Diderot: A Parallel DSL for Computing on Multi-Dimensional Tensor Fields

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

Nicholas Seltzer [†] Lamont Samuels John Reppy Gordon L. Kindlmann

Department of Computer Science; University of Chicago

ABSTRACT

We describe ongoing work developing Diderot, a new parallel domain-specific language (DSL) for computing on continuous fields reconstructed from multi-dimensional images. Diderot uses a syntax similar to mathematical conventions for expressing convolution, differentiation, and tensor operations, making it easy to code computationally complex algorithms. In addition to being easy to code, programs written in Diderot will be compiled for several different parallel architectures, with many optimizations handled automatically by the compiler.

Keywords: Image Data Visualization, Image Analysis, Parallelism, Domain-Specific Language

1 INTRODUCTION

Many visualization programs are difficult to write, even though the algorithms may involve simple mathematical principles. We frequently need to take derivatives of multi-dimensional data or handle tensors which result from such operations. Such ideas can usually be expressed with mathematical notation in one short line, but implementing them may require hundreds of lines of code or require the use of complex libraries. Diderot is a parallel DSL with high-level mathematical notation for expressing computations with tensors and tensor fields. Diderot naturally handles fields of any dimension and tensors of any order. Moreover, Diderot programs will be portable to multiple hardware targets without changing the code at all. Our goal is a language for easily managing computations on continuous fields reconstructed from discrete datasets that leads to efficient, optimized, portable, and parallel code.

Diderot is inspired by previous parallel DSLs. One such language is Scout, a general data-parallel language [6] compiled for GPUs. Another DSL, Shadie, is a a GPU-based volume visualization framework which emphasizes code simplicity [7]. Other work seeks to compile a particular domain of algorithms to a range of parallel architectures such as SMPs, clusters, and GPUs. For example, Liszt is a DSL for expressing computations on finite element meshes and for compiling code to a variety of architectures [3]. Diderot follows this approach, restricting the domain of the language to computations on continuous tensor fields (including scalar and vector fields).

2 LANGUAGE ABSTRACTIONS

Diderot has several built in types and operations for expressing high-level mathematical computations and abstracting away discrete image data. The syntax of Diderot expressions mirrors conventional mathematical notation, making programs easier to read.

The `tensor` is the basic computational type. Tensors are declared as: `tensor[`d_1`,`d_2`,...,`d_m`]`. The `[`d_1`,`d_2`,...,`d_m`]` ($d_i > 1$) is the shape of an m^{th} order tensor. Common tensor types have aliases, e.g., `real` means `tensor[]` and `vec3` means `tensor[3]`. Diderot supports addition ("+") and subtraction ("−") of tensors, as well as common vector and tensor products (i.e., inner p • q, cross p × q, outer p ⊗ q).

[†]nseltzser@cs.uchicago.edu

The `image` type represents multi-dimensional arrays of tensors, and stores meta-data about how the array is oriented in world space. The image can have any number of dimensions and store tensors of any order. Images are declared as: `image(`n`)[`d_1`,`d_2`,...,`d_m`]`. n is the dimension of the image array ($n > 0$) and `[`d_1`,`d_2`,...,`d_m`]` is the shape of the tensors stored in the image. For example, to load a 3D image of scalars:

```
image(3)[] my_data = load ("data.nrrd");
```

Diderot uses univariate piecewise polynomial kernels to reconstruct continuous fields from discrete data by separable convolution. Each kernel has a C^k continuity level that limits the number of times a field reconstructed with the kernel may be differentiated to k. Diderot provides kernels such as C^2 Cubic B-spline `bspln3`, C^1 Catmull-Rom `ctmr`, and C^0 `tent` for linear interpolation.

The `field` type combines kernels and discrete data to create the abstraction of smooth tensor fields over continuous world-space. Fields are declared as `field#`k`(`n`)[`d_1`,`d_2`,...,`d_m`]`, where k is the continuity level. Convolving 3D image `my_data` with a cubic B-spline creates C^2 scalar field `my_field`:

```
field#2(3)[] my_field = bspln3 ⊛ my_data;
```

Besides convolution, fields are created by adding, subtracting, and scaling existing fields, as well as by differentiation:

```
field#1(3)[3] gradient = ∇my_field;
field#0(3)[3,3] Hessian = ∇⊗∇my_field;
```

∇, $\nabla\times$, and $\nabla\otimes$ are higher-level functions that apply analytic differentiation to fields. Fields are not pre-computed; they are abstract functions that can be evaluated at any location to produce a tensor. Kernel differentiation and convolution are not computed until a field is probed at a particular world-space location.

3 COMPUTATION MODEL

Diderot is a C-like language extended with high-level mathematical notation for expressing computations on tensors and tensor fields. Diderot organizes the computation into mostly-autonomous *strands*, which are the unit of parallelism. Each strand has local state and an update method. The execution model is bulk-synchronous and deterministic. Each iteration updates every strand independently. Execution continues until all strands either `stabilize`, producing output, or `die`, with no output.

A Diderot program has a simple structure composed of three parts: global declarations, strand definitions, and strand initialization. Global declarations include variables readable by all strands. The strand definition includes some parameters, strand-local variables (and their initialization), and an update method. Finally, strand initialization creates all the strands with the correct parameters. Like shader languages, Diderot global variables are immutable, but local strand-instance variables are mutable. The following is a simple Diderot program which creates 50 strands that output the numbers 0.0 to 49.0:

```
int numStrands = 50;
strand my_strand (int i) {
    output real out = real(i);
    update { stabilize; }
}
initially [ my_strand(i) | i in 0..(numStrands - 1) ];
```

Not only does Diderot make code easier to write, it enables efficient executables for a variety of architectures, taking advantage of the highly parallel nature of such algorithms. The Diderot compiler currently supports both sequential and SMP executables, and a GPU (OpenCL) version is under development. Traditionally, parallel programs must be tuned for different platforms by hand, but, because Diderot is designed for a specific domain of computation, it is possible for the compiler generate efficient code for significantly different target platforms. Domain knowledge also allows the compiler to take advantage of target features such as SSE vector instructions automatically. Because Diderot is implemented as a compiler, instead of as a library, it is able to optimize across the whole program. For example, it can eliminate redundancies in probing a field's value, gradient, and Hessian.

4 Example Applications

The language abstractions and computational model discussed above support a wide range of visualization and analysis algorithms. Direct volume rendering is one natural application of Diderot. Direct volume rendering creates an image by casting one ray per pixel through a field and determining color from local field properties [4]. The computation of each ray is independent, so we create one strand per ray, with an update function that increments along the ray and updates the color according to the transfer function. Local curvature, summarized in a geometry tensor, `G`, can be used in transfer functions [5] (Figure 1):

```
tensor[3] norm = normalize(∇F(p));
tensor[3,3] P = identity[3] - norm⊗norm;
tensor[3,3] G = -(P•(∇⊗∇F(p))•P)/|∇F(p)|;
```

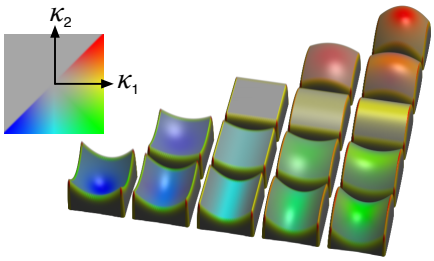

Figure 1: Volume rendering of a synthetic dataset with a transfer function designed to illustrate curvature computation.

Another application of Diderot is line integral convolution (LIC), which visualizes a vector field by blurring a noise texture (Figure 2) [1]. The computation is independent for each pixel, so in Diderot, we can create one strand for each pixel. At every iteration, each strand takes a step along the direction of the underlying vector field and adds the contribution of the new location. A second order Runge-Kutta (Midpoint method) integration updates position `p` in vector field `V` with step size `h` via:

```
p += h*V(p + 0.5*h*V(p));
```

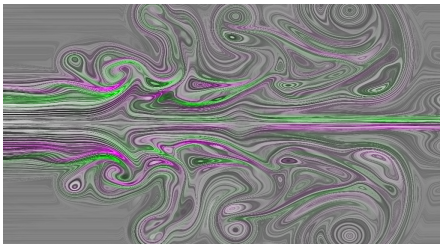

Figure 2: LIC of a turbulent vector field with color based on curl, expressed in Diderot as $\nabla\times$`V(p)`.

Another natural application of Diderot is edge detection (Figure 3). One simple method is to create a separate strand for every pixel and classify edge pixels with a threshold on gradient magnitude and with non-maximal suppression [2]. Each pixel can be classified in parallel:

```
vec3 step = h*normalize(∇F(p));
if(|∇F(p)| > threshold &&
        -|∇F(p+step)| + 2*|∇F(p)| - |∇F(p-step)| > 0)
```

A more refined technique would create a particle system, initialized with particles evenly spread over the image domain. The solution would be computed with one strand per particle, using Newton-Raphson iterations to move them along image gradients towards maxima of gradient magnitude.

Figure 3: Edge detection of a face.

5 Future Work

While we have been able to write several non-trivial Diderot programs, Diderot is still a work in progress. Work is being done to allow strands to communicate and share information. We also plan to support the dynamic creation of strands after computations have started and to add a global computation step to be executed after each iteration. These additions will increase the range of algorithms supported by Diderot. We are also working to make Diderot code callable as a C library. With callable libraries, it will be easy to combine Diderot computations with other processing, and to chain Diderot computations together. Future work will include porting Diderot to MPI clusters. We expect one of the first applications of Diderot will be in education, allowing students to focus on high-level algorithmic design and still get highly parallel executables that efficiently process real-world datasets.

References

[1] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *Proc. SIGGRAPH*, pages 263–270, 1993.

[2] J. Canny. A computational approach to edge detection. *Trans. Pattern Analysis and Machine Intelligence*, 8(6):679–714, 1986.

[3] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Proc. ACM OOPLSA*, pages 835–847, 2010.

[4] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *Proc. SIGGRAPH*, pages 65–74, 1988.

[5] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proc. Visualization 2003*, pages 513–520, 2003.

[6] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: a data-parallel programming language for graphics processors. *Parallel Computing*, 33:648–662, Nov 2007.

[7] Shadie: A domain-specific language for volume visualization. `http://code.google.com/p/shadie/`.

Diderot

A Parallel Domain Specific Language for Computing on Multi-Dimensional Fields

Nicholas Seltzer, Lamont Samuels, John Reppy, Gordon Kindlmann
Department of Computer Science; University of Chicago
http://diderot-language.cs.uchicago.edu

THE UNIVERSITY OF CHICAGO COMPUTER SCIENCE

1. Introduction

Diderot is a language designed for efficient analysis and visualization of multi-dimensional images. By supporting high-level mathematical abstractions in a familiar notation, Diderot makes common image analysis and visualization algorithms easier for domain experts to implement. Diderot also makes it easy to take advantage of the extensive parallelism present in these algorithms without forcing the programmer to learn the low-level details of the target platform.

Diderot is inspired by previous parallel domain specific languages. Scout [6] compiles general-purpose data-parallel programs to GPUs. Shadie [7] simplifies creating GPU-based volume renderers with concise programs. Other DSLs trade generality in parallel architectures for specificity in algorithms, such as Liszt [3] for finite element mesh codes. Diderot follows this approach, being restricted to computations on continuous tensor fields (including scalar and vector fields).

2. Portable Parallelism

Many image analysis and visualization tasks benefit heavily from parallelization. In the examples, the computations are parallelized over per-pixel ray integrals (volume renderings, Figures 1 and 3), per-pixel streamlines (LIC, Figure 2), or other per-pixel computations (edge detection, Figure 4). Diderot is designed to support parallel implementation across a wide range of parallel platforms. By restricting the domain of applications for Diderot, it can achieve better performance on a wide variety of parallel hardware.

Parallelism model

- Diderot computation is divided into lightweight threads called *strands*
- Each strand has some parameters used to distinguish its starting state from the other strands
- All strands have access to global variables, including images and fields
- Each strand has its own local state
- Programs execute in a bulk synchronous fashion, with each strand updated in each step
- Strands terminate by either
  - stabilizing, in which case their state contributes to the output
  - dying, in which case the strand's state is discarded

Portability across different hardware platforms

- We distinguish between the parallelism model the programmer uses and the hardware the program is being run on
- Diderot programs can be run
  - Sequentially on a single CPU core
  - With threads on multiple CPU cores
  - On a GPU
- Diderot also takes advantage of hardware vector instruction sets like Intel SSE

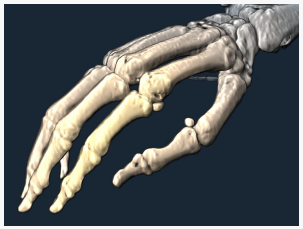

Figure 1: Illustrative volume rendering of a CT scan [4, 5]. This illustrative volume rendering of a CT scan of a human hand is a typical example of a possible application for Diderot. Every strand computes a separate ray integral in parallel, one per-pixel of the output image. The final image was colored with a curvature-based transfer function involving second derivatives of the data (computed in a manner similar to that seen in Figure 3).

3. Language Details

Diderot is designed for expressing parallel computations on tensor fields. Familiar mathematical notation is included for manipulating tensors and continuous fields. In addition to making it easier for domain experts to rapidly implement a variety of image analysis and visualization algorithms, the simplicity of Diderot's notation makes it an ideal candidate language for educational settings where students may not have time to learn more complex methods.

The source code in figure 2 shows each section of the program structure. An example of the use of Diderot's mathematical abstractions is included with figure 3.

Program structure

Globals

- The first part of each program declares program-wide global variables
- This data is not modified once the strands begin computation
- The image datasets, and fields created from them, are globals

Strand definition

- Parameters describe the initial state of the strands
- Local variables make up the state of each strand
- An update function expresses one iteration of per-strand computation

Strand initialization

- Creates the initial parameterized set of strands

High-level mathematical abstractions

Tensors

- tensor[d_1,d_2,...,d_m] is an m^{th} order tensor where d_i denotes the length of the tensor along dimension i
- real and vec3 are aliases for tensor[] and tensor[3] respectively
- Common tensor operations (+, -, •, ×, ⊗) in unicode and ascii are available

Input image datasets

- Parallel computations in image datasets are the purpose of Diderot programs
- image(n)[d_1,d_2,...,d_m] is an n-dimensional array of tensor[d_1,d_2,...,d_m]s
- The image is to be used to reconstruct a continuous field
- Lookup tables and volume rendering transfer functions are also read as images and accessed as fields

Continuous fields

- field#k(n)[d_1,d_2,...,d_m] is an n-dimensional field of tensor[d_1,d_2,...,d_m]
- The #k indicates that the field is C^k continuous – it can only be differentiated k times, and further differentiation is a type error
- Fields are created by convolving (⊛) images with various univariate kernels (e.g. tent and bspln3) – Diderot uses separable convolution for multi-dimensional fields
- Fields can be evaluated at arbitrary locations in the domain
- Common field operations (+, -, ∇, ∇×, ∇⊗) in unicode and ascii are supported

Figure 2: Line Integral Convolution [1]. This LIC was created with the midpoint integration method, with coloration indicating curl. The pieces of Diderot code shown below were selected to illustrate the structure of the program used to create this LIC.

```
field#2(2)[2] V = load("vectors.nrrd") ⊛ bspln3;
field#0(2)[] R = load("noise.nrrd") ⊛ tent;
strand LIC (int xi, int yi) {
    vec2 p = [xi,yi];
    output real sum = 0;
    update {
        p += h*normalize(V(p +
                0.5*h*normalize(V(p))));
        sum += R(p);
        if (step == stepMax) {stabilize;}
    }
}
initially [ LIC(xi, yi) | yi in 0..(imgSizeY-1), xi in 0..(imgSizeX-1) ];
```

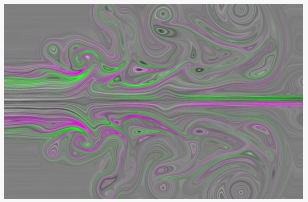

Figure 3: Volume rendering of a synthetic dataset [5]. This volume rendering of a 3-dimensional scalar field was designed to demonstrate computations of curvature using second derivatives of the field. The code segment evaluates the principle curvatures K1 and K2 at position pos:

```
image(3)[] I = load("data.nrrd");
field#2(3)[] F = I ⊛ bspln3;

vec3 grad = -∇F(pos);
tensor[3,3] H = ∇⊗∇F(pos);
vec3 norm = normalize(grad);
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;
```

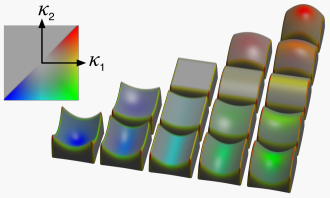

Figure 4: Edge detection [2]. Edge detection is a natural application for Diderot, since edges can be classified by finding maxima in the gradient magnitude of an image, and computations can be done in parallel in different parts of the image. The notation makes determining the gradient trivial and significantly simplifies the task of finding local maxima.

```
Non-maximal suppression:
vec3 step = h * normalize(∇F(p));
if (|∇F(p+step)|<|∇F(p)| &&
        |∇F(p – step)|<|∇F(p)|) {
    RGB += [c*|∇F(p)|, c*|∇F(p)|, 0];
    }
```

4. Future Work

Expand range of supported algorithms

- Allow strands to interact and communicate with each other
- Allow dynamic creation of strands
- Add a global computation phase

Expand range of target platforms

- Add support for MPI clusters

Usability improvements

- Make Diderot executables callable as libraries

REFERENCES

[1] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In Proc. SIGGRAPH, pages 263–270, 1993.
[2] J. Canny. A computational approach to edge detection. Trans. Pattern Analysis and Machine Intelligence, 8(6):679–714, 1986.
[3] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In Proc. ACM OOPLSA, pages 835–847, 2010.
[4] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In Proc. SIGGRAPH, pages 65–74, 1988.
[5] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. M¨oller. Curvature based transfer functions for direct volume rendering: Methods and applications. In Proc. Visualization 2003, pages 513–520, 2003.
[6] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: a data-parallel programming language for graphics processors. Parallel Computing, 33:648–662, Nov 2007.
[7] Shadie: A domain-specific language for volume visualization. http://code.google.com/p/shadie/.