Journal of Computer Graphics Techniques Vol. 1, No. 1
http://jcgt.org 2012

A Compressed Depth Cache

Jon Hasselgren, Magnus Andersson,
Jim Nilsson, and Tomas Akenine-Möller

Intel Corporation, Lund University

Abstract

We propose a depth cache that keeps the depth data in compressed format, when possible. Compared to previous work, this requires a more flexible cache implementation, where a tile may occupy a variable number of cache lines depending on whether it can be compressed or not. The advantage of this is that the effective cache size increases proportionally to the compression ratio. We show that the depth-buffer bandwidth can be reduced, on average, by 17%, compared to a system compressing the data after the cache. Alternatively, and perhaps more interestingly, we show that pre-cache compression in all cases increases the effective cache size by a factor of two or more, compared to a post-cache compressor, at equal or higher performance.

1. Introduction

Reducing memory-bandwidth usage in graphics processors is becoming increasingly important, both from a performance perspective and also from a power-efficiency perspective. The data traffic to and from the depth buffer consumes a significant amount of bandwidth, and it is therefore important to reduce this traffic as much as possible. Common approaches include Z_{max}-culling [Greene et al. 1993], Z_{min}-culling [Akenine-Möller and Ström 2003], depth caching, and depth compression [Morein 2000; Hasselgren and Akenine-Möller 2006].

We approach this problem by looking at the interplay between the depth cache and depth compression and propose a system where the content in the depth cache is kept compressed when possible. The implication of this approach is that tiles (rectangular regions of samples/pixels) that can be compressed in the cache will utilize less storage there, and, hence, the effective cache size is increased with better performance as a result. Alternatively, the cache size can be reduced with unaffected cache performance. By using a compressed level-one depth cache, we show that our system can reduce the depth-buffer bandwidth, on average, by 17%; this suggests that it can be potentially important to further study compressed cache architectures for graphics.

We suspect systems similar to ours have already been implemented, or at least considered, by graphics hardware vendors. However, we have not found any previously published work on such a system, and, as such, this paper aims to fill that gap by describing the implementation alternatives and evaluating the expected performance.

2. Previous Work

The amount of publicly available work on of depth compression is relatively sparse. Morein [2000] presented a depth-buffer compression system, which included a depth cache, using differential differential pulse code modulation (DDPCM) for compression. It is important to note that depths are required to be lossless by contemporary graphics APIs, and therefore, there is always a fallback that represents uncompressed depth data in a tile.

Hasselgren and Akenine-Möller [2006] used patent disclosures to survey of depth-compression techniques in industry, many of which do not otherwise appear in the peer-reviewed scientific literature. In addition, they presented a twist of an existing compression algorithm that improved compression a bit. In their survey, a method called *depth-offset* compression was presented, and it is likely the most simple depth-compression algorithm available. The idea is to find the minimum, Z_{min}, and the maximum, Z_{max}, of the depths in a tile and to cluster the depth values into two groups, namely, one for the depths closest to Z_{min} and another for the depths closest to Z_{max}. The depths are then encoded relative to either Z_{min} or Z_{max}, and, often, it is possible to use relatively few bits for these residuals.

Another interesting algorithm is *plane encoding* [Hasselgren and Akenine-Möller 2006], where the rasterizer provides exact plane equations to the compressor. As a result, only a bitmask is needed per sample/pixel to identify to which plane equation a sample/pixel belongs. Hence, the residuals will always be zero. *Anchor encoding* is a variant that uses a set of plane equations derived from the depths in the tile. The residuals are then encoded relative to one of these planes.

Lloyd et al. [2007] developed a logarithmic shadow-mapping algorithm, and realized that planar triangles become curved in their space, and that, therefore, most previous depth-compression algorithms could not be used. They computed first-order differentials and then use anchor encoding on the differentials. Ström et al. [2008] presented the first public algorithm for compressing floating-point depths. The depth values are reinterpreted as integers in order to represent differences without loss. They use a predictor function based on a small set of the depths in the tile and then apply Golomb-Rice entropy encoding on the residuals. Pool et al. [2012] present a general algorithm for floating-point data, which compresses the differences between a run of floating-point numbers and uses a Fibonacci encoder for entropy encoding. However, any algorithm involving serialized entropy encoding is, in general, too expensive for our purposes. Inada and McCool [2006] use a B-tree index to support random access for lossless texture compression with variable bitrate. However, their tile cache, which is closest to the shading pipeline, is still uncompressed.

Andersson et al. [2011] were the first to attack the problem of compressing depth buffers generated using stochastic motion-blur rasterization. By incorporating the time dimension, t, into the predictor functions, better predictions were possible. They also noted that most previous algorithms for depth compression break down because they exploit the fact that the depths of triangles are linear. This does not hold when triangles start to move. Interestingly, the depth-offset encoding method performed reasonably well even for motion blur.

Compression of cached data in CPUs has received some attention. In general, CPU-based compression targets integer workloads and, in particular, zero or near-zero values. Some techniques also try to detect repeating patterns. Lee et al. [1999] first described a dynamic approach to compressed cache contents. They introduced a cache architecture capable of simultaneously handling both compressed and uncompressed lines. The core idea was to avoid cache-set aliasing by including an extra bit in the index. By doing so, they avoid *data expansion*, resulting from previously compressed lines expanding to cover (in their case) two lines, and relax *fat writes*, which happen when two cache lines are written by the same memory operation.

Alameldeen and Wood [2004] present a CPU system with uncompressed first-level cache, while compressing second-level data when possible. They introduce frequent pattern compression, which is a method for detecting and compressing a number of predefined data patterns. A three-bit prefix, stored with cache-line tag data, designates one of eight possible compression encodings. Most modes are either covering lower-than-word resolution data types (five of eight), beside runs of zeroes or repeating byte values, and one mode designates uncompressed cache lines. For integer applications, the single most useful mode is *zero run*, accounting for about 85% of all compressible patterns. Compression ratios for the integer applications were in the range 1.4–2.4. The compression ratio for the floating-point applications was 1.0–1.3. It is clear that previous work on compressed CPU caches is not particularly applicable to depth compression. In particular, depth data rarely resembles the simplified patterns assumed by the compressed CPU cache approaches.

3. Compressed Depth Cache

An illustration of how our system compares to a common depth-cache system with compression [Hasselgren and Akenine-Möller 2006] is shown in Figure 1. In the common system, we use *post-cache codecs*, which means that we only keep full and uncompressed tiles in the first-level depth cache and place the compressor/decompressor (codec) between the cache and the next level in the memory hierarchy. The cache line size in the common system is therefore always equal to the tile size. Whenever a tile is evicted from the cache, we update the per-tile *header data*, a memory area separate from the depth buffer that flags the compression mode used for each tile. Typically, the Z_{min} and Z_{max} values are also stored in this area for hierarchical occlusion culling [Greene et al. 1993; Akenine-Möller and Ström 2003]. The advantage of this system is that it has very simple cache logic, since the cache line size will be equal to the size of a tile. However, a drawback is that the tile size must be picked so that the compressed tile is large enough to efficiently burst when writing to or reading from RAM (or the next level in the memory hierarchy). This typically means that an uncompressed tile may be unnecessarily large, leading to wasted memory transactions and increased bandwidth when compression fails.

We propose using a more flexible cache where the line size is decoupled from the tile size and simply reflects what is efficient for a memory transaction. Furthermore, in contrast to the common setup, we put the compression/decompression logic before the cache; we call this a *pre-cache codec*. The benefits of this system are twofold. First, we can store compressed tiles in the cache,

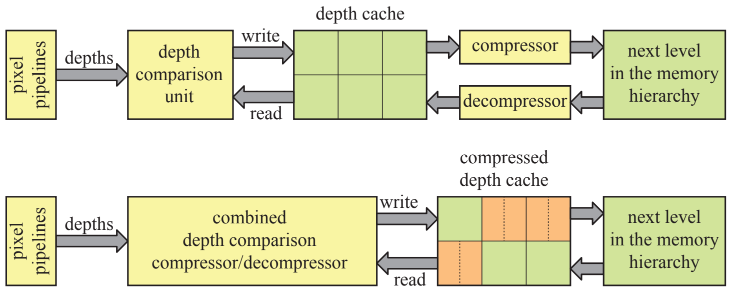


Figure 1. The usual setup in a compressed depth architecture (top). From left to right: the pixel pipelines compute depths, which are delivered to a depth comparison unit. This unit communicates with a depth cache that can hold six tiles of depth data in this illustration. When depth data is communicated between the depth cache and the next level in the memory hierarchy, it will be compressed/decompressed on the fly when possible. Our proposal (bottom) is to keep the content in the depth cache compressed when possible, and to efficiently perform the comparison, and compression/decompression between the pixel pipelines and the compressed depth cache.

thereby growing the effective cache size proportionally to the compression ratio. Second, the tiles that cannot be compressed may more easily be split into a number of cache lines, and we can update only the lines touched by a triangle. This pass-through compressor, which stores cache lines that cover smaller screen-space regions than the full tile, is called RAW in this paper. The challenge in this solution is that we complicate the logic involved in depth testing and updating of a tile. Furthermore, since our compression algorithm is now placed before the cache, it needs to have lower latency and higher throughput than if placed after the cache. However, at the same time, the required throughput between the depth comparison unit and the depth cache (see Figure 1) decreases as the data is compressed in the cache. For example, if we get an average compression rate of 50%, we could harvest the halved throughput, for example, by reducing the datapath width or reducing the clocking of the cache.

The algorithmic flow of our depth system is illustrated in Figure 2. The rasterizer generates *input tiles* of samples for the current triangle. When a tile is received, we first perform hierarchical depth culling to determine whether the tile can be trivially discarded or accepted. For trivially accepted tiles, we attempt to compress the input tile data and allocate the appropriate number of lines in the cache. If the depth-culling result is ambiguous, the tile header

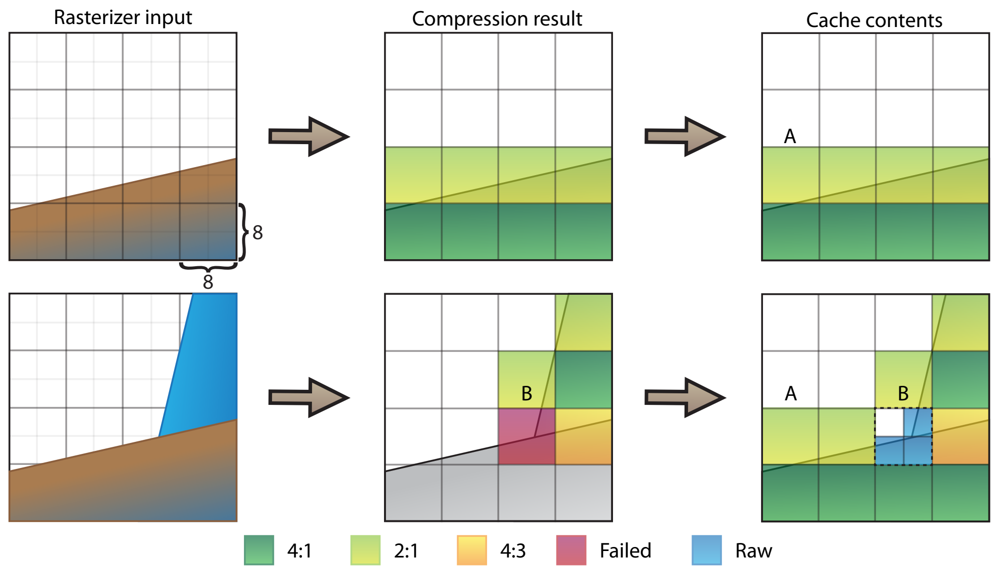


Figure 2. An example scene with two triangles being rasterized. Left: the input from the rasterizer (after z testing). Middle: the results and compression rates generated by the codec. Right: the final data stored in the cache. The tile size in this example is 8×8 pixels, equal in storage to four cache lines when not compressed. This indicates that this example system may compress to 25%, 50%, and 75%. For the first triangle, it is worth noting that tile A could also be stored as a single RAW cache line, which actually requires less data than the compressed representation (2:1 in this case). However, we keep the compressed representation since our system does not allow re-compressing tiles that have been reverted to uncompressed format. The assumption is that the compressed representation will be useful the next time we rasterize a triangle covering that tile. In this example, tile B completely fails compression and is stored in RAW format, but still only three cache lines are required.

data is first accessed to determine whether the frame-buffer depth data is compressed or not. For uncompressed tiles, the coverage mask of the input tile is used to read the appropriate lines into the cache, and then depth testing is done. In our implementation, we assume that tiles incrementally become more difficult to compress and, therefore, we do not attempt to re-compress tiles that already failed compression unless the current triangle overwrites the entire tile. This also greatly simplifies the implementation as recompression would have to consider cases when a tile only partially exists in the cache. For compressed tiles, we read the full compressed tile from the cache and decompress the data. After that, depth testing is done followed by an attempt to merge the resulting data into the compressed representation. If the merge fails, we first attempt a full recompression, and if that also fails, the data is stored uncompressed (RAW). We experimented with partially reading compressed tiles, only accessing the cache lines required to decompress the samples overlapped by the input tile coverage mask. However, most compression algorithms require global header data, and, in practice, the more complex implementation was not motivated by the very modest bandwidth gains.

In practice, the required changes to the depth system and cache logic are quite small. We need to compute cache keys on a per-line granularity, rather than a per-tile granularity, so the codecs should use actual memory addresses rather than a tile index. The biggest challenge occurs when a tile that only partially exists in the cache is evicted. Some operations, such as computing per-tile Z_{min} and Z_{max} values, require the full tile data. We solve this by performing hierarchical depth culling on a per-cache line granularity, thus guaranteeing that the cache line will always exist in the cache. Also, if we want to combine pre-cache and post-cache codecs in the same system, we must verify that the full tile exists in the cache in order to perform post-cache compression. We accomplish this by allowing peeking into the cache to check if the whole tile is present before evicting it. Since evictions are relatively infrequent, we believe this will be reasonably efficient. However, an alternative approach is to allocate one extra bit per cache line in the per-tile header data and directly flag which parts of the tile are present in the cache. This operation is very efficient, but at the cost of a slight bandwidth increase for the tile headers.

In this paper, we focus only on the plane-encoding and depth-offset compression algorithms. The reason is that they have simple implementations, and, therefore, we have been able to design efficient and incremental compression methods, which makes them good candidates for pre-cache codecs. Although we leave it for future work, we would like to mention that other traditional compression algorithms, such as anchor encoding [Hasselgren and Akenine-Möller 2006], could also potentially be adapted for pre-cache compression. In our pipeline, we use a clear mask per tile that indicates which samples are cleared, so the minimum, Z_{min}, and maximum, Z_{max}, depth values for a tile are computed using only valid samples.

3.1. Plane Encoding

In plane encoding, the representation for a tile is a list of plane equations that can reconstruct triangle depth exactly, and a per-sample bit mask that indicates to which plane a sample belongs. On-the-fly decompression from such a representation residing in the cache is straightforward. Assume we would like to decompress the depth of a certain sample/pixel location, (x_s, y_s). The bit mask value is used as an index, i, into the set of plane equations, and the plane equation is simply evaluated as $z = c_0^i + c_x^i \cdot x_s + c_y^i \cdot y_s$, where the constants c_0^i, c_x^i, and c_y^i together define plane equation i.

When a triangle is rasterized, the rasterizer forwards the plane equation to the pre-cache codec. Depth comparisons are done by decompressing depth values as described above. If at least one depth value passes the depth test, the incoming plane equation is added to the compressed representation in the cache, and the bit masks are updated for each affected sample/pixel. Note that the size of the compressed tile will dictate how many plane equations can be stored in a compressed tile; when there are no more available indices for new plane equations, the tile has to be decompressed and put into the cache again in uncompressed format.

There are different strategies for adding a new plane. In the simplest implementation, the planes are just added to the list of planes and compression fails when too many planes overlap a tile. However, better compression is possible by deleting unused planes from the header, either by scanning the index bitmask for unused bit combinations, or by keeping counters of how many samples belong to each plane. In such an implementation, the compressor must be able to work with one more plane than is representable by the compressed format.

3.2. Depth-Offset Compression Algorithm

Depth offset is a very simple compression algorithm, but it works surprisingly well. It does not enable high compression ratios, but it successfully manages to compress many tiles with moderate compression ratios. This makes it rather efficient overall. In addition, it is a simple algorithm from an implementation perspective. Recall that the compressed representation consists of two reference values, $Z_{\min}$ and $Z_{\max}$, a bit, m_{xy}, per sample that indicates whether a sample's residual is relative to $Z_{\min}$ or $Z_{\max}$, and then an n-bit per-sample residual, r_{xy}. The depth values are reconstructed as

$$z(x,y) = \begin{cases} Z_{\min} + r_{xy} & \text{if } m_{xy} = 0, \\ Z_{\max} - r_{xy}, & \text{otherwise.} \end{cases}$$

.

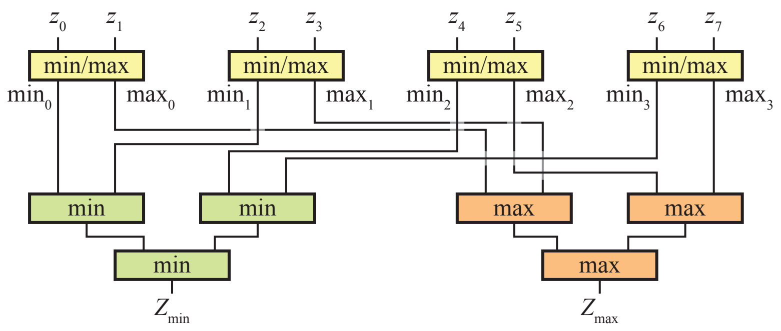


Figure 3. Computation of $Z_{\min}$ and $Z_{\max}$ using tree of comparisons for eight incoming depth values, z_i, $i \in \{0, \ldots, 7\}$.

It should be noted that the best bit distribution depends on the cache line size and the tile size. However, we find that it is often sufficient to quantize $Z_{\min}$ and $Z_{\max}$ to 16 bits precision and use the remaining bits for the residuals. For compression, there are more options, and, below, we present two different ways to compress the depth in a tile when a new triangle is being rasterized.

3.2.1. Brute-force Approach

In this approach, we first decompress all depth values in the tile, as described above, perform depth tests, and update the depths that pass. Then the $Z_{\min}$ and $Z_{\max}$ of these depths are found using, for example, a tree-like evaluation as shown in Figure 3. In general, for s depths, such a tree will use $s/2+2(s/2-1) = 3s/2-2$ comparisons to compute both $Z_{\min}$ and $Z_{\max}$.

The residuals, r_{xy}, and the selector bit, m_{xy}, are straightforward to compute. We just compute residuals from $Z_{\min}$ and $Z_{\max}$, respectively. If either residual is small enough to encode in the given budget, we set m_{xy} to flag the appropriate reference value. Otherwise, the tile fails compression and needs to be stored in uncompressed form.

In the next section, we present a conservative and less expensive approach to updating $Z_{\min}$ and $Z_{\max}$. The rest of the algorithm, however, remains the same.

3.2.2. Opportunistic Approach

We base this compressor on the assumption that the depth pipeline supports hierarchical $Z_{\min}$- and $Z_{\max}$-culling [Greene et al. 1993; Akenine-Möller and Ström 2003]. These algorithms require conservative estimates of the minimum $Z^{\text{tri}}_{\min}$, and the maximum depth, $Z^{\text{tri}}_{\max}$, of a triangle inside a tile. Regardless of exactly how they are computed, we can assume they are readily available since the hierarchical culling unit is placed before the depth compression unit in the pipeline.

We can exploit these estimates during compression by assuming that $Z_{\text{min}} = \min(Z_{\text{min}}, Z^{\text{tri}}_{\text{min}})$, and $Z_{\text{max}} = \max(Z_{\text{max}}, Z^{\text{tri}}_{\text{max}})$ are good estimates for the true minimum and maximum values of the tile. We then compute all residuals as in Section 3.2.1. As a small optimization, we use only the triangle values if the current triangle overwrites the entire tile.

In practice, this will, in the majority of cases, cause our depth range to grow until a tile can no longer be compressed. However, the implementation is more efficient since we can avoid the rather costly Z_{min} and Z_{max} computations. We suggest that this implementation be combined with a post-cache brute-force compressor. The simpler pre-cache codec will handle the high throughput data and keep it compressed in the cache for as long as possible. If the compression fails, the more expensive post-cache codec will refine the Z_{min} and Z_{max} values and re-compress the tile if possible. When the data is read back into the cache, the pre-cache codec can use the refined values as a starting point.

As a further optimization, we note that the residual computations can be done in two passes. First, the residuals are computed from Z_{min}, and in the following pass from Z_{max}. The second pass is conditional and can be skipped if all samples can be encoded relative to Z_{min}. Our tests indicate that it is sufficient with one reference value for 55% of the tiles, which may save substantial power in a hardware implementation.

4. Results

We evaluated our system using a functional simulator, written in C++, where it is possible to change cache settings, tile sizes, and configure the compression algorithms. Our simulator implements common depth-buffer optimizations, such as Z_{min}- and Z_{max}-culling, and fast clears [Hasselgren and Akenine-Möller 2006]. These optimizations are used for all of our measurements, even for the uncompressed reference bandwidth, so the bandwidth gains presented here come strictly from the compression algorithm and the cache system described in this paper. Also, we only present figures for the depth-buffer bandwidth, since the bandwidth for tile header data (Z_{min}, Z_{max}, and clear mask) is the same regardless of which type of cache (pre/post) is used.

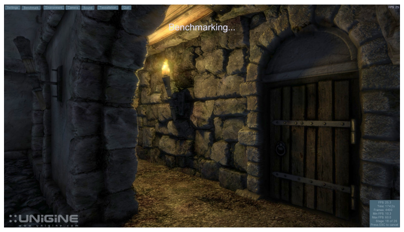
DirectX 11 supports 32-bit floating-point and 24/16-bit integer data. Of

Heaven A - 158K tris, 23 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	28.9M	47%	40%	35%	26%
32 kB	23.6M	47%	41%	34%	27%

Heaven B - 346K tris, 45 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	59.0M	51%	46%	34%	29%
32 kB	52.4M	51%	47%	34%	29%

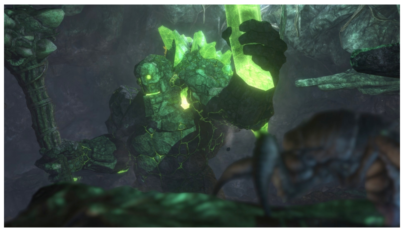


Heaven C - 283K tris, 25 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	38.9M	49%	42%	36%	29%
32 kB	32.9M	49%	45%	36%	32%

Stone Giant A - 447K tris, 23 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	36.9M	47%	41%	38%	32%
32 kB	31.9M	47%	42%	38%	34%



Stone Giant B - 218K tris, 34 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	44.5M	44%	40%	37%	34%
32 kB	40.9M	44%	39%	37%	33%

Dragon - 168K tris, 25 ppt.

Cache	RAW	Post DO	Pre DO	Post C	Pre C
16 kB	27.3M	47%	40%	35%	27%
32 kB	22.1M	48%	44%	34%	29%

Figure 4. The test scenes used in this paper were taken from the Heaven 2.0 benchmark by Unigine, the Stone giant demo by Bitsquid, and a Dragon scene created in-house. We show the number of triangles and average triangle area in pixels (ppt) for each test scene. The tables show bandwidth figures as a fraction of the RAW (no compression) bandwidth for post/pre-cache depth offset (Post/Pre DO) and post/pre-cache plane encoding combined with depth offset (Post/Pre C). We used 8×8 sample tiles with 512-bit cache lines (four lines per tile).

these formats, the 24-bit integer is still by far the dominating use case, and it is used by all our workloads. Looking at codecs for 32-bit floating-point depth data is interesting future work, but outside the scope of this paper. DirectX 11 only supports 24-bit integer depth when coupled with a stencil buffer; we therefore assume that most hardware vendors rely on their depth compression to reduce the bandwidth and store the full 32 bits (D24S8) when a tile fails compression, even if the stencil is unused. Alternatively, the stencil can be compressed along with the depth data, but we have also left this for future work. Thus, the bandwidth figures presented for the RAW algorithm include 32-bit reads and writes per sample.

For the evaluation, we used the scenes shown in Figure 4 rendered at 1920×1080 pixels resolution. We experimented with varying the tile size and bus parameters, but since the results were very similar, we only present numbers for a system using 8×8 sample tiles with 512-bit cache lines, which means that an uncompressed tile occupies four cache lines. We show the performance of two different configurations. The first (Post/Pre DO) uses only a depth-offset codec, which compresses the data to either one or two cache lines (25% or 50%), where we use 6 and 14 bits, respectively, for the residuals. The second configuration (Post/Pre C) combines both depth offset and plane encoding. Here, we found that using plane encoding, with up to four planes per tile for the 25% mode and depth offset for the 50% mode gave the best blend. This combination was just 1% from the bandwidth of using all possible combinations of plane encoding and depth-offset compression. It should be noted that plane encoding is not well-suited as a post-cache codec since it communicates directly with the rasterizer. In order to generate post-cache results for the plane encoder, we still performed the compression pre-cache, but reserved enough cache lines to keep the tile data uncompressed in the cache. An alternative would be to compare with a post-cache anchor codec, but we feel that the results are more representative when comparing the same codec post- and pre-cache.

As can be seen from the results (Figure 4), post-cache depth offset rarely manages to compress below 50%, but we still get a significant 11% relative bandwidth gain from using a compressed cache, which amounts to 5.5% of the total RAW bandwidth. For the second configuration, with plane encoding (25%) combined with depth offset (50%), the pre-cache approach is even more successful, and here we see a 17% relative bandwidth gain or 6.0% of the total RAW bandwidth. Figure 5 shows the per-tile bandwidth of the different compression schemes for the Heaven A test scene.

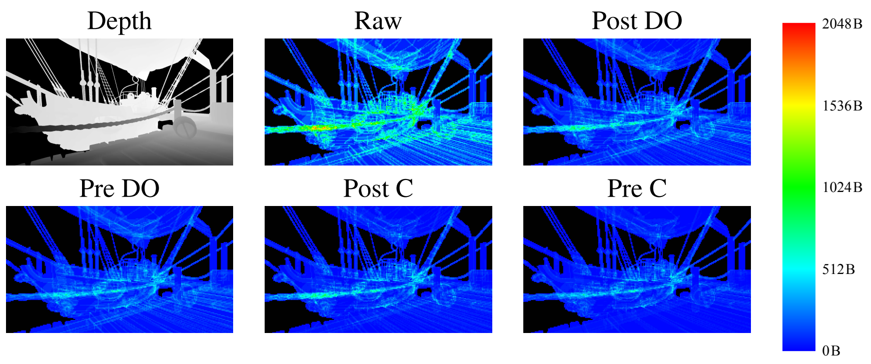


Figure 5. False color coding of the bandwidth for each 8×8 tile using the different compression schemes described in this paper. The images show Heaven A with a 32 kB cache. The black areas are cleared and the colored tiles have yielded memory transactions of at least 64B.

Cache size. As can be seen in Figure 6, increasing the size of the depth cache gives diminishing returns. Typically, the "knee" of the curve indicates the most efficient cache size in terms of performance versus implementation cost. An encouraging result is that the pre-cache codecs do not only outperform the post-cache codecs significantly for a given cache size, but also seem to push the knee of the bandwidth curve to a lower cache size. For example, the knee

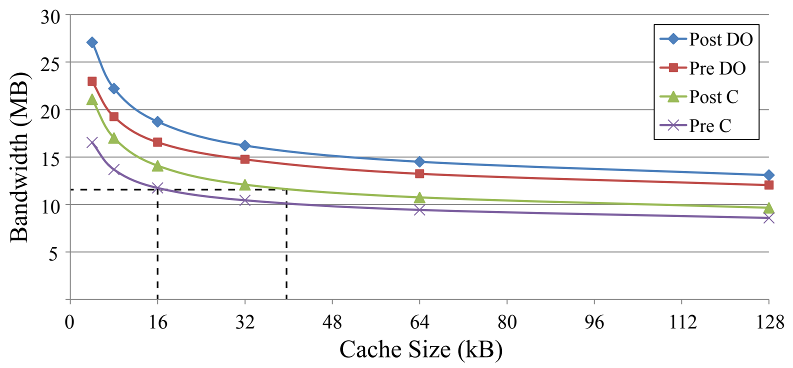


Figure 6. The average depth-buffer bandwidth for the six test scenes with varying cache size. The graph compares post- and pre-encoding with depth offset, and post- and pre-encoding with combined plane encoding and depth offset. The dashed lines show identical bandwidths for the post-cache and pre-cache codec for the combined compressor (Post/Pre C).

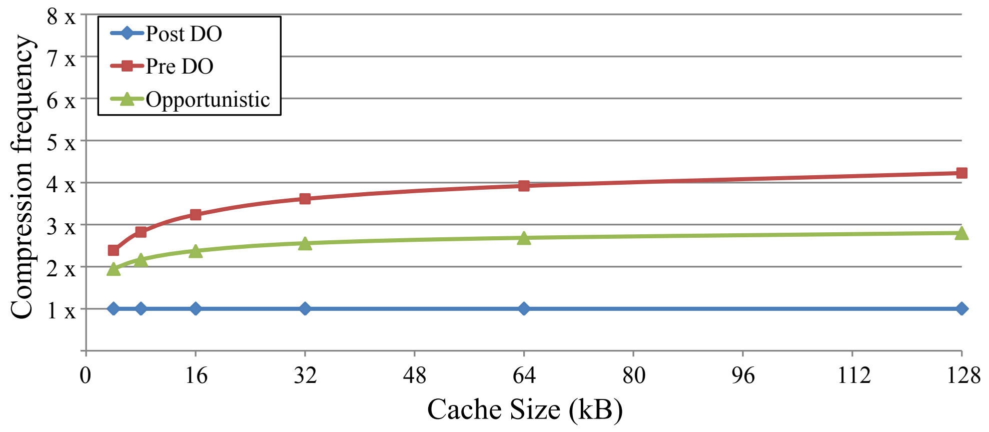
for the pre-cache combined codec seems to lie around 10 kB, whereas the knee for the post-cache lies at around 16 kB and with higher bandwidth usage. An alternative way of reading the diagram is to decide on a target bandwidth and design the cache around that. The dashed lines in Figure 6 show an example of this, where we can reduce the cache size to roughly 40% for the pre-cache codec, which is directly proportional to the compression ratio of about 40%.

Opportunistic depth offset. The impact of using the opportunistic depth-offset compression algorithm (Section 3.2.2) was also measured. We found that it results in a bandwidth increase of 4.2% compared to the brute-force approach, or 1.8% of the RAW bandwidth. However, we still see a worthwhile improvement over the post-cache codecs by 8.3%, or 3.7% of the RAW bandwidth. Depending on the pipeline architecture, it may be beneficial to consider the opportunistic approach if the cost of passing around $Z_{\mathrm{min/max}}$ values are considerably lower than recomputing them.

Recompression frequency. With pre-cache codecs, the number of times a tile is compressed and decompressed will increase as a function of the cache size. In Figure 7, we show how pre-cache compression affects the number of tiles the compressor and decompressor must be able to process per frame. Larger cache sizes means that a tile needs to be compressed and decompressed more times with the pre-cache codecs. This is due to the fact that the tiles stay in the cache longer, and, therefore, they will be accessed more times before being evicted.

We note that compression scales better than decompression, which is good since compression is usually the more costly operation. For our design points of a 16–32 kB cache, the opportunistic depth-offset approach only requires around 2.5× higher compression throughput and about 3.75× higher decompression throughput. This is very low considering that we use tessellated benchmark scenes which tend to use more and smaller triangles than most games.

With an increased focus on energy efficiency for graphics processors (see, for example, the work by Johnsson et al. [2012]), we argue that the tradeoff in our proposed system is very attractive for the following reasons. First, our compressors and decompressors are very simple, using only a number of integer math operations that is largely proportional to the number of samples in a tile. Second, a memory access to DRAM uses more than three orders of magnitude the power of an integer operation [Dally 2009]. Third, there



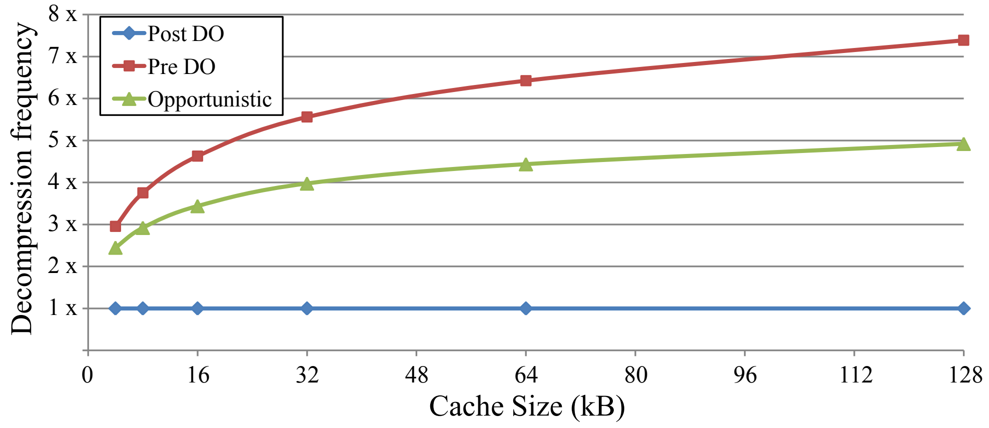


Figure 7. The average compression and decompression frequencies (i.e., the number of times a tile is compressed and decompressed) for the six test scenes with pre-cache, post-cache, and the opportunistic depth-offset approach. The figures are normalized so that the post-cache depth-offset frequency is always $1\times$.

are signs [Keckler et al. 2011] that memory bandwidth development slows down even more than what we are accustomed to. Hence, the motivation for a system with pre-cache codecs is clear and could be even more relevant in the future.

5. Conclusions and Future Work

We have shown that using a flexible depth cache may enable pre-cache data compression and that such compression will roughly increase the cache size by the effective compression ratio. This can either be used to reduce bandwidth to RAM (or to the next level in the memory hierarchy), or to reduce cache size and free up silicon area without affecting bandwidth. In our implementation, we have shown a significant 17% average relative bandwidth reduction for reasonable pipelines, when compared to a post-cache codec. Similarly, we have shown that the cache size can be reduced by the effective compression ratio with no impact on performance. In fact, for all our measurements, the effective cache size was more than doubled when going from a post-cache codec to a pre-cache codec. This is true for the depth-offset-only configuration, and to an even larger extent for the combined depth-offset and plane-encoding configuration.

For future work, we would like to explore other existing depth-compression algorithms and see how they perform in our system. Furthermore, it could be interesting to attempt to make the hardware implementations of complex codecs simpler (perhaps at the cost of reduced compression ratios). Also, since depth-offset compression works rather well for stochastic motion blur rasterization [Andersson et al. 2011], it will be interesting to see what happens to its performance in our system.

Acknowledgements

The authors thank Aaron Lefohn, Charles Lingle, Tom Piazza, and Aaron Coday at Intel for supporting this research. The benchmark scenes used in this paper are courtesy of Unigine and Bitsquid. Tomas Akenine-Möller is a Royal Academy of Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation.

References

Akenine-Möller, T., and Ström, J. 2003. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics, 22*, 3, 801–808. 101, 104, 109

Alameldeen, A. R., and Wood, D. A. 2004. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of the 31st Annual International International Symposium on Computer Architecture*, IEEE Computer Society, Washington, DC, 212–223. 103

Andersson, M., Hasselgren, J., and Akenine-Möller, T. 2011. Depth Buffer Compression for Stochastic Motion Blur Rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, ACM, New York, 127–134. 103, 116

DALLY, W. 2009. Power Efficient Supercomputing. Accelerator-based Computing and Manycore Workshop (presentation). 115

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-Buffer Visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1993*, ACM, New York, 231–238. 101, 104, 109

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, 103–110. 101, 102, 104, 107, 110

INADA, T., AND MCCOOL, M. D. 2006. Compressed Lossless Texture Representation and Caching. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, ACM, New York, 111–120. 103

JOHNSSON, B., GANESTAM, P., DOGGETT, M., AND AKENINE-MÖLLER, T. 2012. Power Efficiency for Software Algorithms Running on Graphics Processors. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, Eurographics Association, Aire-la-Ville, Switzerland, 67–75. 114

KECKLER, S. W., DALLY, W. J., KHAILANY, B., GARLAND, M., AND GLASCO, D. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro 31*, 5, 7–17. 115

LEE, J.-S., HONG, W.-K., AND KIM, S.-D. 1999. Design and Evaluation of a Selective Compressed Memory System. In *International Conference on Computer Design*, IEEE, Washington, DC, 184 –191. 103

LLOYD, D. B., GOVINDARAJU, N. K., MOLNAR, S. E., AND MANOCHA, D. 2007. Practical Logarithmic Rasterization for Low-Error Shadow Maps. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, Eurographics Association, Aire-la-Ville, Switzerland, 17–24. 102

MOREIN, S. 2000. ATI Radeon HyperZ Technology. In *ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, Hot3D Proceedings*, Eurographics Association, Aire-la-Ville, Switzerland. 101, 102

POOL, J., LASTRA, A., AND SINGH, M. 2012. Lossless Compression of Variable-Precision Floating-Point Buffers on GPUs. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, 47–54. 103

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, 96–101. 103

Author Contact Information

Jon Hasselgren
Intel
Schelevägen 19A
223 70 Lund, SWEDEN
jon.n.hasselgren@intel.com

Magnus Andersson
Intel
Schelevägen 19A
223 70 Lund, SWEDEN
magnusa@cs.lth.se

Jim Nilsson
Intel
Schelevägen 19A
223 70 Lund, SWEDEN
jim.k.nilsson@intel.com

Tomas Akenine-Möller
Intel
Schelevägen 19A
223 70 Lund, SWEDEN
tam@cs.lth.se

Jon Hasselgren, Magnus Andersson, Jim Nilsson, Tomas Akenine-Möller, A Compressed Depth Cache, *Journal of Computer Graphics Techniques (JCGT)*, 1(1):101–118, 2012
http://jcgt.org/published/0001/01/05/

Received:	September 9, 2012		
Recommended:	November 30, 2012	Corresponding Editor:	Michael Schwarz
Published:	December 31, 2012	Editor-in-Chief:	Morgan McGuire