Lossless Compression of Variable-Precision Floating-Point Buffers on GPUs

Jeff Pool, Anselmo Lastra, and Montek Singh*
University of North Carolina

(a) Color (b) Depth (c) Geometry

Figure 1: *Our unified lossless compression/decompression system can handle any type of data. We were able to obtain bandwidth savings of 1.5x, 7.7x, and 3.3x for the color, depth, and geometry data, respectively, shown above.*

Abstract

In this work, we explore the lossless compression of 32-bit floating-point buffers on graphics hardware. We first adapt a state-of-the-art 16-bit floating-point color and depth buffer compression scheme for operation on 32-bit data and propose two specific enhancements: dynamic bucket selection and a Fibonacci encoder. Next, we describe a unified codec for any type of floating-point buffer: color, depth, geometry, and GPGPU data. We also propose a method to further compress variable-precision data. Finally, we test our techniques on color, depth, and geometry buffers from existing applications. Using our enhancements to an existing technique, we have improved bandwidth savings by an average of 1.26x. Our unified codec achieved average bandwidth savings of 1.5x, 7.9x, and 2.9x for color (including buffers incompressible by past work), depth, and geometry buffers. Even higher savings were achieved when combined with our variable-precision technique, though specific ratios will depend on the tolerance of the application to reducing its precision.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors;

Keywords: compression, floating-point, variable precision

1 Introduction

A graphics processing unit's (GPU's) performance is ultimately limited by many factors. Historically, memory bandwidth and computational power have been limiting factors for different workloads. Recently, power consumption became as important. Though chips have gotten more capable in terms of number of processors and computing power, available memory bandwidth has not increased at the same rate, a trend that is expected to continue [Keckler et al. 2011]. Furthermore, there is a limit to the power that can be used to drive a chip; there are practical considerations such as heat, fan noise, and power supply limitations to take into account. In this work, we focus on a method to reduce memory traffic by compressing the data transferred to and from off-chip buffers. While our approach (and the state-of-the-art in lossless buffer compression [Ström et al. 2008]) does not reduce the amount of memory used to store the data, it directly impacts the amount of data that is transferred. As a result, both memory bandwidth requirements *and* energy consumption are reduced.

Data compression in GPUs has become commonplace in two areas: texture and buffer compression. Texture compression is a particular type of general buffer compression that is performed once off-chip during asset authoring; these compressed textures are then decompressed on-chip many times during execution of the graphics program. So, texture compression is usually asymmetric, and can often be lossy, since a human is in the loop during compression to verify that results are acceptable. As buffer compression is performed on-chip, it is often much simpler than texture compression. While there has been much prior work on buffer compression, most of it is targeted to integer formats. This work instead focuses on the relatively new floating-point buffer formats, for which very little prior work exists.

We develop a general-purpose lossless compression scheme that is able to handle data from any source: color, depth, geometry, and GPGPU buffers. This is a departure from most past techniques, which go to great lengths to exploit knowledge of the buffer's contents. As GPUs become more general-purpose, we believe that such codec specialization hinders generality. Our goal is to remain buffer-agnostic so that we can reasonably compress any set of data. We target lossless compression in order to serve general data producers and consumers; we cannot assume that a general application can handle lossy compression without destroying its functionality. As this compressor is expected to be used heavily in rendering color and depth data, we retain random-access read- and write-ability to the same degree as in past work: 8x8 tiles are stored together, and geometry is exposed at a similar granularity.

This work is also part of a larger whole. We have built upon Hao and Varshney's [2001] variable-precision integer and fixed-point rendering techniques and explored the energy savings possible in computation by designing hardware [2011a] and using it in variable-precision vertex [2008] and pixel [2011b] shaders. Here, we focus on a remaining large consumer of energy in GPUs: off-chip memory traffic.

*email: {jpool,lastra,montek}@cs.unc.edu

The state-of-the-art in lossless compression of GPU floating-point buffers known to the authors targets 16-bit floating-point color and depth data [Ström et al. 2008]. We examine this work and its performance on 32-bit floating-point buffers to serve as a comparison to our general-purpose codec architecture. We also suggest two enhancements, applicable to both existing work and our proposed compressor, that lead to higher compression ratios. The specific contributions made by this paper are as follows:

- a unified codec architecture capable of handling any type of buffer without regard for its contents,
- dynamic selection of compression buckets,
- examination of an alternate encoder for compressing residuals, and
- range reduction for variable-precision data.

The rest of the paper is organized as follows. Section 2 details related research in the area of compression of numerical data, particularly in the context of graphics applications. We briefly present the state-of-the-art approach to lossless compression of floating-point buffers in Section 3 before discussing our own general-purpose compressor's design in Section 4. We then propose enhancements applicable to either compression scheme in Section 5. We propose a modification to take advantage of variable-precision data in Section 6. Our experimental setup and data sets are given in Section 7. Section 8 presents the resulting compression rates and a discussion of our findings. Section 9 presents our conclusions and areas of future work. Finally, auxiliary material in the ACM Digital Library contains details regarding Fibonacci encoding: the algorithm, expected benefits, and results.

2 Related Research

Compression of numerical data has been well-studied. Common to nearly every approach is the encoding of errors of predicted values rather than the values themselves. This has been used in several approaches to compressing floating-point scientific data [Lindstrom and Isenburg 2006; Ratanaworabhan et al. 2006; Burtscher and Ratanaworabhan 2009; Hidetoshi and Yokoo 1994]. All the viable methods for compressing geometry, color, and depth buffers in graphics applications known to the authors use this basic technique with specialized schemes for particular applications.

2.1 Geometry Buffer Compression

Compressing geometry data can be more complicated than just compressing coherent position values due to associated connectivity and property data. Connectivity data indicate which vertices are grouped together to form faces in a three-dimensional mesh, and property data determine the colors, normals, binormals, any number of texture coordinates, etc. for these vertices and faces. Deering [1995] presented the first major work on geometry compression which approached all of these facets of geometric data in a lossy manner, seeing compression rates of 6-10x. More recently, there has been work on the lossless compression of geometry [Isenburg et al. 2005], which led to compression rates of 30-50%.

The details of geometry compression used in commercial hardware are not readily available. Examining patents awarded for the compression of geometric data shows that most work in the area deals heavily with connectivity data [Gruetzmacher 2010], which is too complicated to implement in hardware. We found few patents on geometry compression issued directly to makers of commercial GPUs. Wittenbrink and Ordentlich [2005] take advantage of data that remains uniform for a series of vertices or faces in order to reduce transmitted data. Other publicly-available information pertains to geometry and tessellation shaders [Goel and Martin 2009; Ramey et al. 2008; Dmitriev and Moreton 2011]. (These techniques are two types of "geometry amplification," which generates new vertex data from existing vertex data. Since geometry amplification is orthogonal to numerical compression, they can be used together for even higher compression rates.) Danilak cites pixel compression specifically, but makes no mention of compressing vertex data, despite its storage and transmission on and to multiple GPUs [2009].

2.2 Color/Depth Buffer Compression

Historically, color and depth buffers in GPUs have had primarily integer formats. Rasmusson et al. [2007] provide a thorough summary of the state-of-the-art in color buffer compression, with an emphasis on integer formats (RGBA8). A similar summary exists for depth buffer compression [Hasselgren and Akenine-Möller 2006]. Hardware-accelerated decompression of compressed formats are mostly based upon S3's Texture Compression (S3TC, or DXTC) [Iourcha et al. 1999]. Many formats used for different purposes have sprung from this format [Microsoft 2011], as well as incremental improvements [Yifei and Dandan 2010]. Most of these texture compression schemes are asymmetric; the data are compressed just once when authored on a single host CPU but decompressed many times on the GPU. So, compression can take an arbitrary amount of time, but decompression must be fast and simple.

Compression of floating-point buffers in hardware is a recent development with the advent of floating point buffer formats. Commercial techniques have not been made public. Strom et al. [2008] present a method for compressing 16-bit floating-point color and depth buffers in a unified manner, with several limitations. That scheme does not allow negative values and assumes the alpha channel is 1.0f. Later work [Wennersten and Ström 2009] addressed compression of the alpha channel, but using these two separate compressors for color data introduces complexity. Further work has been performed to allow for lossy compression of color buffers [Rasmusson et al. 2009], which further decouples color and depth buffer compression.

3 Description of the Current State-of-the-Art

Before presenting out novel approaches, we first discuss the design and operation of an existing lossless floating-point color/depth buffer compression/decompression scheme [Ström et al. 2008].

The input to the compressor block is an 8x8 tile of RGB(A) pixels or depth values represented as 16-bit floating-point numbers, and the output is a stream of bits that will be tagged with how the input was was compressed: uncompressed, "fast-cleared" (consisting of a single value), or compressed to 25% or 50% of its original size. Each tile's tag will depend on the compressability of its contents and is stored in a "tile map" that maps tiles to their compression rates to enable random accesses (mandated by graphics APIs). These floating-point numbers are interpreted as integers so any arithmetic performed is exact and not subject to rounding errors, as might be the case with floating-point arithmetic. The 8x8 tile is divided into four 4x4 tiles for further processing. For color data, the red channel is encoded, then the difference between the green and the red, and finally the difference between the blue and the green channels, to take advantage of any correlation between color channels. Tiles that include negative numbers or alpha values less than one are ignored and stored uncompressed.

Each 4x4 tile is handled similarly. Starting at the top-left value, the difference between one value and the next is computed along the top row and left column, implicitly predicting each value from the

preceding one. These integer differences are encoded (as described in the next paragraph) with the hope that the difference between values will be small and therefore more compactly represented. A more complicated prediction scheme intended to minimize these differences is used for the remaining 3x3 values, choosing either the value above, to the left, or an average of these two as the predicted value. This is intended to handle cases where there is a discontinuity within the tile that would lead to poorly predicted values. A guide bit for values along this discontinuity indicates how the values are to be predicted (and reconstructed). Further, there may be a "restart value" (requiring a 4-bit position and 15-bit value) to indicate a more fitting starting point for values on the other side of the discontinuity. To better handle discontinuities, the entire tile may be rotated 90 degrees counter-clockwise (indicated with a single bit). The beginning of the encoded data stream, then, is a single restart bit, optional restart position and value, a rotate bit, and the top-left value.

Difference values are encoded with a Golomb-Rice encoder. Since the input values were all positive, the differences can be represented with 16 bits, which are then mapped to the positive domain with a simple transformation, which ensures that numbers with similar magnitudes will appear sequentially. Thus, values of -1 and +1 will both have small representations. To encode a value, it is divided by some power of two, 2^k, to yield a quotient and a remainder. Unary encoding is used to encode the quotient, q: a series of q ones with a terminal 0. The k bits of the remainder are stored in binary. The best k value for each 2x2 block in a 4x4 sub-tile is found through exhaustive search and then shared among the four values, stored before the quotient and remainder data. The rest of the encoded stream is as follows: k values (16 bits), a variable number of guide bits, and Golomb-Rice bits (quotient and remainder data).

4 General-Purpose Compressor Design

Our general-purpose compressor is based on the approach described in Section 3. At a high level, our design is meant to handle any type of data: not only color and depth, as in past work, but also geometry or general-purpose data. As GPGPU applications become more prevalent, we believe it would be of great benefit to the simplification of hardware if a single compression/decompression block could serve all clients with good compression rates. To support this, we must accept negative numbers to our compressor, be able to handle data of various layouts (not just square tiles), and still allow for random access to the data. We describe the modifications necessary for each of these capabilities in following sections.

As shown below, due to the general nature of our design, we do not need the guide bits, restart bits (flag/value/position), or rotate bit. This simplifies both the hardware necessary for our design and the encoding/decoding effort. Further, we do not need to store these bits, which saves space; though, of course, we cannot use them to store information. This means that the lack of these bits does not translate directly into a more compact encoding.

4.1 Handling negative values

A limitation of existing approaches is that they can only handle positive values [Ström et al. 2008]. In our approach, we want to handle negative values as well, but this is not straightforward. When input floating-point values are negative, they will have information in the sign bit. When operating under the stipulation that all input values to be compressed are positive, this sign bit is always zero, and so the difference between two floating-point numbers interpreted as integers will never overflow. With varying sign bits, this is not always the case, and overflow can occur when subtracting or re-mapping the difference values, a necessary step in many encoding schemes, including Golomb-Rice encoding. To avoid overflow, we specify that subtraction and remapping take place immediately prior to encoding. Integer hardware that can handle numbers one bit larger than the values themselves can be used to keep track of this overflow without needing to handle storage and transmission of a non-power-of-two number of bits, which would be necessary if further processing is to be performed on residual values. Since we perform direct encoding and do not need to compute guide bits and restart values, we can make this simplification that is unavailable to past work. While the discontinuity around 0.0f when floats are interpreted as integers will lead to disproportionately large residuals and hurt compression rates, functionality is not affected.

4.2 Arbitrary numbers of attributes

Rather than hard-wiring our compressor to deal with some fixed tile size, we must allow it to handle buffers of any layout. To this end, we let the compressor work on vectors of data instead of tiles. To construct a vector of values, the stride of the data is necessary. For geometry data, this stride is readily available from the vertex buffer descriptor. An example will make this approach clear. Consider a vertex attribute buffer containing (*x,y,z*) positions, (*xn, yn, zn*) normals, and (*u,v*) texture coordinates for each of *N* vertices. The stride of this data layout is (3+3+2) values * 4B = 32B per vertex. Using this stride in our scheme, then, the *x* value of the positions would first be compressed. The first value, *x0*, is stored uncompressed, followed by the *encoded* difference between the first and second values, *x1-x0*, and so on. Incrementing the offset by 4B and repeating the stride allows us to encode the remaining attribute values.

This approach has two major benefits. First, it does not assume any particular shape of data; it is simply repeated for each vector. Second, it exploits much of the available coherence in the data. It is much more likely that neighboring *x* values will be related, rather than the *x* and *y* values of a particular vertex. In the case of color data (*r, g, b, a*), a tile is stored one-dimensionally in memory, not in two dimensions. This means that it will be input to the compressor as a series of pixels, and each vector of data will be comprised of a single color component if the correct stride is observed. This addresses an issue noted in past work [Wennersten and Ström 2009] - existing compression formats assume coherence between color channels where there may not be any. However, if there *is* coherence between color channels, we will not exploit it without further effort, as past work [Ström et al. 2008] has done.

4.3 Enabling random access

Random access of input data is key in many common uses of graphics hardware. For color and depth data, hardware access is at the tile level. Geometry has no such predefined subdivision finer than an addressable buffer. The user, though, is free to start rendering at any point in the buffer or update only certain parts of it between rendering commands, such as when animating a subset of particles that are still in a "live" state in a larger particle system. To allow this, we simply compress a subset of the buffer at a time. For instance, in the above example with *N* vertices, we will not compress all *N* vertices at once. Instead, the first *C* vertices will be compressed, then the second *C* vertices, and so on. There is a trade-off inherent in the size of the subset. The fewer vertices we define as a subset, the finer the addressing granularity is. Also, as a Golomb-Rice encoder encodes difference values, it shares some parameters for the whole compressed buffer. The smaller *C* is, the better values shared among the subset will fit the data, leading to a smaller representation. However, as *C* shrinks, these values will be replicated more often and may not be different enough to warrant a unique value. We experimented with different subset sizes and found that a size

of 64 gives good results for all our data sets.

If random access were not a requirement, such as in streaming general-purpose computations, several simplifications would become available. Most importantly, compressed blocks would not be constrained by compression buckets; a continuum of compression rates could be supported rather than just a subset. With this relaxation, the stored data could be compacted, saving space in memory as well as bandwidth. Therefore, there would also be no need for the buffer map, since reading and writing data would happen sequentially from a start address. If a buffer can be declared as read-only, such as most geometry buffers, we could also compact the stored data, since there would be no chance of having to write more data than could fit into the alloted space.

5 Proposed Techniques

Here, we discuss two proposed techniques for increasing the efficiency of any given hardware compression scheme: an algorithm for the dynamic selection of compression buckets for buffer maps and an encoder that can be more efficient for encoding residual values than the unary encoding used in standard Golomb-Rice compression. These novel techniques target two different areas that play a major role in determining the amount of data transmitted: the assignment of an overall compression ratio (which we call a compression "bucket") and the encoding of residual values. These techniques can be used independently if one or the other fits a particular compression scheme.

5.1 Bucket Selection

In past work, buckets have been chosen by the hardware designer and set statically in the hardware itself. Typical bucket values are "Fast Clear (FC), 25%, 50%, and Uncompressed" [Ström et al. 2008]. However, this poses two problems. First, the buckets that best capture one buffer may not serve another buffer as well. Second, in our work with variable-precision data, the buckets that best fit a particular buffer at a high precision may limit the savings possible when the precision is reduced. There are two simple approaches one might take. First, seeing that these buckets are too optimistic for some buffers (see Section 8.1), one could choose higher buckets, such as "FC/50%/75%." However, this will limit the compression rates achievable by highly compressible data sets. Another straightforward approach is to increase the number of buckets, say from four to eight. However, this would increase the storage needed by the buffer map. (In past work, this was called the "tile map;" we feel that "buffer map" better describes its use in a general compressor, which may or may not be tile-based.) The buffer map's presence in an on-chip cache is very important, as every access to memory depends on its contents.

We seek to assign buckets dynamically for every unique buffer, be it a render buffer, depth map, final frame buffer, or input geometry buffer. Each buffer will store its currently selected buckets in its descriptor. We constrain our algorithm in four ways. First, we do not allow more than four buckets per buffer, which keeps the size of the buffer map the same as in past work - 2 bits per buffer. Second, by necessity, the "uncompressed" bucket is non-negotiable; we must assume that there will be input data that will not be able to be compressed at all. This leaves three available buckets that can be chosen dynamically. Third, we are not allowed a pre-pass to examine the buffer; it must be compressed on-the-fly. Lastly, we allow a bucket granularity of eighths. While having more bucket options with a smaller size *may* perform better, it is unreasonable to expect finer granularities when reading from memory. Our $1/8^{th}$ granularity buckets for 32-bit data are the same size as the $1/4^{th}$ buckets used for 16-bit data in past research.

Our dynamic bucket selection algorithm is a three step process. First, the smallest bucket (again, in one eighth increments) that will fit the output data is chosen. If this bucket is already in use, then the algorithm is complete and that bucket is used in the buffer map. If this bucket it not in use and there is still an "open" bucket, this bucket is set to be the chosen bucket and is written to the buffer map. In the worst case, the smallest bucket can not be used, and the next largest bucket must be used from the already-chosen list.

5.2 Fibonacci Encoder

We see larger residuals (compared to the size of the input value) than seen in past work [Ström et al. 2008; Rasmusson et al. 2007]. There are three reasons for this: negative values, frequent unclamped values, and a larger mantissa to total representation ratio. Allowing negative values can cause differing sign bits, leading to differences with maximum magnitudes in common cases. Further, since general data is not expected to be commonly in the range of [0.0..1.0], like color, normal, and many depth formats, differing exponents for neighboring data values will lead to larger residuals, even in same-sign values. Lastly, 32-bit floating-point numbers will be left with relatively more information after subtraction than 16-bit floating-point numbers, as used in past work. This is due to the ratio of mantissa bits to overall bits in the two representations - 23:32 (˜2:3) for 32-bit data, and 10:16 (5:8) for 16-bit data. Taking all this into account, we explore using an encoder other than unary encoding to store quotient values generated by a standard Golomb-Rice encoder. We detail the behavior and expected benefits of a Fibonacci encoder [Fraenkel and Klein 1996] in the auxiliary material.

5.3 Hardware Implementation

Our proposals are able to be implemented in hardware without any major changes in the architecture. Fibonacci coding, while nontrivial, is not particularly difficult. The necessary Fibonacci numbers could be stored in a look-up table; to encode a maximum difference of 33 bits, less than 50 Fibonacci numbers are needed. To encode a value, the algorithm simply marches backwards through the numbers, logging whether or not a number is in the sum and subtracting any included Fibonacci numbers from the encoded value.

Dynamic bucket selection is able also to be implemented in hardware. Each buffer already carries with it a descriptor of some type. For color buffers, this holds component and data formats; geometry data has a data layout header. This is where the chosen buckets for this buffer could be located at a cost of 12 extra bits (to encode 10 possible states for three different buckets). Decompression of data is only slightly more complicated. Since the descriptor information must be available to read and write the correct data in any case, the chosen buckets will also be available before consulting the buffer map. The combination of the chosen buckets and the buffer map will allow the memory controller to request only as many lines as necessary. Compressing data requires only a slight modification. Rather than just assigning one of three buckets based on the input and output size, the buffer is first assigned to one of eight preliminary buckets; this is no more complicated than before. After this, the chosen buckets for the buffer are consulted and possibly updated. Since, as when reading, the chosen buckets and buffer map are available in local memory, accessing these lists is fast and cheap.

6 Compressing Reduced-Precision Data

In past work, variable-precision arithmetic on the GPU [Pool et al. 2008; Pool et al. 2011a; Pool et al. 2011b] has been performed by only using the most significant p bits of the mantissa in computa-

tions. Since this leaves 23-p bits unused, it is unnecessary to move these bits on- and off-chip. Taking advantage of this, we modify the range of the values input to the compressor. As values are input to the compressor, a standard step is to reinterpret the floating-point values as integers. By dropping bits on the right that have been ignored by variable-precision arithmetic, we can lessen the magnitude of the values, and therefore the magnitude of the difference between them, which determines the size of the encoded stream.

The precision of the data undergoing compression or decompression is constant per buffer and assumed to be stored in the data descriptor. For color buffers, this commonly contains information such as width, height, number of components, and compression used (in hardware which supports multiple compression schemes). For input geometry, this would be the vertex buffer declaration, which holds information about each stream, such as data type, number of components, and, in some cases, intended use. By storing the buffer's precision with this standard collection of data, we do not have to store it with the buffer data itself, avoiding overheads. It would be possible to store the precision of each compressed buffer and opportunistically perform variable-precision compression for data that happens to have a number of trailing zeros in the same way that we take advantage of leading zeros. However, we found that trailing zeros in full-precision data are not common enough to overcome the overheads of storing precision with the buffer data.

This approach is different from past techniques in several ways. First, though Rasmusson et al. [2009] describe a similar system of quantization, theirs is intended to perform lossy compression. Therefore, they also monitor errors at runtime and scale back quantization when it could lead to larger errors. Our approach is lossless, though it operates on data that has been quantized. The difference is that the quantization step for variable-precision data has been performed by the artist or programmer [Pool et al. 2011b] and has been judged acceptable; we do not risk incurring further errors. Further, Rasmusson et al. [2009] performed quantization on the residuals of the predicted values. Over the span of a tile, this leads to errors as values are reconstructed erroneously from previous values. Since we quantize the input values themselves, errors do not compound; the input value can be reconstructed exactly.

7 Experimental Setup

We seek to report compression rates seen by real applications during use. Therefore, we do not look merely at the final color or depth data for a frame; rather, we treat each intermediate draw stage when possible for a realistic estimate of saved bandwidth. To do this, we log intermediate buffers from running applications as vectors of floating-point values. We can replay these buffers in a custom simulator and infer which tiles were changed as the result of any given step. These tiles are then re-compressed with the new data, and the bits spent in transferring old and new tile data is counted towards a running total for the experiment. Similarly, we examine the actual vertex data fetched during execution and count only it. During simulation of a buffer, dynamic bucket selection is coherent; that is, the buckets chosen during the first pass on the buffer do not change for successive passes.

We do not explicitly model a cache hierarchy. While such a model would change our absolute results for amount of data transferred, it will not impact our compression rates. Further, any model we devised would be an approximation and lacking the context of a full graphics system; such a full model is out of the scope of this work. Thus, our work should be viewed as presenting novel compression techniques for general data with resulting compression rates, not as a prediction of ground-truth bandwidth savings.

Several implementation details must be noted for adapting the existing technique [Ström et al. 2008] to handle 32-bit variable-precision data. First, the constant error threshold used to select guide bits needs to be much larger; we found 2^{28} to work well for all data sets. Second, to support variable-precision values, this error threshold must be updated when the range of input values changes. This is as simple as shifting this error value at the same time that the values themselves are shifted, and by the same amount.

(a) HDR (b) Parallax

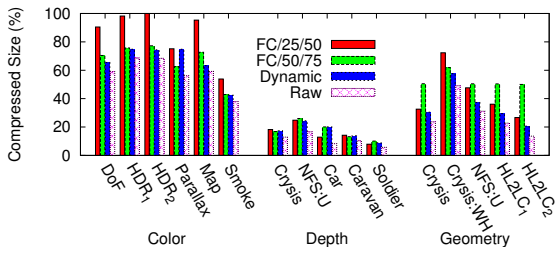
(c) Crysis (d) NFS:U

Figure 2: *Test applications, from which we used color, depth, and geometry buffers.*

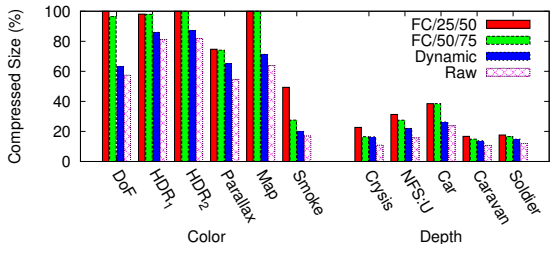
7.1 Data Sets

To test our algorithms, we have used different buffers from several applications; representative applications can be seen in Figure 2. (Refer to Appendix B in the ACM Digital Library for a complete listing of data sets.) For our test color buffers, we used scenes from a high dynamic range rendering demo, a depth of field demo [AMD 2008], a smoke simulation visualization, a parallax mapping demo [NVIDIA Corporation 2010], and a demo that generates terrain data on the GPU to use for rendering ("Map") [Persson 2006]. For testing depth buffer compression, we used depth maps generated and manipulated by an application which demonstrates different shadow mapping techniques [Lauritzen 2007], as well as from the video games "Need for Speed: Undercover" [EA Black Box 2008] ("NFS:U") and "Crysis" [Crytek 2007]. We also used input geometry from these two games to test our compression techniques on vertex positions and attributes, as well as geometry from "Crysis: Warhead" [Crytek Budapest 2008] and several scenes of "Half-Life 2: Lost Coast" [Valve 2005].

It is worth noting that the "Depth of Field" and "Map" examples encode extra information in the alpha channel of the RGBA render target. "Depth of Field" encodes the depth of the scene to use in further processing, while "Map" uses the RGB channels to encode normals of procedurally-generated terrain and the alpha channel to encode this terrain's height. These data sets are not compressible by past work without modification. However, our unified system allows for compression of these nonstandard uses. In the interest of having more data sets, we will disregard the fourth channel in these two data sets when presenting results of the existing technique modified with our proposed techniques (dynamic bucket selection, Fi-



(a) Unified



(b) Tiled

Figure 3: *Our dynamic bucket selection algorithm's performance on our proposed ("Unified") and state-of-the-art ("Tiled") compressors. In general, dynamic selection outperforms static selection by an average of 1.2x for our unified compressor and 1.3x for the tiled compressor. Ideal performance is seen in the "Raw" column.*

bonacci encoding). When comparing our unified compressor with the past approach, however, we do include the fourth channel to show the benefit of a general compression scheme.

8 Results and Discussion

In this section, we present the compression rates achieved by the state-of-the-art lossless color and depth buffer compressor [Rasmusson et al. 2007] as well as our general-purpose compressor. We examine the impact of our three proposed techniques on these two compressors: dynamic bucket selection (8.1), Fibonacci encoding (8.2), and our variable-precision data compression (8.4). To limit the complexity of our findings, we will present each new section having implemented the previous sections' proposals. Fibonacci encoding is compared to unary encoding with dynamic bucket selection enabled, and so on.

8.1 Dynamic Bucket Selection

We first examine our dynamic bucket selection algorithm and results, shown in Figure 3 for our general purpose ("Unified") and modified existing ("Tiled") approaches. The first three columns of each data set show buckets of "FC/25%/50%," as in past work [Ström et al. 2008], "FC/50%/75%," a simple approach to correcting for optimistic buckets, and dynamic bucket selection. The final column of each data set ("Raw") shows the compression achieved if buckets were not imposed; the data could no longer be accessed randomly, but higher compression could be achieved. We view this as a goal, though it is unachievable in all but contrived cases in which compression rates align perfectly with bucket values.

Dynamic bucket selection outperformed static bucket selections in 22 out of 27 test cases. For our general purpose compressor, dynamic bucket selection was generally beneficial. It outperformed the two static bucket selections in 11 out of 16 cases, and by an average of 1.26x for color and geometry data. It was roughly comparable for depth data, which was already significantly compressed. Dynamic bucket selection outperforms the unmodified tile compressor in each case (by an average of 1.34x). Some test sets were simply not compressible by the "FC/25/50" or "FC/50/75" static bucket choices. The finer granularity coupled with the data-dependence of our algorithm leads to better-fitting buckets and better compression rates by an average of 1.25x over all tests.

8.2 Fibonacci Encoding

While one test case showed a 1.7x improvement over Golomb-Rice encoding, the Fibonacci encoder was not a clear improvement, showing an average improvement of 1.06x. Overall, it is impossible to say that one encoder is unequivocally better or worse than the other. Detailed results are provided in the auxiliary material.

8.3 General-Purpose Data Compression

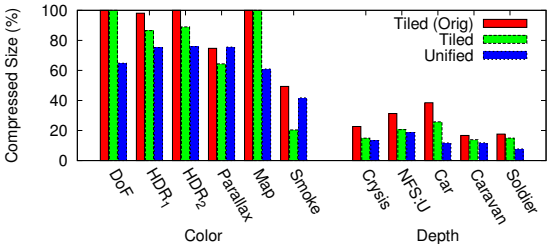


Figure 4: *Performance of an existing compressor, that compressor augmented with dynamic bucket selection and Fibonacci encoding, and our proposed compressor with proposed enhancements. Our unified compressor outperformed the baseline and enhanced baseline compressors by averages of 1.4x and 1.2x, respectively.*

Figure 4 shows how our proposed unified compressor, with dynamic bucket selection and Fibonacci encoding, compares with the original and similarly-modified version of the state-of-the-art tile-based color and depth compressor [Ström et al. 2008]. We have omitted geometric data sets, as their compression in on-chip hardware is novel to our work.

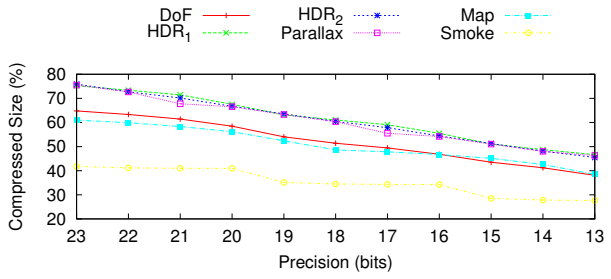
We see that in many cases, our proposed general-purpose compressor achieves better bandwidth savings. Clearly, this should be investigated further, but we propose a possible explanation: incoherent data and uncorrelated channels. The benefit of the specialized tile compressor stems from its guide bits, restart values, and rotation bit. These extra flags are intended to exploit 2D coherence in a single channel of data, say the red channel of a color buffer. When no such coherence exists, then these bits are overhead with no benefit. This tile-based approach also assumes that color channels are correlated, which, as noted in follow-up work [Wennersten and Ström 2009], is not necessarily the case. Our general-purpose compressor is able to exploit much of the same coherence without the overheads of unnecessary guide bits. There are times when these extra bits and channel correlation can make a difference, though, such as in color buffers with smoothly-changing or blocky colors ("smoke" and "parallax"). Our proposed general-purpose compressor outperforms the state-of-the-art for 16-bit floating point data adapted for 32-bit data and is also able to handle many more types of data.

8.4 Variable-Precision Compression

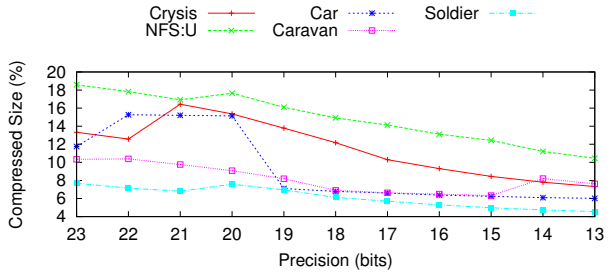
As the precision of the input data is reduced, we can likewise reduce the range of that data, leading to smaller residuals. The ef-

fect of this range reduction on compression performance of color, depth, and geometry is shown in Figure 5. In general, we see a very promising trend: reducing the precision of the input data allows for significantly better compression rates. One minor departure from this trend occurs in some of the depth buffers. There is a discontinuity where the compressed size increases as precision decreases; this is an artifact of the dynamic bucketing. As the precision drops, new buckets are chosen, which do not fit *all* of the data well until precision drops further. However, this behavior still allows for savings and is better than having static buckets. Figure 6 shows the necessity of using dynamic bucketing with range reduction.

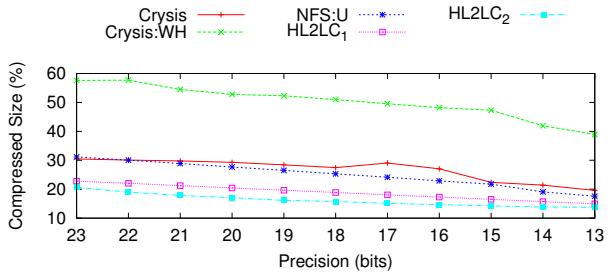
The compression rate gains expected by the range reduction of variable-precision data is hard to predict without knowing the behavior of a particular application. Past work has shown that the precision in some applications can be reduced by up to 12 bits with no noticeable errors [Pool et al. 2011a; Pool et al. 2011b]. This type of reduction could lead to significant extra bandwidth savings.



(a) Color



(b) Depth



(c) Geometry

Figure 5: *Compression rates achieved by our general-purpose compressor on color, depth, and geometry data as the precision of the data is reduced.*

9 Conclusion

We have designed a general-purpose compression and decompression scheme for 32-bit floating-point data on graphics hardware. It both outperforms an existing 16-bit compressor [Ström et al. 2008] adapted to handle 32-bit data *and* is able to compress general data. We have shown this capability by presenting promising compression rates for geometry data (vertex positions, normals, texture coordinates, etc.) for real-world applications. Average rates for color, depth, and geometry data are 1.5x, 7.9x, and 2.9x, respectively.

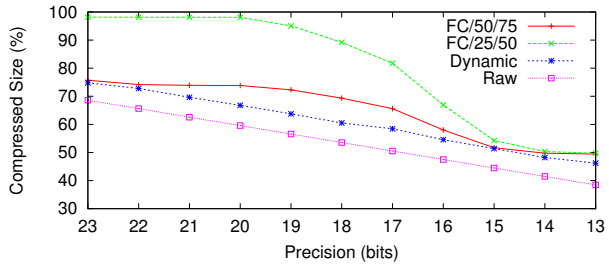


Figure 6: *Range reduction of variable-precision data is much more effective when used with dynamic bucket selection (HDR_1 scene).*

Furthermore, we have proposed two novel techniques applicable to any hardware compression scheme: dynamic bucket selection and the use of a Fibonacci encoder. These proposals increased compression ratios by averages of 1.25x and 1.06x, with maximum improvements of 2.4x and 1.7x, respectively. Note that these are not just compression rates, this also takes quantized storage into account. So, these results should not be viewed as a single tile seeing an improvement of 1.25x (for example) but as several tiles remaining unchanged, and several others improving by 2x. We believe that these techniques are suitable for a hardware implementation and discussed our justification. Lastly, we have shown that extra savings are available by using range reduction on variable-precision data. The additional savings will depend on the specific application but are expected to be between 5% and 20%, for overall color, depth, and geometry compression rates of 1.9x, 10.7x, and 3.6x, respectively.

9.1 Future Work

Our dynamic bucket selection algorithm has been shown to work well in practice. However, its performance *is* dependent on the order in which the data is seen; a flipped or rotated buffer could drastically change the results. Extensions to this algorithm may be possible, such as delaying the selection of a bucket until some minimum number of chunks fall into it. Another approach may be to dynamically re-select buckets with the realization that moving a selected bucket from a smaller value to a higher value does not pose any functional problems; any buffers mapped to the selected bucket will still fit in its new value.

Compression of geometry buffers is often able to be asymmetric; many game applications have geometry that is authored once and read many times. Thus, it is reasonable to expect that a two-pass algorithm could be used on the data after authoring in order to choose the best buckets for a particular buffer. Our dynamic selection decompression scheme would still be necessary to make use of these buffers, as used buckets are still a subset of the available buckets.

Acknowledgements

Financial support was provided by the National Science Foundation under grant CCF-0702712.

References

AMD, 2008. Rendermonkey 1.82. http://developer.amd.com/archive/gpu/rendermonkey/pages/default.aspx, Dec.

BURTSCHER, M., AND RATANAWORABHAN, P. 2009. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput. 58* (January), 18–31.

CRYTEK BUDAPEST, 2008. Crysis Warhead. http://www.ea.com/crysis-warhead, September.

CRYTEK, 2007. Crysis. http://www.ea.com/crysis-1, November.

DANILAK, R. 2009. Efficient multi-chip GPU. *US Patent US 7,616,206 B1*.

DEERING, M. 1995. Geometry compression. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '95, 13–20.

DMITRIEV, K., AND MORETON, H. P. 2011. Method for watertight evaluation of an approximate catmull-clark surface. *US Patent US2011/0085736 A1*.

EA BLACK BOX, 2008. Need for Speed: Undercover. http://undercover.needforspeed.com/home.action, Nov.

FRAENKEL, A. S., AND KLEIN, S. T. 1996. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics 64*, 31–55.

GOEL, V., AND MARTIN, T. 2009. Merged shader for primitive amplification. *US Patent US2009/0295804 A1*.

GRUETZMACHER, G. P. 2010. Method and apparatus for model compression. *US Patent Application 2010/0008593 A1*.

HAO, X., AND VARSHNEY, A. 2001. Variable-precision rendering. In *2001 Symposium on Interactive 3D Graphics*, ACM, New York, NY, USA, I3D '01, 149–158.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2006. Efficient depth buffer compression. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ACM, New York, NY, USA, 103–110.

HIDETOSHI, AND YOKOO. 1994. Adaptive encoding for numerical data compression. *Information Processing & Management 30*, 6, 863 – 873.

IOURCHA, K. I., NAYAK, K. S., AND HONG, Z. 1999. System and method for fixed-rate block-based image compression with inferred pixel values. *US Patent 5956431*.

ISENBURG, M., LINDSTROM, P., AND SNOEYINK, J. 2005. Lossless compression of predicted floating-point geometry. *Comput. Aided Des. 37* (July), 869–877.

KECKLER, S., DALLY, W., KHAILANY, B., GARLAND, M., AND GLASCO, D. 2011. Gpus and the future of parallel computing. *Micro, IEEE 31*, 5 (sept.-oct.), 7 –17.

LAURITZEN, A., 2007. Variance shadow maps demo (D3D10). http://www.punkuser.net/savsm/variance_shadow_map_d3d10_2007-04-26.zip, April.

LINDSTROM, P., AND ISENBURG, M. 2006. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics 12* (September), 1245–1250.

MICROSOFT, 2011. Texture block compression in Direct3D 11. http://msdn.microsoft.com/en-us/library/windows/desktop/hh308955(v=vs.85).aspx, September.

NVIDIA CORPORATION, 2010. NVIDIA Direct3D SDK 10 Code Samples. http://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html, Jun.

PERSSON, E., 2006. Infinite terrain II. http://www.humus.name/index.php?page=3D&ID=65, March.

POOL, J., LASTRA, A., AND SINGH, M. 2008. Energy-precision tradeoffs in mobile graphics processing units. In *2008 IEEE International Conference on Computer Design*, ICCD '08, 60 –67.

POOL, J., LASTRA, A., AND SINGH, M. 2011. Power-gated arithmetic circuits for energy-precision tradeoffs in mobile graphics processing units. *J. Low Power Electronics 7*, 2, 148–162.

POOL, J., LASTRA, A., AND SINGH, M. 2011. Precision selection for energy-efficient pixel shaders. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, NY, USA, HPG '11, 159–168.

RAMEY, W. O., MORETON, H. P., AND ROGERS, D. H. 2008. Decompression of vertex data using a geometry shader. *US Patent US 2008/0266287 A1*.

RASMUSSON, J., HASSELGREN, J., AND AKENINE-MÖLLER, T. 2007. Exact and error-bounded approximate color buffer compression and decompression. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 41–48.

RASMUSSON, J., STRÖM, J., AND AKENINE-MÖLLER, T. 2009. Error-bounded lossy compression of floating-point color buffers using quadtree decomposition. *Vis. Comput. 26* (November), 17–30.

RATANAWORABHAN, P., KE, J., AND BURTSCHER, M. 2006. Fast lossless compression of scientific floating-point data. In *Proceedings of the Data Compression Conference*, IEEE Computer Society, Washington, DC, USA, 133–142.

STRÖM, J., WENNERSTEN, P., RASMUSSON, J., HASSELGREN, J., MUNKBERG, J., CLARBERG, P., AND AKENINE-MÖLLER, T. 2008. Floating-point buffer compression in a unified codec architecture. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, GH '08, 75–84.

VALVE, 2005. Half-Life 2: Lost Coast. http://store.steampowered.com/app/340, Oct.

WENNERSTEN, P., AND STRÖM, J. 2009. Table-based alpha compression. *Computer Graphics Forum 28*, 2, 687–695.

WITTENBRINK, C., AND ORDENTLICH, E. 2005. Sort middle, screen space, graphics geometry compression through redundancy elimination. *US Patent 6961469 B2*.

YIFEI, J., AND DANDAN, H. 2010. Improved texture compression for S3TC. In *Picture Coding Symposium (PCS), 2010*, 386 –389.