A different kind of functional language

John Reppy

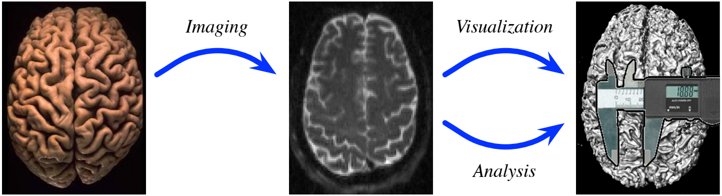
University of Chicago / NSF

November 2012

Parallel languages research

- Manticore: Parallel SML (PML)
- Nesl/GPU
- Diderot
  Joint work with Gordon Kindlmann, Charisee Chiw, Lamont Samuels, Nick Seltzer.

Why image analysis is important



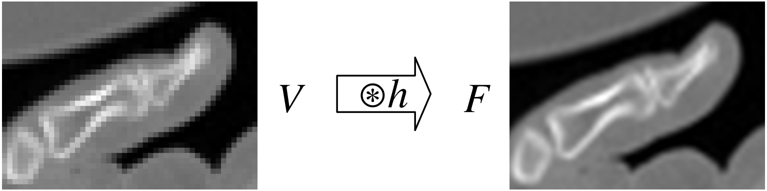
Physical object **Image data** **Computational representation**

- 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

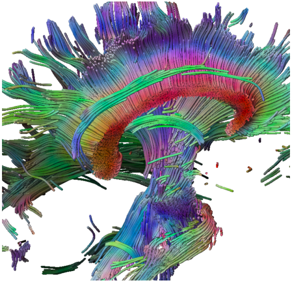
- Direct volume rendering (requires reconstruction, derivatives).
- Fiber tractography (requires tensor fields).
- Particle systems (requires dynamic numbers of computational elements).

These applications have a common algorithmic structure: large number of (mostly) independent computations.

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).

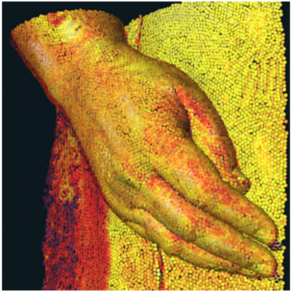


These applications have a common algorithmic structure: large number of (mostly) independent computations.

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).



These applications have a common algorithmic structure: large number of (mostly) independent computations.

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).

These applications have a common algorithmic structure: large number of (mostly) independent computations.

Diderot

Diderot is a parallel DSL for image analysis and visualization algorithms.

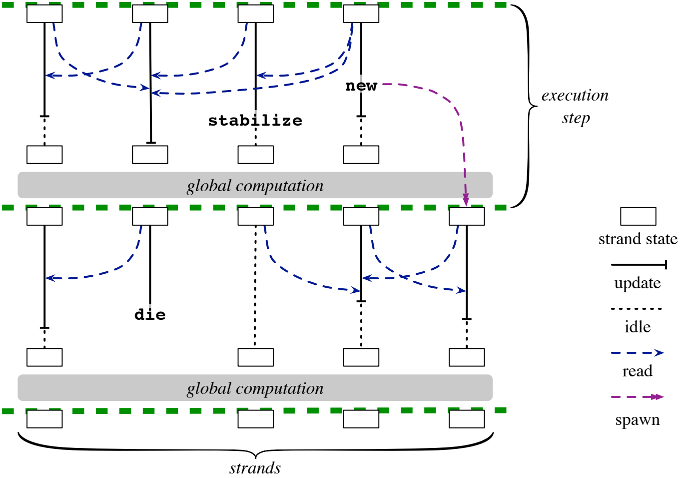
Its design models the algorithmic structure of its application domain: independent strands computing over continuous tensor fields.

A DSL approach provides

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

Diderot parallelism model

Bulk-synchronous parallel with “deterministic” semantics.



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

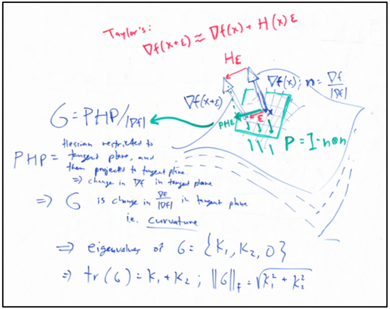
Square roots of integers using Heron's method.

```
// 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;
    }
}
```

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

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

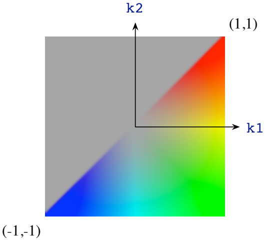
Programmability: from whiteboard to code

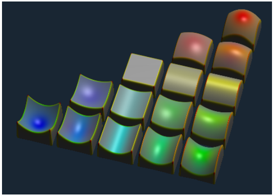



```
vec3 grad = -∇F(pos);
vec3 norm = normalize(grad);
tensor[3,3] H = ∇⊗∇F(pos);
tensor[3,3] P = identity[3] - norm⊗norm;
tensor[3,3] G = -(P•H•P)/|grad|;
real disc = sqrt(2.0*|G|^2 - trace(G)^2);
real k1 = (trace(G) + disc)/2.0;
real k2 = (trace(G) - disc)/2.0;
```

Example — Curvature

```
field#2(3)[] F = bspln3 ⊛ image("quad-patches.nrrd");
field#0(2)[3] RGB = tent ⊛ image("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;
  }
}
```

Fields

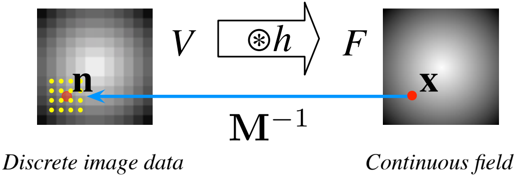
- Fields are functions from $\Re^d$ to tensors.

$$\texttt{field\#}\overset{\textit{levels of continuity}}{k}(\underset{\textit{dimension of domain}}{d})\underbrace{[d_1, \ldots, d_n]}_{\textit{shape of range}}$$

  where $k \geq 0$, $d > 0$, and the $d_i > 1$.
- Diderot provides higher-order operations on fields: ∇, $\nabla\otimes$, *etc.*.
- Diderot also lifts tensor operations to work on fields (*e.g.*, $+$).

Applying tensor fields

A field application $F(\mathbf{x})$ 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

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

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

Applying tensor fields *(continued ...)*

In general, compiling the field applications 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.

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

In the implementation, we view ∇ as a "tensor" of partial-derivative operators

$$
\nabla = \left[\begin{array}{c} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{array} \right] \qquad \nabla \otimes \nabla = \left[\begin{array}{cc} \frac{\partial^2}{\partial x^2} & \frac{\partial^2}{\partial xy} \\ \frac{\partial^2}{\partial xy} & \frac{\partial^2}{\partial y^2} \end{array} \right]
$$

Applying tensor fields *(continued ...)*

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

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

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.

Normalization

- The current compiler uses “direct-style” notation when normalizing tensor and field expressions.
- This approach does not extend to some interesting operations, such as $\nabla\times$.
- Expanding tensor operations to their scalar subcomputations is unwieldy.
- Einstein Index Notation (EIN) provides a compact representation of tensor expressions.
- New IR operator,

$$\lambda\bar{T}.\langle e\rangle_{\alpha}$$

  whose semantics are specified by the EIN expression e, where $\bar{T}$ are tensor parameters and α is a multi-index that determines the shape of the result.

Einstein Index Notation *(continued ...)*

- Concise specification of families of operators. For example, $\lambda(u,v).\langle u_{\alpha i} v_{i\beta}\rangle_{\alpha\beta}$ covers dot product, matrix-vector multiplication, matrix-matrix multiplication, etc.
- Code and data-representation synthesis (need cache-friendly and SSE-friendly mappings).
- Automatic discovery of linear-algebra identities.

Optimizing tensor operations

Consider the expression `trace(a⊗b)`.

This Diderot expression is represented in the compiler as

$$\textbf{let } M = (\lambda(u,v).\langle u_i v_j \rangle_{ij})(a,b)$$
$$\textbf{let } t = (\lambda X.\langle X_{kk} \rangle)(M)$$
$$\textbf{in } t$$

substitution of the definition of M for X yields

$$\textbf{let } t = (\lambda(u,v).\langle u_k v_k \rangle)(a,b)$$
$$\textbf{in } t$$

Replaces a rewrite rule: $\mathrm{Trace}(\mathrm{Outer}(u,v)) \Rightarrow \mathrm{Dot}(u,v)$.

Optimizing tensor operations

Consider the expression `trace(a⊗b)`.

This Diderot expression is represented in the compiler as

$$\textbf{let } M = (\lambda(u,v).\langle u_i v_j \rangle_{ij})(a,b)$$
$$\textbf{let } t = (\lambda X.\langle X_{kk} \rangle)(M)$$
$$\textbf{in } t$$

substitution of the definition of M for X yields

$$\textbf{let } t = (\lambda(u,v).\langle u_k v_k \rangle)(a,b)$$
$$\textbf{in } t$$

Replaces a rewrite rule: $\mathrm{Trace}(\mathrm{Outer}(u,v)) \Rightarrow \mathrm{Dot}(u,v)$.

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.

Questions?

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

Thanks to NVIDIA and AMD for their support.