# QuanTaichi: A Compiler for Quantized Simulations (Supplemental Document)

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

## 1 BIT STRUCT STORE FUSION EXAMPLE

Below is a simple kernel and its IR before and after bit struct store fusion.

```
// Data layout
x = ti.field(dtype=ti.quant.int(16, True))
y = ti.field(dtype=ti.quant.int(16, True))
ti.root.dense(ti.i, 1024).bit_struct(num_bits=32).place(
    x, y)

// Kernel
@ti.kernel
def store():
    for i in range(1024):
        x[i] = 1
        y[i] = 2

// IR before bit struct store fusion
$0 = offloaded range_for(0, 1024) grid_dim=0 block_dim
    =32
body {
  <i32> $1 = loop $0 index 0
  <i32> $2 = const [1]
  <*gen> $3 = get root // the root SNode
  <i32> $4 = const [0]
  <*gen> $5 = [S0root][root]::lookup($3, $4) activate =
      false
  <*gen> $6 = get child [S0root->S1dense] $5 // the
      SNode corresponding to ti.root.dense(ti.i, 1024)
  <i32> $7 = const [1023]
  <i32> $8 = bit_and $1 $7 // from array addressing
  <*gen> $9 = [S1dense][dense]::lookup($6, $8) activate
      = false
  <*bs(ci16@0, ci16@16)> $10 = get child [S1dense->
      S2bit_struct<bs(ci16@0, ci16@16)>] $9 // the
      SNode corresponding to bit_struct(num_bits=32)
      with index $1 (the loop index i)
  <*gen> $11 = [S2bit_struct<bs(ci16@0, ci16@16)>][
      bit_struct]::lookup($10, $4) activate = false
  <^ci16> $12 = get child [S2bit_struct<bs(ci16@0,
      ci16@16)>->S3place<ci16><bit>] $11 // the SNode
      corresponding to place(x)
  $13 : global store [$12 <- $2] // x[i] = 1
  <i32> $14 = const [2]
  <^ci16> $15 = get child [S2bit_struct<bs(ci16@0,
      ci16@16)>->S4place<ci16><bit>] $11 // the SNode
      corresponding to place(y)
  $16 : global store [$15 <- $14] // y[i] = 2
}

// IR after bit struct store fusion
$0 = offloaded range_for(0, 1024) grid_dim=0 block_dim
    =32
body {
  <i32> $1 = loop $0 index 0
  <i32> $2 = const [1]
  <*gen> $3 = get root // the root SNode
  <i32> $4 = const [0]
  <*gen> $5 = [S0root][root]::lookup($3, $4) activate =
      false
  <*gen> $6 = get child [S0root->S1dense] $5 // the
      SNode corresponding to ti.root.dense(ti.i, 1024)
  <i32> $7 = const [1023]
  <i32> $8 = bit_and $1 $7 // from array addressing
  <*gen> $9 = [S1dense][dense]::lookup($6, $8) activate
      = false
  <*bs(ci16@0, ci16@16)> $10 = get child [S1dense->
      S2bit_struct<bs(ci16@0, ci16@16)>] $9 // the
      SNode corresponding to bit_struct(num_bits=32)
      with index $1 (the loop index i)
  <*gen> $11 = [S2bit_struct<bs(ci16@0, ci16@16)>][
      bit_struct]::lookup($10, $4) activate = false
  <i32> $12 = const [2]
  $13 : bit_struct_store $11, ch_ids=[0, 1], values=[$2,
      $12] // fused bit struct store from $13 and $16
      in the IR above, i.e., "(x, y)[i] = 1, 2"
}
```

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.

## 2 MICROBENCHMARKS

*Bit struct store fusion.* If we have two or more bit struct stores in the same bit struct, we can fuse them together a masked store.

In `store`, we have two fields `x, y` of 16 bits each placed in a bit struct, and we can fuse the two atomic stores into one atomic store.

```
@ti.kernel
def store():
    for i in range(n):
        x[i & 32767] = i & 1023
        y[i & 32767] = i & 15
```

`partial_store` has four fields `x, y, z, w` of 8 bits each placed in a bit struct, but only three of them are stored. Our store fusion pass fuses the three stores together, and our atomic demotion pass demotes the atomic store to a regular store.

```
@ti.kernel
def partial_store():
    for i in range(n):
        x[i] = i & 127
        y[i] = i & 31
        z[i] = i & 7
        # do not store w[i]
```

`matmul` is a simple program multiplying a $3 \times 3$ matrix with a constant matrix. The performance boost mostly comes from atomic demotion in this case.

```
@ti.kernel
def matmul():
    for i in range(n):
        F[i] = (ti.Matrix.identity(ti.f32, 3) +
                0.1 * ti.Matrix.one(ti.f32, 3, 3)) @ F[i
                    ]
```

*Encoding/decoding overhead.* `saxpy` is a standard Single-Precision A·X Plus Y program. The computation in the kernel is minimized, and we use this program to evaluate the decoding/encoding overhead of custom floating-point numbers.

```
@ti.kernel
def saxpy(alpha: ti.f32):
    for i in range(n):
        y[i] = alpha * x[i] + y[i]
```

## 3 BIT ARRAY VECTORIZATION

In this section we provide more details on *bit array vectorization*, a domain-specific optimization that allows our system to effectively process vectorized operations on bit arrays of `u1` types ("boolean").

Consider the following example, where data are copied from a 2D $128 \times 128$ `u1` array `x` to `y`:

```
x = ti.field(dtype=ti.quant.int(bits=1, signed=False)
y = ti.field(dtype=ti.quant.int(bits=1, signed=False)

cell = ti.root.dense(ti.ij, (128, 4))
cell.bit_array(ti.j, 32).place(x)
cell.bit_array(ti.j, 32).place(y)

@ti.kernel
def copy():
    for i, j in x:
        y[i, j] = x[i, j]
```

Although our system can easily improve storage efficiency, computationally this bit-wise for loop is inefficient for two reasons. Firstly, we have to use hardware-native 32-bit integer registers for our simulated 1-bit values, which uses only 1/32 of the operation bitwidth. Secondly, when store the results bit-by-bit, the code generator has to issue a large number of expensive atomicRMW operations for thread-safety, since multiple CPU/GPU threads may write to different bits within a single `u32`, leading to data races.

*Bit-wise for loop vectorization.* The situation above inspires us to vectorize bit array load, store, and arithmetics, so that each iteration processes a whole `32xu1` bit array instead of a single `u1`. This not only utilizes the full bitwidth but also eliminates the need for atomicRMW. Unlike traditional vectorized operations supported by modern processors (such as SSE and AVX), bit-level vectorization has limited hardware instructions. Basically, the only "bit-vectorized" operations offered natively by hardware are bit-wise `and (&)`, `or (|)`, and `xor (^)`.

At this point, the compiler can efficiently handle simple element-wise load/store operations, plus some Boolean arithmetics. We further conduct two optimizations that improve code generation quality and capability.

*Bit-vectorized loads with offsets.* In simulations it is often useful to load data from neighborhoods, such as `x[i+1, j]` and `x[i, j+1]`. Assuming bits are vectorized along the `j` axis, a vectorized load of `x[i+1, j]` is perfectly aligned with the bit arrays, but that of `x[i, j+1]` is not.

Actually, in most cases, data to fetch in a vectorized loop iteration do not perfectly align with the underlying bit arrays. Luckily, if we know the offset at compile-time, which is most likely true for stencils, we can still leverage most of the bit vectorization benefits by loading two adjacent bit array entries and use cheap bit operations to synthesize the load result, as shown in Fig. 1.

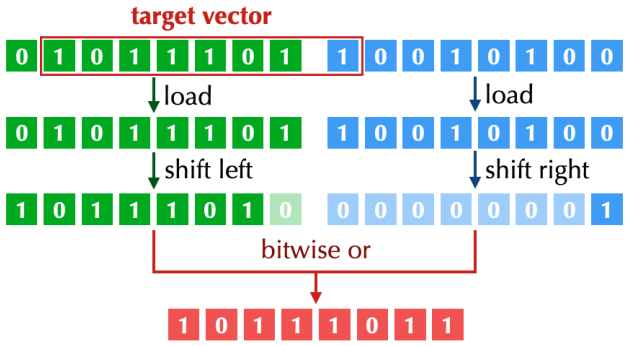


Fig. 1. Bit-vectorized loads with offsets. In this simplified example, we store the bit array using `uint8` and each node represents an element in the bit array. To load `x[i, j+1]` (red box), we first need two vectorized loads (green and blue nodes). Then we use bit shifting operations to extract the target bits and move them to corresponding locations. Finally, we merge them using a bit "or" operation.

After this optimization, the following code snippet can be efficiently compiled:

```
@ti.kernel
def copy_with_offset():
    ti.bit_vectorize(32)
    for i, j in x:
        y[i, j] = x[i, j + 1]
```

*Bit-vectorized integers and adders.* Even when operating on binary inputs and outputs, intermediate values may have large value ranges. To represent these intermediate values efficiently, for example, we store a $n$-bit 32-wide vectorized integer using $n$ `u32` integer buffers, where the $i$-th buffer stores the $i$-th bit of all the integer values being vectorized. These "bit-vectorized" integers allow us to treat each bit of the $n$-bit integer in a vectorized manner, independent of other bits on the $n$-bit integer.

Adding two bit-vectorized integers can be implemented using cheap bit operations, just like implementing a "full adder" using logic gates. Besides adding, we also implemented comparison operations between bit-vectorized integers using bit-wise operators.

In the following example, since variable `count` as range $[0, 4)$, we use a 2-bit vectorized integer to store its value.

```
@ti.kernel
def count_neighbors():
    ti.bit_vectorize(32)
    for i, j in x:
        count = 0
        count += x[i, j + 1]
        count += x[i, j - 1]
        count += x[i + 1, j]
        y[i, j] = count == 3
```

## 4 HIGH RESOLUTION GAME OF LIFE

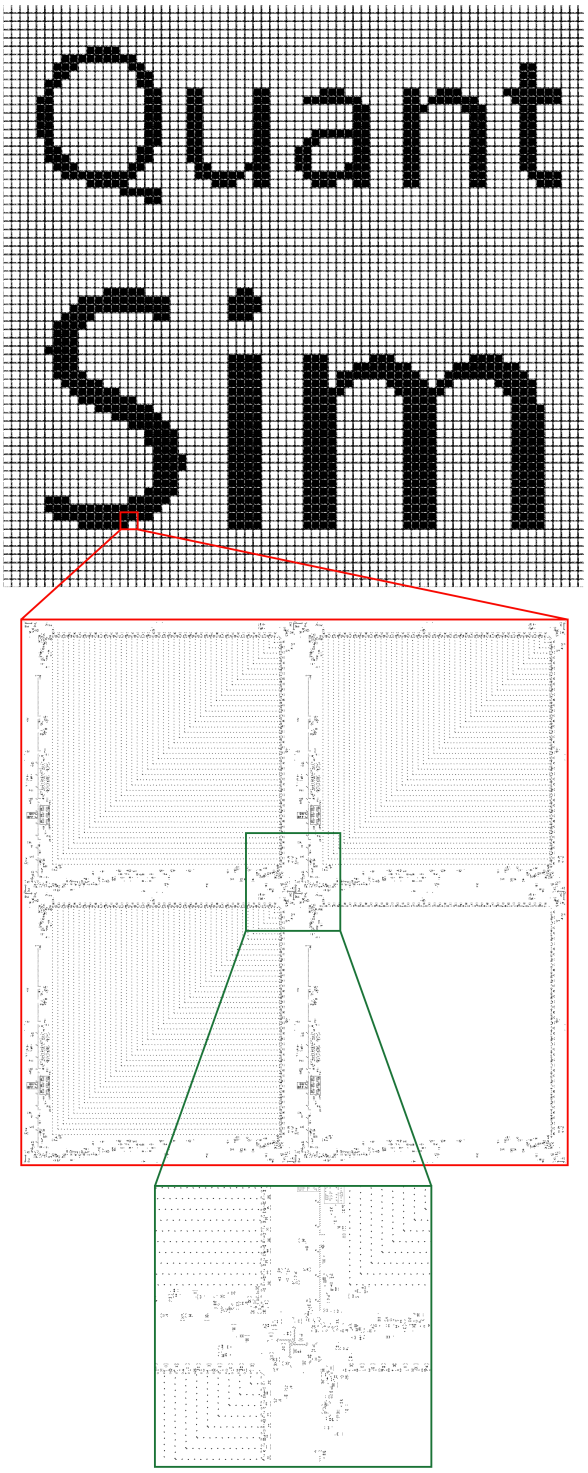


Fig. 2. High resolution Game of Life. Note the colors are inverted for better visibility. Best viewed on a high-resolution screen.

## 5 THE EFFECTIVENESS OF SHARED EXPONENTS.

A visual quantization scheme comparison is shown in Fig. 3. As mentioned in our paper, there is an exponential decay of density in the simulation, and ultimately all pixels should decay into the white background color. When compared to the `float32` version, some quantization schemes lead to unpleasant gray color blocks that never decay. For example, in the fixed-point simulation, the smoke stopped decaying after a few seconds, leaving gray regions that never decays. This is because the precision is too low to distinguish small changes. The non-shared one looks better but still suffers from the lack of precision. The shared exponent version, however, has sufficient fraction bits and looks much closer to the `float32` version compared to two other methods.

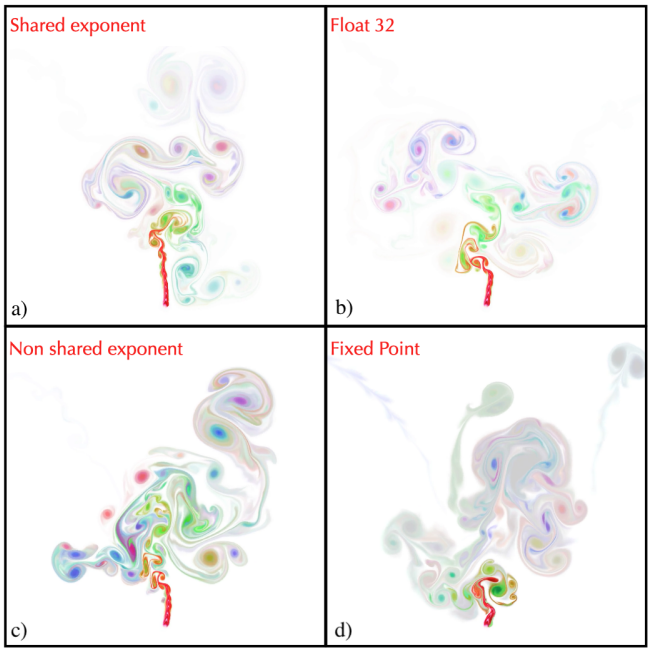


Fig. 3. Shared exponent effectiveness comparison. We compare shared exponent, non-shared exponent, and fixed-point for a 2D smoke simulation. Notice shared exponent float looks clearer while non-shared exponent float suffer from some unpleasant grays due to fewer fraction bits. The fixed-point version is also subject to the same problem caused by low precision and dynamic range. Note that fluids are highly turbulent so the dye patterns are different from time to time, even when using `float32`.

## 6 VISUAL COMPARISON CASES AND QUANTIZATION SCHEMES

Visual comparison cases (Fig. 4) are presented in the supplemental video. Here we provide more details on the quantization scheme of each case.

*Case 1 and 2.* We use `fixed23` for position, `ti.quant.float(exp=6, frac=13)` for velocity (shared exponent). We use `fixed16` for entries in the deformation gradient matrix.


Fig. 4. Five visual comparison cases used to qualitatively compare quantized and full-precision simulators.

*Case 3.* This is an MLS-MPM simulation. We use the same quantization scheme as the 235M-particle simulation, and use a `fixed23` to track the fluid volume.

*Case 4.* In this 2D smoke simulation, we use fixed21 for velocity and shared exponent with 5 bits for exponent and 9 bits for fraction to quantize dye density.

*Case 5.* We use the same quantization scheme as the 421M-voxel smoke simulation.