PREPRINT: To be presented at PLDI '12, June 11–16, Beijing, China

Diderot: A Parallel DSL for Image Analysis and Visualization

Charisee Chiw   Gordon Kindlmann   John Reppy   Lamont Samuels   Nick Seltzer
University of Chicago
{cchiw,glk,jhr,lamonts,nseltzer}@cs.uchicago.edu

Abstract

Research scientists and medical professionals use imaging technology, such as *computed tomography* (CT) and *magnetic resonance imaging* (MRI) to measure a wide variety of biological and physical objects. The increasing sophistication of imaging technology creates demand for equally sophisticated computational techniques to analyze and visualize the image data. Analysis and visualization codes are often crafted for a specific experiment or set of images, thus imaging scientists need support for quickly developing codes that are reliable, robust, and efficient.

In this paper, we present the design and implementation of Diderot, which is a parallel domain-specific language for biomedical image analysis and visualization. Diderot supports a high-level model of computation that is based on continuous tensor fields. These tensor fields are reconstructed from discrete image data using separable convolution kernels, but may also be defined by applying higher-order operations, such as differentiation (∇). Early experiments demonstrate that Diderot provides both a high-level concise notation for image analysis and visualization algorithms, as well as high sequential and parallel performance.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language classifications — Very high-level languages; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Performance

Keywords Domain Specific Languages, Image Analysis, Scientific Visualization, Parallelism

1. Introduction

Biomedical researchers use multi-dimensional imaging to study structure and function of biological systems with a variety of imaging modalities, including *confocal microscopy*, *computed tomography* (CT), and *magnetic resonance imaging* (MRI). Recent advances in imaging hardware have increased their speed, resolution, and flexibility. For example, synchrotron radiation microCT produces scalar-valued volume images with 2048^3 samples (as compared to 512^3 for clinical CT), and routine MRI scans include images of vectors (for blood flow in heart) and tensors (for diffusion anisostropy in brain tissue). Besides increasing the biomedical value of imaging, the quantities and types of data from modern imaging hardware also strain the capabilities of established tools for image processing. Scientists working with the new imaging techniques and applications are the best positioned to create and evaluate methods of image data processing, but they do not have the programming background to create the efficient parallel implementations required to handle the quantity of data involved.

In this paper, we present a domain-specific language (DSL), called Diderot, that simplifies the portable implementation of parallel methods of biomedical image analysis and visualization. *Image analysis* extracts quantitative or geometric descriptions of the image structure in order to characterize specific properties of the underlying organ or tissue. An example is extracting ridge lines and valley lines to find blood vessels and airways, respectively, from a CT lung scan. *Visualization* combines measurements of local image data properties with elements of computer graphics in order to qualitatively depict structures via rendered images. An example is using direct volume rendering to illustrate the over-all shape and inter-relationship of tissue boundaries.

Diderot supports a high-level model of computation that is based on continuous tensor fields. Throughout, we use *tensors* to refer collectively to scalars, vectors, and matrices, which encompasses the types of values produced by the imaging modalities mentioned above, as well as values produced by taking spatial derivatives of images. Diderot permits programmers to express algorithms directly in terms of tensors, tensor fields, and tensor field operations, using the same mathematical notation that would be used in vector and tensor calculus (such as ∇ for the gradient). We intend for Diderot to be useful for prototyping image analysis and visualization methods in contexts where a meaningful evaluation of the methods requires its application to real image data, but the real data volumes are of a size that requires efficient parallel computation. With its support for high-level mathematical notation, Diderot will also be useful in educational contexts where the conceptual transparency of the implementation is of primary importance.

This paper makes several contributions. First, we introduce a novel DSL that directly supports higher-order operations on continuous tensor fields. Diderot's type system tracks properties, such as continuity, which helps ensure sensible numerical results. We also describe how principles of tensor calculus may be applied to address the challenges of compiling a very-high-level language like Diderot. Lastly, we demonstrate the benefits of Diderot relative to hand-coded algorithms for parallel computing performance.

The paper is organized as follows. In the next section, we review the mathematical foundations of image processing on which Diderot is based. We then present Diderot's design in Section 3 and illustrate its features with several examples in Section 4. We then describe important aspects of our implementation, including the implementation of tensor fields. Section 6 presents performance results, followed by a discussion of related work. We describe future plans for the system in Section 8 and then summarize our results in Section 9.

2. Background

A review of background mathematics will simplify later description of the mathematical elements supported by Diderot, and will provide the context for our description of the domain of computation for which Diderot is specialized.

We adopt tensors as a concrete type in Diderot to provide a general way of representing the values stored in images and produced by operations on images. 0-order tensors, or scalars, capture real-valued samples from scans typically shown in grayscale (*e.g.*, CT). 1-order tensors, or vectors, describe directional quantities such as velocity and spatial derivatives of scalar fields. 2-order tensors, represented as matrices, describe linear transforms on vectors, first derivatives of vector fields, and second derivatives of scalar fields.

Measured image data is discretely sampled on a regular grid (regardless of the modality or image type), but the underlying objects being scanned exist in a continuous space, which we call *world space*. Signal processing provides the machinery for reconstructing (or *probing*) continuous signals from discrete data via convolution at arbitrary positions [20]. Convolving one-dimensional discrete data $V[i]$ with a continuous reconstruction kernel $h(x)$ produces a continuous signal $(V \circledast h)(x)$. *Separable* convolution uses a single one-dimensional kernel $h(x)$ to create a multi-dimensional kernel, for example in three dimensions $H(x,y,z) = h(x)h(y)h(z)$. Convolution also provides the means of measuring derivatives in fields. The partial derivatives of $F = V \circledast H$ are found by convolution with separable product of kernels and their derivatives, *e.g.*, $\frac{\partial F}{\partial y}$ is found by convolution with $\frac{\partial H}{\partial y} = h(x)h'(y)h(z)$. Kernels can be chosen according to the needed level of differentiability, while balancing the quality of reconstruction with the computational cost of having larger support (*i.e.* needing a larger neighborhood of image values to compute the convolution sum).

Diderot is specialized for highly parallel analysis and visualization of continuously differentiable tensor fields arising from multidimensional imaging. The continuity of fields is important because relevant features in the image data tend to lie in between pixels and at arbitrary orientations. Simplifying how programmers can express computations within world space helps prevent the results from needlessly reflecting the discrete image grid. The differentiability of fields is required because many structures of interest are defined in terms of spatial derivatives. Extracting the network of blood vessels from a CT scan is a good example of the need for continuity and differentiability. Accurate results depend on tracing the centers of vessel pathways in between pixel locations, where gradients and Hessians (first and second derivatives) are computed to locate the *ridge line* image features that coincide with the vessels [2, 11]. Tensors may also originate in the image data, as with second-order diffusion tensors measured via MRI, for which a common task is tracing paths along the tensor principal eigenvector to model brain connectivity [4]. Diderot simplifies the rapid exploration and implementation of these types of algorithms by naturally supporting the mathematical ingredients with which they are expressed: vectors, tensors, eigensystems, kernels, fields, and derivatives.

We have chosen to develop a new domain-specific language because of weaknesses with existing languages, libraries, and tools. Established high-level array-processing languages commonly used for image processing, including Matlab [18], IDL [15] and Python/Numpy [25], facilitate operations on entire image arrays or their sub-arrays. The computational cost is distributed uniformly over the image, and parallelization according to memory layout is straightforward. An early example of this approach is the *image algebra* formulation of gray-scale image-to-image computation developed by Ritter and Gader [24]. For our intended applications, however, it is not convenient to express the computation in terms of discrete sets of image data values. The transparent implementation of the blood vessel extraction and fiber tractography mentioned above, for example, hinges on a notion of continuous fields, a fundamental abstraction available in Diderot. Also, the parallelism we intend to capture is not structured according to the underlying image data array, but by the output of the visualization or analysis, *i.e.*, rendered pixels in volume rendering, tractography pathways curving through space, and points along ridge feature lines. Third, array-processing languages are often optimized for uniform access rather than irregular or sparse access patterns, so techniques that work adequately for two-dimensional images do not scale well to memory-intensive three-dimensional images, especially when the algorithmic output (as with blood vessel extraction) occupies only a small fraction of the volume data.

Another strategy for creating image analysis and visualization tools is to use a specialized library that already encapsulates many of the useful operations and constructs. A good example is the Insight Toolkit (ITK) [14], a large C++ image processing framework. The drawbacks here, relative to a domain-specific language, relate to program conciseness and simplicity. ITK relies heavily on C++ templates to achieve generality with respect to image type and dimension, so the code can be challenging to understand and debug. In ITK, as with array-processing languages, it is easiest to parallelize algorithms that compute over whole image arrays, since this is the structure of the image registration and segmentation routines for which ITK was originally designed.

3. Language design

The class of applications that Diderot targets are characterized as consisting of many largely independent subcomputations. For example, the rays in a volume renderer, the paths from fiber tractography, and the particles in a particle system. In Diderot, these mostly independent computations are modeled as *strands*, which execute in a *bulk synchronous* fashion [26, 31]. Diderot uses a C-like syntax for its computational notation, but this notation is extended with mathematical types and operators.

3.1 Types and values

Diderot is a monomorphic, statically-typed, language with a fairly small collection of types. The basic concrete values include strings, booleans, integers, and tensors. We have five types of concrete values: booleans, integers, strings, tensors, and fixed-size sequences of values. The type **`tensor`**`[`σ`]` is a tensor with *shape* $\sigma \in \{i \mid 1 < i\}^*$. The number of dimensions in σ (*i.e.*, the length of σ) is the *order* of the tensor. For example, **`tensor`**`[]` is a 0-order tensor and is the type of scalars, while **`tensor`**`[3]` is a 1-order tensor that is the type of 3D vectors. We define type synonyms for common tensor types, including **`real`** and **`vec3`**.

In addition to concrete values, Diderot defines three forms of abstract type: images, kernels, and fields. Images are multidimensional arrays of tensor data. The type **`image`**`(`d`)[`σ`]` is the abstract type of image data, where $d \in \{1,2,3\}$ specifies the dimension of the data and σ specifies the shape of the tensor values at each image sample. We do not program directly with image data. Instead we use convolution kernels to define a continuous reconstruction of the discrete data. The type **`kernel#`**k is the type of a C^k kernel (*i.e.*, it has k continuous derivatives). Diderot provides a number of useful built-in kernels, including the C^0 `tent` for linear interpolation (so named because of its shape), the C^1 interpolating Catmull-Rom cubic spline `ctmr`, and the C^2 (non-interpolating) uniform cubic B-spline basis function `bspln3` [3]. Finally, continuous tensor fields have type **`field#`**k`(`d`)[`σ`]`, where k is the number of continuous derivatives, d is the dimension of the field's domain, and σ is the shape of its range. A field is a function from d-dimensional space to tensors with the shape σ.

```
1  input real stepSz = 0.1;  // size of steps
2  input vec3 eye;           // eye location
3  input vec3 orig;          // pixel (0,0) location
4  input vec3 cVec;          // vector between columns
5  input vec3 rVec;          // vector between rows
6  input real opacMin;       // value with opacity 0.0
7  input real opacMax;       // value with opacity 1.0
8  image(3)[] img = load ("hand.nrrd");
9  field#2(3)[] F = img ⊛ bspln3;
10
11 strand RayCast (int r, int c) {
12   vec3 pos = orig + real(r)*rVec + real(c)*cVec;
13   vec3 dir = normalize(pos - eye);
14   real t = 0.0;
15   real transp = 1.0;
16   output real gray = 0.0;
17
18   update {
19     pos = pos + stepSz*dir;
20     t = t + stepSz;
21     if (inside (pos, F)) {
22       real val = F(pos);
23       if (val > opacMin) {
24         real opac = 1.0 if (val > opacMax)
25                    else (val - opacMin)/(opacMax - opacMin);
26         vec3 norm = -normalize(∇F(pos));
27         gray += transp*opac*max(0.0, -dir • norm);
28         transp *= 1.0 - opac;
29       }
30     }
31     if (t > 40.0) stabilize;
32   }
33 }
34
35 initially [
36   RayCast(ui, vi) | vi in 0 .. imgResV-1,
37                     ui in 0 .. imgResU-1
38 ];
```

Figure 1. Simple direct volume rendering code

3.2 Tensor and field expressions

A key feature of Diderot is that it supports mathematical notation for computing with tensors and fields. Diderot syntax uses Unicode characters to represent mathematical constants (π) and a rich set of operations on tensors. In addition to standard arithmetic operations, these include dot product (`u•v`), cross product (`u×v`), tensor product (`u⊗v`), and vector norm (`|u|`).

Computing with continuous tensor fields is one of the unique characteristics of Diderot. Field values are constructed by convolving image data with kernels (`img⊛bspln3`), but they can also be defined by using higher-order operations, such as addition, subtraction, and scaling of fields. Most importantly, Diderot supports differentiation of fields using the operators ∇ (for scalar fields) and ∇⊗ (for higher-order tensor fields). Two operations on fields are testing whether a point `x` lies within the domain of a field `F` (`inside(x, F)`) and *probing* a field `F` at a point `x` (`F(x)`). As we show in the examples below, probing and differentiating are fundamental to extracting geometric information from fields.

3.3 Program structure

A Diderot program is organized into three sections: global definitions, which include program inputs; strand definitions, which define the computational core of the algorithm; and initialization, which defines the initial set of strands. To illustrate this structure and the features of Diderot, we use the simple direct volume renderer shown in Figure 1 as a running example. This computation requires probing both the scalar field to determine its opacity and the its gradient to determine the surface normal.

3.3.1 Global definitions

Lines 1–9 of Figure 1 define the global variables of our example. Global variables in Diderot are immutable. The first six of these are marked as **`input`** variables, which means that they can be set outside the program (they may also have a default value, as in the case of `stepSz`). The Diderot compiler synthesizes glue code that allows command-line setting of input variables. Line 8 loads image file `"hand.nrrd"` and binds variable `img` to it. The type of `img` is a 3D scalar image, which is checked when the image is loaded. The `load` function may only be used in global part of the program. Note that we do not specify the representation of the image values on disk; *i.e.*, they could be signed ints, floats, *etc.* The compiler generates code that maps image values to reals. Line 9 defines a scalar field `F`, reconstructed by convolution with the `bspln3` kernel, providing the C^2 continuity reflected in the type of `F`.

3.3.2 Strands

Much like a kernel function in CUDA [22] or OpenCL [16], a strand in Diderot encapsulates the computational core of the application. Each strand has parameters (*e.g.*, `r` and `c` on line 11), a *state* (lines 12–13) and an **`update`** method (lines 18–33). The strand state variables are initialized when the strand is created; some variables may be annotated as **`output`** variables, which define the part of the strand state that is reported in the program's output. Unlike globals, strand state variables are mutable, but we avoid features, such as pointers, that would make analysis difficult. In addition, strand methods may define local variables (the scoping rules are essentially the same as C's).

The **`update`** method of the `RayCast` strand starts by advancing the strand's position along a ray (lines 20 and 21). It then tests to see if the position lies within the field `F` domain (line 22). If the strand's position is inside the domain, we *probe* the field `F` and compare the field's value to our lower opacity threshold. By changing the opacity range, we can pick out different features of the image (*e.g.*, skin or bone). In lines 27–29, we find the contribution of the current position to the image using the gradient field to compute diffuse lighting. Note that we use Python's syntax for conditional expressions (lines 25 and 26). In line 32, we check to see if the ray has gone beyond a distance limit, in which case we *stabilize* the strand, which means that it ceases to be updated. As we will see in later examples, a strand may also have a **`stabilize`** method that is invoked when the strand stabilizes.

3.3.3 Initialization

The last part of a Diderot program is the initialization section, which specifies the initial set of strands in the computation.[1] Diderot uses a comprehension syntax, similar those of Haskell or Python, to define the initial set of strands. For example, the volume renderer specifies a grid of initial ray positions in lines 35–38. When the strands are initialized as a grid, it implies that the strands will all stabilize (*i.e.*, they do not die). The grid structure is then preserved in the output. For example, the grid of initial strands created above will produce a grid of pixel values, one per ray.

Diderot also allows one to specify an initial *collection* of strands by using "{ }" as the brackets around the comprehension (instead of "`[ ]`"). In this case, the program's output will be a one-dimension array of values; one for each stable strand.

3.4 Diderot's type system

Diderot has a monomorphic type system that captures the important mathematical properties of the program, such as the continuity of

[1] In the current version of the language, described in this paper, strands are only created at startup. Eventually, we plan to support dynamic strand creation (see Section 8).

$$\frac{\Gamma \vdash V : \texttt{image}(d)[\sigma] \qquad \Gamma \vdash h : \texttt{kernel\#}k}{\Gamma \vdash V \circledast h : \texttt{field\#}k(d)[\sigma]}$$

$$\frac{\Gamma \vdash F : \texttt{field\#}k(d)[\,] \qquad k > 0 \qquad k' = k - 1}{\Gamma \vdash \nabla F : \texttt{field\#}k'(d)[d]}$$

$$\frac{\Gamma \vdash F : \texttt{field\#}k(d)[\sigma,\, d'] \qquad k > 0 \qquad k' = k - 1}{\Gamma \vdash \nabla\!\otimes F : \texttt{field\#}k'(d)[\sigma,\, d',\, d]}$$

$$\frac{\Gamma \vdash F : \texttt{field\#}k(d)[\sigma] \qquad \Gamma \vdash p : \texttt{tensor}[d]}{\Gamma \vdash F(p) : \texttt{tensor}[\sigma]}$$

Figure 2. Key typing judgments for Diderot

fields. Figure 2 presents the most important rules in the type system. The first rule shows how the properties of the image data and convolution kernel determine the type of resulting field. The next two rules capture the fact that differentiation reduces the continuity of the field, but increases its order. Lastly, the typing rule for probe is similar to function application (as one would expect).

4. Examples

Diderot's language design and computational model support a range of visualization and analysis methods. In this section, we illustrate the language's expressiveness with several examples of computations typical of visualization and analysis algorithms.

4.1 Implicit Surface Curvature

One physical property of interest is the *curvature* of implicit surfaces within the image. Just as the gradient of a field is used to find the implicit surface normal, the second derivative of the field, the Hessian, determines curvature (*i.e.*, the change in normal due to motion along the surface). The principal curvatures κ_1 and κ_2 can be computed from the tensor $\mathbf{G}$ defined by [17]

$$\begin{aligned} \mathbf{G} &= \frac{-\mathbf{PHP}}{|\nabla F|} \\ \text{where} & \\ \mathbf{P} &= \mathbf{I} - \mathbf{n} \otimes \mathbf{n}^T \\ \mathbf{n} &= \frac{\nabla F}{|\nabla F|} \\ \mathbf{H} &= \nabla \otimes \nabla F \end{aligned}$$

The eigenvalues of $\mathbf{G}$ are 0, κ_1, and κ_2, so the principal curvatures appear in the tensor invariants of $\mathbf{G}$

$$\begin{aligned} \text{trace}(\mathbf{G}) &= \kappa_1 + \kappa_2 \\ |\mathbf{G}| &= \sqrt{\kappa_1^2 + \kappa_2^2} \end{aligned}$$

where $|\mathbf{G}|$ is the Frobenius norm of $\mathbf{G}$. With some algebra, we get

$$\begin{aligned} \kappa_1 &= \frac{\text{trace}(\mathbf{G}) + d}{2} \\ \kappa_2 &= \frac{\text{trace}(\mathbf{G}) - d}{2} \\ \text{where} & \\ d &= \sqrt{2|\mathbf{G}|^2 - \text{trace}(\mathbf{G})^2} \end{aligned}$$

To visualize surface shape in a volume rendering, we can use a bivariate function of κ_1 and κ_2 to assign color based on local curvature. Diderot code that implements this is shown in Figure 3. Notice that the mathematical specification given above translates directly into Diderot; one can easily see that the code is an implementation of the method that one might derive on the whiteboard. This example also illustrates the use of a field to implement a color assignment function (*i.e.*, the `RGB` field). We sample this field using bilinear interpolation, which is provided by the `tent` kernel. The resulting image from applying this technique to a synthetic data set is shown in Figure 4 (this figure also includes an image of the bivariate colormap function).

```
// RGB colormap of (kappa1,kapp2)
field#0(2)[3] RGB = tent ⊛ load(xfer);
...
  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;
    // find material RGBA
    vec3 matRGB =
        RGB([max(-1.0, min(1.0, 6.0*k1)),
             max(-1.0, min(1.0, 6.0*k2))]);
    ...
  }
```

Figure 3. Computing the surface color based on implicit surface curvatures

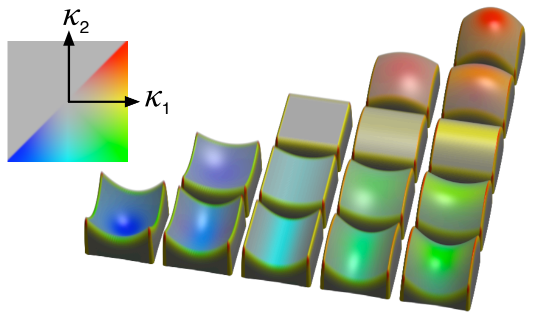

Figure 4. Volume rendering with color determined by implicit surface curvatures (κ_1, κ_2)

4.2 Line Integral Convolution (LIC)

Line integral convolution (LIC) is a vector field visualization method that can be concisely expressed in Diderot. LIC visualizes a vector field by blurring an underlying noise texture (a scalar field) along vector field streamlines [7]. Streamlines are paths everywhere tangent to the vector field, computed by numerical integration; in our benchmark we use the midpoint method (a second-order Runge-Kutta method). For each pixel in the output, we define a strand that computes a streamline. The pixel value is the average of noise texture samples taken along all computed vertices of the streamline seeded at that pixel. The Diderot implementation in Figure 5 simultaneously computes downstream (`forw`) and upstream (`back`) segments of the streamlines of the `vectors` synthetic vector field, sampling the `rand` scalar field at each step, and stopping each after `stepNum` steps. The output graylevel contrast is modulated by the the vector field velocity at the seedpoint `|V(pos0)|`, creating the LIC result shown in Figure 6.

4.3 Particle-based feature sampling

One class of applications that Diderot targets is the use of *particles* to detect and sample image features, such as isocontours [21]. As a simple example, we consider detecting isocontours in a 2D grayscale image. Figure 7 gives the strand definition for this pro-

```
field#1(2)[2] V = load("vectors.nrrd") ⊛ ctmr;
field#0(2)[] R = load("rand.nrrd") ⊛ 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)| / real(1 + 2*stepNum);
      stabilize;
    }
  }
}
```

Figure 5. Line Integral Convolution (LIC)

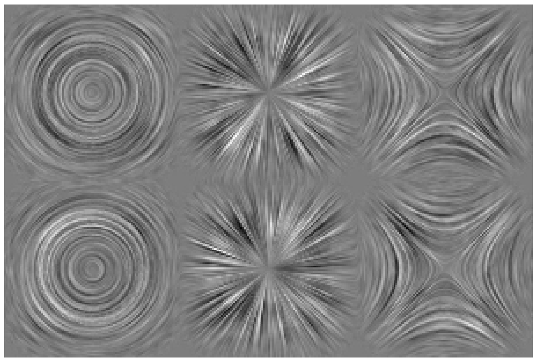

Figure 6. Line Integral Convolution (LIC) on synthetic data

gram. This program defines an initial collection of strands that are positioned in a 2D grid pattern. The image value at a strand's initial position determines the isovalue f_0 that it will search for. The strand's update method uses Newton-Raphson iteration to find the root of $F(\mathbf{x}) = f(\mathbf{x}) - f_0$ by motion along the normalized gradient $\nabla F / |\nabla F|$. The strand's search can terminate in one of two ways: if the length of the position update `|delta|` falls below `epsilon`, then the strand stabilizes — its position will be the output. Otherwise, if the strand wanders outside the field domain or takes too many steps, then the strand dies and produces no output. Since some strands die during execution, the final collection of stable strands will be a subset of the initial collection. Figure 8 shows a visualization of running this algorithm on a grayscale version of a portrait of Denis Diderot by Louis-Michel van Loo. The final positions of the stable strands are rendered as green dots. Note that the image is depicted with nearest-neighbor interpolation to show its individual pixels as squares, while the continuous interpolation (afforded by convolution with the Catmull-Rom cubic spline `ctmr`) creates isocontours that smoothly trace between pixels.

5. Implementation

Compiling a very-high-level language like Diderot requires a mix of traditional compiler techniques with domain-specific transformations and optimizations. In this section, we give an overview of our language implementation and describe the techniques that we

```
field#1(2)[] f = ctmr ⊛ load("ddro.nrrd");

strand sample (int ui, int vi) {
  output vec2 pos = ···;
  // set isovalue to closest of 50, 30, or 10
  real f0 = 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 = ∇f(pos);
    vec2 delta =  // the Newton-Raphson step
      normalize(grad) * (f(pos) - f0)/|grad|;
    if (|delta| < epsilon)
      stabilize;
    pos -= delta;
    steps += 1;
  }
}
```

Figure 7. Detecting isocontours

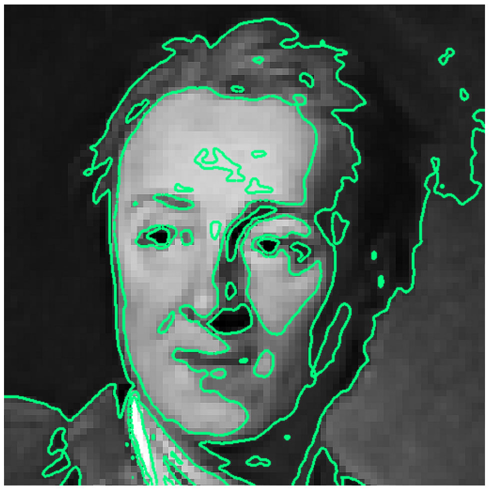

Figure 8. Isocontour detection in a grayscale image

have developed for compiling the higher-order field operations. We also provide a brief description of our runtime system.

5.1 Compiler overview

The Diderot compiler comprises roughly 19,000 lines of Standard ML code, which is organized into three main phases: the front-end, optimization and lowering, and code generation.

The front-end consists of parsing, type checking, and simplification. Although Diderot is a monomorphic language, most of its operators have instances at multiple types. For example, addition works on integers, tensors of all shapes, and fields. Since having to type check each operator as a special case would be prohibitively complicated, we use a mix of *ad hoc* overloading and polymorphism in the type checker. The internal representation of types includes kinded type variables, shape variables, and dimension variables. The type checking process introduces constraints between the variables, which are solved by unification. Once a program is type checked, the operator instances are instantiated at specific monotypes. The typed AST is then converted into a simplified representation, where temporaries are introduced for intermediate values and operator are applied only to variables. At this point we also duplicate code, as necessary, to ensure that fields are statically determined. For example, if the source program contained the line

```
real y = (F1 if b else F2)(x);
```

the simplified representation is transformed to code that is equivalent to

```
real y = F1(x) if b else F2(x);
```

This transformation is necessary because of the way we compile probe operations. While it could result in exponential code growth, we believe that in practice code growth will not be significant.

The optimization and lowering occurs over a series of three intermediate representations (IRs) based on Static Single Assignment (SSA) form [9]. These IRs share a common control-flow graph representation, but differ in their types and operations.

HighIR is essentially a desugared version of the source language with source-level types and operations.

MidIR supports vectors, transforms between coordinate spaces, loading image data, and kernel evaluations. At this stage, fields and probes have been compiled away into lower-level code.

LowIR supports basic operations on vectors, scalars, and memory objects.

The translations between these representations replaces higher-level operations with their equivalent lower-level operations. For example, field probes in the HighIR are expanded into the convolution of image samples and kernel evaluations in the MidIR. We discuss this particular expansion in more detail below.

The final phase is code generation. We have separate backends for different targets: sequential C code with vector extensions [12], parallel C code, OpenCL [16], and CUDA [22] (planned). Because these targets are all block-structured languages, our first step in code generation is to convert the LowIR SSA representation into a block-structured AST. The target-specific backends translate this representation into the appropriate representation and augment the code with type definitions and runtime support. The output is then passed to the host system's compiler.

5.2 Implementing field operations

Diderot's fields are abstract values that represent continuous functions. As such, we use a symbolic representation of field values in the compiler that is used to generate code for the `inside` test and for probing a field. In the simplest case, code that probes a tensor field is translated into code that maps the world-space position to image space and then convolves the image values from the neighborhood of the position using a kernel. This translation occurs when the program is converted from HighIR to MidIR; we then expand kernel evaluations into vectorized arithmetic when we convert from MidIR to LowIR.

While the case of probing a field that is defined directly by convolving an image with a kernel is straightforward, in general the problem is more complicated, since fields can be defined by higher-order operations such as scaling, addition, and differentiation. Thus, before we can translate HighIR to MidIR, we must normalize the field computations. Effectively, this process lowers higher-order operations that work on fields to operations that work on tensors. We call this process *field normalization*.

We can formalize field normalization as a translation from a language of field expressions to a normalized language of field probes and tensor operations. Figure 9 gives the grammars of these two languages; we have simplified the presentation by omitting many operators available in Diderot. In the source language (Figure 9a), we have tensor-valued expressions (denoted by e) and field-valued

$$\begin{array}{rcl}
(f_1 + f_2)(x) & \Rightarrow & f_1(x) + f_2(x) \\
(e * f)(x) & \Rightarrow & e * f(x) \\
\nabla(f_1 + f_2) & \Rightarrow & \nabla f_1 + \nabla f_2 \\
\nabla(e * f) & \Rightarrow & e * \nabla f \\
\nabla(V \circledast h) & \Rightarrow & V \circledast \nabla h \\
\nabla(V \circledast \nabla^i h) & \Rightarrow & V \circledast \nabla^{i+1} h
\end{array}$$

Figure 10. Rewriting rules for field normalization

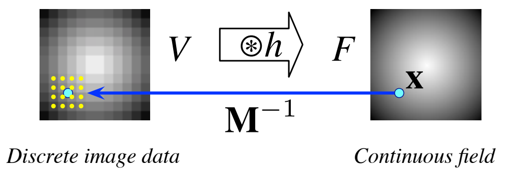

Figure 11. Probing a 2D field $F = V \circledast h$ at $\mathbf{x}$, where $\mathbf{M}$ is the mapping from world coordinates to image-space coordinates

expressions (denoted by f). Using the rewrite rules given in Figure 10, we can transform source-language expressions into the normalized language of Figure 9b. The normalized language enforces three key invariants on the structure of the computation:

1. All differentiation operations have been pushed down to the kernels in convolutions. We add a superscript to the ∇ operator to specify the level of differentiation. ∇^2 should not be confused with the standard notation for the Laplacian.
2. The fields involved in probe operations are defined directly as convolutions.
3. Arithmetic operations on fields have been lowered to operations on tensors.

These properties are key to being able to synthesize code for the probe operations, as is described in the next section.

5.3 Synthesizing probe operations

Once we have normalized the field operations as described above, we must still synthesize code to implement field probes. These operations get 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, as is illustrated by Figure 11. As discussed in Section 2, we use separable kernels to reconstruct the value of a field at a specific point. For example, if $F = V \circledast h$ is a 2-dimensional scalar field, then probing F at the location $\mathbf{x}$ is defined by the equation

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

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

One of the main challenges of synthesizing code for probe operations is that the shape of the result and the nesting of summations can be arbitrarily complicated. For example, assume that F is a 2D scalar field of type **field#**$k(2)[\,]$ and G is a 2D vector field of type **field#**$k(2)[2]$. Then the probe expressions $\nabla \otimes G(\mathbf{x})$ and $\nabla \otimes \nabla F(\mathbf{x})$ both have type **tensor**$[2, 2]$, but the code required to implement the two probes is very different. Tensor calculus provides the notational tools to manage this process, in that the differentiation operator (∇) can be treated as a vector of partial-

$$\begin{array}{rcll}
e & ::= & f(x) & \text{field probe} \\
 & | & \cdots & \\
f & ::= & V \circledast h & \text{convolution} \\
 & | & \nabla f & \text{differentiation} \\
 & | & f + f & \text{field addition} \\
 & | & e * f & \text{field scaling}
\end{array}$$

(a) source language

$$\begin{array}{rcll}
\hat{e} & ::= & \hat{f}(x) & \text{field probe} \\
 & | & \hat{e} + \hat{e} & \text{tensor addition} \\
 & | & \hat{e} * \hat{e} & \text{tensor scaling} \\
 & | & \cdots & \\
\hat{f} & ::= & V \circledast \hat{h} & \text{convolution} \\
\hat{h} & ::= & \nabla^i h & \text{kernel with } i \geq 0 \\
 & & & \text{levels of differentiation}
\end{array}$$

(b) normalized language

Figure 9. The source and target languages of field normalization, where V denotes images and h denotes kernels

differentiation operators. For example, in 2D we have

$$\nabla = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix}$$

We can use this representation to handle the probing of the gradient of a scalar field F

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

Note that the partial-differentiation operators tell us where to use h and where to use the first derivative h' in the reconstruction.

This approach generalizes to multiple levels of differentiation in a natural way. For example, the Hessian of F is defined as $\nabla \otimes \nabla F$, which is normalized to $V \circledast \nabla^2 h$, which can be written as $V \circledast \nabla \otimes \nabla h$. Thus we have

$$\nabla \otimes \nabla h = \begin{bmatrix} \frac{\partial^2}{\partial x^2} & \frac{\partial^2}{\partial xy} \\ \frac{\partial^2}{\partial xy} & \frac{\partial^2}{\partial y^2} \end{bmatrix} h = \begin{bmatrix} \frac{\partial^2}{\partial x^2} h & \frac{\partial^2}{\partial xy} h \\ \frac{\partial^2}{\partial xy} h & \frac{\partial^2}{\partial y^2} h \end{bmatrix}$$

The resulting $x \times 2$ matrix defines the structure of the result of probing the Hessian of F.

When the image data is not scalar (*e.g.*, the 2D vector field G mentioned above), we have further outer iteration over the shape of the image's range. For example, if $G = V \circledast h$ is a 2D vector field, then the probe $G(\mathbf{x})$ will be expressed as

$$\begin{array}{rcl}
G(\mathbf{x}) & = & (V \circledast h)(\mathbf{x}) \\
 & = & \begin{bmatrix} V_1 \circledast h \\ V_2 \circledast h \end{bmatrix} (\mathbf{x}) \\
 & = & \begin{bmatrix} \sum_i \sum_j V_1[\mathbf{n} + \langle i, j \rangle] h(\mathbf{f}_x - i) h(\mathbf{f}_y - j) \\ \sum_i \sum_j V_2[\mathbf{n} + \langle i, j \rangle] h(\mathbf{f}_x - i) h(\mathbf{f}_y - j) \end{bmatrix}
\end{array}$$

Note that while the result of this probe has the same type as $\nabla F(\mathbf{x})$, the underlying computation is different.

Our implementation must also correctly handle mapping between coordinate spaces. An image dataset comes with orientation information that can be represented as a transform $\mathbf{M}$ mapping from position in the image's index space to position in world space. We use the inverse mapping $\mathbf{M}^{-1}$ to convert the position of a probe, which is in world space, back to image space, but we also have to consider the mapping of the probe's result back to world space. Gradients are measured in image space by convolution with kernel derivatives (as described above). Being a *covariant* quantity, gradients are converted to world space by $\mathbf{M}^{-T}$, the inverse transpose of the transform $\mathbf{M}$ for converting positions (a *contravariant* quantity) to world space [27]. On the other hand, when probing a vector field created by convolution of a vector-valued image dataset, the vectors are assumed to be already represented in world space, and hence need no post-probe transformation.

The final step in generating executable code for field probes is to expand the kernel evaluations. This expansion takes place as part of the translation from MidIR to LowIR. The kernels that Diderot supports are all piecewise polynomial, so it straightforward to symbolically differentiate them. The process described in this section results in code that is easily vectorized (in fact the MidIR and LowIRs support vectorized operations).

5.4 Domain-specific optimizations

In addition to the transformations required to support the operations on fields that we have described above, the Diderot compiler performs a number of other optimizations. Specifically, we implement an extended form of constant folding and dead-code elimination that shrinks (or contracts) the program [1] and we eliminate redundant computations using value numbering [5]. While these are optimizations that are found in many compilers, when they are combined with the domain-specific operators in our IR, they produce domain-specific optimizations that a general-purpose compiler would be unlikely to achieve. For example, if a program probes both a field F and the gradient field ∇F at the same position, there are redundant convolution computations that can be detected and eliminated. Another example is the symmetry of the Hessian, which is also detected by our value-numbering pass.

5.5 Runtime support

The Diderot runtime is comprised of common code for loading image data from Nrrd files [29] and writing the program's output to either a text or Nrrd file.[2] The common part of the runtime also provides support for initializing input variables.

In addition to the common code, there is target-specific code for managing strands. Recall that Diderot uses a bulk-synchronous parallelism model [26, 31]. In this model, execution is divided into *super steps*; during a super-step each strand's update method is evaluated once. The program executes until all of the strands are either stabilized or dead. For the sequential target, the runtime implements this model as a loop nest, with the outer loop iterating once per super-step and the inner loop iterating once per strand.

The parallel version of the runtime is implemented using POSIX threads. The system creates a collection of worker threads (the default is one per hardware core/processor) and manages a work-list of strands. To keep synchronization overhead low, the strands in the work-list are organized into blocks of strands (currently 4096 strands per block). During a super-step, each worker grabs and updates strands until the work-list is empty. Barrier synchronization is used to coordinate the threads at the end of a super step.

[2] The Nrrd file format is designed for multi-dimensional image data and includes metadata for the image's coordinate system and axes.

6. Performance evaluation

In addition to providing a very-high-level programming model, we are also interested in providing high-performance; especially on modern multicore systems. In this section we present results from four benchmark programs that represent typical workloads for image analysis, including parallel scaling results for the Diderot implementations. We also compare Diderot's performance with handwritten C programs that use the Teem library [30]. Teem includes convolution-based reconstruction of values and derivatives from discretely sampled fields, which closely matches the mathematical abstraction of a field supported by Diderot, but with a less convenient API, as described in Section 7.

6.1 Experimental method

Our test machine is an 8-core MacPro with 2.93 GHz Xeon X5570 processors and 12Gb of memory running Mac OS X 10.7.2. We used the Apple `clang` C compiler (version 3.0) to compile the Teem version of the benchmarks and as a backend to the Diderot compiler. In both cases, code was compiled with optimization level `-O3`. For each benchmark, we report the average of 40 runs on a lightly-loaded machine; the standard deviations for these experiments were typically less than 0.1 seconds.

6.2 Benchmarks

We present results from four benchmark programs, which are summarized in Table 1. In the table we give the lines of code of both the Teem version (written in C) and the Diderot version. We further split out the lines of code in the computational core of the benchmark. For the Teem version, this number is the loop-nest that performs the computation, while for the Diderot program it is the **`update`** method.[3] From this table it can be seen that Diderot provides a significant advantage in conciseness over using the Teem library, which, itself is a big advantage over writing the code directly in C. The lines-of-code numbers do not include comments, blank lines, or timing code. The table also includes the number of initial strands for each program.

These benchmark programs, which were chosen to be representative of the kinds of applications for which Diderot has been designed, are described in more detail below.

vr-lite: A simple volume renderer that displays surfaces using Phong shading, which depends on the gradient in the scalar field. This simple program is typical of one that might be used in an educational context.

illust-vr: A more complex volume renderer that produces illustrative (or non-photo-realistic) renderings, using various curvature computations based on the gradient and Hessian. This example showcases the tensor calculations that are awkward to express in other languages.

lic2d: Computes a line integral convolution [7] visualization of a vector field, in which a noise texture is blurred along vector field streamlines. This program highlights convolution and differentiation in a vector field.

ridge3d: An initial uniform distribution of points within a portion of CT scan of a lung is moved iteratively towards the centers of blood vessels, using Newton optimization to compute ridge lines [11]. This program computes the eigenvalues and eigenvectors of the Hessian, and permits the implementation to closely resemble the mathematical definition of a ridge line.

[3] For the Teem versions of the benchmark, the computational core does *not* include the setup code used to specify the field properties that are being probed.

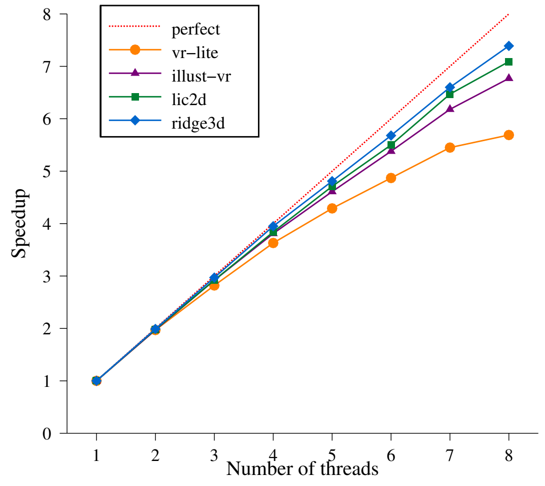

Figure 12. Parallel speedup

6.3 Measurements

For each benchmark program, we measured the wall-clock time it took to execute the computation kernel. This measurement excludes the initial loading of the image data, other initialization, and the writing of the result. Typically, the excluded time was on the order of 0.2 seconds. Table 2 presents the results of our experiments. For each benchmark program, we present the execution time for the Teem version, the sequential Diderot version, and the parallel version on 1, 2, and 8 processors. The Teem programs use a mix of single and double-precision arithmetic in their implementation (the Teem Library uses doubles internally). When compiling a Diderot program, on the other hand, the user must decide if reals are represented as single or double-precision floats. To verify that the difference in precision is not reason that Diderot is faster, we measured both the single and double-precision versions of the Diderot programs. From Table 2, it can be seen that while using double-precision arithmetic does result in a significant slow-down, it does not wholly explain the difference. We have done some detailed profiling of the Teem versions of several of the benchmarks, and we believe that a major part of the difference is Teem's use of callbacks to implement field probes. Diderot's use of vector operations (*i.e.*, SSE) may also account for some of the difference. Thus, compared to Teem, Diderot provides a more concise programming notation, better performance, and easier support for parallel computation.

6.4 SMP performance

The Diderot execution model is designed for both simplicity and efficient execution on parallel hardware. Since strands execute independently, we expect efficient scaling on parallel hardware. In Figure 12, we present the parallel speedup curves for the single-precision version of our benchmarks. We use the sequential version of these programs without the overhead of scheduling (*i.e.*, the **Seq.** column from Table 1). As we expect, all of the benchmarks scale well. For vr-lite, we see some tailing-off at eight threads, which we believe is because of lack of work (notice from Table 1 that vr-lite has the fewest strands). With some experimentation, we found that the biggest limitation to parallelism was the lock that controls access to the work-list. With smaller blocks of strands (recall that we use 4,096 strands per block), we saw a significant reduction in parallel scaling. Other than adjusting the strand-block size, we have

Program	LOC (total:core) Teem	Diderot	# strands	Description
vr-lite	223:44	68:26	165,600	Simple volume-renderer with Phong shading running on CT scan of hand
illust-vr	324:61	83:39	307,200	Fancy volume-renderer with cartoon shading running on CT scan of hand
lic2d	260:66	53:32	572,220	Line Integral Convolution in 2D running on synthetic data
ridge3d	360:55	44:24	1,728,000	Particle-based ridge detection running on lung data

Table 1. The benchmark programs

		Diderot single-precision				Diderot double-precision			
Program	Teem	Seq.	1P	2P	8P	Seq.	1P	2P	8P
vr-lite	26.77	14.92	14.95	7.59	2.62	16.52	16.44	8.35	2.92
illust-vr	132.85	54.17	54.40	27.55	8.00	80.63	82.16	41.18	11.86
lic2d	3.22	2.02	2.03	1.02	0.30	2.47	2.47	1.24	0.37
ridge3d	11.18	8.40	8.36	4.22	1.14	9.34	10.27	5.16	1.39

Table 2. Average performance results over 40 runs (times in seconds)

not done any performance tuning of the runtime, so we expect that we will be able to improve performance and scaling.

7. Related work

There are a variety of domain-specific languages and frameworks that provide similar features supported in Diderot. Many of these are examples of using the power of DSLs to provide programers with high-level parallel programming models.

Teem is a collection of libraries that support image analysis and visualization algorithms [30]. Teem provides support for convolution-based measurements via kernels, but as a C library, the programming model is much less direct. A Diderot programmer can declare a scalar field `F` and then use `F(pos)` and `∇F(pos)` for the field value and gradient at some point `pos`. A Teem programmer would have to create a probing context in which image data and kernels are set, specify the list of all quantities that are to be computed for every probe, and then update the probe context to allocate buffers to store probe results. After calling the probe function at a particular location `pos`, the programmer then copies the value and gradient out of the probe buffer.

Shadie is a DSL for direct-volume rendering applications that is targeted at GPUs [13]. The framework is based on the notion of shaders, which are functions that define what happens along a given ray for the entire visualization. Similar to Diderot, shaders support the ability to perform computations on continuous fields and their derivatives. But Shadie is limited to direct-volume rendering applications, whereas Diderot supports other visualization applications such as LIC, fiber tractography, and particle systems. Furthermore, Shadie's support for field operations is restricted to a set of built-in functions (*e.g.* `cubic_query_3d` for probing the gradient field reconstructed using a cubic spline), whereas Diderot provides a collection of orthogonal operations on fields.

Scout is a high-level DSL for image visualization and analysis [19]. Scout supports a data-parallel programming model based on the concept of *shapes*, which are regions of voxels in the image data. Scout is designed for a different class of algorithms than Diderot. Specifically, algorithms that are defined in terms of computations over discrete voxels, such as stencil algorithms, rather than over a continuous tensor fields.

Spiral is a DSL for digital signal processing algorithms [23]. Its implementation encapsulates significant mathematical knowledge of the various complex algorithms in digital signal processing, which allows Spiral to generate fine-tuned code for a given platform. This notion of developing a high-level mathematical programming model is also emphasized in Diderot. As with Spiral, Diderot allows its users to focus more on the mathematics and allowing the system to generate high-performance code for their platform. Although Spiral provides a powerful mathematical model, it targeted at the somewhat different domain of signal processing.

Delite is an ongoing project to support the development of embedded parallel DSLs [6, 8] and was the inspiration for the Diderot project. The framework aids in decreasing the burden of parallelizing DSL programs (*e.g.*, scheduling) and has been used for machine learning [28] and mesh-based PDEs [10]. While the Delite project and Diderot project share the idea that parallel DSLs are an effective way to provide portable parallelism, they differ in the way that the DSL is presented to the user. Delite embeds the DSL in Scala, which limits the notational flexibility of the design, whereas Diderot's syntax is designed to fit its application domain. Furthermore, Diderot's runtime has been designed to allow Diderot programs to be embedded as libraries in any host language that supports calling C code, whereas Delite applications must be written in Scala, which is not a common language in the visualization community.

8. Future work

While the current version of Diderot is sufficient to implement many interesting visualization algorithms, we have plans to extend the language design, add additional targets, and otherwise improve the system. These plans include obvious extensions, such as adding functions and simple data structures, as well as extensions targeted at image analysis algorithms. We describe some of the latter plans here.

8.1 Portable parallelism

One implementation challenge of Diderot is to extract maximum performance across a wide range of different parallel architectures. Diderot already includes an efficient multicore parallel implementation. We also have an OpenCL backend that is targeted at GPUs, but we plan to extend our implementation to support clusters, including GPU clusters.

8.2 Richer concurrency model

The current design of Diderot limits strand creation to initialization time and does not provide any mechanism for strand communication. While the number of active strands can dynamically vary, because of **`die`**, it only monotonically decreases. For algorithms, such as particle systems, where strands are used to explore the image space, it is useful to be able to dynamically create new strands on the fly. But it only makes sense to create new strands when there is a region that is relatively empty of strands. Therefore, we plan to extend the programming model with three mechanisms: a mechanism for creating new strands, a mechanism for reading the state of nearby strands, and a mechanism for global computations (*i.e.*, reductions) over the whole set of active and stable strands. Keeping with our bulk-synchronous semantics, these mechanisms will be tied to the super steps. New strands will come into existence at the end of the super step, strands will only be able to read the state of other strands as it was at the beginning of the super step, and the global computations will occur at the end of the super step.

8.3 More tensor math

We plan to extend our implementation to support larger set of tensor and field operations, such as divergence ($\nabla\bullet$) and curl ($\nabla\times$). To support this richer set of operators, however, we will have to make changes in our internal representation. Specifically, we are exploring the use of Einstein notation as a compact way to represent tensor computations.

9. Conclusion

We have presented Diderot, a parallel domain-specific language for image analysis and visualization algorithms. Diderot provides the programmer with a very-high-level programming model based on the concepts and notations of tensor calculus, which allows the algorithms to be expressed in their natural mathematical notation. We have described the techniques that we use to implement this high-level model and have presented performance data the demonstrates that we can have both high-level notation and high-performance in the same system.

Acknowledgments

We thank the NVIDIA Corporation for their generous donation of both hardware and financial support. This material is based upon work performed in part while John Reppy was serving at the National Science Foundation.

References

[1] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *JFP*, 7:515–540, Sept. 1997.

[2] S. R. Aylward and E. Bullitt. Initialization, noise, singularities, and scale in height ridge traversal for tubular object centerline extraction. *IEEE TMI*, 21(2):61–75, Feb 2002.

[3] R. Bartels, J. Beatty, and B. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann Publishers, New York, NY, 1987.

[4] P. J. Basser, S. Pajevic, C. Pierpaoli, J. Duda, and A. Aldroubi. In vivo fiber tractograpy using DT-MRI data. *Magnetic Resonance in Medicine*, 44:625–632, 2000.

[5] P. Briggs, K. D. Cooper, , and L. T. Simpson. Value numbering. *SP&E*, 27(6):701–724, June 1997.

[6] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT '11*, Oct. 2011.

[7] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *SIGGRAPH '93*, pages 263–270, New York, NY, Aug. 1993. ACM.

[8] H. Chafi, Z. Devito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *OOPSLA '10*, pages 835–847, Oct. 2010. Part of the Onward! 2010 Conference.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, Oct 1991.

[10] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *SC '11*, pages 1–12, Nov. 2011.

[11] D. Eberly. *Ridges in Image and Data Analysis*. Kluwer Academic Publishers, Boston, MA, 1996.

[12] *Using vector instructions through built-in functions*. Free Software Foundation. URL `http://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html`.

[13] M. Hašan, J. Wolfgang, G. Chen, and H. Pfister. Shadie: A domain-specific language for volume visualization. Draft paper; available at `http://miloshasan.net/Shadie/shadie.pdf`, 2010.

[14] L. Ibanez and W. Schroeder. *The ITK Software Guide*. Kitware Inc., 2005.

[15] *IDL: Interactive Data Language*. ITT Visual Information Solutions. `http://www.ittvis.com/ProductServices/IDL.aspx`.

[16] *The OpenCL Specification (Version 1.1)*. Khronos OpenCL Working Group, 2010. Available from `http://www.khronos.org/opencl`.

[17] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Moller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *VIZ '03*, pages 67–74, Los Alamitos, CA, Oct. 2003. IEEE Computer Society Press.

[18] *MATLAB — The language of technical computing*. The Mathworks, Inc. `http://www.mathworks.com/products/matlab`.

[19] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: A data-parallel programming language for graphics processors. *Journal of Parallel Computing*, 33:648–662, Nov. 2007. ISSN 0167-8191.

[20] E. H. W. Meijering, W. J. Niessen, and M. A. Viergever. Quantitative evaluation of convolution-based methods for medical image interpolation. *Medical Image Analysis*, 5(2):111–126, June 2001.

[21] M. D. Meyer, P. Georgel, and R. T. Whitaker. Robust particle systems for curvature dependent sampling of implicit surfaces. In *SMI '05*, pages 124–133, June 2005.

[22] *NVIDIA CUDA C Programming Guide (Version 4.0)*. NVIDIA, May 2011. Available from `http://developer.nvidia.com/category/zone/cuda-zone`.

[23] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE*, 93(2):232–275, Feb. 2005.

[24] G. X. Ritter and P. D. Gader. Image algebra techniques for parallel image processing. *JPDC*, 4(1):7–44, Feb. 1987.

[25] *NumPy: Numerical Python*. Scientific Tools for Python. `http://numpy.scipy.org`.

[26] D. Skillicorn, J. M. Hill, and W. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[27] B. Spain. *Tensor Calculus: A Concise Course*. Dover, Mineola, NY, 2003.

[28] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *ICML '11*, June 2011.

[29] *NRRD: Nearly Raw Raster Data*. Teem Library, . `http://teem.sf.net/nrrd`.

[30] *Teem website at* `http://teem.sf.net`. Teem Library, .

[31] L. G. Valiant. A bridging model for parallel computation. *CACM*, 33 (8):103–111, Aug. 1990.