Session 2903

John Reppy
University of Chicago

Diderot: A Parallel Domain-Specific Language for Image Analysis and Visualization
Introduction
Diderot

Diderot is a cross-discipline project involving

• Scientific visualization
• Programming languages
Diderot

Diderot is a cross-discipline project involving

- Scientific visualization
- Programming languages

Our goal is to use ideas from programming languages to improve the state of the art in image-analysis and visualization.
Diderot is a cross-discipline project involving

• Scientific visualization
• Programming languages

Our goal is to use ideas from programming languages to improve the state of the art in image-analysis and visualization.

Joint work with Gordon Kindlmann, Lamont Samuels, and Nick Seltzer.
Why image analysis is important
Why image analysis is important

Physical object -> Imaging

Image data -> Visualization

Computational representation -> Analysis
Why image analysis is important

Scientists need tools to extract structure from many kinds of image data.
Image analysis and visualization
Image analysis and visualization

We are interested in a class of algorithms that compute a geometric properties of some object from imaging data.
Image analysis and visualization

We are interested in a class of algorithms that compute geometric properties of some object from imaging data.

These algorithms compute over a continuous field that is reconstructed from discrete data.
Image analysis and visualization

Examples of Diderot applications include:
Examples of Diderot applications include:

- Direct volume rendering (requires reconstruction, derivatives)
Image analysis and visualization

Examples of Diderot applications include:

- Direct volume rendering (requires reconstruction, derivatives)
- Fiber tractography (requires tensor fields)
Image analysis and visualization

Examples of Diderot applications include:

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

Diderot provides *domain-specific* support for image analysis and visualization algorithms.
Design goals

Diderot provides *domain-specific* support for image analysis and visualization algorithms.

Two main design goals:
Design goals

Diderot provides *domain-specific* support for image analysis and visualization algorithms.

Two main design goals:

• Provide a high-level mathematical programming model that abstracts away from discrete image data and the target architecture.
Design goals

Diderot provides *domain-specific* support for image analysis and visualization algorithms.

Two main design goals:
- Provide a high-level mathematical programming model that abstracts away from discrete image data and the target architecture.
- Use domain knowledge to get good performance on a range of parallel platforms.
Diderot programming model
The Diderot programming model is based on a collection of mostly autonomous strands that are embedded in a continuous tensor field.
Diderot programming model

- The Diderot programming model is based on a collection of mostly autonomous *strands* that are embedded in a *continuous tensor field*.
- Each strand has an *update* method, which encapsulates the computational kernel of the algorithm.
The Diderot programming model is based on a collection of mostly autonomous *strands* that are embedded in a *continuous tensor field*. Each strand has an *update* method, which encapsulates the computational kernel of the algorithm. Diderot abstracts away from details such as the discrete image-data, the representation of reals (float vs double), and the target machine (e.g., CPU vs GPU).
The Diderot programming model is based on a collection of mostly autonomous strands that are embedded in a continuous tensor field. Each strand has an update method, which encapsulates the computational kernel of the algorithm. Diderot abstracts away from details such as the discrete image-data, the representation of reals (float vs double), and the target machine (e.g., CPU vs GPU). The computation of a Diderot program is expressed using the concepts and direct-style notation of tensor calculus.
Diderot parallelism model

Bulk synchronous model with deterministic semantics.
Diderot parallelism model

Bulk synchronous model with deterministic semantics.
Diderot parallelism model

Bulk synchronous model with deterministic semantics.
Diderot parallelism model

Bulk synchronous model with deterministic semantics.
Diderot parallelism model

Bulk synchronous model with deterministic semantics.
Diderot parallelism model

Bulk synchronous model with deterministic semantics.
Diderot parallelism model

Bulk synchronous model with deterministic semantics.
Diderot program structure
Diderot program structure

A Diderot program has three main parts:
Diderot program structure

A Diderot program has three main parts:

- Global variable definitions, including program inputs.
Diderot program structure

A Diderot program has three main parts:

• Global variable definitions, including program inputs.
• Strand definitions.
Diderot program structure

A Diderot program has three main parts:

• Global variable definitions, including program inputs.
• Strand definitions.
  • Each strand has a *state*, which includes its *position*. 
A Diderot program has three main parts:

- Global variable definitions, including program inputs.
- Strand definitions.
  - Each strand has a `state`, which includes its `position`.
  - Some state variables are annotated as `outputs`. 
Diderot program structure

A Diderot program has three main parts:

• Global variable definitions, including program inputs.
• Strand definitions.
  • Each strand has a `state`, which includes its `position`.
  • Some state variables are annotated as `outputs`.
  • Each strand has methods, including `update` and an optional `stabilize` method.
Diderot program structure

A Diderot program has three main parts:

• Global variable definitions, including program inputs.
• Strand definitions.
  • Each strand has a *state*, which includes its *position*.
  • Some state variables are annotated as *outputs*.
  • Each strand has methods, including *update* and an optional *stabilize* method.
• Initialization, which defines the initial strands, and global coordination.
Simple volume rendering
Simple volume rendering

• Raycast through data volume (field).
Simple volume rendering

- Raycast through data volume (field).
- Use field values to determine opacity.
Simple volume rendering

• Raycast through data volume (field).
• Use field values to determine opacity.
• Use field gradient for lighting.
Simple volume rendering

• Raycast through data volume (field).
• Use field values to determine opacity.
• Use field gradient for lighting.
• Use opacity range to pick out different features (skin and bones).
field#2(3)[] F = bspln3

load(dataFile);

strand RayCast (int ui, int vi)
{
  ...
  vec3 rayVec = ⋅⋅⋅;
  real rayN = ⋅⋅⋅;
  real rayTransp = 1.0;
  vec3 rayRGB = [0.0, 0.0, 0.0];
  output vec4 outRGBA;
  ...
  // methods
  update {
    ...
  }
}
initially [
  RayCast(ui, vi) | vi in 0..wid-1, ui in 0..ht-1
];
Simple volume rendering

```c
update {
    vec3 rayPos = camEye + rayN*rayVec;
    if (inside (rayPos, F)) {
        real val = F(rayPos);
        if (val > valOpacMin) { // we have some opacity
            vec3 norm = normalize(-∇F(rayPos));
            real alpha = min(1.0, lerp(0.0, 1.0, valOpacMin, val, valOpacMax));
            real ld = max(0.0, norm • lightDir);
            vec3 matRGB = [1.0, 1.0, 1.0];
            vec3 pntRGB = (Ka*matRGB + Kd*ld*modulate(matRGB, lightRGB));
            rayRGB += rayTransp*alpha*pntRGB;
            rayTransp = rayTransp*(1.0 - alpha);
        }
        if (rayN > camVspFar) stabilize;
        rayN += rayStep;
    }
    stabilize { outRGBA = [rayRGB[0], rayRGB[1], rayRGB[2], 1.0-rayTransp]; }```
Computing with tensor fields
Computing with tensor fields

The most important computational abstraction in Diderot is the tensor field, which is a function from a vector space to tensors.
Computing with tensor fields

The most important computational abstraction in Diderot is the *tensor field*, which is a function from a vector space to tensors.

- Fields can be defined from images and manipulated using higher-order operators
Computing with tensor fields

The most important computational abstraction in Diderot is the *tensor field*, which is a function from a vector space to tensors.

- Fields can be defined from images and manipulated using higher-order operators
  - convolution: $V \ast h$
Computing with tensor fields

The most important computational abstraction in Diderot is the tensor field, which is a function from a vector space to tensors.

- Fields can be defined from images and manipulated using higher-order operators
  - convolution: $V \odot h$
  - field arithmetic: $s * F$, $F + G$, ...
Computing with tensor fields

The most important computational abstraction in Diderot is the tensor field, which is a function from a vector space to tensors.

- Fields can be defined from images and manipulated using higher-order operators
  - convolution: $V \odot h$
  - field arithmetic: $s \ast F, F + G, ...$
  - differentiation: $\nabla F, \nabla \times \nabla F$
Computing with tensor fields

The most important computational abstraction in Diderot is the tensor field, which is a function from a vector space to tensors.

- Fields can be defined from images and manipulated using higher-order operators
  - convolution: $V \odot h$
  - field arithmetic: $s \ast F$, $F + G$, ...
  - differentiation: $\nabla F$, $\nabla \times \nabla F$

- Fields can be probed at a location to get a tensor value: $F(x)$. 
Computing with tensor fields

The most important computational abstraction in Diderot is the tensor field, which is a function from a vector space to tensors.

- Fields can be defined from images and manipulated using higher-order operators
  - convolution: \( V \odot h \)
  - field arithmetic: \( s \star F, F + G, \ldots \)
  - differentiation: \( \nabla F, \nabla \times \nabla F \)

- Fields can be probed at a location to get a tensor value: \( F(x) \).
- Field types track continuity, domain, and range.
More examples
update {
...
vec3 grad = -∇F(pos);
vec3 norm = normalize(grad);
tensor[3,3] H = \nabla \otimes \nabla F(pos);
tensor[3,3] G = -(P \cdot H \cdot 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 RGB
    RGB([clamp(-1.0, 1.0, 6.0*k1),
        clamp(-1.0, 1.0, 6.0*k2)]);
...
}
Line Integral Convolution (LIC)

```cpp
field#1(2)[] V = load("vorttest.nrrd") \times ctmr;
field#0(2)[] R = load("vorttest-trnd.nrrd") \times tent;
strand LIC (vec2 pos0) {
    vec2 forw = pos0;
    vec2 back = pos0;
    output real sum = R(pos0);
    int step = 0;

    update {
        forw += h*V(forw + 0.5*h*V(forw));
        back -= h*V(back - 0.5*h*V(back));
        sum += R(forw) + R(back);
        step += 1;
        if (step == stepNum) {
            sum = \|V(pos0)\|*sum/real(1 + 2*stepNum);
            stabilize;
        }
    }
}
```
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)
      die;
    vec2 grad = \nF(pos);
    // delta = Newton-Raphson step
    vec2 delta = normalize(grad) * (F(pos) - isoval)/|grad|;
    if (|delta| < epsilon)
      stabilize;
    pos -= delta;
    steps += 1;
  }
}
Implementation
Probing tensor fields

A probe is compiled into code that maps the world-space coordinates to image space and then convolves the image values from the neighborhood of the position.
Probing tensor fields

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

Discrete image data $V$ \(\otimes h\) \(\rightarrow\) Continuous field $F$
Probing tensors

In 2D, the reconstruction is given by (recall 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

- $s$: support of $h$
- $n = \lfloor M^{-1}x \rfloor$: integer index
- $f = M^{-1}x - n$: fractional part
Probing tensors (general case)
Probing tensors (general case)

In general, generating code for probes is more challenging.
Probing tensors (general case)

In general, generating code for probes is more challenging. The first step is to normalize field operations.

\[ \nabla (s \ast (V \otimes h)) \Rightarrow s \ast (\nabla (V \otimes h)) \]

\[ \Rightarrow s \ast (V \otimes (\nabla h)) \]
Probing tensors (general case)

In general, generating code for probes is more challenging.
The first step is to normalize field operations.

\[ \nabla (s^* (V \otimes h)) \Rightarrow s^* (\nabla (V \otimes h)) \]
\[ \Rightarrow s^* (V \otimes (\nabla h)) \]

In the implementation, we view \( \nabla \) as a “tensor” of partial-derivative operators

\[ \nabla = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} \]
\[ \nabla \otimes \nabla = \begin{bmatrix} \frac{\partial^2}{\partial x^2} & \frac{\partial^2}{\partial x \partial y} \\ \frac{\partial^2}{\partial x \partial y} & \frac{\partial^2}{\partial y^2} \end{bmatrix} \]
Probing tensors (general case)

\[
(V \otimes \nabla h)(x) = V \otimes \left[ \begin{array}{c} \frac{\partial}{\partial x} h(x) \\ \frac{\partial}{\partial y} h(x) \end{array} \right]
\]

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

\[
\sum_{i=1}^{s} \sum_{j=1}^{s} V[n + \langle i, j \rangle] h(f_x - i) h'(f_y - j)
\]
Probing tensors (general case)

$$(V \circ \nabla h)(x) = V \circ \left[ \frac{\partial}{\partial x} h(x) \right]$$

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

A later stage of the compiler expands out the evaluations of the kernels.
Probing tensors (general case)

\[(V \otimes \nabla h)(x) = V \otimes \left[ \frac{\partial}{\partial x} h(x) \right. \right. \]
\[\left. \left. \frac{\partial}{\partial y} h(x) \right] \]
\[= \left[ \sum_{i=1}^{s} \sum_{j=1}^{s} V[n + \langle i, j \rangle] h'(f_x - i) h(f_y - j) \right. \right. \]
\[\left. \left. \sum_{i=1}^{s} \sum_{j=1}^{s} V[n + \langle i, j \rangle] h'(f_x - i) h(f_y - j) \right] \]

A later stage of the compiler expands out the evaluations of the kernels.
Probing code has **high arithmetic intensity** and is a good candidate for vectorization and GPUs.
Targeting GPUs
Targeting GPUs

GPU support is a major focus of the Diderot project.
Targeting GPUs

GPU support is a major focus of the Diderot project.

• Existing GPU programming models are low level and expose hardware details.
Targeting GPUs

GPU support is a major focus of the Diderot project.

• Existing GPU programming models are low level and expose hardware details.
• Diderot frees the programmer from these details, but the compiler must still handle them.
Targeting GPUs

GPU support is a major focus of the Diderot project.

• Existing GPU programming models are low level and expose hardware details.

• Diderot frees the programmer from these details, but the compiler must still handle them.

• Diderot’s parallelism model is GPU friendly.
Targeting GPUs

GPU support is a major focus of the Diderot project.

- Existing GPU programming models are low level and expose hardware details.
- Diderot frees the programmer from these details, but the compiler must still handle them.
- Diderot’s parallelism model is GPU friendly.
- Our initial GPU support is based on a direct mapping of strands to OpenCL work items.
Targeting GPUs

GPU support is a major focus of the Diderot project.

• Existing GPU programming models are low level and expose hardware details.
• Diderot frees the programmer from these details, but the compiler must still handle them.
• Diderot’s parallelism model is GPU friendly.
• Our initial GPU support is based on a direct mapping of strands to OpenCL work items.
• To support dynamic strand creation, we plan to switch to a persistent thread model.
Targeting GPUs

GPU support is a major focus of the Diderot project.

• Existing GPU programming models are low level and expose hardware details.
• Diderot frees the programmer from these details, but the compiler must still handle them.
• Diderot’s parallelism model is GPU friendly.
• Our initial GPU support is based on a direct mapping of strands to OpenCL work items.
• To support dynamic strand creation, we plan to switch to a persistent thread model.
• The compiler needs to be smart about GPU issues: memory organization and control-flow.
Conclusion
Status

• First version of language design is complete.
Status

- First version of language design is complete.
- Working compiler that produces C code with GCC vector extensions.
Status

• First version of language design is complete.
• Working compiler that produces C code with GCC vector extensions.
• Sequential performance is better than previous sequential implementations.
Status

- First version of language design is complete.
- Working compiler that produces C code with GCC vector extensions.
- Sequential performance is better than previous sequential implementations.
- Linear speedups on SMP systems.
Status

- First version of language design is complete.
- Working compiler that produces C code with GCC vector extensions.
- Sequential performance is better than previous sequential implementations.
- Linear speedups on SMP systems.
- OpenCL backend is almost working; CUDA backend is planned.
Status

• First version of language design is complete.
• Working compiler that produces C code with GCC vector extensions.
• Sequential performance is better than previous sequential implementations.
• Linear speedups on SMP systems.
• OpenCL backend is almost working; CUDA backend is planned.
• Some features under development:
Status

• First version of language design is complete.
• Working compiler that produces C code with GCC vector extensions.
• Sequential performance is better than previous sequential implementations.
• Linear speedups on SMP systems.
• OpenCL backend is almost working; CUDA backend is planned.
• Some features under development:
  • Automatic embedding in a python-based GUI.
Status

- First version of language design is complete.
- Working compiler that produces C code with GCC vector extensions.
- Sequential performance is better than previous sequential implementations.
- Linear speedups on SMP systems.
- OpenCL backend is almost working; CUDA backend is planned.
- Some features under development:
  - Automatic embedding in a python-based GUI.
  - Lifting more tensor operations to fields.
Status

• First version of language design is complete.
• Working compiler that produces C code with GCC vector extensions.
• Sequential performance is better than previous sequential implementations.
• Linear speedups on SMP systems.
• OpenCL backend is almost working; CUDA backend is planned.
• Some features under development:
  • Automatic embedding in a python-based GUI.
  • Lifting more tensor operations to fields.
  • Support for reusing common code (e.g., camera controls).
Conclusion

Parallel domain-specific languages, like Diderot, can provide several advantages:
Conclusion

Parallel domain-specific languages, like Diderot, can provide several advantages:

• Allow non-expert programmers to harness the power of current and future parallel hardware.
Parallel domain-specific languages, like Diderot, can provide several advantages:

• Allow non-expert programmers to harness the power of current and future parallel hardware.
• Allow for truly portable parallel codes.
Conclusion

Parallel domain-specific languages, like Diderot, can provide several advantages:

• Allow non-expert programmers to harness the power of current and future parallel hardware.
• Allow for truly portable parallel codes.
• Support for domain-specific notation and abstractions.
Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

The contents of this presentation were provided by individual(s) and/or company listed on the title page. The information and opinions presented in this presentation may not represent AMD’s positions, strategies or opinions. Unless explicitly stated, AMD is not responsible for the content herein and no endorsements are implied.