Run-length Encoding on Graphics Hardware

Ruth Rutter

Project Report for completion of a
Master of Science in Computer Science.

University of Alaska, Fairbanks
Fall 2011

Committee:
Orion Lawlor
Jon Genetti
Chris Hartman

Table of Contents

Abstract

In this report, we present a series of improvements to the basic run-length encoding algorithm. Several modified algorithms are generated, from basic CPU single-threaded compression to a highly parallelized version on NVidia CUDA hardware. We also investigated changes to memory access locality in the parallel versions. With a comparison of the various compression times and their corresponding data transfer times, we conclude that despite RLE being a sequential algorithm, parallelizing it on graphics hardware provides a decent speedup.

1. NChilada

This project was a subsection of NChilada, a collaborative project between the universities of Illinois at Urbana-Champagne, Washington, and Alaska at Fairbanks. The Washington group, led by Tom Quinn, provided the cosmology expertise, L.V. Kale's Illinois group provided parallel computing expertise and the runtime system, and Dr. Orion Lawlor's Fairbanks group provided the rendering expertise.

NChilada is an n-body simulator focused on the simulation of galaxy and planet formation and dark matter[1]. It produces a list of 3-dimensional floating point coordinates that represent the locations of galaxies within the simulated universe. This data set is several gigabytes of continuous floating point data.

The production of the data sets is time intensive, requiring approximately eight hours to produce a single image. This means that particle data sets are typically simulated offline, then rendered and explored interactively.

To make the data usable, the visualizer, Salsa, uses an array of graphics cards to render the particle lists into a 3D volume [2]. This volume is a discretization of the continuous universe provided by the simulator, and represents each galaxy as a grayscale color point within a 512 cubed volume. Each element in the volume is a single byte, so the volume is approximately 128MB. Each graphics card produces a sparsely populated data structure the size of the whole volume, but with only a subset of the data. These partial renderings are passed to a central controller and merged to produce the final result.

1.1 Problem

The usability of the simulator is currently limited due to the latency of encoding and transporting the data to the client. The goal of this project was to decrease the time required by streamlining a subsection of the rendering process.

The main problem was that 128MB of data requires substantial time to transfer. On a typical cluster with a 10MB/s connection speed, it takes 12.8 seconds. Rotation can be executed on the client, but any other change in the viewpoint, including zooming and changing the data content displayed, forces frequent recalculation and re-rendering

of the data set. At 12.8 seconds per modification, it is impossible to attain a usable framerate, so the time required needs to be reduced.

In total, the data begins as a list of points from the simulator, then is passed to the renderer, where it is distributed among the GPUs. They render it, and transfer it from the graphics cards to the host, and from there to the central controller. The central controller merges them and sends them to the client.

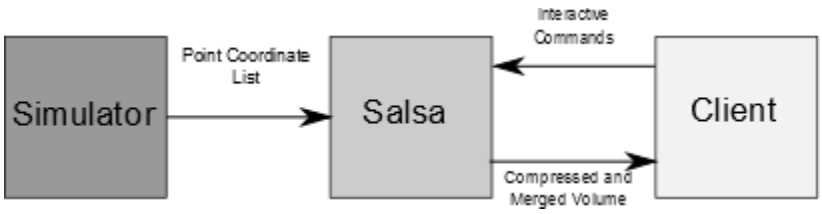

Figure 1. Overview of System Structure.

This many transfers accrues a sizeable time cost, impairing the user's ability to explore the data. For this project, we were most concerned with the time to transfer from the rendering GPUs to the central controller.

1.2 Solution

Since the goal is to decrease the transfer time, there are two variables that could possibly be modified. Either the transfer speed needs to increase, or the amount of data to transfer needs to decrease. Since the latter is more easily attainable, the first obvious improvement is to implement some form of compression for the data.

There are a limited number of options for compression. This is further restricted by the fact that scientific data should not be modified. Some options for compression are presented in the table below.

	Uncompressed	RLE	Gzip	JPEG
Time (ms)	0	~556	~1550	~6400
Size	~128MB	~3MB	~1.5MB	~1.7MB
Transfer time (s) (10 MB/s)	12.8	0.3	0.15	0.17
Lossless	Yes	Yes	Yes	No

Of these four options, no compression is too slow on transfer time, Gzip is slow, and JPEG is both lossy and slow. Run-length encoding, or RLE, is the best option presented. It's relatively fast, simple, and unlike many other algorithms, lossless. It produces compressed files consisting of pairs of values, the first holding the run of a data value, the second holding that value. For example, a run of *x* bytes with a value of 0 would be compressed to two bytes: *x* and 0.

The most basic implementation of RLE has a worst case of producing a file twice the size of the original, but can easily be modified to avoid this. For most data it is efficient, and when the input file is mainly monochromatic, it performs quite well. Since the data expected in this project is a vast majority black, RLE should produce good results.

1.3 Current Performance Analysis

The current implementation uses a single-core, single-threaded algorithm for compression. The times for best, worst, and average case are laid out in the table below. The best case is a completely black data set, and the average case uses an actual volume from the renderer. Worst case is a set where each value differs from the next, in this case a repetitive count from 0 to 254 and from 0 to 255. For comparison, the times from compressing the same files with gzip are also provided.

File	CPU Single Thread (s)	Gzip (s)
Zero	0.540	1.15
Sequence254	1.292	1.16
Sequence255	1.293	1.16
Voxel3D Potential	0.556	1.55

This naive compression doesn’t scale well with noise in the data, and on this testing system, takes over 500ms. At best, and not allowing for any other time cost, this delivers 2 frames per second. Adding in the other time requirements would produce a rate of seconds per frame. While this simple compression provides a great improvement over transferring raw data, it still does not provide a sufficiently quick transfer for useful interaction with the data.

1.4 Parallelizing

The next obvious improvement is to parallelize the compression algorithm. While the RLE algorithm isn't inherently parallel, it can be implemented parallelized with only a small loss of compression. This may be done by segmenting the input file and compressing each section sequentially, independent from the rest. Given that the compressed length for each section isn't known before compression, a secondary step will be required to stitch the individual results together.

CPU compression was not the focus of this project, so comparable times were generated with a parallelized algorithm that accessed each element in a correctly sized array. This returns the same compression time of approximately 91 milliseconds independent of data, benchmarked on an Intel Core i5-2400 3.10GHz quad-core machine.

2. GPU

The most common and economical way to massive parallelism is the GPU. Excluding Microsoft products, there are three major interfaces to access this parallelism: OpenGL, OpenCL, and CUDA.

OpenGL is the oldest and most prevalent of the three [3]. It was developed for graphical purposes, and allows easy control of individual pixels. This means that it is easily adapted to handle data calculations that are neighbor-independent, but is not so useful for other applications. While OpenGL is the most stable, it does not allow random memory accesses, and therefore is greatly limited.

OpenCL was developed as a cross-platform programming standard for executing programs on peripheral devices [4]. Unlike OpenGL, it provides flexible access to memory. While the concept is quite interesting, at the start of this project, it was still young and very unstable.

CUDA is Nvidia's Compute Unified Device Architecture, developed to allow easier computation on peripheral devices such as graphics cards [5]. At the time of this project, CUDA was several years old, and therefore more widespread and stable than OpenCL.

It also provides random access to elements, and so is more flexible than OpenGL.

2.1 CUDA

For languages that are compatible with CUDA, constructs called device kernels may be written into the main host code. They are written in the same language as the host code, for this project C++, in which they are only distinguished from other functions by the __global__ keyword and the syntax of the function call. These kernels are compiled and transferred to the device for execution, instead of running on the host.

CUDA allows for the user to specify how many threads they want initialized for each kernel. These threads are arranged in blocks, which are then organized into grids. There is a strict limit of threads per block, which varies from card to card, but the number of blocks per grid is presented as unlimited. In reality, it varies from GPU to GPU; any threads beyond the limit are faked, with their work being swapped in and out of real threads.

Threads are numbered within a block from 0 to 511. If a problem requires more than one block, this will lead to ambiguity when assigning work, so thread identification is also calculated using the block identification number and the block dimension number.

Memory, both host and device, must be allocated by the host, and is accessed through host and device pointers, respectively. Data is shared between the two using a CUDA-specific memcpy function.

There are two main drawbacks to compression on the GPU. One is that with mass parallelization, the complexity increases. Tracking which section of data each thread needs to access based solely on the thread id can be complicated. The second is the data transfer time necessary to move the data to and from the GPU. In this project specifically, transfer to the GPU is irrelevant, since the data is there already from being rendered by the visualizer. This actually provides an automatic benefit to compressing on the GPU, since anything it produces will cost less time to transfer to the host than the uncompressed 128MB volume.

2.2 Prior Work

Prior work on similar projects includes Ana Balevic's research into and implementation of various parallel compression algorithms for GPUs [6]. One of her implementations also explored RLE on the GPU, but used a different modification to allow for parallelization of the sequential algorithm. Instead of assigning each thread a section of data to encode, she assigned each element to a thread, which set a flag if the element was not equal to the previous element. Running a parallel prefix sum operation on this array of flags provides both the run lengths and the output index, at which point the threads output the symbols and counts.

The algorithm she presents is slightly more complex than the RLE algorithms used here, and is not targeted at arrays of the size expected in this project. While it is more efficient to declare as many threads as possible to break down the work and hide memory latency, there is a limit to the usefulness of this approach. At some point, more threads are simply for the benefit of fitting the calculations neatly, and provide no real functionality. Declaring a thread per each element of the volumes in this project may be approaching that limit.

Another related project explored hybrid storage of large volumes of detailed data in graphics memory. Wilson, Ma, and McCormick present a hybrid rendering technique for maintaining the integrity of highly detailed data while decreasing the size of the data set [7]. While this provides some interesting options, it is not entirely applicable to this specific project, since the data expected is not large enough or sufficiently fine-grained to require using their approach.

2.3 Process

The first step is to allocate the necessary memory on both the host and the device. This includes an input buffer on the host and the device, an output buffer on both, and a separate array on the device to hold values necessary for allowing parallelism. The file is then loaded into the host buffer and memcopied to the corresponding device buffer. This memcopy may not be the entire data set, depending on the parameters provided to the program based on the quality of the hardware. It could instead be handled by a loop that passes in a chunk, usually 1/512, of the set.

Conceptually, the rendered volume can be thought of as a cube with an edge length of 512 (not necessarily). The most obvious way to break this down into manageable sections is to create 512*512 threads, and assign each 512 bytes, or one row in the cube, to compress.

The actual encoding and compression is executed in two device kernels. Kernel 1 is initiated following the first memcopy. It defines the number of threads specified for encoding, in most cases 512*512. It then assigns each thread a location on one face of the cube and encodes to the width of the cube.

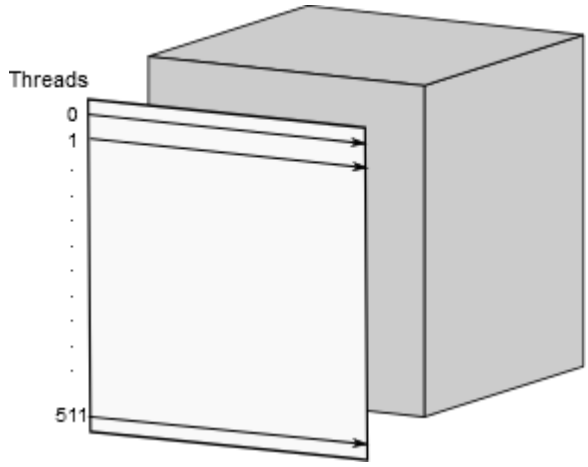

Figure 2. Thread assignment for one layer.

To allow for the limitations of what a single byte can store, each row is broken down into two sections of length 256. Because of this, each thread will produce at least four bytes of encoded data. Each thread is running independently, which means that the length of encoded data produced by previous threads is unknown. To resolve this problem, each thread writes the number of encoded bytes it produced into a separate array, and writes the actual bytes into an output array at the same index it read uncompressed data from. This produces an array double the size of the input buffer that is very sparsely populated. At this point, kernel 1 returns control to the host.

The host runs a thrust-provided parallel prefix inclusive sum calculation on the array containing the number of bytes written by each thread. As the inclusive sum includes the element being summed to, the value in each element will be equal to the number of bytes written by all previous threads as well as the corresponding thread. This provides the number of bytes the thread will write by subtracting the preceding value, which doubles as the output index.

The host initiates kernel 2, which will stitch together the sections of encoded data in the output buffer and produce the final compressed data. It defines the same number of threads as kernel 1, and each thread is responsible for writing one row of encoded data

to the output location provided by the buffer of output addresses. The location and the amount of data to copy is handed to a memcpy, which inserts the data into the correct spot in the output buffer. The output buffer is copied back to the host and staged for transfer to the client.

2.4 Memory Locality

When visualizing traversing a file as a two dimensional construct, access switches from a linear stream to a series of rows. The problem with compressing in rows in parallel is that the memory accesses have no locality. When an algorithm is parallelized, locality becomes a balance between local thread locality and a more global inter-thread locality. Increasing thread locality does not necessarily benefit global locality; in fact, if the thread locality is too high, loss of synchronicity between threads will decrease the global locality.

In this case, each thread's memory access is 512 bytes from its neighbours, so on every read, each thread will have to wait for its memory to be loaded again. Given the absolute simplicity of the calculations, this is obviously a bottleneck.

In order to streamline this, the memory the threads access must be more compact. One method of obtaining this effect is to pass each thread a column rather than a row. For each set of 512 threads, their memory accesses are adjacent, decreasing the wait time.

If this compression pattern handles the output in the same manner as row-based compression, the final data is transposed from how it should be. This occurs because the individual encoded chunks, which are now produced from columns, are still written into rows in the first output buffer before being stitched together.

3. Results

3.1 Performance

These timing runs were mostly performed on an NVIDIA GeForce GTX 460M, with an Intel Core i7-2630QM. The CPU parallel approximation was timed on the Intel Core i5 detailed previously.

As shown in the table below, modifying the pattern to allow for higher memory access locality provided a decent speed up.

File	Row-compressed (ms)	Column-compressed (ms)
Zero	328	51
Sequence254	2009	1865
Sequence255	2008	51
Voxel3D potential	338	60

For comparison between the parallelized CPU version and the CUDA column-oriented version, the relevant times are presented below.

File	CUDA column-compressed (ms)	CPU, OpenMP (ms) (Intel Core i5-2400)
Zero	51	91
Sequence254	1865	91
Sequence255	51	91
Voxel3D potential	60	91

The transfer time from the GPU to the host for the GPU-compressed file is approximately 1.5ms. The same transfer for the CPU-compression algorithm is pre-compression, and is approximately 32ms. The transfer time from the host to the central controller is similar for both, allowing comparison to remain simply between the accumulated compression and GPU-host transfer time. For the GPU version, the final time is 61.5ms, and for the CPU version, the final time is 123ms. This provides us a speedup of 2x on this testing system.

3.2 Future Work

There are several improvements that could be made to this specific approach. It currently uses a call to the THRUST library, which, like any generic templated function,

requires extra time. This might be possible to avoid by writing a custom parallel prefix sum function that uses assumptions about the incoming data. Another possible improvement would be to switch the compression pattern from columns back to rows, but assign threads their data starting at the front face of the cube, so all memory accesses are adjacent. This would provide the strongest locality for each read, but might prove problematic by increasing the complexity.

There are also improvements that could be made beyond this subsection. One further improvement could be to streamline the merging of the compressed files. It might be worth exploring if parallelization would benefit this process. Another option might be to try a volume-focused compression algorithm. Since the data produced is known to be mostly black, compressing it by filling the inter-point spaces with small volumes might be more efficient.

3.3 Conclusions

Parallel RLE on the GPU is faster than the equivalent algorithm on the CPU, as long as memory access locality is high, in this case by accessing columns instead of rows. Modifications must be made to allow such a sequential algorithm to function parallelized, but the speedup is sufficient to compensate for the transfer time between the device and host and the increased space requirement. There are several modifications to the final algorithm presented here that could be explored in the future, but overall, this project has met its goal of streamlining this stage of the renderer.

References

[1] Gioachin F, Kale L.V., (2009) Dynamic High-Level Scripting in Parallel Applications. In: IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1-11.

[2] Quinn T, Kale L, Gioachin F, Lawlor O, Lufkin G, Stinson G (2004) Salsa: a parallel, interactive, particle-based analysis tool. Poster at Supercomputing 2004. http://charm.cs.uiuc.edu/posters/CosmologySC04.pdf

[3] The Khronos Group (2011) The OpenGL Graphics System: A Specification (Version 4.2 (Core Profile) - August 8, 2011). http://www.opengl.org/registry/doc/glspec42.core.20110808.pdf

[4] NVidia (2011) OpenCL Programming Guide for the CUDA Architecture. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf

[5] NVidia (2011) NVidia CUDA C Programming Guide, Version 4.0. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[6] Balevic, Ana (2009) Fine-Grain Parallelization of Entropy Coding on GPGPUs. http://tesla.rcub.bg.ac.rs/~taucet/docs/papers/HIPEAC-ShortPaper-AnaBalevic.pdf

[7] Brett Wilson, Kwan-Liu Ma, Patrick S. McCormick (2002) A Hardware-Assisted Hybrid Rendering Technique for Interactive Volume Visualization. http://maxradi.us/documents/volvis2002/

Code

Kernel.cu

Excerpts from the program showing the kernels and the compression calculations.

```
// Kernel 1, Row-oriented
__global__ void rleEncode(const char* inBuff, char* outBuff, int* lengthBuff)
{
  int rowLen = blockDim.x;
  int block_offset = blockDim.x * blockDim.x * blockIdx.x;
  int thread_offset = block_offset + (threadIdx.x * rowLen);
  int bytes_offset = blockDim.x * blockIdx.x + threadIdx.x;

  int b_data_offset = (rowLen*2)*(blockDim.x * blockIdx.x + threadIdx.x);

  int out_bytes=0;//total bytes written out by this thread

  char value = inBuff[thread_offset]; //current value
  int count = 1; //count for current value

  for (int pos = 1; pos < rowLen; pos++) {
     char new_value = inBuff[pos + thread_offset];
     if ((new_value == value) && (count < 255)) {
        count++;
     } else {
        outBuff[b_data_offset + out_bytes++] = count;
        outBuff[b_data_offset + out_bytes++] = value;
        value = new_value;
        count = 1;
     }
  }
  outBuff[b_data_offset + out_bytes++] = count;
  outBuff[b_data_offset + out_bytes++] = value;
  lengthBuff[bytes_offset] = out_bytes;
}
```

```
// Kernel 1, Column-oriented
__global__ void rleEncodeTransposed(const char* inBuff, char* outBuff, int* lengthBuff)
{
  int rowLen = blockDim.x;
  int block_offset = blockDim.x * blockDim.x * blockIdx.x;
  int thread_offset = block_offset + threadIdx.x;
  int bytes_offset = blockDim.x * blockIdx.x + threadIdx.x;

  int b_data_offset = (rowLen*2)*(blockDim.x * blockIdx.x + threadIdx.x);
  int out_bytes=0;//total bytes written out by this thread

  char value = inBuff[thread_offset]; //current value
  int count = 1; //count for current value

  for (int pos = 1; pos < rowLen; pos++) {
    char new_value = inBuff[pos * rowLen + thread_offset];
    if ((new_value == value) && (count < 255)) {
      count++;
    } else {
      outBuff[b_data_offset + out_bytes++] = count;
      outBuff[b_data_offset + out_bytes++] = value;
      value = new_value;
      count = 1;
    }
  }
  outBuff[b_data_offset + out_bytes++] = count;
  outBuff[b_data_offset + out_bytes++] = value;
        lengthBuff[bytes_offset] = out_bytes;

}

//kernel 2
__global__ void Defrag(char* inBuff, char* outBuff, int* lengthBuff){

  int read_block_offset = blockDim.x * blockDim.x * 2 * blockIdx.x; //squared or not?
        int read_offset = read_block_offset + threadIdx.x * blockDim.x * 2;

        int length_block_offset = blockDim.x * blockIdx.x;
        int length_offset = length_block_offset + threadIdx.x;

        int thread_bytes;
        int output_offset;
```

```
	if(read_offset == 0){
		output_offset = 0;
		thread_bytes = lengthBuff[length_offset];
	}
	else{
		output_offset = lengthBuff[length_offset-1];
		thread_bytes = lengthBuff[length_offset] - lengthBuff[length_offset-1];
	}

	memcpy(outBuff+output_offset, inBuff+read_offset, thread_bytes);
}


int callIncSum(int* lengthBuff, int lengthBuffSize){

	thrust::device_ptr<int> thrust_length_buff = thrust::device_pointer_cast(lengthBuff);
	thrust::inclusive_scan(thrust_length_buff, thrust_length_buff+lengthBuffSize,
thrust_length_buff);

	return thrust_length_buff[lengthBuffSize-1];
}
```

CudaRLEEncoder.cpp

Excerpts from the program showing the kernel calling structure.

```
CudaRLEEncoder::CudaRLEEncoder(int chunk_width, int chunk_count) throw (cudaError_t)
{
	this->chunk_width = chunk_width;
	this->chunk_count = chunk_count;
	this->thread_count = chunk_width * chunk_count;
	gpu_in_buff = gpu_out_buff = 0;
	gpu_final_data = 0;
	gpu_length_buff = 0;
	gpu_in_size = chunk_width * chunk_width * chunk_count;
	gpu_out_size = thread_count * (2 * chunk_width);
	length_buff_len = thread_count;
	total_kernel_time = total_aggregate_time = 0;

	throwOnError( cudaSetDevice(0), "Failed to set cuda device 0." );  throwOnError(
cudaMalloc((void**) &gpu_in_buff , gpu_in_size), "Failed to allocate gpu input buffer." );

	throwOnError( cudaMalloc((void**) &gpu_out_buff , gpu_out_size), "Failed to allocate
gpu output buffer." );

	throwOnError( cudaMalloc((void**) &gpu_length_buff , length_buff_len*sizeof(int)
), "Failed to allocate gpu length buffer." );

}

int CudaRLEEncoder::encodeChunkSet(char* inBuff, int inPos, char* outBuff, int outPos) throw
(cudaError_t)
{
	float time;
  cudaEvent_t start, stop;

	throwOnError( cudaEventCreate(&start), "Failed to create start event.");
	throwOnError( cudaEventCreate(&stop), "Failed to creat stop event.");

	throwOnError( cudaMemcpy(gpu_in_buff, inBuff + inPos, gpu_in_size,
cudaMemcpyHostToDevice), "Failed to copy host buffer to gpu.");

	throwOnError( cudaEventRecord(start, 0), "Failed to record start event.");
```

```
	invokeCudaKernel(chunk_count, chunk_width, gpu_in_buff, gpu_out_buff,
gpu_length_buff);

	throwOnError( cudaDeviceSynchronize(), "Failed synchronizing cuda device after kernel
invocation.");

	int bytesWritten = callIncSum(gpu_length_buff, length_buff_len);

	throwOnError( cudaMalloc((void**) &gpu_final_data , bytesWritten), "Failed to allocate
device final data buffer." );

	gpuCudaKernelDefrag(chunk_count, chunk_width, gpu_out_buff, gpu_final_data,
gpu_length_buff);
	throwOnError( cudaEventRecord(stop, 0), "Failed to record stop event.");
	throwOnError( cudaEventSynchronize(stop), "Failed to synchronize on stop event.");

	throwOnError( cudaDeviceSynchronize(), "Failed synchronizing cuda device after defrag
kernel invocation.");

	throwOnError( cudaEventElapsedTime(&time, start, stop), "Failed to compute elapsed
time.");
	total_kernel_time += time;

	throwOnError( cudaMemcpy(outBuff+outPos, gpu_final_data, bytesWritten,
cudaMemcpyDeviceToHost), "Failed to copy gpu out buffer to host.");

	return bytesWritten;
}
```