== Abstract. ==

One of the difficulties faced when using a general purpose graphing processing on memory intensive tasks, is the considerable amount of time taken to transfer data from a CPU. Such is the case when one tries to upload a projection index from a CPU onto a GPU. One way to minimize the amount of data that needs to be transferred is through the use of compression. In this paper a Run Length Encoding (RLE) compression scheme will be used to minimize the size of the data needed to be transferred. The idea is to compress a projection index using the RLE scheme and then uncompress it within the GPU using the parallel prefix sum as a building block. The parallel prefix sum scan will help determine how to allocate and copy the decompressed projection index within the GPU. To conclude, a benchmark test was performed comparing the different algorithms suggested and determining whether there's any improvement in performance by loading compressed and uncompressing, as opposed to loading an uncompressed projection index.

/\*Transfer a compressed projection index column and uncompress it in the GPU. The compression scheme used would be RLE (Run Length Encoding) and the algorithm for uncompressing in parallel within the GPU should be the Prefix Sum algorithm.\*/

== Introduction ==

The projection index supports thread-level parallelism and therefore could potentially make good use of a GPU. However, most of the time spent when doing a query evaluation with a projection index, is spent in transferring data from the CPU to the GPU. Gosink et al [#x], improve on this bottleneck by reducing the size of the data that needs to be transferred; they do so by changing the encoding of the data that needs to be transferred. In this paper, to reduce the size of the data that will be transferred from the CPU to the GPU, compression is used. Then after the index is transferred compressed, it is decompressed using an uncompressing algorithm within the GPU itself.

The compressed projection index is sent to the GPU as two separate arrays. One array is the frequencies, which represent the number of times an attribute value repeats, and the other array consists of the attribute values themselves. For the Run Length Encoding compression scheme to be useful in reducing the size of the data, the projection index must be created on columns that are not unique and that allow themselves to be compressed somewhat. After the algorithm is sent to the GPU, it is decompressed using a parallel decompression algorithm. Two algorithms were designed to perform this job in parallel, and both of them use the Prefix sum algorithm to do it.

== Parallel Prefix sum algorithm ==

The prefix sum algorithm is an essential building block for uncompressing a projection index that has been previously compressed in the RLE encoding format. #x et al [#x], present a method to calculate the prefix sum of an array in parallel. The article classifies two types of prefix sums, or scans as they are also called, inclusive and exclusive. The inclusive scan generates a new array in which every element j is the sum of all elements up to and including j. The exclusive scan on the other hand is an operation that contains the sum of all previous elements, but not j itself, both scans are illustrated in figure 1. A scan can be performed sequentially to run on a single thread. Two arrays are kept for such scan, one is the input array, and the other is the output array. The input array contains the original elements before the scan, the output array is the array generated after the scan. To perform the scan, a loop is executed over the elements in the input array assigning the sum of the previous element of the input array and the output array to the current element in the output array. The algorithm is illustrated in the following listing (listing 1).

Listing 1. (Taken from Harris [#x])

void **scan**( float\* output, float\* input, int length)

{

output[0] = 0; *// since this is a prescan, not a scan*

for(int j = 1; j < length; ++j)

{

output[j] = input[j-1] + output[j-1];

}

}

To perform the algorithm in parallel, a first approach was given. The first algorithm presented in the paper is the naïve parallel scan. This algorithm assumes that there is one processor for each data element. For a GPU running CUDA this cannot be accomplished as the number of elements will often surpass the amount of processors available. To work around this problem a double-buffer array is used such that the work on array of 512 elements at a time. 512 elements at a time, because this is the largest block size and data can only be synchronized within the block. The algorithm is illustrated in listing 2, and an illustration of how it performs the additions is illustrated in Figure 1. It is important to note that these operations are performed within the same array; the elements are added such that the distance between the elements increases each time by a power of 2.

Listing 2. (Taken from Harris [#x])

**for** *d* := 1 **to** log2*n* **do**

**forall** *k* **in parallel do**

**if** *k* \_ 2*d* **then** *x*[*k*] := *x*[*k* − 2*d*-1] + *x*[*k*]



Figure 1: Naïve parallel scan performed on 8 elements. (Taken from Harris [#x])

The naïve parallel scan has a work complexity equal to sum from d = 1 to log base 2 n n - 2^(d-1) = O(n log base 2 n ) addition operations. This scan's work complexity is even greater than the sequential scan which is of O(n) and therefore it is not work-efficient. The factor of Log base 2 n can significantly worsen the performance for the algorithm as n increases.

They developed a work-efficient scan algorithm; to do this they employed an algorithmic pattern that is based on an algorithm used to build balanced binary trees in parallel. The algorithm consists of two phases: the reduce phase and the down-sweep phase. In the reduce phase also called the up-sweep phase, the tree is traversed from the leaves to the root computing partial sums of neighboring nodes each time increasing the distance between them by a power of 2, until reaching the root of the tree which would hold the sum of all the nodes in the array. Pseudocode for this phase is listed in listing 3, and an illustration of the process is given in figure 2.

Listing 3.





Figure 2. Up-sweep or reduce phase on 8 elements. (Taken from Harris [#x])

Following the first phase of the algorithm, is the second phase which completes computing the scan by performing a down-sweep phase. The down-sweep phase starts from the root of the tree and uses the partial sums computed in the first phase. It discards the last sum of all elements, and replaces it with an element of value 0. A series of swap adds follows in which the sum of neighboring elements is assigned to the rightmost element. In this phase the distance between the neighboring elements decreases by powers of 2, starting from the last distance in the up-sweep phase. The Algorithm is listed in listing 4, and an illustration of its process is given in figure 3.

Listing 4.





Listing 4: “An illustration of the down-sweep phase of the work efficient parallel sum scan algorithm. Notice that the first step zeros the last element of the array.” (Taken from Harris [#x])

Note: In the paper they discussed more about this stuff, including some optimization techniques.. should I discuss them in the paper.?

== Design of Algorithms for Uncompressing ==

Two algorithm design approaches are taken for decompressing a projection index in the GPU (The projection index in these terms may also be referred to be a string). Both approaches use the prefix scan differently. The first algorithm which was called the unbalanced approach uses the prefix sum algorithm as an indicator for each thread to know from where to where to write the elements to allocate the uncompressed index. The algorithm is performed in two phases, and it is unbalanced because the workload of each thread is different, with threads handling elements that are heavily repeated doing most of the work.

The second algorithm which was called the Load Balanced approach has five phases. In two of these phases parallel prefix sums are performed and the ending result of the last prefix sum is an array representing the uncompressed index. The algorithm is nicely load balanced because the amount of work done by each thread is the same. However, this algorithm almost twice as much memory as the unbalanced approach, and performs more than twice the amount of kernel calls.

The input to both algorithms is a pair of arrays that represent the RLE compressed index. One array contains the symbols S, which are the different attribute values found on the projection index. The other array F contains the frequencies or repetitions of each index in the encoded algorithm. The length of both arrays is the same, as they correspond with each other, this length is denoted as C as it is the number of elements in compressed form. The length of the uncompressed index is denoted as U. The following figure (figure #x) illustrates the process where X3Y1Z7 is uncompressed sent as two arrays with length C = 3, and once uncompressed having a length of U = 11.

== The Unbalanced approach ==

The algorithm starts doing the decompression by adding an element of value 0 to the array of frequencies, F. After doing that it obtains array X of length C+1 by performing an exclusive scan on the array of frequencies F, which also has been modified to have C+1. The last element of the exclusive scan X is the sum of all the frequencies, which is also the length of the decompressed array, U. It is used to allocate the amount of memory necessary in the GPU to hold the uncompressed array. The exclusive scan, array X, is then used to have each thread uncompress each element by writing it from one initial offset to where the element that is being repeated is changed. These two offsets are given by the exclusive scan and thus the decompression is performed. The workload of this algorithm is not balanced, as the amount of work a thread does depends directly on the frequency of the element it is decompressing. Threads decompressing elements with few repetitions will take considerably less time, than threads decompressing elements with various repetitions. The pseudocode for this algorithm is given in Listing 5.

Listing 5.

Phase 1: add element of value 0 at the end of array F.

get X as exclusive-scan of C+1 elements of F

Phase 2: for i:= 0 to C

forall k in parallel do

for j := X[i] to X[i+1]

result [j] = S[j]

== The Load Balanced approach ==

Taking in consideration that the first algorithm was not load balanced and that it depends greatly on the on the nature of the data it handles, a second approach was taken. This algorithm starts similar to the first one by obtaining an exclusive scan of the array of frequencies. However, in this algorithm the frequencies array is not modified, and thus it has a length of C, and so the exclusive scan, X will also have length of C. Adding the last element of the exclusive scan to the last frequency; one obtains the size of the uncompressed array, U. Having this size, memory for the decompressed array is allocated in the GPU, and phase two is on track. In the 2nd phase, uncompressed array is initialized, A by having each thread assign a value 0 to all its positions. After Phase two is completed, Phase three has each thread writes a 1 to positions given by the elements of the exclusive scan X. In other words, X represents the positions of the uncompressed array, where there will be a change in symbol. Phase four follows, and an inclusive scan is performed on the now modified array A. The output of this scan is the positions of the corresponding symbols in the uncompressed array. In the final phase, phase five, these positions are used to write the actual symbols onto the final uncompressed index denoted B.

Listing 6.

Phase 1: get X as Exclusive-scan of F

Phase 2: for i = 0 to U

forall k in parallel do

write 0 to item i in array A

Phase 3: for i = 0 to C

forall k in parallel do

write a 1 to item X[i] in array A

Phase 4: overwrite A as an Inclusive-scan of array A

Phase 5: for i = 0 to U

forall k in parallel do

write item S[A[i]] to Uncompressed Index B

To improve further on the performance of this algorithm, one could possibly do away with the 5th phase by sending the query one wants to perform in the projection index in terms of the positions of the elements in S. Not only would this save time, but also memory, as array B of size U would no longer need to be created.

== Performance Analysis==

To conclude the project a benchmark test will compare and find the cases where a compressed index can be more readily available to the GPU by uncompressing as opposed to loading it as an uncompressed index.

Projection index with 1024 different elements and then doubles the amount of elements.

Projection index with fixed size of elements and then increasing the number of different elements from 2 different to having all elements with a frequency of 3.

== Problems ==

It may be noticed that one of the problems that was not addressed was the fact that it is possible to have a compressed index that will not fit in the GPUs memory once it is uncompressed within the GPU.

== Conclusions and Future work ==

The load balanced algorithm is mostly limited by its fourth phase in which an inclusive-scan is performed in order to obtain the position of corresponding elements in the decompressed index.

One possible avenue for future work is to compare the GPU's uncompression against the CPU, as it may not be a good algorithm for transferring an index quickly in the GPU, but it may be a good way to perform uncompression when using a GPU.