Vivaldi: A Domain-Specific Language for Volume Processing and Visualization on Distributed Heterogeneous Systems

Hyungsuk Choi, Student Member, IEEE, Woohyuk Choi, Student Member, IEEE, Tran Minh Quan, Student Member, IEEE, David G. C. Hildebrand, Hanspeter Pfister, Senior Member, IEEE, Won-Ki Jeong, Member, IEEE

Abstract—As the size of image data from microscopes and telescopes increases, the need for high-throughput processing and visualization of large volumetric data has become more pressing. At the same time, many-core processors and GPU accelerators are commonplace, making high-performance distributed heterogeneous computing systems affordable. However, effectively utilizing GPU clusters is difficult for novice programmers, and even experienced programmers often fail to fully leverage the computing power of new parallel architectures due to their steep learning curve and programming complexity. In this paper, we present Vivaldi, a new domain-specific language for volume processing and visualization on distributed heterogeneous computing systems. Vivaldi’s Python-like grammar and parallel processing abstractions provide flexible programming tools for non-experts to easily write high-performance parallel computing code. Vivaldi provides commonly used functions and numerical operators for customized visualization and high-throughput image processing applications. We demonstrate the performance and usability of Vivaldi on several examples ranging from volume rendering to image segmentation.

Index Terms—Domain-specific language, volume rendering, GPU computing, distributed heterogeneous systems.

1 INTRODUCTION

With advent of advances in imaging technology, acquisition of large-scale volume data has become commonplace. High-performance acquisition devices and computer simulations in medicine, biology, geoscience, astrophysics, and mechanical engineering routinely produce terabytes of data per day. In the past few decades, significant research efforts have developed high-throughput volume processing and visualization techniques on parallel machines. However, most of the research efforts have been focused on improving rendering quality and speed, and only little attention has been paid to the flexibility and usability of the resulting programs and systems. The current state of volume processing and visualization falls into two categories: Off-the-shelf commercial and open source systems (e.g., [4, 20, 2]) that are monolithic and difficult to customize, and flexible software libraries and toolkits (e.g., [23, 1]) that require expert programming skills and large software development efforts.

This situation has been exacerbated through trends in parallel computing. Modern high-throughput systems are heterogeneous, e.g., combining multi-core CPUs with massively parallel GPUs via a system bus or network. These distributed heterogeneous systems have become a commodity such that even small research labs can afford small-scale computing clusters with multi-core processors and accelerators. Experts forecast that the trend to many-core heterogeneous computing will accelerate and become the de-facto standard for computer hardware [21]. This creates a growing gap between the needs of domain scientists and their expertise, which is in science and engineering, and not in distributed parallel computing. For example, using GPUs in a distributed memory system requires learning specific APIs (Application Programming Interfaces) such as MPI (Message Passing Interface), OpenCL, or CUDA, which have a steep learning curve even for computer science majors. Optimizing performance on these systems demands a deep understanding of the target architectures, care-
ful handling of complex memory and execution models, and efficient scheduling and resource management.

There have been research efforts to provide high-level programming APIs for GPU systems [29, 28] but none of them fully address the issues of flexibility and usability for visualization applications. Even recently introduced domain-specific image processing and visualization languages [7, 22, 17] do not fully support distributed heterogeneous computing platforms. To address this problem, we developed Vivaldi, a domain-specific language specifically designed for volume rendering and visualizing on distributed heterogeneous systems.

Vivaldi has been specifically designed for ease of use while providing high-throughput performance leveraging state-of-the-art distributed heterogeneous hardware. It encompasses a flexible and easy-to-use programming language, a compiler, and a runtime system. The language is similar to Python, making it easy to adopt for domain experts who can focus on application goals without having to learn parallel languages such as MPI or CUDA. The memory model provides a simple and unified view of memory without any data transfer code needed to be written by the users. Multiple copies of a data object can reside across different compute nodes while the runtime system performs synchronization automatically and transparently via lazy evaluation and dirty flags. Vivaldi provides domain-specific functions and numerical operators commonly used in visualization and scientific computing to allow users to easily write customized, high-quality volume rendering and processing applications with minimal programming effort. We envision that Vivaldi will bridge the gap between high-level volume rendering systems and low-level development toolkits while providing more flexible and easier customization for volume processing and visualization on distributed parallel computing systems.

2 RELATED WORK

Volume Rendering Systems Domain scientists often use off-the-shelf volume rendering systems such as Amira [4], commercial software for biomedical data analysis, and Osirix [20], an open-source DI- COM image viewer that supports some image processing functions. VisIt [27] and ParaView [2] are open-source systems that support large data visualizations on distributed systems. Despite the expressive power of these systems, their size and complexity is a barrier to entry for new users and domain scientists. Even though some of these systems provide limited programmability via plug-in functions, they are far from the level of flexibility that Vivaldi is targeting. In addition, those systems were developed before general-purpose GPUs like NVIDIA’s Tesla became popular, so they do not support the systems without OpenGL-enabled GPUs.

Visualization Libraries and Toolkits Open source toolkits, such as ITK [13] and VTK [23], provide comprehensive collections of visualization and image processing functions. However, these libraries have been developed for experienced programmers and are not easy for domain scientists to employ in their research code. Equalizer [10] and SPVN [9] provide libraries and frameworks for easy parallelization of OpenGL-based rendering code. However, their usage is limited to graphics tasks and lack memory and execution abstractions for computing on distributed systems. Voreen [1] is a high-level volume rendering engine that supports an intuitive dataflow editor for fast prototyping of code. However, Voreen is lacking support for distributed heterogeneous computing systems and requires users to write shader code for functionality that goes beyond the system-provided modules.

Languages for GPU Computing Extensive research efforts have been made to develop high-level computing languages and APIs for GPUs. Brook for GPUs [6] is an early language that uses GPUs as streaming processors. CUDA [19] and OpenCL [25] are low-level C-like languages for single-node systems that require explicit memory and execution management by the user. Thrust [5] and DAX toolkit [18] aim high-throughput and scalability of massively data parallel computing on many-core systems, such as GPUs, but they do not extend to distributed systems without explicit parallelization using MPI. Charm++ [15] is an object-oriented distributed parallel programming library with C++ extensions, which recently added a limited support of GPU execution, but is low-level compared to Vivaldi and does not support the GPU code translation. Muller et al. [26] introduced CUDASA, an extension of CUDA for multi-GPU systems that requires familiarity with CUDA. Zhang et al. [29] introduced GStream, a streaming API for GPU clusters that is based on C++ and provides high-level abstraction of task-parallel processing on distributed systems. However, scheduling and communication have not been optimized and users still need to write GPU code. Vo et al. [28] proposed Hyperflow, a streaming API for multi-GPU systems that is similar to GStream. Hyperflow lacks support for distributed systems and requires native GPU code. Ha et al. [11] proposed a similar streaming framework but focused mainly on optimization strategies for data communication in image processing pipelines.

Domain Specific Languages for Visualization Domain-specific language for visualization were first introduced as a graphics shading language, such as Shade Trees [8] and RenderMan [12]. Scout [16] is a domain-specific language for hardware-accelerated fixed-fuction visualization algorithms. Halide [22] is a domain-specific language for image processing that mainly focuses on generating optimal code for different computing architectures rather than providing a simpler programming environment. Recently, Diderot [7] was introduced as a domain-specific language for image processing and visualization. Diderot shares some similarities with our work, but focuses more on general mathematical operators and does not support distributed multi-GPU systems.

Our work is inspired by Shadie [17], a domain-specific language for visualization that uses Python syntax and high-level abstractions for volume rendering. As the name implies, it centers around the concept of shaders, which are short, self-contained pieces of code specifying desired visualization features. Although Shadie is easy to learn and use for novice users, its lack of general-purpose processing functions and the concept of shaders significantly impair the flexibility of the system. Moreover, Shadie does not support distributed GPU computing. Vivaldi addresses these issues and is the first domain-specific language for visualization that supports modern distributed GPU architectures.

3 LANGUAGE DESIGN

The main design goal of Vivaldi is to provide simple and intuitive programming abstractions that hide the details of the target system. We base the language on Python, which has several advantages. First, the learning curve for new users is low, even for those who do not have prior programming experience. Second, Vivaldi can import any of a large number of existing Python packages that extend its use into a broad range of applications. For example, importing the NumPy and SciPy packages makes a large collection of numerical functions available even though Vivaldi does not natively provide them. Lastly, Vivaldi users can get help for most programming questions from a wide array of Python tutorials and from the large Python user community.

3.1 Vivaldi Overview

The target architecture of Vivaldi is a cluster of heterogeneous computing nodes (i.e., with CPUs and GPUs in each node) with a distributed memory system. Vivaldi hides the architecture-specific details and provides abstractions to manage computing resources and memory, providing a unified high-level programming environment. Users do not need to know CUDA or OpenCL to use GPU acceleration because functions written in Vivaldi are automatically compiled to GPU code. Users also do not need to know MPI because communication between nodes is automatically and transparently handled by the memory manager in the Vivaldi runtime system.

Vivaldi treats each computing resource, such as CPU or GPU, as an independent execution unit as long as they are interconnected within the cluster. Each execution unit is assigned a unique execution ID and is capable of processing tasks. For example, if there are two CPU cores and two GPUs in the system, then the CPU cores are assigned IDs 0 and 1, and the GPUs are assigned IDs 2 and 3. A task is defined by a function, input and output data, and a task decomposition
The execution modifiers describe how the output values for each input element are generated in a data-parallel fashion similar to GL shaders or CUDA kernels. For example:

- **execid**: Specifies the execution ID of the unit where the function is run.
- **range**: Specifies the size of the output memory object.

Code 1 is an example of Vivaldi source code for MIP (Maximum Intensity Projection) volume rendering. `mip()` is a user-defined worker function implementing a maximum intensity projection by iteratively sampling along a ray and returning the maximum value. A volume ray casting operation can be easily implemented using a Vivaldi iterator (line 3). In this case we are using an orthogonal projection, for which `orthogonal_iter()` returns a line iterator for a ray originated from each pixel location $x,y$ and parallel to the viewing direction. The default eye location is on the positive $z$-axis looking at the origin, and the volume is centered at origin. The user can change the eye location and viewing direction by using `LookAt()` function. The location and orientation of volume can also be changed using model transformation functions, e.g., `Rotate()` and `Translate()`.

Line 14 is where `mip()` is executed with various execution modifiers. The `range` execution modifier generates a $512 \times 512$ 2D output image (i.e., a framebuffer) as a result. `dtype` is used to enforce the input volume type as short. `execid` accepts a list of execution IDs $gid$ that are generated by `get_GPU_list(4)` (in this example a list containing four GPU IDs is created). `split(result,x=2,y=2)` will split the result buffer (in this case a 2D image) by two along each axis to generate four tasks that run in parallel. In this example, the number of execution IDs and the number of tasks are same, so each GPU will handle a quarter size of the output buffer. We will now discuss task decomposition in Vivaldi in more detail.

### 3.2 Execution Model

Execution of a Vivaldi program requires data and functions. Vivaldi provides Vivaldi memory objects as a default data type to store values. A Vivaldi memory object is an n-dimensional array of the same numeric type defined on a rectilinear grid, and is compatible with a NumPy array object. Vivaldi memory objects can be created as an output of a function execution by either user-defined or Vivaldi native functions (e.g., data I/O functions).

As shown in Code 1, a Vivaldi function is defined by the `def` identifier as in Python. There are two different types of functions—the main function (def `main()`) and worker functions (e.g., def `mip()`). The main function can only be executed on a CPU node, but worker functions are compiled into different target architectures, such as CPUs or GPUs. Therefore, depending on the execution setup, the same worker function could be running on different hardware.

Vivaldi memory objects (e.g., `volume` and `result`) serve as input and output buffers for worker functions. Although arbitrarily many inputs can be used for a function there is only one output object per function execution. The function execution can be configured using various execution modifiers using the following syntax:

```
output = function(input, parameters).execution modifiers
```

The execution modifiers describe how the output values for each input element are generated in a data-parallel fashion similar to GL shaders or CUDA kernels. For example:

- **execid**: Specifies the execution ID of the unit where the function is run.
- **range**: Specifies the size of the output memory object.
worker function. A task can be processed by an execution unit assigned by the task scheduler. Tasks are generated by applying the split execution modifier to worker functions. Vivaldi provides four different types of task generation methods as shown in Fig. 3. The various parallel processing models discussed below can be implemented using only a single line of Vivaldi code.

**Input Split** The input data is split into partitions, and each partition is used to generate an output of same size. Because multiple output data are generated for the same output region there must be an additional merging step to consolidate the data. In Fig. 3, funcA is the worker function and funcB is the merging function that has been specified using the merge execution modifier. This model works well for the case when the input data size is very large and the output data size is relatively small, so splitting the input data can generate multiple tasks that fit to execution units. Sort-last parallel volume rendering is a good example of this model because the input data can be an arbitrarily large 3D volume but the size of the output data (i.e., the output image) is small and limited by the screen size.

**Output Split** The input data is duplicated for each task and each task generates a subset of the output data. Since the output from each task does not overlap with other outputs there is no merge function required. An example of this parallel model is sort-first parallel volume rendering where each task is holding the entire input data and renders only a sub-region of the screen, e.g., for parallel rendering onto a large display wall comprising multiple monitors.

**In-and-Out Split with Identical Split Partitions** Both input and output data are split into same number of identical partitions. Because each task is holding an equally sized and shaped subset of input and output data, this model applies well to data-parallel problems. For example, numerical computing using a finite difference method fits well to this model because only adjacent neighbor information is required to calculate finite difference operators. Isosurface extraction is another example that can be easily mapped to this parallel model. Since each task only needs to store a small subset of input data, this model can handle very large input if many parallel execution units are available.

**In-and-Out Split with Multiple Inputs and Different Split Partitions** Multiple input data are split into different partitions, but the combination of all the input partitions matches the split partitions of the output data. Case 4 in Fig. 3 shows the two inputs split into two pieces, each along a different direction, and the output split into four pieces. Each task stores two pieces of half the input data and a quarter of the output data. An example of this parallel model is the multiplication of two matrices, where the first input matrix is decomposed along the y axis and the second matrix is decomposed along the x axis, which matches the two-dimensional decomposition of the output matrix.

3.3.2 Task Execution

Vivaldi’s parallel task scheduler is responsible for mapping tasks to available execution units. Since execution units are connected via network in a distributed system, we must use a scheduling algorithm that minimizes the communication between different execution units. In addition, due to the heterogeneity of the execution units, we must consider memory constraints, e.g., GPU memory is usually limited in size compared to CPU memory. Therefore, we implemented a locality-aware task scheduler that reduces the memory footprint as much as possible. The scheduler manages two lists—a task queue to store tasks to be executed in the order of priority, and an idle list to keep track of the execution units that are currently idle. The scheduler executes a task by taking the top-most task (i.e., the task with the highest priority) in the task queue and assigning it to one of the idle execution units in the idle list. In this scheduling algorithm we have to consider two criteria: how to assign priorities to tasks in the task queue, and to which idle execution unit to assign the chosen task.

**Task priority based on memory footprint** We assign a task four different types of priority as follows:

- **Disk write**: The task involves disk write operations that often lead to deleting unused memory objects after writing back to disk. Since this task can reduce the memory footprint, it has the highest priority.
- **Memory copy**: The task is copying data between execution units to prepare inputs for other function executions. This task should be processed earlier so that functions can start as early as possible to reduce idling time. Therefore, it has a high priority.
- **Function execution**: Most of tasks fall into this category. The task generates a new memory object as a result and may delete some of its input memory objects if they are not referenced by the program after function execution. This task has a medium priority.
- **Disk read**: The task is reading data from disk. This task does not delete existing memory objects, so it has the lowest priority.

The goal of this task priority scheme is that tasks that may reduce the dependency to memory objects are processed earlier than other tasks so that the system can free unused memory objects as much as possible.

**Locality-aware task assignment** The scheduler assigns a task to an idle execution unit as follows:

- When a function is executed, select the execution unit that contains most of the input data.
- When split data objects are going to be merged, select the execution unit that contains most of the input data.
- When none of above apply, use round-robin assignment.

The goal of this assignment strategy is that execution units that have reusable data will be favored for function execution.
By combining these two criteria, the scheduler can reduce memory footprint, which leads to more tasks being assigned to execution units, while minimizing data communication between execution units.

### 3.4 Memory Model

One of the most difficult aspects of parallel programming on a distributed memory system is communication between nodes. MPI provides simplified APIs for message passing between processes, but designing the communication patterns, i.e., how and when to communicate, is still entirely up to the programmers and can easily become a performance bottleneck of the system. Moreover, since our target systems are heterogeneous compute clusters, users must handle multiple layers of deep memory hierarchy with varying communication latency. Therefore, data communication among heterogeneous nodes posed a big challenge in developing Vivaldi.

To address this problem, we developed a virtual memory model for distributed heterogeneous computing platforms. The core idea is that users do not need to explicitly specify the location of the data, as would be necessary for CUDA and MPI, but only need to specify where the computation takes place. The Vivaldi runtime will automatically detect whether a data copy is required or not, and performs the memory transaction accordingly in an efficient manner. Users develop the program as if it were running on a single machine and can treat each input as a single memory object at any given time, whereas multiple copies of the same volume may exist on different nodes. The optimal data location is automatically determined using *lazy-evaluation* at function execution time via execution modifiers. Data synchronization between different execution units is automatically performed behind the scene using *dirty flags*.

Fig. 4 shows an example of a basic memory transaction using the Vivaldi memory model. In line 1, memory object *a* is created in the default execution unit (one of the CPU cores) because data loaded from disk must be stored in CPU memory. Line 2 executes function *funcA*. The input to this function is *a* and the execution unit to be used for computation is 1, which is a GPU. When *funcA()* is executed, memory object *a* is copied to the GPU, and the result of *funcA()* is stored in the GPU. In line 3, *funcB()* is executed on the GPU, but the input is *b* and the output is *a*. Since *a* has been updated in the GPU, the copy of *a* in the CPU is marked as dirty, but the actual data is not copied. This is an example of lazy evaluation for synchronization to minimize data transfer. *a* in the CPU is updated with the correct copy of *a* in line 4 because *save_image()* is a disk I/O function and the input to this function must reside in CPU memory. Since *a* is dirty in the CPU, data synchronization is performed and the dirty flag is reset.

Based on these simple communication rules we can now explain the more complicated memory model for parallel computing in Vivaldi. By combining these two criteria, the scheduler can reduce memory footprint, which leads to more tasks being assigned to execution units, while minimizing data communication between execution units.

### 3.5 Domain Specific Functions

Vivaldi is specifically targeted for volume processing and visualization. We have implemented several domain-specific functions com-
monly used for numerical computation and volume rendering, and each function has been implemented for multiple execution unit architectures. We now discuss some example functions. The complete list can be found in the supplemental material.

Samplers Since input data is defined on a rectilinear grid, Vivaldi provides various memory object samplers for hardware-accelerated interpolation, for example:

- `point_query` is used to access the data value nearest to the given location (or sampled at the integer coordinates) from an n-dimensional memory object.
- `linear/cubic_query` implements fast cubic interpolation based on the technique by Sigg et al. [24]. It is often used in volume rendering when sampling at arbitrary locations for discrete data defined on an n-dimensional rectilinear grid.

Neighborhood Iterators Many numerical and image processing algorithms, e.g., finite difference operators and convolution filters, require local information. Vivaldi provides several iterator abstractions to access neighborhood values near each data location (see Fig. 6). User-defined iterators are used for iteration on line-, plane-, and cube-shaped neighborhood regions. Vivaldi’s built-in viewer-defined iterators are used for iteration on a line generated from the center of each screen pixel depending on the current projection mode, such as orthogonal- and perspective-projections. For example, `line_iter(start,end)` creates an user-defined iterator that starts from point `start` and moves a distance `step` along the line segment `(start,end)`. `plane_iter(center,radius)` creates a line iterator for a line segment defined by the intersection of a viewing ray and the 3D volume cube to be rendered (see Fig. 6). `cube_iter` is a cube iterator used to implement a 3D mean filter in Vivaldi, where `cube` is the current voxel location and `r` is the radius of the `(2r+1)3` neighborhood.

Code 2. 3D mean filter implementation using a cube iterator

```python
1. def mean_filter(vol,r,x,y,z):
2.     cube = cube_iter(c,r)
3.     sum = 0
4.     for pos in cube:
5.         val = point_query_3d(vol, pos)
6.         sum = sum + val
7.     return sum / ((2*r+1)*(2*r+1)*(2*r+1))
```

Differential Operators Vivaldi provides first and second order differential operators frequently used in numerical computing.

Fig. 6. Neighborhood iterator examples. Circles are locations where the iterator traverses. Yellow circles are user-defined (or Vivaldi system generated) anchor points to determine the traversal region.

Fig. 7. Halo communication. Black line: output buffer region. Blue line: output buffer region with input halo. Red line: output buffer region with output halo. Gray region: invalid region. Top row: input halo only. There is a halo communication because input halo becomes invalid after function execution. Bottom row: in-and-out halo. Since input halo is of size 2, there is no halo communication for two consecutive function execution.

- `linear/cubic_gradient` is a first order differential operator to compute partial derivatives in the n-dimensional Vivaldi memory object using linear / cubic interpolation that returns an n-tuple vector.
- `laplacian` is a second order differential operator to compute the divergence of the gradient (Laplace operator).

Shading Models Vivaldi provides two built-in shading models – `phong()` and `diffuse()` – to easily achieve different shading effects for surface rendering.

Halo Communicators Domain decomposition is a commonly used parallelization technique in distributed computing. Decomposition can be easily implemented using the `split` execution modifier, but there might be additional communication required across neighborhood regions depending on the computation type. For example, if differential operators are used in an iterative method, then adjacent neighbor values need to be exchanged in each iteration for correctness of the solution. Extra regions storing neighbor values are called halo, and optimal stencil code generation techniques for halo communication have been actively studied by GPU researchers [30]. Vivaldi also provides the `halo` execution modifier for automatic and implicit communication between halos.

Vivaldi supports two types of halos. An `input halo` is the extra region around the input data usually defined by the operator size. For example, if the operation uses a convolution filter with a radius `r`, then the input halo size should be `n + 1` so that the pixels at the boundary can compute the correct convoluted values (Fig. 7 top with `n = 1`). An `output halo` is the extra region around the output region usually used for running multiple iterations without halo communication. For example, if a heat equation solver runs `n` iterations, then in the `i`-th iteration we can set the input halo size to `n – i` and the output halo size to `n – i – 1`. This is assuming that `i` ranges from `0` to `n – 1`, and the Laplacian operator used in the heat equation solver requires a 1-neighborhood (i.e., directly adjacent) of values. After the first iteration the `n – 1` out of `n` halo region is still valid and can be used in the next iteration. In other words, we can read in the halo of size `n` at the beginning, and run `n` iterations of the heat equation solver without halo communication at all. Fig. 7 bottom is the case with `n = 2` and `i = 0`. This `in-and-out` halo approach can reduce the communication latency because it can reduce the number of communication calls so that the overhead of halo communicator, e.g., data preparing time, can be reduced as well. In addition, we do not need to synchronize between tasks per each iteration, so the total running time can be reduced as well. Halo communicator performance will be discussed in Section 6 in detail.
4 IMPLEMENTATION

4.1 Development Environment

Vivaldi was mainly implemented in Python with additional NumPy modules such as os, sys and time. The MPI and CUDA APIs were used for inter-node communication and GPU execution. We used CUDA version 5.5 and OpenMPI 1.7.2 which offer direct peer-to-peer communication and RDMA (Remote Direct Memory Access) for fast data communication between GPUs. To call MPI and CUDA from Python we employed the MPI4Py and PyCUDA wrapper packages. The system was developed and tested on a cluster with three nodes, one master and two compute nodes connected via a QDR Infiniband network (although Vivaldi also supports Ethernet network without GPU direct). Each compute node is equipped with eight NVIDIA Tesla M2090 GPUs with four octa-core AMD Opteron CPUs and running the CentOS 6.5 linux operating system and Python 2.6.

4.2 Runtime System

The Vivaldi runtime system runs multiple MPI processes. The main process is in charge of compilation of the input Vivaldi source code to intermediate Python and CUDA code. The main manager runs the parsed main() function line by line using Python’s exec command, creates tasks, and coordinates task execution. The memory manager, task scheduler and execution unit processes handle memory creation and deletion, task distribution, and task execution based on the priority of tasks and availability of computing resources. There is a dedicated MPI process that handles data I/O for communication between execution units and disk access. Each process is waiting for the signal from other process using MPI_ANY_SOURCE.

4.3 Translator

Even though Vivaldi has Python-like syntax, it cannot be executed directly by the Python interpreter. In addition, worker functions can run on different target platforms. Therefore, the input Vivaldi source code is translated into the intermediate code, such as Python for the CPU and CUDA for the GPU. Then Vivaldi’s runtime system executes the intermediate code using Python-based APIs, such as PyCUDA, allowing CPU and GPU code to be run seamlessly under a common Python framework.

To be more specific, there are two types of code translation in the translator box in Fig 2. First, the main function is translated to Python code by converting each worker function call with modifiers to a task using runtime helper functions, such as the task and data management functions (task_creator, VIVALDI_WRITE, VIVALDI_GATHER, etc. See the supplementary material for details). Second, Vivaldi’s worker function needs to be translated to target-specific source code, such as CUDA. We developed a translator consists of three steps. The first step is preprocessing to remove redundant white space characters and to merge or split lines to avoid ambiguity during compilation. The second step is parsing to split the code into logical blocks and to process each line in a block to infer the data type of each variable based on the operator precedence. Note that Vivaldi follows implicit data type inference as in Python, so each variable’s type must be inferred at compile time using the input data type given by the dtype execution modifier. The last step is code translation in which Vivaldi’s Python-style code is converted to CUDA code with explicit variable type declaration. A more detailed description of the translation process can be found in the supplementary material.

4.4 Memory Management

The Vivaldi memory manager handles the creation and deletion of memory objects in CPU and GPU memory. Since there is no explicit malloc or free command in Vivaldi, memory management is done implicitly based on the smart pointer technique [3] using a retain counter that keeps track of referencing of memory object and allows memory release during runtime. Whenever a function starts or stops using a memory object, its retain counter will be increased or decreased, respectively. When the retain count becomes zero, then the scheduler sends release signal to every computing unit so that unnecessary memory objects are released right away. For split case, each subset of memory object maintains its own retain counter so that partial memory object can be freed as soon as it is not used anymore.

4.5 Built-in Viewer

Vivaldi provides a viewer built on PyQt. A regular worker function can be registered to the viewer, similar to glut display function, by calling enable_viewer() to handle refreshing the screen. A manual transfer function editor is also provided, and the transfer function value can be accessed via calling transfer() in a worker function. Since Vivaldi targets GPGPU clusters, rendered image may need to be transferred to the host or node that has an OpenGL-enabled GPU. An example code using Vivaldi’s built-in viewer is shown in Fig. 8.

5 EXAMPLES

We have tested Vivaldi on several large-scale visualization and computing applications as shown below.

5.1 Distributed Parallel Volume Rendering

In this experiment, we tested two different parallel rendering schemes using split execution modifier in Vivaldi. The first example is sort-last parallel volume rendering, where input data is split and distributed across multiple execution units and the resulting images are merged at the end in order of distance from the viewer. The test dataset we used is from light sheet fluorescence microscopy (LSFM) of a juvenile zebrafish imaged with two different fluorescent color channels. The volume size is about 4.8 GB. In this example, we use the input split method to render the data using four GPUs in parallel. Fig. 8 shows the Vivaldi code and output image of this dataset.

Vivaldi’s built-in viewer is enabled by enable_viewer(), whose input parameters are a user-defined rendering function render() and a compositing function composite(). Note that render and composite functions are user-defined to allow for the implementation of different shading models and compositing rules. In the render function shown here, two transfer functions are used to assign different colors for each data channel. Since two different color and alpha values are generated per sampling we implemented a user-defined color and alpha value selection rule, where two color values are interpolated by a weighted sum using their alpha values, and the maximum among two alpha values is chosen as the final alpha value. Finally, the standard front-to-back alpha compositing is performed by calling alpha_compositing() provided by Vivaldi. Once the rendered images are generated by the render() function, one per execution unit, then the final image is generated by compositing the intermediate rendered images. The compositing rule is implemented in composite() by passing the front and back pixel values to the function automatically based on the current viewer’s location and the relative orientation of the volume. Using this function users do not need to implement a spatial search data structure (e.g., kd-tree) to resolve the order of pixels.

Sort-first parallel rendering – input data is duplicated across multiple execution units and each unit takes care of a portion of the output image – can also be easily implemented in Vivaldi using the output split method. A downsampled zebrafish dataset is rendered using four GPUs in Fig. 1 (B). Since the entire input data must be copied to each execution unit, the size of the input data for sort-first rendering can be limited compared to that of sort-last rendering.

5.2 Distributed Numerical Computation

Many numerical computing algorithms can be easily implemented and parallelized using in-and-out split and halo functions in Vivaldi. We implemented an iterative solver using a finite-difference method for 3D heat equations in Vivaldi and compared it with a C++ version to assess the performance and usability of Vivaldi. As shown in Code 3, Vivaldi version only requires 12 lines of code for a fully-functional distributed iterative heat equation solver on a GPU cluster. The equivalent C++ version, provided in the supplementary material, required at least roughly 160 lines of code (only when CUDA and MPI related
lines are counted) in addition to requiring knowledge of CUDA and MPI.

Code 3. Iterative 3D heat equation solver

```python
def heatflow(vol, x, y, z):
    a = laplacian(vol, x, y, z)
    b = point_query_3d(vol, x, y, z)
    dt = 1.0 / 6.0;
    ret = b + dt * a
    return ret

def main():
    vol = load_data_3d('data.raw', float)
    list = get_GPU_list(8)
    n = 10
    for i in range(n):
        vol = heatflow(vol, x, y, z).range(vol).execid(list)
        vol = vol.float()
```

6.3 Streaming Out-of-Core Processing

Vivaldi can process large data in a streaming fashion. We implemented a segmentation algorithm to extract cell body regions in electron microscopy brain images using Vivaldi. The segmentation processing pipeline consists of 3D image filters such as median, standard deviation, bilateral, minimum (erosion), and adaptive thresholding. The input electron microscopy dataset is about 30 GB in size.

6.1 Performance Evaluation

In order to assess the parallel runtime performance of Vivaldi, we selected three examples for benchmark testing. Since there are no similar domain specific languages to compare against, we compare against equivalent C++ code. The three benchmark program we used are:

- **Volren**: A distributed GPU volume renderer using an isosurface rendering using Phong shading model. We tested sort-first (output split) volume rendering for the input volume of size roughly 2 GB and the output image of 1920 × 1080 full HD (High-Definition) resolution. We used the step size 2 for ray marching.

- **Bilateral**: A 3D bilateral filter processing a 512 × 512 × 512 floating-point 3D volume. We implemented a bilateral filter using C++, CUDA, and MPI for the comparison. We tested a single iteration of bilateral filtering with the filter size 11 × 11 × 11. This is an example of a highly scalable algorithm because the bilateral filter is a non-iterative filter and there is no communication between nodes during filter execution.

- **Heatflow**: A 3D iterative heat flow simulation on a 512 × 512 × 1512 floating point 3D volume using a finite-difference method. Similar to the bilateral benchmark code, we implemented an iterative solver using C++, CUDA, and MPI for the comparison. In this example, we used in-and-out halo of size 10 and the total number of iteration is 50, so halo communication is performed once every 10 iterations. This example is demonstrating scalability of our system when halo communication is involved.

We ran each program using 1, 2, 4, 8, and 16 GPUs for testing scalability. The results are shown in Table 1.

Volren is a benchmark program to test Vivaldi’s parallel volume rendering performance. Although volume rendering algorithm is a scalable problem, its strong scaling performance on a distributed system was less ideal. For example, using 8 GPUs results in about 5 times speed up, which is mainly due to the communication overhead. Since our cluster system has GPGPU boards that do not have OpenGL rendering capability, Vivaldi must transfer the rendered images back to the host node per every frame to display on the screen, which is an expensive operation.

Unlike Volren, Bilateral is a strongly scalable problem since there is no communication between execution units, and Vivaldi shows a lin-
Fig. 9. Streaming out-of-core processing for cell body segmentation in a 3D electron microscopy zebrafish brain image. (a): input data, (b): median filter, (c): standard deviation, (d): bilateral filter, (e): minimum operator (erosion), (f): adaptive thresholding, (g): volume rendering of segmented cell bodies.

Table 1. Running time of benchmark programs (in seconds). The speed up factor is shown in parentheses.

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>Lines</th>
<th>GPU 1</th>
<th>GPU 2</th>
<th>GPU 4</th>
<th>GPU 8</th>
<th>GPU 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volver</td>
<td>Vivaldi Sort-First</td>
<td>33</td>
<td>2.44</td>
<td>1.37</td>
<td>0.86</td>
<td>0.47</td>
<td>0.37</td>
</tr>
<tr>
<td>Bilateral</td>
<td>Vivaldi C++</td>
<td>160~</td>
<td>94.23</td>
<td>47.22</td>
<td>23.77</td>
<td>11.85</td>
<td>7.89</td>
</tr>
<tr>
<td>Heatflow</td>
<td>Vivaldi input halo</td>
<td>12</td>
<td>11.19</td>
<td>5.84</td>
<td>3.47</td>
<td>2.09</td>
<td>1.65</td>
</tr>
<tr>
<td></td>
<td>Vivaldi in-and-out halo</td>
<td>12</td>
<td>11.17</td>
<td>5.72</td>
<td>3.02</td>
<td>1.68</td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>C++ input halo</td>
<td>160~</td>
<td>11.32</td>
<td>5.72</td>
<td>2.63</td>
<td>1.47</td>
<td>1.05</td>
</tr>
<tr>
<td></td>
<td>C++ in-and-out halo</td>
<td>160~</td>
<td>11.33</td>
<td>5.77</td>
<td>3.02</td>
<td>1.58</td>
<td>1.19</td>
</tr>
</tbody>
</table>

6.2 Limitations

Even though Vivaldi provides a unified view of the memory object for distributed processing, a gather operation from arbitrary memory locations is not well defined when the data is processed in out-of-core mode. In that case, only a subset of the entire data resides locally, and random access across arbitrary nodes can be extremely inefficient. Strictly speaking, this is not a limitation of Vivaldi but of a class of streaming and out-of-core processing problems. The solution for this problem is using pre-defined neighbor access functions, such as iterators, and halo regions such that accessing local neighborhood values and communication can be done efficiently.

The current version of Vivaldi only supports regular grids, e.g., images and volumes, because GPUs favor data-parallel processing. In the future, we plan to implement functions and iterators for processing of data on unstructured grids. The current Vivaldi is well-suited to ray-casting approaches to visualization because rasterization rendering is used for GPUs, which are not provided, although one can use OpenGL with Vivaldi (via PyOpenGL) to render polygons. In addition, the current Vivaldi is compatible only with NVIDIA GPUs due to the adaptation of PyCUDA, which can be mitigated by employing architecture-independent APIs such as OpenCL.

We observed some performance issues with Python, especially loops and memory access, because native Python is not optimized for performance. This issue could be resolved using C-based runtime engines, similar to other Python-based APIs such as SciPy [14].

7 CONCLUSIONS

In this paper, we introduced a novel domain-specific language for volume processing and visualization on modern distributed heterogeneous parallel computing systems, such as GPU clusters. The proposed language is based on Python grammar and completely hides programming details of memory management and data communication for distributed computing. Vivaldi also provides high-level language abstraction for parallel processing models commonly used in visualization and scientific computing. Therefore, users can easily harness the computing power of the state-of-the-art distributed systems in their visualization and computing tasks without much of parallel programming knowledge. We assessed the usability and performance of the proposed language and runtime system in several large-scale visualization and numerical computing examples.

In the future, we will focus more on performance optimization of task scheduling and data communication algorithms targeting large-scale distributed systems (i.e., supercomputers). Vivaldi for loosely-coupled distributed computing, i.e., grid and cloud computing, would be an interesting future work. Extending Vivaldi to other parallel architectures, such as Intel’s Many Integrated Core (MIC) architecture, AMD’s Accelerated Processing Unit (APU), and mobile GPUs, is another research direction to explore. We are also planning to support a larger class of visualization primitives other than volumes, such as vectors, polygons, graphs, etc. Vivaldi interpreter will be an useful add-on feature for quick prototyping of code, too.

ACKNOWLEDGMENTS

This work has been partially supported by the National Research Foundation of Korea grant NRF-2012R1A1A1039929 and NRF-2013K2A1A2055315, NSF grant OIA-1125087 and NVIDIA. David G. C. Hildebrand was supported by an NSF EAPSI fellowship (IIA award 1317014).
REFERENCES


