QuanTaichi: A Compiler for Quantized Simulations

YUANMING HU, Taichi Graphics & MIT CSAIL
JIAFENG LIU, State Key Laboratory of CAD&CG, Zhejiang University
XUANDA YANG, Zhejiang University
MINGKUAN XU, Taichi Graphics & Tsinghua University
YE KUANG, Taichi Graphics
WEIWEI XU, State Key Laboratory of CAD&CG, Zhejiang University
QIANG DAI, Kuaishou Technology
WILLIAM T. FREEMAN, MIT CSAIL
FRÉDO DURAND, MIT CSAIL

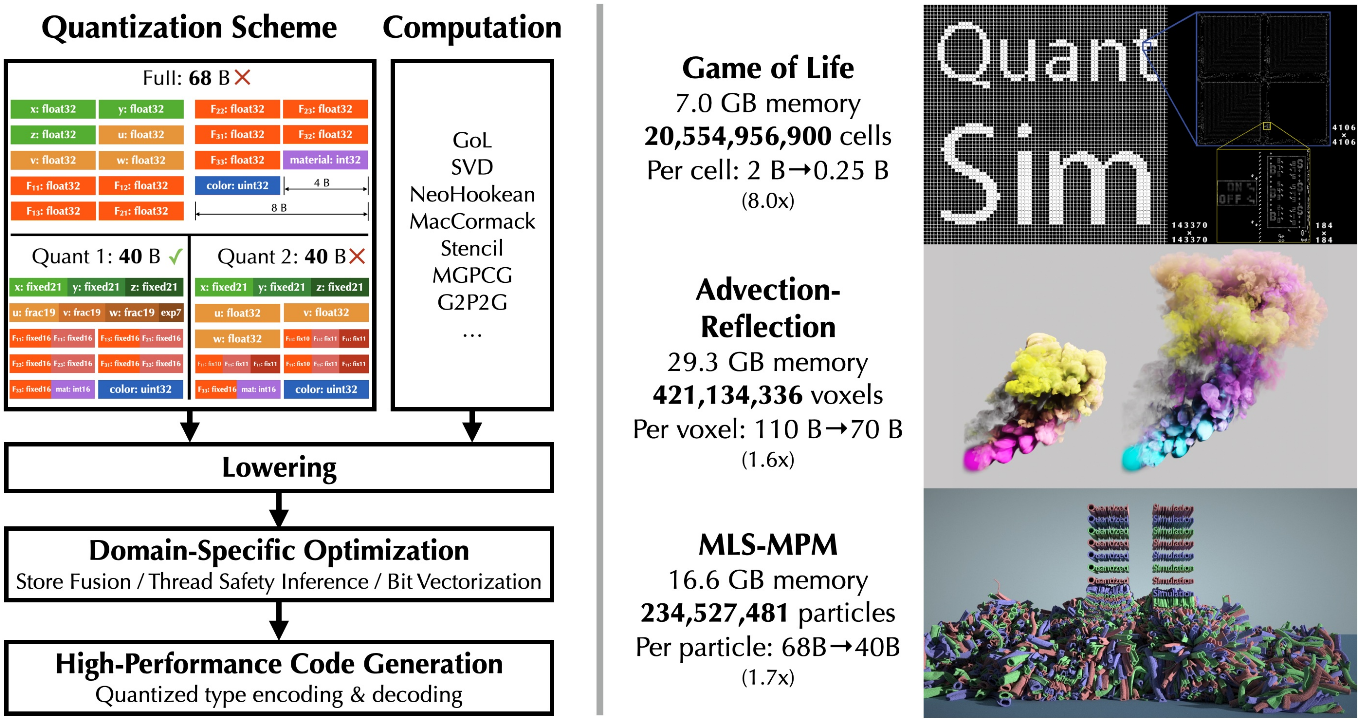


Fig. 1. **Left:** Our system *decouples* quantization schemes from computation, allowing programmers to improve simulation resolutions using lower-precision ("quantized") numerical data types that consume less memory. Programmers can easily switch between different quantization plans to rapidly explore the design space of quantized simulators, to achieve a good balance between numerical precision and memory consumption. Our compiler is in charge of optimization and code generation. **Right:** High-resolution simulation demos built by our system. All demos here run on a single GPU with ≤ 32 GB memory, and each frame takes around one minute.

High-resolution simulations can deliver great visual quality, but they are often limited by available memory, especially on GPUs. We present a compiler for physical simulation that can achieve both high performance and significantly reduced memory costs, by enabling flexible and aggressive *quantization.* Low-precision ("quantized") numerical data types are used and packed to represent simulation states, leading to reduced memory space

Authors' addresses: Yuanming Hu, Taichi Graphics & MIT CSAIL, yuanming@taichi.graphics; Jiafeng Liu, State Key Laboratory of CAD&CG, Zhejiang University, jiafengliu@zju.edu.cn; Xuanda Yang, Zhejiang University, xuandayang@gmail.com; Mingkuan Xu, Taichi Graphics & Tsinghua University, xmk17@mails.tsinghua.edu.cn; Ye Kuang, Taichi Graphics, yekuang@taichi.graphics; Weiwei Xu, State Key Laboratory of CAD&CG, Zhejiang University, xww@cad.zju.edu.cn; Qiang Dai, Kuaishou Technology, daiqiang@kuaishou.com; William T. Freeman, MIT CSAIL, billf@mit.edu; Frédo Durand, MIT CSAIL, fredo@mit.edu.

0730-0301/2021/8-ART182
https://doi.org/10.1145/3450626.3459671

and bandwidth consumption. Quantized simulation allows higher resolution simulation with less memory, which is especially attractive on GPUs. Implementing a quantized simulator that has high performance and packs the data tightly for aggressive storage reduction would be extremely labor-intensive and error-prone using a traditional programming language. To make the creation of quantized simulation practical, we have developed a new set of language abstractions and a compilation system. A suite of tailored domain-specific optimizations ensure quantized simulators often run as fast as the full-precision simulators, despite the overhead of encoding-decoding the packed quantized data types. Our programming language and compiler, based on *Taichi*, allow developers to effortlessly switch between different full-precision and quantized simulators, to explore the full design space of quantization schemes, and ultimately to achieve a good balance between space and precision. The creation of quantized simulation with our system has large benefits in terms of memory consumption and performance, on a variety of hardware, from mobile devices to workstations with high-end GPUs. We can simulate with levels of resolution that were previously only achievable on systems with much more memory, such as multiple GPUs. For example, on a *single* GPU, we can simulate a Game of Life with 20 billion cells (8× compression per pixel), an Eulerian fluid system with 421 million active voxels (1.6× compression per voxel), and a hybrid Eulerian-Lagrangian elastic object simulation with 235 million particles (1.7× compression per particle). At the same time, quantized simulations create physically plausible results. Our quantization techniques are *complementary* to existing acceleration approaches of physical simulation: they can be used in combination with these existing approaches, such as sparse data structures, for even higher scalability and performance.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; • **Computing methodologies** → **Parallel programming languages**; **Physical simulation**.

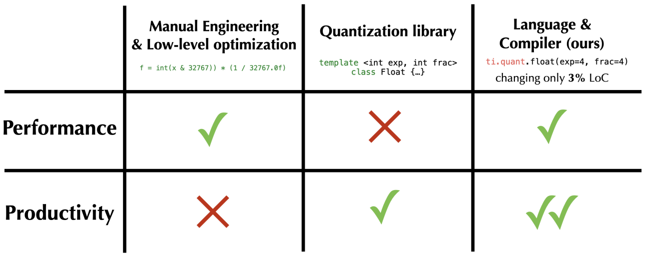
Additional Key Words and Phrases: Domain-specific languages, GPU computing, Quantized computation.

ACM Reference Format:
Yuanming Hu, Jiafeng Liu, Xuanda Yang, Mingkuan Xu, Ye Kuang, Weiwei Xu, Qiang Dai, William T. Freeman, and Frédo Durand. 2021. QuanTaichi: A Compiler for Quantized Simulations. *ACM Trans. Graph.* 40, 4, Article 182 (August 2021), 16 pages. https://doi.org/10.1145/3450626.3459671

1 INTRODUCTION

Computer graphics applications, such as physical simulation, require high resolution for visual quality. Unfortunately, as simulations scale up, they often run out of available memory to store the physical states, especially when running on GPUs with hard memory space limits. Existing techniques that scale up simulations are mostly focused on improving computation performance. The space for improving memory efficiency is largely underexploited.

Fortunately, many simulations do not need standard full-precision IEEE 754 data types, such as `float` and `double` in the C programming language. While these general-purpose floating-point formats are usually the only formats supported by processors for computation, we observe that, for storage, we can use more options, including low-bit integers, truncated fixed-point real numbers, and tuples of floating-point real numbers with shared exponents. This directly motivates us to leverage low-precision data types in simulation to save memory space and bandwidth[1].



	Manual Engineering & Low-level optimization `f = int(x & 32767)) * (1 / 32767.0f)`	Quantization library `template <int exp, int frac> class Float {…}`	Language & Compiler (ours) `ti.quant.float(exp=4, frac=4)` changing only **3%** LoC
Performance	✓	✗	✓
Productivity	✗	✓	✓✓

Fig. 2. Features of different approaches. Our compiler approach aims for both productivity and performance.

While "using fewer bits in data types to save space" sounds like a straightforward idea, doing this robustly and efficiently is a significant challenge. Manually coding programs that operate on low-precision and quantized data types is extremely laborious and error-prone. For example, writing code to decode a low-precision float point number into IEEE 754 `float32` involves numerous low-level bit operations and can easily make simulator code unreadable. Quantized data type libraries may simplify development, but they often result in unsatisfactory performance, since general-purpose compilers may not easily optimize related memory operations that read or write only *part of* a hardware-native integer type such as 32- or 64-bit integers. See Fig. 2 for a high-level comparison between different approaches to implementing quantized simulators.

Moreover, determining the right quantization scheme often requires repeated *trial-and-error*. The most effective way to validate a quantization scheme is to implement the simulator and actually run it. Therefore, *flexibly switching between different quantization schemes* is vitally important for practically developing quantized simulators.

We introduce a language and compiler for *quantized simulation*, where low-precision ("quantized") numerical data types are used to represent simulation states, leading to reduced memory space and bandwidth consumption. In our system, developers write simulators as if they are using a traditional parallel imperative programming language, such as C++ and CUDA. When doing memory-space optimization, they do not modify *any* of the computation code. Instead, they use a simple language to specify numerical value quantization schemes and flexibly explore quantized versions of the simulator. Rapid experiments lead to properly compressed simulation states, improved memory space and bandwidth efficiency. Note that in memory-bound programs the overhead of encoding and decoding quantized data types would be negligible, and quantization may improve performance due to reduced memory bandwidth consumption.

Our tailored programming interface and compiler can greatly simplify the development of quantized simulators. Our system provides language-level abstraction and first-class compiler support for quantized computations and domain-specific optimizations. Programmers can easily specify customized and quantized data types for physical state storage. Since our system *decouples* quantization

[1] Another way to save memory space is data compression, which is particularly useful for streaming. Note that data compression algorithms, such as LZ77 [Ziv and Lempel 1977], rely on a data context ("sliding window") to work, so they may not easily support random accesses. Since many physical simulation tasks heavily rely on random accesses, in this work we do not consider data compression.

schemes from computation kernels, developers can easily experiment with different low-bit data formats, easily achieve a good balance between precision and space via rapid experiments.

We summarize our contributions as follows:

(1) A simple programming interface for *quantized simulation* that provides programmer bit-level control over numerical data types. The numerical data type interface is orthogonal to the actual computation, this allows the programmers to rapidly experiment with different quantization schemes.
(2) A compilation system that automatically generates efficient code for encoding/decoding quantized data types. Our system supports x64, ARM64, CUDA, and Apple Metal backends.
(3) A suite of domain-specific compiler optimizations further improves the memory performance of compiled quantized computation. These optimizations bring 4.10× performance improvement on our microbenchmarks and up to 1.58× on the large-scale GPU simulators.
(4) Systematic evaluations of our system. We demonstrate that our system pushes the resolution of physical simulations to unprecedented resolutions. Under proper quantization, we achieve 8× higher memory efficiency on each Game of Life cell, 1.57× on each Eulerian fluid simulation voxel, and 1.7× on each material point method [Stomakhin et al. 2013] particle. To the best of our knowledge, this is the first time these high-resolution simulations can run on a single GPU. Our system achieves resolution, performance, accuracy, and visual quality simultaneously.

Our system is implemented as an extension to the open-source *Taichi* programming language [Hu et al. 2019] (https://github.com/taichi-dev/taichi). It is now officially a part of Taichi. The evaluation suite is hosted at https://github.com/taichi-dev/quantaichi. Commands to reproduce key experiments are included in this paper.

2 RELATED WORK

2.1 Bit-level compression

Compressed color formats in graphics and image processing. Compressed data types have proven success in graphics. For example, the 16-bit color format "R5G6B5", where 5, 6, and 5 bits are used to store the red, green, and blue channels of a pixel, was widely adopted for legacy LCDs and graphics APIs (e.g., `GL_RGB565` in OpenGL). The 32-bit RGBE format (used in the RADIANCE rendering system [Ward 1994]) used 8 bits for red, green and blue channels each, and a shared 8-bit exponent, providing larger dynamic ranges. In modern production rendering, RGBE formats are used for saving communication bandwidth. For example, Eisenacher et al. [2013] used an RGB9e5 format for path tracing weights.

Quantized neural networks. In deep learning, quantized neural networks (e.g., [Courbariaux et al. 2016; Hubara et al. 2017; Jacob et al. 2018; Kim and Smaragdis 2016]) and specialized hardware (e.g., [Jouppi et al. 2017b]) for them have been studied extensively, to use quantized data formats to improve computation throughput. Instead of using single-precision `float32` data type, recent work explores using low-bit data types such as fixed-point numbers, `int8` and even 1-bit integers for deep neural network training and inference. See [Guo 2018] for a good survey.

Manipulating bits in programming languages. In programming languages such as C/C++, bit-level compression is often implemented using efficient bitwise operators, such as and "`&`", or "`|`", and xor "`^`". Meanwhile, C++ "bit fields", whose behavior is not yet standardized, can sometimes be used for basic bit-level compression of integral types:

```
struct S {
    // 3 bits: value of x
    // 6 bits: value of y
    // 2 bits: value of z
    unsigned char x : 3, y : 6, z :2;
};
```

Although an extensive study has been conducted in quantized computation, a domain-specific system for productively developing high-performance simulators is missing. As a result, a developer who wants quantized computation has to write low-level code that is hard to maintain or resort to handcrafted libraries with performance issues.

2.2 Floating-point formats

IEEE Standard for Floating-Point Arithmetic (IEEE 754)[IEEE 2008] serves as the guideline of floating-point format and computation. Notably, the IEEE 754 single- and double- precision floating-point formats (i.e., `float` and `double` in C), have been the prevalent floating-point formats used in computer graphics and scientific computing. They occupy 32 and 64 bits in memory, respectively. Floating-point bits include a sign bit, a few exponent bits, and significand bits. Since computing applications have different preferences between precision, compute throughput, memory bandwidth and space, many variants of float-point numbers do exist. For example, when higher precision is needed, C provides the non-standard `long double` format that usually comes with 80 bits, and the IEEE 754-2008 revision also defines "quadruple" with 128 bits, and "octuple" with 256 bits, both of which are rarely used. In fact, formats with lower precision are more frequently used, a typical example being the 16-bit `half`-precision format (5 bits for exponent and 10 bits for fraction). Recently the Brain Floating Point (`bfloat16`, 8 bits for exponent and 7 bits for fraction) format has been introduced in Google TPUs [Jouppi et al. 2017a] for deep learning. `half` and `bfloat16` demonstrate interesting trade-offs between dynamic range (number of exponent bits) and accuracy (number of fraction bits, a.k.a. mantissa). See Fig. 3 for an depiction of these floating-point formats. A good survey on mixed-precision arithmetic in numerical methods is [Abdelfattah et al. 2020].

We provide a flexible programming interface for specifying custom floating- and fixed- point data types. Programmers can easily switch between different data types to achieve a good balance between space and precision.

2.3 High-resolution simulations

Work in graphics explores high-resolution simulation on multicore CPUs [Aanjaneya et al. 2017; Liu et al. 2018, 2016; McAdams et al. 2010; Setaluri et al. 2014] and massively parallel GPUs [Gao et al. 2018; Wang et al. 2020; Wu et al. 2015, 2018]. Corresponding sparse

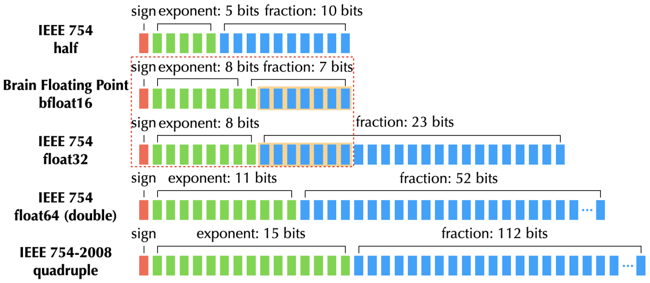


Fig. 3. Various floating-point data types.

data structures are proposed to support the underlying structured grid, often with a certain degree of bit-compression [Hoetzlein 2016; Houston et al. 2006; Museth 2013; Setaluri et al. 2014]. Most of these attempts are based on manual low-level performance engineering using C++ and CUDA. In our work, we use a programming language approach for scaling up simulations, based on Taichi [Hu et al. 2019]. We focus on consumer-level computers with a single GPU for simplicity, yet our techniques can be applied to multi-GPU and multi-node settings as well.

Our system practically pushes the limit of simulation resolutions by alleviating the memory space constraints. For example, with the help of quantization, our single GPU MLS-MPM [Hu et al. 2018] simulation of 235 million particles, has a higher resolution than the existing highest resolution MPM simulation on 8 GPUs (134 million particles, see [Wang et al. 2020]).

2.4 Programming systems for simulation

Engineering high-performance simulation systems can be a time-consuming task since a lot of low-level performance engineering is needed to fully exploit the capabilities of modern computer architecture. Domain-specific languages (DSL) play an important role in improving the productivity of simulation systems. One thread of work provides a high-level graph-based abstraction for meshes that discretize the domain. For example, Liszt [DeVito et al. 2011] focuses on solving PDE on meshes. Several domain-specific languages exist for physical simulation. Simit [Kjolstad et al. 2016] and Ebb [Bernstein et al. 2016] represent simulation problems using sparse linear algebra and relational data models. Insightful discussions on DSLs for simulation can be found in [Bernstein and Kjolstad 2016]. TACO [Chou et al. 2018; Kjolstad et al. 2017] is a sparse linear algebra compiler that can be potentially useful for solving linear systems in simulations.

More closely related to this work is the Taichi programming language [Hu et al. 2019]. Taichi is a DSL with first-class support for sparse data structures, a critical component of modern high-performance physical simulators. Taichi also supports differentiable programming [Hu et al. 2020; Huang et al. 2021], allowing developers to evaluate gradients of physical simulators for machine learning and optimization purposes. We will briefly cover key Taichi features related to this work.

3 TAICHI BACKGROUND

We built our system on top of Taichi [Hu et al. 2019], a data-oriented programming language designed for simulation applications. We extend its type system, computation and data layout intermediate representation (IR), and code generator. Taichi supports spatial sparsity and differentiable programming [Hu et al. 2020], and our quantization system is orthogonal to these existing features of Taichi. Our quantization system can also be implemented based on other imperative programming languages, as long as the corresponding compilers are available for modifications.

We partially reuse the LLVM code generation pipeline in Taichi, for x64 and CUDA on consumer-level desktop computers. Taichi also powers the physics engine on 500 million *mobile devices* in the Kuaishou app, allowing users around the world to generate AR effects augmented with physics. Therefore, we have also implemented our system on the Apple Metal backend and evaluated its performance on an iPhone (section 7.4).

The Taichi language has two parts: a computation language and a data layout language. This decoupling allows users to freely explore data layouts without modifying computational kernels. This work further allows programmers to freely switch data types of numerical values. Existing data types supported in Taichi are `ti.f32/f64` (32/64-bit IEEE 754 floating-point number), `ti.u8/16/32/64` (unsigned integers), and `ti.i8/16/32/64` (signed integers)[2]. In our system, users can flexibly define new data types for more compact storage. See [Hu 2020] for a detailed introduction to the syntax of Taichi.

Computation language. Although the frontend (computation language) of Taichi is embedded in Python, Taichi will inspect the input Python abstract syntax tree and compile them into high-performance executable kernels on parallel devices. Taichi's frontend has a Python-style syntax, enhanced with automatic parallelization, and leads to executables with comparable performance to C++ or CUDA. Two simple Taichi kernels are shown below:

```
@ti.kernel
def saxpy(a: ti.f32):
    for i in x:
        # Parallel for loop over
        # active indices of x
        z[i] = a * x[i] + y[i]

@ti.kernel
def conditional_stencil():
    for i, j in y: # 2D parallel for loop
        if y[i, j] < 0:
            y[i, j] = x[i-1, j] - 2*x[i, j] + x[i+1, j]
```

The Taichi computation language is expressive: programmers can easily write a ray tracer with `if` branching and `while` loops in Taichi.

Data layout language. More closely related to this work is the data layout language and IR. Taichi supports a flexible language to specify data layouts (see [Hu et al. 2019] for more details). Here we only introduce the concept of a Taichi `field`, which are essentially multidimensional dense or sparse tensors. Each element of a field

[2] In this paper we will use a consistent format to refer to these standard data types, for example `f32` or `float32` for 32-bit IEEE 754 floating-point number, `i16` or `int16` for 16-bit signed integer.

can be a scalar (e.g., density), a small vector (e.g., velocity), or a small matrix (e.g., stress tensor). No matter how the internal data layout of a Taichi field is defined, in computational kernels (`@ti.kernel`), field elements are always accessed via an `x[i, j, k]`-style syntax, regardless of its data layout. The following code declares a field of `i32` elements, and then materializes along the `i` and `j` axes, each dimension with 32 and 64 elements.

```
x = ti.field(dtype=ti.i32)
ti.root.dense(ti.ij, (32, 64)).place(x)
# Equivalent to int x[32][64] in C
```

4 QUANTIZED NUMERICAL DATA TYPES FOR SIMULATIONS

In this section, we introduce *quantized data types*, which are often customized to trade precision for memory efficiency. Our system provides both customized integral data types (lossless) and real data types (usually lossy).

4.1 Customized integral types

Integral data types are relatively easy to specify since they are simply series of binary bits. For signed integral types, we pick the classical two's complement data format for negative numbers. We provide to the user the following APIs to create signed (default) and unsigned integral types:

```
i5 = ti.quant.int(bits=5)
u19 = ti.quant.int(bits=19, signed=False)
```

4.2 Custom real types

We offer both fixed-point and floating-point data types. Fixed-point numbers have advantages when their range is strictly bounded in the simulation. For example, when representing x-coordinates of particles, if the boundary condition ensures their values are within $[-2, 2)$, then fixed-point real types can be safely used. They also provide higher precision compared to floating-point types of the same number of bits, since all the bits are used to represent the fraction part. However, the limited dynamic range of fixed-point types can be problematic when handling other physical properties, such as velocity. Therefore, we also offer *floating-point* types, for values with high dynamic ranges.

Fixed-point real numbers. This is the easiest way to represent a real number using integral data types. In fact, we directly reuse a custom integral type plus a real scaling factor to represent custom fixed-point numbers. For example, if the range of the fixed-point number is $[-3.14, 3.14)$ and we have 17 bits, the value can be simply represented by $r = s \times i$, where i is a 17-bit signed integer and $s = 3.14/2^{16}$. Note that there is one sign bit in the underlying integer hence we use 2^{16} instead of 2^{17} as the denominator.

Fixed-point real numbers can be specified as follows:

```
fixed17 = ti.quant.fixed(frac=17, range=3.14)
# Range = [-3.14, 3.14)

ufixed5 = ti.quant.fixed(frac=5, signed=False, range=2)
# Range = [0, 2)
```

When the value is known to be always non-negative, the programmer can use `signed=False` to omit the sign bit and allow the fraction part to have one more bit for higher precision.

Floating-point real numbers. For real numbers with improved dynamic ranges, we allow exponent bits in real number data types:

```
f18  = ti.quant.float(exp=4, frac=14)
uf22 = ti.quant.float(exp=6, frac=16, signed=False)
```

Same as `ti.quant.fixed`, when we know the stored values must be positive, the user can choose to save the sign bit for one more significand bit and get a higher precision. Note that, different from IEEE754 where the sign bit is the leading bit of the format, in our system we include the sign bit as part of the fraction bits. This is an intentional design to simplify the "shared exponent" case, as introduced below.

Shared exponents. In simulations, real values often have physical meanings, and components of a physically meaningful vector typically do not need the same amount of precision, when their absolute values differ a lot. For example, in a 3D velocity vector $\vec{u} = (u, v, w)^T$, if we know the x-component u has a much larger (absolute) value compared to y- and z-components, then we probably do not care about the exact value of v and w. This motivates us to use a "shared exponent" for all components, and leave more bits for components with larger absolute values.

We illustrate the internal organization of the real number types in Fig. 4. Note that for floating-point numbers with independent exponents, we omit the leading "1" following IEEE 754. For shared-exponent cases, we do need a leading "1" to mark the beginning of the digits.

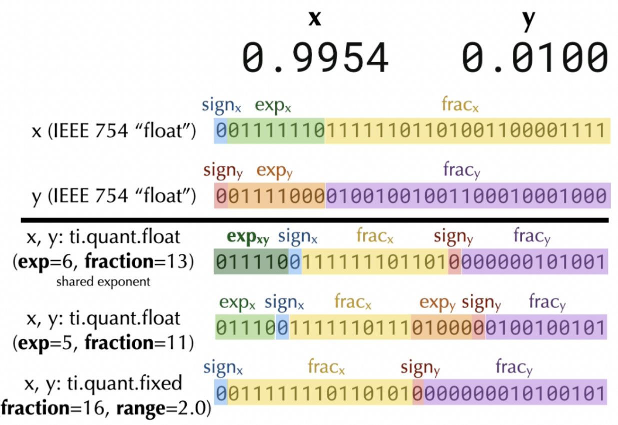


Fig. 4. Real number types representing a 2D vector (x, y). [Reproduce: `python3 misc/visualize_quant_types.py -c [0/1/2]`].

Special cases. Currently, programs are compiled under "fastmath", and we assume no `nan` or `inf` is generated during computation. We do *not* support subnormal numbers, and underflowing real numbers are flushed to zero. Overflowing is undefined behavior. It is possible to implement a "debug mode" that can detect these special cases on the fly, at the cost of performance.

4.3 Compute types

Since most of the custom data types are not natively supported on hardware, we usually have to resort to decoding/encoding mechanisms to translate between representations that are storage-friendly and those that are computation-friendly (Fig. 5, top). This means we have to specify a `compute` type for each custom data type:

```
i21 = ti.quant.int(bit=21, compute=ti.i64)
bfloat16 = ti.quant.float(exp=8, frac=8, compute=ti.f32)
```

By default, the system will use `i32` and `f32` as compute types for integral and real types. A loud compilation error will be emitted if the storage type has more bits than the compute type.

For performance considerations, it may be occasionally profitable to *directly operate on the custom types* without encoding/decoding, especially when the hardware supports related operations (Fig. 5, bottom). We show a few usages of this type of quantization in sections 7.2 and 7.4.

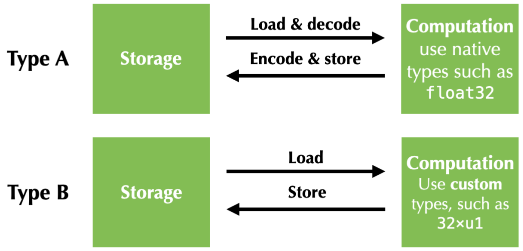


Fig. 5. Two types of quantization investigated in our work. Note that "Type B" quantization may not be always possible and profitable compared to "Type A", since not all custom data types can be directly manipulated by hardware.

4.4 Bit adapters

Loading and storing data with custom types are typically not natively supported on hardware, so we need two types of *bit adapters* to pack custom data types into hardware supported data types with bit width 8, 16, 32, 64:

- **Bit structs** allow users to pack custom data types into hardware-native types. For example, a `u16` bit struct may contain `u5, u6, u5` as components.
- **Bit arrays** pack repeated data types. For example, users can use a 32-bit bit array to store 32 × `u1` types or 8 × `i4` types.

We extend the data layout language of Taichi [Hu et al. 2019]. Bit adapters are extensions to the existing Taichi Structural Node (SNode) system. We refer the readers to [Hu et al. 2019] for more details on SNodes, which are *not* a prerequisite to the remaining of this manuscript.

A *bit struct* serves similarly to a `struct`, but has bit-level granularity. The following code declares two fields of quantized data types, and materialize them into two 2D 4 × 2 arrays:

```
u4 = ti.quant.int(num_bits=4, signed=False)
i12 = ti.quant.int(num_bits=12)

p = ti.field(dtype=u4)
q = ti.field(dtype=u4)

ti.root.dense(ti.ij, (4, 2))
       .bit_struct(num_bits=16)
       .place(p, q)
```

The `p` and `q` fields are laid in an array of structure (AOS) order in memory. Note the containing bit struct of a `(p[i, j], q[i, j])` is 16-bit wide.

Let's now look at a more practical example. In a 3D Eulerian fluid simulation, a voxel may need to store a 3D vector for velocity, and an integer value for "cell category" with three possible values: "source", "Dirichlet boundary", and "Neumann boundary". The developer can then use a single 32-bit `bit_struct` to store all information on a voxel:

```
velocity_component_type =
   ti.quant.float(exp=6, frac=8, compute=ti.f32)
velocity = ti.Vector(3, dtype=velocity_component_type)

# Since there are only three cell categories,
# 2 bits are enough
cell_category_type =
   ti.quant.int(bits=2, signed=False, compute=ti.i32)
cell_category = ti.field(dtype=cell_category_type)

# The bit struct for 512x512x256 voxels
voxel = ti.root.dense(ti.ijk, (512, 512, 256))
          .bit_struct(num_bits=32)

# Place three components of velocity into the voxel,
# and let them share the components.
voxel.place(velocity, shared_exponent=True)
# Place the 2-bit cell category
voxel.place(cell_category)
```

The compression scheme above allows us to store 13 bytes ($4B \times 3 + 1B$) into just 4 bytes. Note that users can still use `velocity` and `cell_category` in the computation code, as if they are `float32` and `uint8`.

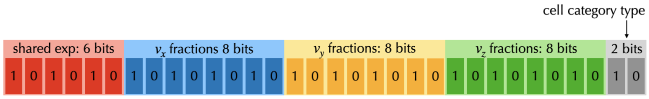


Fig. 6. The `bit_struct` for a 3D smoke simulation voxel. Three components of `velocity` (v_x, v_y, v_z) share one common exponent which is placed in the highest 6 bits. The fractions of `velocity` occupy 24 following bits. The `cell_category` is placed in the lowest 2 bits.

Bit arrays are micro data structures that reinterpret hardware-native data types into arrays of low-bit types. For example, a programmer may want to store 8 × `u4` values in a single `u32` type, to represent bin values of a histogram (Fig. 7):

```
bin_value_type = ti.quant.int(num_bits=4, signed=False)

# The bit array for 512x512 bin values
array = ti.root.dense(ti.ij, (512, 64))
          .bit_array(ti.i, 8, num_bits=32)

# Place the unsigned 4-bit bin value into the array
array.place(bin_value_type)
```

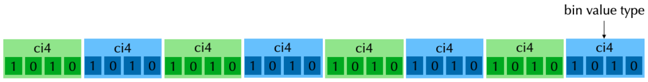


Fig. 7. The bit array for bin values of a histogram. Eight 4-bit components are packed in a single u32.

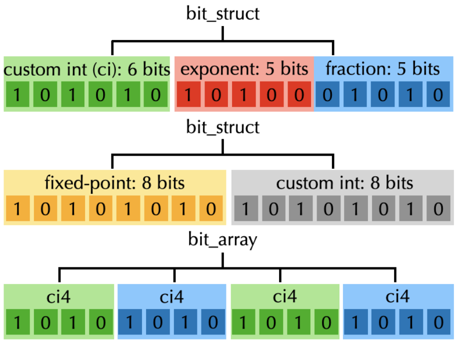


Fig. 8. Illustration of our new type system. Here we show 3 examples of 16-bit bit_structs/bit_arrays. **Top:** One 6-bit custom integer (“ci” in short) and a floating-point number with 5-bit exponent and 5-bit fraction are placed in the bit_struct. **Middle:** There are one 8-bit fixed-point number and one 8-bit custom integer placed in the bit_struct. **Bottom:** A 16-bit bit_array is composed of four 4-bit custom integers.

4.5 Decoupling numerical formats from computation

Similar to Halide [Ragan-Kelley et al. 2012] that decouples scheduling from algorithms, our system decouples numerical formats from computation.

This decoupling has crucial practical benefits in quantized simulations: it is challenging to predict how many bits are required for a numerical type in simulation, and the only way to confirm a quantization scheme does not cause too much truncation error, is to get the simulation running and observe the simulation result, quantitatively or qualitatively. This means the search for the optimal quantization scheme is, unfortunately, repeated trial-and-error. Real simulation code is much more complex, the only way to allows users to rapidly experiment with different quantization schemes is through a system design that separates numerical formats from the computation.

5 CODE GENERATION

In this section, we present a basic implementation of our compilation system, which mechanically translates the operations on custom data types to executable instructions on processors. We leave discussions on possible domain-specific optimizations to section 6.

Our system is implemented based on Taichi, which has a hierarchical static-single assignment (SSA) intermediate representation (IR) system. Our system runs on x64, ARM64, CUDA, and Apple Metal devices. For x64, ARM64 and CUDA, code is just-in-time compiled using LLVM; for Apple Metal, Taichi emits Metal Shading Language source files and then leverages the Metal runtime system to launch GPU kernels.

5.1 Type system

We *replaced* the original type system of Taichi, which only supports primitive data types such as `int32` and `float32`. We developed a new type system in the form of a hierarchical data structure composition system, internally implemented using a shallow tree of data types. See Fig. 8 for illustrations.

5.2 Loading and storing custom integers

Loading a custom integer type from memory needs *addressing* and *decoding*. For *addressing*, we introduce *bit pointers* that precisely represent the starting bit of a custom integer type. *Decoding* is simply zero extension for unsigned integers and signed extension for signed ones. Similarly, custom integer stores need addressing and encoding (truncation).

Bit pointers. Classical pointers only have byte granularity (“byte pointers”, such as `char *` in C), but in our system we want a finer-resolution pointer at bit granularity, denoted as “bit pointer”. A bit pointer has two components:

(1) a classical byte pointer that points to a byte or any other primitive types such as `i32, i64`;
(2) an offset value that specifies a bit-level offset within the primitive type.

See Fig. 9 for an illustration.

Loading from and storing to bit pointers. Hardware-native load/store instructions can only operate at `u8, u16, u32, u64` granularity. Therefore, to load or store data addressed by a bit pointer, we have to “simulate” partial bit loads and stores using hardware-supported memory operations and a series of bit operations (such as shifting) to extract or insert the bits we want.

Loading is relatively easy. We simply load the entire bit struct and use simple bit operations to extract the needed bits. For example, to extract the $[5, 8)$ bits from a 32-bit bit struct s, we can simply let the code generator emit `(s>>5)&7`.

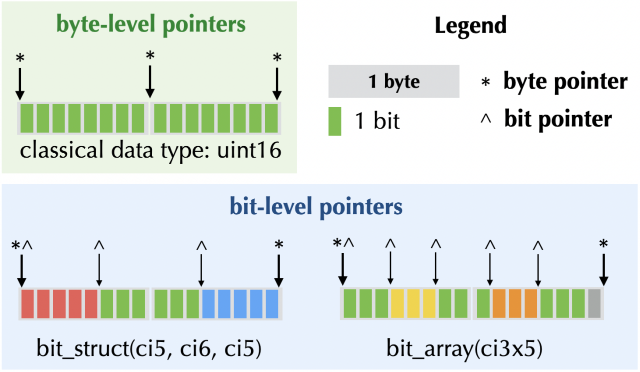


Fig. 9. Byte pointers “*” and bit pointers “^”. **Top left:** A traditional byte pointer only has a byte-level resolution. **Bottom:** Bit pointers have a bit-level resolution, and can easily point to components of bit structs and bit arrays.

Storing to a bit pointer means partially writing of these primitive types. This can be done via a load, a series of bit operations (Fig. 10), and finally a store. It is worth noting that sometimes multiple threads may write to the same bit struct, so we need the following read-modify-write operation to be atomic for thread safety. Note that the atomic read-modify-write (atomicRMW) has to be implemented via a while loop plus atomic compare-and-swap (atomicCAS), and is relatively expensive. We implement corresponding compiler optimizations to avoid atomicRMW as much as possible when thread safety is not a concern (section 6). Our implementation of an atomicRMW add operator is included in the following pseudocode.

```
// 32-bit atomicRMW for partial bits addition
u32 atomicRMW_partial_add32(u32 *ptr, u32 offset, u32
     bits, u32 value) {
    u32 mask = ((~(u32)0) << 32 - bits)) >> (N - offset
         - bits); // create a bit mask
    u32 new_value = 0;
    u32 old_value = *ptr;
    do {
        // read
        old_value = *ptr;
        // modify
        new_value = old_value + (value << offset);
        new_value = (old_value & (~mask)) | (new_value &
              mask);
    }while ( // write via atomicCAS
    !__atomic_compare_exchange(ptr, &old_value, &
         new_value, true,  std::memory_order::
         memory_order_seq_cst, std::memory_order::
         memory_order_seq_cst));
    return old_value;
}
```

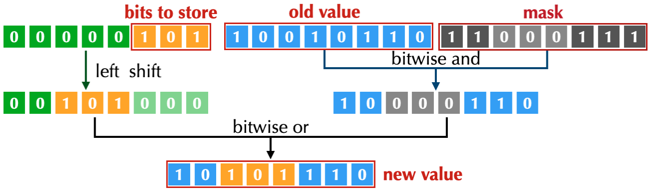


Fig. 10. An illustration of storing bits partially in a bit struct. Here we show the process of inserting a 3-bit custom integer placed in an 8-bit bit struct, using a series of cheap bit-wise operations.

5.3 Efficiently decoding and encoding real numbers

Fixed-point numbers. Since fixed-point numbers are simply integers multiplied by a compile-time constant scaling factor, after we load the underlying integer, the real number can be easily decoded by multiply the integer by the scaling factor. Encoding is the exact reverse process.

On fixed-point atomic adds, we override the decode-compute-encode cycle. We directly scale the real increment into an integer increment, and then use an atomic add on the integer type instead. This allows us to use atomic adds on integer to replace atomic adds on real numbers. In some cases, such as on OpenGL ES and Metal where only integral atomic adds are supported via the API and hardware, using integral instead of floating-point atomic add leads to a significant performance improvement (section 7.4).

Rounding. We adopt the round to *nearest* scheme when casting the scaled real number to an integer. Enforcing the current rounding scheme is critically important and has a direct impact on simulation results. See Fig. 11 for a comparison between different rounding schemes.

Floating-point numbers have exponent and fraction bits, which we handle independently. For the fraction part, we simply adopt an integer truncation with rounding to nearest. The exponent part, however, cannot be simply truncated. It is worth noting that the exponent format of IEEE 754 floating-point numbers does *not* use two's complement for negative numbers[3], hence we need an integer to add operation before we truncate. Overflowing exponent bits are currently treated as undefined behavior, and in practice, we do not find this to be a problem as long as enough exponent bits are reserved.

Subnormal numbers. Our system does not support subnormal floating-point numbers. These numbers are directly treated as zeros when encoding. To simplify and accelerate the decoding/encoding process, we turn on the "flush to zero" (FTZ) flags on CPU and GPUs `float32` data types.

Shared exponents. Floating-point numbers with a shared exponent need special treatment. Suppose now we are encoding a set of `float32` numbers into binary bits. Denote the numbers as exponent-fraction pairs (e_i, f_i), where e_i and f_i can be extracted from the IEEE 754 floating-point compute type via cheap bit-wise operations. The encoding process works as follows:

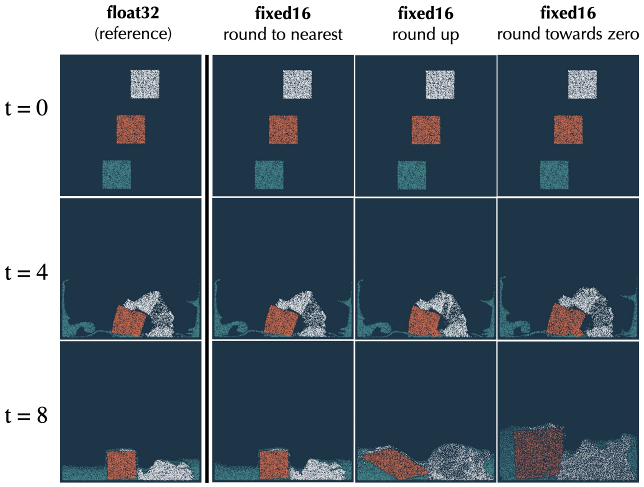


Fig. 11. Rounding scheme matters. In this 2D MLS-MPM experiment [Hu et al. 2018] we use 16-bit fixed-point numbers for the deformation gradient variable on each particle. We stick to the "round to nearest" scheme, which ensures a close approximation to the `float32` reference. Note that the "round up" scheme leads to a shearing artifact to the elastic material (red), and that "round towards zero" results in an expanding volume.

[3] For example, the exponent of the `float32` type has range $[-126, 128)$ instead of $[-128, 128)$. The exponent of the represented floating-point number is 2^{e-127}, where e is the unsigned integer represented by the exponent bits.

(1) Compute the maximum exponent of all floating-point numbers, $e = \max\{e_i\}$. Encode e to the exponent type format of the shared exponent type.
(2) Each number now has an exponent offset $o_i = e - e_i$. For $e_i \neq 0$, we have to prepend the fraction part with e_i zeros. The padding zeros may lead to a precision degradation on values with small e_i, but having a small e_i itself implies the value is less important compared to the largest value sharing the same exponent.
(3) Insert the shared exponent e and fractional bits into the bit struct.

When decoding we need to reconstruct the exponent offset o_i from each fraction bits. o_i can be reconstructed via a call into `__builtin_clz`, which computes the leading zeros of the fraction part [4].

In practice, the decoding/encoding procedures for custom floating-point formats are tricky to get right, since there are a lot of variants such as signed v.s. unsigned, normal v.s. FTZ, shared v.s. non-shared exponents. Our implementations are included in the `taichi/codegen/codegen_llvm_quant.cpp` file.

6 DOMAIN-SPECIFIC OPTIMIZATIONS

Theoretically, everything we have described so far can be implemented via a C++ library, with heavy operator overloading and templated data type classes. A "quantization library" in C++ may not be easy to use, but if zero-cost abstractions are properly used, it would have no performance difference from our domain-specific language. In this section, we show that, apart from usability, our compiler-based approach also has fundamental performance advantages over the library-based approach. This is because our tailored compilation system can conduct domain-specific optimizations that general-purpose compilers (such as `gcc` and `clang`) are not capable of.

Bit structs and bit arrays are first-class citizens of our compilation pipeline. The optimizer can easily analyze and optimize memory operations on their containing data types, leading to higher performance.

Before we detail our automatic optimizations, we stress that, if the data types are implemented via a "quantization library", a programmer can do all these optimizations manually through tedious low-level engineering. However, these optimizations are highly coupled with the underlying data layout and format, and manual optimization locks the code to a particular quantization scheme, leading to a "leaky abstraction". Since seeking the optimal quantization scheme needs repeated trial and error and even making problem-specific adjustments, manually doing the optimizations is not practical.

In this work we introduce three effective optimizations: 1) *bit struct store fusion*, 2) *thread safety inference*, and 3) *bit array vectorization*. The first two optimizations, which we cover below, help to improve performance on a broad range of simulation workloads. The third optimization, *bit array vectorization*, covered in the supplemental document, may create significant performance improvements on computations using 1-bit data types, such as Game of Life and bitwise neural networks [Kim and Smaragdis 2016]. Benchmarks that validate the effectiveness of these optimizations are detailed in section 7.1 and 7.2.

6.1 Bit struct store fusion

In real-world applications, fields in a single bit struct are often accessed together, so it is highly possible that different components of a bit struct get stored by multiple statements in a single kernel. In this case, we can use a single atomicRMW for all the stores into that bit struct.

We introduce a new statement, namely `BitStructStoreStmt(addr, field1, field2, ...)`, in extension to the original general-purpose `GlobalStoreStmt(addr, field)` in Taichi IR, for domain-specific optimizations on bit struct stores.

We also add a few tailored optimization passes. The first pass converts `GlobalStoreStmt` into `BitStructStoreStmt` for easier analysis, and the second pass merges related `BitStructStoreStmt` into a single equivalent `BitStructStoreStmt`. See our supplemental document for a real-world example of this IR optimization.

Note that `BitStructStoreStmt` takes multiple field inputs. In the code generator, we additionally implemented a multi-field version of the partial bit storing procedural. This improves performance since the expensive atomicRMW is now amortized by all the fields to store into that bit struct.

We only optimize bit struct *stores*, because bit struct *loads* (load from an address and then extract the bits) are relatively easy to analyze and optimize by a general-purpose optimizer, such as the LLVM target-independent optimizer we are using. In contract, bit struct stores involve atomicRMW and general-purpose optimizers tend to be conservative regarding optimization. This is likely due to the difficulty of aliasing analysis and the optimizer's lack of thread-safety knowledge. Another reason could be that, in their IR, a single atomicRMW statement would have been lower into quite a few non-trivial basic blocks with complex control flow connecting them.

6.2 Thread safety inference

Take one step further, when there is certainly no data race on the bit struct stores, we can fully replace the atomicRMW with a much cheaper non-atomic version. Our compiler searches for two patterns for this optimization:

Element-wise accesses. In parallel simulators, many operations happen in an "element-wise" manner: each independent thread processes one particle or voxel at a time. Memory loads/stores related to the particle or voxel are then completely free from data races. In this case, we can safely demote the atomicRMW by a non-atomic version. For example, in the following 2D grid boundary velocity projection code,

```
@ti.kernel
def project_velocity():
    for i, j in v:
        if j < 3 and v[i, j][0] <= 0:
            v[i, j][0] = 0
```

[4] The most straightforward way to compute the exponent offset is to use a `std::log(float)` function call. However, this is too expensive in practice. Therefore we heavily use bit-wise operations that are much cheaper.

Table 1. An ablation study on three microbenchmark programs. "Demotion" means atomic demotion, and "fusion" means bit struct store fusion.

Cases	Backend	All optimizations off	Demotion only	Fusion only	All optimizations on	No quantization
Store	x64	688.570 ms	680.164 ms	571.710 ms	338.451 ms	312.925 ms
	CUDA	3.741 ms	3.729 ms	1.874 ms	0.623 ms	0.623 ms
Partial store	x64	882.831 ms	332.785 ms	536.609 ms	309.125 ms	345.032 ms
	CUDA	5.619 ms	2.759 ms	2.888 ms	2.762 ms	2.751 ms
Matmul	x64	272.056 ms	76.015 ms	185.773 ms	76.349 ms	-
	CUDA	9.704 ms	1.705 ms	5.410 ms	1.705 ms	-

our optimizer will safely infer that the store to the first component of the velocity vector at `v[i, j]` (a bit struct) does not need any atomic operation since no other thread will access the same bit struct.

Storing the entire bit struct. Recall that the reason why we need atomicRMW instead of the non-atomic version is to prevent overwriting other parts of the bit struct, which may be written simultaneously by another thread. However, it may be the case that the entire bit struct is stored in a single `BitStructStoreStmt`, then we do not need to worry about overwriting, since this thread is writing to the whole bit struct anyway. We find this pattern to be particularly frequent on particle and grid simulations.

7 APPLICATIONS AND EVALUATIONS

In this section, we showcase the applications of our system and evaluate its performance and accuracy under memory space constraints. We set up three microbenchmarks (each round 50 lines of code) and three large-scale benchmarks (100 to 1500 lines of code). The large-scale simulation statistics are listed in Table 2.

7.1 Microbenchmarks

Optimizations. We set up a suite of microbenchmarks to unit-test our domain-specific optimizations and study their impact on performance. Details and code of the benchmark cases are in the supplemental document.

The numbers obtained with "all optimizations off" mimics the performance of a library-based approach: via template metaprogramming and operator loading, the member functions of quantized data type classes directly emits operations to a general-purpose compiler, LLVM in our case. LLVM will not be able to merge the bit struct stores, since it lacks a high-level understanding of the bit struct stores.

The benchmark results (Table 1) validate that a compiler-based approach is substantially beneficial compared to a more traditional library-based approach. Computing the geometric mean of running time of all the cases on CPUs and GPUs, turning on store fusion leads to 1.43× (x64 CPU)/1.91× (CUDA) speed up, and further turning on atomic demotion leads to another 1.93× (x64 CPU)/2.15× (CUDA) speed up.

Encoding/decoding overheads. We evaluated custom floating-point value encoding and decoding overheads on a classical `saxpy` kernel. We compared the performance of kernels using native `float32` and customized "`float30`" quantized data type, for the `x` and `y` arrays. On GPU, the quantized version is only 1% slower than the native version. We believe this is because the processor is waiting for memory accesses instead of waiting for the additional computation due to encoding/decoding. On CPU, however, the quantized version is 20% slower than the native version. The larger slow down on CPU than GPU, is likely because our non-vectorized CPU program is more sensitive to additional computations than the GPU program, since relatively more memory bandwidth are available for each non-vectorized CPU thread than a GPU thread. Note that in `saxpy`, computation is almost minimized, and the majority of computation would be decoding and encoding: this is an extreme-case analysis, especially on CPUs.

7.2 Game of Life

We first test our system on the classical Conway's Game of Life, a 2D grid-based simulation that is extremely simple to code but computational hungry at a high resolution. Each cell (i, j) can have two states, either *live* $(l_{i,j} = 1)$ or *dead* $(l_{i,j} = 0)$. The cell states follow a set of simple evolution rules, depending on its 3×3 neighborhood:

- **Birth**: each dead cell with *exactly* three neighbors becomes a live cell;
- **Survival**: each live cell with two or three live neighbors continues to be a live cell;
- **Overpopulation**: each live cell with four or more live neighbors dies;
- **Isolation**: each live cell with zero or one live neighbor dies.

Essentially, the next-step cell state $l'_{i,j}$ is defined as

$$l'_{i,j} = f\left(\sum_{x=i-1}^{i+1}\sum_{y=j-1}^{j+1} l_{x,y}\right),$$

where f maps the number of active neighbors into the cell state of the next time step.

Storage. We created two copies of the grid, namely l' and l, and iterate back and forth. We use two hierarchical grids to store the current and next frames. Using bit arrays that pack 32 `u1` (1-bit unsigned integer) type into a single `u32`, each cell takes only 1 bit in each grid, leading to a 2 bit/cell total storage footprint. Note that in traditional languages such as C, programmers will have to use the `char` (`u8`) type for each cell, unless they manually pack/unpack the states. In our system, users can effortlessly improve storage efficiency by 8×, without any modification of the computation code.

Table 2. Statistics of our large-scale demos. The Game of Life demo adopts more steps per frame as the camera zooms out, and the numbers here are the maximum steps per frame after the camera fully zooms out. For the memory allocated, note that the memory allocator of Taichi maintains auxiliary data structure and enforces padding, leading to more memory allocated than actual used, especially on the Game of Life and MLS-MPM demos. See our video for more visual results.

Solver	Demo	Frames	Steps/ frame	Seconds/ frame	Δt	Grid	Active cells	Particles	Byte/element		GPU	GB allocated
									Full	Quant		
Game of Life	Fig. 1	720	1024	30.2	-	$131,072^2$	$10,487,808,100$	-	2	0.25	3080	4.0
	Fig. 12	720	1024	53.0	-	$262,144^2$	$20,554,956,900$	-	2	0.25	3080	7.0
Eulerian Fluids	Fig. 14 (top)	300	1	68.8	3×10^{-2}	$2,048^3$	$421,134,336$	-	110	70	V100	29.3
	Fig. 14 (bottom)	280	1	58.3	3×10^{-2}	$2,048^3$	$421,134,336$	-	110	70	V100	29.7
MLS-MPM	Fig. 17	240	129	76.2	7.8×10^{-5}	$4,096^3$	$72,392,832$	234,527,481	68	40	3090	16.6

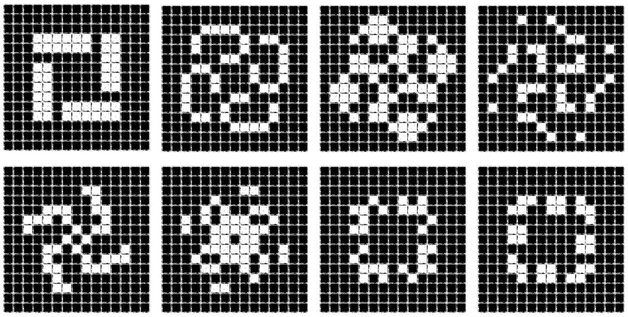


Fig. 12. The evolution of the 15 × 15 galaxy pattern. Each cell is a 2048^2 OTCA metapixel, and a metapixel step is 32768 Game of Life time steps. Each metapixel evolves following the Game of Life rules when zoomed out. [Reproduce: cd gol && python3 galaxy.py -a [cpu/cuda] -o output]

Initialization. To demonstrate the capability of our Game of Life simulator, we initialize the grid using tiled OTCA metapixels[5]. An OTCA metapixel consists of 2048 × 2048 cells, and it has an interesting behavior: when looking at a distance, the whole 2048 × 2048 metapixel behaves like a single 1 × 1 Game of Life cell (Fig. 12). We also set up a Game of Life pattern with words "Quant Sim" using 70 × 70 OTCA metapixels, reaching over 20 billion cells in a single simulation (Fig. 1). For a high-resolution visualization of this demo, please refer to our video and supplemental material.

The effectiveness of bit vectorization. In each time step of Game of Life, a cell loads its 3×3 neighborhood states from the old state buffer and stores its new state to a new state buffer. Bit array vectorization improves performance by simultaneously handling 32 cells in a single thread. We compare the running time between three different implementations:

(1) QuickLife from Golly[6], a fast algorithm leveraging sparsity in Game of Life. The implementation in Golly is highly optimized on CPUs.
(2) Ours, without bit vectorization. Since this implementation only utilizes 1/32 of the bitwidth and leads to excessive atomicRMW, we do not expect it to deliver satisfactory performance.

Table 3. Per step running time of three different implementations, on CPUs and GPUs. The benchmark is initialized using a random pattern with 41690^2 active cells with 50% live cells. We run CPU benchmarks on an Intel i7 processor (six cores each at 2.6 GHz) and 16GB of memory. We run GPU benchmarks on an NVIDIA RTX 3080 GPU with 10GB of GPU memory.

Implementation	Time per step (CPU)	Time per step (GPU)
QuickLife	318.5 ms	N/A
Non-vectorized brute-force	9313.4 ms	546.0 ms
Vectorized brute-force	417.2 ms	3.4 ms

(3) Ours, with bit vectorization. This is essentially (2) with a extra `ti.bit_vectorize(32)` pragma. Our compilation passes do the vectorization job automatically.

As expected, the performance of our non-vectorized implementation is much worse than QuickLife on CPU, while the same algorithm achieves comparable performance with the significant optimization of bit vectorization on CPU. The effectiveness of bit vectorization is further verified on GPU where the vectorized version is more than 150× faster than its non-vectorized counterpart (Table 3).

Our implementations do not utilize spatial sparsity yet, and compared to QuickLife it does more work. This partially explains why our vectorized simulator is 31% slower than QuickLife on CPUs.

7.3 Eulerian fluid simulation

We developed a sparse-grid-based advection-reflection [Zehnder et al. 2018] fluid solver to evaluate our system on grid-based physical simulators.

For advection, we use the MacCormack scheme [Selle et al. 2008], with RK3 path integration. For projection ("reflection"), we use multigrid preconditioned conjugate gradients (MGPCG) [McAdams et al. 2010] to solve the Poisson problem. We follow the MGPCG solver design in [Hu et al. 2019]. We use two-level pointer grids (`ti.root.pointer(ti.ijk, ...).pointer(ti.ijk, ...).dense(ti.ijk, ...).place(...)`) for each level of the grid hierarchy.

The majority of memory consumption comes from the top level of the grid hierarchy. This is because the second level only has 1/8 voxels due to multigrid coarsening. Also note that physical properties such as dye density (R, G, B) and velocity (u, v, w) only exist at the top level, and we need to store multiple copies of them for the MacCormack advection scheme.

[5] https://www.conwaylife.com/wiki/OTCA_metapixel
[6] http://golly.sourceforge.net/

Quantization scheme. We focus our quantization on the advection solver, since it costs the majority of memory space (84 out of 110 B). By representing each component of velocity (u, v, w) using a 21-bit fixed-point number, we pack them in a single 64-bit bit struct. Also, we managed to pack three channels of dye density in a 32-bit bit struct by using floats with shared exponents: we use 9 bits for fractions and 5 bits for exponent, which adds up to 32 bits. Since we adopt a MacCormack advection solver, we use three bit structs to store v_t, v_{t+1} and an auxiliary $\bar{v}_{t+1}$ respectively. Similarly, the dye density needs three bit structs. Note that another bit struct is used as the velocity after the reflection operator being applied. In this way, each voxel occupies 44 bytes, while using float32 needs 84 bytes (Fig. 13).

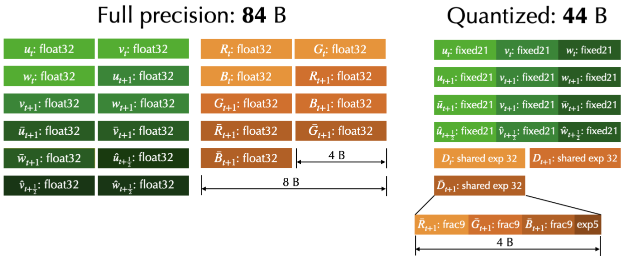


Fig. 13. Eulerian smoke simulation quantization scheme. For each voxel, we pack velocity (u, v, w) in a 64-bit bit struct by three 21-bit fixed-point numbers. Dye density (R, G, B) is represented by 32-bit shared-exponent numbers with 5 bits for exponent and 9 bits for fraction. In this way, our smoke advection memory consumption is reduced by 48% from 84 bytes to 44 bytes. Considering 26 B from the MGPCG solver, the memory consumption per voxel for the entire fluid solver is reduced from 110B to 70B, a 1.57× improvement.

The effectiveness of shared exponents. To prove the effectiveness of shared exponents, we compare shared exponent, non-shared exponent and fixed-point using a 2D smoke simulation. All of these experiments use one 32-bit bit struct to store dye density (R, G, B) channels. In contrast, the uncompressed `float32` reference uses 96-bits per voxel.

For the simulation with shared exponent, we use 5 bits for exponent and 9 bits for fraction. The non-shared exponent simulation also uses 5 bits for exponent, but only 5 bits for fractions for each channel. The fixed-point simulation uses 10-bit fixed-point numbers per channel. .

Note that there is an exponential decay of density in the simulation, and ultimately density on each cell will decay into zero.To quantitatively study the preciseness of the decaying behavior using different data formats, we sum up all the cell density (Table 4) and find that among all quantization schemes, the shared exponent version has the closest total density compared to the `float32` reference simulation. The non-shared exponent floating-point format has fewer fraction bits, leading to a lower precision. The fixed-point format has a smaller dynamic range compared to the shared exponent floating-point format. This indicates that shared-exponent floating-point formats achieve both good precision and dynamic range, compared to non-shared exponent floating-point formats and fixed-point formats. For a visual comparison and analysis, please see our supplemental document.

Table 4. Total density comparison. Note that since the decaying factor is a constant on all pixels and channels, the total density is a predictable value over the simulation, regardless of the turbulent nature of the fluid simulation. This experiment runs on a GTX 1080 Ti GPU with 11 GB memory. [Reproduce: cd eulerian_fluid && python3 run_shared_exp.py –data-type [0/1/2/3] -o outputs]

Data Type	Total density
float32	15425.828
shared exponent: exp5 + frac9	17072.586
fixed-point: 10	24662.654
non-shared exponent: exp5 + frac5	41368.633

Table 5. Performance comparison of Eulerian fluid advection. This experiment also runs on a GTX 1080 Ti GPU with 11 GB memory. [Reproduce: cd eulerian_fluid && python3 run_benchmark.py -e [0/1/2]]

Data Type		Time
Velocity	Dye density	
float32	float32	25.052 s
shared exponent: exp5 + frac9	fixed-point 10	31.776 s
fixed-point 10	fixed-point 10	24.760 s

Performance of quantized simulations. Finally, we compare the performance of our quantized simulator against the `float32` reference implementation. We find the quantization scheme using shared exponent to be roughly 27% slower than the full-precision version, likely because encoding/decoding floats with shared exponent takes some additional computation. Interestingly, the all-fixed-point version is slightly faster than the reference version, despite needing more floating-point multiplications to encode/decode. Since the advection kernel is memory-bound, the quantized version consumes less memory bandwidth, hence running faster.

7.4 Moving Least Squares Material Point Method

To test our system on hybrid Lagrangian-Eulerian methods where both particles and grids are used, we implemented the Moving Least Squares Material Point Method [Hu et al. 2018] with G2P2G transfer [Wang et al. 2020].

In MPM simulation, storing per-particle data is very memory- and bandwidth-consuming. Note that in MLS-MPM over 70% storage is for particles. A 4096^3 background sparse grid is used, leveraging the first-class spatially sparse data structure support in the original Taichi system. We use a 4^3 leaf block size.

When simulating elastic objects, we store position, velocity, and deformation gradients on each particle. After a few trial-and-error, we end up with a quantization scheme that compressed 68 B particle attributes to 40 B, a 1.7× improvement. Note that in MPM we also need to store the grids and other acceleration structures such as the particle list, which is not quantized here. The overall memory efficiency improvement is 1.3×. See Fig. 15 for more details on

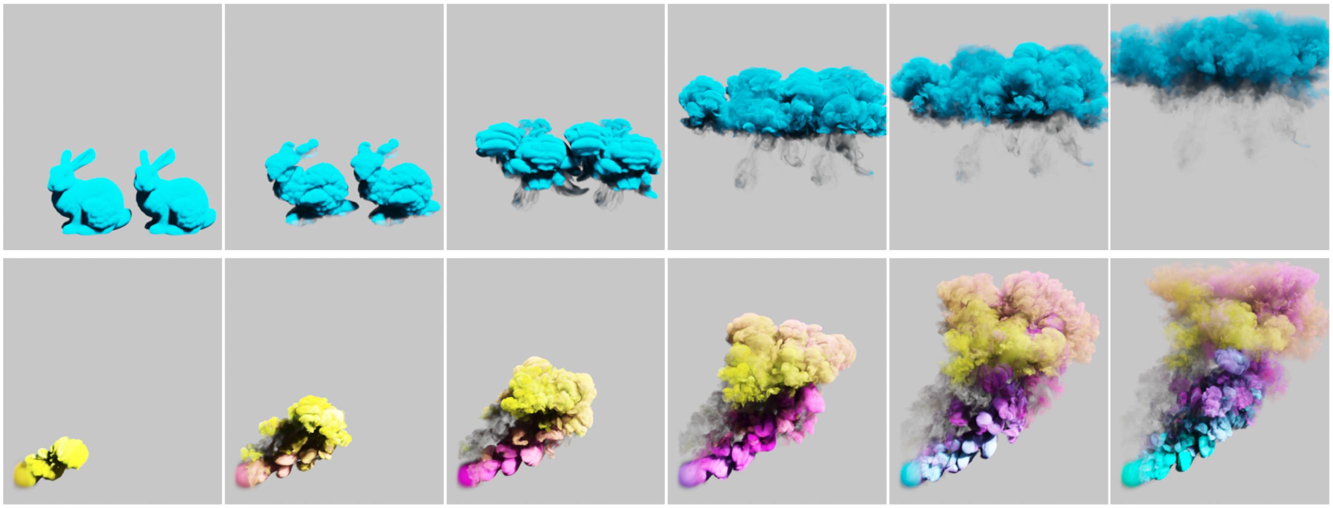


Fig. 14. Two quantized smoke simulations. We use the quantization scheme described in section 7.3. Both simulation run on a 2048^3 sparse grid with 421M active voxels. **Top**: Smoke initialized from two bunny meshes. **Bottom**: Smoke emitting from a spherical source. [Reproduce: cd eulerian_fluid && python3 run.py –demo [0/1] -o outputs]

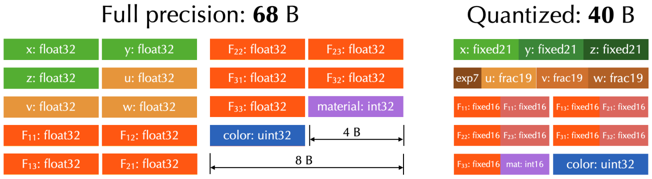


Fig. 15. MLS-MPM particle attribute quantization scheme. For each particle, we store position (x, y, z) using 21-bit fixed-point numbers, velocity (u, v, w) using floating-point numbers with 17 fraction bits a shared 7-bit exponent. For deformation gradient $F_{3\times3}$, we use 16-bit fixed-point numbers. We also store material and color information. This brings down a particle storage footprint from 68 bytes to 40 bytes (1.7× fewer).

the quantization scheme. A 235M-particle visual demo is shown in Fig. 17.

Performance. In a GPU MLS-MPM simulator, the performance is limited by memory bandwidth and available FLOPs. Using quantization increases the amount of computation due to the need for encoding/decode, yet at the same time it reduces the memory bandwidth. We conduct a systematic performance study on our system, scanning all possible combinations of quantization, optimization, and particle-grid transfer scheme (separate P2G/G2P and fused G2P2G [Wang et al. 2020])[7]. Results are listed in Table 6. Interestingly, we find our quantized simulator runs 1.03× (G2P2G)/1.14×(P2G+G2P) faster than the full-precision simulator, likely because the quantized simulator saves memory bandwidth. We get higher speed up on the P2G+G2P transfer scheme, because this version with two kernels needs more memory bandwidth. Our domain-specific optimizations leads to 1.09× (G2P2G) and 1.80×(P2G+G2P) higher performance on the quantized simulator. Compared to Wang et al. [2020], a CUDA MLS-MPM solver heavily engineered towards performance, our quantized version is 3× slower, most likely due to the fact that we did not implement the AOSOA data layout optimization. However, regarding memory consumption our system is 5.7× more efficient (1.4GB ours v.s. 8GB [Wang et al. 2020]). Note that the memory efficiency gap is a combination of our quantization scheme and the fact that [Wang et al. 2020] is optimized towards performance instead of memory.

Table 6. Domain-specific optimization ablation study on the MLS-MPM benchmark, with and without the G2P2G optimization [Wang et al. 2020]. Results are collected on a RTX 3090 GPU. We seed 16,777,216 particles in a 256^3 domain. [Reproduce: `python3 quan_mpm_benchmark.py [--no-ad] [--no-fusion] [--no-quant] [--no-g2p2g]`]

Optimizations	G2P2G Time (s)	P2G+G2P Time (s)
No optimization	16.52	28.43
Store fusion only	15.99	17.19
Atomic demotion only	16.36	16.69
All optimizations on	15.09	15.77
No quantization	15.57	17.96

Quantization on mobile devices. Since mobile devices have relatively limited computing power and a strong need for real-time response, typically only small-scale simulations run there and storage is not really an issue. Surprisingly, we still find using quantized data types on the background grid to be beneficial: since mobile GPUs usually only have high-performance native atomicAdd support for `i32` but not for `f32`, using `ti.quant.fixed(fraction=32)` on

[7] "P2G" means "scattering particle data to grid" and "G2P" means "gather grid data to particles". "G2P2G" is a fused version, where grid data are first gathered to particles and then scattered to the next-time-step grid.

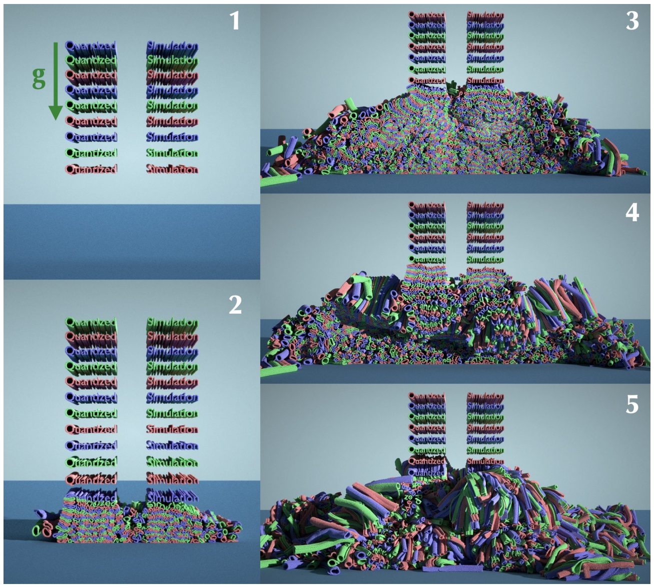


Fig. 17. A 235M-particle MLS-MPM simulation. **(1)** 4,693 elastic tubes fall down, **(2, 3)** form an interesting mountain structure, and **(4, 5)** ultimately collapse due to instability. Note the tubes are lengthy along the camera direction. [Reproduce: cd mls_mpm && python3 -m demos.demo_quantized_simulation_letters -o outputs]

the grids converts software-emulated `f32` atomicAdd to hardware-native `i32` atomicAdd significantly improves P2G performance in our MLS-MPM program on an iPhone XS (Fig. 16). See our supplemental video for more visual comparisons.

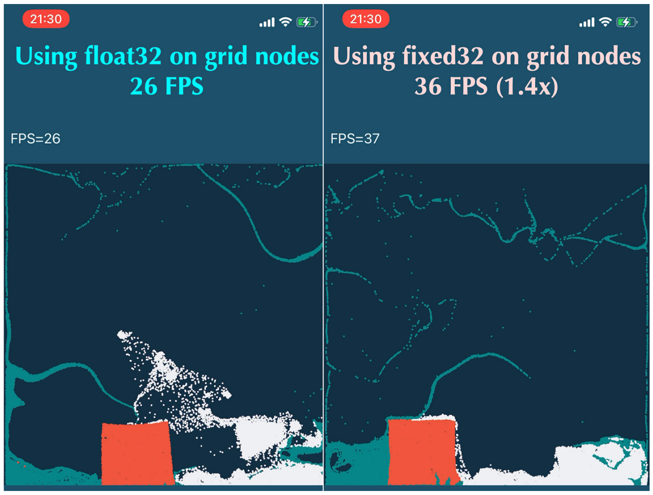


Fig. 16. A benchmark of 36,000 MLS-MPM particles with 256 × 256 grid nodes, on an Apple iPhone Xs. Using our `ti.quant.fixed(fraction=32)` types on the grid nodes (right) improves FPS by 1.4× compared to native `float32` (left). Note that the `fixed32` version is captured at a later simulation stage since it has a higher FPS.

Table 7. Quantitative error analysis of each scheme against `float64`. The simulation using each scheme runs five times. We report the average value (and standard deviation) of MSEs collected for each scheme.

Quantization Scheme	Particle position MSE
float32 reference	$2.69(\pm0.6) \times 10^{-11}$
Quantize position	$1.68(\pm0.3) \times 10^{-8}$
Quantize position & velocity	$1.79(\pm0.01) \times 10^{-3}$
Quantize position, velocity & deformation gradient	$1.87(\pm0.01) \times 10^{-3}$

Quantitative quantization error analysis. To quantitatively analyze quantization errors, we conduct a 2D elastic object collision experiment using MLS-MPM. We used double precision (`float64`) as the reference solution, and computed the mean squared error (MSE) of corresponding particle positions in the `float64` and quantized simulations. The simulation lasts 200 frames. For quantized simulations, we used 20-bit fixed-point numbers for particle positions, and floating-point numbers with 10 fraction bits and 6 exponents bits for particle velocity. Four 16-bit fixed-point numbers are used for particle deformation gradients. We compare the error of four schemes: 1) full precision (`float32`), 2) quantize position only, 3) quantize position & velocity, and 4) quantize position, velocity & deformation gradient. The results are shown in Table 7 and Figure 18.

Clearly, MSEs go up as more physical properties are quantized. In long-running simulations like this, tiny quantization errors accumulate and may result in a visually noticeable difference in the final state. Still, the simulations remain physically plausible. Interestingly, we find that using `fixed16` instead of `float32` for deformation gradients does not introduce more quantization error, leading to a 2× higher memory efficiency on the most memory-consuming physical property on each particle at no cost of precision. See our supplemental video for more details.

7.5 Visual comparisons

To investigate how much visual difference quantization injects into the simulator, we collected five groups of simulation videos. Each group has three videos: two of them are generated using full-precision `float32` simulators [8], and one is generated using quantized simulators. Visually, after quantization the simulations remain physically plausible. Five cases are presented in Fig. 19. See our supplemental document and video for more details on visual comparisons.

7.6 Discussions

Productivity. Thanks to our decoupling of numerical data formats from numerical computation, the amount of code modified to transform a traditional full-precision physical simulator into a quantized simulator is no bigger than 3% of the total solver code. For example,

[8] Two runs of the same full-precision simulations may lead to slightly different results. This is because parallel floating-point computations (e.g., parallel reduction of floating-point numbers) intrinsically have fluctuations. Note that floating-point add is not associative and the result depends on the CPU/GPU thread scheduler.

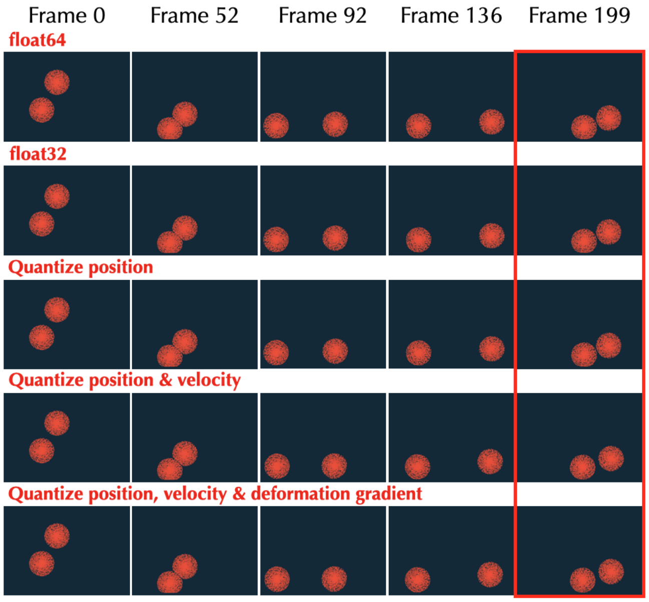

Frame 0
Frame 52
Frame 92
Frame 136
Frame 199
float64
float32
Quantize position
Quantize position & velocity
Quantize position, velocity & deformation gradient

Fig. 18. Visual results of our quantitative error analysis experiments. Two elastic balls fall under gravity. Our quantized simulation is physically plausible, despite the errors shown in Table 7.

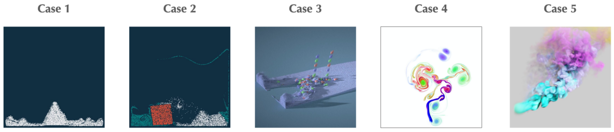


Fig. 19. Five visual comparison cases used to qualitatively compare quantized and full-precision simulators.

the whole MLS-MPM solver has roughly 1000 lines of code, while specifying a single quantization scheme takes only 30 lines of code (3%). To quantize the 900-line fluid simulator, no more than 20 (2.2%) lines of code are added.

Failure cases. It is not uncommon that using data types that have too low precision or dynamic range leads to simulation artifacts. Fortunately, our system which allows rapid trial-and-error, and allows programmers to simply use more bits in this case. For example, we found using 16-bit fixed-point numbers for fluid volume ratio $J = V_t/V_0$ [Tampubolon et al. 2017] leads to a clear volume gain. This can be easily fixed by letting J have 23 bits instead of 16 (Fig. 20).

8 CONCLUSION

In order to make quantized simulation practically programmable, we have developed a tailored programming interface, compilation system, and domain-specific optimizations. Our system is orthogonal to many of the existing work to accelerating simulations, including sparse data structures [Hoetzlein 2016; Hu et al. 2019; Museth 2013; Setaluri et al. 2014]. Using our system, programmers can effortlessly switch between different quantization schemes, leading to 1.57 ~ 8.00× memory space efficiency improvement. A user study shows that 3D quantized simulation results are indistinguishable from full-precision ones. Our system is performant and easy to use: by modifying no more than 3% of the simulator code, a developer can quantize a MLS-MPM or Eulerian fluid simulator, running at a comparable speed to the full-precision version.

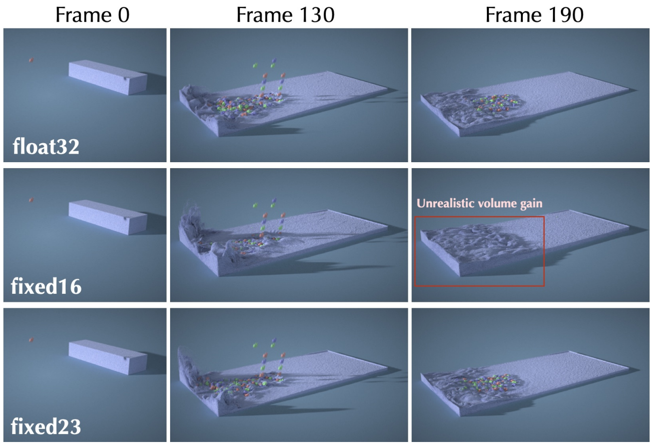


Fig. 20. A 12M-particle MLS-MPM fluid simulation with volume ratio J on each particle quantized. Note that using `fixed16` leads to an unrealistic volume gain compared to the `float32` reference. The programmer can easily fix this by adding 7 more bits to the variable and use `fixed23` instead.

Future work. Currently, programmers still have to manually experiment with different quantization schemes. It would be helpful to have a system that automatically figures out suitable quantization schemes. Regarding engineering, adding a debugging system that detects overflowing of fixed- and floating- point numbers can help programmers more easily diagnose issues in a quantization scheme. Although our discussions are focused on a shared-memory environment, our quantization compiler can also help multi-GPU and distributed memory computation, since quantized physical states can significantly reduce communication overhead in those scenarios. Combining our system with emerging low-precision computation hardware, such as NVIDIA TensorCores, is a meaningful future direction.

ACKNOWLEDGMENTS

We would like to thank Bo Zhu, Sylvain Paris, and Paris Smaragdis for early inputs to this work. This work is supported by the National Science Foundation under Cooperative Agreement PHY-2019786 (The NSF AI Institute for Artificial Intelligence and Fundamental Interactions, http://iaifi.org/). This work is also partially supported by the Toyota Research Institute (TRI). However, this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity. Weiwei Xu is partially supported by the NSFC grant (No. 61732016). Yuanming Hu is partly supported by research fellowships from Adobe and Facebook. Yuanming Hu would like to thank NVIDIA for providing an RTX 3090 GPU.

REFERENCES

Mridul Aanjaneya, Ming Gao, Haixiang Liu, Christopher Batty, and Eftychios Sifakis. 2017. Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–12.

Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, Erin Carson, Terry Cojean, Jack Dongarra, Mark Gates, Thomas Grützmacher, Nicholas J Higham, Sherry Li, et al. 2020. A survey of numerical methods utilizing mixed precision arithmetic. *arXiv preprint arXiv:2007.06674* (2020).

Gilbert Louis Bernstein and Fredrik Kjolstad. 2016. Perspectives: Why New Programming Languages for Simulation? *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–3.

Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Trans. Graph.* 35, 2 (2016), 21:1–21:12.

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. 2011. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 9.

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted deferred shading for production path tracing. In *Computer Graphics Forum*, Vol. 32. Wiley Online Library, 125–132.

Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana-Tampubolon, Eftychios Sifakis, Yuksel Cem, and Chenfanfu Jiang. 2018. GPU Optimization of Material Point Methods. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 32, 4 (2018), 102.

Yunhui Guo. 2018. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752* (2018).

Rama Karl Hoetzlein. 2016. GVDB: Raytracing sparse voxel database structures on the GPU. In *Proceedings of High Performance Graphics*. Eurographics Association, 109–117.

Ben Houston, Michael B Nielsen, Christopher Batty, Ola Nilsson, and Ken Museth. 2006. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Transactions on Graphics (TOG)* 25, 1 (2006), 151–175.

Yuanming Hu. 2020. The Taichi programming language. In *ACM SIGGRAPH 2020 Courses*. 1–50.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *ICLR* (2020).

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. 2018. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 4 (2018), 150.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 201.

Zhiao Huang, Yuanming Hu, Tao Du, Siyuan Zhou, Hao Su, Joshua B Tenenbaum, and Chuang Gan. 2021. PlasticineLab: A Soft-Body Manipulation Benchmark with Differentiable Physics. *ICLR* (2021).

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017b. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017a. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 1–12. https://doi.org/10.1145/3140659.3080246

Minje Kim and Paris Smaragdis. 2016. Bitwise neural networks. *arXiv preprint arXiv:1601.06071* (2016).

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A language for physical simulation. *ACM Trans. Graph.* 35, 2 (2016), 20:1–20:21.

Haixiang Liu, Yuanming Hu, Bo Zhu, Wojciech Matusik, and Eftychios Sifakis. 2018. Narrow-band Topology Optimization on a Sparsely Populated Grid. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 251:1–251:14.

Haixiang Liu, Nathan Mitchell, Mridul Aanjaneya, and Eftychios Sifakis. 2016. A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Transactions on Graphics (TOG)* 35, 6 (2016), 1–12.

Aleka McAdams, Eftychios Sifakis, and Joseph Teran. 2010. A parallel multigrid Poisson solver for fluids simulation on large grids. In *Symposium on Computer Animation*. ACM/Eurographics Association, 65–74.

Ken Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.* 32, 3 (2013), 27.

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 1–12.

Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. 2008. An unconditionally stable MacCormack method. *Journal of Scientific Computing* 35, 2-3 (2008), 350–371.

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 33, 6 (2014), 205.

Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. 2013. A material point method for snow simulation. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 102.

Andre Pradhana Tampubolon, Theodore Gast, Gergely Klár, Chuyuan Fu, Joseph Teran, Chenfanfu Jiang, and Ken Museth. 2017. Multi-species simulation of porous sand and water mixtures. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–11.

Xinlei Wang, Yuxing Qiu, Stuart R Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, and Chenfanfu Jiang. 2020. A massively parallel and scalable multi-cpu material point method. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 30–1.

Gregory J Ward. 1994. The RADIANCE lighting simulation and rendering system. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 459–472.

Jun Wu, Christian Dick, and Rüdiger Westermann. 2015. A system for high-resolution topology optimization. *IEEE transactions on visualization and computer graphics* 22, 3 (2015), 1195–1208.

Kui Wu, Nghia Truong, Cem Yuksel, and Rama Hoetzlein. 2018. Fast fluid simulations with sparse volumes on the GPU. In *Computer Graphics Forum (Proc. Eurographics)*, Vol. 37. Wiley Online Library, 157–167.

Jonas Zehnder, Rahul Narain, and Bernhard Thomaszewski. 2018. An advection-reflection solver for detail-preserving fluid simulation. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–8.

Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.