Scaling Joins to a Thousand GPUs

Hao Gao
NVIDIA
haog@nvidia.com

Nikolay Sakharnykh
NVIDIA
nsakharnykh@nvidia.com

ABSTRACT

Relational join is a fundamental data processing operation that is used ubiquitously in relational databases as well as the extract, transform, load (ETL) stage of machine learning applications. With its high-throughput memory, GPU is well-suited for efficient join operation. However, a single GPU has limited memory capacity, which restricts the scale of applications that can run fully in GPU memory. In this paper, we evaluate a multi-GPU hash join on large-scale GPU clusters. We present a scalable distributed join algorithm, introduce a GPU-friendly compression scheme for decreasing the communication volume, and introduce a two-level shuffle algorithm tuned for modern heterogeneous GPU clusters. On 1024 A100 GPUs, our implementation achieves a total throughput of 667 billion input tuples per second on 16TB random integer dataset without compression, and up to 1.8 trillion input tuples per second with compression when joining “lineitem” and “orders” tables from the TPC-H dataset with scale factor 100k.

1 INTRODUCTION

Nowadays, large amounts of data are collected every day from social media, ecommerce websites, IoT devices, and medical and agricultural equipment. To gain insight into the data, relational join is often featured as a fundamental data processing operation used in relational databases, as well as the extract, transform, load (ETL) stage of machine learning applications.

Performing relational joins on GPUs, while showing great performance, is often limited by the capacity of the GPU memory. One strategy of solving the memory capacity limitation is to spill the dataset to an often larger CPU memory. The performance of spilling depends on the interconnect used between the CPU and GPU memory. If the CPU and GPU memory are connected through a high-bandwidth interconnect like NVLink, spilling can be implemented efficiently. In that case, the performance of the join pipeline is limited by the other factors like computation or GPU memory throughput [10]. However, more often the GPU memory and CPU memory are connected through PCI-e, and the low bandwidth of the PCI-e becomes the bottleneck.

In this paper, we consider an alternative strategy by using a distributed hash-based join algorithm to scale out to a large number of GPUs, connected through a high-throughput low-latency network. It has been shown that distributed join can scale out efficiently on a CPU cluster with such a network [2]. To the best of our knowledge, there is no study to investigate scaling up to a similar massive scale on GPU clusters.

Our contributions in this paper are as follows:

(1) Discuss optimization strategies when implementing distributed hash-join on GPUs.
(2) Present a GPU-friendly compression scheme that can be used to accelerate data transfers.
(3) Evaluate hash-based distributed join with up to 1024 GPUs.

The rest of the paper is organized as follows. Section 2 provides necessary background information. In Sections 3, 4, and 5, we present the distributed, repartitioned join algorithm, with optimizations to make it scalable on modern GPU clusters. Then, we present our performance results and discuss insights in Section 6. Finally, we review related work in Section 7 and conclude in Section 8.

2 BACKGROUND

In this section, we provide an overview of distributed join algorithms, hardware technologies, and software stacks to set the stage for this work.

The critical decision when designing a distributed algorithm is how to break down the work so that each processor can handle a subset of the problem. For distributed join, this means we must partition both tables so that matching elements are assigned to the same partition. Depending on how this partition is performed, distributed join algorithms can be classified into the radix hash join and the sort-merge join. In radix hash join, both input tables are partitioned based on the hash value of the key on each row. After the partition is communicated to the corresponding processor, local join can be implemented through building and probing a hash table. In sort-merge join, both input tables are sorted by keys and partitioned based on key ranges. When the partition arrives at the target processor, local join can be performed by merging the sorted partitions. Previous distributed join studies on CPU [2] and on GPU with small-scale [4] indicate that the radix hash join achieves better raw throughput compared to sort-merge join. The algorithm discussed in this paper is based on the radix hash join.

This paper uses commodity NVIDIA GPUs to evaluate the performance. Each NVIDIA GPU consists of a group of streaming multiprocessors (SMs). Each SM has a group of CUDA cores for execution and a combined L1 data cache and programmable shared memory. Because the shared memory is on-chip, it has much higher throughput and lower latency compared to DRAM. In our distributed hash join implementation, we frequently use shared memory for data reuse, converting random access patterns to streaming ones, or avoiding the costly accesses to the DRAM. We use the CUDA programming language to program GPUs. In CUDA, threads are organized in groups called thread blocks. When executing, each thread block is assigned to a SM, and all threads in the thread block can access the shared memory.

A scalable computing infrastructure must not only have a great number of processors, but also the accompanying computing and

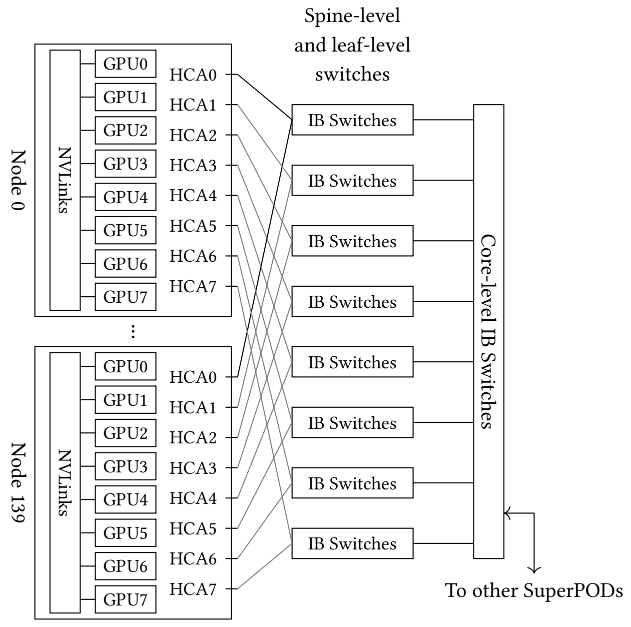


Figure 1: DGX SuperPOD compute-plane network topology

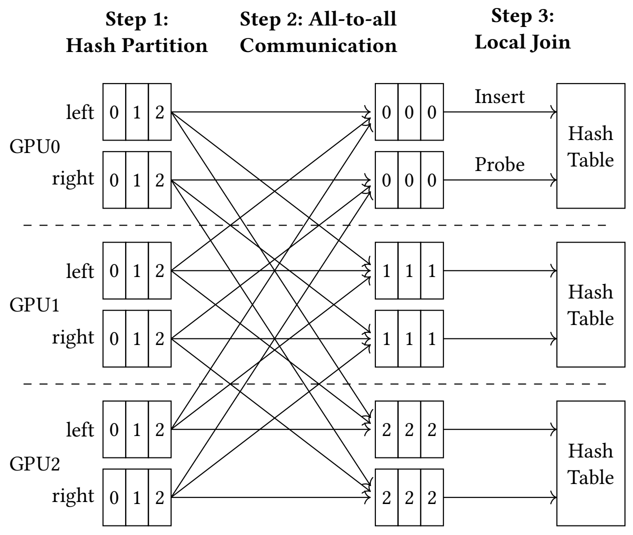


Figure 2: Repartitioned Join Algorithm on 3 GPUs
This figure shows the repartitioned join algorithm on three GPUs. At the start of the algorithm, we assume that the left and right tables are already distributed among GPUs so that each GPU has 1/3 of each table. During the first step, each GPU partitions the table into three partitions using the algorithm discussed in Section 3.1. The number inside the box represents the partition number. Then, each GPU sends partition0 to GPU0, partition1 to GPU1, and partition2 to GPU2, forming an all-to-all communication pattern. Finally, assuming that the left table has fewer rows compared to the right table, each GPU independently constructs a hash table out of all rows in the left table, and probes the hash table with all rows in the right table.

storage network to use these processors efficiently. To achieve this goal, modern GPU clusters are complex, often featuring a heterogeneous interconnect. For example, DGX SuperPOD is the NVIDIA reference architecture for scalable infrastructure. Each DGX SuperPOD has at most 140 nodes. Each node features eight A100 GPUs fully connected through NVLink and also eight Infiniband HDR 200Gbps host channel adapters (HCAs) for internode compute network. Nodes are connected in a full fat-tree topology through three-levels of Infiniband switches: the leaf-level switches, the spine-level switches and the core-level switches. To scale beyond 140 nodes, multiple SuperPODs can be connected together through the core-level switches. The topology is rail-optimized that the same HCAs of all nodes within a SuperPOD are connected through the leaf-level switches and spine-level switches. The core-level switches are used only for traffic across HCAs or across different SuperPODs. Figure 1 illustrates the network topology of a DGX SuperPOD. The complexity of the computing infrastructure poses a challenge for the hash join algorithm to adapt to in order to use the computing resources efficiently.

To communicate device buffers over Infiniband efficiently, we use GPUDirect RDMA. GPUDirect RDMA allows the HCA devices to directly access GPU memory, which improves performance by removing the unnecessary memory transfers to the host memory. Our implementation uses the open source communication library UCX to leverage GPUDirect RDMA.

3 REPARTITIONED JOIN ALGORITHM

Overall, the repartitioned join algorithm follows three main steps. During the first step, we compute the hash value of each row, and partition both input tables according to the hash values into partitions, one for each GPU in the system. After this step, matching rows belong to the same partition. The second step sends each partition to its corresponding remote GPU. Because each GPU must communicate with all other GPUs, this step forms an all-to-all communication pattern. Compression can optionally be used to improve communication performance, which we will discuss in detail in Section 4. After this step, rows with the same keys reside in the same GPU so that each GPU can perform local join independently. In this paper, we call the combination of the hash partition step and the all-to-all communication step the *shuffle* operation. For the last step of the repartitioned join algorithm, each GPU constructs a hash table from the smaller table in terms of the number of rows, and probes the hash table for each row in the bigger table. For both hash partition and local join, we use the murmur3 hash function [1]. Figure 2 shows an example of this algorithm in action with three GPUs. We summarize the algorithm in Algorithm 1.

In the following sections, each of these three steps is discussed in more detail.

3.1 Hash Partition

During the hash partition step, each table is partitioned according to the hash value of each row. The output of this step is a new table of the same size as the input table. In the new table, rows are reordered such that rows with the same hash values are grouped together.

Algorithm 1 Repartitioned join algorithm

```
   Suppose there are $N$ GPUs.
1: Hash partition the left table on the current GPU $L$ into $N$ partitions, $L_0, L_1, ..., L_{N-1}$.
2: **for** $i \leftarrow 0$ to $N-1$ **do**
3:     Send $L_i$ to the GPU $i$, with compression if needed.
4:     Receive the incoming partition from the GPU $i$, with decompression if needed.
5: **end for**
6: Concatenate all incoming partitions into a single table $L'$.
7: Repeat step 1-6 for the right table $R$ to form the communicated table $R'$.
8: Local join on $L'$ and $R'$ and materialize the result.
```

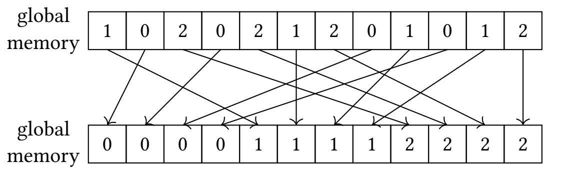
We use a two-pass algorithm for hash partition on GPUs. For both passes, rows are assigned to threads in a round-robin fashion. The purpose of the first pass is to compute the output location of each partition for each thread block. In this pass, each thread loops through its assigned rows, computes the hash values, and atomically adds to a histogram private to each thread block in shared memory. After all threads in the thread block finish the loop, the histogram in shared memory is flushed to the global memory. Then, we use prefix sums to establish the output locations. During the second pass, each thread loads the same set of rows again, and scatters the input rows to their corresponding output locations.

As shown in Figure 3a, scatter in global memory requires random accesses, which perform much worse than sequential accesses. Spatial locality can be improved using a multi-pass scatter, restricted to a certain region of output in each pass [5]. For the scatter in the hash partition, the order of rows within each partition is insignificant. Because we know the output location for each thread block during the first pass, the region of output can be automatically restricted by incrementing the pointer of each partition through atomic add, without resorting to a multi-pass algorithm. Rui and Tu [13] adopted this approach in the partitioning step of their hash join algorithm.

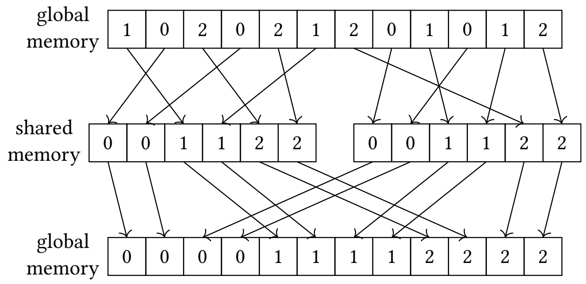
Our approach improves this further by first scattering to shared memory and then copying each partition to global memory, taking advantage of memory coalescing. The motivation is that random accesses within shared memory are much faster than the random accesses to global memory. Figure 3b demonstrates this strategy. We assume the shared memory can hold six elements in this figure. First, we load the first six elements and scatter them into shared memory. Then, we copy these six elements from shared memory to the desired output location. Within the same partition (represented by the same number), memory writes are coalesced. In this example, memory coalescing is limited to two elements. In the real world, shared memory can hold much more than six elements, and the write pattern to global memory is mostly sequential if the number of partitions is moderate. We then process the next six elements until all input elements are scattered.

3.2 All-to-all communication

Suppose there are N GPUs in the system. After the hash partition step in Section 3.1, each table on each GPU has N partitions. During the all-to-all communication step, each GPU sends partition 0 to GPU 0, partition 1 to GPU 1 , partition 2 to GPU 2, and so on. Before the data can be sent, the remote GPU needs to allocate the receive buffer to hold the incoming data. So for each column, the first step in the all-to-all communication is to send the partition size, located on the host memory, to the host process corresponding to each remote GPU. After each host process receives the N incoming partition sizes, it allocates the receive device buffer and uses a prefix sum to calculate the starting location of each incoming partition. Then, each host process calls the communication library to initiate the device transfers. To achieve the best performance, we use GPUDirect RDMA to enable kernel-bypass and transfer the data directly from the GPU device buffer to the network interface controllers (NICs) without copying to a bounce buffer in CPU memory.



(a) Random writes to global memory



(b) Use shared memory to coalesce global memory writes

Figure 3: Scatter in Hash Partition

3.3 Local Join

After the all-to-all communication discussed in Section 3.2 finishes, rows with the same hash values are stored in the same GPU. Therefore, during the local join step, no communication across GPUs is needed.

In our implementation, a join on a single GPU involves the following steps. First, we build a hash table from each row in the smaller table. Then, we probe the hash table for each row in the larger table twice: first to calculate the number of matching pairs to allocate buffers and then to record the matching pairs. Finally, we materialize the keys and payloads of all matching pairs into the output buffer. We choose the smaller table for hash table insertion and the larger table for probing because random reads have a higher throughput compared to random atomic operations. For the hash table, we use open addressing to handle the conflicts.

To support large or multicolumn keys, we do not store the keys directly inside the hash table. Instead, each entry in the hash table stores two values: the 4B hash value and the 4B local row ID of the build table on each GPU. The downside of this approach is we need

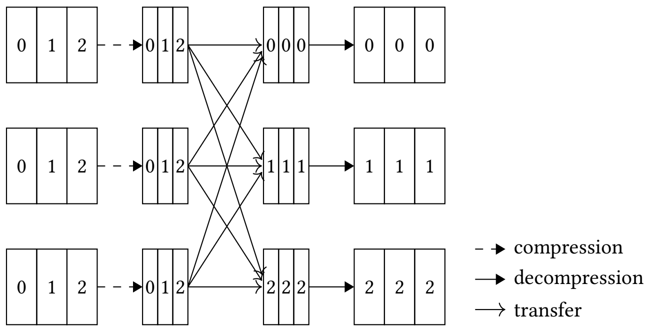


Figure 4: Use compression to decrease communication volume during all-to-all communication

an extra random read to the input table to compare the keys during probing.

We choose 50% occupancy for the hash table, meaning the number of entries in the hash table is double of the number of rows in the build table. In general, lower occupancy means less hash conflicts and better performance, but the hash table consumes more device memory.

To build the hash table, we assign rows to threads in a round-robin fashion. For each row, first we compute the hash value of the row. Then, we try to insert the row into the hash value location using atomic compare-and-swap (CAS). Atomic CAS is needed in this case because multiple threads could insert to the same location at the same time, and only one of these threads should succeed. If the insertion is successful, we additionally store the local row ID into the hash table. If the insertion is unsuccessful, the location has been reserved to another row, and we try the next location until an empty spot is found.

To probe the hash table, we again assign rows to threads in a round-robin fashion and compute their hash values to get the location in the hash table. For each location, we first compare the hash value. If the hash value is a match, we additionally load the key from the input table at the local row ID, which is stored in the hash table. These extra random reads to the input table are the disadvantages of our approach. If the key matches as well, we record the matching row IDs. Then, we move on to the next location and repeat the same process, until we encounter an empty spot.

4 COMPRESSION

When executing the repartitioned join algorithm on systems connected through Infiniband, the bottleneck is the communication throughput during the all-to-all communication step. In this section, we propose to use software-based lossless compression to alleviate this problem by reducing the communication volume.

It has been demonstrated that compression is a viable way to accelerate data transfers on scientific [12] and deep learning [9] workloads. For distributed repartitioned join, the high-level idea is that each GPU compresses the data before sending it over the network. When the message arrives, the target GPU decompresses the data to its original values. Figure 4 shows the integration of compression for the all-to-all communication step of repartitioned join on one table.

One commonly used compression technique is removing duplicates. For example, run-length encoding (RLE) replaces a sequence of the same data values by a single value and a count. LZ family compressors [16] replace repeated occurrences of data by references to a single copy. Another commonly used technique is entropy encoding. For example, Huffman encoding [7] uses fewer bits to represent more commonly used symbols. In general, the compression ratio of these techniques is data-dependent and the performance is hardware-dependent. Standard generic compressors like Deflate [3] are not GPU-friendly due to the serial nature of decoding and parsing of the Huffman symbols.

In this paper, we seek a compression scheme that can achieve a good compression ratio on analytical datasets, while being friendly to the highly parallel nature of the GPU architecture. With these goals in mind, we introduce an efficient cascaded compression scheme that combines RLE, Delta encoding, bitpacking and frame of reference (FOR) layers. This scheme works well for integers in analytical datasets, as we could capture data duplications with RLE, decrease the range of values with Delta and frame of reference, and have a simple and fast entropy encoder with bitpacking.

Next, we introduce the RLE, Delta, bitpacking, and FOR layers used in cascaded compression.

- The RLE layer compresses repeated values into (value, run length) pairs. For example, in Figure 5a, we replace the three shaded 4s in the input sequence with a 3 in the run array and a 4 in the value array. Overall in this example, we replace 18 input integers with 10 integers. In general, the compression ratio of RLE is data-dependent. In the worst case scenario when the input sequence has no repetition, RLE can double the number of integers.
- Delta encoding replaces the input sequence with the differences between consecutive data elements. For example, in Figure 5b, we compute the difference between the two shaded elements in the input sequence, $15004 - 15003$, and store the result -1 to the shaded entry in the compressed sequence. Delta encoding by itself does not compress the data, but it creates more consecutive duplicates and reduces the range such that the output sequence is easier to compress for the subsequent RLE and bitpacking layers.
- In the frame of reference (FOR) and bitpacking layer, first we scan the input sequence and record the minimum value as the reference. Then, we compute the differences between input elements and the reference. Finally, we use the minimum number of bits to represent the differences. For example, the values in the input sequence in Figure 5c are about 15000, and we need to use at least the 16-bit integer type to store them. After the FOR and bitpacking layer, all elements are within 256 and therefore could be fitted in an 8-bit integer type.

Cascaded compression combines these layers by feeding the output of a layer as the input of another layer to yield the best compression ratio. Figure 5d shows the cascaded compression with two RLE layers, one delta layer, and bitpacking layers. In the figure, only the shaded boxes are stored in the compressed output. Other boxes are temporary outputs.

The number of each layer has a significant impact on the performance and compression ratio. Naturally, adding extra layers makes the compression and decompression slower. Furthermore, using RLE on data with no repetition increases the compressed size. Because compression ratio is data-dependent, we sample each column to determine the best compression configuration.

Each GPU must send data to all other GPUs during the all-to-all communication step in the repartitioned join algorithm. This means that each GPU needs to compress as many as the number of GPUs buffers for each column. Launching a separate kernel for each compression layer of each buffer is not ideal for the following reasons. First, the table size per GPU is often limited by the device memory size and needs to be kept constant. As we scale the workload to a large number of GPUs, each buffer sending to a remote GPU gets smaller, so a single kernel cannot fully utilize the GPU. Second, after each layer finishes, we must write the result back to the global memory. Then, the next layer loads the same data back from global memory. These extra global memory reads and writes degrade the performance. Additionally, we need extra device memory to host the temporary output of each layer, which puts more pressure on the already scarce device memory capacity when running most analytical workflows.

In our implementation, we fuse compression and decompression of all layers and all buffers into a single kernel. Each buffer is assigned to a single thread block, which enables us to use shared memory for storing temporary output of each internal layer, without the roundtrips to the global memory. When the number of GPUs is less than 512, we chunk each buffer to make sure that there are enough thread blocks to efficiently use the GPU.

5 TWO-LEVEL SHUFFLE

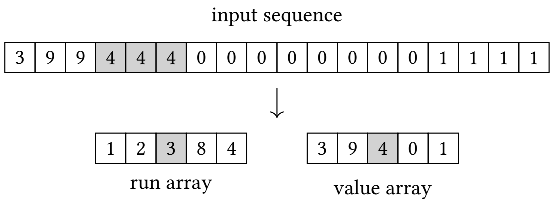
Today's multi-GPU cluster systems are highly complex in their interconnect topology. No longer do we have a set of uniform CPUs connected to a single hierarchy of network switches. Often, the individual nodes are multi-GPU with their own interconnect such as NVLink, and then a set of Infiniband switches span thousands of those nodes. Such topologies create challenges for hash join shuffle implementation.

First, different interconnects have vastly different throughput, which requires that communication algorithms adapt. For example, decreasing in communication volume resulting from the compression scheme discussed in Section 4 can improve Infiniband transfers greatly. However, it is usually not worth the extra costs of compression and decompression within the NVLink domain as NVLink has much better throughput.

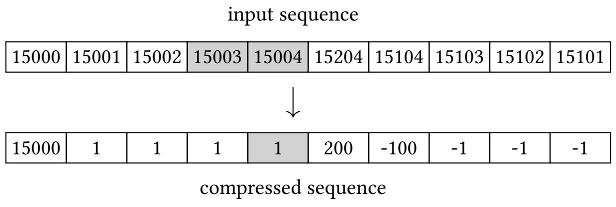
Second, latencies between pairs of GPUs across nodes are not uniform. For example, the DGX SuperPOD fabric is rail-optimized such that the same HCAs of two nodes within a system are connected through the leaf switch, whereas communicating between different HCAs needs to go through the core switch.

To improve communication efficiency during the shuffle stage, especially for small messages, we modify the repartitioned join algorithm that splits the communication into the following two stages:

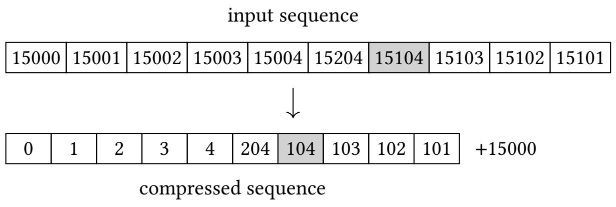
(1) Across DGX nodes where GPUs are connected through Infiniband



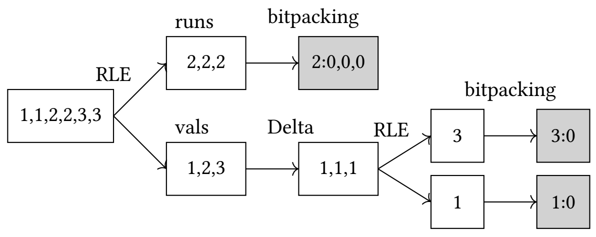
(a) Run-length encoding



(b) Delta encoding



(c) Frame of reference and bitpacking



(d) Combine two RLE, one Delta, and bitpacking layers together

Figure 5: Cascaded Compression

(2) Within a single DGX node where GPUs are connected through NVLink

We summarize the modified repartitioned join with two-level shuffle in Algorithm 2. In this algorithm, compression is only applied to the Infiniband traffic (Step 2-5). To avoid hash conflicts, we use different hash keys in the two hash partition steps (Step 1 and 8) and the local join step (Step 15). In principle, we could combine the two hash partition stages (Step 1 and Step 8) into one by partitioning the table into MN partitions. We choose to keep them separate because hash partition works the best when the number of partitions is moderate. Recall from Section 3.1 that we use shared memory to convert random accesses during the scatter into sequential accesses. The strategy works well only when the number of partitions is moderate.

If the number of partitions is large, there are few elements per partition, and the write pattern is back to random. By separating the two hash partition stages, we ensure that the write pattern is sequential even for a large number of GPUs.

Algorithm 2 Modified repartitioned join algorithm with two-level shuffle

```
Suppose there are $N$ nodes with $M$ GPUs per node. We use $(i, j)$ to represent the GPU $j$ on node $i$. Suppose the current GPU has index $(x, y)$.
1:  Hash partition the left table on the current GPU $L$ into $N$ partitions, $L_0, L_1, ..., L_{N-1}$.
2:  for $i \leftarrow 0$ to $N-1$ do
3:      Send $L_i$ to the GPU $(i, y)$, with compression if needed.
4:      Receive the incoming partition from the GPU $(i, y)$, with decompression if needed.
5:  end for
6:  Concatenate all incoming partitions into a single table $L'$.
7:  Repeat step 1-6 for the right table $R$ to form the communicated table $R'$.
8:  Hash partition $L'$ into $M$ partitions, $L'_0, L'_1, ..., L'_{M-1}$, with a different hash seed.
9:  for $j \leftarrow 0$ to $M-1$ do
10:     Send $L'_j$ to the GPU $(x, j)$.
11:     Receive the incoming partition from the GPU $(x, j)$.
12: end for
13: Concatenate all incoming partitions into a single table $L''$.
14: Repeat step 8-13 for the right table $R'$ to form the communicated table $R''$.
15: Local join on $L''$ and $R''$ and materialize the result.
```

Compared to one-level shuffle, two-level shuffle has the extra cost of a hash partition (Step 8) and all-to-all communication (Step 9-13) within the NVLink domain. If each partition is large enough, the one-level shuffle is already efficient, with throughput close to the NIC's limit. In that case, the extra overhead of the two-level shuffle makes it run slower. On the other hand, two-level shuffle has fewer messages over the Infiniband and does not have cross-rail traffic. When the number of GPUs is large and each partition is small, latency could become a significant factor and two-level shuffle could be more efficient overall. We evaluate the performance comparison between one-level and two-level shuffle with different table sizes in Section 6.3.

After the first shuffle in Infiniband domain finishes, rows with the same hash values are located on the same node, and no cross-node communication is needed afterwards. In principle, we can use any single-node multi-GPU join algorithm in the place of Step 8-15 in Algorithm 2, not limited to repartitioned hash join. For example, we could directly load and store from a remote GPU to allow a fine-grained computation and communication overlap. An efficient single-node multi-GPU join algorithm is beyond the scope of this paper.

6 EVALUATION

6.1 Experimental Setup

We evaluate the repartitioned join algorithm on Selene, an NVIDIA-internal cluster based on the DGX SuperPOD architecture. Selene features 4480 NVIDIA A100 GPUs through 560 DGX A100 systems. Each DGX A100 has eight NVIDIA A100 GPUs with 640 GB device memory in total, and eight NVIDIA Mellanox 200 Gbps HDR Infiniband network cards for interconnect in the compute plane. The DGXs are connected through 850 Infiniband switches with a fat-tree topology, organized into three levels, as illustrated in Figure 1. Inside each DGX, GPUs are fully connected through NVLink.

We evaluate the algorithm on two different datasets. The first is the same as the dataset used in [2] for performance comparison. In this dataset, each input table has two columns. The first column is an 8B key column with random integers, and the second column is an 8B payload column with global row ids. The second dataset is TPC-H "lineitem" and "orders" tables. We use this dataset for evaluating the distributed join performance with compression.

In all experiments, we assume that the data is already located in GPU memory before the join operation begins. All tables, including the input tables, the output tables and the intermediate results, are kept in columnar format. We use a memory pool for efficient memory allocations, and the entire memory pool is preregistered with Infiniband. We use MPI as a launcher, with one GPU per MPI rank. MPI is also used to transfer host buffers. We use UCX directly for transferring device buffers.

Selene is a shared computing resource that has a lot of jobs executing at the same time. The distributed join execution time of the same experiment fluctuates depending on the exact set of nodes assigned to the experiment, as well as the network traffic at the time of the execution. Looking at the time breakdown by steps, we notice that the fluctuation mainly comes from the all-to-all communication step in Infiniband, while the running time for local operations, like hash partition, compression and local join, remains nearly constant throughout multiple runs. To mitigate the fluctuation, we run each experiment several times and report the minimum time. We choose the minimum time as it captures the case when the distributed join is least affected by the activities of other applications.

6.2 Random Integer Key Dataset

This section evaluates the scale-out efficiency of the distributed, repartitioned join algorithm on random keys. In this experiment, both the left table and the right table have 500 millions rows per GPU. Both tables have two columns, an 8B key column and an 8B payload column. Keys are unique random integers in the left table, and each row in the right table has 30% possibility to have a match in the left table.

Figure 6 shows the result of this experiment. We observe that repartitioned join, with both one-level and two-level shuffle, scales well to 1024 GPUs. As each table is large in this experiment, we observe that the join with two-level shuffle is slower compared to the join with one-level shuffle, due to the extra cost of hash partition and all-to-all communication when shuffling in the NVLink domain. The best performance achieved is 10677 GB/s with one-level shuffle

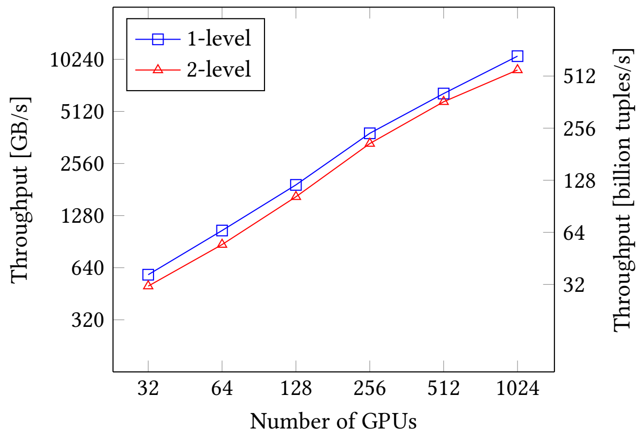


Figure 6: Weak-scaling performance for distributed join on random keys, with one-level or two-level shuffle

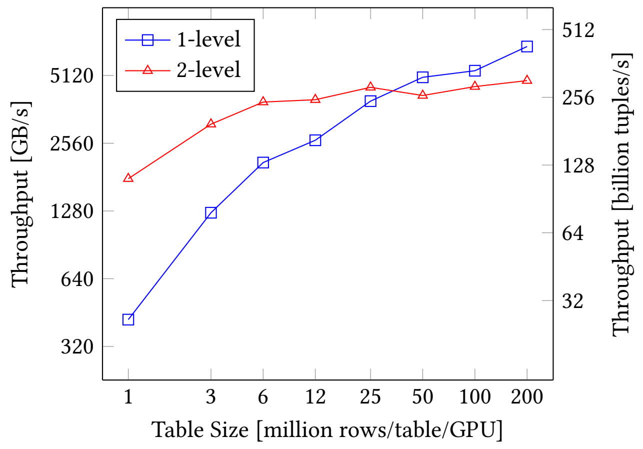


Figure 7: Distributed join performance vs. table size on 512 GPUs, with one-level or two-level shuffle

on 1024 GPUs with 16384 GB input size, which corresponds to 667 billion input tuples per second.

Because the keys are random integers, we do not evaluate compression in this experiment.

6.3 Performance vs. Table Size

This section evaluates the impact of the table size on the performance of the repartitioned join algorithm. The workload is the same as Section 6.2, but the number of GPUs is kept constant at 512 GPUs, while the table sizes vary between 1 million rows per table per GPU to 200 million rows per table per GPU.

The result is presented in Figure 7. In general, we observe that one-level shuffle is suited for large tables and two-level shuffle is suited for small tables. The cross point in this case is around 25 million rows per table per GPU. For large input sizes like 200 million rows per table per GPU, both one-level and two-level shuffle use the Infiniband network efficiently. The extra cost of hash partition and all-to-all communication within NVLink domain for the two-level shuffle makes it perform worse compared to the one-level shuffle. On the other hand, when the input size is small, like 1 million rows per table per GPU, the size of each partition is only about 2000 rows. So, latency in addition to the throughput could influence the performance. A repartitioned join with two-level shuffle performs better in this case because it only has $64 \times 63 = 4032$ Infiniband messages. In comparison, one-level shuffle has $512 \times 511 = 261632$ messages. In addition, two-level shuffle does not have cross-rail traffic, so the latency per message could be lower.

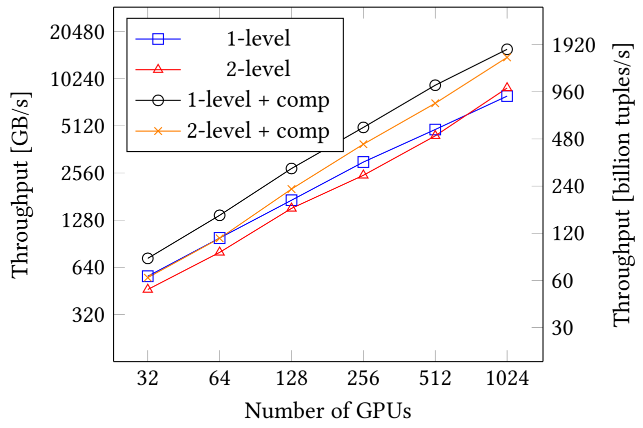


Figure 8: Weak-scaling performance on TPC-H dataset, with or without compression, with one-level or two-level shuffle

6.4 TPC-H Dataset

This section evaluates the performance of the distributed repartitioned join algorithm on the TPC-H dataset, and compares the performance with and without cascaded compression. We perform the inner join of the O_ORDERKEY, O_ORDERPRIORITY columns from the “orders” table, with the L_ORDERKEY column from the “lineitem” table. O_ORDERKEY and L_ORDERKEY are 8B key columns, and O_ORDERPRIORITY is a 4B payload column. Many TPC-H queries include a join of these two columns, for example, Query 4. Also, it is the largest and most expensive equality join to perform for a given scale factor.

We generate the dataset with scale factor (SF) 100k, and divided each table into 1024 parts of the same sizes. For an experiment with N GPUs, the first N parts of each table are used, with one part per GPU. Therefore, the data size per GPU is constant: 146.48 million rows for the orders table and 585.94 million rows for the “lineitem” table.

The result of this experiment is shown in Figure 8. We observe that the repartitioned join algorithm with both one-level and two-level shuffle scales well. Like in Section 6.2, when the number of GPUs is small, join with two-level shuffle is slower than join with one-level shuffle. As the number of GPUs increases, data size per GPU remains constant, but the message size between a pair of GPUs decreases proportionally. Compared to Section 6.2, the input size is smaller, so the decreasing message size makes latency more important. As a result, when the number of GPUs increases, the performance gap between the one-level shuffle and the two-level shuffle is closed, since the two-level shuffle has less Infiniband

Column	RLE	Delta	bitpacking
O_ORDERKEY	2	1	true
O_ORDERPRIORITY	0	0	true
L_ORDERKEY	2	1	true

Table 1: Cascaded compression scheme for each column

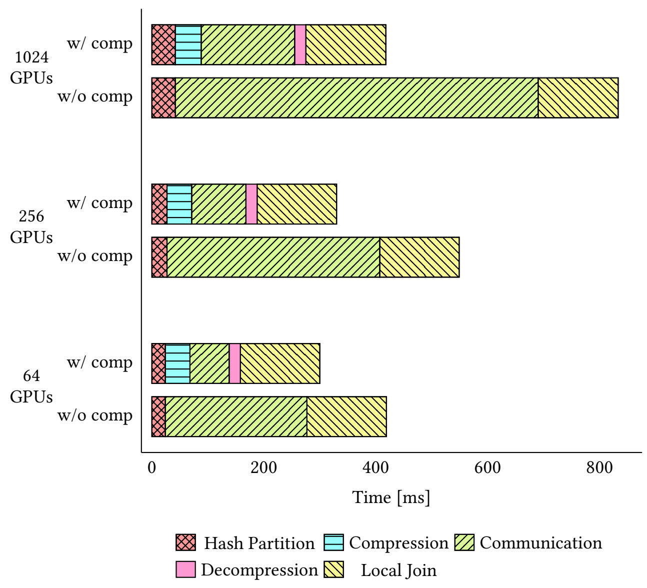


Figure 9: Time Breakdown of weak-scaling distributed joins of TPC-H "lineitem" and "orders" tables, with or without compression

messages. For this dataset, cascaded compression offers significant benefits. Without compression, the best performance that we get is 1015 billion tuples/second. With compression, we can achieve 1793 billion tuples/second, which is a 1.77x improvement.

As discussed in Section 4, the number of layers in cascaded compression has a significant impact on the compression performance and ratio, and we use a selector to sample the input columns to determine the best configuration. The sampling time is considered a preprocessing step. It is not counted towards the distributed join time, as the compression configuration is a property of the column. If we run the join multiple times, the sampling only needs to be performed one time. Table 1 shows the cascaded schemes chosen by the selector. For these three columns, bitpacking layers are always used. The number of RLE and Delta layers vary depending on the columns.

To investigate further into how compression helps improve distributed join performance, Figure 9 shows the time breakdown of all steps when weak-scaling the distributed join on TPC-H dataset with 64, 256 and 1024 GPUs. In this plot, the uncompressed version uses one-level shuffle, and for 1024 GPUs, we observe 78% of the total join time is spent in the all-to-all communication step. The compressed implementation uses the one-level shuffle as well, but uses cascaded compression to decrease the communication volume. For 1024 GPUs, we observe that the all-to-all communication time decreases from 648ms to 167ms, which more than makes up for the extra 46ms cost on compression and 20ms on decompression. We also observe that the compression and decompression time stay constant when weak scaling from 64 GPUs to 1024 GPUs, indicating that our strategy for fusing all layers and batches into a single kernel during cascaded compression and decompression is scalable.

7 RELATED WORK

Join algorithms on a single GPU. Single-GPU join implementations have been extensively studied in the last decade. The first comprehensive single-GPU join analysis was presented by He et al. [6] in 2008, with four different join algorithms on the GPU, including a partitioned hash join. The paper concludes that GPU was 2-7x faster compared to CPU, including the data transfers between the CPU and the GPU.

Later, Kaldewey et al. [8] showed similar speedup while using CUDA Unified Virtual Addressing (UVA) on pinned host memory for efficient host-device transfers. The benefit of using UVA is that it removes the requirement that the combined memory usage of the build table, the probe table and the hash table must fit inside the scarce device memory. The paper showed that the performance was limited by random CAS performance for hash table insertion and by PCI-e transfer throughput for probing. The paper also evaluated the performance of traditional no-partitioned join compared to the partitioned join on the GPU.

Several years later in 2017, Rui and Tu [13] revisited He's work [6], introduced updated partitioned hash join and sort-merge join algorithms that took advantages of the new hardware features like native atomic operations in global memory and shared memory, and concluded that the GPU was 5-10x faster compared to the state-of-the-art implementation on the CPU.

Recently, Sioulas et al. [14] implemented a hardware-conscious partitioned join algorithm and demonstrated that the achieved throughput was close to the memory bandwidth limit when the tables are located in device memory. The paper also presented strategies when at least one relation did not fit inside device memory, and showed their strategies can saturate the PCI-e bandwidth.

Lutz et al. [10] compared NVLink and PCI-e as interconnect technologies between the GPU and the CPU, and showcased that with a high-bandwidth interconnect like NVLink, the bottleneck shifted from the interconnect to the computation or GPU memory throughput.

This paper is related to these single-GPU contributions in two ways. First, we tackle the device memory limit by scaling out to multiple GPUs with a distributed implementation instead of spilling to the CPU memory. The distributed implementation allows us to perform joins on a dataset as large as 16 TB. In comparison, none of these papers operate on such scale. Second, these single-GPU implementations complement our work as the local join step. This paper uses a no-partitioned join within a single GPU, but the distributed repartitioned join introduced is modular enough that we can replace the local join with most in-GPU algorithms presented in the single-GPU join literature.

Multi-GPU Join on single node and clusters. Guo et al. [4] proposed distributed hash join and sort-merge join for multi-GPU clusters and evaluated the performance with different paths to remote GPUs. Their design used GPUDirect RDMA similar to our approach. However, the scale of their evaluation is limited to 8-16 GPUs total and the cluster did not have NVLink-connected GPUs.

Paul et al. [11] proposed a data transfer strategy with multi-hop transmission and adaptive routing to effectively use the hypercube topology on a single DGX-1 node. They demonstrated near-perfect NVLink utilization and outperform [4] solution by 2.5x. The authors noted that high performance network interconnects such as RDMA can be an opportunity to further improve the scale of multi-GPU architectures for huge data sets, which is demonstrated in our work.

Distributed Join on CPU clusters. Performance of distributed joins on a cluster of CPUs was analyzed in Barthels et al. [2], and it demonstrated a throughput of 48.7 billion input tuples per second on 4096 cores using the radix hash join with compression. To our knowledge, this is the best scale-out result for joins on any architecture. The hash join algorithm presented in the paper is similar to our approach. The paper used RDMA with one-sided communication provided by MPI to achieve best Infiniband performance and to decrease CPU load. The compression algorithm used in the paper was simple: the 16-byte tuples were compressed into 8-byte values. In comparison, our cascaded compressor is more sophisticated and more general. Using the same dataset, our implementation achieves 547 billion input tuples per second on 1024 GPUs without compression. On the TPC-H dataset with compression, we achieve two orders of magnitude better performance of 1.8 trillion tuples per second. This was made possible thanks to much higher GPU memory and interconnect speeds, as well as our novel communication optimizations.

Using compression for optimizing communications. Compression was used in many other computational domains to optimize network transfers. Young et al. [15] evaluated various compression techniques on the CPU to improve Graph500 scaling on Ethernet networks.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we presented the distributed repartitioned join on GPUs, investigated using compression and two-level shuffle for optimizing communication traffic, and evaluated the scaling performance up to 1024 GPUs. To our knowledge, this is the first in-depth analysis of distributed join performance at the scale of thousands of GPUs. We conclude that

- Distributed hash-join can be scaled effectively up to 1024 GPUs connected through Infiniband, and such systems can be a great fit for large scale analytics or ETL jobs with complex joins.
- Without data compression, execution time is dominated by the all-to-all communication step. Therefore, using compression to decrease communication volume is critical for the best performance. The cascaded compression introduced in this paper is GPU-friendly, and improves the performance of distributed join by 1.77x on 1024 GPUs.
- On small tables, latency as well as throughput could impact performance. The two-level shuffle presented could improve distributed join performance on latency-bound workload by reducing the number of messages over Infiniband and taking advantage of the topology of modern GPU clusters.

Despite the promising results, there are a lot of opportunities to improve on this work. First, our implementation does not overlap computation and communication. As a future work, directly loading or storing from remote GPU buffers within the NVLink domain enables fine-grained computation and communication overlap, and fits nicely with the two-level shuffle introduced in this paper. Second, our cascaded compression technique works well for integer data, such as the join keys in TPC-H dataset, but does not compress other types of data efficiently, such as strings. Other compression techniques should be explored in this case. Third, both the random key dataset and the TPC-H dataset used in this paper have keys with uniform distribution. The scalability of the repartitioned join algorithm on skewed datasets needs to be evaluated further. Finally, the no-partitioned local join used in this paper does not use cache effectively. One possible improvement is to switch to a partitioned join during the local join step, like discussed in Sioulas et al. [14].

REFERENCES

[1] Austin Appleby. 2016. Murmur3 hash function. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[2] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed Join Algorithms on Thousands of Cores. *Proceedings of the VLDB Endowment* 10, 5 (2017), 517–528. https://doi.org/10.14778/3055540.3055545

[3] Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. https://doi.org/10.17487/RFC1951

[4] Chengxin Guo, Hong Chen, Feng Zhang, and Cuiping Li. 2019. Distributed Join Algorithms on Multi-GPU Clusters with GPUDirect RDMA. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 65, 10 pages. https://doi.org/10.1145/3337821.3337862

[5] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. 2007. Efficient Gather and Scatter Operations on Graphics Processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (Reno, Nevada) *(SC '07)*. Association for Computing Machinery, New York, NY, USA, Article 46, 12 pages. https://doi.org/10.1145/1362622.1362684

[6] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[7] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[8] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware* (Scottsdale, Arizona) *(DaMoN '12)*. Association for Computing Machinery, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[9] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *Proceedings of the 6th International Conference on Learning Representations* (Vancouver, BC, Canada).

[10] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[11] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254

[12] P. Ratanaworabhan, Jian Ke, and M. Burtscher. 2006. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)* (Snowbird, UT, USA). Institute of Electrical and Electronics Engineers (IEEE), New York, NY, USA, 133–142. https://doi.org/10.1109/DCC.2006.35

[13] Ran Rui and Yi-Cheng Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management* (Chicago, IL, USA) *(SSDBM '17)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/3085504.3085521

[14] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (Macao, China). Institute of Electrical and Electronics Engineers (IEEE), New York, NY, USA, 698–709. https://doi.org/10.1109/ICDE.2019.00068

[15] Jeffrey Young, Julian Romera, Matthias Hauck, and Holger Fröning. 2016. Optimizing communication for a 2D-partitioned scalable BFS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (Waltham, MA, USA). Institute of Electrical and Electronics Engineers (IEEE), New York, NY, USA, 1–7. https://doi.org/10.1109/HPEC.2016.7761596

[16] Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714