FitFloat: Read/Write Random-Access Compressed Floating-Point Arrays for GPUs

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*—In many floating-point applications that use GPUs, device memory is a bottleneck limiting the maximum problem size. One solution is to use Unified Virtual Memory, but the limited physical GPU memory likely causes page evictions, incurring potentially large performance penalties. We present FitFloat, a drop-in floating-point array replacement supporting user-specified precision on GPUs with the goal of reducing storage requirements of scientific applications while maximizing performance over Unified Memory. FitFloat targets applications limited by the GPU's memory capacity, allowing larger problem sizes to be efficiently processed. We evaluate the storage savings and performance improvement of our approach relative to using native arrays on 11 single-precision and 3 double-precision codes from the HeCBench suite on 2 different GPUs. In the best case, FitFloat yields a speedup of 739 and provides a geometric-mean speedup of 16.1 for the float codes and 20.1 for the double codes on an RTX 4090.**

***Index Terms*—Main-memory compression, Floating-point data, Flexible precision, GPU/CPU parallelization**

I. Introduction

Application domains such as scientific computing tend to process large amounts of floating-point data and often employ GPUs for their high performance. However, a major bottleneck is the GPU memory, which is typically much smaller than the memory of the CPU in the system. Moreover, GPU memory is fixed and cannot be extended. This limits the problem sizes that fit on the device and complicates porting codes to systems with different device memory sizes, preventing certain applications from fully exploiting the performance of GPUs.

One solution is to divide the problem into batches and run each batch one at time [1]. However, doing so is only possible for some codes, and transferring data multiple times to and from the GPU is slow. Another solution is to use less precise floating-point types, allowing for more values to fit on the GPU. However, many applications cannot afford the high loss in precision incurred when switching to the next smaller hardware type, for example when going from double to single precision, even though the larger type may provide more than enough precision. Additionally, not all GPUs support 16-bit or smaller floating-point types. Hence, applications designed for such data types are not portable to older GPUs.

Another solution is to use Unified Memory, a single address space accessible from the CPU and the GPU, effectively increasing the GPU's memory capacity [2]. When accessing a word in unified memory on the device, the system automatically migrates the page if needed. Although the programmer can help by prefetching or pinning pages, the limited physical GPU memory likely causes page evictions, incurring potentially large performance penalties.

Lossy data compression is a popular solution for the storage of scientific data, but almost no approaches exist that support compression of floating-point arrays while allowing random access to individual elements. Whereas ZFP [3] provides this feature, it only does so on CPUs. There are currently no compression techniques allowing for read and write random access of floating-point arrays on GPUs.

In this paper, we present FitFloat, a drop-in floating-point array replacement supporting user-specified precision on GPUs with the goal of reducing storage requirements of scientific applications while maximizing performance over Unified Memory solutions. FitFloat targets applications limited by the GPU's memory capacity, allowing larger problem sizes to be efficiently processed. More specifically, FitFloat allows up to almost twice the number of array elements to be stored in global memory, while often making processing much faster compared to native arrays, even when the entirety of the FitFloat array does not fit in global memory.

FitFloat is a compressed in-memory storage format allowing the user to select the number of exponent bits and, separately, the number of mantissa bits used to represent the floating-point values in an array, providing fine-grained control. With this flexibility, FitFloat makes it possible to select the *lowest acceptable precision for each array*, providing users the option to reduce the size of the exponent, the mantissa, or both. FitFloat and native arrays can be used in the same code, meaning FitFloat arrays can be incrementally introduced. Moreover, each FitFloat array can be configured independently.

Accesses to a FitFloat array are always performed with the native float or double type. FitFloat transparently deconstructs the native type to lossily fit the user specification when writing and reconstructs the value back to the native type when reading, requiring minimal changes to the source code. From the programmer's perspective, all memory accesses to the FitFloat array use the full-width floating-point type and no low-precision hardware or operations are required.

Comparing FitFloat to native arrays using Unified Memory

SC26, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-4789-7/26/$31.00

on the 14 HeCBench codes we use for evaluation, FitFloat provides a geometric-mean speedup of 16.1 for the float codes and 20.1 for the double codes on an RTX 4090 and a geometric-mean speedup of 8.2 for the float codes and 8.6 for the double codes on an A100 for the smallest tested configuration. In the best case, it yields a speedup of up to 739.0 on the 4090 and 366.4 on the A100. Moreover, it provides substantial speedups on all 14 codes while also reducing the memory footprint of these applications. In the worst case, FitFloat still provides high speedups for all tested codes on the 4090 and for all but two codes on the A100.

This paper makes the following main contributions.

- It proposes a novel approach for user-specified-precision floating-point arrays on GPUs that are easy to use and support both float and double as the backing data type.
- It describes our high-performance implementation to encode and decode the reduced-precision values, utilizing GPU-specific optimizations to improve the performance, allowing for read and write random access to individual elements in the compressed array.
- It presents a performance comparison of FitFloat arrays and native floating-point arrays allocated in Unified Memory on 14 codes from the HeCBench suite running on 2 GPUs from different generations.

The latest version of our FitFloat implementation and our computational artifact are available in open source on GitHub [4] and on Zenodo [5]. The rest of the paper is organized as follows. Section II discusses related work. Section III explains our implementation in detail. Section IV summarizes the experimental methodology. Section V presents and analyzes the results. Section VI concludes the paper.

II. Related Work

In this section, we discuss related work on utilizing reduced precision to improve performance, reduce storage, or save energy. Then, we discuss existing approaches that provide reduced-precision types and how our approach differs.

A. Applications Utilizing Reduced Precision

Mixed precision describes techniques that combine different floating-point types in the same code for performance or storage reasons [6], [7], [8]. Less precise types are used as much as possible, and more precise types are reserved for critical operations. Transprecision [9] describes similar techniques but focuses on energy savings. Precision tuning [10], [11] describes the process of changing the data types in an application to effectively utilize mixed precision and transprecision.

Many prior works propose frameworks and tools to assist with incorporating mixed precision into existing applications by analyzing binaries dynamically or modifying the source code [12], [13], [14]. Other approaches use machine learning or exploit characteristics specific to certain algorithms to achieve mixed precision [15], [16], [17], [18], [19].

Micikevicius et al. [20] introduce a methodology for training neural networks (NNs) using mixed precision without losing model accuracy. With the introduction of half-precision types in NVIDIA GPUs, Ho and Wong [21] present an automated framework to assist users in migrating their codes to better exploit the smaller types. Ioualalen and Martel [22] introduce a technique to tune the precision of NNs in such a way that the network operates at lower precision without modifying the quality by more than a user-specified percentage. Haidar et al. [23] present a redesign of a mixed-precision iterative refinement technique to harness the half-precision type on newer GPUs. Grützmacher et al. [24] propose FRSZ2 to alleviate the GPU-memory-bandwidth bottleneck in GMRES solvers, providing a speedup compared to uncompressed methods and other compressed methods, without losing accuracy.

Molahosseini et al. [25] present Software-Defined Floating-Point (SDF) formats specifically for belief propagation on CPUs. The goal is to provide the same accuracy as native floating-point types while reducing the memory footprint and bandwidth. Vandierendonck [26] proposes a customized floating-point format for CPUs that makes assumptions about the floating-point values in graph algorithms to reduce their width. They are converted to native hardware types as needed.

Clearly, there is a strong desire for reducing precision in floating-point workloads. Next, we discuss related work that provides non-standard precision for applications.

B. Non-standard Precision Approaches

There are several software-based floating-point libraries that offer precision not provided by hardware types, including the GNU Multiple Precision Arithmetic Library [27], the Boost Multiprecision Library [28], Berkeley SoftFloat [29], ARPREC [30], and MPFR [31]. Their main goal is extending rather than reducing precision.

Tagliavini et al. [32] introduce FlexFloat, a C/C++ library for transprecision applications. A FlexFloat is a floating-point type with a user-specified number of bits in the mantissa and the exponent. The library picks the backing floating-point type used for arithmetic operations based on the available hardware types, rounding up to the nearest type that provides a sufficient number of bits. When assigning a value to a FlexFloat type, the value is *sanitized*, meaning the exponent is rounded to fit the user-specified number of exponent bits, and the mantissa is truncated to the user-specified number of mantissa bits. Despite this flexibility, the end goal is to settle on the native floating-point type that is best suited for the user's needs. The sanitizing process is used only during precision tuning, and the backing hardware type is no longer sanitized once finalized. In contrast, our approach stores each floating-point value in the user-specified precision, thus saving more memory.

Anderson and Gregg [33] introduce a set of non-standard floating-point multibyte types for CPUs they call *flytes*. Their approach truncates trailing mantissa bits, stores the data into byte-multiple containers, and works as substitutions for IEEE float and double types. For example, *flyte24* is a float with the lower 8 bits of the mantissa truncated, thus requiring 24 bits in memory. Importantly, these formats do not modify the exponent field. This reduced-precision format is intended for data storage only, and a flyte is reconstructed into a standard

float or double type for arithmetic operations by padding the truncated mantissa with zeros. Mukunoki and Imamura [34] extend this work to GPUs. Although these approaches share a similar goal, our FitFloats provide two important advantages: they are not limited to sizes that are multiples of 8 bits and they can separately reduce the exponent and mantissa sizes.

Flegar et al. [35] propose FloatX, a C++ framework for customized floating-point arithmetic on CPUs. FloatX allows for user-specified mantissa and exponent bit counts smaller or equal to the IEEE double type. Their framework has three goals. First, FloatX types should be an extension of the standard double type and provide the same semantics whenever possible. Second, FloatX types should be interoperable with built-in numeric types. Third, FloatX types should not require more space than a double. The authors integrated FloatX with several libraries such as BLAS, LAPACK, and Ginkgo. In contrast, our work aims to reduce the storage requirements of GPU floating-point applications, provides more flexibility by allowing reduced-precision of both IEEE float and double types, and, most importantly, can be integrated into CUDA programs with little modification to the original source code.

Lindstrom et al. [3] describe ZFP, a numerical format for compressed in-memory multidimensional arrays on CPUs. The user can specify how many bits of compressed storage to allocate to each element in the array or can set an absolute error tolerance, which results in a variable number of bits per block. Either way, the actual size of the arrays depends on how well the data can be compressed and may not match the user specifications. These arrays support random access but, due to the block-based compression scheme, accessing one element compresses or decompresses an entire block. Moreover, ZFP arrays use operator overloading to simplify integration into existing codes. In contrast to ZFP, our FitFloat approach guarantees the user-specified size for each array element, compresses and decompress each element without accessing other elements, and targets GPUs.

TABLE I: Summary of the state of the art

Name	Hardware	User-Precision Flexibility	Types
FitFloat	GPU+CPU	Mantissa and exponent separately	Float, Double
FlexFloat	CPU	Only hardware-supported types	N/A
Flytes	GPU+CPU	Only mantissa, only byte granularity	Float, Double
FloatX	CPU	Mantissa and exponent separately	Double
ZFP Arrays	CPU	Bits per element	Float, Double

Table I summarizes the related works and how they compare to FitFloat. It describes if they run on the CPU or GPU, how flexible they are at meeting the user-specified precision, and what backing types are supported. These results highlight the uniqueness of FitFloat and the key advantages it provides.

III. FitFloat Approach

A FitFloat array is defined by the number of floating-point elements in the array, the desired backing type (float or double), the number of exponent bits, and the number of mantissa bits. The user can choose to reduce only the mantissa, only the exponent, or both fields relative to the backing type. To save space, each FitFloat element is stored contiguously in memory, overlapping word boundaries when necessary. Note that FitFloat never generates misaligned memory accesses.

FitFloat arrays support all floating-point values, including subnormals, NaNs, and infinities. They are accessed with the random-access operator (`[]`) just like native arrays. The only restrictions are that they do not support atomic accesses nor pointer arithmetic as the FitFloat type is not a pointer but rather a struct holding the backing data pointer, the number of elements, and the actual size in memory. However, there are many programs that access arrays without using atomic operations or pointers into the middle of the array. The struct takes up 25 bytes, which is larger than a pointer but negligible when considering the space savings of the entire array.

```
/*************** Native float code ***************/

__global__
void vadd(float* a, float* b, float* c, long N) {
  long idx = /*...*/;
  if (idx < N) {
    c[idx] = a[idx] + b[idx];
  }
}

float *h_a, *h_b, *h_c;
float *d_a, *d_b, *d_c;
/* Allocate on host and device and init on host */

cudaMemcpy(d_a, h_a, ..., cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, ..., cudaMemcpyHostToDevice);
vec_add<<<...>>>(d_a, d_b, d_c, ...);

/***************** FitFloat code *****************/

#include "FitFloat.h"

// FitFloats with 5 exponent and 20 mantissa bits
typedef FitFloatArray<float, 5, 20> FFArr32;

__global__
void vadd(FFArr32 a, FFArr32 b, FFArr32 c, long N) {
  long idx = /*...*/;
  if (idx < N) {
    c[idx] = a[idx] + b[idx];
  }
}

float *h_a, *h_b, *h_c;
FFArr32 d_a, d_b, d_c;
/* Allocate on host and device and init on host */

FloatH2FFDmemcpy(d_a, h_a, ...);
FloatH2FFDmemcpy(d_b, h_b, ...);
vec_add<<<...>>>(d_a, d_b, d_c, ...);
```

Listing 1: Example of augmenting CUDA code with FitFloat

A. Using FitFloat

We provide host and device code to assist with the integration of FitFloat arrays into source code, including customized memcpy functions for transferring native host arrays to device FitFloat arrays and device FitFloat arrays to native host arrays. Note that FitFloat arrays can be allocated traditionally (with CUDA `malloc`) or with Unified Memory, depending on what the application requires.

Our code is available in one CUDA header file. It consists of two classes, one containing a device FitFloat implementation

and the other containing a host FitFloat implementation, as well as some helper host and device functions. Integrating FitFloat into existing code requires including the header file, modifying the type of the targeted floating-point arrays in kernel declarations, and replacing any host-to-device and device-to-host memcpy calls with our provided memcpy functions. No other changes are needed. In particular, the bodies of the kernels that access the arrays remain unaltered. Listing 1 provides an example of how to change CUDA code (top half) to use FitFloat arrays (bottom half).

B. Accessing FitFloat Elements

The array is accessed with the provided backing type for interoperability and to minimize changes to the source code. In other words, the code simply reads floats (or doubles) from the array and writes floats (or doubles) to the array. Hence, all floating-point operations like addition and multiplication remain unaltered.

When writing an element, the sign, exponent, and mantissa of the given value are isolated using shift and mask operations. If the exponent's value is too large or too small for the specified number of bits, we overflow to infinity or underflow to zero. We adjust the bias to a value that is appropriate for the used number of exponent bits. Then, the mantissa is truncated to fit the specified number of bits. Lastly, the 3 fields are concatenated and written to the array starting at the correct bit position.

Reading from a FitFloat array proceeds analogously. The word or words where the element is stored are read from memory, the value is extracted using shift and mask operations, and the 3 fields are isolated. The bias of the exponent is adjusted to the given backing type. The mantissa is padded with zeros to reach the required number of bits. Finally, the 3 fields are concatenated into the format of the backing floating-point type and returned.

C. GPU Implementation Details

When declaring a FitFloat array, the user provides the backing floating-point type, the number of elements E in the array, and the number of bits for the exponent and the mantissa. FitFloats are implemented as CUDA/C++ templates such that as much of the work as possible can be done at compile-time. Assuming the exponent, mantissa, and sign bit require a total of UB bits, the constructor allocates device memory to fit $E * UB$ bits, rounded up to the next word multiple. Furthermore, E is rounded up to the next multiple of 32 to allow for simpler warp-based code as discussed later in this section. We use the same word size as the backing type. For example, if the backing type is a float, we use a word size of 32 bits. The array is passed as a parameter to kernel launches, similar to how a floating-point array is passed.

1) Writing to FitFloat arrays: When a thread attempts to write to a FitFloat array element, the following occurs transparently, based on the given value and index into the array. The binary representation of the value is bit-for-bit treated as an unsigned integer of the same size. First, the sign, exponent, and mantissa fields are isolated. If the value is a NaN or infinity, meaning the exponent has all bits set to 1, all resulting exponent bits are also set to all 1s. In case of a normal or subnormal value, we check if it needs to be rounded up to infinity or down to 0 (if the number of exponent bits is too small to represent the value). If rounding is necessary, the exponent and mantissa bits are set accordingly. If no rounding is necessary, the exponent is debiased and rebiased according to the number of user-specified bits. Then, the mantissa is truncated to fit the specified number of bits. Finally, the sign bit is copied verbatim.

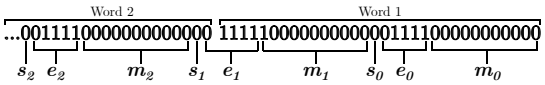

Fig. 1: FitFloat array layout with 3 elements spanning 2 words. The first element is on the right. There are 5 exponent and 11 mantissa bits. The backing word size is 32 bits.

The resulting sign, exponent, and mantissa are concatenated into the lower UB bits of a word. Next, the bit position in the backing array is calculated by multiplying the index by UB. Then, the index into the backing array is computed by dividing the bit position by the word size. The number of bits to shift the concatenated word by is also calculated from the bit position. When an element spans 2 words, the LSBs of the element are stored in the MSBs of the word at index p, and the remaining bits are stored in the LSBs of the word at index $p+1$. Figure 1 illustrates how the array elements are laid out.

To update the memory correctly and safely in parallel, synchronization must be utilized. When updating an element, the affected bits from the corresponding word or words are first cleared using a bitwise AND operator before the value is written using a bitwise OR operator. We opted to use an `atomicCAS` to read the current word, clear the desired bits, and set the needed bits instead of using an `atomicAND` followed by an `atomicOR` since we found the `atomicCAS` to provide better performance. If the FitFloat element spans two words in the backing array, another `atomicCAS` is used to update the following word in the same way.

2) Reading from FitFloat arrays: When a thread reads an element from a FitFloat array, a similar process happens transparently. After reading in 2 backing words, they are shifted to right-align the bits of the word we need. If the backing type is 32 bits wide, we employ a `__funnelshift_rc` operation to shift both words with one instruction. Otherwise, 2 standard shift operations are used.

With the bits of the element now in the LSBs of a word, the sign, exponent, and mantissa are isolated. If the exponent bits are all set to 1, the decoded exponent is also set to all 1s. Otherwise, if the exponent is not zero, the exponent is rebiased. Lastly, the three fields are concatenated, padding the mantissa with zeros as necessary, to reconstruct the original format before returning the value. Note that no synchronization is required for reading.

3) Special cases: There are certain cases where synchronization is not necessary even when writing. In these cases,

we provide "quick-write" functions to be used in place of the standard array access operator. They can be employed when all threads write to consecutive elements in the FitFloat array, and no 2 threads write to the same element, such as when initializing or copying an array. The idea is that, within a warp, threads can communicate efficiently and safely using warp-shuffle and warp-vote functions to combine the FitFloat elements into full words before writing them to the backing array, eliminating the need for synchronization between warps. We do this at a granularity that ensures that no warp is starting or ending in the middle of a word in the backing array.

In the quick-write function, each warp thread is given a consecutive word to write to the FitFloat array. It transforms the word as explained previously. The resulting subwords are shuffled (using `__shfl_xor`) and progressively combined until they are larger than half the word size. Then, they are transferred to other threads (using `__shfl`) such that the first few warp threads have all the bits they need to create a full word each. Only UB threads in a warp write to the backing array since each thread initially contributed UB bits.

Assuming a warp size of 32, a whole warp always writes a multiple of 32 bits to the backing array. Hence, with a word size of 32 bits, only full words are ever written, thus not requiring any inter-warp synchronization. To provide the same benefit for 64-bit words, we provide another special-case function that assigns each thread 2 consecutive elements.

D. CPU Implementation Details

We also wrote a CPU implementation of FitFloat arrays that is parallelized using OpenMP (OMP). We use it for our specialized device-to-host and host-to-device memory-copy functions that take a native floating-point array on the host, encode each element on the CPU in parallel, and use CUDA's memcpy to transfer the resulting FitFloat data to a FitFloat array on the device and vice versa. This makes it easier to integrate FitFloat into existing codes. As FitFloat is designed for GPUs, we do not evaluate any CPU codes in this paper. The CPU implementation is purely for transferring FitFloat arrays from the host to the device. However, when using Unified Memory, which we do in all experiments in this paper, no explicit data transfer to the device is needed after encoding (i.e., no CUDA memcpy functions are called) as the memory is automatically migrated to the GPU.

The CPU code is similar to the GPU code. When synchronization is required, we use OMP atomics. Moreover, we always use 2 shift operations when reading. Lastly, as there are no warps on CPUs, we forego synchronization by giving each CPU thread a block of elements to process that is a multiple of the backing word size long. This way, each thread can use bitwise AND and bitwise OR without requiring atomics, since the total number of bits each thread updates is always a multiple of the word size.

E. FitFloat Memory Footprint and Precision

Table II shows the percentage of memory *saved* when using FitFloats with 32-bit floats as the backing type along with the approximate digits of precision of each FitFloat element. For example, a FitFloat array with 17 mantissa and 6 exponent bits requires 25% less memory than the corresponding float array. Alternatively, the FitFloat array can store 33% more elements in the same amount of memory.

TABLE II: FitFloat memory savings and decimal precision with float backing type for a few FitFloat configurations

Exponent Bits	Mantissa Bits	Memory Savings (%)	Digits of Precision	Approximate Range of Positive Normal Values
8	23	0.00	7.22	3.40×10^{38}
8	17	18.75	5.42	3.40×10^{38}
8	15	25.00	4.82	3.40×10^{38}
7	23	3.12	7.22	1.84×10^{19}
7	19	15.62	6.02	1.84×10^{19}
7	17	21.88	5.42	1.84×10^{19}
7	15	28.12	4.82	1.84×10^{19}
7	12	37.50	3.91	1.84×10^{19}
6	22	9.38	6.92	4.29×10^{9}
6	21	12.50	6.62	4.29×10^{9}
6	12	40.62	3.91	4.29×10^{9}
5	15	34.38	4.82	6.55×10^{4}
4	17	31.25	5.42	2.56×10^{2}
4	12	46.88	3.91	2.56×10^{2}

What precision to use (i.e., what FitFloat bit counts to use) is application dependent. This is akin to how programmers already choose, for instance, between single- and double-precision floats. FitFloats only extend the number of choices.

Along with domain knowledge, there are several options to assist programmers in selecting good FitFloat configurations. Precision auto-tuning is a popular strategy for determining the precision to use, and many of the related works listed in Section II discuss this approach. Moreover, range profiling can be employed to gather value ranges [36], [37]. Once this information has been obtained, Table II can be used to select FitFloat sizes that meet the precision threshold.

IV. Experimental Methodology

We evaluate the performance of using FitFloat arrays in the context of the 14 HeCBench codes described below. We ran the programs on the 2 systems listed in Table III, which contain 2 GPUs. All codes are compiled with `-O3 -arch=sm_XX -Xcompiler -fopenmp`, where `sm_XX` denotes the compute capability of the target GPU.

TABLE III: Systems used for experiments

	System 1	System 2
CPU	Threadripper 2950X	Xeon Gold 6226R
Base Clock	3.5 GHz	2.9 GHz
Sockets	1	2
Cores Per Socket	16	16
Threads Per Core	2	2
Main memory	64 GB	64 GB
GPU	RTX 4090	A100
Compute Capability	8.9	8.0
Base Clock	2.2 GHz	0.8 GHz
Boost Clock	2.5 GHz	1.4 GHz
SMs	128	108
CUDA Cores per SM	128	64
Main memory	24 GB GDDR6x	40 GB HBM2e
Peak memory bandwidth	1.01 TB/s	1.56 TB/s
Operating System	Fedora 37	Fedora 36
`g++` Version	12.2.1	12.2.1
`nvcc` Version	12.0	12.0
GPU Driver	525.85	535.113

We measure the runtime of the CUDA kernels using the existing timing code provided by HeCBench and take the median of 9 runs. For the codes with multiple kernels, we measure all kernel runtimes separately, compute the geometric mean, and report the median. We do this for both the FitFloat and the native version of each code. Furthermore, for the FitFloat versions, we repeat this experiment for different combinations of exponent and mantissa bits. For float-backed FitFloat arrays, we tested 4 to 8 exponent and 12 to 23 mantissa bits, yielding a total of 17 to 32 bits per FitFloat word. For double-backed FitFloat arrays, we tested 4 to 11 exponent and 28 to 52 mantissa bits, giving a total of 33 to 64 bits per FitFloat word.

The HeCBench codes generate their own random inputs, and we employ a fixed seed to ensure the same input is used across runs to make the comparisons fair. Importantly, we verified that the reduced precision does not affect the computation. In other words, all tested bit counts are safe and yield the same results. We further verified that the number of memory accesses to the FitFloat arrays with all evaluated bit counts is the same as with the native arrays. Since BSearch produces a different result when run with fewer than 6 exponent bits, we do not evaluate it with 4- or 5-bit exponents.

To evaluate FitFloat with its intended purpose in mind, we compare it to native arrays in Unified Memory. For our first set of experiments, we sized the inputs for all benchmarks such that the native arrays exceed the memory capacity but the FitFloat arrays fit when using elements with fewer than 32 bits on the float benchmarks and 64 bits on the double benchmarks. In our second set of experiments, we sized the inputs for all benchmarks such that both the native and FitFloat arrays exceed the memory capacity for all evaluated bit counts. To make the comparison as fair as possible, the FitFloat and native arrays are allocated in Unified Memory. Moreover, we do not provide any memory usage hints to the GPU driver managing the unified memory, we do not prefetch any data, and we fully initialize all arrays on the host.

A. HeCBench Codes

HeCBench [38] is a collection of benchmarks, samples, and mini-applications from popular benchmark suites such as Rodinia [39], Chai [40], and open-source projects, containing over 350 codes in total. We selected the first 14 HeCBench programs in alphabetical order for evaluation that support FitFloat (i.e., they use floating-point arrays, do not use pointers to access array elements, and do not perform atomic updates to the array). As HeCBench is a general collection of benchmarks, a large majority of the codes do not use floating-point arrays, making them ineligible for FitFloat evaluation. 11 of the selected codes use one or more float arrays and the remaining 3 codes use one or more double arrays.

For each selected benchmark, we created a FitFloat version by replacing all global floating-point arrays used in a kernel with FitFloat arrays. We then compare the performance of the FitFloat version to that of the native version. Next, we explain what the codes compute and which arrays we targeted.

The **Accuracy** (*float*) code computes how many predicted classifications match the ground truth. The predication and ground truth values are stored in a single float array. The **Adam** (*float*) code runs an adaptive moment estimation optimizer. There are 4 float arrays in the kernel. This code makes repeated memory accesses to the same element in a nested loop. We factored the memory accesses out of the inner loop and separately analyze this optimized version, which we call AdamOPT. The **Adv** (*double*) code performs an adaptive integration on a 3D integral. There are 5 double arrays in the kernel. The **Aidw** (*float*) code computes the adaptive inverse distance weighting for multivariate interpolation with 7 arrays. The **Asta** (*double*) code performs a parallel tiled transpose within and across cycles using an array-of-tiled-structure-of-arrays approach. We target the 1 double array. The **Attention** (*float*) code computes the scaled dot product attention used in transformers. It uses 3 arrays. The **Bilateral** (*float*) code runs a bilateral smoothing filter on image data. There are 2 float arrays, an input and an output. The **Bincount** (*float*) code computes a histogram of a given input float array. The **BScholes** (*float*) code computes the Black-Scholes formula on some data and outputs it to a float array. The **BSearch** (*float*) code contains 4 classic and vectorizable binary search algorithms. There are 2 float arrays in each implementation. With 4- and 5-bit exponents, the codes overflow or underflow, so we omit these exponent sizes. The **Burger** (*double*) code computes the 2D Bateman-Burgers equation, a PDE. It uses 4 double arrays. The **Car** (*float*) code performs content adaptive resampling on an image. It uses 5 float arrays. The **Chi2** (*float*) code computes the chi-squared test used in contingency table analysis. It uses 1 float array. The **FHD** (*float*) code performs an advanced magnetic resonance imaging (MRI) reconstruction. It uses 7 float arrays.

V. Results

In this section, we present and analyze our experimental results. In Subsections V-A and V-B, we discuss the performance benefit of using FitFloat arrays compared to native arrays using Unified Memory across all evaluated HeCBench codes. Then, in Subsection V-C, we analyze the performance of individual codes to provide insight as to which types of codes gain the most benefit from FitFloat arrays. Lastly, in Subsection V-D, we explore how FitFloat increases the L2 cache hit rate and decreases the amount of data transferred between the host and the device due to page faults.

We use Unified Memory for our comparison in two distinct cases. 1) We sized the inputs such that, when using the full floating-point type, the total size of the arrays exceeds the global memory capacity. However, when the number of bits per FitFloat is smaller than the width of the backing type, the total size of the arrays fits in global memory. 2) We sized the inputs such that, for all evaluated bit counts, the total size of both the native and FitFloat arrays exceeds the memory size.

As FitFloat is a first-of-its-kind compressed in-memory storage format for floating-point arrays on the GPU, Unified Memory is the only solution comparable to ours. No other

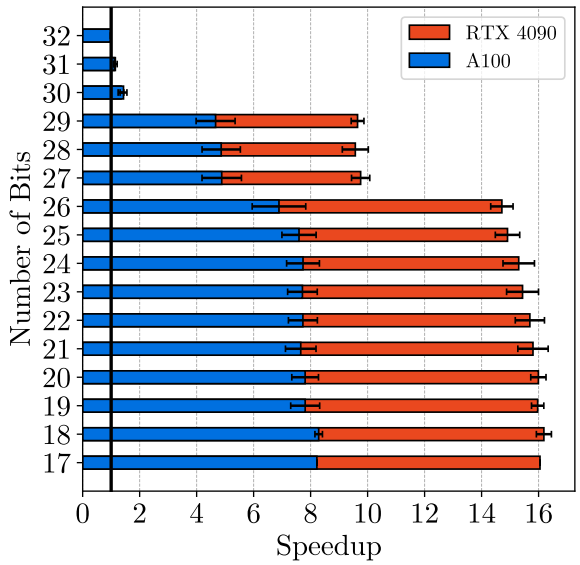

Fig. 2: Geometric-mean speedup across 11 float benchmarks using FitFloat over Unified Memory. FitFloat arrays with fewer than 32 bits per element fit within global memory.

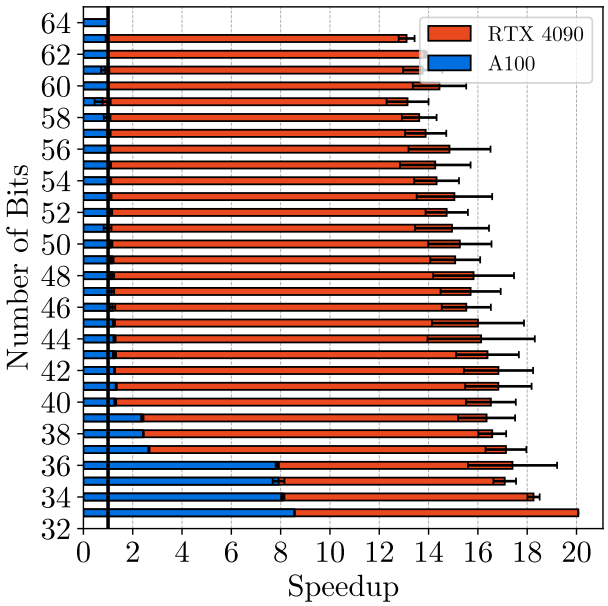

Fig. 3: Geometric-mean speedup across 3 double benchmarks using FitFloat over Unified Memory. FitFloat arrays with fewer than 64 bits per element fit within global memory.

solution allows for read and write random-access to individual elements of a compressed floating-point array on the GPU. The first case mentioned above is the main use case FitFloat is designed for, allowing larger problem sizes to be stored within the global memory capacity, and our results (Section V-A) demonstrate that FitFloat excels at this over Unified Memory. The results for the second case (Section V-B) show that FitFloat arrays are still superior to native arrays with Unified Memory even when the FitFloat arrays also do not fit within the global memory capacity.

A. *Only FitFloat Arrays Fit in Global Memory*

Figures 2 and 3 show the geometric-mean speedups on the RTX 4090 and A100 GPUs across all float and double codes, respectively. In these experiments, the native array sizes exceed the global memory capacity while the FitFloat configurations with fewer than 32 bits (64 bit for doubles) per element fit. The speedup runs along the x-axis, and the number of bits for each FitFloat element runs along the y-axis. Higher is better. These speedups represent the geometric mean across the 11 float codes (Figure 2) and the 3 double codes (Figure 3). Since multiple FitFloat configurations yield the same number of bits per element, we compute the geometric mean across multiple configurations, use it for the bar length, and show the range of the geometric-mean speedups for the multiple configurations using error bars. The color identifies the device.

On the 4090, FitFloat always provides a speedup over using native arrays when the number of bits per FitFloat element is smaller than the native word size, with the performance increasing as the element size decreases. FitFloat achieves speedups from 1.1 to 16.1 for the float codes and 13.1 to 20.1 for the double codes. On the A100, the trends are the same, with speedups ranging from 1.1 to 8.2 for the float codes. For the double codes, FitFloat performs about on par with native arrays for bit counts in the range of 64 down to 51. Below 51 bits, the speedups range from 1.1 to 8.6.

These results highlight the strength of FitFloat. Not only is the memory footprint of the floating-point arrays reduced, but substantial speedups are achieved due to more elements residing in the same amount of memory, thus minimizing memory movement and page migrations. With Unified Memory, when a memory address is accessed whose page does not currently reside in global memory, the driver must migrate that page from the host to the device, potentially evicting a page that must later be migrated back to the device. With FitFloat bit counts smaller than 32 (64 for double codes), after the initial page migration (when a page is first accessed), all pages can stay in global memory for the duration of the kernel as there is no need to evict any pages in this set of experiments.

The performance of FitFloat arrays increases as the bits per element decreases. Since the FitFloat array size decreases with the number of bits, fewer overall global memory accesses are required as the same number of elements take up less space. Note that the resulting speedups are not due to fewer *array* accesses. In fact, we verified that the FitFloat codes execute the same number of accesses as the native codes. In other words, these codes are not sensitive to precision and perform the same computation for all tested bit counts. FitFloat's performance is due to moving less data and fitting more elements in the caches since each element is smaller, explaining why we see an increase in performance with smaller FitFloat sizes.

The A100 delivers lower speedups than the RTX 4090 as it has a much lower computation-to-bandwidth ratio. Since FitFloat performs best when the computation overhead of encoding and decoding the elements is hidden behind the

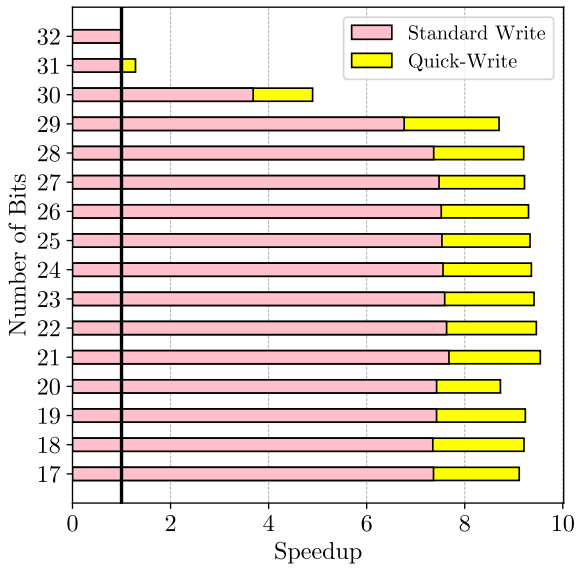

Fig. 4: Geometric-mean speedup across 5 float benchmarks using FitFloat with and without quick-write over Unified Memory on an RTX 4090. FitFloat arrays with fewer than 32 bits per element fit within global memory.

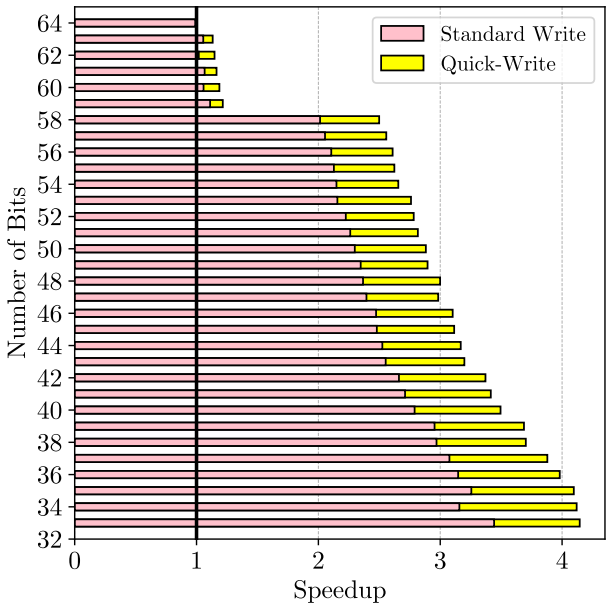

Fig. 5: Geometric-mean speedup across 1 double benchmark using FitFloat with and without quick-write over Unified Memory on an RTX 4090. FitFloat arrays with fewer than 64 bits per element fit within global memory.

memory latency, the speedups provided by FitFloat are not as high on the A100 but still on par or faster than native arrays. We also ran FitFloat on other GPU types and found the speedups to lie between the A100 and RTX 4090 results.

In most cases, FitFloat is much faster on these codes, highlighting the fast individual element processing. There is no other compression scheme like FitFloat for GPUs, and these results show that FitFloat is an order of magnitude faster than Unified Memory for scientific codes working with datasets that are too large to fit within the global memory.

1) Quick-Write Analysis: We now present results of using our quick-write function in the case where FitFloat arrays fit in globlal memory with fewer bits than the native size and native arrays do not. There are 5 float codes (Aidw, Attention, Bilateral, Car, and FHD) and 1 double code (Burger) that only require replacing writes to the array using the standard `[]` operator with the `quick_write()` function. We focus on these codes. The other codes would require more significant changes to employ quick-write.

Figures 4 and 5 show the geometric-mean speedups on the RTX 4090 GPU across the 5 float and 1 double codes using quick-write, respectively. The color of the bars correspond to the performance using quick-write and the standard `[]` operator. The speedup calculation in both cases is based on the performance of just the 6 quick-write codes.

With fewer than 32 bits per FitFloat on the float codes, quick-write outperforms the standard write. The speedups of the standard write over native arrays range from 0.98 to 7.68. With quick-write, they range from 0.99 to 9.54, making quick-write 17.5% to 32.9% faster than the standard write.

The double results show the same trend. With fewer than 64 bits per FitFloat element, the standard write speedups over native arrays range from 1.02 to 3.44 and the quick-write speedups range from 1.13 to 4.14. In other words, quick-write is 7.32% to 30.5% faster than the standard write.

For both the float and double codes, the trends remain the same in the case where both FitFloat and native arrays do not fit in global memory. Hence, we do not show these results.

Overall, these results show that using quick-write can provide a substantial boost in FitFloat performance for all bit counts smaller than the native word size due to quick-write not requiring the synchronization that the standard write needs. Even codes such as Aidw and FHD that already perform on par or provide a speedup with FitFloat when using the standard write see a boost in performance with quick-write.

B. Even FitFloat Arrays Exceed Global Memory

This subsection presents the results from our experiments comparing FitFloat arrays to native arrays in the case where all FitFloat and native array sizes exceed the global memory capacity. Figures 6 and 7 present the geometric-mean speedups on the RTX 4090 and A100 across all float and double codes, respectively. The figure layout is the same as before.

The general performance trends of FitFloat in these results are the same, with fewer bits per FitFloat element resulting in higher speedups. On the 4090, the speedups for the float codes range from 1.3 to 5.0, with all bit counts under 32 providing a speedup. For the double codes, the FitFloat performance is roughly on par with native arrays for bit counts down to 58. With fewer bits, the speedups range from 1.1 to 10.3. The FitFloat performance on the A100 is again lower than on the 4090 due to the aforementioned reasons, and a few bit counts now incur a slowdown. For the float codes, the speedups range

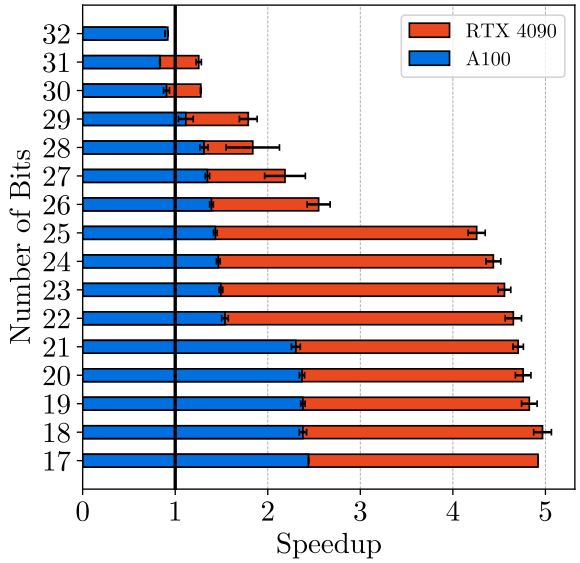

Fig. 6: Geometric-mean speedup across 11 float benchmarks using FitFloat over Unified Memory. Both native and FitFloat array sizes exceed the global memory capacity.

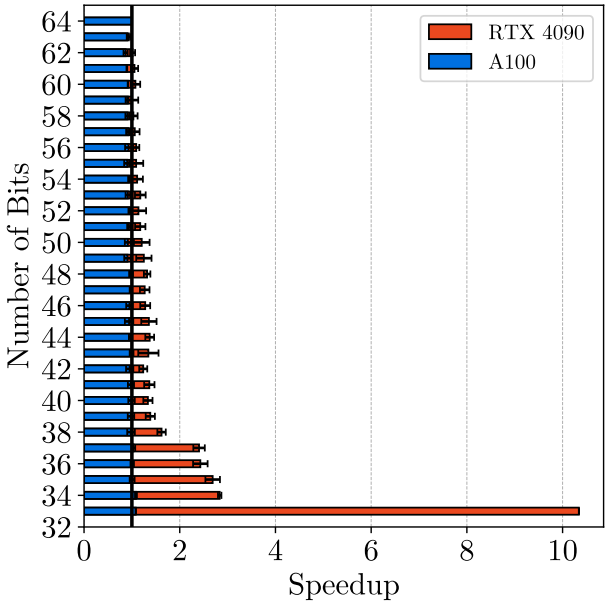

Fig. 7: Geometric-mean speedup across 3 double benchmarks using FitFloat over Unified Memory. Both native and FitFloat array sizes exceed the global memory capacity.

from 1.1 to 2.4, and the worst case is about 0.8. For the double codes, the performance is mostly on par with native arrays, ranging from 0.9 to 1.1.

For the float codes, the performance trends are similar to the previous case. For the double codes on the 4090, the FitFloat performance starts to improve with fewer than 58 bits per element. This is *not* because the FitFloat arrays are able to fit in global memory (we verified that the array sizes in these experiments do not fit for all evaluated FitFloat bit counts). Rather, these codes have an active working set of elements that

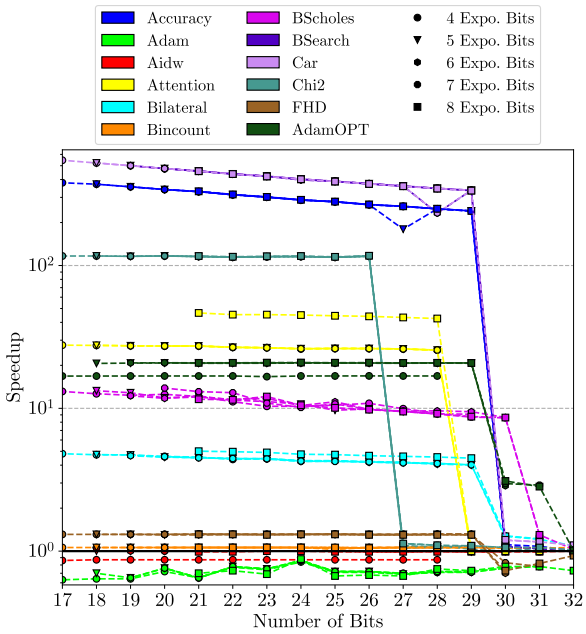

Fig. 8: Median speedup of 11 float benchmarks using FitFloat over Unified Memory on the RTX 4090. FitFloat arrays with fewer than 32 bits per element fit within global memory.

the threads process at a time (i.e., temporal locality), which, for small enough FitFloat elements, fits in global memory, causing fewer page migrations than the native case.

These results show that, in cases where not even the FitFloat array fits in global memory, the performance is generally at least on par and often substantially better than that of native arrays, especially for float codes. Hence, FitFloat is also beneficial over Unified Memory for scientific GPU codes that work with very large floating-point arrays.

C. Individual Benchmark Analysis

In this subsection, we analyze the performance of the HeCBench codes individually in the primary use case of FitFloat, that is, the case where FitFloat arrays with fewer bits than the native word size (32 for floats and 64 for doubles) fit within global memory, and all other arrays do not.

Figure 8 presents the speedup of the 11 HeCBench float codes on the RTX 4090. Figure 9 does the same but on the A100. The number of bits for each element in the FitFloat array runs along the x-axis, and the speedup runs along the y-axis. Higher is better. The symbol of each data point signifies the number of exponent bits. Note that most of the symbols overlap as there is no difference in performance between the different exponent sizes. The color of each data point indicates which code the speedup is from. These figures use a log scale on the y-axis.

On the 4090, the Adam code is the only code that incurs a slowdown for all tested bit counts, around 0.4 in the worst case. When optimizing the memory accesses in this code (see Section IV), the performance drastically improves, going from a slowdown to a speedup for all bit counts fewer than

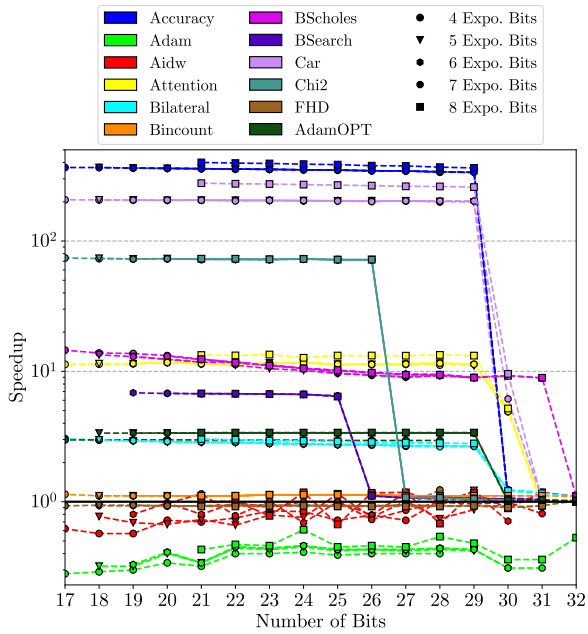

Fig. 9: Median speedup of 11 float benchmarks using FitFloat over Unified Memory on the A100. FitFloat arrays with fewer than 32 bits per element fit within global memory.

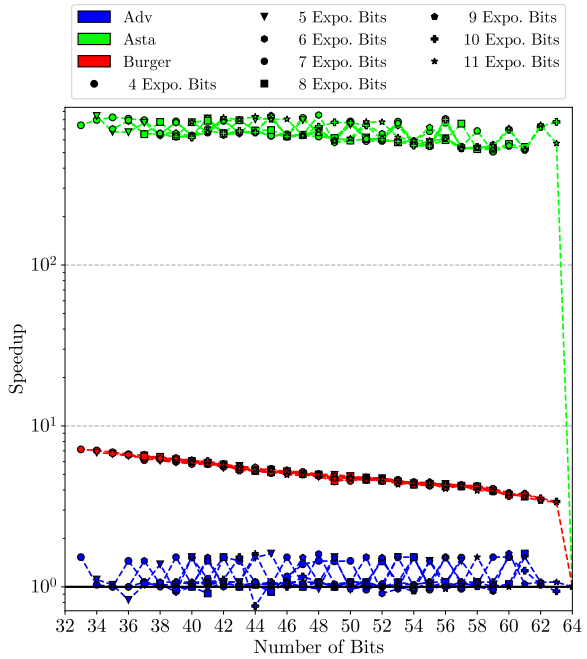

Fig. 10: Median speedup of 3 double benchmarks using FitFloat over Unified Memory on the RTX 4090. FitFloat arrays with fewer than 64 bits per element fit within global memory.

32 (AdamOPT in the figure). Aidw and FHD also incur a slowdown in the worst case but are on par or provide a speedup for at least some bit counts. All other codes yield a speedup when using 31 or fewer bits per FitFloat element. For these codes, when using 17 bits per element, the speedups range

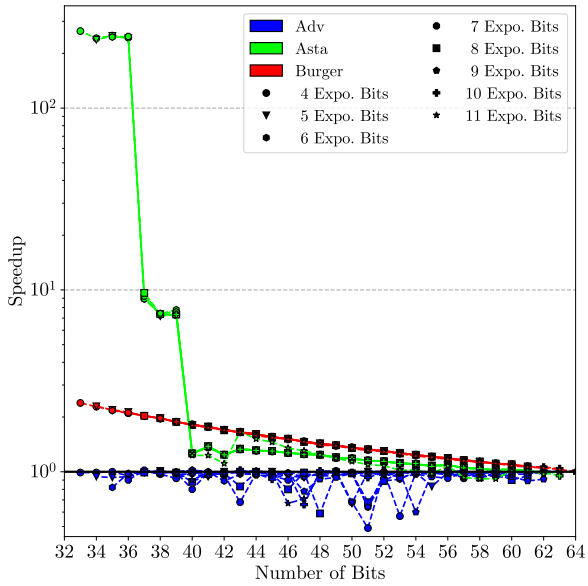

Fig. 11: Median speedup of 3 double benchmarks using FitFloat over Unified Memory on the A100. FitFloat arrays with fewer than 64 bits per element fit within global memory.

from 1.1 for the Bincount code to 555.2 for the Car code.

On the A100, the trends are similar. The Adam code has a slowdown of 0.28 in the worst case. The Aidw and FHD codes incur more of a slowdown on the A100. All other codes yield a speedup with fewer than 32 bits per FitFloat element, ranging from 1.1 for the Bincount code to 366.4 for the Accuracy code.

Figures 10 and 11 present the speedup of the 3 HeCBench double codes on the RTX 4090 and on the A100, respectively. On both devices, the Adv code oscillates between being on par or faster than the native arrays. This code loads the input arrays from global memory into shared memory, which boosts performance, but it makes repeated writes to the output array, which hurts performance, in particular with 64-bit words and an odd number of bits. This is because, with an even number of bits, an integer number of FitFloat elements fit into every 32 words of the backing array. With an odd number of bits, this is only the case every 64 words, leading to more atomic conflicts since more elements overlap. The Burger and Asta codes provide a speedup with fewer than 64 bits per FitFloat, with the Asta code providing the highest speedup. On the 4090, when using 33 bits per FitFloat element, the speedups range from 1.5 for the Adv code to 739.0 for the Asta code. On the A100, the speedups range from 1.0 for the Burger code to 265.2 for the Asta code.

Looking at the float and double codes together, FitFloat provides a speedup for all 14 benchmarks on the RTX 4090 after applying optimizations to the Adam code. On the A100, it provides a speedup for all but 2 codes, Aidw and FHD. As explained in Section V-A, these performance gains are the benefit of not needing to evict pages after the initial migration. For a majority of the codes, the performance of FitFloat arrays increases as the bits per element decreases. The codes where FitFloat does not perform as well have characteristics that

impede FitFloat's performance, as we explain next.

Analyzing the Adam code, we found that it exploits full coalescing when accessing the native arrays. In contrast, FitFloat may have to resort to partial coalescing and must contend with conflicting atomics when using fewer than 32 bits per element as there are multiple FitFloat elements stored in each 32-bit word. Executing multiple atomics that access the same memory location is known to reduce performance [41]. Furthermore, and as mentioned before, the Adam code makes repeated reads and writes inside a nested for-loop. With our manual optimizations (factoring the redundant array accesses out of the inner loop), FitFloat provides a speedup even on this code. In fact, in the native array case, the manually optimized Adam code has the same performance as the original Adam code, indicating that the compiler performs this optimization automatically for the native array version. However, the compiler cannot do so for the FitFloat version because it does not know that the optimization is also legal for FitFloat types.

The native Attention and FHD codes also exploit full coalescing. However, the Attention code only uses 1 array and each thread only writes 1 element, which is why it performs better than the unoptimized Adam code, which uses 4 arrays and each thread writes many times to multiple elements. The FHD, in contrast, reads and writes to a single array and, therefore, performs in between the unoptimized Adam code and the Attention code. Aidw uses coalesced writes to multiple output arrays, but each thread reads from the same memory locations in the input arrays.

All the particularly well-performing codes either only read, such as BSearch and Accuracy, or load the data from global- to shared-memory arrays or local variables, perform the computations, and then write the result, such as the Car and Burger codes. Lastly, BScholes uses one float array to store the output of the computation, and each thread writes to this array once.

In summary, to achieve especially good performance with FitFloat arrays, the following should be considered. When reading, repeated accesses should be avoided by storing the read elements in a local variable or shared memory when possible to avoid the penalty of accessing contiguous elements spanning two cache lines. Similarly, repeated writes to the same memory location should be avoided to reduce the time spent encoding values. Most importantly, the number of writes should be minimized to avoid the slow atomic operations.

The results from these experiments show that, for all codes on the 4090, and almost all codes on the A100, Fitfloat allows for larger problem sizes to be run on the GPU while providing much better performance than native arrays using Unified Memory. When applying good GPU programming practices, as seen in the case of optimizing the Adam code, FitFloat can provide large speedups.

D. FitFloat Data Movement Impact

This subsection presents results exploring the performance impact of FitFloat outside of program runtime. In particular, we measure how the L2 cache hit rate changes when employing FitFloat compared to native arrays using Unified Memory as well as the number of bytes being transferred between the host and the device triggered by Unified Memory page faults.

We evaluate all codes for the data transfer experiment and all but one code (Aidw) for the cache hit rate experiment as the profiler struggles to profile the Aidw code in a reasonable time. We use a subset of FitFloat configurations that spans the range of configurations from our other experiments. For the float codes, we use 9 configurations in the range of 17–32 bits per FitFloat element. For the double codes, we use 9 configurations in the range of 33–64 bits per element. We evaluate FitFloat in the case where FitFloat arrays with fewer bits per element than the native word size fit in global memory.

TABLE IV: Geometric-mean L2 cache hit rate across all float codes for FitFloat and native arrays. FitFloat bit counts under 32 fit in global memory.

FitFloat Bit Count	FitFloat Hit Rate (%)	Native Hit Rate (%)
32	66.8	67.7
30	76.1	67.7
28	73.4	67.7
26	76.1	67.7
24	75.2	67.7
22	75.8	67.7
21	78.7	67.7
19	79.3	67.7
17	76.3	67.7

TABLE V: Geometric-mean L2 cache hit rate across all double codes for FitFloat and native arrays. FitFloat bit counts under 64 fit in global memory.

FitFloat Bit Count	FitFloat Hit Rate (%)	Native Hit Rate (%)
64	62.8	64.5
60	67.4	64.5
57	74.6	64.5
52	82.5	64.5
48	79.9	64.5
45	84.8	64.5
40	81.3	64.5
36	80.8	64.5
33	82.3	64.5

TABLE VI: Geometric-mean page fault memory transfer reduction across all float codes and double codes using FitFloat over Unified Memory. FitFloat bit counts under 32 for float codes and 64 for double codes fit in global memory.

Bit Count	Memory Reduction	Bit Count	Memory Reduction
32	0.90x	64	0.92x
30	0.95x	60	1.46x
28	1.45x	57	1.56x
26	1.22x	52	1.71x
24	1.62x	48	1.85x
22	1.88x	45	1.99x
21	1.86x	40	2.22x
19	2.15x	36	2.46x
17	2.54x	33	2.70x

Tables IV and V show the geometric-mean L2 cache hit rate of FitFloat arrays with varying bit counts and native arrays across the float codes and double codes, respectively. The native hit rate is constant across FitFloat bit counts as

the native element size is not changing. For the float codes, the native L2 hit rate is 67.7%. The FitFloat L2 hit rate is 66.8% at 32 bits per element, a little lower than the native hit rate. However, as the bits per element decreases, the hit rate increases, reaching a peak geometric-mean hit rate of 79.3% at 19 bits per element on these codes. The trend is the same for the double codes. The native hit rate is 64.5%, and the FitFloat hit rate at 64 bits per element is a little lower at 62.8% but then increases as the bit count decreases, reaching a peak geometric-mean hit rate of 84.8% for 45 bits per element. Note that these results are application dependent and not all of the tested codes peak at the same bit counts.

Table VI presents the geometric-mean memory reduction. The memory reduction is the ratio of how much *less* data FitFloat arrays transfer via page faults compared to native arrays, that is, the number of bytes transferred using native arrays over the number of bytes transferred using FitFloat arrays. Higher is better. For example, a memory reduction of 2.0 means half as much data is transferred via page faults.

At 32 bits per element for the float codes, FitFloat arrays provide a memory reduction of 0.9, indicating that more data is transferred in this configuration. As the bits per element decreases, the memory reduction increases, with a reduction of 2.54 at 17 bits per element. The double codes exhibit the same trend, starting at 0.92 for 64 bits per element and increasing to 2.70 at 33 bits per element.

At bit counts close to the native word size, FitFloat arrays transfer a little more data via page faults due to the extra memory it allocates (e.g., the FitFloat struct). Similarly, when looking at the L2 hit rate, FitFloat arrays with bit counts matching the native word size have a slightly lower hit rate than native arrays. Once the FitFloat elements are small enough, the cache hit rate and the amount of data transferred outperforms native arrays as expected because more array elements fit in the cache and global memory. Consequently, the overall performance of these codes improves while still computing the same results.

Note that the speedup increases (see Figures 2 and 3) tend to be much higher than the memory reduction. This is because page faults are very slow, causing the native code to spend a substantial amount of time waiting for data. Lowering the number of page faults by employing FitFloat can greatly reduce this waiting time, yielding substantial speedups.

VI. Summary

This paper presents FitFloat, a compressed in-memory storage format for floating-point arrays on GPUs. It allows the user to select the number of exponent bits and, separately, the number of mantissa bits used to represent the floating-point values in an array. Accesses to a FitFloat array are always performed with the native float or double type. FitFloat transparently encodes the native type to lossily fit the user specification when writing and decodes the value back to the native type when reading, requiring minimal changes to the source code. FitFloat targets applications limited by the GPU's memory capacity, making it possible to efficiently process larger problem sizes. As far as we know, FitFloat is the first compressed floating-point array format for GPUs supporting read and write random accesses.

FitFloat arrays allow for floating-point sizes in between the floating-point types provided by the hardware. Even on newer GPUs supporting smaller types (e.g., 16- and 8-bit floats), the optimal precision for an array is often between two hardware types (e.g., more precision than 16-bit but less than 32-bit floats). Currently, applications would employ the larger of the two types, using more memory than necessary. FitFloat makes it possible to get close to the minimum required precision, thus reducing the storage requirement and substantially increasing the performance over large native arrays in Unified Memory.

We evaluated the performance of using FitFloat arrays relative to using native IEEE 754 floating-point arrays on 14 benchmarks from the HeCBench suite. 11 of these programs use float arrays and the remaining 3 programs use double arrays. When the native arrays are too large to fit in the GPU's global memory (and must be allocated in Unified Memory) but the FitFloat arrays do fit, FitFloat provides a geometric-mean speedup of up to 16.1 for the float codes and 20.1 for the double codes on an RTX 4090 GPU, and an geometric-mean speedup of up to 8.2 for the float codes and 8.6 for the double codes on an A100 with the smallest tested FitFloat configuration. In the best configuration, FitFloat yields a speedup of 739.0 on the 4090 and 366.4 on the A100.

We also evaluated our quick-write function, which allows writes to FitFloat arrays without synchronization in certain cases. Using quick-write is up to 32.9% faster than standard FitFloat writes. By profiling all 14 codes with and without using FitFloat, we found that FitFloat improves the L2 cache hit rate by up to 17.8% and transfers up to 2.7 times fewer bytes triggered by page faults compared to native arrays.

Our comparisons of FitFloat to native arrays in Unified Memory show FitFloat to generally be much faster while also reducing the storage requirements for these codes, highlighting FitFloat's fast individual element compression and decompression. We are not aware of any other approach like FitFloat for GPUs, and our experiments demonstrate that FitFloat is an order of magnitude faster than Unified Memory on scientific codes working with arrays that are too large to fit in the GPU's global memory.

Acknowledgments

This work has been supported by the U.S. Department of Energy, National Nuclear Security Administration under Award DE-NA0003969, by the U.S. National Science Foundation under Award CCF-2403380, and by an equipment donation from NVIDIA Corporation.

References

[1] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15, https://doi.org/10.1109/SC41405.2020.00073.

[2] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of Unified Memory Access performance in CUDA," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6, https://doi.org/10.1109/HPEC.2014.7040988.

[3] P. Lindstrom, J. Hittinger, J. Diffenderfer, A. Fox, D. Osei-Kuffuor, and J. Banks, "ZFP: A compressed array representation for numerical computations," *The International Journal of High Performance Computing Applications*, vol. 39, no. 1, pp. 104–122, 2025, https://doi.org/10.1177/10943420241284023.

[4] "Fitfloat github repository," https://github.com/burtscher/FitFloat, 2026.

[5] A. Rodriguez and M. Burtscher, "Fitfloat: Read/write random-access compressed floating-point arrays for gpus artifact," 2026. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.21542360

[6] A. Buttari, J. J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Exploiting Mixed Precision Floating Point Hardware in Scientific Computations." in *High Performance Computing Workshop*, 2006, pp. 19–36.

[7] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, pp. 113–es.

[8] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 4, pp. 1–22, 2008, https://doi.org/10.1145/1377596.1377597.

[9] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, and N. Wehn, "The transprecision computing paradigm: Concept, design, and applications," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1105–1110, https://doi.org/10.23919/DATE.2018.8342176.

[10] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 273–286, 2000, https://doi.org/10.1109/92.845894.

[11] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, and L. Benini, "A transprecision floating-point architecture for energy-efficient embedded computing," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–5, https://doi.org/10.1109/ISCAS.2018.8351816.

[12] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, 2013, pp. 369–378, https://doi.org/10.1145/2464996.2465018.

[13] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12, https://doi.org/10.1145/2503210.2503296.

[14] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 300–315, 2017, https://doi.org/10.1145/3093333.3009846.

[15] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 333–343, https://doi.org/10.1145/3213846.3213862.

[16] A. Borghesi, G. Tagliavini, M. Lombardi, L. Benini, and M. Milano, "Combining learning and optimization for transprecision computing," in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, 2020, pp. 10–18, https://doi.org/10.1145/3387902.3392615.

[17] B. Saiki, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock, "Combining precision tuning and rewriting," in *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2021, pp. 1–8, https://doi.org/10.1109/ARITH51176.2021.00013.

[18] A. Adjé, D. Ben Khalifa, and M. Martel, "Fast and efficient bit-level precision tuning," in *International Static Analysis Symposium*. Springer, 2021, pp. 1–24, https://doi.org/10.1007/978-3-030-88806-0_1.

[19] Y. Wang and C. Rubio-González, "Predicting performance and accuracy of mixed-precision programs for precision tuning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[20] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[21] N.-M. Ho and W.-F. Wong, "Exploiting half precision arithmetic in Nvidia GPUs," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7, https://doi.org/10.1109/HPEC.2017.8091072.

[22] A. Ioualalen and M. Martel, "Neural network precision tuning," in *Quantitative Evaluation of Systems: 16th International Conference, QEST 2019, Glasgow, UK, September 10–12, 2019, Proceedings 16*. Springer, 2019, pp. 129–143, https://doi.org/10.1007/978-3-030-30281-8_8.

[23] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 603–613, https://doi.org/10.1109/SC.2018.00050.

[24] T. Grützmacher, R. Underwood, S. Di, F. Cappello, and H. Anzt, "FRSZ2 for In-Register Block Compression Inside GMRES on GPUs," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 240–249, https://doi.org/10.1109/SCW63240.2024.00038.

[25] A. S. Molahosseini, J. Lee, and H. Vandierendonck, "Software-defined number formats for high-speed belief propagation," *IEEE Transactions on Emerging Topics in Computing*, 2025, https://doi.org/10.1109/TETC.2025.3528972.

[26] H. Vandierendonck, "Software-defined floating-point number formats and their application to graph processing," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–17, https://doi.org/10.1145/3524059.3532360.

[27] "The GNU Multiple Precision Arithmetic Library," https://gmplib.org/, 2023.

[28] "Boost Multiprecision Library," https://github.com/boostorg/multiprecision, 2025.

[29] J. R. Hauser, "Berkeley SoftFloat," https://www.jhauser.us/arithmetic/SoftFloat.html, 2024.

[30] D. H. Bailey, H. Yozo, X. S. Li, and B. Thompson, "ARPREC: An arbitrary precision computation package," 2002, https://doi.org/10.2172/817634.

[31] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, pp. 13–es, 2007, https://doi.org/10.1145/1236463.1236468.

[32] G. Tagliavini, A. Marongiu, and L. Benini, "FlexFloat: A Software Library for Transprecision Computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 1, pp. 145–156, 2020, https://doi.org/10.1109/TCAD.2018.2883902.

[33] A. Anderson and D. Gregg, "Vectorization of multibyte floating point data formats," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016, pp. 363–372, https://doi.org/10.1145/2967938.2967966.

[34] D. Mukunoki and T. Imamura, "Reduced-Precision Floating-Point Formats on GPUs for High Performance and Energy Efficient Computation," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 144–145, https://doi.org/10.1109/CLUSTER.2016.77.

[35] G. Flegar, F. Scheidegger, V. Novaković, G. Mariani, A. E. Tomás, A. C. I. Malossi, and E. S. Quintana-Ortí, "Floatx: Ac++ library for customized floating-point arithmetic," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–23, 2019, https://doi.org/10.1145/3368086.

[36] A. W. Brown, P. H. Kelly, and W. Luk, "Profiling floating point value ranges for reconfigurable implementation," in *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing*, 2007, pp. 6–16.

[37] ——, "Profile-directed speculative optimization of reconfigurable floating point data paths," in *Proceedings of the Workshop on Reconfigurable Computing at HiPEAC*, 2008, p. 53.

[38] Z. Jin and J. S. Vetter, "A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 325–327, https://doi.org/10.1109/ISPASS57527.2023.00041.

[39] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. Ieee, 2009, pp. 44–54, https://doi.org/10.1109/IISWC.2009.5306797.

[40] J. Gómez-Luna, I. El Hajj, L.-W. Chang, V. García-Floreszx, S. G. De Gonzalo, T. B. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 43–54, https://doi.org/10.1109/ISPASS.2017.7975269.

[41] B. A. Burtchell and M. Burtscher, "Characterizing CUDA and OpenMP Synchronization Primitives," in *2024 IEEE International Symposium on Workload Characterization (IISWC)*, 2024, pp. 295–308, https://doi.org/10.1109/IISWC63097.2024.00034.