Graphics DSLs

CS 343S

April 15th, 2025

Images, Simulations, Knitting, and Diagrams

- Halide: A language for fast, portable computation on images
- Taichi: A language for high-performance computation on spatially sparse data structures
- Knitout: Low-level knitting machine instructions
- Penrose: from mathematical notation to beautiful diagrams

Images, Simulations, Knitting, and Diagrams

- **Halide: A language for fast, portable computation on images**
- Taichi: A language for high-performance computation on spatially sparse data structures
- Knitout: Low-level knitting machine instructions
- Penrose: from mathematical notation to beautiful diagrams

Halide: A language for fast, portable computation on images

- Observations:
  - Most image processing pipelines are just per-pixel functions
  - ***Fast*** image processing code is a composition of a small set of loop optimizations (machine-specific)
- Idea: specify image processing algorithms as a per-pixel function:

$$blurx(\text{x, y}) = \frac{in(\text{x - 1, y}) + in(\text{x, y}) + in(\text{x + 1, y})}{3}$$

$$blury(\text{x, y}) = \frac{blurx(\text{x, y - 1}) + blurx(\text{x, y}) + blurx(\text{x, y + 1})}{3}$$

- Separately specific loop optimizations (shown in a few slides)

Implementing a Blur

$$blurx(x, y) = \frac{in(x - 1, y) + in(x, y) + in(x + 1, y)}{3}$$

$$blury(x, y) = \frac{blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)}{3}$$

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Implementing a Blur

$$blurx(x, y) = \frac{in(x - 1, y) + in(x, y) + in(x + 1, y)}{3}$$

$$blury(x, y) = \frac{blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)}{3}$$

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Implementing a Blur

$$blurx(x, y) = \frac{in(x - 1, y) + in(x, y) + in(x + 1, y)}{3}$$

$$blury(x, y) = \frac{blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)}{3}$$

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
    }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Implementing a Blur

$$blurx(x, y) = \frac{in(x - 1, y) + in(x, y) + in(x + 1, y)}{3}$$

$$blury(x, y) = \frac{blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)}{3}$$

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
#pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Implementing a Blur

$$blurx(x, y) = \frac{in(x - 1, y) + in(x, y) + in(x + 1, y)}{3}$$

$$blury(x, y) = \frac{blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)}{3}$$

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

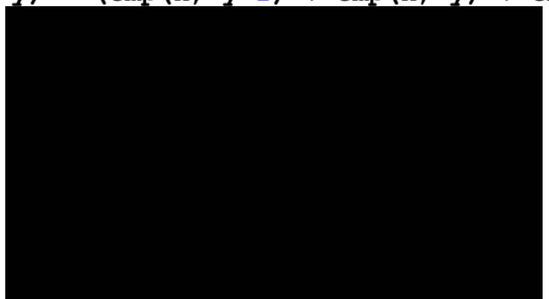
```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Implementing a Blur

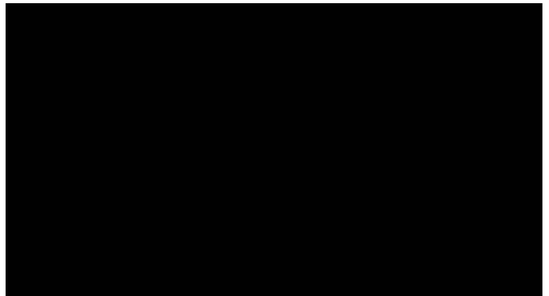
$$blurx(x, y) = \frac{in(x - 1, y) + in(x, y) + in(x + 1, y)}{3}$$

$$blury(x, y) = \frac{blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)}{3}$$

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```



```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```



Implementing a Blur

$$blurx(x, y) = \frac{in(x - 1, y) + in(x, y) + in(x + 1, y)}{3}$$

$$blury(x, y) = \frac{blurx(x, y - 1) + blurx(x, y) + blurx(x, y + 1)}{3}$$

$$blury\,.\,tile(x, y, xi, yi, 256, 32)\,.\,vectorize(xi, 8)\,.\,parallel(y);$$

$$blurx\,.\,chunk(x)\,.\,vectorize(x, 8);$$

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

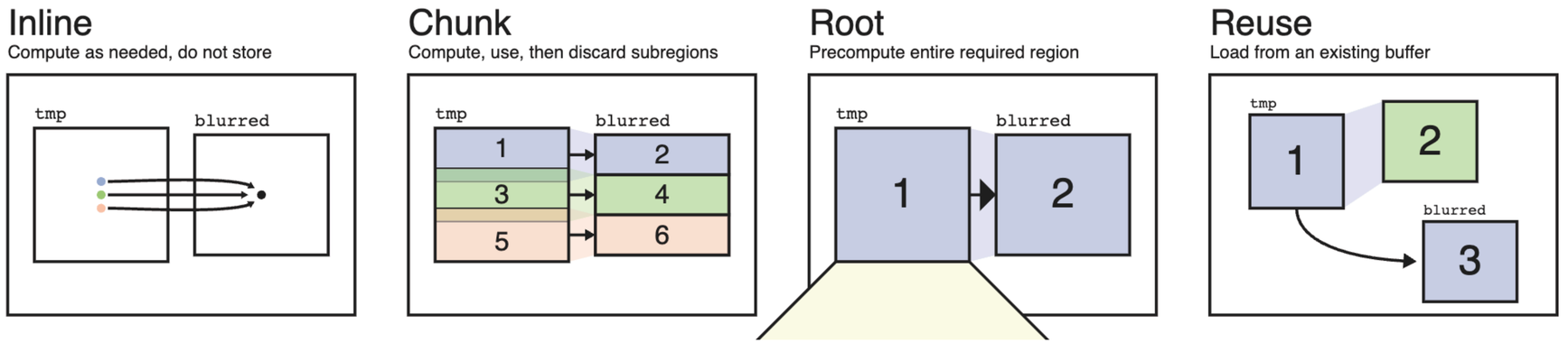
```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Representing Algorithms

- Proposed a functional, data-parallel algorithm *specification*: what to compute
- Supports:
  - Basic arithmetic
  - Loads from external images / other functions
  - If-then-else
  - (Finite-sized) reductions
- **Notably: not Turing-complete!**

Representing Schedules (AKA Optimizations)

- Decide how a producer is scheduled with respect to its consumer



- Decide traversal order (e.g. column-first, row-first, channels-first)

- Add parallelization / loop unrolling / etc.

Design Decisions

- Embedded in C++
  - Enables using C++ metaprogramming to search for good schedules!
  - Avoids needing to develop and maintain a parser
  - Added benefit: one of the (at-the-time) 5 supported Google languages
  - Easy use of LLVM compiler infrastructure

Images, Simulations, Knitting, and Diagrams

- Halide: A language for fast, portable computation on images
  - Decouple algorithms from optimizations for portability
- **Taichi: A language for high-performance computation on spatially sparse data structures**
- Knitout: Low-level knitting machine instructions
- Penrose: from mathematical notation to beautiful diagrams

Taichi

Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures

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

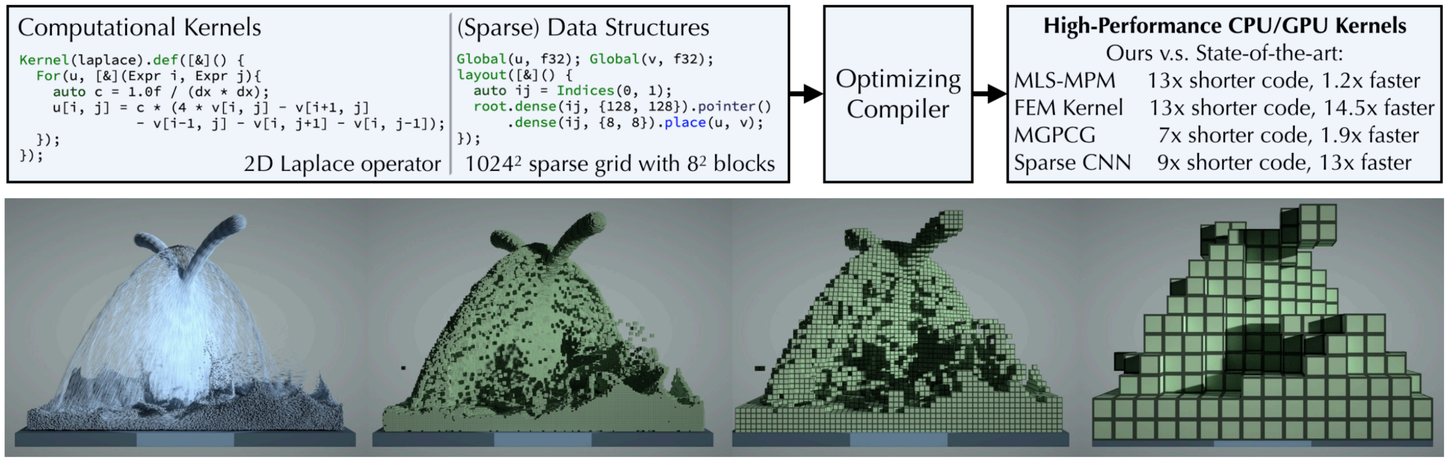


Fig. 1. (Top) We propose the Taichi programming language, which exposes a high-level interface for developing and processing spatially sparse multi-level data structures, and an optimizing compiler that automatically reduces data structure overhead. Programmers write code as if they are accessing dense voxels, while specifying the data arrangement *independently*. Our compiler automatically generates optimized, high-performance code tailored to the data structure. This results in concise code and better performance than highly-optimized reference implementations for various tasks. (Bottom) A fluid simulation using the material point method, where two liquid jets collide with each other, forming a thin sheet structure. We used a three-level sparse voxel grid with sizes 1^3, 4^3, 16^3. Involved voxels are visualized in green. Both simulation and rendering are done using programs written in Taichi.

Taichi

Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures

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

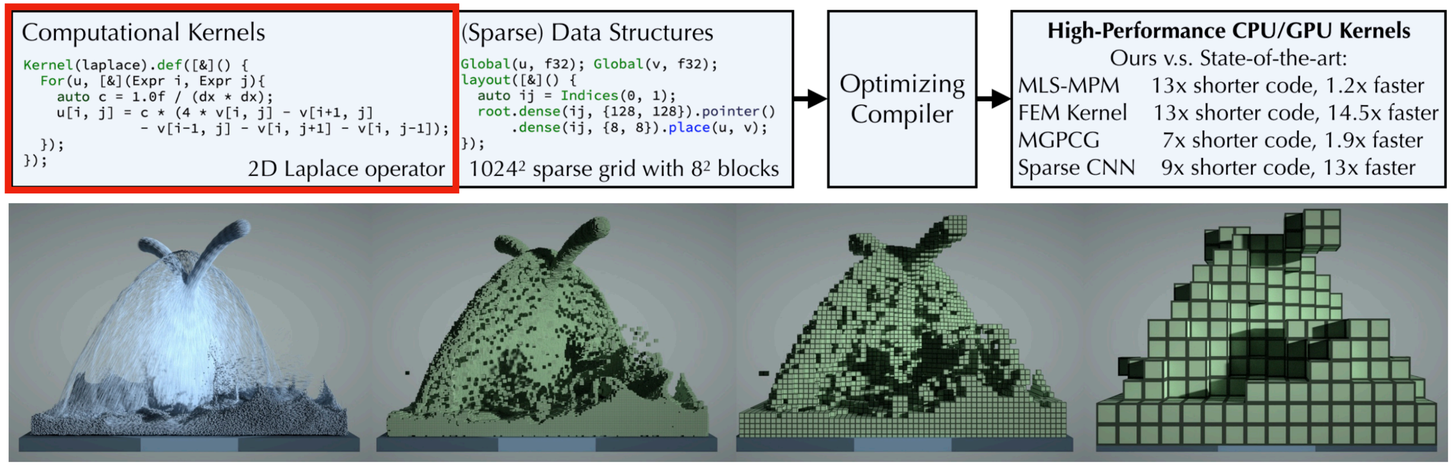


Fig. 1. (Top) We propose the Taichi programming language, which exposes a high-level interface for developing and processing spatially sparse multi-level data structures, and an optimizing compiler that automatically reduces data structure overhead. Programmers write code as if they are accessing dense voxels, while specifying the data arrangement *independently*. Our compiler automatically generates optimized, high-performance code tailored to the data structure. This results in concise code and better performance than highly-optimized reference implementations for various tasks. (Bottom) A fluid simulation using the material point method, where two liquid jets collide with each other, forming a thin sheet structure. We used a three-level sparse voxel grid with sizes 1^3, 4^3, 16^3. Involved voxels are visualized in green. Both simulation and rendering are done using programs written in Taichi.

Taichi

Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures

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

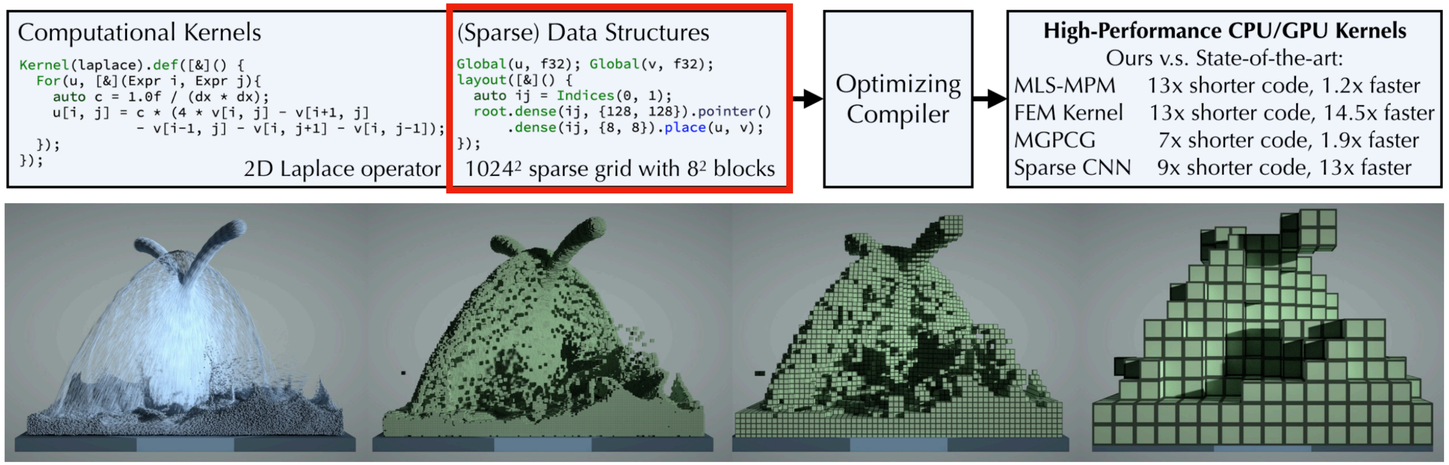
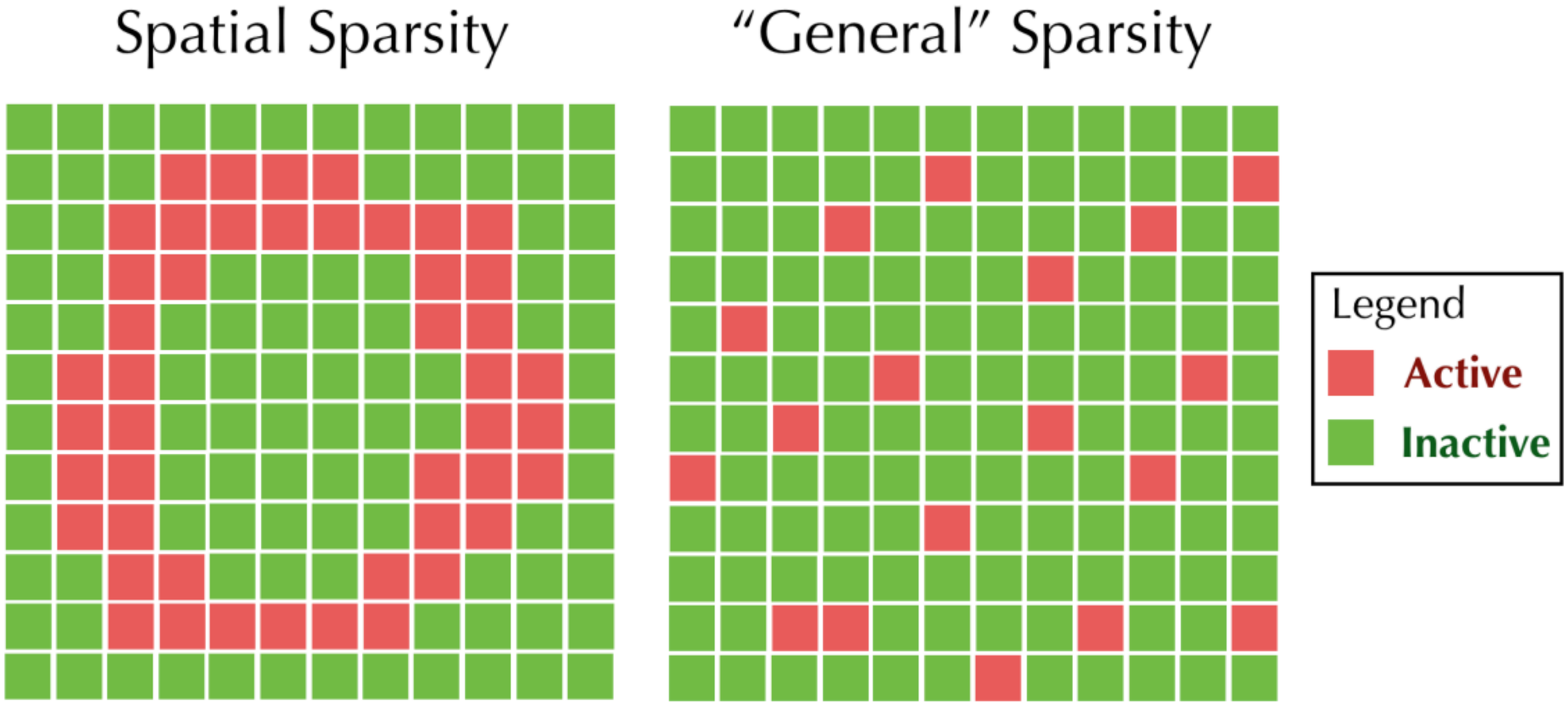


Fig. 1. (Top) We propose the Taichi programming language, which exposes a high-level interface for developing and processing spatially sparse multi-level data structures, and an optimizing compiler that automatically reduces data structure overhead. Programmers write code as if they are accessing dense voxels, while specifying the data arrangement *independently*. Our compiler automatically generates optimized, high-performance code tailored to the data structure. This results in concise code and better performance than highly-optimized reference implementations for various tasks. (Bottom) A fluid simulation using the material point method, where two liquid jets collide with each other, forming a thin sheet structure. We used a three-level sparse voxel grid with sizes 1^3, 4^3, 16^3. Involved voxels are visualized in green. Both simulation and rendering are done using programs written in Taichi.

Different Kinds of Sparsity

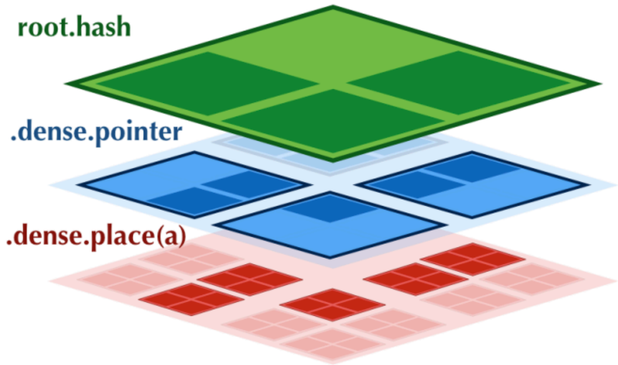


Aside: Goals and Design Decisions

- Expressivity
  - Complex computation kernels and control flow
- Performance
  - Data structure design is vital: but also (task and machine)-specific
- Productive
  - Algorithm code written as if ***grids are dense***
- Portable
  - No low-level target-specific intrinsics

Design Decisions

- Decouple data structures from computation
  - Provide dense-like abstraction, let DS specification map indexing
- Regular grids as building blocks
  - No meshes (yet), no graphs, no trees, etc.
- Specify data structures through hierarchical composition
- Compile-time constant data structure hierarchy (no trees)
- Single-Program-Multiple-Data (SPMD) with sparse iterators



Programming Model: Dense Grids

$$u_{i,j} = \frac{1}{\Delta x^2}(4v_{i,j} - v_{i+1,j} - v_{i-1,j} - v_{i,j+1} - v_{i,j-1})$$

Programming Model: Dense Grids

$$u_{i,j} = \frac{1}{\Delta x^2}(4v_{i,j} - v_{i+1,j} - v_{i-1,j} - v_{i,j+1} - v_{i,j-1})$$

```
for i, j in v:
  c = 1 / (dx * dx)
  u[i, j] = c * (4 * v[i, j] - v[i+1, j] -
                       v[i-1, j] - v[i, j+1] - v[i, j-1])
```

Interface: Iteration (over non-empty elems)

$$u_{i,j} = \frac{1}{\Delta x^2}(4v_{i,j} - v_{i+1,j} - v_{i-1,j} - v_{i,j+1} - v_{i,j-1})$$

```
for i, j in v:
    c = 1 / (dx * dx)
    u[i, j] = c * (4 * v[i, j] - v[i+1, j] -
                             v[i-1, j] - v[i, j+1] - v[i, j-1])
```

Interface: Loads

$$u_{i,j} = \frac{1}{\Delta x^2}(4v_{i,j} - v_{i+1,j} - v_{i-1,j} - v_{i,j+1} - v_{i,j-1})$$

```
for i, j in v:
  c = 1 / (dx * dx)
  u[i, j] = c * (4 * v[i, j] - v[i+1, j] -
                  v[i-1, j] - v[i, j+1] - v[i, j-1])
```

Interface: Stores

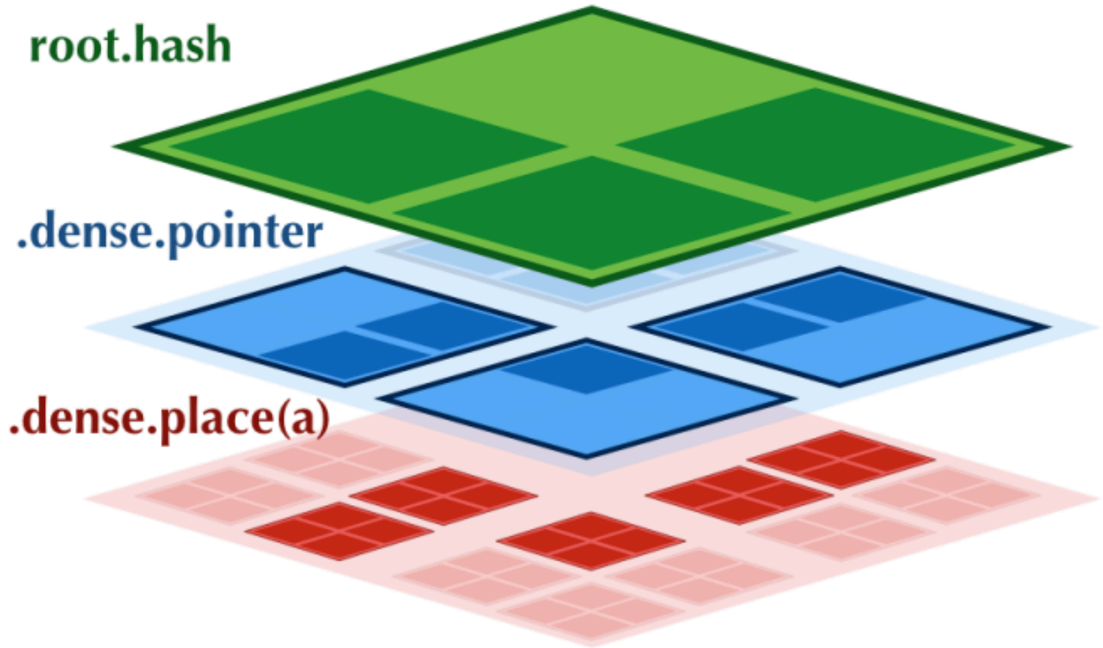
$$u_{i,j} = \frac{1}{\Delta x^2}(4v_{i,j} - v_{i+1,j} - v_{i-1,j} - v_{i,j+1} - v_{i,j-1})$$

```
for i, j in v:
  c = 1 / (dx * dx)
  u[i, j] = c * (4 * v[i, j] - v[i+1, j] -
                   v[i-1, j] - v[i, j+1] - v[i, j-1])
```

Data Structure Specification

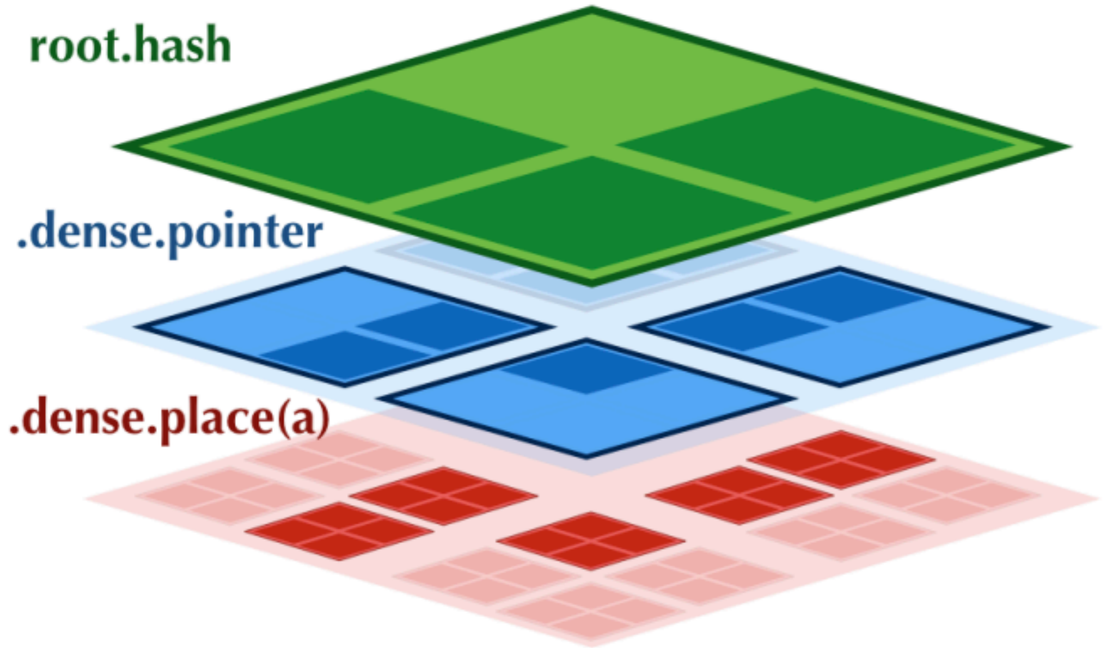
- Specify structure via tiled types and decorators
  - Types: dense, hashed, dynamic
  - Decorators*: morton, bitmasked, pointer

```
Level0 = Hash<(i, j), Level1>;

Level1 = (Level2*)[4]; // 2x2

Level2 = Value[4]; // 2x2
```



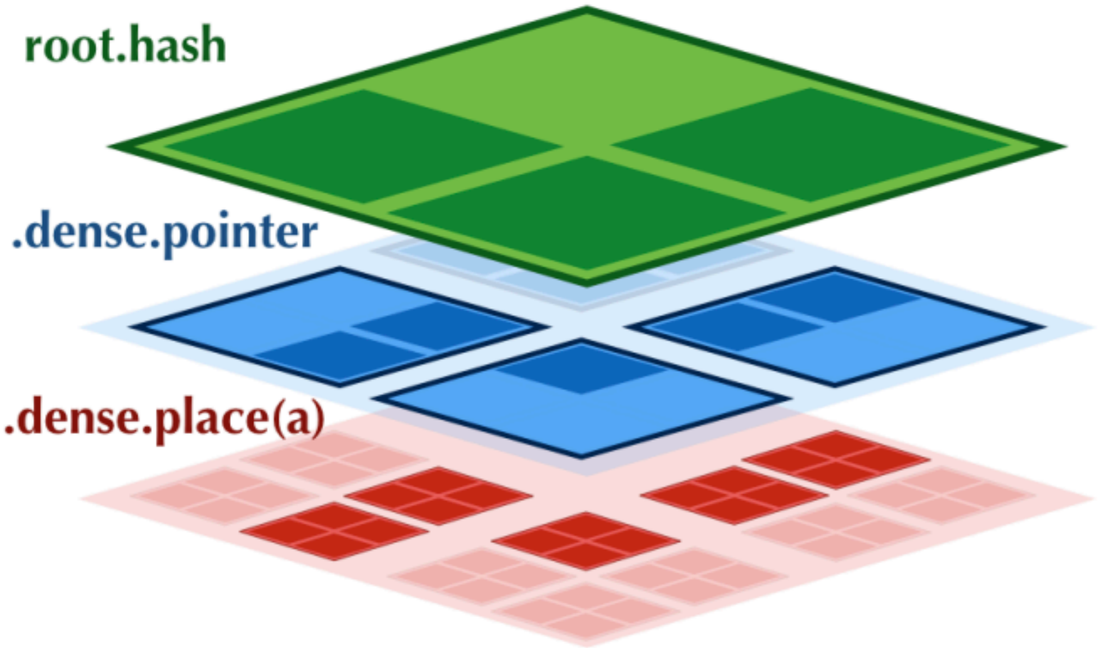
Generating Interface Code

```
Value read(Level0 root, int i, int j) {
  if (Level1 l0 = root.find(i, j)) {
    int i1 = i % 4, j1 = j % 4;
    if (l1[i1 * 2 + j1]) {
      int i2 = i1 % 2;
      int j2 = j1 % 2;
      return (*l1)[i2 * 2 + j2];
    }}
  return (Value)0; }
```



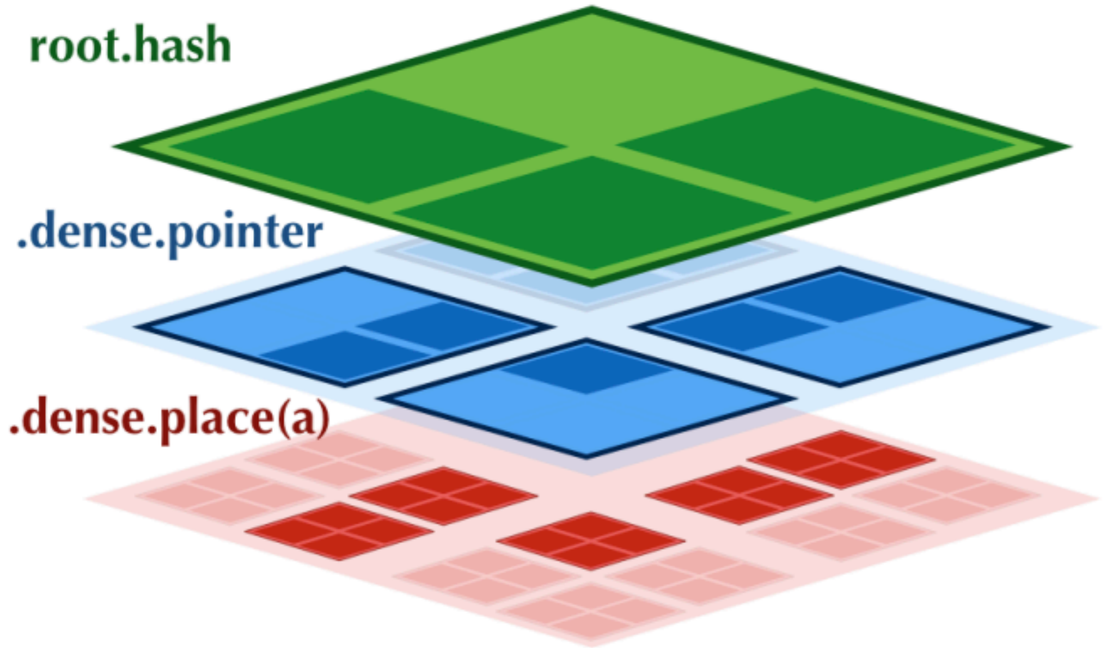
Generating Interface Code

```
Value read(Level0 root, int i, int j) {
  if (Level1 l0 = root.find(i, j)) {
    int i1 = i % 4, j1 = j % 4;
    if (l1[i1 * 2 + j1]) {
      int i2 = i1 % 2;
      int j2 = j1 % 2;
      return (*l1)[i2 * 2 + j2];
    }}
  return (Value)0; }
```



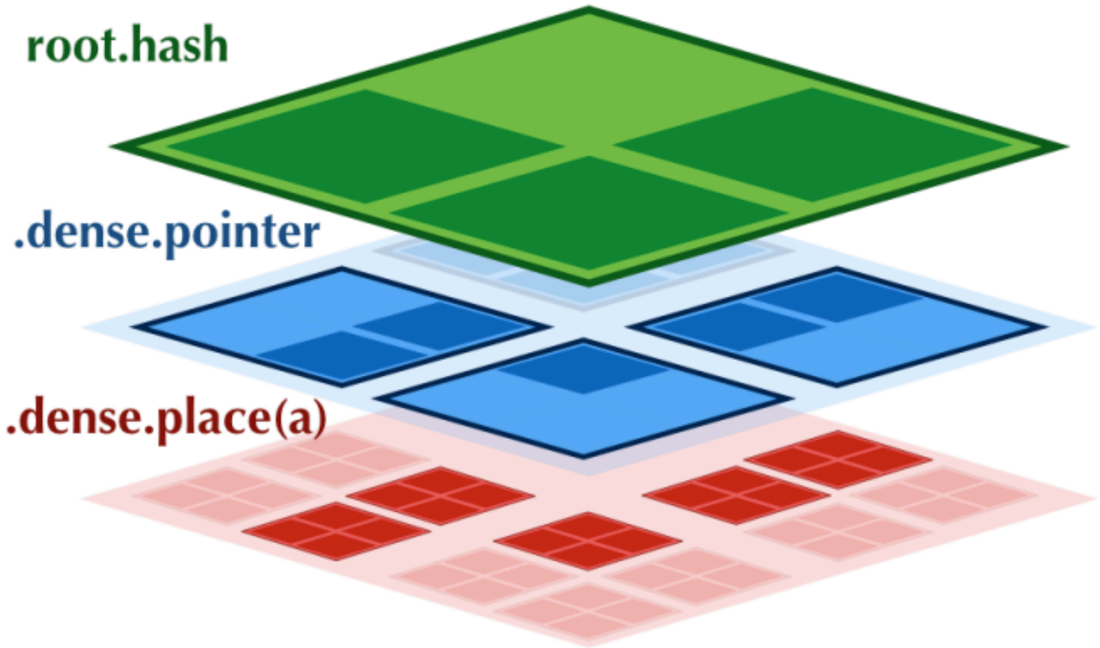
Generating Interface Code

```
Value read(Level0 root, int i, int j) {
  if (Level1 l0 = root.find(i, j)) {
    int i1 = i % 4, j1 = j % 4;
    if (l1[i1 * 2 + j1]) {
      int i2 = i1 % 2;
      int j2 = j1 % 2;
      return (*l1)[i2 * 2 + j2];
    }}
  return (Value)0; }
```



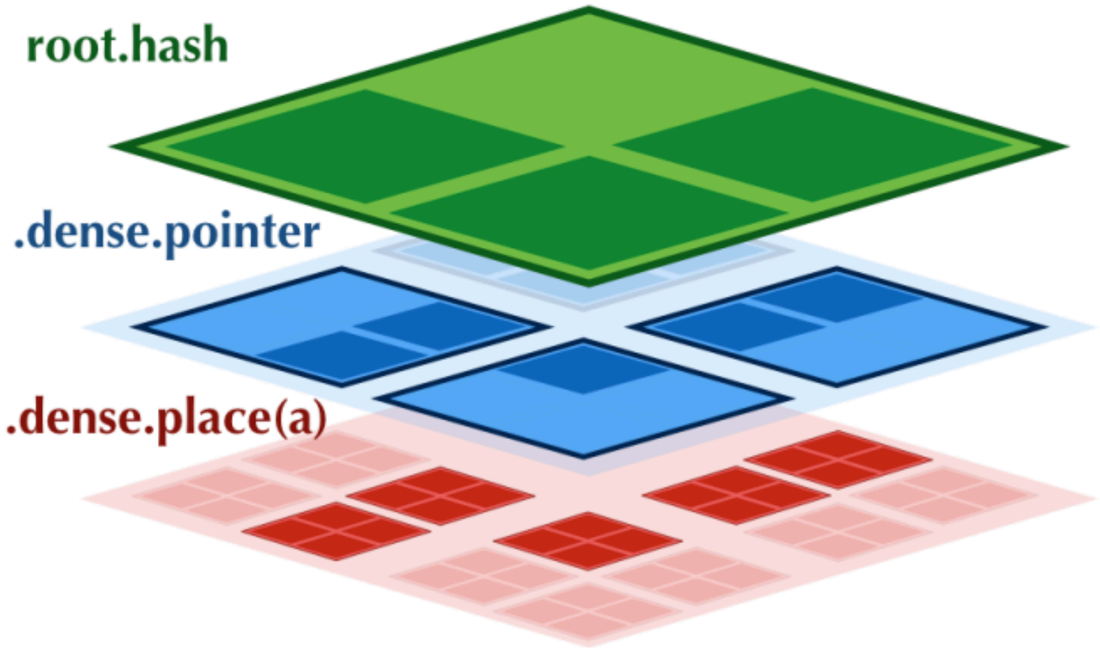
Generating Interface Code

```
Value read(Level0 root, int i, int j) {
  if (Level1 l0 = root.find(i, j)) {
    int i1 = i % 4, j1 = j % 4;
    if (l1[i1 * 2 + j1]) {
      int i2 = i1 % 2;
      int j2 = j1 % 2;
      return (*l1)[i2 * 2 + j2];
    }}
  return (Value)0; }
```



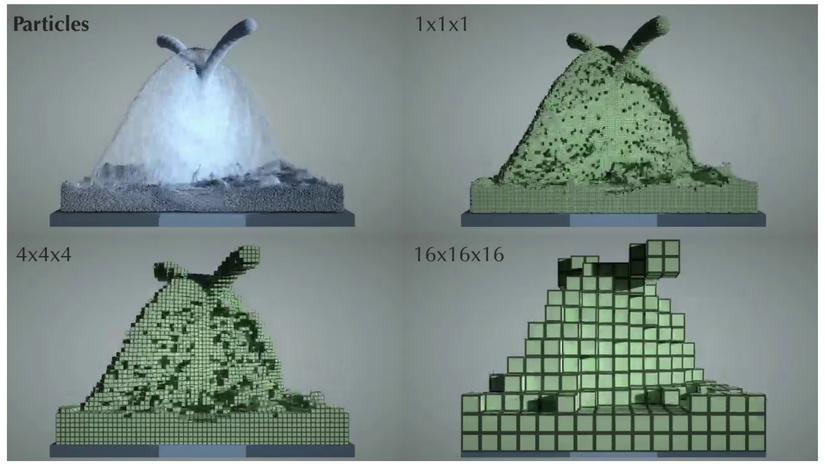
Generating Interface Code

```
Value read(Level0 root, int i, int j) {
  if (Level1 l0 = root.find(i, j)) {
    int i1 = i % 4, j1 = j % 4;
    if (l1[i1 * 2 + j1]) {
      int i2 = i1 % 2;
      int j2 = j1 % 2;
      return (*l1)[i2 * 2 + j2];
    }}
  return (Value)0; }
```



Results

- Excellent performance (+ pretty simulations)
- Capable of representing a large number of sparse grids from literature
  - SPGrid, OpenVDB, Hierarchical Particle Buckets, Hybrids, etc
- Algorithm code ***never changes***



Halide and Taichi

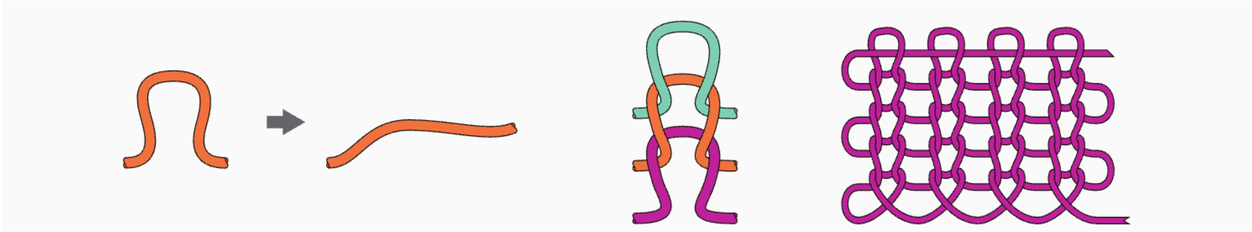
- Decouple algorithm from “performance optimizations”
- Allows writing an algorithm *once*
  - *But* need a (schedule | data structure) per machine
- Allows iteration over choosing the right optimizations *without changing program correctness*
- If you’re interested in these sorts of languages: take CS 343D!

Images, Simulations, Knitting, and Diagrams

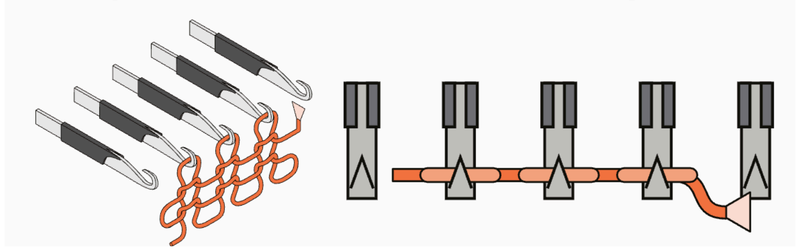
- Halide: A language for fast, portable computation on images
  - Decouple algorithms from optimizations for portability
- Taichi: A language for high-performance computation on spatially sparse data structures
  - Decouple algorithms from data structures for portability
- **Knitout: Low-level knitting machine instructions**
- Penrose: from mathematical notation to beautiful diagrams

Knitout: Low-level machine knitting instructions

- Some background:
  - Loops of yarn pulled through other loops can prevent unraveling, and produce structured fabric



  - A knitting machine uses needles (in *beds*) to keep loops in place until the yarn (attached to a *carrier*) can add new loops through them



images from Jenny Lin: https://textiles-lab.github.io/posts/2017/11/27/kout1/

Knitout: Low-level machine knitting instructions

- Needle Operations:
  - **tuck**: adds a loop to a needle
  - **knit**: pulls a loop through the loops of a needle
- Yarn operations:
  - **in**: bring a yarn carrier into action from grippers
  - **inhook**: select a specific yarn type on a hook
- Configuration operations:
  - **rack**: adjusts the alignments of the needle beds
  - **stitch**: adjusts the stitch size

A Knitout program

inhook 6 // pick yarn carrier 6

tuck - f20 6 // move left (-) to start on front bed needle #20, use carrier 6

tuck - f18 6 // tuck alternating needles

. . .

tuck + f1 6 // turn around and tuck alternating odd needles

. . .

knit - f20 6 // do the same with the **knit** command instead

. . . // alternate left (-) and right (+) knitting directions per row

A Knitout program output

image (and example) from Jenny Lin: https://textiles-lab.github.io/posts/2017/11/27/kout1/

Design Decisions

- An external (parsed) language
- Straight-line machine-executable code
- No loops, conditionals, etc. Just what the machine can do
- Multiple embedded versions developed (e.g. in Javascript) to **use metaprogramming for control flow**

```
carrier = 6, rows = N, columns = 20
knitout.inhook(carrier)
for (int j = 0; j < columns; j += 2) { knitout.tuck_left(j, carrier) }
for (int j = 1; j < columns; j += 1) { knitout.tuck_right(j, carrier) }
for (int i = 0; i < rows; i++) {
  for (int j = columns - 1; j >= 0; j--) { knitout.knit_left(j, carrier) }
  for (int j = 0; j < columns; j++) { knitout.knit_right(j, carrier) }
}
```

Images, Simulations, Knitting, and Diagrams

- Halide: A language for fast, portable computation on images
  - Decouple algorithms from optimizations for portability
- Taichi: A language for high-performance computation on spatially sparse data structures
  - Decouple algorithms from data structures for portability
- Knitout: Low-level knitting machine instructions
  - Knitting machines are programmable too
- **Penrose: from mathematical notation to beautiful diagrams**

Penrose: From Mathematical Notation to Beautiful Diagrams

Penrose: From Mathematical Notation to Beautiful Diagrams

KATHERINE YE, Carnegie Mellon University
WODE NI, Carnegie Mellon University
MAX KRIEGER, Carnegie Mellon University
DOR MA'AYAN, Technion and Carnegie Mellon University
JENNA WISE, Carnegie Mellon University
JONATHAN ALDRICH, Carnegie Mellon University
JOSHUA SUNSHINE, Carnegie Mellon University
KEENAN CRANE, Carnegie Mellon University

```
Point p, q, r, s
Segment a := {p, q}
Segment b := {p, r}
Point m := Midpoint(a)
Angle theta := ∠(q, p, r)
Triangle t := {p, r, s}
Ray w := Bisector(theta)
Ray h := PerpendicularBisector(a)
```

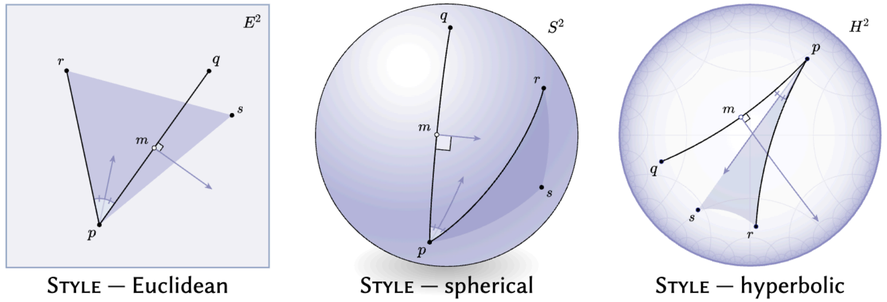


Fig. 1. Penrose is a framework for specifying how mathematical statements should be interpreted as visual diagrams. A clean separation between abstract mathematical objects and their visual representation provides new capabilities beyond existing code- or GUI-based tools. Here, for instance, the same set of statements *(left)* is given three different visual interpretations *(right)*, via Euclidean, spherical, and hyperbolic geometry. (Further samples are shown in Fig. 29.)

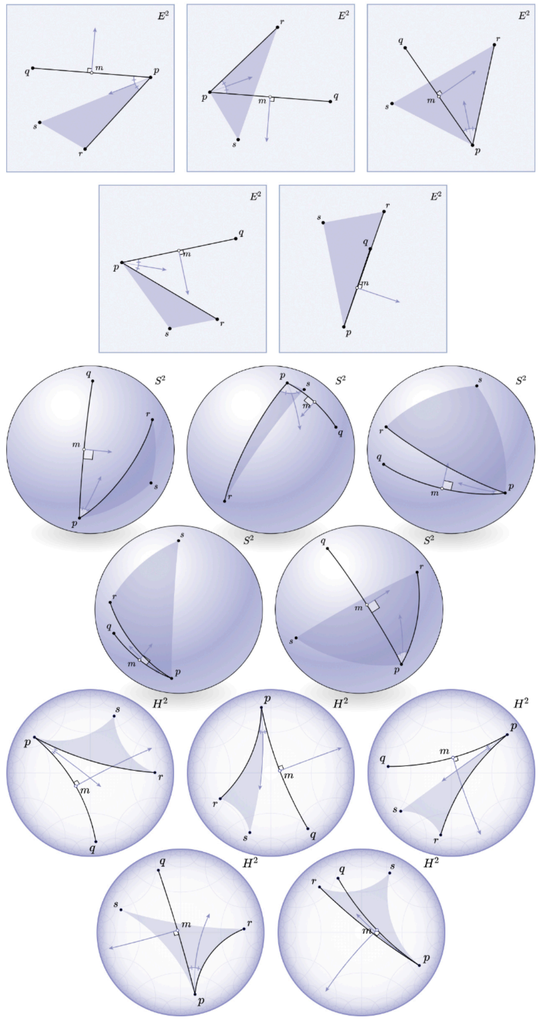


Fig. 29. Though the energy in the objective function can help guide a sampling strategy, choosing which diagram looks "best" to a given user is necessarily a subjective decision. The ability to automatically generate many alternatives makes it easier to find a satisfactory diagram. Here we show a gallery of automatically-generated variants of Fig. 1.

https://penrose.cs.cmu.edu/

Design Goals

- Mathematical objects should be expressed in a “familiar” way
- System should not be limited to a fixed set of domains
- Should be possible to apply many different visualizations to the same mathematical content
- No inherent limit on visual sophistication
- Should be fast enough to facilitate an iterative workflow
- Effort spent on diagramming should be generalizable and reusable

System Design

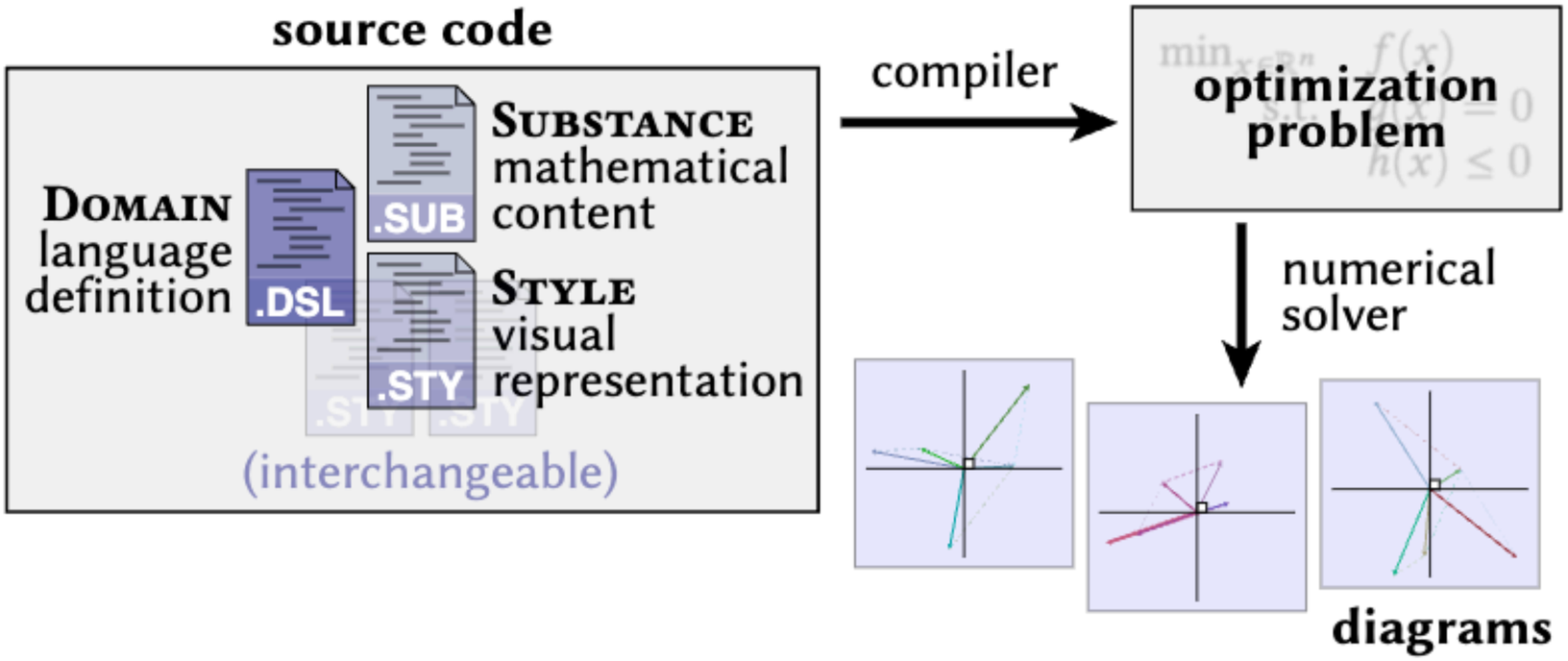


Fig. 5. Similar to the TEX ecosystem, most users need only use the SUBSTANCE language, but can benefit from packages written by more expert DOMAIN and STYLE programmers.

Language Framework

- **Domain** schema: declares objects, relationships, and notation available in a domain
- **Substance** program: makes specific mathematical assertions within some domain
- **Style** program: defines a generic mapping from mathematical statements in a domain to a visual representation

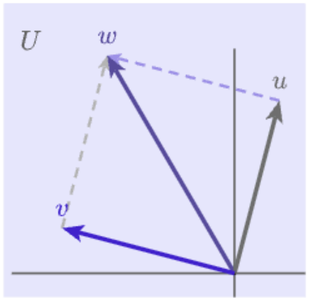
Domain Schema

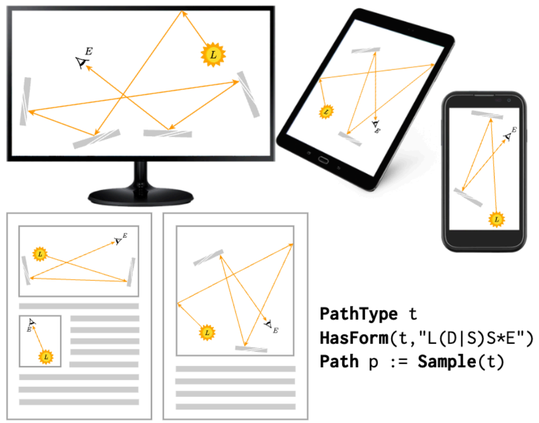
```
type Scalar, VectorSpace, Vector      -- LinearAlgebra.dsl
function add: Vector * Vector -> Vector
function norm: Vector -> Scalar
function scale: Scalar * Vector -> Vector
...
predicate In: Vector * VectorSpace
predicate Unit: Vector
predicate Orthogonal: Vector * Vector
...
notation "v1 + v2" ~ "add(v1,v2)"
notation "|y1|" ~ "norm(y1)"
notation "s * v1" ~ "scale(s,v1)"
notation "Vector v ∈ V" ~ "Vector a; In(a,U)"
notation "v1 ⊥ v2" ~ "Orthogonal(v1,v2)"
...
```

Substance Programs

For any vector space X, let $u, v \in X$ be orthogonal vectors of equal length, and let $w = u+v$. Then u and w make a $45°$ angle.

```
VectorSpace X
Vector u, v ∈ X
Orthogonal(u, v)
EqualLength(u, v)
Vector w ∈ X
w := u + v
```





```
PathType t
HasForm(t,"L(D|S)S*E")
Path p := Sample(t)
```

Style Programs

```
forall VectorSpace U {                -- LinearAlgebra.sty
   U.originX = ? -- to be determined via optimization
   U.originY = ? -- to be determined via optimization
   U.origin = (U.originX, U.originY)
   U.xAxis = Arrow { -- draw an arrow along the x-axis
          startX : U.originX - 1
          startY : U.originY
          endX   : U.originX + 1
          endY   : U.originY
      thickness   : 1.5
          style   : "solid"
          color   : Colors.lightGray
   } -- (similar declarations omitted for the y-axis)
}
forall Vector u, VectorSpace U -- match any vector
where In(u, U) {              -- in some vector space
   u.arrow = Arrow {
      startX : U.originX
      startY : U.originY
        endX : ?
        endY : ?
      color : Colors.mediumBlue
   }
   u.text = Text {
        string : u.label -- label from Substance code
         color : u.arrow.color  -- use arrow's color
             x : ?
             y : ?
   }
   u.start = (u.arrow.startX, u.arrow.startY)
   u.end = (u.arrow.endX, u.arrow.endY)
   u.vector = minus(u.arrow.end, u.arrow.start)
   encourage near(u.text, u.end)
   ensure contained(u.end, U.shape)
}
forall Vector u, Vector v
where Orthogonal(u, v) {
   local.perpMark = Curve {
        pathData : orientedSquare(u.shape, v.shape,
                                  U.origin, const.perpLen)
        strokeWidth : 2.0
        color : Colors.black
        fill : Colors.white
   }
     ensure equals(dot(u.vector, v.vector), 0.0)
}
... -- (similar rule omitted for Unit)
Vector `x2`  {
   override `x2`.shape.color = Colors.green;
}
```

Style Programs

```
forall VectorSpace U {                -- LinearAlgebra.sty
   U.originX = ? -- to be determined via optimization
   U.originY = ? -- to be determined via optimization
   U.origin = (U.originX, U.originY)
   U.xAxis = Arrow { -- draw an arrow along the x-axis
          startX : U.originX - 1
          startY : U.originY
          endX   : U.originX + 1
          endY   : U.originY
      thickness   : 1.5
          style   : "solid"
          color   : Colors.lightGray
   } -- (similar declarations omitted for the y-axis)
}
forall Vector u, VectorSpace U    -- match any vector
where In(u, U) {                  -- in some vector space
   u.arrow = Arrow {
      startX : U.originX
      startY : U.originY
        endX : ?
        endY : ?
      color : Colors.mediumBlue
   }
   u.text = Text {
        string : u.label -- label from Substance code
         color : u.arrow.color  -- use arrow's color
             x : ?
             y : ?
   }
   u.start = (u.arrow.startX, u.arrow.startY)
   u.end = (u.arrow.endX, u.arrow.endY)
   u.vector = minus(u.arrow.end, u.arrow.start)
   encourage near(u.text, u.end)
   ensure contained(u.end, U.shape)
}
forall Vector u, Vector v
where Orthogonal(u, v) {
   local.perpMark = Curve {
        pathData : orientedSquare(u.shape, v.shape,
                                  U.origin, const.perpLen)
        strokeWidth : 2.0
        color : Colors.black
        fill : Colors.white
   }
     ensure equals(dot(u.vector, v.vector), 0.0)
}
... -- (similar rule omitted for Unit)
Vector `x2`  {
   override `x2`.shape.color = Colors.green;
}
```

Style Programs

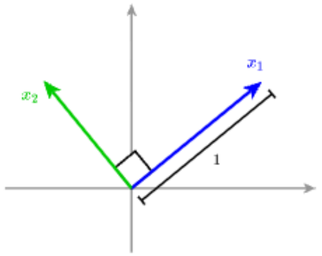
```
forall VectorSpace U {                -- LinearAlgebra.sty
   U.originX = ? -- to be determined via optimization
   U.originY = ? -- to be determined via optimization
   U.origin = (U.originX, U.originY)
   U.xAxis = Arrow { -- draw an arrow along the x-axis
          startX : U.originX - 1
          startY : U.originY
          endX   : U.originX + 1
          endY   : U.originY
      thickness   : 1.5
          style   : "solid"
          color   : Colors.lightGray
   } -- (similar declarations omitted for the y-axis)
}
forall Vector u, VectorSpace U -- match any vector
where In(u, U) {              -- in some vector space
   u.arrow = Arrow {
      startX : U.originX
      startY : U.originY
        endX : ?
        endY : ?
      color : Colors.mediumBlue
   }
   u.text = Text {
        string : u.label -- label from Substance code
         color : u.arrow.color  -- use arrow's color
             x : ?
             y : ?
   }
   u.start = (u.arrow.startX, u.arrow.startY)
   u.end = (u.arrow.endX, u.arrow.endY)
   u.vector = minus(u.arrow.end, u.arrow.start)
   encourage near(u.text, u.end)
   ensure contained(u.end, U.shape)
}
forall Vector u, Vector v
where Orthogonal(u, v) {
   local.perpMark = Curve {
        pathData : orientedSquare(u.shape, v.shape,
                                  U.origin, const.perpLen)
        strokeWidth : 2.0
        color : Colors.black
        fill : Colors.white
   }
     ensure equals(dot(u.vector, v.vector), 0.0)
}
... -- (similar rule omitted for Unit)
Vector `x2`  {
   override `x2`.shape.color = Colors.green;
}
```

```
VectorSpace X
Vector x1, x2
In(x1, X)
In(x2, X)
Unit(x1)
Orthogonal(x1, x2)
label x1 $x_1$
label x2 $x_2$
```

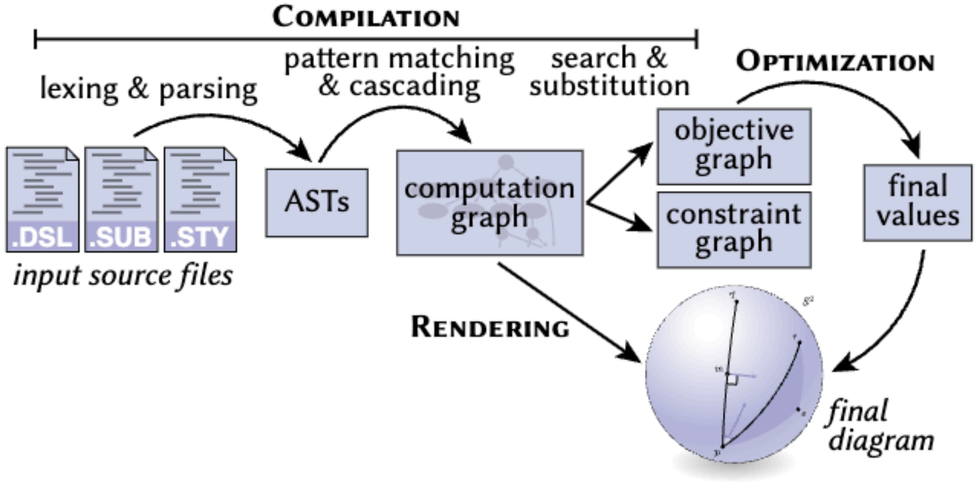
(unsugared)

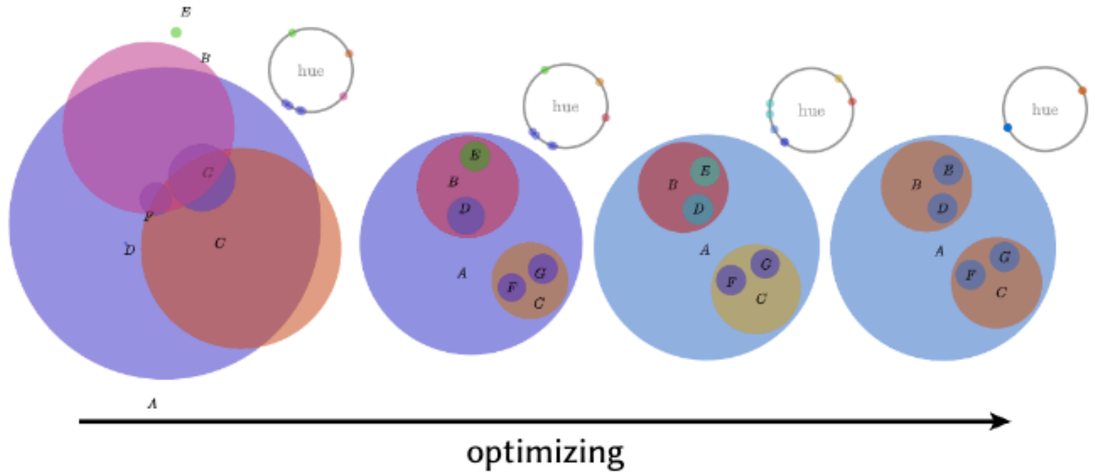
```
VectorSpace X
Vector x1, x2 ∈ X
Unit(x1)
x1 ⊥ x2
label x1 $x_1$
label x2 $x_2$
```

(sugared)



Layout Engine



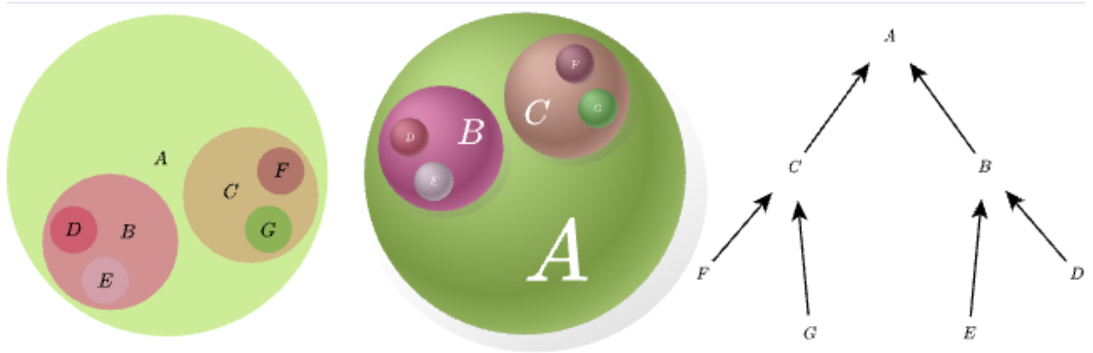


Another example: Sets

```
type Set                                        -- Sets.dsl
predicate Intersecting : Set s1 * Set s2
predicate IsSubset : Set s1 * Set s2
predicate Not : Prop p
notation "A ⊂ B" ~ "IsSubset(A, B)"
notation "A ∩ B = ∅" ~ "Not(Intersecting(A, B))"
```

```
Set A, B, C, D, E, F, G        F ⊂ C        -- Sets.sub
B ⊂ A                          G ⊂ C
C ⊂ A                          E ∩ D = ∅
D ⊂ B                          F ∩ G = ∅
E ⊂ B                          B ∩ C = ∅
```

```
forall Set x {                              -- Sets-Disks.sty
    x.shape = Circle { strokeWidth : 0.0 }
    x.text = Text { string : x.label }
    ensure contains(x.shape, x.text)
    encourage sameCenter(x.text, x.shape)
    layer x.shape below x.text
}
forall Set x; Set y
where IsSubset(x, y) {
    ensure contains(y.shape, x.shape)
    ensure smallerThan(x.shape, y.shape)
    ensure outsideOf(y.text, x.shape)
    layer x.shape above y.shape
    layer y.text below x.shape
}
forall Set x; Set y
where NotIntersecting(x, y) {
    ensure disjoint(x.shape, y.shape)
}
```



How did they evaluate?

- Case studies:
  - Sets
  - Functions
  - Geometry
  - Linear Algebra
  - Meshes
  - Ray Tracing
- Performance reporting (not really comparisons)

Images, Simulations, Knitting, and Diagrams

- Halide: A language for fast, portable computation on images
  - Decouple algorithms from optimizations for portability
- Taichi: A language for high-performance computation on spatially sparse data structures
  - Decouple algorithms from data structures for portability
- Knitout: Low-level knitting machine instructions
  - Knitting machines are programmable too
- Penrose: from mathematical notation to beautiful diagrams
  - Mathematical visualizations are multiple DSLs of their own