

[17] Cheng, D.Z. Du, L. Wang and B. Xu, "Relay Sensor Placement in Wireless Sensor Networks", ACM/Springer WINET, Vol. 14, Issue 3 (2008), pp. 347-355

[18] CUDA Programming Guide, 3.2, NVIDIA. Available: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

[19] Tuning CUDA Applications for Fermi. Available: http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_FermiTuningGuide.pdf

[20] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens, "Scan Primitives for GPU Computing," Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS, 2007, pp. 97--106.

[21] Thrust, http://code.google.com/p/thrust/

[22] G. Lin and G. Xue, "Steiner Tree Problem with Minimum Number of Steiner Points and Bounded Edge-length", Information Processing Letters, Vol. 69(1999), pp. 53-57.

[23] D. Chen, D.Z. Du, X.D. Hu, G. Lin, L. Wang and G. Xue, "Approximations for Steiner Trees with Minimum Number of Steiner Points", Journal of Global Optimization, Vol. 18 (2000), pp. 17–33.

[24] NVIDIA CUDA Library Documentation 2.3, http://www.clear.rice.edu/comp422/resources/cuda/html/index.html

[25] R. Rice and J. Plaunt, "Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data," IEEE Transactions on Communication Technology, vol. 19, pp. 889-897, 1971.

[26] NVIDIA Compute Visual Profiler Available: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/VisualProfiler/Compute_Visual_Profiler_User_Guide.pdf.