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 $h$:

\[ F = V \ast h \]
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.
vec3 grad = -\nabla F(pos);
vec3 norm = normalize(grad);
tensor[3,3] H = \nabla \otimes \nabla F(pos);
tensor[3,3] G = -(P \cdot H \cdot P) / |\text{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.

```diderot
// 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 ]
```

June 11, 2012
Diderot program structure

Square roots of integers using Heron’s method.

```diderot
// 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.
Square roots of integers using Heron’s method.

```plaintext
// 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.

```diderot
// 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 (|rootˆ2 - 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.

```diderot
// 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.

```diderot
// 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 ($\nabla$), etc.

- A shared-nothing bulk-synchronous parallel execution model that abstracts away from details of communication, synchronization, and resource management.
Example — Curvature

\textbf{field}\#2(3)[] \( F = \text{bspln3} \odot \text{load("quad-patches.nrrd")}; \)
\textbf{field}\#0(2)[3] \( \text{RGB} = \text{tent} \odot \text{load("2d-bow.nrrd")}; \)
...

\textbf{strand} RayCast (\textbf{int} \ui, \textbf{int} \vi) {
  ... 
  \textbf{update} { 
    ... 
    \textbf{vec3} \quad \text{grad} = -\nabla F(pos); 
    \textbf{vec3} \quad \text{norm} = \text{normalize}(\text{grad}); 
    \textbf{tensor}[3,3] \quad H = \nabla \otimes \nabla F(pos); 
    \textbf{tensor}[3,3] \quad P = \text{identity}[3] - \text{norm} \otimes \text{norm}; 
    \textbf{tensor}[3,3] \quad G = -(P \cdot H \cdot P) / |\text{grad}|; 
    \textbf{real} \quad \text{disc} = \sqrt{2.0 \times |G|^2 - \text{trace}(G)^2}; 
    \textbf{real} \quad \text{k1} = (\text{trace}(G) + \text{disc})/2.0; 
    \textbf{real} \quad \text{k2} = (\text{trace}(G) - \text{disc})/2.0; 
    \textbf{vec3} \quad \text{matRGB} = // \text{material RGBA} 
      \quad \text{RGB}([\text{max}(-1.0, \text{min}(1.0, 6.0 \times \text{k1})), 
        \quad \text{max}(-1.0, \text{min}(1.0, 6.0 \times \text{k2})]); 
    ... 
  } 
  ... 
}

June 11, 2012
Example — 2D Isosurface

```diderot
int stepsMax = 10;

strand sample (int ui, int vi) {
    output vec2 pos = ...;
    // set iso-value 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;
    }
}
```
Implementation issues

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)

\[
F(x) = \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[n + \langle i, j \rangle]h(f_x - i)h(f_y - j)
\]

where \( s \) is the support of \( h \), \( n = [M^{-1}x] \) and \( f = M^{-1}x - n \).
Probing tensor fields (*continued ...*)

In general, compiling the probe operations is more challenging.

For example, we might have

\[
\text{field}^2 \text{[2]} F = h \otimes V;
\]

\[\cdots \nabla (s \ast F)(x) \cdots\]

The first step is to normalize the field expressions.

\[
\nabla(s \ast (V \otimes h))(x) \Rightarrow (s \ast (\nabla(V \otimes h)))(x) \\
\Rightarrow s \ast ((\nabla(V \otimes h))(x)) \\
\Rightarrow s \ast (V \otimes (\nabla h))(x)
\]
Probing tensor fields (continued ...)

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

\[
\nabla (s * F)(x) = s * (V \odot (\nabla h))(x)
\]

\[
= s * (V \odot \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix}) h(x)
\]

\[
= s * \left[ \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[n + \langle i, j \rangle] h'(f_x - i) h(f_y - j) \right]
\]

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 turbulence 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 turbulence 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 turbulence 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 turbulence 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