ViSlang: A System for Interpreted Domain-Specific Languages for Scientific Visualization

Peter Rautek, Stefan Bruckner, M. Eduard Gröller, and Markus Hadwiger

Abstract— Researchers from many domains use scientific visualization in their daily practice. Existing implementations of algorithms usually come with a graphical user interface (*high-level interface*), or as software library or source code (*low-level interface*). In this paper we present a system that integrates domain-specific languages (DSLs) and facilitates the creation of new DSLs. DSLs provide an effective interface for domain scientists avoiding the difficulties involved with *low-level interfaces* and at the same time offering more flexibility than *high-level interfaces*. We describe the design and implementation of ViSlang, an interpreted language specifically tailored for scientific visualization. A major contribution of our design is the extensibility of the ViSlang language. Novel DSLs that are tailored to the problems of the domain can be created and integrated into ViSlang. We show that our approach can be added to existing user interfaces to increase the flexibility for expert users on demand, but at the same time does not interfere with the user experience of novice users. To demonstrate the flexibility of our approach we present new DSLs for volume processing, querying and visualization. We report the implementation effort for new DSLs and compare our approach with Matlab and Python implementations in terms of run-time performance.

Index Terms—Domain-specific languages, Volume visualization, Volume visualization framework.

1 INTRODUCTION

Domain-specific languages (DSLs) offer increased expressiveness compared to general purpose programming languages and higher flexibility compared to graphical user interfaces at low computational overhead. By abstracting the details of the computer soft- and hardware, the user can focus on the relevant (i.e., domain-specific) problems. The cost of learning new DSLs pays off for the user if the language is focused enough to avoid code that is not related to the actual domain, and if at the same time the language is expressive enough to solve the relevant problems. Domain-specific languages are especially well-suited in situations where the low-level software APIs and the high-level domain problems are far apart in syntax and semantics. Scientific visualization in general is a good target for employing DSLs for exactly these reasons. For instance, the implementation of a feature extraction and rendering algorithm requires a lot of knowledge and code to arrive at even a basic setup. This basic setup (that is not specific to the actual algorithm) includes data loading, initialization and state manipulation of the graphics processing unit, as well as the windowing system. The actual algorithm is only a small portion of the code. Even more severe is the gap between parallel hardware programming and high-level domain problems. A domain scientist is ideally not confronted with the complexity of programming highly parallel hardware (e.g., graphics processing units). At the same time experts need access to the processing power of GPUs and need the flexibility to combine multiple visualization algorithms to answer specific questions. GPUs are a powerful, cost-effective and widely available data processing infrastructure. General purpose computing languages that are executed on the GPU, like CUDA and the open standard OpenCL, expose the computing power for all kinds of applications. Although many domain scientists are knowledgeable in programming, the parallel nature of GPUs bears additional challenges that often require in-depth knowledge in computer science and software engineering. Therefore, the programming of GPUs is usually done by computer scientists or knowledgeable software developers. DSLs can address the large gap between low-level programming and high-level problems using abstraction of algorithms. The low-level problems are hidden by the high-level DSL allowing a larger audience to make use of the underlying computing infrastructure.

In this paper we present a system that lowers the cost of developing novel DSLs. Further it integrates multiple DSLs in one solution to leverage their flexibility. Figure 1 gives an overview of our design. ViSlang is a library and an execution environment with an extension mechanism. By integrating it into a visualization system the runtime can execute commands and thereby modify the behavior of the visualization system. The ViSlang runtime acts as the unified programming interface to the user. DSLs that extend ViSlang are called *slangs*. User input is executed by the ViSlang interpreter and commands that start with the keyword *using* are forwarded to the corresponding slang. In the example of Figure 1, the user writes a small program that makes use of the slang *renderer*. The slang *renderer* offers a DSL that allows to map data properties to visualization properties. A function *updateRendering* is declared and a *trigger* is defined that executes the function whenever the value of the variable x is changed. At the right of Figure 1 the user tries different values for variable x, leading to interactive updates of the visualization. The example of Figure 1 gives an overview of the ViSlang runtime system. In practice multiple DSLs are used in conjunction to combine data processing and visualization modules going beyond the functionality of the individual algorithms. ViSlang currently focuses on processing and visualization of static volumes. This is not a restriction of the system design but of its current implementation. However, one of the major design goals of ViSlang is its extensibility and other data structures can be added over time.

The major contribution of this paper is a language and system design that addresses the major risks and challenges of DSLs while preserving their benefits as listed by Van Deuresen et al. [37]. Specifically we address the following issues with our design:

Extensibility: ViSlang can integrate multiple domain-specific languages, addressing the issue that one DSL is not expressive enough for all problems across all domains that make use of visualization. With this approach we anticipate that algorithms address different aspects of visualization with concise yet expressive languages. We show that a procedural DSL is well suited for querying volumes, while a declarative DSL is well suited for the configuration of a visualization algorithm. Further, we propose a combination with a functional DSL to compute certain statistics of the data.

- *Peter Rautek is with KAUST, E-mail: peter.rautek@kaust.edu.sa.*
- *Stefan Bruckner is with University of Bergen, E-mail: stefan.bruckner@uib.no.*
- *M. Eduard Gröller is with Vienna University of Technology and VrVis Research Center, E-mail: meister@cg.tuwien.ac.at.*
- *Markus Hadwiger is with KAUST, E-mail: markus.hadwiger@kaust.edu.sa.*

Manuscript received 31 Mar. 2014; accepted 1 Aug. 2014; date of publication xx xxx 2014; date of current version xx xxx 2014. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

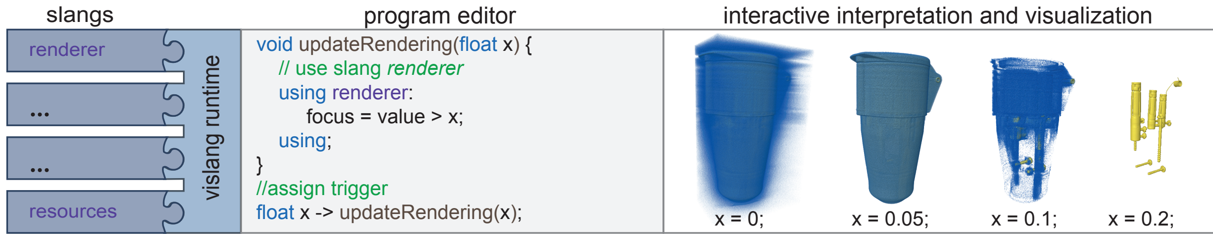

Fig. 1. System Overview: Slangs are DSLs that encapsulate a certain functionality and register it with the ViSlang runtime. The ViSlang runtime implements an embedded interpreter that executes programs in an interactive environment. The user interacts with an integrated program editor to issue ViSlang commands.

Implementation Overhead: We show that ViSlang keeps the implementation overhead for a new DSL low by using recurring patterns. Elements of the grammar can be reused, thereby reducing the effort to parse, analyze, and execute a new DSL. Data structures optimized for parallel algorithms can be reused to enable parallel execution without the low-level issues of GPU programming. We argue that the implementation overhead for new DSLs is small and that the benefits outweigh the costs. To further reduce the overhead of integrating a new DSL with ViSlang we provide a meta-language that instantiates templates to create new slangs.

Low-Level vs. High-Level Programming: Algorithms that are executed in parallel are generally harder to program than sequentially executed programs. It is desirable to offer an easier interface to these algorithms for non experts. In many cases critical parts of algorithms can be formulated independently of the semantics of parallel execution. These parts can be made accessible to inexperienced programmers using a DSL. We show an implementation of a parallel *map-reduce* algorithm. The mapping function is exposed as a DSL while the complexity of the implementation of the parallel reduction is hidden. We describe the implementation of a rendering algorithm and a logical query language. Both algorithms hide the low-level details of the parallel execution and offer a high-level programming interface.

Compatibility: All these challenges are addressed, maintaining compatibility to existing user interface concepts. We demonstrate how to add DSLs without changing existing interfaces. This offers increased flexibility on demand for expert users.

In this paper we first review related work in Section 2. We describe a set of slangs that demonstrate the different kinds of DSLs in Section 3. The extension mechanism and the meta-language are described in Section 4. The integration of reusable data structures that are optimized for parallel execution is described in Section 5. We discuss how implementation overhead is reduced and present runtime measurements in Section 6. The description of data types, and statements in ViSlang that are tailored for visualization, as well as the general control flow are described in Appendix A of the supplemental material. The ViSlang grammar in EBNF is given in Appendix B.

2 Related Work

Several approaches are commonly used to address the complexity of low-level interfaces for facilitating abstraction. For instance, software libraries are low-level interfaces that offer abstractions of algorithms and data structures. However, they require a lot of programming experience, and are not natively integrated in run-time environments. To allow a user to manipulate certain aspects of a visualization system at run-time, several approaches have been used. *Turn-key* user interfaces are a simple mechanism that enables the manipulation of parameters. They are frequently used due to the ease of implementation. However, they limit the flexibility of the user to certain parameter settings. Data flow systems encapsulate software modules and expose them to the user in a graphical user interface. Using the drag and drop metaphor, networks of data flows are specified connecting different modules. Our approach was inspired by the observation that the common visual programming approach of data flow systems tends to oversimplify the situation. It is not possible for the user to try variations of the algorithms by implementing the configurable parts of the software. This leads to systems that are not flexible enough for the user to take full advantage of the available visualization algorithms. Figure 2 shows the different levels of run-time interfaces of a visualization application and the corresponding target user group ordered by increasing level of expertise.

The novice user gets an application that is pre-configured for classical *turn-key* interaction. The data flow interface is used by more advanced users that re-configure the application at run-time. The expert user profits from increased flexibility, and might use interfaces on all three levels. Typically, only expert programmers have the skills to modify existing source code and to reuse low level libraries.

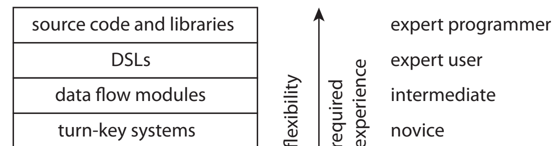

Fig. 2. Interfaces by increasing flexibility and user experience: Users of different levels of experience get different views on an application. Novice and intermediate users adapt parameters and combine modules with a graphical user interface. Experts program certain aspects of modules (DSLs) or entirely new functionality.

2.1 Domain-Specific Languages for Visual Computing

Domain-specific languages have been extensively used in visual computing for a long time. Graphics APIs like OpenGL and shader languages like HLSL, GLSL, Cg, etc. are low-level interfaces for programming the graphics hardware of modern computer systems. These APIs are used to modify the state and the data stored on the graphics card. API calls initiate the graphics pipeline to process data primitives and to compute pixel colors. Shaders are a good example of widely used DSLs for programming parts of the graphics pipeline. More recently general purpose computing languages, like CUDA and OpenCL, are available that enable programming of graphics hardware in a more flexible way. These languages are APIs for parallel computing applications in the field of visual computing and beyond. As they are low-level APIs they do not address the special needs of non-experts. A large body of previous work addresses the gap between low- and high-level APIs.

For instance the work of Hultquist and Raible [15] targets computational fluid dynamics applications. It is conceptually similar to our approach as it presents a system with an interpreter supporting the rapid development of new visualization components. Computationally expensive components are written in C, while the rapid prototyping functionality is offered to the user at run-time using Scheme as a programming language. We follow a similar approach but integrate support for the creation of entirely new DSLs that can execute on parallel hardware.

Johnson and Huang [17] present a declarative DSL for specifying feature classes and for deriving statistical matches of these classes. Their language is a good example for a concise yet powerful DSL.

Chiw et al. [6] describe Diderot, a domain-specific language for image analysis and visualization. The language is tailored to the rapid prototyping of algorithms that are executed on parallel hardware. The language includes support for concepts and notations well known from tensor calculus. This high level of abstraction allows the programmer to write tensor operations using familiar notation. The authors demonstrate that the program runs very efficiently, although it is specified with high-level concepts. Hašan et al. [13] present Shadie, a domain-specific (Python like) language that is translated to CUDA kernels. The run-time handles data loading, and parameters are automatically passed to the CUDA kernels. The recent work of Choi et al. [7] presents Vivaldi, a DSL inspired by Shadie. It supports execution on distributed GPU architectures. McCormick et al. [23, 24] and Jablin et al. [16] present Scout, a high-level hardware accelerated language. Scout programs are compiled at run-time and executed on the GPU, hence omitting the complexity of parallel programming and still profiting from a highly parallel architecture. Stockinger et al. [34] present a framework for efficient evaluation of range queries. Queries are specified by the user as logical combinations of range selectors for large multivariate data. The queries that define the interesting parts of the data are evaluated and the results are visualized.

Ragan-Kelley et al. [31] present Halide, a domain-specific language for image processing pipelines. By separating algorithmic code from code that optimizes the execution, a concise language is presented that executes fast on parallel hardware.

Each of these approaches is focused on one specific aspect of the data analysis and visualization pipeline. The work of Johnson and Huang [17] focuses on distributions of certain properties, Diderot [6] on tensor calculus, Shadie [13] on approximating the volume rendering integral, Scout [16, 23, 24] on the visualization mapping, Halide [31] and Vivaldi [7] on efficient image and volume processing, and the work of Stockinger et al. [34] on data selection by querying for data ranges. Unlike these approaches, we present a language design that generalizes the problem of integrating DSLs into visualization systems. We address the need for more than one domain-specific language in visualization with the extensibility of our language. This allows us to implement different slangs for different aspects of the visualization pipeline.

Duke et al. [9] have used Haskell to integrate three different DSLs to solve a specific visualization problem with a functional programming language. They report on the benefits of employing multiple DSLs for scientific visualization in their work [10]. Brown et al. [4] present Delite, a framework to generate embedded DSLs that can execute on heterogeneous parallel hardware. They show OptiML as an example DSL for machine learning algorithms. We follow a similar idea and also suggest to generate and integrate novel DSLs in one common framework. However, we present a full system that integrates DSLs with a visualization environment and use an interpreted language to provide immediate visual feedback for the end user. Further, our framework does not require the DSLs to be embedded DSLs and has less restrictions on the syntax of new DSLs.

The usefulness of DSLs and abstraction of low-level implementation details in scientific visualization was previously also demonstrated on massively parallel clusters. The work of Glatter et al. [11], and Kendall et al. [19] are noteworthy examples that demonstrate the advantages of DSLs as interfaces. Vo et al. [38] present results of using a well known MapReduce framework for visualization. Although ViSlang is not meant to run on clusters, but rather on GPUs, it is similar as it separates low-level implementation details from high-level APIs using DSLs. However, unlike other work ViSlang focuses on the integration of multiple DSLs into one system and the support for creating new DSLs.

2.2 Libraries and Data Flow Systems

Libraries and toolkits like VTK [33], and ITK [42], provide modules that can be re-purposed and combined to form more complex volume processing and visualization systems. Libraries are a common approach to offer modular and reusable software. In this aspect, they are similar to domain-specific languages. However, a library typically offers its functionality via a low-level interface that requires advanced software engineering skills. Lefohn et al. [21] present Glift, a library of highly efficient, and reusable GPU data structures. It uses templates to offer generic data structures and algorithms similar to the Standard Template Library (STL).

The widely adopted data flow concept enables combination of existing modules in novel ways, and to extend the given functionality by integrating new modules. AVS [36], OpenDX [22], SCIRun [40], METK [27], Voreen [25], VisTrails [1], as well as the work of Rieder et al. [32] are prominent examples for data flow frameworks.

Our approach is similar to the data flow concept, but can handle more complex interactions between different modules. By offering a common interface to all modules for the user, data can not only be passed from one module to the next but also be transformed by a user-defined program. We also believe that the programming interface is more natural and less cumbersome for programing certain parts of an application. Our approach also allows for a seamless integration with the data flow paradigm and offers a novel additional interface to the user.

Other frameworks like VisIt [5] and ParaView [14] incorporate a Python scripting interface. Scripting is a powerful tool to enable higher levels of flexibility for the expert users. The ViSlang runtime inherits these advantages of scripting interfaces. Additionally, ViSlang facilitates the easy integration of novel DSLs, each potentially more focused to one particular task than a general purpose language like Python.

The Dax Toolkit [26] enables the combination of fine grained operations (*worklets*), which makes it particularly suitable for large scale computing. Unlike traditional module based data flow systems, *worklets* are combined before execution leading to less overhead and more parallelism. We employ a similar strategy and combine multiple user defined functions into one OpenCL kernel. This enables ViSlang to outperform other systems like Python and Matlab that call kernels sequentially.

2.3 Interpreted Languages for Science

The high popularity and intensive usage of interpreter systems like R [30], Matlab [35], Mathematica [41] and Python [28], shows that scientists from many domains are used to perform basic programming tasks. Recent work [12] builds on top of Python and its many libraries for high-performance scientific computing and visualization. We propose an interpreted language that targets scientific visualization, and that addresses the specific needs of domain scientists to have access to advanced visualization, and parallel processing algorithms.

```
using VolumePredicate;
predicate distanceLess[voxel vox in v] (float x, float y, float z, float dist) {
   float d = (vox.x-x)*(vox.x-x) + (vox.y-y)*(vox.y-y) + (vox.z-z)*(vox.z-z);
   return (sqrt(d) < dist);
}
predicate valueAbove[voxel vox in v] (float thresh)
{  return (vox.value>thresh); }
```

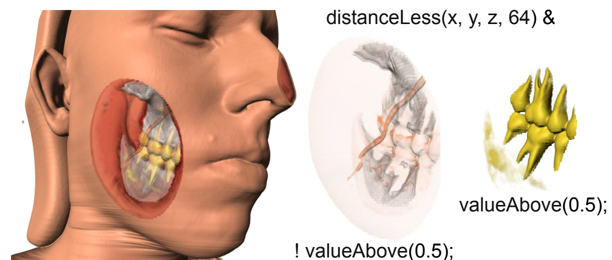

Fig. 3. The two volume predicates *distanceLess* and *valueAbove* are used to extract two regions. The *distanceLess* predicate specifies a focus region. In combination with *valueAbove* two *vsets* are extracted. The combined result is shown as well as each *vset* with separate transfer functions.

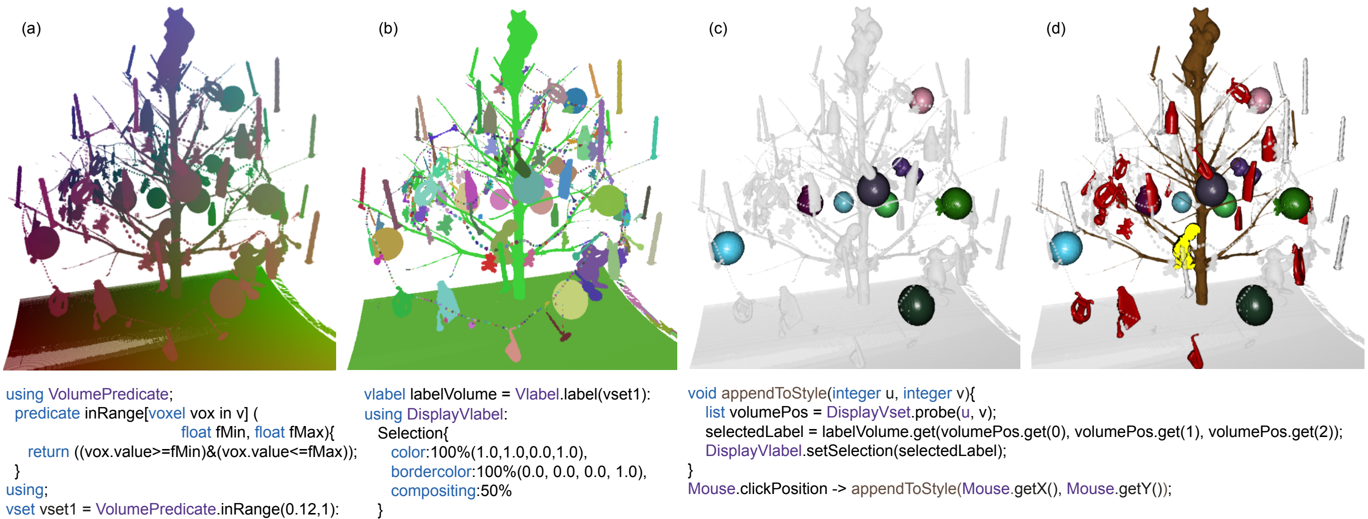

Fig. 4. ViSlang is used to program a visualization system that assigns visualization styles to different regions on mouse click. (a) shows the application of a declarative slang resulting in a segmentation of the volume. (b) shows the labeling of the individual regions and the declaration of visualization styles. (c) shows how mouse clicks trigger the assignment of visualization styles. (d) shows the result after several styles were assigned by the user.

3 Slangs

One of the main advantages of the ViSlang concept is that it enables the seamless integration of multiple programming paradigms in a common environment. For instance, for presentation and interaction, declarative approaches gained considerable popularity as they decouple specification and execution, and support retargeting. Well known examples include Cascading Style Sheets [39] (CSS), or, more recently QML [29], the Qt Modeling Language, which provides a declarative interface to the popular Qt user-interface toolkit. Within the field of visualization, ProtoVis [2] and its successor *D*3 [3] have impressively demonstrated the advantages of declarative language designs. In signal and image processing, functional approaches are of particular interest as they offer a natural and efficient representation. They can represent data flow graphs in a direct manner and provide optimization opportunities such as lazy evaluation and easy data parallelism [18]. Rendering algorithms, on the other hand, mostly lend themselves towards imperative languages due to the need for fine-grained control.

Our approach makes it possible to exploit the respective advantages of different paradigms in a single application through the combination of different slangs. In this section, we illustrate the power of ViSlang outlining three slangs that demonstrate the integration of different kinds of DSLs. The *Volume Predicate* slang described in Section 3.1 encapsulates the semantics of a logical query in subroutines. It is a *procedural* DSL for the specification of logical predicates that are executed in parallel and either yield *true* or *false* for each voxel. The *Vlabel Visualization* slang described in Section 3.2 is a *declarative* DSL for the specification of visualization styles. The *Map-Reduce* slang in Section 3.3 is a *functional* DSL. It is used to specify mapping functions that are passed as argument to one of the reduce functions.

From an end user perspective all DSLs are called from ViSlang. A slang block starts with the keyword *using* and ends with the end of input, or the *using;* instruction. The ViSlang parser forwards slang blocks to the specified slangs and waits until the commands are parsed. If the slangs report parser errors, they are presented to the user and interpretation is stopped. If parsing succeeds the execution phase starts. The commands are executed in order. When a slang instruction is reached the slang's execution interface is called and the slang takes the previously parsed commands and executes them.

3.1 Volume Predicate Slang

The *Volume Predicate* slang abstracts the parallel evaluation of a 3D logical query. Predicates are defined as subroutines using mathematical expressions and memory access functions for individual voxels. The subroutines are combined with the logical operators *or*, *and*, *not*. This slang is used to program queries that might be tailored to the application domain and the data sets under investigation. The result of a volume query is a binary volume, which is called *vset* in ViSlang.

In Figure 3, we show an example that combines multiple predicates defined by the user. At the top of Figure 3 the predicates *distanceLess* and *valueAbove* are shown. The *distanceLess* predicate is true if the Euclidean distance between the voxel *vox* and a point in space (defined with the arguments *x, y, z*) is smaller than the argument *dist*. Combining the *distanceLess* predicate with the *valueAbove* predicate results in two vsets in this example. At the bottom of Figure 3 the result and the individual vsets are shown. Each vset as well as the remainder of the volume are jointly rendered with individual transfer functions.

3.2 Vlabel Visualization Slang

The *Vlabel Visualization* slang implements a declarative language that takes data of type *volume* and a corresponding label volume (called *vlabel* in ViSlang) and performs ray-casting. It allows the user to specify different visualization styles and to assign particular labels to these styles. A style is specified using a very concise syntax controlling the weights of different colors, and other visual properties.

In the example of Figure 4, a predicate *inRange* is specified. It takes the two arguments *fMin* and *fMax* and returns *true* if the value of the voxel is in this range. Figure 4 (a) shows the predicate and the result after applying it. The predicate is transformed to an OpenCL program by the volume predicate slang. The *colon* symbol is used to call ViSlang's *display* function immediately after the statements are evaluated. As a result the *vset renderer* generates the parameter-less visualization of the *vset* by applying RGB colors to the normalized volumetric coordinates.

In Figure 4 (b) a labeling algorithm is called to assign different labels to individual disconnected regions in the vset. Again the *colon* symbol is used to call the *display* function. The default visualization of the *vlabel* assigns a random color to each region. Using the *DisplayVlabel* slang, the style *Selection* is declared at the bottom of Figure 4 (b). The declarations of other styles are similar and therefore omitted for brevity. Figure 4 (c) shows the code to specify the behavior of the system. A function *appendToStyle* is defined. The *probe* method of object *DisplayVset* is used to transform 2D image coordinates into 3D volumetric coordinates defined by the first hit of the ray. The label *selectedLabel* is retrieved as a 3D look-up in the *vlabel* data structure. The slang *DisplayVlabel* is used to assign the style to the region with

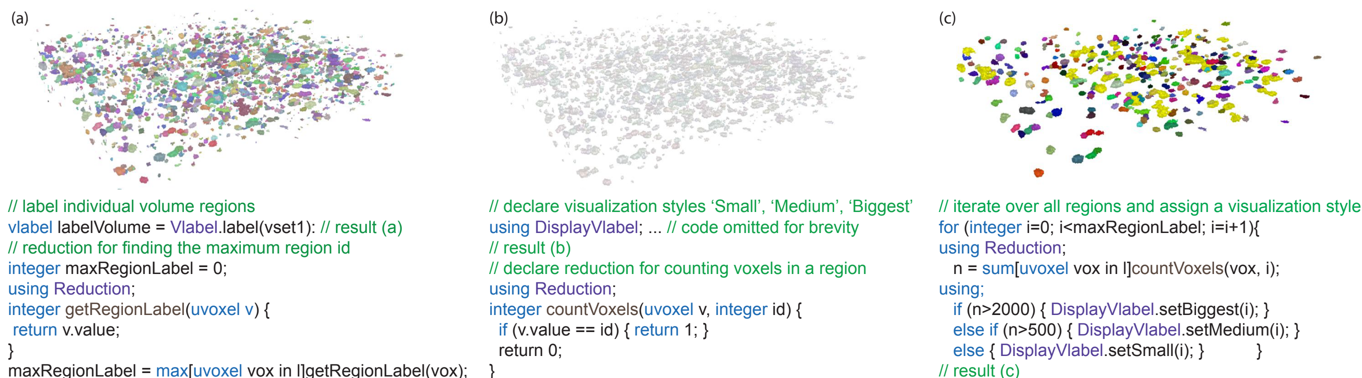

Fig. 5. The Map-Reduce slang is used to efficiently compute the number of labels as well as the number of voxels in each region. In conjunction with the other slangs a visualization is programmed that assigns different styles to regions of different size. (a) shows the result after the individual regions are labeled. (b) shows the result after the styles are defined and the background style is applied to all regions by default. (c) shows the result after all regions were assigned a style.

label *selectedLabel*. The last line of the code assigns a trigger to the variable Mouse.Position that executes the function *appendToStyle* every time the user clicks on the visualization. The image in Figure 4 (c) is the visualization after the user assigned one style to multiple regions. By slightly modifying the *appendToStyle* function, the user can assign different styles to different regions. Figure 4 (d) shows the result after the user assigned five different styles: random coloring with specular highlights for the Christmas baubles, white with shading for the candles, red with shading for the decoration, brown with shading for the trunk of the tree, and yellow cartoon shading that shines through other objects for the figure in the middle of the image.

3.3 Map-Reduce Slang

The *Map-Reduce* slang abstracts the parallel evaluation of a mapping function followed by a 3D reduction operation [20]. The user specifies a function that maps voxel attributes of (multiple) input volume(s) to one output value. The mapping function is then passed to one of the predefined reduction operations *addition*, *minimum*, *maximum* or *multiplication*. With the *Map-Reduce* slang the user can quickly implement reduction operations that are executed on the GPU. Typical algorithms that can be implemented by the user include: counting all voxels that are true in a *vset*, or finding the maximum of all voxels in a *vlabel*. More complex mapping functions can be used to query for certain value ranges, for instance, to count voxels or to compute the centroid of a connected component with a specific *id* in a *vlabel* volume. For multiple input volumes, the computation of properties of intersections and unions is possible as well as performing more complex filtering operations.

Figure 5 shows an example of using the *Volume Predicate*, the *Vlabel Visualization*, and the *Map-Reduce* slang in conjunction. A 3D energy-dispersive X-ray spectroscopy (EDS) dataset is visualized. Figure 5 (a) shows the result of applying a volume predicate and using the labeling algorithm. The Map-Reduce slang is used to find the maximum region number (*maxRegionLabel*). Figure 5 (b) shows the visualization after styles for small, medium, and large regions as well as the background are declared (the code is similar to style declarations before and therefore omitted for brevity). At this point no styles are assigned to individual regions. Therefore everything is visualized in the background style. The mapping function *countVoxels* is used in conjunction with the *sum* reduction. Figure 5 (c) shows the reduction to compute the number of voxels per region. One of three visualization styles is assigned accordingly. The smallest regions are made transparent. Medium sized regions get a random color and large regions are assigned a yellow style. This example shows how multiple slangs are used to program a visualization that is tailored to a particular EDS dataset. The accompanying video features a similar setup that generates an animation during assignment of the visualization styles.

4 Extension Mechanism and the Meta-Language

Although DSLs are extremely useful interfaces, they pose an additional burden on the application developer. Implementing a new *slang* technically is a matter of deriving a C++ class from ViSlang's slang class. However, developing a new DSL involves lexing, parsing, semantic analysis, abstract syntax tree (AST) transformations and interpretation. Additional features that are important for end users of the DSL are error reporting, debugging, syntax high-lighting, and performance optimization, just to name a few. Furthermore, DSLs for visualization are most useful when they abstract parallel algorithms. The need for a parallel execution environment introduces even more complexity for the DSL developer. In sum the additional work is often too costly and therefore DSLs are not as often employed as they would be useful.

To reduce the effort of implementing a new slang, the programmer can profit from using recurring patterns. ViSlang already incorporates a parser framework, a code generator and optimized GPU data structures that are used to manage GPU memory resources, share resources efficiently among slangs and apply optimizations. Since the ViSlang library aims to support a wide variety of visualization algorithms, we chose a set of libraries and an execution environment that are efficient, standardized, open, platform- and hardware independent, and suitable for parallel execution. The lexer and parser framework of ViSlang is *Boost Spirit* [8]. *Boost Spirit* is used to define recursive descent parsers inlined in C++. Unlike other parser generators it omits an additional build step and therefore does not depend on external tools. As a parallel execution environment we found OpenCL to be a perfect match for ViSlang. It is a just-in-time (JIT) compiled language and therefore seamlessly integrates with our interpreter. Building on this software infrastructure we have identified three common patterns for the development of new slangs in ViSlang:

Parsing: We implemented recurring syntax concepts like expressions, lists, arguments, etc. on top of *Boost Spirit*. These concepts can be reused and repurposed with very low effort. For example when instancing the *list* syntax, any parser can be used as argument. Therefore, parsing of lists of any kind is implemented quickly. Syntax constructs can be combined with the well known parser operators *sequence*, *not*, *and*, *optional*, *Kleene star*, and *plus*. Although reusing this parser framework and existing grammar elements might tremendously reduce the effort to specify a new DSL, we do by no means enforce it. Alternatively, parsers can be generated with any other popular parser generator framework. Especially grammars and parsers of existing DSLs can be integrated without additional overhead.

OpenCL code generation: ViSlang implements classes for OpenCL code generation. By using OpenCL template files with a simple annotation syntax, an OpenCL algorithm can be abstracted with low effort. At run-time, code is injected into the template and the OpenCL program is generated. A DSL that generates and just-in-

time compiles OpenCL programs, can benefit from implementing the *OpenCLAlgorithm* and *Injector* interfaces. When deriving from the *OpenCLAlgorithm* class, ViSlang handles OpenCL program compilation and offers extended debugging functionality. OpenCL template files and OpenCL programs can be inspected and modified at run-time. This greatly reduces the implementation effort for this pattern.

OpenCL data structures: By re-using the data structures that were optimized for sparse volumes, implementation effort can be greatly reduced. *struct*s are offered for the different data structures that can be included in C++ and OpenCL. *Set*- and *Get*-methods are used to transparently handle virtual memory management, address calculations, and on-the-fly memory allocation.

To even further reduce the effort of implementing a DSL, we developed a meta-language for the specification and generation of new slangs. A Slang is defined in terms of a *name*, a *template*, a set of *parameters*, a *grammar*, and a *description*. The meta-language automates some of the implementation work by instantiating templates.

```
1  using MetaSlang;
2  // defining a new slang
3  name: Filter3D;
4  // use template volume2volume
5  template: volume2volume;
6  // defining parameters
7  parameters: integer kernelWidth;
8  integer kernelHeight; integer kernelDepth;
9  // the grammar accepts a kernel function
10 grammar:
11   kernelFunction = functionName > '(' > arguments
         > ')' > '=' > expression > ';';
12   functionName = ViSlang::Identifier;
13   arguments = ViSlang::List(argument);
14   argument = ViSlang::ArgumentDeclaration;
15   expression = ViSlang::ArithmeticExpression;
16 // a human readable description for the end user
17 description: ...
```

Listing 1. Using the Meta-Language to define a new slang.

In order to avoid compromising the performance of a new DSL we create the parser and the execution code in C++ including OpenCL boilerplate code. For example the program in Listing 1 defines the new Slang *Filter3D* that is meant for the specification of a 3D filter kernel. It builds on the *volume2volume* template, meaning that OpenCL code is generated for volume processing. The *Filter3D* slang has the parameters *kernelWidth*, *kernelHeight*, anf *kernelDepth*. The end user will have access to these parameters via ViSlang. The *grammar* defines the syntax of the DSL. It consists of a list of parsers. In Listing 1 the parsers *kernelFunction*, *functionName*, *arguments*, *argument*, and *expression* are defined. The meta-language generates parsers and abstract syntax tree nodes accordingly.

```
1 // setting the parameters
2 Filter3D.kernelWidth = 5;
3 Filter3D.kernelHeight = 5;
4 Filter3D.kernelDepth = 5;
5 // using the DSL interface
6 using Filter3D;
7  f(float dx, float dy, float dz, float a, float s)=
8  a*exp(-1.0*(dx*dx)+(dy*dy)+(dz*dz)*(1.0/2*s*s));
```

Listing 2. Using the slang Filter3D to generate a 3D Gaussian filter.

In the example of Listing 1 the *kernelFunction* is defined as a sequence (>) of other parsers and a set of symbols ('(', ')', '=', and ';'). The meta-language supports the common parsers *sequence*, *Kleene star*, *option*, *logical or*, and all ViSlang parsers, the parsers that make up the ViSlang language. A slang created by the meta-language is compiled with a C++ compiler and automatically gets registered with the ViSlang system. At runtime it accepts user input according to its grammar. Listing 2 shows an example of user input that is accepted by the new Filter3D slang. The Filter3D slang parses the input, creates an abstract syntax tree and sets the parameters of the OpenCL kernel. To implement the execution phase the programmer has to transform the AST to executable code. This can be done by using the OpenCL code generation tools of ViSlang. AST nodes like function declaration, function call, arithmetic- and boolean expressions, while-, for-, if statements and literals generate OpenCL code. These features of ViSlang greatly reduce the cost of development for new DSLs.

5 ViSlang Runtime, Memory Management and Data Structures

The ViSlang runtime interprets the code by executing it sequentially. A memory manager keeps track of the allocated data structures and of the variables in scope. The variables are derived from one common class that supports linking and triggering of events. This not only allows the user to link variables and trigger events when a value changes, but also to link variables to a graphical user interface. When a new variable is declared an equivalent GUI element is created that is linked to the variable. An assignment of a new value to a variable causes a GUI update to be triggered and, likewise, manipulation of the GUI will trigger assigned ViSlang functions. The ViSlang library that includes the ViSlang runtime is largely separated from the rest of the visualization system and can be integrated with other systems written in C++. In fact our test environment and a console application that link to the ViSlang library do not make use of any visualization system, yet can be used to operate on volumes. Since the integration of rendering algorithms works differently in different systems, these parts were implemented specifically for one visualization system. Also the templates for the Meta-Language are specific to one system and need to be replaced when integrating with a different visualization system.

ViSlang implements optimized data structures for parallel execution of algorithms in OpenCL. As an example of a managed data structure we describe *vsets*, an optimized data structure for storing and manipulating sparse binary volumes. Conceptually, *vsets* are volumes with each voxel representing a boolean value. This is a common case for many algorithms that try to extract higher level semantics from the data. The most prominent example are binary segmentation algorithms, but many other algorithms also need to keep track if voxels belong to a certain class.

To benefit from spatial coherence we use a bricking scheme where each vset is conceptually subdivided into independent volumetric bricks. All homogenous bricks are represented very efficiently with only one value. Inhomogeneous bricks (with at least one entry being different from the others), are completely represented in memory, storing one bit for each voxel. The memory manager allocates memory from the graphics hardware for a pool of bricks. Each vset holds a list of indices into this memory pool. An empty vset is represented by a list of zeros and no memory is used from the brick pool. If an algorithm sets a bit in a given vset to true, the memory manager allocates an empty brick from the brick pool. This allocation is done at the run-time of the OpenCL program. The memory manager has a list of locks, to synchronize memory access to the bricks in the brick pool. Further, a list of empty bricks is held by the memory manager. A pointer into the list is required to keep track of the next empty brick to be allocated.

Figure 6 shows the brick pool of the memory manager and the allocations and deallocations during the life-cycle of several vsets. Figure 6 (a) shows the brick pool after the allocation of the three vsets orange, green, and violet. The vsets have lists that point to the brick pool. Zero as index of a vset means that the brick is empty, 1 means that each entry of the brick is set to true. The brick pool holds the memory for non-empty bricks of all allocated vsets starting with index 2. Between the memory layouts shown in Figure 6 (a) and (b), the following events happened: *vset violet* sets all bits that previously were true to false in bricks 8 and 9. Since every brick contains a counter that keeps track of the number of bits that are set to true, this situation is detected and the entries into the brick list are updated to 0. The bricks

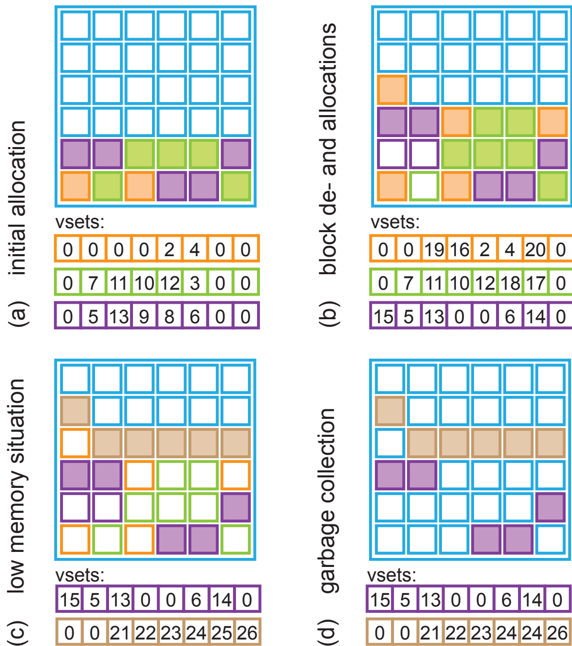

Fig. 6. The brick pool of the memory manager and the life-cycle of several vsets. Filled bricks are assigned to the vset of the corresponding color. Empty bricks are either ready to be assigned by the memory manager (blue) or used and waiting for garbage collection (other colors).

8 and 9 are marked for garbage collection. *vset violet* also sets bits to true in different bricks of the volume, and therefore new bricks (14 and 15) are allocated. *vset orange* allocates a new brick (16). *vset green* allocates brick 17, and all bits of the brick 3 of *vset green* are set to false, causing that it points to 0. Later in the same brick of *vset green*, bits are again set to true. The memory manager at this point does not know that the old brick is already free again and assigns a new brick with number 18. *vset orange* allocates new bricks (19 and 20). Three bricks of the brick pool (drawn in white) are not used, but cannot be allocated by the memory manager, because they are not in the list of empty bricks anymore. In Figure 6 (c) the green and orange vsets are deallocated. Deallocation of a vset causes the memory manager to set all bits in all used bricks of the vset to false. After the allocation of a new vset (*vset brown* allocates bricks 21 to 26) in Figure 6 (c), the memory manager detects a low memory situation (i.e., the number of free bricks is low). The garbage collector inserts the 14 bricks that are empty into the list of empty bricks. The result after garbage collection can be seen in Figure 6 (d).

In our implementation a brick size of 32^3 is used, which results in 4 KB per brick. Additionally we store one integer per brick that keeps track of the number of bits that are set to true. A vset is represented by an array of integers, each holding an index of a brick. An empty vset of size $w \times h \times d$ is efficiently represented by an array of size $ceil(w/32) \times ceil(h/32) \times ceil(d/32)$. For instance a vset of dimensions 1024^3 is represented by 128 KB of memory. A typical example of a sparse vset with 10% of bricks being non-homogenous is represented with less than 13.2 MB of memory allowing for a large number of sparse vsets to be represented on modern graphics hardware.

The memory manager has an additional overhead that depends on the number of allocated bricks. The empty brick list contains one entry per brick. To guarantee thread-safe allocation of bricks (which is critical for OpenCL algorithms), the memory manager must store a lock for each brick represented as an integer in our implementation. For a memory pool of 1 GB, this requires in total 2 MB of additional storage plus 4 bytes for the head pointer of the empty list.

Vsets are an example of an optimized data-structure that is part of the ViSlang system. A new slang can benefit from these data-structures using their interfaces regardless of their low-level implementation details.

functionality	predicate (loc)	renderer (loc)	map-reduce (loc)
ViSlang interface	112	114	161
code generation	383	0	171
parser	216	277	174
other C++	35	272	129
total C++	746	663	635
OpenCL / (%)	24 / (8%)	166 / (23%)	102 / (22%)

Table 1. Examples of implementation overhead for DSLs in lines of code (loc). The last row shows additional OpenCL code as lines of code and as percentage of the total OpenCL code of the algorithm.

Although our implementation is currently limited to single GPU, in-core data structures for regular volumetric data, this is not a general limitation of our system. With *vset*s we demonstrate the separation of low-level implementation, mid-level re-useability for new DSLs and high-level programming for end users. In the future we will follow the same implementation patterns with data structures for other use cases in visualization like flow-data and time varying data. Although, it should be rather easy to compile and run our system on other platforms, so far we have tested ViSlang on Windows only. However, to avoid complications in the future we exclusively rely on C++, OpenCL, and portable libraries like Boost.

6 Results

6.1 Overhead of Algorithm Abstraction

In this section we attempt to estimate the overhead for the integration of an algorithm with ViSlang. To quantify the overhead we classified the additional lines of code that are required to implement a DSL and to integrate it into ViSlang. Obviously the overhead depends on the functionality of the slang and the complexity of the grammar. For the slangs presented in this paper the overhead is on the order of several hundred lines of code. A classification of the lines of code according to their primary functionality is shown in Table 1. The additional C++ code is only a small fraction of the overall C++ code that is used to build the algorithms. We quantify the additional OpenCL code in absolute numbers and as a fraction of the total OpenCL code. The percentages are shown in the last row of Table 1. The C++ code is reported in absolute numbers. All examples were created without the meta-language, giving an estimate for the (less frequent but) more general case that the DSL is written from scratch. For the common case that a new DSL makes use of the meta-language the lines of code are reduced dramatically. The *ViSlang interface* and *parser* code are generated entirely and the *other C++* and *OpenCL* code can be greatly reduced.

6.2 Run-time Measurements

ViSlang strives to minimize the implementation turnaround time as well as the execution runtime for parallel algorithms. The typical turnaround time for setups without an interpreted language (like C++ with OpenCL) is on the order of tens of seconds to minutes. In this context, we define turnaround time as the total time from compiling a program to the visualization of the result. The turnaround time includes parsing, compiling, execution of the host language, just-in-time compilation of an OpenCL program, setup of kernels and memory objects, execution of the OpenCL algorithm, and visualization of the result. For instance, the typical turnaround time for the *VolumePredicate* and for the *Map-Reduce* slang is one second. In ViSlang this includes parsing and execution of the slang, OpenCL code generation and injection as well as the OpenCL just-in-time compilation. The turnaround time depends on aspects like the complexity of the algorithm, the user-defined program, the complexity of the set of parameters, and other factors. The dominant cost in ViSlang is OpenCL JIT compilation with about 90-95% of the total build time. However, a reduction of the total turnaround time to one second means a dramatic increase in productivity for the end user.

It is important to understand that just-in-time compilation only takes place on demand, i.e., when the DSL interface of a slang is used and the underlying OpenCL program has to be generated. Once the OpenCL code is generated it can be reused without recompilation. This is critical for data processing and rendering algorithms that are executed multiple times per second.

To quantify the overhead that is introduced by the abstraction layer of the algorithm, we measured the compilation and run-time for abstracted and hard-coded versions of three different algorithms. All measurements were done on a PC with 12 dual core 3.33 GHz Intel Xeon CPUs and an NVIDIA Quadro 5000 GPU.

algorithm	jit (ms)	run-time (ms)					
		spheres		christmas tree		EDS Mag	
		abst.	hard.	abst.	hard.	abst.	hard.
a) max	692	30	7	58	31	48	22
b) thresh	680	29	25	152	145	84	78
c) render	36	84	8	130	63	114	34

Table 2. Comparison of abstracted with hard-coded algorithms. Columns from left to right: 1. algorithm, 2. just in time (jit) compile time for abstracted algorithm, 3.-8. run-times for abstracted (abst.) and hard-coded (hard.) algorithms for three different data sets. All numbers are in milliseconds.

In Table 2 we provide measurements for the algorithms a) the reduction operation *max*, b) thresholding a volume, and c) the vlabel rendering. The run-time comparisons are shown for three different datasets: spheres (128x128x128), christmas tree (512x499x512), and EDS Mag (1024x987x72). Algorithms a) and b) employ dynamic generation of OpenCL code. Therefore, JIT compilation times are much higher than for algorithm c). However, the run-time measurements for algorithms a) and b) show that the overhead of abstraction is very low (4-27 ms in our experiments). For the rendering algorithm c) we used an image size of 1024x1024 and a sample distance of one. The compilation for the rendering algorithm is much shorter since it only involves parsing, AST generation and execution in ViSlang and no OpenCL recompilation is required. At runtime the more costly abstraction layer of the rendering algorithm results in significantly lower performance. Comparing these cases clearly shows that there is a trade-off between compilation time and runtime. It is important to consider this trade-off for the implementation of new slangs in ViSlang and for interpreted DSLs in general.

	Matlab	Python	ViSlang
max (GB/s)	17.22	0.17	15.71
map-reduce (GB/s)	0.08	0.03	8.40
predicate (GB/s)	1.35	0.04	3.20

Table 3. Data processing rate in GB/s (higher is better).

To quantify the runtime performance of algorithms implemented in ViSlang we compared with the two frequently used interpreted languages Matlab and Python. Table 3 shows the average data throughput in GB/s that were measured for three different operations. The operations are: maximum reduction (max), map-reduce operation: identity mapping and maximum reduction (map-reduce), and a logical predicate $(x > 0)\&true$ (predicate).

The results clearly demonstrate the benefits of the flexibility of ViSlang's approach. While Matlab slightly outperforms ViSlang for the case of a simple maximum reduction, the more general map-reduce operation is much faster in ViSlang. Matlab has optimized vector operations that include typical reduction operations. In our map-reduce experiment we take the identity function as a mapping function, which is mathematically equivalent to performing the maximum reduction only. However, Matlab cannot make use of its set of optimized operations and therefore drops sharply in performance. In ViSlang the mapping function is automatically translated to OpenCL. Therefore, it results in a comparably small performance drop when used in a map-reduce operation. We get similar results for the logical predicate. Matlab slightly outperforms ViSlang when using a standard predicate like $(x > 0)$. When applying a logically equivalent predicate $(x > 0)\&true$ optimization is omitted and ViSlang outperforms Matlab.

Although all three languages are well suited for scientific applications, ViSlang is an integrated solution for visualization with parallel data processing allowing for high performance applications, increased flexibility, low turnaround times, and high data throughput. We argue that these benefits outweigh the cost of integrating new DSLs in ViSlang.

7 Conclusions and Future Work

We have presented an interpreted language capable of including multiple DSLs. Each language can address a different aspect of the visualization pipeline or domain science. DSLs are realized as slangs of the main language, extending the functionality of the visualization system, and allow the user to program on a higher level of abstraction, oblivious of the implementation details of the underlying algorithms. This is especially beneficial for parallel algorithms. The increased implementation effort for the developer of a DSL is addressed with reusable components and the meta-language that is offered by the ViSlang system. GPU accelerated data structures are provided that are specialized for visualization algorithms resulting in high performance implementations. We showed that our approach is capable of integrating different programming paradigms. DSLs are not restricted to one syntax but are free to specify the most natural syntax for the problem domain. In our results we show that ViSlang offers short turnaround times, with interactive visual feedback, suitable for rapid prototyping of visualization applications with high runtime performance.

In the future, we want to experiment with the integration of existing DSLs like QML and $D3$ to offer support for a large body of user interaction methods and visualization algorithms. Further, we want to extend our managed data structures to deal with out-of-core algorithms. Exposing the out-of-core semantics via a DSL could benefit visualization experts that deal with the implementation of large data applications. The ViSlang system together with its collection of slangs will be made available as an open source project.

Acknowledgments

The research presented in this publication was supported by the King Abdullah University of Science and Technology (KAUST) Visual Computing Center, and the ViMaL project (FWF - Austrian Science Fund, no. P21695).

References

[1] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, and H. T. Vo. Vistrails: Enabling interactive multiple-view visualizations. In *Proceedings of IEEE Visualization 2005*, pages 135–142, 2005.

[2] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, 2009.

[3] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[4] K. Brown, A. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 89–100, 2011.

[5] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. Bethel, D. Camp, O. Rübel, M. Durant, J. Favre, and P. Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. Chapman and Hall/CRC, Oct 2012.

[6] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12)*, pages 111–120, 2012.

[7] H. Choi, W. Choi, T. Quan, D. Hildebrand, H. Pfister, and W.-K. Jeong. Vivaldi: A domain-specic language for volume processing and visualization on distributed heterogeneous systems. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Scientific Visualization / Information Visualization 2014)*, 20(12), 2014. in press.

[8] J. de Guzman, H. Kaiser, and D. Nuffer. *Spirit, Version 2.5*. http://www.boost-spirit.com/, accessed: 2014-03-30.

[9] D. Duke, R. Borgo, C. Runciman, and M. Wallace. Experience report: visualizing data through functional pipelines. *ACM SIGPLAN Notices*, 43(9):379–382, Sept. 2008.

[10] D. Duke, R. Borgo, M. Wallace, and C. Runciman. Huge data but small programs: Visualization design via multiple embedded DSLs. In A. Gill and T. Swift, editors, *Practical Aspects of Declarative Languages*, volume 5418 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2009.

[11] M. Glatter, J. Huang, S. Ahern, J. Daniel, and A. Lu. Visualizing temporal patterns in large multivariate data using textual pattern matching. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1467–1474, 2008.

[12] C. Harrison, P. Navratil, M. Moussalem, M. Jiang, and H. Childs. Efficient dynamic derived field generation on many-core architectures using python. In *Proceedings of the Workshop on Python for High Performance and Scientific Computing (PyHPC 2012)*, pages 11–20, 2012.

[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, 2011.

[14] A. Henderson. *ParaView Guide, A Parallel Visualization Application.* Kitware Inc., 2007.

[15] J. Hultquist and E. Raible. Superglue: A programming environment for scientific visualization. In *Proceedings of Visualization '92*, pages 243–250, 1992.

[16] J. Jablin, P. McCormick, and M. Herlihy. Scout: High-performance heterogeneous computing made simple. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, pages 2093–2096, 2011.

[17] C. Johnson and J. Huang. Distribution driven visualization of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 15(5):734–746, 2009.

[18] D. Johnston, M. Fleury, and A. Downton. A functional methodology for parallel image processing development. In *Proceedings of the International Conference on Visual Information Engineering 2003*, pages 266–269, 2003.

[19] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson. Simplified parallel domain traversal. In *Proceedings of the Supercomputing Conference (SC 2011)*, pages 1–11, 2011.

[20] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. Morgan Kaufmann, 2012.

[21] A. Lefohn, J. M. Kniss, R. Strzodka, S. Sengupta, and J. D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006.

[22] B. Lucas, G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe. An architecture for a scientic visualization system. In *Proceedings of Visualization '92*, page 107114, 1992.

[23] 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(10-11):648–662, 2007.

[24] P. McCormick, J. Inman, J. P. Ahrens, C. Hansen, and G. Roth. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *Proceedings of IEEE Visualization 2004*, pages 171–178, 2004.

[25] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. Hinrichs. Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations. *IEEE Computer Graphics and Applications*, 29(6):6–13, 2009.

[26] K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *Proceedings of IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV 2011)*, 2011.

[27] K. Mühler, C. Tietjen, F. Ritter, and B. Preim. The medical exploration toolkit: An efficient support for visual computing in surgical planning and training. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):133–146, 2010.

[28] Python Software Foundation. *Python Language Reference*. http://www.python.org/, accessed: 2014-03-30.

[29] Qt Project. *Qt Quick Tooling Whitepaper*. http://qt-project.org/wiki/QtQuickToolingWhitepaper, accessed: 2014-03-30.

[30] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. ISBN 3-900051-07-0, http://www.R-project.org, accessed: 2014-03-30.

[31] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4), 2012.

[32] C. Rieder, S. Palmer, F. Link, and H. K. Hahn. A shader framework for rapid prototyping of gpu-based volume rendering. *Computer Graphics Forum*, 30(3):1031–1040, 2011.

[33] W. J. Schroeder, K. Martin, and W. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, Third Edition*. Kitware, Inc., 2003.

[34] K. Stockinger, J. Shalf, W. Bethel, and K. Wu. Query-driven visualization of large data sets. In *Proceedings of IEEE Visualization 2005*, pages 167–174, 2005.

[35] The MathWorks Inc. *MATLAB*. Natick, Massachusetts. http://www.mathworks.com/, accessed: 2014-03-30.

[36] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: a computational environment for scientific visualization. *Computer Graphics and Applications, IEEE*, 9(4):30–42, 1989.

[37] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[38] H. Vo, J. Bronson, B. Summa, J. Comba, J. Freire, B. Howe, V. Pascucci, and C. Silva. Parallel visualization on large clusters using mapreduce. In *Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV 2011)*, pages 81–88, 2011.

[39] W3C. *Cascading Style Sheets*. http://www.w3.org/TR/CSS2/, accessed: 2014-03-30.

[40] D. Weinstein, S. Parker, J. Simpson, K. Zimmerman, and G. Jones. Visualization in the scirun problem-solving environment. In C. Hansen and C. Johnson, editors, *The Visualization Handbook*, pages 615–632. Elsevier, 2005.

[41] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer, Second Edition*. Addison-Wesley, 1991.

[42] T. S. Yoo, M. J. Ackerman, W. E. Lorensen, W. Schroeder, V. Chalana, S. Aylward, D. Metaxas, and R. Whitaker. Engineering and algorithm design for an image processing API: A technical report on ITK - the insight toolkit. In *Proceedings of Medicine Meets Virtual Reality*, pages 586–592, 2002.