Table 1. *Ablation on Clipping Level.* We test the variation in the total number of encoded points and the geometric compression ratio achieved by varying the clipping level of the Huffman Tree. The number of encoded points increases rapidly and tends to saturate around a clipping level of 11-12. At the clipping level of 12, the compression ratio decreases slightly due to the decoder table overhead. We choose the highest possible clipping level since it reduces the size of the separate buffer leading to less number of global memory accesses. Beyond 12, the shared memory allocation bottlenecks. The numbers are calculated and reported on the Morro Bay scene which consists of 350M points.

Clipping Level	Decoder Table Size	Encoded	Geometry CR
12	4096	98.34	3.52x
11	2048	97.31	3.67x
10	1024	95.23	3.56x
9	512	92.63	3.34x
8	256	87.00	2.86x

Algorithm 1 shows the pseudo-code for how the entire point cloud is encoded and prepared for the GPU to decompress and rasterize. This is done as a pre-processing step on the CPU and can take a few minutes.

Algorithm 1: Encoding the entire point cloud.
points: the entire set of points to encode
W: total number of warps in a batch

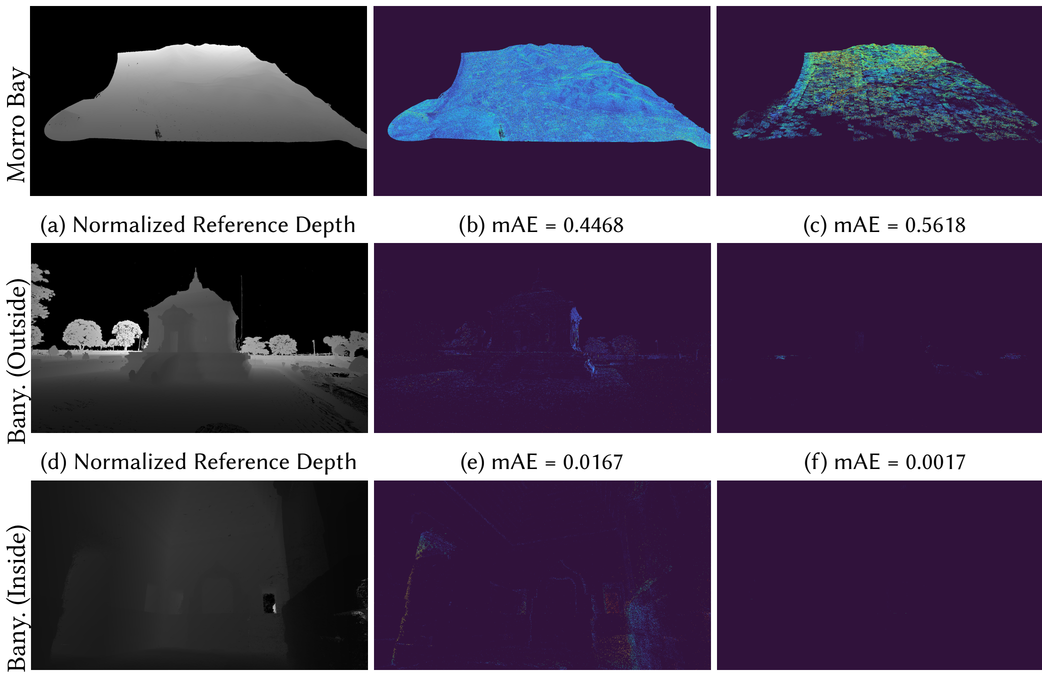
```
points ← mortonOrder(points)                   // Sort the points in morton order
for batch ∈ points do
    dX, dY, dZ ← getDeltaValues(batch)  // Calculate independent deltas in XYZ
    hTree, hTable ← getHuffman(dX, dY, dZ)     // Create Huffman Tree for deltas
    batchBuffer1, batchBuffer2 ← [], [] // Initialize the buffers for this batch
    for w ← 0 to W do
        warpPoints ← batch[w : w + 1]             // Points relevant to this warp
        warpBuffer1, warpBuffer2 ← [], []   // Initialize buffers for this warp
        for t ← 0 to 31 do
            threadBuffer1, threadBuffer2 ← huffmanEncode(warpPoints[t :
             t + 1], hTree)               // Encode the point data for the thread
            warpBuffer1 ← warpBuffer1 + threadBuffer1 // Append to warp Buffer
             1
            warpBuffer2 ← warpBuffer2 + threadBuffer2 // Append to warp Buffer
             2
        end
        warpBuffer1 ← sortAndRearrange(warpBuffer1)          // For this warp,
         re-arrange the first buffer
    end
    batchBuffer1 ← batchBuffer1 + warpBuffer1       // Append to batch Buffer 1
    batchBuffer2 ← batchBuffer2 + warpBuffer2       // Append to batch Buffer 2
end
```

Algorithm 2 shows the pseudo-code for the algorithm that every thread in a warp follows to read its respective data from the packed memory layout. For a thread to know where its data is from, it needs to book-keep the reads of every other thread. This is done very cheaply using `__ballot_sync` every decoding iteration.

Algorithm 2: A thread in a 32-thread Warp reading from re-ordered memory.
EncodedData: the re-packed encoded bit stream
L: maximum codeword length
tid: thread index

Input : *EncodedData*: the re-packed encoded bit stream
Input : *L*: maximum codeword length
Input : *tid*: thread index

```
ptr ← 0                                          // Data pointer
myMemBlock ← 0                                   // My memory block
remainingBits ← 0                                // Useful bits in my memory block
for i ← 0 to N do
    needToFetch ← (remainingBits < L)  // Need to fetch another memory block if out of bits
    warpMask ← __ballot_sync(0xFFFFFFFF, needToFetch)  // Communicate my fetch requirement to other threads
    if needToFetch then
        offset ← __popcount(warpMask × 2^(32−tid))  // Determine my fetch location using information from other threads
        myMemBlock.append(EncodedData[ptr + offset])  // Fetch next 4-byte memory block
        remainingBits ← remainingBits + 32  // Add 32 more useful bits due to 4-byte fetch
    end
    ptr ← ptr + __popcount(warpMask)  // Move data pointer by the number of threads that fetched
end
symbol, length ← decode(myMemBlock)  // Decode the next L bits using the decoder table
remainingBits ← remainingBits − length           // Only length bits used
```



(a) Normalized Reference Depth (b) mAE = 0.4468 (c) mAE = 0.5618

(d) Normalized Reference Depth (e) mAE = 0.0167 (f) mAE = 0.0017

(g) Normalized Reference Depth (h) mAE = 0.0063 (i) mAE = 0.0000026

Fig. 1. We calculate the Absolute Error on the depth maps (obtained from the first render pass) and report their averages. Due to selecting a subset of points for far-away batches, we get errors in the projected depths as shown. However, these errors are not significant. Col. 2 shows errors in the method by Schütz et al. [2022] who used quantized coordinates. Col. 3 shows errors due to our subset level-of-detail method.

REFERENCES

Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2022. Software Rasterization of 2 Billion Points in Real Time. *Proc. ACM Comput. Graph. Interact. Tech.* (2022).