== 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 uncompressed 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 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 was used. Then after the index is transferred compressed, it is uncompressed 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 uncompressed using an uncompressing algorithm. The algorithms that were designed to perform this job use the Prefix sum algorithm.

== 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. 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 this operations are performed within the same array, the elements are added such that the distance between the elements increases each time by a multiple 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 ==

In this paper two algorithm design approaches are presented for uncompressing an index in the GPU.

== The Naive unbalanced approach ==

== The Load Balanced approach ==

== Performance Analysis==

== Future work ==

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.