

SLEEK: Compressing Memory Copies for Floating-Point Data on GPUs

Anju Mongandampulath Akathoott
Department of Computer Science
Texas State University
San Marcos, USA
anju.m.a@txstate.edu

Andrew Rodriguez
Department of Computer Science
Texas State University
San Marcos, USA
andrew.rodriguez@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, USA
burtscher@txstate.edu

Abstract—Scientific codes that process large amounts of data often leverage GPUs for performance gains, but the relatively small global memory capacity can be problematic. As a remedy, we present SLEEK, a software-based main-memory compression technique for single- and double-precision floating-point data that works on existing systems. Its high speed, comparable to that of memory copies on GPUs, makes it a promising solution for scientific applications grappling with memory-size limitations. SLEEK provides both lossless and guaranteed-error-bounded lossy compression that is CPU–GPU compatible. It supports all IEEE 754 floating-point values, including infinities, NaNs, and subnormals. SLEEK’s lossy compressor yields geometric-mean compression ratios of up to 49.8 and 98.9 on single- and double-precision SDRBench inputs. On an RTX 4090 GPU, it compresses up to $1.71\times$ and decompresses up to $1.65\times$ faster than the speed of device-to-device memory copies of the uncompressed data. SLEEK exceeds the throughput of existing GPU compressors while also yielding higher compression ratios than many of them.

Index Terms—Software-based memory compression, floating-point data, guaranteed error bounds, GPU parallelization

I. INTRODUCTION

Many scientific computations process large volumes of data, often on GPUs due to their performance benefits. One of the biggest challenges in this context is the limited global memory capacity of GPUs, which tends to be much smaller than the main memory of CPUs. Since GPU memory is built for very high throughput, it is tightly coupled with the system hardware and cannot be extended by plugging in more modules.

One possible solution is main-memory compression, that is, keeping the data compressed in the main (global) memory and decompressing it on the fly, at the time of reading. However, main-memory-compression approaches are not widely used since they either do not allow writes to the data, require special hardware, or are too slow, especially on GPUs.

Software-based related work such as Ligr+ [1] supports main-memory compression on CPUs but only for reading data. Applying such techniques on GPUs is challenging since GPUs have very high memory bandwidths, requiring the compression throughput to also be very high. If it is too low, any benefits from the reduced memory footprint will be nullified, yielding an overall slowdown of the application.

Whereas there are a few works on hardware compression of memory pages on GPUs [2]–[4], the only prior work we

could find on software main-memory compression for GPUs is MPLG [5], which targets read-only graph data structures. It decompresses the data as needed for processing but does not support re-compression. Also, since graph data structures such as CSR are index-based, MPLG employs a compression algorithm that is specific to integers. In contrast, scientific applications predominantly deal with floating-point data.

In this paper, we present SLEEK, a *software-based solution for fast main-memory compression and decompression of both single- and double-precision floating-point data on GPUs* that can be integrated into existing and new applications. As far as we know, there is no prior work on this topic. In the following description of SLEEK, *main-memory compression* refers to a pure software solution that works on existing hardware.

There are several key challenges in developing an effective main-memory compressor for floating-point data on GPUs. First, it should operate at speeds comparable to normal read and write speeds of global memory. Second, it should yield high-enough compression ratios to significantly reduce the memory footprint of the application, allowing larger datasets to be stored. Third, it should support lossless and various levels of lossy compression to allow the user to trade off precision for compression ratio and speed. In case of lossy compression, it should provide guaranteed error bounds to cap the maximum error. Fourth, it should support heterogeneous systems as it may be desirable to send compressed data to/from the GPU, necessitating CPU-based compression and decompression.

No existing compressors meet all of these criteria (Table I). Almost all of them are much slower than directly reading or writing data in global memory. The few algorithms that do approach this speed provide very low compression ratios. Most lossy compressors do not guarantee the user-selected error bound. Finally, almost no CPU/GPU compatible compressors exist. In contrast, SLEEK meets all of these demands.

(1) Throughput: SLEEK exceeds the performance of *memcpy* on a number of current high-end GPUs. This means it can copy data from one part of the memory to another faster than a conventional memory-copy operation, in addition to compressing or decompressing the data in the process. SLEEK achieves this by hiding the computation overheads almost completely with its high-speed implementation.

(2) Compression ratio: In lossless and especially in lossy

TABLE I: Comparison of compressor features

Compressor	Memcopy throughput	Guaranteed error bound	CPU-GPU compatible
MGARD-X	✗	✗	✓
FZ-GPU	✗	✗	✗
cuSZp	✗	✗	✗
PFPL	✗	✓	✓
BitComp	✗	✗	✗
SLEEK	✓	✓	✓

mode, SLEEK reaches compression ratios that are higher than those of some much slower algorithms. Moreover, it guarantees the user-provided point-wise absolute error bound. Most other lossy algorithms quantize the floating-point values by dividing each of them by twice the error bound before compressing the quantized values. During decompression, they de-quantize by multiplying by twice the error bound to recreate the original value. Due to the finite precision of the IEEE 754 floating-point representation, the recreated values may be imprecise and violate the error bound, in some cases by large amounts [6]. The new quantizer we developed for SLEEK does not suffer from this problem as it emulates all floating-point operations with integer instructions and decreases the error bound to the nearest power of two.

(3) Cross-system compatibility: Even though SLEEK’s main target is GPUs, it also includes fast CPU counterparts of its lossless and lossy compressors and decompressors. The compressed and the decompressed data produced by the GPU and CPU versions match bit-for-bit. This allows applications to use SLEEK to compress data on the host and decompress on the device and vice-versa as needed.

Many HPC applications such as deep learning [7] and bioinformatics [8] that process floating-point workloads face global-memory limitations in GPUs. Moreover, applications such as computational fluid dynamics [9], molecular dynamics [10], and n -body simulations [11] execute time-stepped iterations, often repeatedly processing floating-point data in tiles or blocks. SLEEK targets these kinds of applications.

Our experiments demonstrate that, on these types of data, SLEEK’s lossy compressor for double-precision floats yields geometric-mean compression ratios up to 98.9. It is faster than `cudaMemcopyD2D`¹ (of uncompressed data) by up to $1.71\times$ in compression and $1.65\times$ in decompression on an RTX 4090.

This paper makes the following main contributions.

- It introduces SLEEK, a new main-memory compression algorithm for single- and double-precision floating-point data that delivers speeds comparable to or exceeding the speed of `memcopy` on high-end GPUs.
- It describes both lossless and lossy algorithms that yield substantial compression ratios at high speeds. They support all floating-point values, including infinities, NaNs, and denormals. In case of lossy compression, the algo-

rithm guarantees point-wise absolute error bounds without the need to check for violations.

- It presents GPU and CPU implementations of SLEEK that produce compressed and decompressed outputs that are bit-for-bit identical, allowing data compressed on a GPU to be decompressed on a CPU and vice versa.
- It demonstrates the ability of SLEEK to deliver much higher throughput at a given compression ratio compared to the state-of-the-art algorithms from the literature.

Our CUDA and C++/OpenMP SLEEK implementations are available in open source as described in the artifact appendix.

The rest of the paper is organized as follows. Section II provides background. Section III discusses SLEEK in detail. Section IV describes the experimental methodology. Section V presents and analyzes the results. Section VI summarizes related work, and Section VII concludes the paper.

II. BACKGROUND

This section provides background on main-memory compression, the difference between lossless and lossy compressions, and guaranteeing point-wise error bounds in lossy compression of floating-point data.

A. Main-memory compression

Main-memory compression reduces the memory footprint of applications by applying compression and decompression at runtime on data stored in the main memory. Storing data in compressed format allows an application to avail more memory than is actually present. This is particularly beneficial for applications that repeatedly process chunks of large datasets on GPUs, which have relatively small amounts of global memory.

For example, consider a GPU with 8 GB of global memory. Assume that an application needs to repeatedly process two datasets, A and B, each of size 5 GB, one after the other. Since the combined size of A and B exceeds the memory capacity, the application will have to transfer the datasets back and forth between the GPU and the host, which is slow. In contrast, if there is a good compression algorithm that can compress each of these datasets into, say, 1 GB, the application can store both A and B in compressed format and use the remaining 6 GB space to decompress the needed dataset in each phase. At the end of the phase, the dataset is re-compressed. For such a solution to be useful, in addition to providing good compression ratios, the compression and decompression must operate at high speeds. SLEEK meets both of these criteria.

B. Lossless versus lossy compression

Applications that require highly accurate data prefer lossless compression methods that can be fully reversed to recreate the original data precisely from the compressed data. They apply a chain of transformations to the data to encode the information in less space. The requirement to be able to recreate the original data exactly limits the compression ratios (the ratio of the size of the original data to that of the compressed data) that such algorithms can deliver [12].

¹This is a GPU-to-GPU copy that does not involve the PCI bus or the CPU.

In contrast, if an application can tolerate some error in the recreated data, it is often better to use lossy compression, which can yield significantly higher compression ratios [13]. Many lossy compression algorithms employ a quantization function that maps the large range of floating-point values in the input to a smaller, finite set of discrete values, which are referred to as *bins*. Each bin represents a sub-range of the full span of the input values. The number of bins is usually much smaller than the number of distinct values in the input dataset, and, hence, multiple input values may get mapped to the same bin. This means that the sequence of bin numbers obtained from the original data generally contains fewer unique values and more discernible patterns, making it more compressible.

A chain of one or more lossless transformations is then applied to the bin number sequence generated by the quantization step, yielding the final compressed data. On decompression, the lossless transformations are reversed, returning the exact bin-number sequence. The following dequantization typically maps each bin number to the center of the sub-range of the floating-point values it represents. This is the source of the *loss* — all distinct original values in a sub-range are reconstructed to the center value of that sub-range. SLEEK supports both lossless and lossy compression.

C. Error bounds in lossy compression

Users of lossy compression may want to specify a *point-wise absolute error bound*, ϵ , indicating that, for each original value v , the reconstructed value r is acceptable if $v - \epsilon \leq r \leq v + \epsilon$. Leading lossy compressors perform quantization by dividing each floating-point value by 2ϵ and rounding the result to the nearest integer, and dequantization by multiplying the quantized value by 2ϵ [14]. In the rest of this paper, the term *error bound* refers to the point-wise absolute error bound.

Guaranteeing strict error bounds is a challenge due to the finite precision of floating-point operations and the non-uniform distribution of representable values in the IEEE 754 floating-point format [15]. Hence, many existing lossy compressors do not fully guarantee the error bound [16]. The few that do incorporate special mechanisms to deal with values for which they cannot guarantee the error bound, including losslessly encoding them inline [6] or storing them separately [14], [17]. This leads to a mixing of bin numbers and floating-point values in the output, which necessitates additional steps in the decompressor and reduces compressibility and throughput. The lossy compressor in SLEEK efficiently addresses this challenge in a novel way. It is able to guarantee the error bound on all values, including infinities (INF), not-a-number (NaN), and subnormals, without any special mechanism such as violation detection or outlier handling.

III. SLEEK ALGORITHM AND IMPLEMENTATION

SLEEK has two components — (1) a lossless compressor and (2) a lossy compressor that guarantees point-wise absolute error bounds. Both components support single- and double-precision floating-point inputs in IEEE 754 format. SLEEK includes CPU and GPU versions that are cross-compatible. For

example, one can compress on the CPU and decompress on the GPU or vice versa. In this section, we detail our algorithms, their key features, and their parallel implementations.

Algorithm 1 Lossless SLEEK Encoder

```

1: TB ← 32;                                ▷ number of bits in a float
2: MSB ← 1 <<< (TB - 1);                    ▷ only MSB set
3: EXP_MSK ← 0xFF00'0000; ▷ exp mask after rotation
4: EXP_ONE ← 0x0100'0000; ▷ after rotation
5: max_val ← 0;
6: for each index  $i$  in the group do
7:   val ← __float_as_int(in[i]);           ▷ interpret as uint
8:   val ← rotate_left(val);                ▷ rotate sign into LSB
9:   if ((val & EXP_MSK) ≠ 0) then
10:    val ← val - MSB;
11:    if ((val & EXP_MSK = 0) ∨ (val ≥ MSB)) then
12:      val ← val - EXP_ONE;
13:    val ← TCMS((signed)val);             ▷ to magnitude sign
14:    temp[i] ← val;
15:    max_val ← max(max_val, val);
16:  $n$  ← TB - __clz(max_val);
17: return  $n$ ;

```

A. Lossless compressor

This subsection details the operation of SLEEK's lossless compressor on single-precision floating-point values. The algorithm for double-precision inputs is analogous.

The lossless compressor groups the input values into 512-byte subchunks and applies a data reduction technique to each group. It exclusively operates on the bit-wise integer interpretation of each 32-bit value, that is, the exact binary IEEE 754 representation (like *reinterpret_cast*), without any rounding or numerical conversion. It transforms these integers with the goal of increasing the number of leading zeros in the common case as shown in Algorithm 1. Based on our analysis of scientific data from various domains [18], we found the common case to be floating-point values with (de-biased) exponents that cluster near zero. The high-level idea is to bijectively map the biased exponents of the IEEE 754 representation to new values such that the typically frequent exponents turn into values with many leading zero bits, which are then compressed. The parallelization and code optimizations are discussed in Section III-D.

The constant TB in Algorithm 1 denotes the number of bits in each input value. For single-precision floats, TB = 32. The constant MSB is a bit sequence of length TB, where the only set bit is in the most significant position. The constant EXP_MSK has the 8 most significant bits set, corresponding to where the exponent field is after rotating a float value by one position to the left. The variable *max_val* is initialized to zero (Line 5). The algorithm iterates over all values in the group (Line 6). The bit pattern representing a float is first interpreted as an unsigned integer (Line 7) and written to *val*, which is then rotated to the left by one position to move the

most significant bit, which represents the sign of the original float value, to the least significant position (Line 8). Since our goal is to maximize the number of leading zeros, moving the sign bit to the LSB position helps by getting it out of the way.

After the rotation, the top 8 bits of *val* correspond to the exponent of the float. Due to the bias in the exponent of IEEE 754 floating-point values, the frequent native exponents tend to have few or no leading zero bits. We bijectively map the 256 possible values the exponent can take into a new set of 256 values such that the exponents expected to occur more frequently are mapped to values with many leading zeros. SLEEK performs this mapping as follows. If the exponent is zero, we leave it as is (Line 9) because zero is a relatively frequent value. Otherwise, we subtract 128 from it (Line 10), which in effect de-biases the exponent (the actual bias is 127). We subtract 128 instead of 127 so as not to map the highest possible exponent to a negative value, which would complicate the decompressor. If the result of the subtraction is zero, we re-map it to -1 because 0 is already taken and the mapping must be lossless (Lines 11-12). Since the mapped value 0 is lowered by 1, all smaller values are also lowered by 1 to maintain the bijective nature of the mapping (also Lines 11-12).

The bit pattern in *val* is then interpreted as a signed integer and converted from twos-complement to a modified magnitude-sign (TCMS) format [12], which often yields more leading zeros, before it is stored into a temporary array (Lines 13-14). During processing, we also identify the maximum (unsigned) value *max_val* of all transformed values in the group (Line 15). In the end, we count the number of leading zeros in *max_val*. Subtracting this count from *TB* yields *n*, the number of bits required to losslessly represent all transformed values in the group as the bits at positions above *n* are guaranteed to be zeros.

As the last step, we remove $TB - n$ leading zeros from all transformed values and pack them into the output, where each encoded value occupies *n* bits. This is the step that actually reduces the data size. Since the value of *n* is required to demarcate the encoded values during decompression, we also store *n* as part of the compressed output. The decompression simply follows the reverse transformations in the opposite order, reproducing the original data bit-for-bit.

Note that each step of the compressor and decompressor compiles into just a few fast integer machine instructions. None of the steps involve any floating-point instructions.

B. Lossy compressor

SLEEK’s lossy compressor guarantees point-wise absolute error bounds on all input values including infinities, NaNs, and denormals. Conceptually, it quantizes a value *v* by dividing it by twice the error bound (2ϵ) and rounding the result. Next, it typecasts the result to an integer, which is then transformed into magnitude-sign format, yielding the bin number of *v*. The intuition here is that several values that are nearby on the number line may fall into the same bin, helping achieve higher compression ratios in the later data-reduction step. As

mentioned, the corresponding dequantization process (multiplication of the quantized value by 2ϵ) is not guaranteed to produce a value *r* in the range $[v - \epsilon, v + \epsilon]$ because floating-point results must be rounded to a representable value. SLEEK employs two techniques to avoid this problem.

First, it adjusts the user-provided error bound to the nearest smaller power of 2. This yields anywhere between a 1 and 2 times tighter error bound than what the user requested. Being a power of 2, multiplying or dividing a value *x* by (twice) the adjusted error bound is tantamount to adding or subtracting an integer from the exponent field of *x*, meaning that no rounding error is introduced as the result can be precisely represented. ZFP also uses error bounds that are powers of two [19].

Second, SLEEK only quantizes values that fall in the range where representable floating-point values are no more than the adjusted error bound apart. Outside of this range, there is no reason to use quantization. In fact, it is detrimental as we show below. Instead, SLEEK encodes those values losslessly by applying a new transformation that creates more leading zeros. Moreover, SLEEK ignores the sign of all values that quantize to 0 but preserves it losslessly for all other values.

Another important feature of SLEEK is that it performs all required operations such as multiplication and division by 2ϵ and rounding using only integer addition/subtraction and bit-wise operations on the exponent and mantissa fields of the float. This has two key advantages — (i) It offers better portability, since integer arithmetic is guaranteed to produce the exact same results on different systems and is not subject to rounding modes, and (ii) simple integer operations tend to have a lower latency than floating-point operations.

1) *Inner workings of the lossy compressor:* We demonstrate the operation as well as important aspects of our approach on a mini floating-point format that adheres to IEEE 754 rules. Each float in this example is represented by 7 bits: the most significant bit is the sign bit (*S*), the next 4 bits are the biased exponent (*E*), and the 2 least significant bits form the mantissa (*M*). The bias is 7 (*Y*). The value of a normal float represented in this format is: $value = -1^S \times 2^{E-Y} \times 1.M$. The first column in Table II lists the representation of all positive values in this format. The operation of our algorithm on negative values is symmetric (not shown). Note that the first 4 rows represent NaN and infinity values and the last 4 rows represent subnormal values, including zero. The second column of the table shows the value of the floats in decimal format.

SLEEK employs several constants and parameters. The constant *m* represents the number of mantissa bits, which is 2 in our example. The user provides the value of the point-wise absolute error bound. We round it down to the nearest power of 2 and use this adjusted value as the error bound ϵ for the reasons stated above. Note that this adjustment lowers the user-provided value by a factor of 1.5 on average. The parameter $B = 2^m$ stands for the number of unique mantissa values. $B = 4$ in our example. The parameter *EB2* is twice the adjusted error bound ϵ and *inv_EB2* is $1/EB2$. Assuming a user-provided error bound of 0.3, this yields an adjusted error bound $\epsilon = 0.25$, $EB2 = 0.5$, and $inv_EB2 = 2.0$.

TABLE II: Example showing SLEEK’s lossy compression that guarantees the error bound on all values. For comparison, the last 6 columns demonstrate what happens if the values are quantized conventionally.

Floating-point number		SLEEK				Scenario: Quantize all values					
Bit pattern	Original value (v)	Category	Encoded value (Bin)	Decoded value	Error	u = v / EB2_Q	Rounded u	b (Bin)	w = b * EB2_Q	Rounded w	Error
0-1111-11	NaN	Special values (preserved losslessly)	70	NaN	0.000	NaN	NaN	122	NaN	NaN	0.000
0-1111-10	NaN		68	NaN	0.000	NaN	NaN	122	NaN	NaN	0.000
0-1111-01	NaN		66	NaN	0.000	NaN	NaN	122	NaN	NaN	0.000
0-1111-00	INFINITY		64	INFINITY	0.000	INFINITY	INFINITY	120	INFINITY	INFINITY	0.000
0-1110-11	224.000000	Normal values that are preserved losslessly	62	224.00	0.000	336.000	INFINITY	120	INFINITY	INFINITY	INFINITY
0-1110-10	192.000000		60	192.00	0.000	288.000	INFINITY	120	INFINITY	INFINITY	INFINITY
0-1110-01	160.000000		58	160.00	0.000	240.000	INFINITY	120	INFINITY	INFINITY	INFINITY
0-1110-00	128.000000		56	128.00	0.000	192.000	192.000	116	120.000	128.000	0.000
0-1101-11	112.000000		54	112.00	0.000	168.000	160.000	114	100.000	96.000	16.000
0-1101-10	96.000000		52	96.00	0.000	144.000	160.000	114	100.000	96.000	0.000
0-1101-01	80.000000		50	80.00	0.000	120.000	128.000	112	80.000	80.000	0.000
0-1101-00	64.000000		48	64.00	0.000	96.000	96.000	108	60.000	64.000	0.000
0-1100-11	56.000000		46	56.00	0.000	84.000	80.000	106	50.000	48.000	8.000
0-1100-10	48.000000		44	48.00	0.000	72.000	80.000	106	50.000	48.000	0.000
0-1100-01	40.000000		42	40.00	0.000	60.000	64.000	104	40.000	40.000	0.000
0-1100-00	32.000000		40	32.00	0.000	48.000	48.000	100	30.000	32.000	0.000
0-1011-11	28.000000		38	28.00	0.000	42.000	40.000	98	25.000	24.000	4.000
0-1011-10	24.000000		36	24.00	0.000	36.000	40.000	98	25.000	24.000	0.000
0-1011-01	20.000000		34	20.00	0.000	30.000	32.000	96	20.000	20.000	0.000
0-1011-00	16.000000		32	16.00	0.000	24.000	24.000	92	15.000	16.000	0.000
0-1010-11	14.000000		30	14.00	0.000	21.000	20.000	90	12.500	12.000	2.000
0-1010-10	12.000000		28	12.00	0.000	18.000	20.000	90	12.500	12.000	0.000
0-1010-01	10.000000		26	10.00	0.000	15.000	16.000	88	10.000	10.000	0.000
0-1010-00	8.000000		24	8.00	0.000	12.000	12.000	84	7.500	8.000	0.000
0-1001-11	7.000000		22	7.00	0.000	10.500	10.000	82	6.250	6.000	1.000
0-1001-10	6.000000		20	6.00	0.000	9.000	10.000	82	6.250	6.000	0.000
0-1001-01	5.000000		18	5.00	0.000	7.500	8.000	80	5.000	5.000	0.000
0-1001-00	4.000000		16	4.00	0.000	6.000	6.000	76	3.750	4.000	0.000
0-1000-11	3.500000	14	3.50	0.000	5.250	5.000	74	3.125	3.000	0.500	
0-1000-10	3.000000	12	3.00	0.000	4.500	5.000	74	3.125	3.000	0.000	
0-1000-01	2.500000	10	2.50	0.000	3.750	4.000	72	2.500	2.500	0.000	
0-1000-00	2.000000	8	2.00	0.000	3.000	3.000	68	1.875	2.000	0.000	
0-0111-11	1.750000	8	2.00	0.250	2.625	2.500	66	1.563	1.500	0.250	
0-0111-10	1.500000	6	1.50	0.000	2.250	2.500	66	1.563	1.500	0.000	
0-0111-01	1.250000	6	1.50	0.250	1.875	2.000	64	1.250	1.250	0.000	
0-0111-00	1.000000	4	1.00	0.000	1.500	1.500	60	0.938	1.000	0.000	
0-0110-11	0.875000	4	1.00	0.125	1.313	1.250	58	0.781	0.750	0.125	
0-0110-10	0.750000	4	1.00	0.250	1.125	1.250	58	0.781	0.750	0.000	
0-0110-01	0.625000	2	0.50	0.125	0.938	1.000	56	0.625	0.625	0.000	
0-0110-00	0.500000	2	0.50	0.000	0.750	0.750	52	0.469	0.500	0.000	
0-0101-11	0.437500	2	0.50	0.063	0.656	0.625	50	0.391	0.375	0.063	
0-0101-10	0.375000	2	0.50	0.125	0.563	0.625	50	0.391	0.375	0.000	
0-0101-01	0.312500	2	0.50	0.188	0.469	0.500	48	0.313	0.313	0.000	
0-0101-00	0.250000	2	0.50	0.250	0.375	0.375	44	0.234	0.250	0.000	
0-0100-11	0.218750	0	0.00	0.219	0.328	0.313	42	0.195	0.188	0.031	
0-0100-10	0.187500	0	0.00	0.188	0.281	0.313	42	0.195	0.188	0.000	
0-0100-01	0.156250	0	0.00	0.156	0.234	0.250	40	0.156	0.156	0.000	
0-0100-00	0.125000	0	0.00	0.125	0.188	0.188	36	0.117	0.125	0.000	
0-0011-11	0.109375	0	0.00	0.109	0.164	0.156	34	0.098	0.094	0.016	
0-0011-10	0.093750	0	0.00	0.094	0.141	0.156	34	0.098	0.094	0.000	
0-0011-01	0.078125	0	0.00	0.078	0.117	0.125	32	0.078	0.078	0.000	
0-0011-00	0.062500	0	0.00	0.063	0.094	0.094	28	0.059	0.063	0.000	
0-0010-11	0.054688	0	0.00	0.055	0.082	0.078	26	0.049	0.047	0.008	
0-0010-10	0.046875	0	0.00	0.047	0.070	0.078	26	0.049	0.047	0.000	
0-0010-01	0.039062	0	0.00	0.039	0.059	0.063	24	0.039	0.039	0.000	
0-0010-00	0.031250	0	0.00	0.031	0.047	0.047	20	0.029	0.031	0.000	
0-0001-11	0.027344	0	0.00	0.027	0.041	0.039	18	0.024	0.023	0.004	
0-0001-10	0.023438	0	0.00	0.023	0.035	0.039	18	0.024	0.023	0.000	
0-0001-01	0.019531	0	0.00	0.020	0.029	0.031	16	0.020	0.020	0.000	
0-0001-00	0.015625	0	0.00	0.016	0.023	0.023	12	0.015	0.016	0.000	
0-0000-11	0.011719	0	0.00	0.012	0.012	0.008	6	0.005	0.008	0.000	
0-0000-10	0.007812	0	0.00	0.008	0.012	0.008	6	0.005	0.008	0.000	
0-0000-01	0.003906	0	0.00	0.004	0.012	0.008	6	0.005	0.008	0.000	
0-0000-00	0.000000	0	0.00	0.000	0.000	0.000	0	0.000	0.000	0.000	

Assume we want to quantize a floating-point value v . First, we compute $v \times inv_EB2$ and round the result. Next, we typecast the rounded float to an integer and transform it into magnitude-sign format, which becomes the bin number b of v . The fourth column of Table II lists the bin numbers. They are all even because the least significant bit, which is the sign, is 0 in all cases since we only show positive values. The dequantization reverses these operations. It first extracts the sign bit and the absolute value u of b , computes $r = u \times EB2$ and applies the sign bit back to r .

Note that there is a threshold floating-point value $T = EB2 \times B$ beyond which the above quantization approach no longer uses all bin numbers, making the encoding inefficient. Hence we preserve those values losslessly, which not only increases accuracy but also compressibility. In Table II, where the threshold $T = 2.0$, we can clearly see that consecutive representable values above this threshold are more than ϵ apart. Hence, there is no point in quantizing such values, which is why we only apply quantization to a value v iff $|v| < 2.0$.

In our example, the bottom 32 floats in the table satisfy this criterion and, therefore, we quantize those values. Columns 4, 5, and 6 of the corresponding 32 rows show the resulting bin numbers (aka the encoded values), the results of the dequantization (obtained by multiplying the quantized value by $EB2$ and rounding the result), and the introduced errors, $abs(v - r)$. We note that the errors are all $\leq \epsilon = 0.25$.

To losslessly preserve values that cannot be quantized within the error bound, prior work keeps the original float value [6], [20]. Doing so has a major drawback — The corresponding values have bit patterns that are quite dissimilar to those of bin numbers and tend to have few leading zeros, which hurts compressibility. As mentioned, we avoid this problem by rounding the user-provided error bound down to the nearest power of two as multiplying and dividing by a power of two amounts to just adding and subtracting a value from the exponent, which yields a precise result as long as there is no over- or underflow. To avoid the over-/underflow issue, we exploit the fact that all quantized values fall into a relatively small range. After all, the maximum bin number used by the values we quantize is only B (ignoring the sign bit).

If we interpret the bit pattern representing the threshold T as an integer, $threshold_i$, we can see that the range of bin values between B and $threshold_i$ is unused. SLEEK closes this gap by losslessly mapping each value u that is not quantized to the value $u_i - offset$, where u_i is the integer interpretation of the bit pattern representing the absolute value of u , and $offset = threshold_i - B$, i.e., the size of the aforementioned gap. In the end, we again store the sign bit in the LSB position after shifting all result bits to the left by one position. Together, this encoding lowers the magnitude of u_i , thus creating more leading zeros without introducing any loss.

In our example, threshold $T = 2.0$, $threshold_i = 32$, and $offset = 32 - 4 = 28$. The top 32 rows in Table II have floats with value ≥ 2.0 . Hence, we subtract 28 from their integer representation and shift in the sign bit to obtain the green encoded values. During decompression, if the encoded value

after shifting out the sign bit is $\geq B$, we simply add $offset$ to it, which recreates the original value losslessly. The top 32 rows of Column 5 show the corresponding decoded values. As Column 6 indicates, all of these reconstructed values have zero error, including infinity and the NaNs.

2) *Quantization algorithm*: Algorithm 2 sketches the resulting quantization process for 32-bit single-precision values, where m is 23 (the number of bits in the mantissa field), eb_e is the biased exponent of the error bound ϵ , and thr_e is the biased exponent of the threshold T . For 64-bit double-precision values, some constants and the type of some variables need to be changed. Note that the algorithm does not use any floating-point operations. It achieves the desired effect by applying appropriate integer arithmetic and bit-wise operations on the sign, exponent, and mantissa fields of the bit pattern representing the floating-point value.

Algorithm 2 Lossy SLEEK Quantizer (of unsigned int val)

```

1: int abs ← (val << 1) >> 1; ▷ absolute value (elim. sign)
2: int val_e ← abs >> m; ▷ extract exponent
3: int enc ← 0; ▷ default value
4: if val_e ≥ thr_e then ▷ at or above threshold
5:   enc ← abs − offset; ▷ lossless encoding
6: else ▷ lossy encoding
7:   int mant ← val & ((1 << m) − 1); ▷ extract mantissa
8:   int shift ← thr_e − val_e; ▷ bias cancels out
9:   mant ← mant | (1 << m); ▷ insert implicit 1
10:  mant ← mant + (1 << (shift − 1)); ▷ round
11:  enc ← mant >> shift; ▷ shift out unnecessary bits
12: enc ← concatenate(enc, ~sign); ▷ magnitude-sign
13: if enc ≠ 0 then
14:   enc ← enc − 1; ▷ map -0 to +0 and fill gap
15: return enc;

```

The algorithm includes an optimization not yet covered because Section III-B only discusses positive values. In particular, Line 14 maps all -0 values (bin number 1) to +0 (bin number 0). To close the resulting gap, we subtract 1 from all bin numbers greater than zero. Doing so flips the sign bit stored in the LSB, which is why we invert the sign on Line 12.

After this quantization step is done, which quantizes all values within the error bound without exception, groups of consecutive bin numbers are processed together to find the minimum number of bits required to store them. Then, they are packed by removing the common leading zeros as described for the lossless compressor in Section III-A.

C. Conventional quantization (not used in SLEEK)

For illustration, the rightmost 6 columns of Table II show what would happen if all floating-point values were quantized conventionally using our mini floating-point format. Each value, independent of its magnitude, is divided by 2 times the user-provided error bound. The parameter $EB2_Q$ is 0.625, which is the nearest representable float for 2×0.3 .

The first problem is overflow. For any value v , if the rounded result of $v/EB2_Q$ is larger than the largest representable

float, v is quantized to the bin number of infinity. Hence, upon dequantization, the corresponding decoded value becomes infinity. The last column in the table shows that this happens for 3 large but finite values, violating the error bound. SLEEK avoids overflow by quantizing large values losslessly.

The second problem is rounding errors. A floating-point value v that cannot be represented exactly is rounded to the next higher or lower representable value, depending on which value is closer. Table II shows 6 finite decoded values, none of which suffer from overflow, that also violate the error bound (red numbers in the rightmost column). For example, consider the value 14.0, which is quantized to 20.0, yielding bin number 90, and then dequantized to 12.0, which is off by more than 6 times the error bound. This is the result of rounding errors in both the quantizer and the dequantizer. The quantizer needs to multiply 14.0 by 1.6 (the inverse of 0.625), but that is not a representable value, so it multiplies by the rounded value of 1.5 instead, which yields 21. This result is not representable, so it is rounded to 20. The dequantizer multiplies the floating-point interpretation of the corresponding bin number by 0.625 ($EB2_Q$), yielding 12.5, which is rounded to 12.0. SLEEK does not suffer from rounding errors because it only multiplies and divides by powers of 2.

The third problem is that, for floating-point values above the threshold, quantization yields bin numbers that have gaps between them, which happens because the distance between representable values grows with the magnitude of the floats. For example, the bin numbers 70, 78, 86, 94, 102, etc. are never generated during quantization, resulting in inefficient use of the range of possible bin numbers. Recall that the bin numbers are all even because we consider only positive values in this example. Column 4 shows that SLEEK addresses this problem by using every bin number up to a maximum and not using any bin numbers above this maximum, thus maximizing the number of leading zero bits in the used bin numbers.

D. Parallelization and optimization

We implemented SLEEK in CUDA for GPUs and also in C++/OpenMP for CPUs. Both versions exclusively use integer operations to perform the compression and decompression. As a result, SLEEK guarantees bit-for-bit identical compressed and decompressed output on CPUs and GPUs.

Since neither our lossless nor our lossy compressors need any special processing for outliers, each value in the input can be transformed or quantized in parallel, making this part of the code embarrassingly parallel. SLEEK divides the data into 16 kB chunks. On the CPU, we make each thread process a chunk at a time. On the GPU, we assign each chunk to a separate thread block. The amount of work required to compress a chunk is data dependent. To alleviate any resulting load imbalance, we use a dynamic schedule — once the assigned chunk has been processed, the thread or thread block picks the next unprocessed chunk from the worklist.

Even if the majority of the transformed values in a chunk have many leading zeros, the presence of a single value with

few leading zeros will make the entire chunk less compressible. To alleviate this issue, we further divide each 16 kB chunk into 32 subchunks, each of size 512 bytes, and compress each subchunk separately. This way, the reduced compressibility caused by a value with few leading zeros will only affect a subchunk rather than the whole chunk, yielding better overall compression ratios. On the GPU, this division allows us to assign one warp per subchunk.

During compression, each thread or thread block stores the compressed size cs of its chunk and the total compressed size tes of all previous chunks in a global *carry* array. Note that tes is the prefix sum of cs and provides the starting position of where to write the compressed chunk. On the CPU, the prefix sum is updated with atomic operations. On the GPU, we use Merrill and Garland’s decoupled look-back technique [21] to quickly send the write position to the next thread block.

To increase parallelism, we independently compress each chunk into local buffers, output the compressed size as soon as it is available, finish the compression, and then wait for the carry information from the previous thread or thread block. Once this information is known, the compressed data from the buffer is copied to the output. Chunks that cannot be compressed are copied to the output verbatim to minimize any possible expansion of the “compressed” data. The decompressor recognizes this case if the compressed chunk size is the same as the original chunk size. The complete output consists of (1) a header that records the original uncompressed size, (2) a list of all compressed chunk sizes, and (3) the compressed chunks. Each compressed chunk starts with the number of leading zero bits in each subchunk, which is followed by the non-eliminated bits of each subchunk. To simplify and speed up the code, we pad the last chunk in the input with zeros so that there are no partial chunks. We remove this padding in the decompressor with the aid of the header information.

SLEEK incorporates several additional optimizations. In the GPU code, we minimize and coalesce memory accesses as much as possible and use shuffle instructions to exchange data between threads in a warp without accessing memory. Both the compressor and the decompressor read the input from the main (global) memory exactly once. Once all transformations have been applied, the final output is written to main (global) memory exactly once, reducing the number of main-memory accesses. The GPU code keeps almost all intermediate data of each thread block in the fast *shared memory* (a software managed L1 data cache). The CPU code keeps most of the intermediate data in two 16 kB buffers. They are accessed alternately leading to good L1 data-cache presence.

IV. EXPERIMENTAL METHODOLOGY

We compare the throughput of SLEEK’s lossless and lossy compressors to that of the global memory (`cudaMemcpy-DeviceToDevice`) on the three systems detailed in Table III. We compiled the GPU codes using the “-O3 -arch=sm_89”, “-O3 -arch=sm_86” and “-O3 -arch=sm_80” flags for the RTX 4090, RTX 3080, and A100 GPUs, respectively. We ran each experiment 9 times and use the median runtime. The file

TABLE III: Details of the systems used for experiments

	System 1	System 2	System 3
CPU	Threadripper 2950X	Xeon Gold 6226R	Xeon Gold 6226R
Clock Frequency	3.5 GHz	2.9 GHz	2.9 GHz
Sockets	1	2	2
Cores Per Socket	16	16	16
Threads Per Core	2	2	2
Main memory	48 GB	128 GB	64 GB
GPU	RTX 4090	RTX 3080	A100
Compute Capability	8.9	8.6	8.0
Base Clock	2.2 GHz	1.44 GHz	0.8 GHz
Boost Clock	2.5 GHz	1.71 GHz	1.4 GHz
SMs	128	80	108
CUDA Cores per SM	128	128	64
Global memory	24 GB HBM2e	12 GB GDDR6X	40 GB GDDR6X
Peak mem. bandwidth	1008 GB/s	760 GB/s	1555 GB/s
Operating System	Fedora 41	Fedora 41	Fedora 37
g++ Version	14.2.1	14.2.1	12.3.1
nvcc Version	12.6	12.6	12.0

reading/writing and result verification are not included, nor is the time to copy the data to and from the GPU.

Table IV lists the details of the 7 single- and 3 double-precision floating-point input suites (with 89 files in total) we used in our experiments. These inputs, which represent real-world scientific datasets from various domains, stem from the SDRBench repository for compression evaluation [18], [22].

Distinct datasets may contain values that span hugely different ranges, rendering the use of a uniform absolute error bound inappropriate. To alleviate this problem, users sometimes apply a *Normalized Absolute (NOA)* error bound [16]. In this case, the user defines a single error bound ϵ that is normalized to $\epsilon_{noa} = |\epsilon \times (v_{max} - v_{min})|$ for a dataset with values in the range $[v_{min}, v_{max}]$. For each original value v , the reconstructed value r is now considered acceptable if $v - \epsilon_{noa} \leq r \leq v + \epsilon_{noa}$. We pre-computed the range of values in each input and use NOA error bounds.

We report the performance as throughput (the uncompressed input size divided by the runtime). The compression ratio (CR) is computed by dividing the original file size by the compressed file size. Note that CR and throughput are higher-is-better metrics. For each input suite, we compute the geometric-mean throughput and CR and also report the overall geometric-mean of these geometric means. Using the geometric instead of the arithmetic mean helps reduce the influence of inputs that perform exceptionally well compared to the overall trend [23].

The comparison to *memcpy* throughputs are shown using bar plots. The y-axis uses a linear scale in all of these plots. The term D2D stands for “device to device”. Legend entries of the form “1E-n” in the bar plots denote the error bounds.

TABLE IV: Information about the input suites (size is per file)

Name	Description	Format	Files	Dimensions	Size (MB)
CESM-ATM	Climate	Single	33	26 × 1800 × 3600	674
EXAALT Copper	Molecular Dyn.	Single	6	Various 2D	68 to 358
Hurricane Isabel	Weather Sim.	Single	13	100 × 500 × 500	100
HACC	Cosmology	Single	6	280,953,867	1124
NYX	Cosmology	Single	6	512 × 512 × 512	537
SCALE	Climate	Single	12	98 × 1200 × 1200	564
QMCPACK	Quantum MC	Single	2	33,120 × 69 × 69	631
NWChem	Molecular Dyn.	Double	1	102,953,248	824
Miranda	Hydrodynamics	Double	7	256 × 384 × 384	302
Brown Samples	Synthetic	Double	3	33,554,433	268

We also compare SLEEK to state-of-the-art lossy GPU compressors, which are summarized in Section VI. These results are from System 1. We use x/y-scatter plots to show the compression ratio and either the compression or decompression throughput. For space reasons, we omit CPU results.

V. RESULTS

In this section, we evaluate the performance of SLEEK’s GPU compressors and decompressors. First, we compare them to *cudaMemcpyD2D* in terms of throughput on three GPUs from different generations. Then, we evaluate their compression ratios. Finally, we compare SLEEK to state-of-the-art lossy GPU compressors in terms of both throughput and compression ratio. Note that *cudaMemcpyD2D* refers to copying the original uncompressed data from GPU memory to GPU memory, i.e., the data never leaves the GPU.

A. Lossy compression

This section discusses the throughput of SLEEK’s lossy compressor on single- and double-precision inputs as we vary the NOA error bound from 1E-1 to 1E-6.

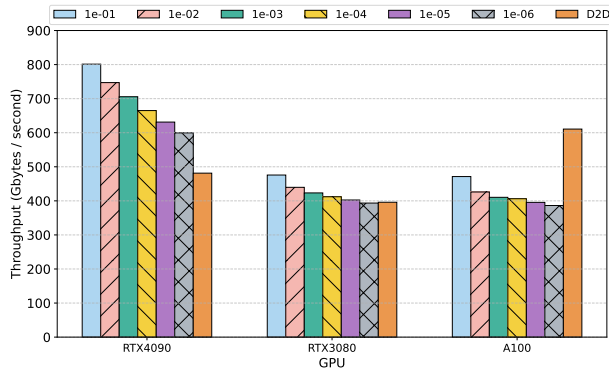
1) *Single-precision inputs*: Figure 1 compares the compression and decompression throughput on single-precision inputs against that of copying the uncompressed data using *cudaMemcpyD2D*. The legend indicates the error bound.

On the RTX 4090, SLEEK’s compressor is faster by 1.25× to 1.66× depending on the error bound. The highest performance is observed for the largest tested error bound. As larger error bounds yield more compressibility, the size of the data written after compression is lower, explaining the higher performance. Similarly, decompression reads less data, which is why it is 1.18× to 1.57× faster than *cudaMemcpyD2D*.

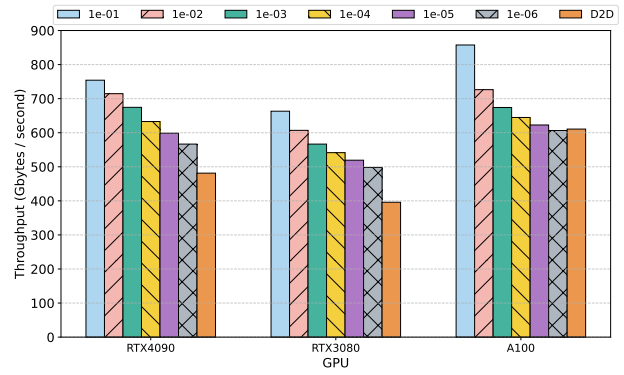
On the RTX 3080, the trends are similar. Compared to *cudaMemcpyD2D*, SLEEK’s compressor is 0.99× to 1.20× faster on compression and 1.26× to 1.67× faster on decompression. Note that the absolute performance of SLEEK as well as that of *cudaMemcpyD2D* is lower than on the RTX 4090. Interestingly, the compression throughput is markedly lower than the decompression throughput on the RTX 3080 (and the A100). This is primarily due to the aforementioned carry propagation (see Section III-D), which is only needed in the compressor. The RTX 4090 is able to hide the carry-propagation latency, but the RTX 3080 and A100 are not.

On the A100, the tested device with the highest memory bandwidth but the lowest compute-to-bandwidth ratio, SLEEK’s compressor is slower than *cudaMemcpyD2D* as indicated by the speedups that are below 1.0 (from 0.63× to 0.77×). In contrast, the decompressor is between 0.99× and 1.40× faster than *cudaMemcpyD2D* for all evaluated error bounds. Note that the performance of *cudaMemcpyD2D* is higher on the A100 than on the other two GPUs.

2) *Double-precision inputs*: Figure 2 shows a similar comparison for double-precision inputs. Whereas the trends are the same as for the single-precision inputs, SLEEK yields higher speedups on the double-precision inputs because they compress more (see below). On the RTX 4090, the compressor

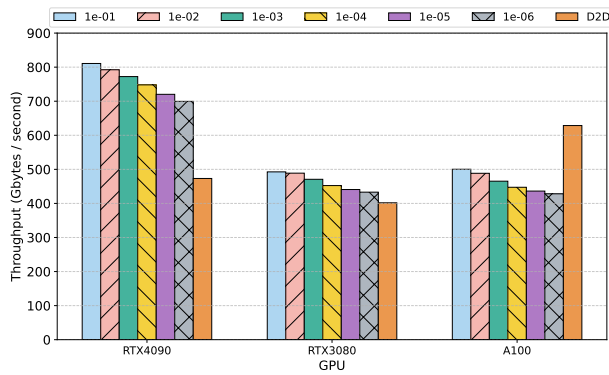


(a) Compression throughput

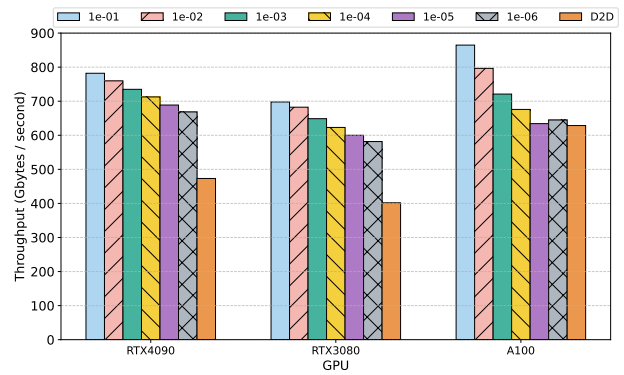


(b) Decompression throughput

Fig. 1: Throughput comparison of SLEEK’s lossy compressor on single-precision floating-point data against CudaMemcpyD2D on three different GPUs. The legend entries “1e-n” stand for $\epsilon = 10^{-n}$



(a) Compression throughputs



(b) Decompression throughputs

Fig. 2: Throughput comparison of SLEEK’s lossy compressor on double-precision floating point data against CudaMemcpyD2D on three different GPUs. The legend entries “1e-n” stand for $\epsilon = 10^{-n}$

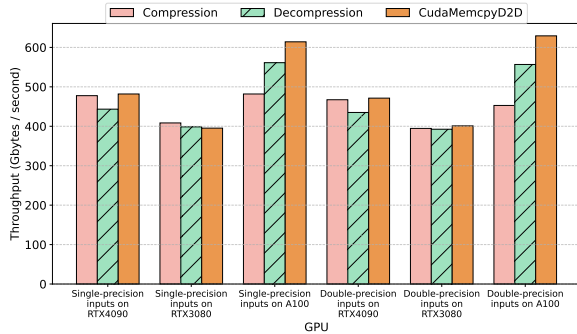


Fig. 3: Throughput comparison of SLEEK’s lossless compressor against CudaMemcpyD2D on three different GPUs.

is $1.48\times$ to $1.71\times$ faster and the decompressor is $1.41\times$ to $1.65\times$ faster than `cudaMemcpyD2D`. On the RTX 3080, the compressor is faster by $1.08\times$ to $1.23\times$ and the decompressor is faster by $1.45\times$ to $1.74\times$. On the A100, the compressor is still slower as indicated by the speedups being between $0.68\times$ and $0.80\times$, but the decompressor is faster by $1.01\times$ to $1.38\times$.

Again, the lower performance is expected since the A100 has a very high memory bandwidth but relatively low computation throughput. Consequently, the compression overhead cannot be hidden as well behind the memory accesses on this GPU.

B. Lossless compression

Figure 3 compares the compression and decompression throughputs of SLEEK’s lossless compressor to the throughput of `cudaMemcpyD2D`. The first three groups of bars show the results for the single-precision data, and the remaining three groups show the same for the double-precision data.

SLEEK performs on par with `cudaMemcpyD2D` on the RTX 3080. On the RTX 4090, the compression throughput matches that of `cudaMemcpyD2D` but decompression is a few percent slower. On the A100, which has the lowest compute-to-bandwidth ratio of the 3 tested GPUs, SLEEK’s compressor reaches 79% and 72% of the throughput of `cudaMemcpyD2D` on the single- and double-precision inputs, respectively. The decompressor is again faster (due to the lack of carry propagation) and yields over 90% of the throughput of `cudaMemcpyD2D` on the single- and double-precision inputs.

C. Compression ratios

Figure 4 shows the compression ratios of SLEEK’s lossless and lossy compressors. The lossy compressor is run with six NOA error bounds. The first group of bars shows the compression ratios for single-precision inputs and the second group for double-precision inputs. Each bar shows the geometric-mean compression ratio across all relevant input suites.

The compression ratios of the lossy compressor lie between 2.5 and 49.8 for the single-precision inputs and between 5.6 and 98.9 for the double-precision inputs, with the lowest value corresponding to the tightest error bound of $1E-6$ and the highest value corresponding to an error bound of $1E-1$. Double-precision inputs tend to compress more than single-precision inputs because more bits (i.e., more information) can be dropped from 64-bit doubles than from 32-bit floats to reach the same error bound. The lossless compressor’s ratios are lower than those of the lossy compressor — 1.18 for single- and 1.09 for double-precision inputs, respectively.

This behavior is expected. As we tighten the error bound and ultimately allow no error in the lossless compressor, the amount of information that must be preserved steadily increases. This makes it progressively harder to compress the data, explaining why the compression ratios go down.

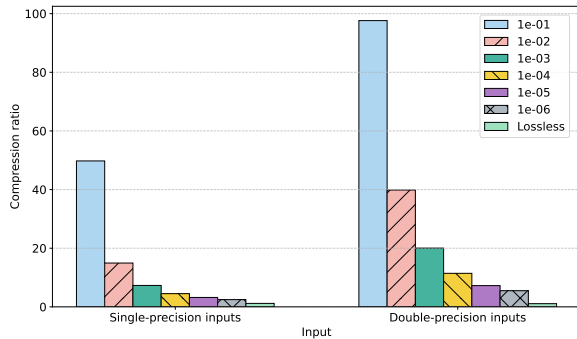


Fig. 4: Compression ratios of SLEEK’s lossless and lossy compressor using various normalized absolute error bounds

D. Comparison with other compressors

Figure 5 compares the lossy compressor of SLEEK to five state-of-the-art GPU compressors on System 1 using the single-precision inputs and the four NOA error bounds $1E-1$, $1E-2$, $1E-3$, and $1E-4$, which are represented by circles, triangles, squares, and pentagons, respectively. The x-axis shows the throughputs and the y-axis the compression ratios. Note that both axes use a logarithmic scale. We do not show FZ-GPU results for the $1E-3$ and $1E-4$ error bounds as it crashes on some of the inputs.

SLEEK yields much higher throughputs than the other compressors for every error bound used. When compressing with an error bound of $1E-1$, it is between $2.71\times$ and $64\times$ faster. At the tightest measured error bound of $1E-4$, it is between $2.44\times$ and $56.9\times$ faster. When decompressing with an error bound of $1E-1$, it is $1.87\times$ to $167\times$ faster than the

other compressors. With an error bound of $1E-4$, SLEEK is $1.69\times$ to $143\times$ faster than the other compressors.

Even though SLEEK is much faster, it delivers competitive compression ratios. Only PFPL compresses substantially more. For all other evaluated compressors, SLEEK’s compression ratios are on-par or higher at all tested error bounds.

Figure 6 compares SLEEK to the other GPU compressors using double-precision inputs, which FZ-GPU does not support. SLEEK’s compressor is faster than the other codes for every tested error bound. With an error bound of $1E-1$, it is between $2.95\times$ and $1011\times$ faster. For an error bound of $1E-4$, it is between $3.15\times$ and $937\times$ faster. When decompressing with the largest error bound of $1E-1$, SLEEK is between $1.13\times$ and $116\times$ faster. For the tightest error bound of $1E-4$, it is between $1.76\times$ and $114\times$ faster.

VI. RELATED WORK

Most of the research on main-memory compression focuses on hardware techniques for CPUs [24]–[29]. Many of them assume that compressed pages are of different sizes, and they perform page re-allocations as the amount of compressibility changes [24], [27]. They are not suitable for GPUs since GPUs operate at much higher memory bandwidths [30].

GPU hardware-based solutions such as selective memory compression [2] and buddy compression [4] have also been proposed, but they either merely handle read-only pages or employ a larger-but-slower buddy-memory connected with a high-bandwidth interconnect. These works primarily study the architectural changes required for hardware compression [31]. They cannot be directly used in current systems as they need special hardware support [32]. Moreover, they do not target floating-point data. While a few works on domain-specific compression for GPUs exist [33], [34], general-purpose compression of floating-point data remains largely unexplored [4].

Some other works target improving the host-to-device transfers [35]–[37] by overlapping computation and data transfer for heterogeneous computing, often guided by knowledge about the application, hardware, and software. They do not address main-memory compression.

The CPU-based LIGRA+ framework [1] supports software-based main-memory compression for read-only graph data structures. MPLG [5] is the only software-based main-memory compression technique proposed for GPUs that we are aware of, but it also targets read-only graph data structures and, hence, employs compression techniques specific to integers. In contrast, SLEEK is a software-based GPU main-memory compression technique for floating-point data that works on current systems and supports both reading and writing.

A few software-based lossless floating-point compressors for GPUs exist [38]–[40]. They are all slower than memcyp speeds. There are also several well-known software-based lossy floating-point data compressors for GPUs such as FZ-GPU [41], cuSZp [42], MGARD-X [43], Bitcomp [44], and PFPL [6]. Of these, only PFPL and SLEEK guarantee the error bound across all categories of values [6], [16]. Moreover, only MGARD-X, PFPL, and SLEEK support CPUs and GPUs. We

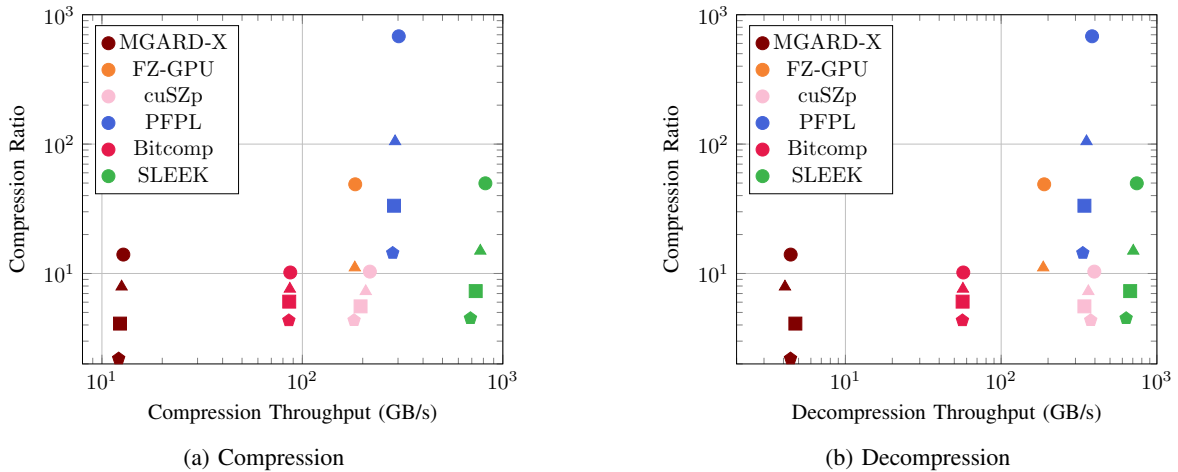


Fig. 5: RTX 4090 geometric-mean compression ratio and throughput on single-precision data for 4 error bounds

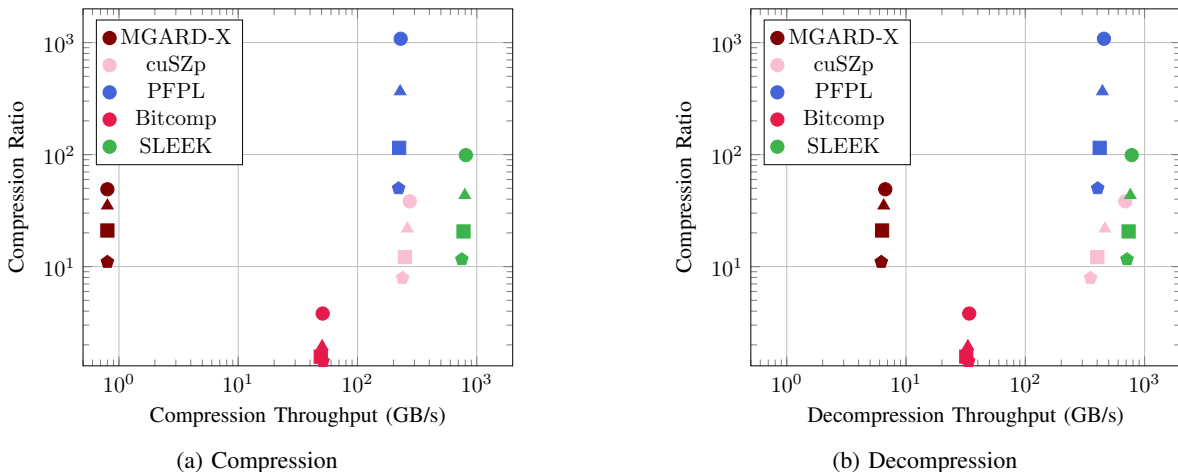


Fig. 6: RTX 4090 geometric-mean compression ratio and throughput on double-precision data for 4 error bounds

compare SLEEK’s throughput and compression ratio to these 5 compressors. Whereas some of them compress more, none of them both compress and decompress at speeds comparable to global memory accesses like SLEEK does.

VII. SUMMARY

The limited global memory capacity in GPUs is a major challenge for many scientific workloads that process large amounts of floating-point data. One potential solution is to keep the data in compressed format in memory, and decompress and recompress it on the fly as required. The effectiveness of on-the-fly compression (also known as main-memory compression) depends heavily on the speed of the compression and decompression algorithms. If they are not sufficiently fast, they can negate any benefits of the reduced memory footprint and introduce a performance bottleneck.

In this paper, we present SLEEK, a software-based main-memory compressor for GPUs that operates at speeds comparable to and, in many cases, faster than making a memory copy of the uncompressed data while yielding substantial

compression ratios. SLEEK supports lossless and lossy compression with guaranteed point-wise absolute error bounds. It handles all single- and double-precision IEEE 754 floating-point values, including subnormals, infinities, and NaNs.

We demonstrate the effectiveness of SLEEK on three generations of GPUs. Compared to the throughput of cudaMemcpyD2D, SLEEK’s compressor is up to $1.71\times$ faster when compressing and up to $1.65\times$ faster when decompressing single-precision data on an RTX 4090 GPU. We further show that SLEEK’s lossy compressor is substantially faster than the state-of-the-art lossy GPU compressors. For error bounds varying from $1E-6$ to $1E-1$, it yields geometric-mean compression ratios between 2.5 and 49.8 on single-precision inputs and between 5.6 and 98.9 on double-precision inputs.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant #2403380 and by the Department of Energy, Office of Science, Office of Advanced Scientific Research (ASCR), under Award #DE-SC0022223.

REFERENCES

- [1] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with ligra+," in *2015 Data Compression Conference*. IEEE, 2015, pp. 403–412.
- [2] A. Nihaal and M. Mutyam, "Selective memory compression for gpu memory oversubscription management," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 189–198.
- [3] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A framework for memory oversubscription management in graphics processing units," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 49–63.
- [4] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, "Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 926–939.
- [5] N. Azami and M. Burtscher, "Compressed in-memory graphs for accelerating gpu-based analytics," in *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2022, pp. 32–40.
- [6] A. Fallin, N. Azami, S. Di, F. Cappelto, and M. Burtscher, "Fast and effective lossy compression on gpus and cpus with guaranteed error bounds," in *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025.
- [7] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2021, pp. 1–14.
- [8] M. S. Nobile, P. Cazzaniga, A. Tangherloni, and D. Besozzi, "Graphics processing units in bioinformatics, computational biology and systems biology," *Briefings in bioinformatics*, vol. 18, no. 5, pp. 870–885, 2017.
- [9] F. D. Witherden and A. Jameson, "Future directions in computational fluid dynamics," in *23rd AIAA Computational Fluid Dynamics Conference*, 2017, p. 3791.
- [10] C. W. Hopkins, S. Le Grand, R. C. Walker, and A. E. Roitberg, "Long-time-step molecular dynamics through hydrogen mass repartitioning," *Journal of chemical theory and computation*, vol. 11, no. 4, pp. 1864–1874, 2015.
- [11] M. Burtscher and K. Pingali, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," in *GPU computing Gems Emerald edition*. Elsevier, 2011, pp. 75–92.
- [12] N. Azami, A. Fallin, and M. Burtscher, "Efficient lossless compression of scientific floating-point data on cpus and gpus," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 395–409. [Online]. Available: <https://doi.org/10.1145/3669940.3707280>
- [13] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao, Z. Chen, and F. Cappelto, "SZ3: A Modular Framework for Composing Prediction-Based Error-Bounded Lossy Compressors," *IEEE Transactions on Big Data*, vol. 9, no. 2, pp. 485–498, 2023.
- [14] S. Di and F. Cappelto, "Fast Error-Bounded Lossy HPC Data Compression with SZ," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2016, pp. 730–739.
- [15] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.
- [16] A. Fallin and M. Burtscher, "Lessons learned on the path to guaranteeing the error bound in lossy quantizers," *arXiv preprint arXiv:2407.15037*, 2024.
- [17] D. Tao, S. Di, Z. Chen, and F. Cappelto, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 1129–1139.
- [18] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, and F. Cappelto, "SDRBench: Scientific Data Reduction Benchmark for Lossy Compressors," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 2716–2724.
- [19] J. Diffenderfer, A. L. Fox, J. A. Hittinger, G. Sanders, and P. G. Lindstrom, "Error analysis of zfp compression for floating-point data," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. A1867–A1898, 2019. [Online]. Available: <https://doi.org/10.1137/18M1168832>
- [20] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappelto, "Optimizing Error-Bounded Lossy Compression for Scientific Data by Dynamic Spline Interpolation," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1643–1654.
- [21] D. Merrill and M. Garland, "Single-pass parallel prefix scan with decoupled look-back," NVIDIA, Tech. Rep. NVR-2016-002, March 2016.
- [22] "SDRBench Inputs," <https://sdrbench.github.io/>, 2023. [Online]. Available: <https://sdrbench.github.io/>
- [23] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: the correct way to summarize benchmark results," *Commun. ACM*, vol. 29, no. 3, p. 218–221, Mar. 1986. [Online]. Available: <https://doi.org/10.1145/5666.5673>
- [24] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 74–85.
- [25] S. Mittal and J. S. Vetter, "A survey of architectural approaches for data compression in cache and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1524–1536, 2015.
- [26] M. Laghari, Y. Liu, G. Panwar, D. Bears, C. Jearls, R. Srinivas, E. Choukse, K. W. Cameron, A. R. Butt, and X. Jian, "Memory allocation under hardware compression," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 966–982.
- [27] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 172–184.
- [28] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," *HPCA*, 2014.
- [29] B. Abali, H. Franke, D. E. Poff, J. R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith, "Memory Expansion Technology (MXT): Software support and performance," *IJRD*, vol. 45, no. 2, pp. 287–301, 2001.
- [30] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 345–357.
- [31] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 325–334.
- [32] S. Kim, S. Lee, T. Kim, and J. Huh, "Transparent dual memory compression architecture," *PACT*, 2017.
- [33] S. Wang, S. Wang, W. Yang, X. Zhang, S. Wang, S. Ma, and W. Gao, "Towards analysis-friendly face representation with scalable feature and texture compression," *IEEE Transactions on Multimedia*, vol. 24, pp. 3169–3181, 2021.
- [34] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 776–789.
- [35] C. Margiolas and M. F. O'Boyle, "Portable and transparent host-device communication optimization for gpgpu environments," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 55–65.
- [36] J. Bhimani, M. Leeser, and N. Mi, "Design space exploration of gpu accelerated cluster systems for optimal data transfer using pcie bus," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–7.
- [37] D. Jeong, J. Park, and J. Kim, "Demand memcopy: overlapping of computation and data transfer for heterogeneous computing," *IEEE Access*, vol. 10, pp. 79 925–79 938, 2022.
- [38] N. Azami, A. Fallin, and M. Burtscher, "Efficient lossless compression of scientific floating-point data on cpus and gpus," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 395–409.
- [39] NVIDIA, "nvCOMP," <https://github.com/NVIDIA/nvcomp>, 2024, accessed: 2025-03-29.

- [40] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher, "Mpc: a massively parallel compression algorithm for scientific data," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 381–389.
- [41] B. Zhang, J. Tian, S. Di, X. Yu, Y. Feng, X. Liang, D. Tao, and F. Cappelto, "FZ-GPU: A Fast and High-Ratio Lossy Compressor for Scientific Computing Applications on GPUs," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://dl.acm.org/doi/10.1145/3588195.3592994>
- [42] Y. Huang, S. Di, X. Yu, G. Li, and F. Cappelto, "cuSZp: An Ultra-Fast GPU Error-Bounded Lossy Compression Framework with Optimized End-to-End Performance," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'23. Denver, CO, USA: Association for Computing Machinery, 2023.
- [43] J. Chen, L. Wan, X. Liang, B. Whitney, Q. Liu, D. Pugmire, N. Thompson, J. Y. Choi, M. Wolf, T. Munson, I. Foster, and S. Klasky, "Accelerating Multigrid-based Hierarchical Scientific Data Refactoring on GPUs," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 859–868.
- [44] NVIDIA, "nvcomp: Nvidia gpu data compression library," <https://github.com/NVIDIA/nvcomp>, accessed: 2025-08-29.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. OVERVIEW OF CONTRIBUTIONS AND ARTIFACTS

A. Paper's Main Contributions

- C_1 It introduces SLEEK, a new main-memory compression algorithm for single- and double-precision floating-point data that delivers speeds comparable to or exceeding the speed of *memcpy* on GPUs.
- C_2 It describes both lossless and lossy algorithms that yield substantial compression ratios at high speeds. They support all floating-point values, including NaNs, infinities, and denormals. The lossy compression algorithm guarantees point-wise absolute error bounds without the need to check for violations.
- C_3 It presents GPU and CPU implementations of SLEEK that produce compressed and decompressed outputs that are bit-for-bit identical, allowing data compressed on a GPU to be decompressed on a CPU and vice versa.
- C_4 It demonstrates the ability of SLEEK to deliver much higher throughput at a given compression ratio compared to the state-of-the-art algorithms from the literature.

B. Computational Artifacts

- A_1 This computational artifact includes scripts to run our SLEEK code on the SDRBench inputs.
URL: <https://doi.org/10.5281/zenodo.18705900>

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3	Figures 1 to 4

II. ARTIFACT IDENTIFICATION

A. Computational Artifact A_1

Relation To Contributions

Artifact A_1 provides codes implementing C_1 , C_2 , and C_3 , i.e., this artifact contains our GPU implementation of SLEEK along with relevant scripts to download the SDRBench inputs, compile and run our code, gather results, and generate figures similar to those shown in the paper. Furthermore, the artifact contains our CPU implementation of SLEEK and verifies that the compressed and decompressed outputs are bit-for-bit identical. It does this by comparing the compressed CPU output to the compressed GPU output and the decompressed CPU output to the decompressed GPU output.

Expected Results

After running the provided scripts, the generated figures should resemble Figures 1 to 4 in the paper. The compression ratios should exactly match those shown in the paper. The throughput trends should be similar to those in the paper but will not match exactly as these results are system dependent.

Expected Reproduction Time (in Minutes)

The expected time for setup is 30 minutes, depending on the download speed, including the time to clone the repository, the time to download and install the dependencies, and the time to download the SDRBench inputs.

The expected time for execution is 90 minutes when running on a system with an 11th Gen Intel Core i7-11700 CPU and an NVIDIA GeForce RTX 3060 Ti GPU. However, the execution time is system dependent.

Artifact Setup (incl. Inputs)

Hardware: We used several different CPUs and GPUs, which are listed in Table 3 in the paper. Using similar hardware should yield results similar to those shown in the paper. The minimum hardware requirement is a CUDA-capable GPU with a compute capability of 8.0 or higher. Lower compute capabilities have not been tested.

Software: To run our codes and generate the figures, the following codes need to be installed. Older versions of these codes may work but are untested.

Python version 3.13:

<https://www.python.org/downloads/release/python-3130/>

Matplotlib Python visualization library version 3.10.1:

<https://pypi.org/project/matplotlib/3.10.1/>

Our SLEEK implementation:

<https://github.com/burtscher/SLEEK/>

Datasets / Inputs: We used several datasets from the SDRBench suite. Our scripts automatically download the inputs used in the paper from this URL: <https://sdrbench.github.io/>. Furthermore, because the total size of the inputs is over 45 gigabytes, we provide an option to download a subset of the inputs we used.

Installation and Deployment: To compile our codes, the `nvcc` compiler and a `g++` compiler with OpenMP support are required.

NVIDIA CUDA compiler version 12.0:

<https://developer.nvidia.com/cuda-12-0-0-download-archive>

NVIDIA driver version 525.85:

<https://www.nvidia.com/en-us/drivers/details/198554/>

GNU Compiler Collection version 12.2:

<https://ftp.gnu.org/gnu/gcc/gcc-12.2/>

Artifact Execution

The following experiments produce results that mimic the results shown in the paper. There is one Bash script to download the SDRBench inputs, compile and execute our codes, and generate the figures. The script runs the lossless and lossy SLEEK codes with several different error bounds on the downloaded SDRBench inputs. Then, it performs a device-to-device `cudaMemcpy` with the inputs for comparison.

For the experiments shown in the paper, we ran each code 9 times and used the median runtime. To reduce the

time required to run the artifact, the script runs each code 3 times and reports the median runtime. Lastly, the script parses the compression ratio as well as the encoding and decoding throughputs and generates figures showing geometric-mean compression ratios and throughputs.

The workflow is as follows:

- T_1 Download our codes using the link to the repository provided above.
- T_2 Download and install the required dependencies listed above, if necessary.
- T_3 From the top-level directory in the repository, execute the following command to run the Bash script named `full-workflow.sh`.
- T_4 Analyze the 6 generated figures (PDF files).

The tasks must be completed in this order: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$. For task T_1 , execute the following commands to download the repository and change to the top-level directory.

```
git clone https://github.com/burtscher/SLEEK/  
cd SLEEK
```

From the top-level directory, to assist with task T_2 , execute the following command that checks if the required dependencies are installed, along with what versions are installed.

```
bash check-deps.sh
```

Task T_3 comprises the bulk of this artifact. There is a single script named `full-workflow.sh` that completes the following steps. 1) It downloads the single- and double-precision inputs from SDRBench. 2) It compiles and executes the SLEEK GPU codes on those inputs. 3) It verifies that the CPU and GPU codes output bit-for-bit the same compressed and decompressed file for the given inputs. 4) It generates figures showing the compression ratio, encoding throughput, and decoding throughput of the GPU codes. To run this script, execute the following command.

```
bash full-workflow.sh
```

Task T_4 requires opening the generated figures, stored in a directory named `figures`, with an installed PDF viewer or web browser and comparing to Figures 1–4 in the paper. The `xdg-open` command is a standard way to open files in the default application that suits the given file type. The following commands provide an example of opening the generated figures from the command line.

```
cd ./figures  
xdg-open Single-Precision.Compression.Ratios.pdf
```

Below is the full set of commands to complete all tasks.

```
git clone https://github.com/burtscher/SLEEK.git  
cd SLEEK  
bash check-deps.sh  
bash full-workflow.sh  
cd ./figures  
xdg-open Single-Precision.Compression.Ratios.pdf  
xdg-open Single-Precision.Compression.Throughputs.pdf  
xdg-open Single-Precision.Decompression.Throughputs.pdf  
xdg-open Double-Precision.Compression.Ratios.pdf  
xdg-open Double-Precision.Compression.Throughputs.pdf  
xdg-open Double-Precision.Decompression.Throughputs.pdf
```

Artifact Analysis (incl. Outputs)

To analyze the results, compare the generated figures to Figures 1 through 4 in the paper. The results are system dependent but should follow the general behavior and trends shown in the paper.

The 3 figures for both sets of inputs correspond to the following metrics: 1) compression ratio, 2) encoding throughput, and 3) decoding throughput. After the `full-workflow.sh` script has been executed, the `figures` directory contains six figures saved as PDF files, three for the single-precision results and three for the double-precision results.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

Links to all of the required dependencies are listed above in the Artifact Description. Each dependency has its own installation requirements, and the links lead directly to the version of the software used in our paper. Note that older versions of these codes may work but are untested.

Artifact Execution

After the codes and scripts have been downloaded and the required dependencies have been installed (task T_1 and T_2 above), task T_3 requires executing the script named `full-workflow.sh` that runs the experiment workflow.

This script first performs a safety check and runs the script named `check-deps.sh` to ensure nothing is executed without the required dependencies.

Next, the inputs from the SDRBench suite are downloaded. This is done by running two scripts named `download_inputs_single.py` and `download_inputs_double.py` to download the single- and double-precision inputs, respectively. These scripts download the tar files from <https://sdrbench.github.io/> for each dataset used in the paper, extracts the individual inputs, places them in a folder named `single_inputs` or `double_inputs`, and then deletes the archive file. In total, these inputs are over 45 gigabytes in size. A subset of the inputs can be downloaded instead, which totals to 5.4 gigabytes in size. By default, the full set of inputs is downloaded. To download the subset, set the flag inside of the `full-workflow.sh` script named `use_limited_input_set` to `true`.

Following the downloads, the SLEEK codes are compiled and executed on the inputs, starting with the single-precision inputs then the double-precision inputs.

Focusing on the single-precision inputs, the script named `compile_compressors_single.sh` is called first, compiling the GPU and CPU SLEEK codes. This script first finds the correct GPU architecture to use when compiling the SLEEK codes by compiling and running a code named `grabcc.cu`. The GPU codes are compiled with `nvcc` and the following flags: `-O3 -arch=sm_XX` where `sm_XX` is replaced with the correct architecture flag to use for the system's GPU. The CPU codes are compiled with `g++` and the following flags: `-O3 -fopenmp -march=native -std=c++17`.

After compiling, the script named `compress_decompress_inputs_single.py` is executed, which runs the GPU SLEEK codes on the single-precision inputs losslessly and lossily with several different error bounds. Each version of SLEEK is repeated 3 times for each input, and the median encoding and decoding throughput is recorded along with the compression ratio. In the paper, we run SLEEK 9 times and report the median. To change the number of iterations, modify the `ITERATIONS`

variable in the script. After all inputs in a dataset have been processed, the geometric-mean compression ratio, encoding throughputs, and decoding throughputs are recorded. Once all datasets have been processed, the geometric-mean of the geometric-mean metrics recorded earlier are saved to a file named `single_inputs.results` to be used when generating the figures later. This is a plain-text file and can be viewed with a text editor to analyze the exact results used to generate the figures.

Next, the script named `verify_cpu_gpu_single.py` is executed. This script verifies that the CPU and GPU SLEEK codes output bit-for-bit identical compressed and decompressed files for each single-precision input.

Lastly, the script named `generate-figures.py` is executed. This script generates 3 figures showing the compression ratio, encoding throughputs, and decoding throughputs of SLEEK at different error bounds, respectively. The throughput figures include CUDA device-to-device (D2D) memcpy results for comparison. These figures are placed in a directory named `figures`.

After these steps are completed, a similar process of compiling, running, verifying, and generating results is performed for the double-precision inputs, outputting the results in a file named `double_inputs.results` and saving 3 figures in the `figures` directory as before.

Artifact Analysis (incl. Outputs)

After the `full-workflow.sh` script has been executed, the `figures` directory contains six figures saved as PDF files, three for the single-precision results and three for the double-precision results. The figures are named `X.Compression.Ratios.pdf`, `X.Compression.Throughputs.pdf` and `X.Decompression.Throughputs.pdf` for the compression ratio, compression throughput, and decompression throughput results, respectively, where `X` is either `Single-Precision` or `Double-Precision`.

The compression ratio figure layout is as follows. The error bound runs along the x-axis, and the compression ratio achieved with that error bound runs along the y-axis. If using the full input set from SDRBench, the compression ratios should exactly match those shown in Figure 2 of the paper, as the compression ratio is system independent. When using a subset of the inputs, the trend should remain the same but the results will not match exactly. In general, a coarser error bound provides a higher compression ratio, with lossless compression providing the least amount of compression.

The compression and decompression throughput figure layouts are as follows. The throughput in gigabytes per second (GB/s) runs along the y-axis, and the code that achieved that throughput runs along the x-axis. The codes listed in these figures consist of lossless SLEEK, lossy SLEEK with 6 different error bounds, and device-to-device (D2D) CUDA memcpy. The hatching of each bar corresponds to the code type, indicated by the legend in the top right. These results will not exactly match the ones shown in the paper, as they

are system dependent. However, the general trends should be the same. The D2D memcpy copies data from one in-memory buffer to another. The speed will depend on the memory throughput of the GPU. SLEEK, in contrast, compresses the data as it is transferred to another in-memory buffer, therefore reducing the amount of data written after compression or read before decompression. Depending on the GPU, the compression and especially the decompression overheads can be hidden by the memory access latency, thus achieving higher throughput than CUDA memcpy.