Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures (Supplemental Document)

YUANMING HU, MIT CSAIL
TZU-MAO LI, MIT CSAIL and UC Berkeley
LUKE ANDERSON, MIT CSAIL
JONATHAN RAGAN-KELLEY, UC Berkeley
FRÉDO DURAND, MIT CSAIL

1 INTERMEDIATE REPRESENTATION SAMPLE

Below is a simple kernel and its IR, in multiple compilation stages.

```
// Kernel
Kernel(inc).def([&]() {
  For(a, [&](Expr i) {
    a[i] = b[i] + 1;
  });
});
// AST
for tmp2 where a_global active {
  #a_global[tmp2] = (load #b_global[tmp2] + 1)
}
// IR
<i32x1> $0 = alloca
for $0 where a active, step 1 {
  $2 = local load [ [$0[0]]]
  <i32x1> $3 = ptr [S1], index [$2] activate=false
  $4 = global load $3
  <i32x1> $5 = const [1]
  $6 = add $4 $5
  $7 = local load [ [$0[0]]]
  <i32x1> $8 = ptr [S0], index [$7] activate=true
  $9 : global store [$8 <- $6]
}
// Access Lowered
<i32x1> $0 = alloca
for $0 where S0 active, step 1 {
  <i32x1> $2 = local load [ [$0[0]]]
  <i32x1> $3 = ptr [S1], index [$2] activate=true
  <i32x1> $4 = shuffle $2[0]
  $5 = linearized(ind {}, stride {})
  $6 = [S4][root]::lookup(root, $5) coord = {$4}
       activate = false
  $7 = get child [S4->S3] $6
  $8 = bit_extract($4 + 0, 7~14)
  $9 = linearized(ind {$8}, stride {128})
  $10 = [S3][dense]::lookup($7, $9) coord = {$4}
       activate = false
  $11 = get child [S3->S2] $10
  $12 = bit_extract($4 + 0, 0~7)
  $13 = linearized(ind {$12}, stride {128})
  $14 = [S2][dense]::lookup($11, $13) coord = {$4}
       activate = false
  $15 = get child [S2->S1] $14
  <i32x1> $16 = shuffle $15[0]
  <i32x1> $17 = global load $16
  <i32x1> $18 = const [1]
  <i32x1> $19 = add $17 $18
  <i32x1> $20 = ptr [S0], index [$2] activate=true
  <i32x1> $21 = shuffle $2[0]
  $22 = linearized(ind {}, stride {})
  $23 = [S4][root]::lookup(root, $22) coord = {$21}
       activate = false
  $24 = get child [S4->S3] $23
  $25 = bit_extract($21 + 0, 7~14)
  $26 = linearized(ind {$25}, stride {128})
  $27 = [S3][dense]::lookup($24, $26) coord = {$21}
       activate = false
  $28 = get child [S3->S2] $27
  $29 = bit_extract($21 + 0, 0~7)
  $30 = linearized(ind {$29}, stride {128})
  $31 = [S2][dense]::lookup($28, $30) coord = {$21}
       activate = false
  $32 = get child [S2->S0] $31
  <i32x1> $33 = shuffle $32[0]
  <i32x1> $34 : global store [$33 <- $19]
}
// Final Optimized
<i32x1> $0 = alloca
for $0 where S0 active, step 1 {
  <i32x1> $2 = local load [ [$0[0]]]
  $3 = linearized(ind {}, stride {})
  $4 = [S4][root]::lookup(root, $3) coord = {$2}
       activate = false
  $5 = get child [S4->S3] $4
  $6 = bit_extract($2 + 0, 7~14)
  $7 = linearized(ind {$6}, stride {128})
  $8 = [S3][dense]::lookup($5, $7) coord = {$2} activate
       = false
  $9 = get child [S3->S2] $8
  $10 = bit_extract($2 + 0, 0~7)
  $11 = linearized(ind {$10}, stride {128})
  $12 = [S2][dense]::lookup($9, $11) coord = {$2}
       activate = false
  $13 = get child [S2->S1] $12
  <i32x1> $14 = global load $13
  <i32x1> $15 = const [1]
  <i32x1> $16 = add $14 $15
  $17 = get child [S2->S0] $12
  <i32x1> $18 : global store [$17 <- $16]
}
```

2 MGPCG CODE COMPARISON

Below is the 7-point stencil code (Relaxation_And_Dot_Product_Interior_Helper.h) from [McAdams et al. 2010]. Note although a dense 2-level grid is used (instead of a more complex sparse one), the code is already convoluted. The redundant code is necessary, however, for performance since they enforce complex strides to be evaluated at compile time.

```
enum WORKAROUND {
  x_block_size = 4,
  y_block_size = 4,
  z_block_size = 4,
  padded_y_size = y_size + 2,
  padded_z_size = z_size + 2,
  coarse_y_size = y_size / 2,
  coarse_z_size = z_size / 2,
  coarse_padded_y_size = coarse_y_size + 2,
  coarse_padded_z_size = coarse_z_size + 2,
  x_shift = padded_y_size * padded_z_size,
  y_shift = padded_z_size,
  z_shift = 1,
  coarse_x_shift = coarse_padded_y_size *
       coarse_padded_z_size,
  coarse_y_shift = coarse_padded_z_size,
  coarse_z_shift = 1,
  x_plus_one_shift = x_shift,
```

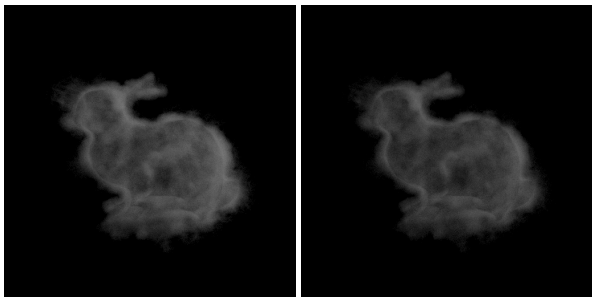

Fig. 1. Smoke in the shape of a bunny rendered with our volumetric path tracer (left) and Tungsten's (right). Both rendered with 128 samples per pixel and a path length limit of 128.

```
  x_minus_one_shift = -x_shift,
  y_plus_one_shift = y_shift,
  y_minus_one_shift = -y_shift,
  z_plus_one_shift = z_shift,
  z_minus_one_shift = -z_shift
};
...
template <class T, int y_size, int z_size> void
Relaxation_And_Dot_Product_Interior_Size_Specific_Helper
<T, y_size, z_size>::
Compute_Delta_X_Range(const int xmin,
                      const int xmax,
                      const int partition_number) {
  const T one_sixth = 1. / 6.;
for(int block_i=xmin;block_i<=xmax;
    block_i+=x_block_size)
for(int block_j=1;block_j<=y_size;block_j+=y_block_size)
for(int block_k=1;block_k<=z_size;block_k+=z_block_size)
for(int i=block_i;i<block_i+x_block_size;i++)
for(int j=block_j;j<block_j+y_block_size;j++)
for(int k=block_k;k<block_k+z_block_size;k++) {
  int index = i * x_shift + j * y_shift + k * z_shift;
  delta[index] = one_sixth * (
    u[index + x_plus_one_shift] +
    u[index + x_minus_one_shift] +
    u[index + y_plus_one_shift] +
    u[index + y_minus_one_shift] +
    u[index + z_plus_one_shift] +
    u[index + z_minus_one_shift]) -
    u[index];
}}
```

Code in our language that defines the data structure and the same stencil:

```
int block_size = 4;
layout([&] {
    root.dense(ijk, n / block_size)
        .dense(ijk, block_size).place(u);
    root.dense(ijk, n / block_size)
        .dense(ijk, block_size).place(delta);
});

Kernel(relaxation).def([&]{
  Parallelize(8);
  For(y, [&](Expr i, Expr j, Expr k){
    delta[i, j, k] = 1.0f/6.0f *
      (u[i, j, k - 1] + u[i, j, k + 1] + u[i, j - 1, k]
     + u[i, j + 1, k] + u[i - 1, j, k] + u[i + 1, j, k])
       - u[i, j, k];
  });
});
```

3 VOLUMETRIC PATH TRACING RENDERINGS

A visual comparison of our rendering result compared to Tungsten's is in Fig. 1.

4 SIMPLIFICATION PASS DETAILS

The simplification pass of the Taichi compiler does the following optimizations:

- Dead instruction elimination.
- Common subexpression elimination.
- Store forwarding.
- Useless local store elimination, i.e. delete stores whose results that are overwritten by future stores.
- Simplify if statements into conditional stores.

Two simplifications passes are applied to the program, before and after the access lowering pass, respectively. The main goal of the first pass is to reduce the number of instructions, while the second pass is mainly to optimize access. Note that the access simplification is essentially a common subexpression elimination pass over the micro-access instructions.

Other possible optimizations such as constant folding are left for the backend compiler.

5 CODE SAMPLES

5.1 MLS-MPM with Comments

```
#include <taichi/lang.h>
#include <taichi/util.h>
#include <taichi/visual/gui.h>
#include <taichi/common/bit.h>
#include <taichi/system/profiler.h>
#include "svd.h"
TC_NAMESPACE_BEGIN
using namespace Tlang;
auto mpm_benchmark = []() {
  Program prog(Arch::gpu);
  // No accessing lowering in this specific application is needed, since
  // the scratchpad optimizations already significantly reduce data structure access
  prog.config.lower_access = false;
  // simulation constants
  constexpr int dim = 3, n = 256, grid_block_size = 4, n_particles = 775196;
  const real dt = 1e-5_f * 256 / n, dx = 1.0_f / n, inv_dx = 1.0_f / dx;
  auto particle_mass = 1.0_f, vol = 1.0_f, E = 1e4_f, nu = 0.3f;
  real mu = E / (2 * (1 + nu)), lambda = E * nu / ((1 + nu) * (1 - 2 * nu));
  // simulation precision
  auto f32 = DataType::f32;
  // global variables
  Vector particle_x(f32, dim), particle_v(f32, dim), grid_v(f32, dim);
  Matrix particle_F(f32, dim, dim), particle_C(f32, dim, dim);
  Global(grid_m, f32);
  Global(l, i32);
  Global(gravity_x, f32);
  // load input
  int max_n_particles = 1024 * 1024;
  std::vector<Vector3> p_x;
  p_x.resize(n_particles);
  std::vector<float> benchmark_particles;
  auto f = fopen("dragon_particles.bin", "rb");
  TC_ASSERT_INFO(f, "./dragon_particles.bin not found");
  benchmark_particles.resize(n_particles * 3);
  std::fread(benchmark_particles.data(), sizeof(float), n_particles * 3, f);
  std::fclose(f);
  for (int i = 0; i < n_particles; i++) {
    for (int j = 0; j < dim; j++)
      p_x[i][j] = benchmark_particles[i * dim + j];
  }
  bool particle_SOA = false;
  // layout function call, materialize the data structure
  layout([&]() {
    auto i = Index(0), j = Index(1), k = Index(2), p = Index(3);
    SNode *fork;
    if (!particle_SOA)
      fork = &root.dynamic(p, max_n_particles);
    auto place = [&](Expr &expr) {
      if (particle_SOA) {
        root.dynamic(p, max_n_particles).place(expr);
      } else {
        fork->place(expr);
      }
    };
    for (int i = 0; i < dim; i++)
      for (int j = 0; j < dim; j++)
        place(particle_F(i, j));
    for (int i = 0; i < dim; i++)
      for (int j = 0; j < dim; j++)
        place(particle_C(i, j));
    for (int i = 0; i < dim; i++)
      place(particle_x(i));
    for (int i = 0; i < dim; i++)
      place(particle_v(i));
    TC_ASSERT(n % grid_block_size == 0);
    auto &block = root.dense({i, j, k}, n / grid_block_size).pointer();
    constexpr bool block_soa = true;
    if (block_soa) {
      block.dense({i, j, k}, grid_block_size).place(grid_v(0));
      block.dense({i, j, k}, grid_block_size).place(grid_v(1));
      block.dense({i, j, k}, grid_block_size).place(grid_v(2));
      block.dense({i, j, k}, grid_block_size).place(grid_m);
    } else {
```

```
    block.dense({i, j, k}, grid_block_size)
        .place(grid_v(0), grid_v(1), grid_v(2), grid_m);
  }
  block.dynamic(p, pow<dim>(grid_block_size) * 128).place(l);
  root.place(gravity_x);
});
// sort particle indices into their belonging block
Kernel(sort).def([&] {
  BlockDim(1024);
  For(particle_x(0), [&](Expr p) {
    // compute the block coordinates
    auto node_coord = floor(particle_x[p] * inv_dx - 0.5_f);
    // insert the particle index
    Append(l.parent(),
           (cast<int32>(node_coord(0)), cast<int32>(node_coord(1)),
            cast<int32>(node_coord(2))),
           p);
  });
});
// Particle to grid transfer
Kernel(p2g_sorted).def([&] {
  // GPU block dim
  BlockDim(128);
  // allocate scratch pads for the velocity and mass channels
  Cache(0, grid_v(0));
  Cache(0, grid_v(1));
  Cache(0, grid_v(2));
  Cache(0, grid_m);
  For(l, [&](Expr i, Expr j, Expr k, Expr p_ptr) {
    // for each particle, compute its momentum contribution and scatter to surrounding grid nodes
    auto p = Var(l[i, j, k, p_ptr]);
    auto x = Var(particle_x[p]), v = Var(particle_v[p]),
         C = Var(particle_C[p]);
    auto base_coord = floor(inv_dx * x - 0.5_f), fx = x * inv_dx - base_coord;
    Matrix F = Var(Matrix::identity(dim) + dt * C) * particle_F[p];
    particle_F[p] = F;
    Vector w[] = {Var(0.5_f * sqr(1.5_f - fx)), Var(0.75_f - sqr(fx - 1.0_f)),
                  Var(0.5_f * sqr(fx - 0.5_f))};
    auto svd = sifakis_svd(F);
    auto R = Var(std::get<0>(svd) * transposed(std::get<2>(svd)));
    auto sig = Var(std::get<1>(svd));
    auto J = Var(sig(0) * sig(1) * sig(2));
    auto cauchy = Var(2.0_f * mu * (F - R) * transposed(F) +
                      (Matrix::identity(3) * lambda) * (J - 1.0f) * J);
    auto affine =
        Var(particle_mass * C - (4 * inv_dx * inv_dx * dt * vol) * cauchy);
    int low = 0, high = 1;
    // The AssumeInRange intrinsics tells the compiler how big the scratchpad should be
    auto base_coord_i =
        AssumeInRange(cast<int32>(base_coord(0)), i, low, high);
    auto base_coord_j =
        AssumeInRange(cast<int32>(base_coord(1)), j, low, high);
    auto base_coord_k =
        AssumeInRange(cast<int32>(base_coord(2)), k, low, high);
    for (int a = 0; a < 3; a++)
      for (int b = 0; b < 3; b++)
        for (int c = 0; c < 3; c++) {
          auto dpos = dx * (Vector({a, b, c}).cast_elements<float32>() - fx);
          auto weight = w[a](0) * w[b](1) * w[c](2);
          auto node = (base_coord_i + a, base_coord_j + b, base_coord_k + c);
          // Atomic adds for safe parallelism
          Atomic(grid_v[node]) +=
              weight * (particle_mass * v + affine * dpos);
          Atomic(grid_m[node]) += weight * particle_mass;
        }
  });
});
// grid operations
Kernel(grid_op).def([&]() {
  For(grid_m, [&](Expr i, Expr j, Expr k) {
    auto v = Var(grid_v[i, j, k]);
    auto m = Var(grid_m[i, j, k]);
    int bound = 8;
    // normalize momentum into velocity
    If(m > 0.0f, [&]() {
      auto inv_m = Var(1.0f / m);
```

```
      v *= inv_m;
      // apply gravity
      auto f = gravity_x[Expr(0)];
      v(1) += dt * (-1000_f + abs(f));
      v(0) += dt * f;
    });
    // boundary conditions
    v(0) = select(n - bound < i, min(v(0), Expr(0.0_f)), v(0));
    v(1) = select(n - bound < j, min(v(1), Expr(0.0_f)), v(1));
    v(2) = select(n - bound < k, min(v(2), Expr(0.0_f)), v(2));
    v(0) = select(i < bound, max(v(0), Expr(0.0_f)), v(0));
    v(2) = select(k < bound, max(v(2), Expr(0.0_f)), v(2));
    If(j < bound, [&] { v(1) = max(v(1), Expr(0.0_f)); });
    grid_v[i, j, k] = v;
  });
});
// grid to particle transfer
Kernel(g2p).def([&]() {
  // GPU block dim
  BlockDim(128);
  // allocate scratchpads for the velocity channels
  Cache(0, grid_v(0));
  Cache(0, grid_v(1));
  Cache(0, grid_v(2));
  For(l, [&](Expr i, Expr j, Expr k, Expr p_ptr) {
    auto p = Var(l[i, j, k, p_ptr]);
    auto x = Var(particle_x[p]), v = Var(Vector(dim)),
         C = Var(Matrix(dim, dim));
    for (int i = 0; i < dim; i++) {
      v(i) = Expr(0.0_f);
      for (int j = 0; j < dim; j++) {
        C(i, j) = Expr(0.0_f);
      }
    }
    auto base_coord = floor(inv_dx * x - 0.5_f);
    auto fx = x * inv_dx - base_coord;
    Vector w[] = {Var(0.5_f * sqr(1.5_f - fx)), Var(0.75_f - sqr(fx - 1.0_f)),
                  Var(0.5_f * sqr(fx - 0.5_f))};
    int low = 0, high = 1;
    auto base_coord_i =
        AssumeInRange(cast<int32>(base_coord(0)), i, low, high);
    auto base_coord_j =
        AssumeInRange(cast<int32>(base_coord(1)), j, low, high);
    auto base_coord_k =
        AssumeInRange(cast<int32>(base_coord(2)), k, low, high);
    for (int p = 0; p < 3; p++)
      for (int q = 0; q < 3; q++)
        for (int r = 0; r < 3; r++) {
          auto dpos = Vector({p, q, r}).cast_elements<float32>() - fx;
          auto weight = w[p](0) * w[q](1) * w[r](2);
          auto wv =
              weight *
              grid_v[base_coord_i + p, base_coord_j + q, base_coord_k + r];
          v += wv;
          C += outer_product(wv, dpos);
        }
    particle_C[p] = (4 * inv_dx) * C;
    particle_v[p] = v;
    particle_x[p] = x + dt * v;
  });
});
// initial particle reordering
auto block_id = [&](Vector3 x) {
  auto xi = (x * inv_dx - Vector3(0.5f)).floor().template cast<int>() /
            Vector3i(grid_block_size);
  return xi.x * pow<2>(n / grid_block_size) + xi.y * n / grid_block_size +
         xi.z;
};
std::sort(p_x.begin(), p_x.end(),
          [&](Vector3 a, Vector3 b) { return block_id(a) < block_id(b); });
for (int i = 0; i < n_particles; i++) {
  for (int d = 0; d < dim; d++) {
    particle_x(d).val<float32>(i) = p_x[i][d];
  }
  particle_v(0).val<float32>(i) = 0._f;
  particle_v(1).val<float32>(i) = -3.0_f;
```

```
    particle_v(2).val<float32>(i) = 0._f;
    for (int p = 0; p < dim; p++)
      for (int q = 0; q < dim; q++)
        particle_F(p, q).val<float32>(i) = (p == q);
  }
  // main simulation loop
  auto simulate_frame = [&]() {
    grid_m.parent().parent().snode()->clear_data_and_deactivate();
    auto t = Time::get_time();
    for (int f = 0; f < 200; f++) {
      grid_m.parent().parent().snode()->clear_data();
      sort();
      p2g_sorted();
      grid_op();
      g2p();
    }
    prog.profiler_print();
    auto ms_per_substep = (Time::get_time() - t) / 200 * 1000;
    TC_P(ms_per_substep);
  };
  // Visualization omitted...
};
TC_REGISTER_TASK(mpm_benchmark);
TC_NAMESPACE_END
```

See Hu et al.'s article [Hu et al. 2018] for the derivation of the algorithm.

5.2 FEM Linear Elasticity Kernel

This kernel is the implementation of Equation (1) of Liu et al.'s work [2018].

```
Kernel(compute_Ap).def([&] {
  For(Ap(0), [&](Expr i, Expr j, Expr k) {
    auto cell_coord = Var(Vector({i, j, k}));
    auto Ku_tmp = Var(Vector(dim));
    Ku_tmp = Vector({0.0f, 0.0f, 0.0f});
    // Unrolled for loop
    for (int cell = 0; cell < pow<dim>(2); cell++) {
      auto cell_offset =
          Var(Vector({-(cell / 4), -(cell / 2 % 2), -(cell % 2)}));
      auto cell_lambda = lambda[cell_coord + cell_offset];
      auto cell_mu = mu[cell_coord + cell_offset];
      // Unrolled for loop
      for (int node = 0; node < pow<dim>(2); node++) {
        auto node_offset = Var(Vector({node / 4, node / 2 % 2, node % 2}));
        // Unrolled for loop
        for (int u = 0; u < dim; u++)
          // Unrolled for loop
          for (int v = 0; v < dim; v++)
            Ku_tmp(u) += (cell_lambda * K_la[cell][node][u][v] +
                          cell_mu * K_mu[cell][node][u][v]) *
                         p[cell_coord + cell_offset + node_offset](v);
      }
    }
  });
});
```

5.3 MGPCG Program

Please check out `examples/cpp/mgpcg.cpp`.

5.4 CNN Kernel

```
  Kernel(forward).def([&] {
    if (opt && gpu) {
      if (cache_l1)
        CacheL1(weights);
    }
    if (!gpu) {
      Parallelize(8);
      Vectorize(block_size);
    } else {
      BlockDim(256);
    }
    For(layer2, [&](Expr i, Expr j, Expr k, Expr c_out) {
```

```
      auto sum = Var(0.0f);
      for (int c_in = 0; c_in < num_ch1; c_in++) {
        for (int dx = -1; dx < 2; dx++) {
          for (int dy = -1; dy < 2; dy++) {
            for (int dz = -1; dz < 2; dz++) {
              auto weight = weights[Expr(dx + 1), Expr(dy + 1), Expr(dz + 1),
                                    c_in * num_ch2 + c_out];
              sum += weight * layer1[i + dx, j + dy, k + dz, c_in];
            }
          }
        }
      }
      layer2[i, j, k, c_out] = sum;
    });
  });
```

5.5 Volume Renderer

The code is at `examples/cpp/volume_renderer.cpp`.

REFERENCES

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 Transactions on Graphics (TOG)* 37, 4 (2018), 150.

Haixiang Liu, Yuanming Hu, Bo Zhu, Wojciech Matusik, and Eftychios Sifakis. 2018. Narrow-band topology optimization on a sparsely populated grid. In *SIGGRAPH Asia 2018 Technical Papers*. ACM, 251.

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