Diderot: A Parallel DSL for Image Analysis and Visualization

Charisee Chiw
Gordon Kindlmann
John Reppy
Lamont Samuels
Nick Seltzer

University of Chicago

June 11, 2012

Diderot

The Diderot project is a collaborative effort to use ideas from PL to improve the state-of-the-art in scientific image analysis and visualization.

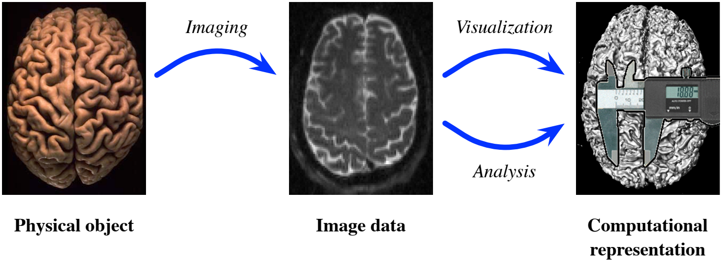
We have two main goals for Diderot:

- Improve programmability by supporting a high-level mathematical programming notation.
- Improve performance by supporting efficient execution; especially on parallel platforms.

Roadmap

- Image analysis
- Parallel DSLs
- Diderot design and examples
- Implementation issues
- Performance
- Conclusion

Why image analysis is important

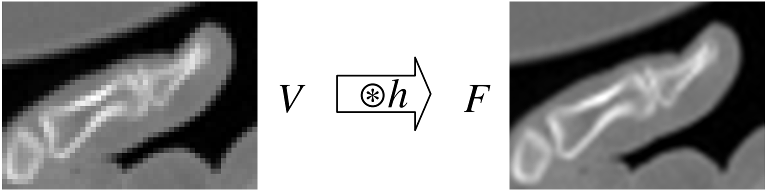


- Scientists need software tools to extract structure from many kinds of image data.
- Creating new analysis/visualization programs is part of the experimental process.
- The challenge of getting knowledge from image data is getting harder.

Image analysis and visualization

- We are interested in a class of algorithms that compute geometric properties of objects from imaging data.
- These algorithms compute over a continuous tensor field F (and its derivatives), which are reconstructed from discrete data using a separable convolution kernel h:

$$F = V \circledast h$$



Discrete image data *Continuous field*

Image analysis and visualization

Example applications include

- Direct volume rendering (requires reconstruction, derivatives).
- Fiber tractography (requires tensor fields).
- Particle systems (requires dynamic numbers of computational elements).

Image analysis and visualization

Example applications include

- Direct volume rendering (requires reconstruction, derivatives).
- Fiber tractography (requires tensor fields).
- Particle systems (requires dynamic numbers of computational elements).

Image analysis and visualization

Example applications include

- Direct volume rendering (requires reconstruction, derivatives).
- Fiber tractography (requires tensor fields).
- Particle systems (requires dynamic numbers of computational elements).

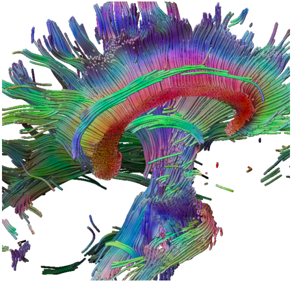
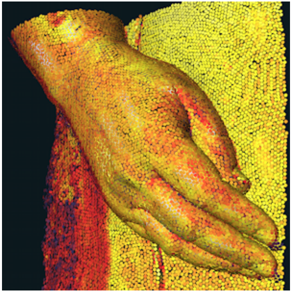


Image analysis and visualization

Example applications include

- Direct volume rendering (requires reconstruction, derivatives).
- Fiber tractography (requires tensor fields).
- Particle systems (requires dynamic numbers of computational elements).



Parallel DSLs

Domain-specific languages provide a number of advantages:

- High-level notation supports rapid prototyping and pedagogical presentation.
- Opportunities for domain-specific optimizations.

Parallel DSLs provide additional advantages

- High-level, abstract, parallelism models.
- Portable parallelism.

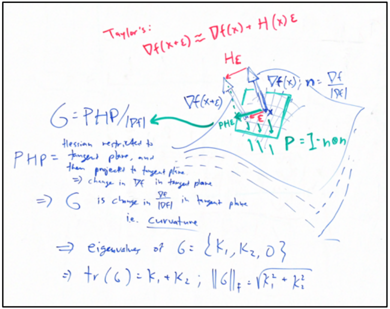
Parallel DSLs meet the Diderot design goals of improving programmability and performance.

Related work

Other examples of parallel DSLs:

- Liszt: embedded DSL for writing mesh-based PDE solvers.
- Shadie: DSL for volume rendering applications.
- Spiral: program generator for DSP code.

Programmability: from whiteboard to code




```
vec3 grad = -∇F(pos);
vec3 norm = normalize(grad);
tensor[3,3] H = ∇⊗∇F(pos);
tensor[3,3] P = identity[3] - norm⊗norm;
tensor[3,3] G = -(P•H•P)/|grad|;
real disc = sqrt(2.0*|G|^2 - trace(G)^2);
real k1 = (trace(G) + disc)/2.0;
real k2 = (trace(G) - disc)/2.0;
```

Diderot program structure

Square roots of integers using Heron's method.

```
// global definitions
input int N = 1000;
input real eps = 0.000001;

// strand definition
strand SqRoot (real val)
{
    output real root = val;

    update {
        root = (root + val/root) / 2.0;
        if (|root^2 - val|/val < eps)
            stabilize;
    }
}

// initialization
initially [ SqRoot(real(i)) | i in 1..N ]
```

Diderot program structure

Square roots of integers using Heron's method.

```
// global definitions
input int N = 1000;
input real eps = 0.000001;

// strand definition
strand SqRoot (real val)
{
    output real root = val;

    update {
        root = (root + val/root) / 2.0;
        if (|root^2 - val|/val < eps)
            stabilize;
    }
}

// initialization
initially [ SqRoot(real(i)) | i in 1..N ]
```

Globals are *immutable*, and are used for *program inputs* and other shared globals.

Diderot program structure

Square roots of integers using Heron's method.

```
// global definitions
input int N = 1000;
input real eps = 0.000001;

// strand definition
strand SqRoot (real val)
{
    output real root = val;

    update {
        root = (root + val/root) / 2.0;
        if (|root^2 - val|/val < eps)
            stabilize;
    }
}

// initialization
initially [ SqRoot(real(i)) | i in 1..N ]
```

Strands are the elements of a *bulk synchronous* computation.

Diderot program structure

Square roots of integers using Heron's method.

```
// global definitions
input int N = 1000;
input real eps = 0.000001;

// strand definition
strand SqRoot (real val)
{
    output real root = val;

    update {
        root = (root + val/root) / 2.0;
        if (|root^2 - val|/val < eps)
            stabilize;
    }
}

// initialization
initially [ SqRoot(real(i)) | i in 1..N ]
```

Strands have *parameters* that are used to initialize them.

Strands have *state*, which includes *outputs*.

Diderot program structure

Square roots of integers using Heron's method.

```
// global definitions
input int N = 1000;
input real eps = 0.000001;

// strand definition
strand SqRoot (real val)
{
    output real root = val;

    update {
        root = (root + val/root) / 2.0;
        if (|root^2 - val|/val < eps)
            stabilize;
    }
}

// initialization
initially [ SqRoot(real(i)) | i in 1..N ]
```

Strands have an *update method* that is invoked each *super step*.

Diderot program structure

Square roots of integers using Heron's method.

```
// global definitions
input int N = 1000;
input real eps = 0.000001;

// strand definition
strand SqRoot (real val)
{
    output real root = val;

    update {
        root = (root + val/root) / 2.0;
        if (|root^2 - val|/val < eps)
            stabilize;
    }
}

// initialization
initially [ SqRoot(real(i)) | i in 1..N ]
```

Strands have an *update method* that is invoked each *super step*.

Strands can *stabilize* or *die* during the computation.

Diderot program structure

Square roots of integers using Heron's method.

```
// global definitions
input int N = 1000;
input real eps = 0.000001;

// strand definition
strand SqRoot (real val)
{
    output real root = val;

    update {
        root = (root + val/root) / 2.0;
        if (|root^2 - val|/val < eps)
            stabilize;
    }
}

// initialization
initially [ SqRoot(real(i)) | i in 1..N ]
```

The initial collection of strands is created using *comprehension notation*.

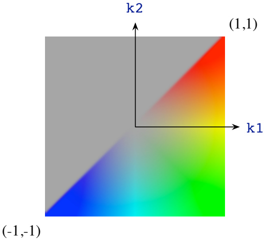
Diderot design summary

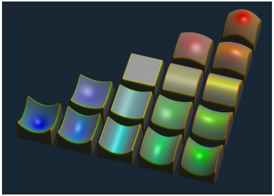
The Diderot language design has two major aspects:

- A high-level mathematical programming model that uses the concepts and direct-style notation of tensor calculus to work with image data. These include tensor operations ($\bullet$, $\times$) and higher-order field operations (∇), *etc.*
- A shared-nothing bulk-synchronous parallel execution model that abstracts away from details of communication, synchronization, and resource management.

Example — Curvature

```
field#2(3)[] F = bspln3 ⊛ load("quad-patches.nrrd");
field#0(2)[3] RGB = tent ⊛ load("2d-bow.nrrd");
...
strand RayCast (int ui, int vi) {
  ...
  update {
    ...
    vec3 grad = -∇F(pos);
    vec3 norm = normalize(grad);
    tensor[3,3] H = ∇ ⊗ ∇F(pos);
    tensor[3,3] P = identity[3] - norm⊗norm;
    tensor[3,3] G = -(P•H•P)/|grad|;
    real disc = sqrt(2.0*|G|^2 - trace(G)^2);
    real k1 = (trace(G) + disc)/2.0;
    real k2 = (trace(G) - disc)/2.0;
    vec3 matRGB = // material RGBA
        RGB([max(-1.0, min(1.0, 6.0*k1)),
             max(-1.0, min(1.0, 6.0*k2))]);
    ...
  }
...
}
```





Example — 2D Isosurface

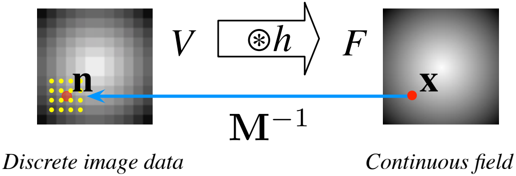
```
int stepsMax = 10;
...
strand sample (int ui, int vi) {
  output vec2 pos = ···;
// set isovalue to closest of 50, 30, or 10
  real isoval = 50.0 if F(pos) >= 40.0
              else 30.0 if F(pos) >= 20.0
              else 10.0;
  int steps = 0;
  update {
    if (inside(pos, F) && steps <= stepsMax) {
    // delta = Newton-Raphson step
      vec2 delta = normalize(∇F(pos)) * (F(pos) - isoval)/|∇F(pos)|;
      if (|delta| < epsilon)
        stabilize;
      pos = pos - delta;
      steps = steps + 1;
    }
    else die;
  }
}
```

Diderot compiler and runtime

- Compiler is about 21,000 lines of SML (2,500 in front-end).
- Multiple backends: vectorized C and OpenCL (CUDA under construction).
- Multiple runtimes: Sequential C, Parallel C, OpenCL.
- Designed to generate libraries, but also supports standalone executables.

Probing tensor fields

A probe gets compiled down into code that maps the world-space coordinates to image space and then convolves the image values in the neighborhood of the position.



In 2D, the reconstruction is (note that h is separable)

$$F(\mathbf{x}) = \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[\mathbf{n} + \langle i,j \rangle] h(\mathbf{f}_x - i) h(\mathbf{f}_y - j)$$

where s is the support of h, $\mathbf{n} = \lfloor \mathbf{M}^{-1}\mathbf{x} \rfloor$ and $\mathbf{f} = \mathbf{M}^{-1}\mathbf{x} - \mathbf{n}$.

Probing tensor fields *(continued ...)*

In general, compiling the probe operations is more challenging.

For example, we might have

```
field#2(2)[] F = h ⊛ V;

··· ∇(s * F)(x) ···
```

The first step is to normalize the field expressions.

$$\begin{aligned} \nabla(s * (V \circledast h))(x) &\Rightarrow (s * (\nabla(V \circledast h)))(x) \\ &\Rightarrow s * ((\nabla(V \circledast h))(x)) \\ &\Rightarrow s * (V \circledast (\nabla h))(x) \end{aligned}$$

Probing tensor fields *(continued ...)*

Each component in the partial-derivative tensor corresponds to a component in the result of the probe.

$$\begin{aligned}
\nabla(s * F)(x) &= s * (V \circledast (\nabla h))(x) \\
&= s * (V \circledast \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} h)(x) \\
&= s * \begin{bmatrix} \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[\mathbf{n} + \langle i,j \rangle] \, h' \, (\mathbf{f}_x - i) \, h(\mathbf{f}_y - j) \\ \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[\mathbf{n} + \langle i,j \rangle] \, h(\mathbf{f}_x - i) \, h' \, (\mathbf{f}_y - j) \end{bmatrix}
\end{aligned}$$

A later stage of the compiler expands out the evaluations of h and h'.

Probing code has high arithmetic intensity and is trivial to vectorize.

Experimental framework

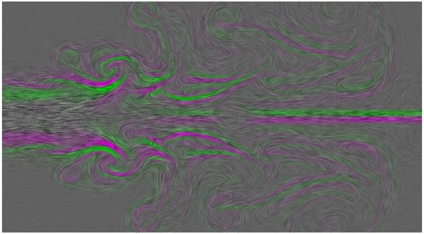
- SMP machine: 8-core MacPro with 2.93 GHz Xeon X5570 processors (SSE-4)
- Four typical benchmark programs
  - **vr-lite** — simple volume-renderer with Phong shading running on CT scan of hand
  - **illust-vr** — fancy volume-renderer with cartoon shading running on CT scan of hand
  - **lic2d** — line integral convolution in 2D running on turbulance data
  - **ridge3d** — particle-based ridge detection running on lung data

Experimental framework

- SMP machine: 8-core MacPro with 2.93 GHz Xeon X5570 processors (SSE-4)
- Four typical benchmark programs
  - **vr-lite** — simple volume-renderer with Phong shading running on CT scan of hand
  - **illust-vr** — fancy volume-renderer with cartoon shading running on CT scan of hand
  - **lic2d** — line integral convolution in 2D running on turbulance data
  - **ridge3d** — particle-based ridge detection running on lung data

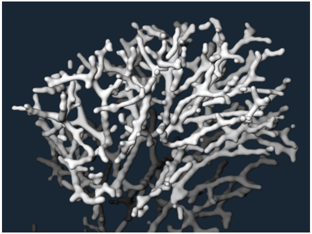
Experimental framework

- SMP machine: 8-core MacPro with 2.93 GHz Xeon X5570 processors (SSE-4)
- Four typical benchmark programs
  - **vr-lite** — simple volume-renderer with Phong shading running on CT scan of hand
  - **illust-vr** — fancy volume-renderer with cartoon shading running on CT scan of hand
  - **lic2d** — line integral convolution in 2D running on turbulance data
  - **ridge3d** — particle-based ridge detection running on lung data



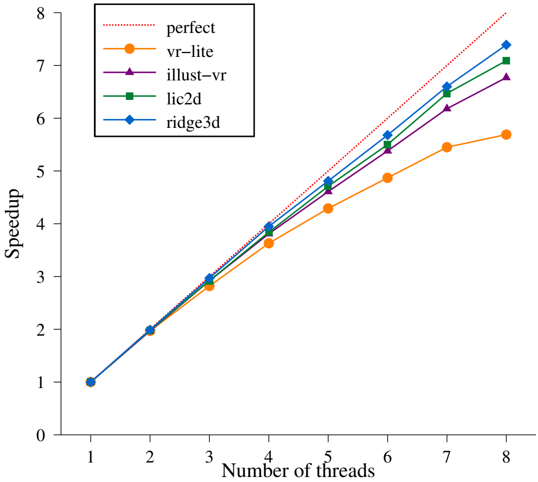
Experimental framework

- SMP machine: 8-core MacPro with 2.93 GHz Xeon X5570 processors (SSE-4)
- Four typical benchmark programs
  - **vr-lite** — simple volume-renderer with Phong shading running on CT scan of hand
  - **illust-vr** — fancy volume-renderer with cartoon shading running on CT scan of hand
  - **lic2d** — line integral convolution in 2D running on turbulance data
  - **ridge3d** — particle-based ridge detection running on lung data



SMP scaling

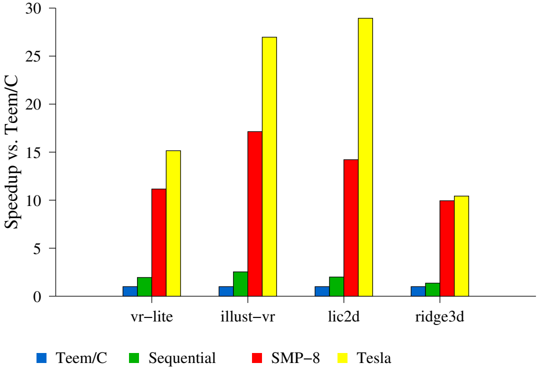
Parallel performance scaling with respect to sequential Diderot.



Comparison across platforms

Compare performance on three platforms: sequential (MacPro), 8-way parallel (MacPro), and NVIDIA Tesla C2070.

Baseline is Teem/C implementation on MacPro.



Conclusion

Diderot provides:

- High-level programming notation.
- Domain-specific optimizations.
- Portable parallel performance.

These advantages apply to Parallel DSLs in general!

Thanks to NVIDIA and AMD for their support.

Questions?

http://diderot-language.cs.uchicago.edu