== Abstract. ==

One of the difficulties faced when using a general purpose graphics processing on memory intensive tasks, is the considerable amount of time it takes to transfer data from a CPU to a GPU. Such is the case when one tries to upload a projection index onto the graphics processor in order to process a query. To ameliorate this problem, one may reduce the amount of data that needs to be transferred by compressing it. In this paper a Run Length Encoding (RLE) compression scheme is 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 decompress it within the GPU using a decompression algorithm. Two algorithms were designed for this purpose: one is load unbalanced and the other is load balanced. Both algorithms used the parallel prefix sum as a building block. The parallel prefix sum helps the algorithms determine how to allocate and copy the decompressed projection index within the GPU. To conclude, benchmark tests were performed comparing the two different algorithms suggested against moving the projection index uncompressed to the graphics processor’s memory. The tests were performed in two different GPUs, one with computability 1.1 and the other 1.3. It was determined from the results that the performance of the algorithms depends mostly on the GPU used and its computability, the percentage of compression of the projection index, and the nature of the data being processed including its data type. In general, for both GPUs an improvement in performance was observed. The improvement was consistent for the load balanced algorithm, but it was not for the load unbalanced algorithm.

== Introduction ==

GPUs are both powerful and inexpensive devices that could be used in order to obtain more computational throughput. Not only can they perform faster than CPUs in number of float point operations per second but they also have a higher growth rate in performance. Over the last decade the growth rate of GPU performance has been higher than that of the CPUs. GPUs were primarily designed and used for performing algorithms around graphic computation, such as transforming, rendering and texturing geometric primitives such as triangles. However, many have sought techniques for using the GPU to perform tasks that are not related to graphics and that are typically handled by the CPU.

Many researchers view GPUs as today’s most powerful computational hardware for the dollar. Researchers and developers in areas not related to graphics have also become interested in harnessing this power for general-purpose computing. This effort is collectively known as GPGPU (it stands for “General-Purpose computing on the GPU”) [#x].

Indexing is a process that involves converting a collection of data for faster search and retrieval. This process often involves creating a data structure or populating one that is already in memory. There have been various indexing structures used for CPUs. However, they are often not easily parallelizable and therefore cannot benefit from an implementation on a GPU. Indices whose performance may benefit from a GPU implementation must allow SIMD operations in parallel by several threads. Also, the time it takes to transfer the data structure from the CPU to the GPU should also be significantly less, as to make it worth using a GPU.

GPUs are also limited by the amount of memory they have and there are also limitations imposed by the data buses that transfer data to the GPU. Gosink et al [5]. proposed a new indexing data structure that reduces the size of the data that needs to be transferred from the CPU to the GPU. They use a binning mechanism in order to encode the data such that it takes less space. In their paper they refer to the actual data from the projected column as high resolution data, and the encoded data as low resolution data. To illustrate this idea with an example, take arbitrarily a column of integers for example where each element is four bytes long in a 32-bit Intel architecture. When each integer is encoded using their binning scheme, each element is reduced to occupy only one byte, that in all saves three bytes of space for each element that needs to be transferred to the GPU. In this case, our original integer values would be the high-resolution data, and our encoded values would be the low-resolution data (See Figure 6). When the encoding is changed information is lost and because of this one cannot completely answer a given query with low resolution date. To solve this problem, a second data structure is required to resolve the query where the encoded data is not enough to determine which attribute values fulfill and do not fulfill the condition. For these boundary bin values, they use a data structure that contains the high resolution data called the Data Parallel OrBic Structure. This structure maintains an offset table that helps it identify where each bin’s full resolution data is located. It also maintains another table that identifies the row identifiers for each of the elements in the full-resolution table (See Figure 7).

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 contrast, 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 a decompression algorithm within the GPU itself.

In other words, by compressing the problem is turned from a memory-intensive process to a more computationally-intensive, thus reducing the bottleneck of the

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 RLE 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 Prefix sum.

== 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 types of scans are illustrated in figure 1.

Figure 1. Illustrates both the exclusive scan and inclusive scan of an arbitrary array of frequencies.

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 and 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 where the sum of the previous element of the input array and the output array is assigned to the current element in the output array. The algorithm is illustrated in the following listing (listing 1).

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

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

{

output[0] = 0; *// since this is an exclusive scan*

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

{

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

}

}

To perform the algorithm in parallel, Harris started with a simple naïve algorithm and moved to one more complex but with better performance. 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 number of processors available. To work around this problem a double-buffer array is used, such that warps may work on arrays of 512 elements at a time. 512 elements are processed 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. Naïve parallel scan (Taken from Harris [#x])





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.

Harris also developed a work-efficient scan algorithm; to do this he 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. As it traverses, it computes partial sums of neighboring nodes each time increasing the distance between them by a power of 2, until reaching the root of the tree. The root of the tree 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, the second phase 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 pseudocode for the algorithm is listed in Listing 4, and an illustration of its process is given in Figure 3.

Listing 4.





Listing 4: The down-sweep phase of the work efficient parallel sum scan algorithm. It can be noted that the first step discards the last element of the array replacing it with a 0 (Taken from Harris [#x]).

Harris finalized with this last algorithm for doing a parallel prefix sum on a set of elements. He then went on improving the implementation by dealing with aspects of the GPU itself. The direct implementation of the work efficient algorithm had bank conflicts due to its shared memory access patters. To solve this problem padding was added to each shared memory array index. A macro was added to compute the bank-conflict-free shared memory array indices. Also, the work efficient algorithm was originally designed to work only with arrays with sizes that are powers of two. They extended the algorithm to work with arrays of arbitrary sizes by dividing the array into blocks that could be scanned by a single thread block. The total sums of those blocks are then scanned as blocks. The generated array of this scan is then used to increment the block next to the block where the sum was originated from (see figure #x). More detail on this work can be read from the article Parallel Prefix Sum (Scan) with CUDA [#x].



== Design of Algorithms for Decompression ==

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 or sequence). Both approaches use the prefix scan differently. The first algorithm which is 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, 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 uses almost twice as much memory as the unbalanced approach, and performs more than twice the number 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 Load Unbalanced approach ==

The algorithm starts doing the decompression by appending an element of value 0 at the end of 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 second phase, the uncompressed array A is initialized by having each thread assign a value 0 to all its positions. After the second phase is completed, Phase three has each thread write 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. The pseudocode for this algorithm is given in Listing 6, and an example illustration of the process is given in Figure #x.

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==

== Data Analyzed ==

To test the algorithms, synthetic sequences were generated in the CPU to represent a projection index after being sorted, compressed, and loaded in main memory. This was done to simplify the interpretation of the different factors influencing the performance on benchmark tests. For a compressed index, two different arrays are created, one with characters or integers representing the attribute values, and one with the number of times each of those attribute values repeat themselves. The distribution of this data was simulated in three different ways: In the first distribution, the data was generated such that the next element repeats itself once more than the previous one. Compressed sequences were generated like this first with 500 elements, and then increasing by 500 elements until reaching 5000 elements (See figure #x). This type of distribution was presumed not to favor the load unbalanced algorithm much. This data distribution is also referred as the sequentially incremented data distribution. The second data distribution has 1024 different attribute values which are only repeated twice, thus having very little compression. The size of the compressed sequences always stays the same but the number of times an element repeats itself is doubled on each iteration. So essentially, one would have indices of the same size when compressed and of different sizes when they are uncompressed. Finally, the last distribution of data consisted of an uncompressed sequence or index of fixed size; 16777216 elements in total. The number of different elements was then incremented from 1024 to 8’388,608 by doubling on each iteration. This approach does the opposite of the previous data distribution by creating uncompressed indices of the same size, but of different sizes when compressed. An illustration of the different data distributions is shown in Figure #x.

Figure #x. Compressed form of sequences under the three different data distributions.

== Performance in GPU with Computability 1.1 ==

Initially, tests were performed in a 9400m NVidia GPU with 16 cores and 256 Mb of video main memory. Both algorithms were tested against copying the uncompressed index directly to the GPU. The first data distribution described (the sequentially incremented one) was used for this test. Characters were used as attribute values of the index, at first. The outcome of the test in this GPU was that neither the Load Unbalanced (LU) nor Load Balanced (LB) algorithms were a good approach to improve the time it takes to transfer. Transferring the uncompressed index (UC) proved to be a better option (See Table #x and Figure #x). Presumably, the GPU did not have sufficient cores to make the computations necessary quickly enough to compensate with the speed at which the uncompressed index was being transferred. Additionally, for a GPU with computability of 1.1, it was expected that the writes would not always coalesce if the write access patterns on the array were not organized in a sequential order. There are such writes in the Load Balanced Algorithm.



Figure #x. Time taken to transfer a sequence of characters and decompressing using the Load Unbalanced Algorithm, and the Load Balanced Algorithm (LB) versus transferring the uncompressed index.

There was not much difference between the Load Balanced algorithm and the unbalanced algorithm. Each phase of the load balanced algorithm was shown in a pie chart (see Figure #x) to determine which phases took the majority of time to do the decompression. From the pie chart, it can be determined that Phase 4 takes the most time of the algorithm, presumably because the graphics processor only had 16 cores, and the sheer size of the array. Phases 2 and 5 follow Phase 4 in amount of time taken. Phase 2 takes a long time due to the sheer size of the array which could only improve with a greater number of processing elements. The writes in this phase should be coalesced because they are sequential. The long time taken in Phase 5 is attributed to the computability issue. This is issues is shared by Phase 3, but it is only significant in Phase 5 as the amount of computation in Phase 3 is much less.

Figure #x. Performance analysis of the different phases of the algorithm. Notice that only Phases 2, 4, and 5 were significant.

== Performance Enhancements and Performance with Integers ==

The fifth phase of the Load balanced algorithm involves many read/write operations. The read operations could be accelerated by bringing the array of symbols into texture memory. This would have the effect of caching this constant array and thus it would improve performance of this phase. A test to verify this was performed, using characters as attribute values, a slight improvement on performance was observed (See figure #x).

Figure #x. Performance of Phase 5 with texture memory (P5T), and without it (P5). Notice that there is a small improvement to Phase 5 when using Texture memory.

Finally, the test for this GPU with the first data distribution was repeated using integers, and the results were also very different. Sending the uncompressed index was no longer the best option, and using the Load Unbalanced and Load balanced algorithms made the index more readily available to the GPU (see Table #x and Figure #x). The reason for this sharp and astonishing difference between using characters and integers could be due to the size of integers being four times that of the characters. Since the uncompressed index only involved characters, the number of elements and the actual size in bytes were the same. When the attribute values of the index were changed to integers the size of the uncompressed sequences quadrupled, thus, the slope of the curve pertaining to the uncompressed index (UC) steeped by a factor of four. Moreover, the level of compression achieved when using characters was less because it took 3 additional bytes to represent the number of repetitions each element had. Overall, it was concluded that a change in data type of the attribute values of the projection index has an impact on the size of the uncompressed index, the compressibility of the index, and, as a consequence of the other, performance when using a decompression algorithm within the GPU.



Additional data was aggregated to this test to determine when it was not convenient to transfer the index compressed to the GPU when using integers. The additional data and outcome can be seen in Table #x. When the uncompressed size of the index is approximately 176K (45150 elements) or less it was fastest to send the uncompressed index.



== Performance in GPU with Computability 1.3 ==

Tests were also performed in a GeForceGTX285 NVidia GPU with 240 cores and 1 GB of RAM memory. This GPU has a higher computability, 1.3 as opposed to the previous 1.1, greater number of cores, faster cores, and more memory. The original test using the data distribution of sequentially incremented attribute values was also used, but this time the attribute values were integers. The outcome of this test was similar to the last test performed on the first GPU, but the gap between sending Uncompressed and decompressing within the GPU widened. Both Load Unbalanced (LU) and Load Balanced (LB) algorithms were faster approaches to make the index available rather than transferring the uncompressed projection index (see figure #x). Furthermore, it was noticed that the Load Balanced algorithm was more efficient than the load unbalanced approach for this set of data. The data is not friendly to the Load Unbalanced algorithm because the last element will repeat itself more than all the other, thus the work is not well distributed among all threads (see figure #x).



Figure #x. #TODO

Figure #x. Comparison between the Load Balanced and Load Unbalanced Algorithm, as the number of elements in the uncompressed index is increased.

The test was also performed by maintaining the size of the compressed index constant while doubling the frequency of each element on every iteration. This way, the threads in the load unbalanced algorithm would have the same amount of work. At first the Load Unbalanced algorithm was faster than the Load Balanced algorithm, but as the amount of repetitions increased the Load Balanced algorithm took less time to decompress. The Load Balanced algorithm (LB) was only influenced by the increasing size of the uncompressed array within the GPU. On the other hand, the Load Unbalanced algorithm (LU) not only is slowed down by the fact that it has to write on a bigger array, but it also does not distribute the work among all the possible threads. Since there are only 1024 different elements, only 1024 threads do work at a time. The LU algorithm also cannot make as good use of the GPU as the LB algorithm when the elements have too many repetitions. Sending the uncompressed index (UC) was only the better option for the first cases up to where the frequency was eight. At this frequency the sequence was still fairly small with 8192 elements in its uncompressed form. Notice also that there was no compression for a frequency of only two (See Table #x).



Finally, a test using the third data distribution was performed on the two algorithms. For this test the size of the index was fixed to 16777216 elements in uncompressed form, and the number of different elements in the index was varied, achieving different levels of compression. In other words, the fewer different elements the more compression achieved and the more different elements the lesser the compression. Using the previous experiment as reference, the amount of time taken to transfer the uncompressed index without an algorithm was taken to be 335.80 milliseconds. The outcome of this experiment shows that obviously when no compression is achieved, it is best to send the uncompressed index without decompressing. The compression percentage measure in our results is taken as the percentage in size of the compressed index as a fraction of the original uncompressed index. In the test results, notice that if the compression is of at least 50% on a large index, such as this one, both Load Balanced and Load Unbalanced algorithms performed better than sending the uncompressed index (See Table #x).



On a further look, one can also notice that the speedup for the load balanced algorithm is steady as the index becomes more and more compressed. The speed up for the load unbalanced algorithm, however, is very inconsistent. This is due to two competing factors: on is the decreasing time to copy a more compressed index on each iteration, and the other is the increasing time threads take by having less different elements that are repeated more frequently.

/\*On a deeper look, the amount of compression achieved with the first and second data distributions is very high extremely high; the data is reduced to in the lowest iteration 0.79% to 0.079% of the original data. However a lot of the real world data will not allow itself to be compressed that much. The second distribution was also \*/

== Conclusions and Future work ==

From the results, it was determined that the performance of the algorithms depends mostly on the GPU used and its computability, the level of compression of the index, and the nature of the data being processed.

Overall the percentage of compression of the index is the most important factor to determine whether the index should be sent compressed and decompressed in the GPU, or uncompressed without decompressing. In the tests performed, only sequences with a compression size of 50% and lower of the original size were attempted. Under these conditions it was not convenient to send the compressed index, and decompress it within the GPU for cases where the attribute values of the index were characters or when using integers with an uncompressed size of ~176K (45150 elements) and below.

The computability of the GPU allows for coalesced accesses for different access patterns to global memory, which were encountered in some phases of the Load Balanced algorithm. The greater number of cores and increased clockrate of this GPU allowed the decompression algorithms to accelerate in the performance tests. However, both algorithms do not take equally advantage of the higher computability of the GPU. The LU algorithm cannot take advantage of the greater number of cores when the number of elements is not varied enough as to launch a great number of threads. It also cannot take advantage of it when the repetitions are focused on a few single elements, as the threads handling those elements would be the only executing. The load balanced algorithm, on the other hand is not affected by these other factors and all its phases distribute the work among all threads.

Finally, the nature of the data refers to both the way the data is distributed and the data types of the attribute values. The size of the data type of the attribute value determines how much space is saved when compressing. In the case of characters, there would be no space savings when each character repeats itself five times or less. This is because the integer necessary to specify the times it repeats would take four additional bytes. The data distribution also determines the compressibility of the index; the more varied elements in the data the lesser the compression in a sorted index. The performance of the Load Balanced algorithm was only dependent on it from this effect. In contrast, the Load Unbalanced was also affected by the number of repetitions of each element. The greater the number of repetitions each element had the higher the workload of the threads writing the elements and the more the performance gain was reduced for it.

/\*The computability of the GPU helps arithmetic ratio by moving would-be memory operations to computational operations.

also an important factor in performance that

The computability of the GPU allows for coalesced accesses for different access setups that may not be in sequential order. Both algorithms performed badly on the GPU with computability of 1.1, and performed well on the GPU with computability 1.3.

The load balanced and unbalanced algorithms both are dependent on the level of compression of the projection index. This problem is inherent from the RLE compression scheme.

One possible avenue for future work is to compare the GPU's decompression 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 decompression when using a GPU.

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.

/\* Much analysis may still be done on the data as the percentage of \*/

/\* There are other possible performance enhancements related to use of different memories \*/

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.

Both algorithms suffer mostly from the amount of time it takes to actually move a poorly compressed index from the CPU to the GPU, such as when an index has each element repeat only two times. This problem is inherent in the data, as it is not always compressible under the RLE scheme and thus moving to the GPU and decompressing would be a waste of time. However there were good cases where compressing and decompressing the projection index proved to save time.

Overall, it was concluded that a change in data type of the attribute values of the projection index has an impact on the size of the uncompressed index, the compressibility of the index, and, as a consequence of the other, performance when using a decompression algorithm within the GPU.

It was determined from the results that the performance of the algorithms depends mostly on the GPU used and its computability, the percentage of compression of the projection index, and the nature of the data being processed

Improvements in performance for these two algorithms were observed in both GPUs with computability of 1.1, the algorithm showed no signs of performance improvement for characters, mainly because of the number of cores and their clock rates were too small to compete with the speed of the GPU’s bus.

\*/