EE 274 Project Final Report

1. Introduction

One of the most exciting frontiers in data compression is hardware acceleration. We investigate how hardware-level parallelism, specifically GPUs, can accelerate data compression. We look into the DietGPU repository from Facebook Research, which utilizes GPUs to accelerate the ANS compression and decompression processes. We mainly focus on the compression of floating-point numbers, which is a critical aspect of scientific computing and allows for ANS compression of floating-point exponents (which, in many applications, can be low-entropy). Additionally, since sparse data is a prevalent pattern in many applications, such as machine learning where an average of half of the data often consists of zeros after passing through a ReLU activation layer, we also incorporate a feature for compressing sparse data to further enhance overall system performance. We have extended the capabilities of DietGPU by implementing float64 compression and sparse float compression and decompression.

1.1. Overview: GPU Architecture

GPUs have thousands of small streaming multiprocessors, which allows GPUs to handle multiple tasks simultaneously. In contrast to CPUs, which have fewer but more powerful cores optimized for sequential processing, GPUs do better in executing a large number of lightweight tasks concurrently. Also GPUs are optimized for floating-point arithmetic, making them ideal for float data compression.

GPUs could do ANS encoding and decoding operations in parallel and handle multiple symbols simultaneously. Additionally, GPUs' high memory bandwidth ensure rapid access to pdfs and data, reducing data access latency. We can conclude that the architecture of modern GPUs makes them exceptionally well-suited for ANS encoding and decoding as well as float data compression. Their capacity for parallel execution and optimized hardware resources enables significant improvements in data compression and decompression performance.

There are three key concepts for GPU coding, warps, blocks, and grids. We will mention them in following paragraphs frequently because they are important for executing parallel computations efficiently.

1. **Warp**: Warp is the smallest unit of execution in a GPU, consisting of 32 threads. These threads can do the same instruction simultaneously.
2. **Block**: Threads within a block can communicate and synchronize with each other by sharing memory and data. Blocks are often used to partition the problem into smaller parts.
3. **Grid**: A grid is a higher-level grouping that contains multiple blocks. Blocks within a grid do not communicate directly with each other.

2. Code Review

We are adding to the existing DietGPU repository from Facebook Research, which includes GPU-accelerated versions of rANS. As an extension, they also applied the rANs implementation to compression of floating point exponents for Float16 and Float32. According to the README for the repository, the exponents of floating point numbers have much lower entropy than the significands, so it makes most sense to ANS compress the exponents, while leaving the significands uncompressed.

2.1. ANS Compression

DietGPU supports ANS compression of 1-byte symbols. Overall, ANS encoding on a GPU works as follows:

1. **Histogram**: the frequencies of each symbol are computed, with each thread taking a different subset of data.
    a. Each warp creates a histogram in shared memory (shared among threads in a block and much faster to access than regular GPU memory) and increments each bin via atomic add operations. To optimize memory operations and allow for coalesced reads, each thread reads in 16 bytes at once, and adjacent threads in a warp read adjacent 16-byte segments.
    b. Each thread takes a different symbol and adds the frequencies of that symbol across all warp-level histograms to create the final histogram.

   c. Compute the PDF and CDF, quantizing the probabilities to a specified number of bits and using a prefix sum from the CUB library to get the CDF.

2. **ANS encode 16-byte blocks**: Overall, each thread takes a different section of the input data, and each lane in a warp has a separate ANS state (32 ANS states per block). Each GPU block works separately, handling a different section of the input data and writing the output to a different location in memory.

   a. Load the PDF and CDF into shared memory for fast access.

   b. For each input symbol (accessed such that reads are coalesced):

      i. If the ANS state for any thread is too large, write some bits to the output. Warp-level primitives are used to ensure that each thread writes to a different byte of memory, with output bytes remaining consecutive.

      ii. Update the state: $x = \lfloor x/\text{pdf}(s) \rfloor \cdot M + \text{cdf}(s) + (x \mod \text{pdf}(s))$, where x is the state and s is the input symbol.

   c. Write the final state to the header.

3. **ANS encode the rest**: also encode the remaining bytes that don't fully comprise a 16-byte block.

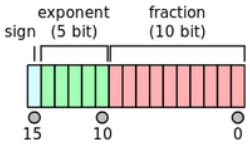
4. **"Coalesce" the ANS output blocks**: each GPU block wrote its output to a separate section of memory, so there are gaps between the outputs of different blocks. So, DietGPU performs a prefix sum on the number of compressed bytes per block and uses that prefix sum to write all blocks to one contiguous chunk of memory.

2.2. Float Compression

Float compression has two stages. First, each floating point number is split into two parts: the exponent, which will be compressed, and the rest of the bits, which will be directly written to the output. Next, the exponents are compressed using ANS.

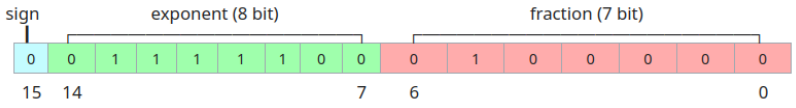
Each type of float supported is split differently:

1. **Float16**:



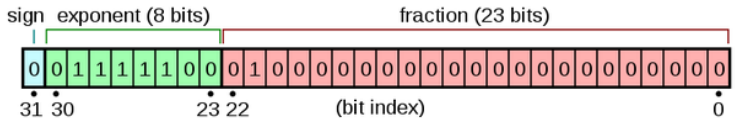
   The exponent is shorter than 8 bits, so the most significant 8 bits are compressed and the least significant 8 bits are left uncompressed.

2. **BFloat16**: This is the most significant 16 bits of a Float32. It is a compressed version of a 32-bit float that keeps the dynamic range while reducing the precision of the significand.



   To isolate the exponent for compression, DietGPU circularly shifts the bits to the left by 1 (moving the sign bit to the end of the "fraction" bits) and then takes the most significant 8 bits.

3. **Float32**:



   To get the ANS compression input, DietGPU circularly shifts the bits to the left by 1 and then takes the most significant 8 bits.

The uncompressed section requires some special handling. Unlike with Float16, the number of bits left uncompressed (24) is not a power of two, so it doesn't fit into any of the standard integer types. So, DietGPU stores the uncompressed section in two datasets, the first containing the least significant 16 bits of each floating point number and the second containing the next 8 bits.

For Float32 data, the output of the compressor is as follows:

```
[Header] [Lower 2 bytes of all floats] [Next 1 byte of all floats] [ANS compressed exponents]
```

where each section in brackets is 16-byte aligned.

2.3. ANS Decompression

Overall, ANS decoding on a GPU works as follows:

1. **Decode Table**: Build the rANS decoding table (pdf/cdf) from the compression header.
    a. Each thread creates a pdf bucket in shared memory.
    b. Do exclusive scan on the shared memory and create the overall cdf table.
2. **Perform decoding**: Since the decoder have no idea how large the decompression job is, it just launch a grid that is sufficiently large enough to saturate the GPU and loop until there isn't enough work to do. The decoder combines multiple warps into one block to ensure that the decoding process is warp uniform.
    a. Load the PDF and CDF into shared memory for fast access.
    b. For each per-block size data:
        i. If the compressed input is not be a whole multiple of a warp (remainder). Only the lanes that cover the remainder could have read a value in the input, and thus, only they can write a value in the output. This partial data will be handled first.
        ii. Decode one warp at a time until there is no more compressed data to decode. We get $s = cdf(x \mod M)$ and update the state $x = \lfloor x/M \rfloor * \text{pdf}(s) + x \mod M - cdf(s)$.
    c. Write the decoded result to the output.

2.4. Float Decompression

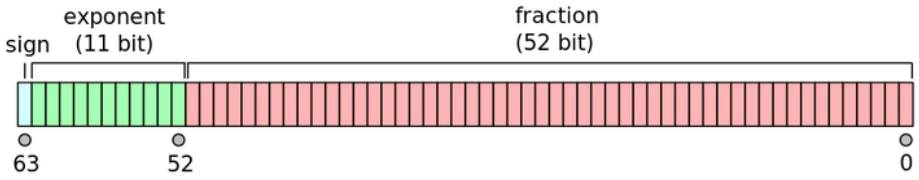
For float decompression, the decoder will first decompress the exponents using ANS, then it will join the decompressed exponents with the non-compressed fraction to get the original float value.

1. **Float16**: Multiply the compressed part by 256 and add it with the non-compressed part.
2. **BFloat16**: This is similar to Float16, the only difference is that after multiplying the compressed part by 256 and add it with the non-compressed part, we shift the bits to the right by 1 using GPU intrinsics.
3. **Float32**: Because the range changes, to join Float32, we multiply the compressed part (8 bits) by 16777216 and add it with the non-compressed part (24 bits), and then shift the bits to the right by 1.

3. Methods

We extended DietGPU by implementing Float64 compression and decompression.

3.1. Float64 Compression



The exponent has more than 8 bytes, so we choose to create two different datasets to be ANS compressed. The first dataset consists of the most signifcant 8 bits of the 11-bit exponent, and the second dataset consists of the next 3 bits of the exponent and first 5 bits of the significand.

We decided to compress the datasets separately (*i.e.*, run ANS twice) because we expect the datasets to be quite different from each other in terms of symbol frequency, so combining the datasets would increase the entropy. This was verified by a simple Python experiment: we randomly sampled 1 million random Float64 values from a standard normal distribution and computed the optimal compression ratio using the Shannon entropy of the data. For two rounds of ANS compression, the optimal compression ratio (`output bytes / input bytes`) of the floats was 0.87, whereas the optimal compression ratio for using a combined ANS dataset was 0.90.

The uncompressed 48 bytes are handled similar to Float32, with the overall compressor output as follows:

```
[Header] [Lower 4B of all floats] [Next 2B of all floats] [1st ANS dataset] [2nd ANS dataset]
```

3.2. Float64 Decompression

We will pass two datasets to the `joinFloat` function (which combines the floating point exponents and significands), consisting 16 bits compressed data in total. Each dataset has 8 bits of compressed data because we are compressing the datasets separately.

When joining the dataset, we have the option of performing non-aligned joining, where we join the floats one by one, or we can choose 16-aligned joining, in which we decode 128 bits in a vectorized way. Vectorized operations are more optimal than non-vectorized operations, as they allow more data accesses to happen in parallel.

3.3. Sparse Float Compression

For sparse float compression, we begin by computing a bitmask that identifies the nonzero indices within the input data. Then, we perform an exclusive scan operation on this bitmask to generate a sequence that assists in constructing an array containing only the indices of the nonzero values, which we call the "index array". The exclusive scan is performed by the `thrust` library (link), which has efficient GPU implementations of many basic algorithms.

To construct the index array, we scan the bitmask again, comparing the values at index `idx` and `idx+1`. A mismatch indicates a valid index corresponding to a nonzero number. Following this, we apply a standard float compression technique, followed by an ANS compressor. We store the generated bitmask, and for further compression gains, we apply the same float compression technique and ANS compressor to compress the bitmask. This comprehensive process enables us to efficiently compress sparse floating-point data while preserving essential information for subsequent processing and analysis, optimizing both storage and transmission efficiency. We provide an example for the compression process:

Original data:

```
0 0 0 0 0 0 1 0 4 5
```

Bitmap:

```
0 0 0 0 0 0 1 0 1 1
```

After exclusive scan:

```
0 0 0 0 0 0 0 1 1 2
```

Getting index array:

```
0 0 0 0 0 0 6 0 8 9
```

Dense index array:

```
6, 8, 9
```

Dense original data:

```
1, 4, 5
```

Since the exclusive scan loses the last bit of information, we add a separate processing step to handle the last bit to get the correct index array. After getting dense index array, we put the array of dense input data into the regular DietGPU float compressor.

The overall compressed data looks like:

```
[Header] [Bitmap, packed into bits] [Output of Float Compression on Nonzeros]
```

3.4. Sparse Float Decompression

The sparse float decompression method is a counterpart process to our compression process. To decode the original data, we begin by decompressing the nonzero values and the bitmask using an ANS decoder, followed by a float decoder, following the same principles as other DietGPU decoding processes. With the decoded data in hand, we utilize the nonzero index array to accurately place the decoded values into the dense array, effectively recovering the sparse data. Specifically, the data at `idx` in the decoded dense array corresponds to the value of `bitmap[idx]`, enabling us to reconstruct the sparse floating-point data correctly. This parallelizable scan and recovery approach ensures the efficient restoration of the original data while preserving essential information, thereby optimizing both storage and transmission efficiency.

4. Our Code

We have modified or added the following files:

dietgpu/float/FloatTest.cu: This is the main testing file, to which we added tests for Float64 compression.

dietgpu/float/FloatBenchmark.cu *(added)*: This contains bandwidth and compression ratio benchmarking code for float compression. It also tests correctness.

dietgpu/float/SparseFloatBenchmark.cu *(added)*: This contains bandwidth and compression ratio benchmarking code for sparse float compression. It also tests correctness.

dietgpu/float/GpuFloatCompress.cu: We updated the `getMaxFloatCompressedSize` function to support Float64 compression.

dietgpu/float/GpuFloatCompress.cuh: We added the core logic for Float64 compression, including performing ANS compression on two separate datasets, splitting Float64 data into the exponent and significand, determining batch size, and computing ANS histograms. For input data that is 16-byte aligned, Float64 compression is vectorized for performance (*i.e.*, data is read in blocks of 16 bytes, which allows us to take advantage of the GPU's memory bandwidth).

dietgpu/float/GpuFloatDecompress.cuh: We added the primary logic for Float64 decompression, including performing two rounds of ANS decompression, merging the exponents with the significands. For data that is 16-byte aligned, Float64 compression is vectorized (*i.e.*, data is written to the output in blocks of 16 bytes).

dietgpu/float/GpuFloatUtils.cuh: Adds Float64 support to float compression and decompression data structures.

dietgpu/float/GpuSparseFloatCompress.cu *(added)*: We use an interface that mirrors the base DietGPU `floatCompress`, and call our custom implementation for sparse float compression.

dietgpu/float/GpuSparseFloatDeompress.cu *(added)*: Similarly, we use an interface that mirrors the base `floatDecompress`, but call our custom implementation for sparse float decompression.

dietgpu/float/GpuSparseFloatCompress.cuh *(added)*: This file contains CUDA functions for sparse compression, such as generating a bitmap and constructing dense original data and dense index arrays.

dietgpu/float/GpuSparseFloatDecompress.cuh *(added)*: This file houses CUDA functions for sparse decompression.

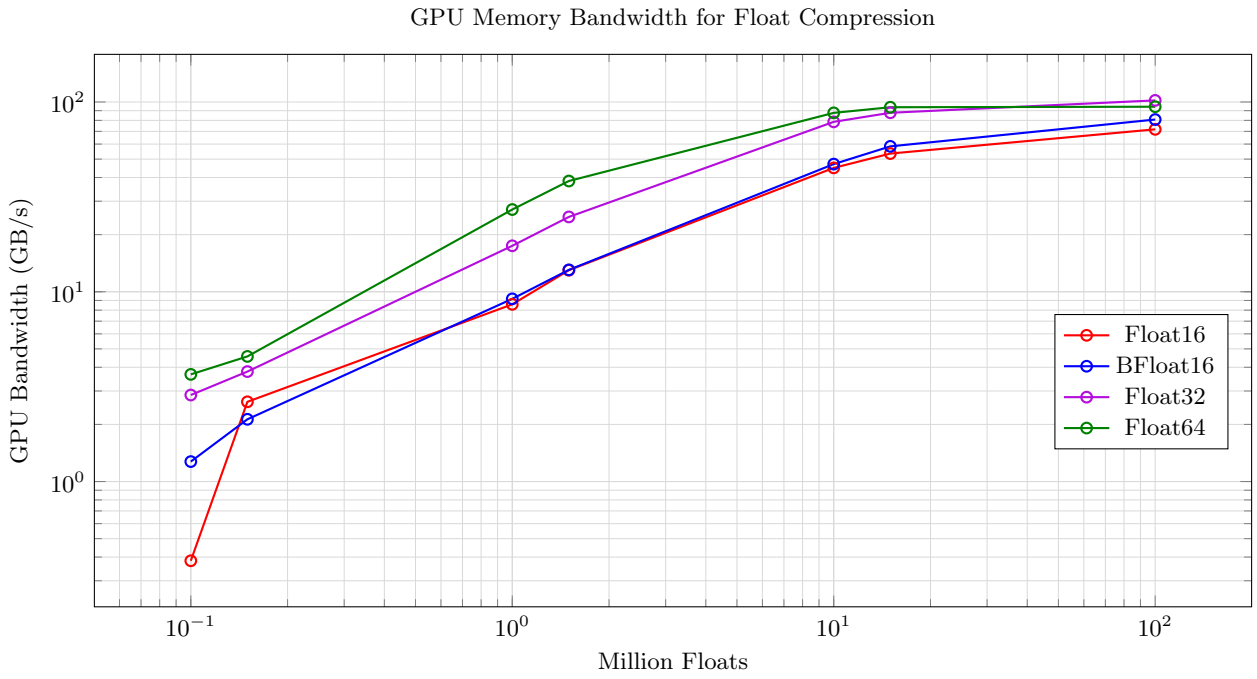
Our changes, line-by-line, can be found on the "Files Changed" tab of Github's compare tool.

5. Results

Here are some sample runs using an NVIDIA TITAN X GPU. We use the normal distribution with parameters (0, 1) to approximate a typical quasi-Gaussian data distribution commonly observed in real machine learning datasets.

5.1. Float64 Experiments

5.1.1 Float64 Compression

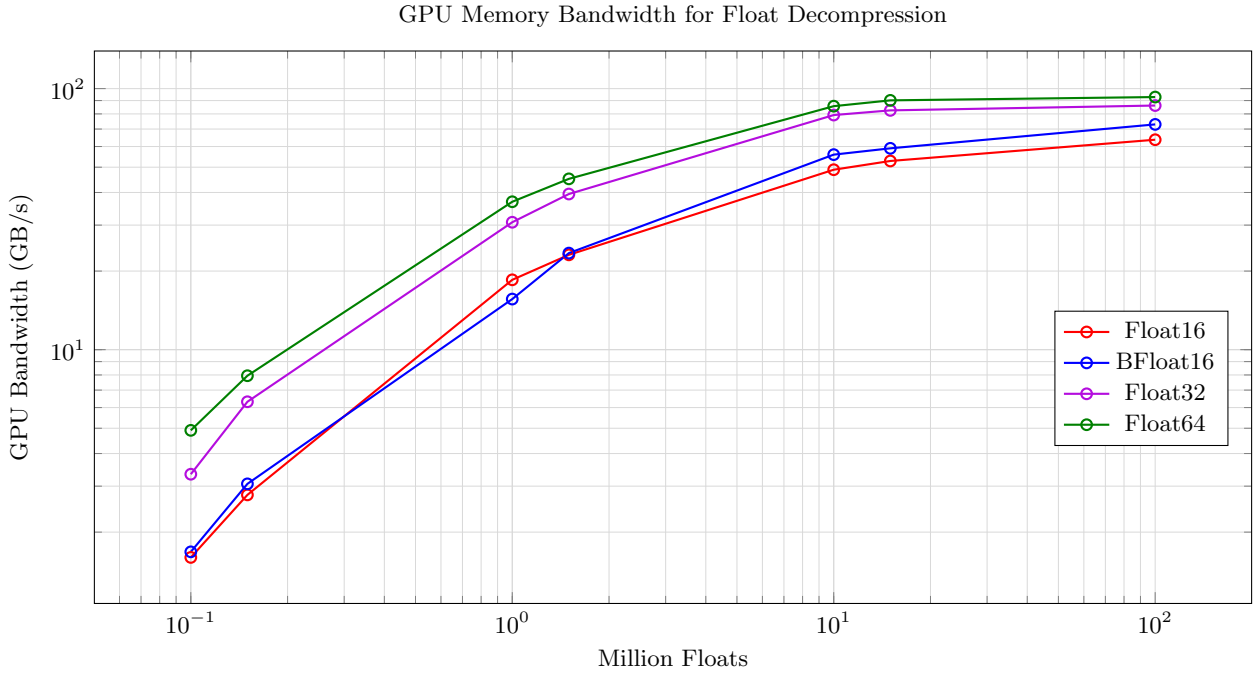


The peak memory bandwidth for the NVIDIA Titan X is approximately 500 GB/s. When working with 100 million Float65 values, our operations operate at around 1/5th of this peak bandwidth. In general, our results are consistent with those previously listed on the README of the DietGPU git repository. The compression bandwidth increases with the total number of bytes being compressed (number of floats, times bytes per float). This is because GPU computation has some overhead that remains constant with respect to the size of the data, such as GPU kernel initialization. So, a general algorithm will approach its optimal performance as the amount of data increases.

Also, Float32 and Float64 require less computation per byte of data than Float16. This is because Float16 performs ANS compression on half of each float, whereas Float32 and Float64 only perform ANS compression on one quarter of each float. By proportionally reducing the time required for data compression, the overall throughput is increased. However, as more of each float is left uncompressed, the optimal compression ratio is worse for Float32 and Float64.

For 100 million floats, Float64 performance slightly worse than Float32. This could have to do with overhead involved in performing two rounds of ANS compression (*e.g.*, having to generate two ANS histograms).

5.1.2 Float64 Decompression



The decompression bandwidth is comparable to the compression bandwidth, *i.e.*, it also achieves 1/5 of the peak GPU memory bandwidth. Unlike with compression, Float64 does not achieve worse bandwidth than Float32 at 100 million floats. As decompression does not involve building ANS histograms, this indicates that generating two ANS histograms might be causing the performance degredation for Float64 compression.

5.1.3 Float64 Compression Ratio: Random Data

Table 1: **Experiment**: Draw 10 million floats from $N(0, 1)$. Calculate the compression ratio as `output bytes / input bytes` (so, a compression ratio closer to 1 indicates less compression).

Float Type	**Float16**	**BFloat16**	**Float32**	**Float64**
Comp. Ratio	0.87	0.68	0.83	0.88
Opt. Ratio*	—	0.66	0.83	0.87

** From the empirical Shannon entropy of 1 million samples from $N(0, 1)$.*

We measure the compression ratio on large dataset generated form from a standard normal distribution $N(0, 1)$.[1] Our implementation achieves a compression ratio close to the optimal ratio determined by the Shannon entropy. The compressibility of Float64 data, however, is limited: its significand, which has high entropy, takes up about 83% of the bits of a Float64.

Overall, our results highlight both the efficiency and effectiveness of our implementation (with respect to the theoretical compressibility of the data): it achieves excellent runtime performance, while achieving compression rates close to the Shannon entropy.

Table 2: Real-world simulation data:

Sim Dataset**	Comet	Head Impact	Control	Plasma
F64 Comp.	0.91	0.86	0.90	0.88

** *userweb.cs.txstate.edu/~burtscher/research/datasets/FPdouble/*

5.1.4 Float64 Compression Ratio: Real-world Data

We also conducted float compression experiments on real-world Float64 data from scientific computing datasets. On these datasets, we achieved compression rates that are comparable to the ones achieved for the random data, demonstrating that the results from the random data experiment reasonably match performance on real-world data.

5.1.5 Comparison with a CPU rANS Implementation

We compare our float compression and decompression bandwidth with a source mentioned in lecture: https://github.com/rygorous/ryg_rans. This is an efficient C++-based implementation of several rANS variants (including a SIMD version) by Fabian Giesen. The peak bandwidth listed is with the SIMD compressor, which achieves up to 550 MB/s compression and decompression bandwidth.[2] The peak bandwidth of Float64 compression and decompression is over 90 GB/s, or over 160× higher bandwidth.

5.1.6 Comparison with ZSTD

We also wanted to compare our performance to ZSTD, a highly-optimized state-of-the-art compressor. ZSTD uses tANS as an entropy coding step after performing LZ77, so it is a reasonable comparison metric.

On the Comet dataset from the real-world data listed above, ZSTD achieves a compression rate of 139 MB/second with a compression ratio of 0.88, whereas DietGPU achieves a throughput of about 50 GB/second and a superior 0.91 compression ratio. GPU acceleration causes a 360× bandwidth performance increase compared to ZSTD, while achieving comparable compression.

Similarly, on the Head Impact dataset, ZSTD achieved a compression rate of 423 MB/second with a compression ratio of 0.94. In contrast, DietGPU achieved a compression rate of about 60 GB/second with a slightly better compression ratio of 0.86. This amounts to a 141× bandwidth performance increase.

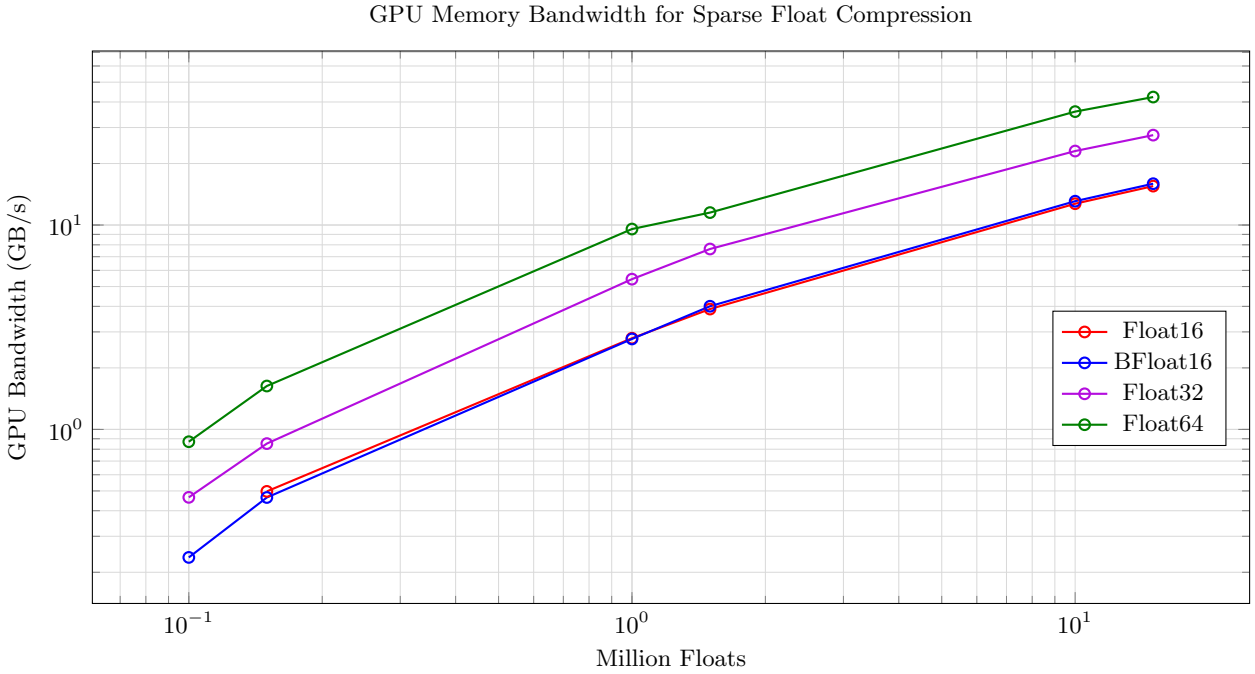
5.2. Sparse Float Experiments

All our experiments are done using a normal distribution $N(0, 1)$, and in these datasets, approximately 50% of the values are zeros.

[1] Standard deviation empirically does not have much impact on the compression ratio.

[2] The performance listed on the Github README is slightly better than the performance achieved by running the implementation on the `popeye2.stanford.edu` machine, so we will use the numbers reported by the author.

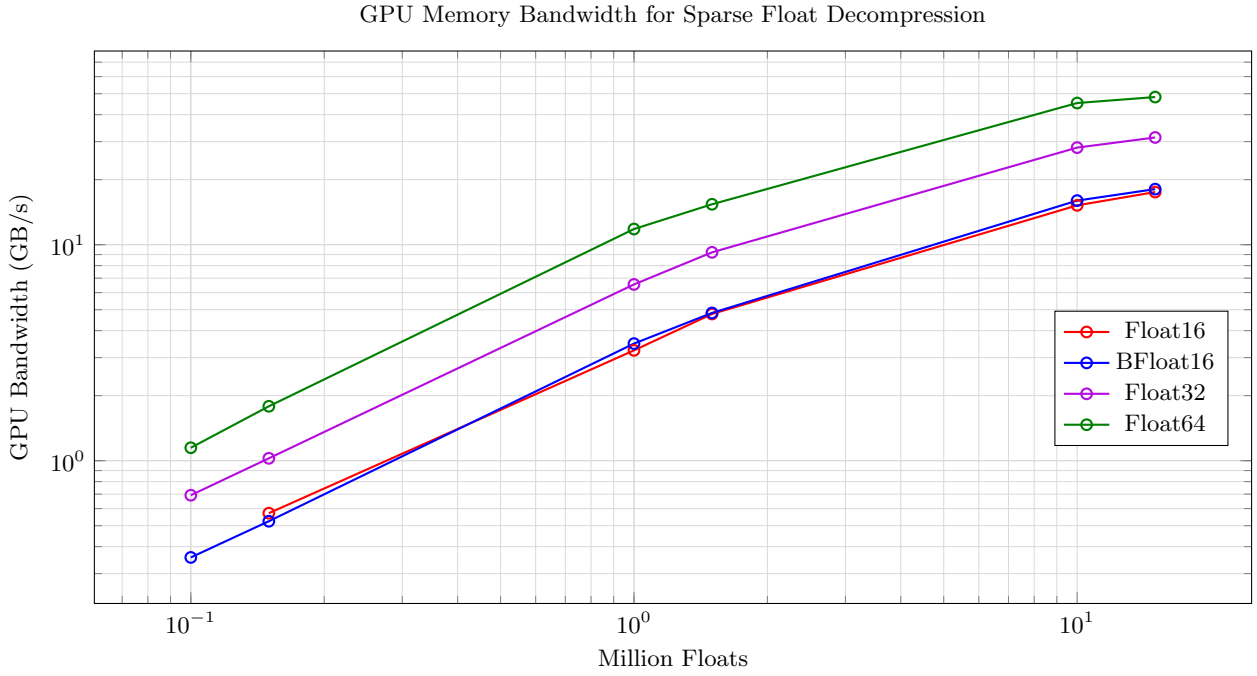
5.2.1 *Sparse Float Compression*



Generally, sparse compression throughput is lower compared to the baseline without handling sparsity. However, the results are sensible because we sparse float compression involves more computation: we launch multiple CUDA kernels for preprocessing. This extra computation comes with the tradeoff of achieving good compression ratios for sparse data. Additionally, we must use CUDA synchronization before calling the `thrust` library's exclusive scan function, which could cause some performance degradation.

We still, however, achieve reasonably good GPU memory bandwidth: it approaches 50 GB/s for a size-15 million Float64 dataset, which is about 1/10th of the peak GPU memory bandwidth.

5.2.2 Sparse Float Decompression



Similarly, the decompression part exhibits lower throughput compared to the baseline, non-sparse, DietGPU float compression.

5.2.3 Sparse Float Compression Ratio

Table 3: **Experiment**: Sample float data from from $N(0, 1)$, with a 50% probability that any value is zero. The compression ratio is `output bytes / input bytes`.

Float Type	Float16	BFloat16	Float32	Float64
Comp. Ratio	0.49	0.40	0.45	0.45
Base DietGPU*	0.76	0.66	0.83	0.85

** Regular DietGPU, without specialized float compression.*

We compare the compression rates between sparsity compression and the base DietGPU compression. We can observe an almost 2× improvement in compression ratio, which is expected since we are dealing with data where approximately 50% of values are zeros. Considering the significant savings in compression ratio, the overhead incurred in processing the data is deemed acceptable and justified by the substantial improvement in compression efficiency.