Vivaldi: a Python-like domain-specific language for volume rendering and processing on distributed multi CPU-GPU systems

Hyungsuk Choi

Department of Computer Engineering
Graduate School of UNIST

2015

Vivaldi: a Python-like domain-specific language for volume rendering and processing on distributed multi CPU-GPU systems

Hyungsuk Choi

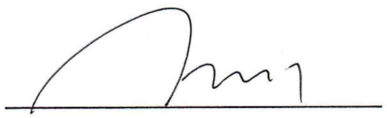
Department of Computer Engineering
Graduate School of UNIST

Vivaldi: a Python-like domain-specific language for volume rendering and processing on distributed multi CPU-GPU systems

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Hyungsuk Choi

01. 15. 2015
Approved by



Advisor
Won-Ki Jeong

Vivaldi: a Python-like domain-specific language for volume rendering and processing on distributed multi CPU-GPU systems

Hyungsuk Choi

This certifies that the thesis of Hyungsuk Choi is approved.

01. 15. 2015

Advisor: Won-Ki Jeong

Beomseok Nam: Thesis Committee Member #1

Jae-Young Sim: Thesis Committee Member #2

Abstract

In this thesis, a Python-like domain-specific language called Vivaldi is proposed. Vivaldi is based on Python, but can also provide parallel volume rendering and processing on distributed heterogeneous systems. Nowadays, visualization requires high performance for processing large data and creating high quality images. Therefore, computing systems have also advanced to meet this requirement; one example is graphics processing unit (GPU) clusters. However, even though high-performance systems exist, effective utilization of these systems is not easy for non-experts such as domain scientists and researchers because they require expert programming skills and a significant amount of software development for parallel programming, distributed systems, and heterogeneous architecture.

One aim of Vivaldi is to minimize these required programming skills using a domain-specific language for volume rendering and visualization that is Python-like and platform independent. In this language, a parallel visualization model, virtual shared memory model, and platform independent architecture for distributed heterogeneous systems are proposed. The parallel visualization model provides a simple means to implement visualization pipelines. The virtual shared memory model enables the use of cluster memory without Message Passing Interface (MPI) and CUDA. Finally, the platform independent design integrates central processing unit(CPU)s and GPUs into a common, domain-specific language. Vivaldi code was compared to C++ implementations to evaluate its performance according to number of lines, performance, and scalability. The results show that Vivaldi achieved comparable scalability and performance while requiring much less programming effort.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Visualization is a tool widely used for data from microscopes to telescopes. For example, image processing is necessary in cell analysis images that contain significant levels of noise. Volume rendering and iso-surface volume rendering is used to visualize medical data; it can help to visualize skin, bone, or other tissue transparently. Nowadays, visualization requires more computation than before. The size of the image data has increased with advances in imaging technology, and high-quality images demand more computation. Therefore, a high-performance computing system is necessary to maintain the same frame rate or computation times.

Currently, graphics processing unit (GPU) clusters have the best performance of modern computing systems, but they are difficult for novices to use. GPUs are widely used computing accelerators that have more than thousands of cores for parallel workloads. They are frequently used for accelerating visualizations. Clusters are groups of computers that have high computational power and a large amount of memory because each computer has their own computing unit and memory. A GPU cluster is a kind of cluster that has nodes with multiple GPUs, and therefore it has both large memory and high-performance computing abilities.

1.2 Thesis Statement

When users visualize data using a GPU cluster, they can choose several different methods to achieve the same result. Some methods are very easy but do not allow customization, others are difficult but allow full customization. In this thesis, they are classified into three groups: library programming, domain-specific language, and turn-key systems.

If user can program using a library, then they can customize anything in a program as desired. For example, a user could modify data transactions and process synchronization using Message Passing Interface (MPI) [1] or write their own GPU kernel and optimize it using CUDA or OpenCL. However, these

libraries have a steep learning curve and require a detailed understanding of the system. A turn-key system is the opposite of a programming library; it is a computer system that has been customized for a particular application. The name indicates that an end user can just "turn a key" and the system is ready to go, e.g., [2] or [3]. However, theses systems have some limited customization. A domain-specific language achieves a better balance between programming libraries and turn-key systems for domain-scientists and researchers who use GPU clusters. Because it is a system that is positioned between libraries and turn-key systems, it has more flexibility than libraries as well as more customization than turn-key systems. Domain-specific languages have an efficient and powerful syntax that may be better than general-purpose languages for a specific domain problem, However, at the same time, it is programmable like a programming language.

Currently, there are several domain-specific languages for visualization, but none of them support GPU clusters. Diderot [4] is a parallel domain-specific language for image analysis and visualization. It focuses on general mathematical operators and supports multi-CPUs and GPUs. Halide [5] is a new programming language designed to make it easier to write high-performance image processing code on modern machines. This language separates the algorithm from the pipeline and automatically optimizes the pipeline for high-performance. It support heterogeneous execution using CPUs plus GPUs. Both plan to support GPU clusters in the future, but neither currently support cluster back ends. Therefore, domain-scientists and researchers require a new domain-specific language for visualization using GPU clusters.

This thesis proposes the domain-specific language Vivaldi and its implementation on GPU clusters. It is simple to use because of its Python-like syntax, reduces programming effort by using many domain-specific features, and provides parallel visualization using a new syntax. In addition, the system design provides three kind of parallelism during runtime: task, data, and pipeline parallelism.

1.3 Challenges

Designing a domain-specific language based on Python has several challenges because Python is a general purpose language that works on a single CPU. First, manual data transfers require additional programming effort, therefore the language provides a virtual shared memory system that automatically synchronizes and distributes volume data from the hard disk to the CPUs and GPUs. Second, Python is not a language for

parallel visualization. Hence, Vivaldi splits the programming code into pipeline and volume filter parts. In addition, it adds abstract visualization models for parallel processing and a new syntax for using them. Additinally, it provide vector data structures, domain-specific functions, neighbourhood iterators, and halo communication methods to reduce programming effort on the part of users.

Implementing Vivaldi on a GPU cluster also has its challenges. First, different computing units have different characteristics and require different programming languages. Vivaldi solves this problem by task-parallel processing and execution units. Each task contains high-level information independent of the actual device. An execution unit contains the implementation for a specific device. Hence, Vivaldi can manage a heterogeneous system. Next, scheduling on distributed heterogeneous systems can lead to large performance differences. Vivaldi has a priority-based task scheduling which manage data transaction and task execution together. In addition, remote direct memory access for GPUs has priority over CPU memory transfers.

1.4 Contributions

The first contribution of this thesis is the first domain-specific language for volume rendering and processing on a GPU cluster. The second contribution is a system design and implementation that can be adapted for other accelerators and has room for optimization in the task-parallel processing and execution unit that inlcudes the translator and compiler.

2 Related Work

Volume ray casting [6] is a an image-based volume rendering technique. It creates 2D images from a 3D volume. This technique comprises four steps: ray casting, sampling, shading, and compositing. In the ray casting step, a ray is cast for each pixel and sampling is done for each step of the ray casting. The parallel execution model of Vivaldi is well suited to a ray casting algorithm. A user writes the volume ray casting algorithm for one pixel, and specifies the pixels for which they want to execute the algorithm in the pipeline. In addition, this design is well suited to image processing algorithms that are executed pixel-by-pixel. The volume ray casting algorithm requires a significant amount of computation; therefore, GPUs are widely used to accelerate it. Commonly, the use of a GPU acheives a 100× performance increase. However, it is becoming harder to provide real time volume rendering.

To support real time volume rendering, researchers have approached the problem from several different angles. Some researchers have studied fast volume rendering, while others have attempted to balance image error and performance. *Ambient volume scattering* [7] is one example of fast volume rendering. It comprises a fast ray-casting algorithm with pre-integration and mesoscopic scale direct volume rendering such that the step size is much bigger than usual ray casting. As a result, the authors achieved high-quality ambient volume scattering with performance almost as good as simple volumetric ambient occlusion. Other groups have attempted to balance image error and performance to achieve real time volume rendering. *Interactive progressive visualization with space-time error control* [8] is a proposed method for balancing fixed frame rate volume rendering and fixed image quality volume rendering. In detail, the method renders a image with automatic parameters, evaluates the image, changes the parameters according to this evaluation, and renders it again. For example, it spends more or less time rendering if the image quality or frame rate, respectively, degrades.

This thesis approaches the performance problem from a different angle. Vivaldi provides an easy way to utilize high-performance GPU clusters using just few variable changes. For example, in an optimal case, 16 GPUs can handle 16 times more memory, and a 2 fps program can be converted to real-time (32 fps) just by changing the GPU device to a GPU cluster. In addition, a user can also use fast algorithms with Vivaldi.

Each domain-specific language has a different front end. Halide [5] is a one example of a DSL. It has a C++ based front end and decouples the pipeline and algorithm. In this language, the user only defines the computation they want in the front end, then the runtime system automatically determines what to store and the order of computation. This is because, as they state, optimized programming code can achieve up to a 70× performance gain using the same CPU. Vivaldi has a translation similar to Halide; however, it does not support programming code optimization during the translation step. Another difference is that in Vivaldi, how to split and parallelize can be specified in the programming code, but in Halide this is automatically decided. The difference comes from the different goals of the two languages. Vivaldi is a language for research that uses high-performance systems, hence it allows the user to specify parallel execution, the execution schedule is determined during runtime, and the experiment environments are frequently changed. However, Halide is for writing high-performance code on modern systems, therefore the program execution is automatically optimized and tuned, needing two hours to two days for specific devices and types of data. However, it creates an executable file as a result.

Another example is Shadie [9], a domain-specific language for volume rendering. It has a Python-like front end for writing volume filters, additional syntax for volume loading, and a graphic user interface(GUI) for mouse interaction. The Python-like syntax is easy for novice users, and the GUI also intuitive, but it only provide a rendering of a volume and does not provide the ability to write visualization pipelines. Vivaldi is inspired by Shadie, specifically, its set of domain-specific functions and Python-like front end. However, the front-end style and domain-specific functions sets are extended in Vivaldi to support high-performance environments, new functions are added, it is more memory efficient, and additional domain-specific functions are added called neighbourhood iterators for easier visualization. Vivaldi has a more Python-like front end for writing volume filters, and also it has the ability to write pipelines, as one goal of Vivaldi is to support writing pipelines (the connection of multiple volume filters). Another difference is that Shadie supports only one GPU. In contrast, Vivaldi supports GPU clusters. Because volume rendering requires a significant amount of computation, a result cannot be created using only one GPU in real time. Therefore, Vivaldi has an increased number of GPUs that it can control for real time volume rendering.

The back end of Vivaldi is designed for heterogeneous computing. Hyperflow [10] is a paper that

proposes a system architecture for multi-CPU-GPU systems. In this system, data is divided into small blocks. CPUs process a small number of blocks and GPUs process a larger number of blocks according to their performance difference. As a result, this system performs better than homogeneous systems such as systems of only CPUs or only GPUs. The system uses data and task parallelism as a default scheduling strategy. In addition, they provide pipeline parallelism as data streaming with a visualization pipeline. Small data blocks go through the pipeline, and the next block starts just after previous block leaves. The Vivaldi back end system is based on Hyperflow. The common features with the Hyperflow is that Vivaldi has processes that is in charge of a device, task parallelism and the scheduler are used to split data into small pieces for parallel execution. Therefore, the Vivaldi system can support heterogeneous computing as well as three kinds of parallelism. However, Vivaldi and Hyperflow have several differences. First, they have different target systems. The multi-CPU-GPU system in [10] is a laptop with two GPUs. In contrast, the target system of Vivaldi is a distributed GPU cluster. Therefore, the Vivaldi back end is remodelled based on MPI processes. Furthermore, data is actually transferred between processes because of a lack of shared memory. The second difference comes from the different types of program. Hyperflow is a dataflow architecture, hence it simply maps executable objects to the user task as a CPU version or GPU version. In contrast, Vivaldi translates user code to multiple executable objects because it is a domain-specific language.

3 Language Design

Vivaldi is based on Python because of its advantage of an easy-to-learn syntax. This advantage comes from its compatibility to existing Python modules such as NumPy and SciPy. However, Vivaldi has some new syntax and restrictions to support distributed heterogeneous systems using a domain-specific language. For example, it allows only a subset of the Python syntax for easy translation and is executed in *data-parallel fashion* for CUDA compatibility.

Vivaldi code consists of two parts, the main function and worker functions. The main function is used to specify the visualization pipeline, including file I/O from the hard disk. The worker functions are volume filters that are executed in the actual computation devices. The programming code of Figure 1 is very similar

```
1  def mip(volume,x,y):
2    step = 1.0
3    line_iter = orthogonal_iter(volume, x, y, step)
4    max = 0
5    for point in line_iter:
6      I = linear_query_3d(volume, point)
7      if max < I: max = I
8    return max
9
10 def main():
11   volume = load_data_3d(DATA_PATH+'/CThead.dat')
12   result = mip(volume,x,y).range(x=0:512,y=0:512)
13                            .split(result,x=2,y=2)
14   save_image(result,'mip.png')
```

Figure 1: Vivaldi version of parallel maximum intensity projection volume rendering using four GPUs on a GPU cluster.

to Python. For example, the function definition uses the *def* syntax, there are no data type declarations, the *if* and *for* statements are the same, and it uses tab-based indent checking. However, in lines 12–13, there is new syntax for parallel processing called execution modifiers: *range* and *split*. For instance, suppose we want to create a 512 × 512 result image using the MIP function. We use the *output split model* that is provided for parallel processing. Four processes then execute the MIP function 256 × 256 times for the assigned range and generate a divided output buffer. In line 14, the *save image* function requests the result to be output as "mip.png." The divided buffers are then merged, sent to the I/O dedicated process, and written to hard disk.

3.1 Main Function

Vivaldi provide a new syntax called *execution modifiers* that configure the parallel execution of the worker function. The usage of execution modifiers is as follows:

Output = worker function(input, parameters).execution modifier

An execution modifier is located after a worker function or other execution modifier. Vivaldi cannot parallelize ordinary Python functions using other functions or built-in functions. Some examples of execution modifiers are as follows:

- execid: specifies the execution ID of the execution unit where the function is run
- range: specifies the size of the output memory object
- split: specifies parallel execution by splitting the input and/or output memory object
- merge: specifies a user defined merging function

Execution modifiers are translated before execution because they are not a syntax in the original Python (note that there are another reasons related to virtual memory model implementation that are explained later). The execution modifiers are converted to built-in task generators. The task generator creates multiple tasks using worker functions and execution modifiers and registers the tasks at the scheduler. Therefore, the program progresses until the line that waits for external dependencies to finish. The scheduler then optimizes the pipeline. For example, a function without sink is not executed, even if there are functions in the main function. In same way, data without sink is also not allocated during runtime.

3.2 Worker Functions

All user-written functions except the main function are worker functions and are volume filters applied to every voxel. The advantage of this design is that if a user writes a volume filter, then the user can customize the execution range and parallelization method without changing the volume filter.

In a worker function, the user can write a custom volume filter using Python-like syntax. However, there must be some restrictions to ensure platform independent execution because some syntax and functions do not exist in different devices. Hence, only commonly executable functions are implemented. The first restriction is that execution must be in data-parallel fashion. This means that the result of a nearby thread cannot be used for the current computation. Second, a user cannot use Python modules in the worker function because not every programming language has Python modules. For example, CUDA does not have NumPy, so it will generate a compile error. However, if the translator developer prepares a conversion from NumPy to CUDA functions, then the user can use NumPy in the worker function. In addition, some short cuts for *if* and *for* statements are not currently supported.

3.3 Parallel Processing Model

Vivaldi provides parallel programming models for easy parallel processing. The user selects a parallel processing model using a split modifier, then the runtime system creates multiple tasks using the worker function, input/output data, and execution modifiers. Than scheduler maps task and execution units.

Input split model: This model splits the input data into small pieces and each piece creates an intermediate output. The intermediate output is combined by the merge function that is defined by the merge execution modifier and a user-written merge function. This model is well suited to implementing sort-last rendering because it splits the input data and merges intermediate output using alpha compositing. This model splits the input data and the user writes the alpha compositing him/herself.

Output split model: This model splits output data into small pieces. It distributes the same input data and each process creates part of the output. This model easily achieves linear scalability if the load is equally distributed because multiple processes can participate in a computation and each process only their area. Sort-last (screen decomposition) volume rendering is a good example of the output split model.

In-and-out split model with identical split shape: This model splits input and output data into the same shape. In this model, the divided input/output data and computation range are exactly matched. However, using larger input data is allowed. This model is well suited for numerical computing using

a finite difference method because only adjacent neighbour information is required for computation. In addition, Vivaldi provides related functions like gradient, interpolations, and neighbourhood iterators. This model can handle out-of-core data because it splits input data into small pieces and the runtime system releases already-used data in real time.

In-and-out split model with different split shape: This model is another case of the in-and-out split model. The combination of split input data is the same as the number of output splits, but the shape of the split is different. The simplest example of this model is matrix multiplication. Each task uses a pair of partial input data to generate a partial output.

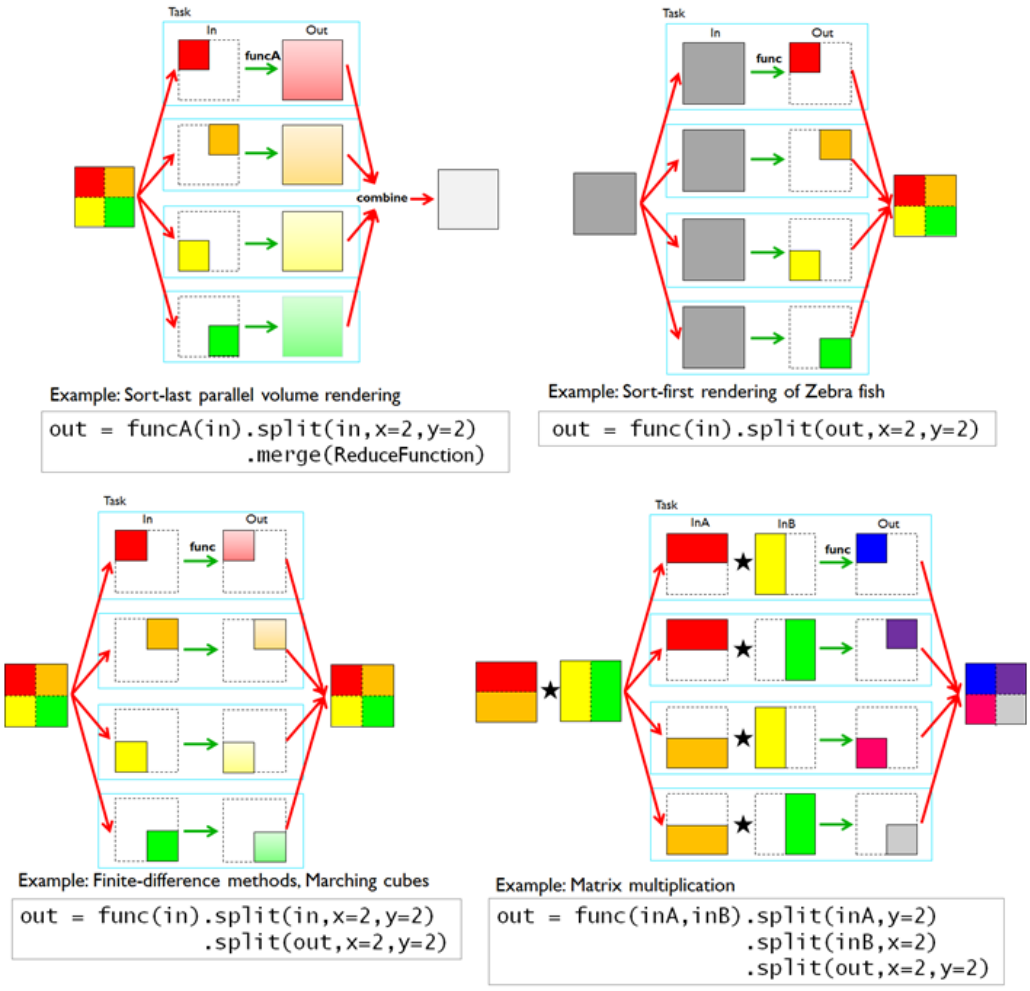


Figure 2: Parallel processing models are specified by how the data is split. They are well suited for visualization pipelines.

3.4 Domain-Specific Functions

Vivaldi provides domain-specific functions and data structures for easy visualization and processing. They are implemented for each execution unit.

Sampler: Sampler has two main purposes: it provides commonly used interpolations and access to divided data that has a different buffer shape to the original data. Therefore, the user has to use the *...query...* function to access the correct locations of the data instead of directly accessing using a bracket.

- *point/linear/cubic_query_nd()* return the value of a given location from an n-dimensional volume. Point returns the value of the nearest voxel, linear returns the linear interpolation result, and cubic returns the cubic interpolation result. Currently, these functions are implemented up to three dimensions.

Neighbourhood Iterators: Neighbourhood values are widely used for image processing and numerical

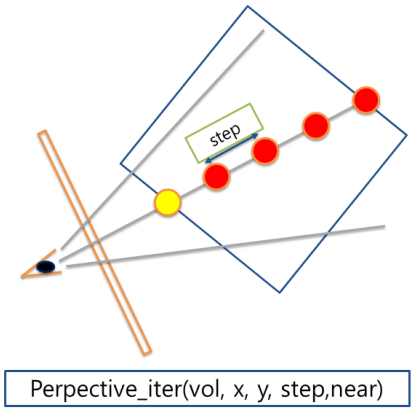


Figure 3: Example of a perspective iterator that automatically calculates volume intersection points and travels inside the volume. The yellow and red points indicate the start and internal points, respectively.

algorithms such as finite difference operators and convolution filters. Vivaldi provides some iterators related to data shapes such as lines, planes, and cubes. In addition, it also provides iterators for visualization such as orthogonal and perspective projection functions.

```
1 def mean_filter(vol,r,x,y,z):
2   c = make_float3(x,y,z)
3   cubeItr = cube_iter(c,r)
4   sum = 0
5   for pos in cubeIter:
6     val = point_query_3d(vol, pos)
7     sum += val
8   sum /= powf(2*r+1,3)
9   return sum
```

Figure 4: Example of a 3D mean filter using a cube iterator.

Figure 4 shows an example of a 3D mean filter implementation using a neighborhood iterator. The iterator *cube_iter* requires the center point and radius as inputs and returns an iterator that traverses a cube shape. This iterator is used in the for statement in Line 5.

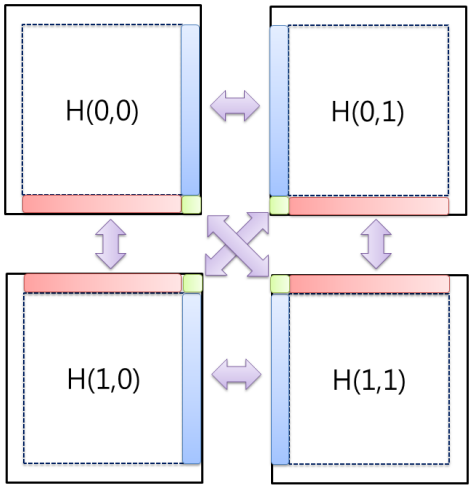
Differential Operators: Vivaldi provides first and second differential operators.

- *linearcubic_gradient_nD()* are first order gradient operators. "Linear" and "cubic" indicate the type of interpolation for sampling.
- *laplacian()* is the second order differential operator that computes the divergence of the gradient.

Shading Model: Vivaldi provides two built-in shading models: *phong()* and *diffuse()*.

Halo Communication: Vivaldi provides an automatic halo communication method using the **halo** modifier. Halo is commonly used for data decomposition algorithms. For example, sort-last volume rendering with interpolation requires halo for smooth boundaries. Convolutions and numerical computations also require halo values for correct results. However, halo communication requires a large amount of programming.

The user can split the data using the **split** modifier and then specify the halo using the **halo** modifier. The virtual memory system then automatically creates a halo for the data. Figure 5 is an example of halo communication. In the first iteration, the runtime system creates data with a halo of size one. After function execution, a halo of size zero is created because that is the default halo size. Before the start of the next iteration, the runtime system automatically communicates the halo and starts the next step.

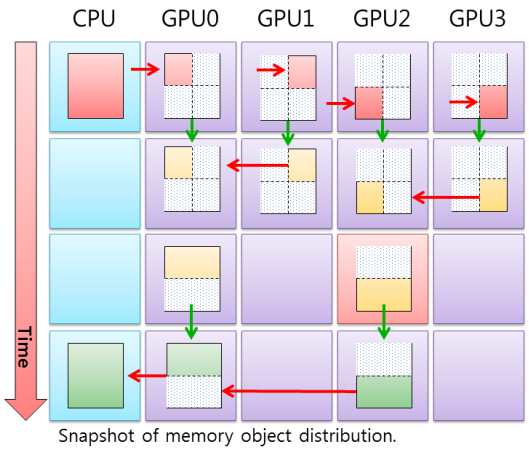



```
1 for i in range(n):
2   H = func(H,..).split(H,...)
3                  .halo(H,1)
4                  ...
```

Figure 5: Example of iterative halo communication. If the input halo size is one, and output halo size is zero, then halo communication occurs at every iteration.

3.5 Virtual Shared Memory Model

The virtual shared memory model is an important model to provide in a Python-like environment for distributed heterogeneous systems. It automatically synchronizes and distributes data among entire computing devices. It uses a NumPy array for compatibility with ordinary Python. In addition, it uses lazy evaluation, locality awareness, and "retain count" for efficient memory management. Lazy evaluation determines the data location as late as possible to minimize memory allocation. Locality awareness minimizes the amount of data transfer because the network can easily become a performance bottleneck. Retain count provides runtime memory release to provide out-of-core processing.



Snapshot of memory object distribution.

```
1 a = load_data_3d(..)
2 b = AtoB(a).split(a,x=2,y=2)
3             .split(b,x=2,y=2)
4             .execid(1,2,3,4)
5 c = BtoC(b).split(b,y=2)
6             .split(c,y=2)
7             .execid(1,2)
8 print c
```

Figure 6: Vivaldi memory objects are transparently shared by the CPU and GPU. This model automatically synchronizes and distributes the object. In addition, this memory object is compatible with a NumPy array. Hence, the user can use the object like an ordinary NumPy array without MPI programming. The red arrows indicates data transactions between processes, and the green arrows indicate data creation through function execution. The diagram shows data communication as time progresses.

In Figure 6, the listing shows an example of distributed computing using virtual shared memory model. In line 1, 3D data is loaded and assigned to "a." In line 2, function AtoB attempts to split "a," which was one

unit in the CPU but it is scattered by the runtime system by lazy evaluation. Next, the AtoB function creates data “b.” In line 3, BtoC attempts to halve the size of “b,” then the runtime system gathers “b” to match the input size. The final line attempts to print the data “c,” but it is distributed in GPU0 and GPU2. Hence, the runtime system stops the progress of the main function and merges “c” on GPU0 because merging data on a GPU is faster than on the CPU. It is then copied to the CPU in NumPy array form. Therefore, the user can print distributed data without MPI programming.

Lazy evaluation: This is a technique that minimizes memory usage by allocating memory as late as possible. This allows the memory to be used for other purpose until it is allocated. In addition, the total memory size may not be large enough to load all the data that is used during execution. For example, assume there is a GPU with 5 GB memory, and three chunks of 2 GB memory are necessary. The system cannot load every datum because the total memory size is smaller than 6 GB. However, if the system loads 2 GB data, processes it, and releases it, then Vivaldi can compute this data. In this way, Vivaldi can handle out-of-core data.

Locality awareness: The system considers locality to minimize data transfer because the network is much slower than computation and it can easily become a bottleneck. For example, when the scheduler assigns a function task to an execution unit, there are no guarantees that all necessary volumes are in the execution unit. In that case, the system needs to gather the volumes into an execution unit. At this point, the system considers how much data will be transferred before the actual transfer. This is possible because the scheduler manages a global data lookup table as well as the information about the data necessary for the task.

Retain count: This count is used for runtime memory release, similarly to garbage collection. It records how many tasks require the data, then releases that data if the retain count is zero. Every subvolume in the system has a retain count to allow for partial memory allocation and release that is especially important for out-of-core rendering. For example, assume there are 4 GB of data and memory size is 2 GB. Vivaldi then splits the data into two pieces. Next, the scheduler assigns an ID of 1 to the first subvolume and ID 2 to the second subvolume. It then loads the data with ID 1, processes it, and releases it. Next, Vivaldi loads the data with ID 2, processes it, and releases it. In this way, Vivaldi can compute out-of-core data that is larger

than the total memory size.

4 System Design

The Vivaldi system uses task-parallel processing to enable it to use heterogeneous systems. In this model, there are four types of tasks: data read, memory copy, function, and data write. The tasks are generated in the main function, managed by the scheduler, and mapped to processes in Vivaldi.

4.1 Vivaldi Architecture

4.1.1 Execution Unit

The heterogeneous devices are managed by an execution unit that is in charge of actual devices such as CPUs and GPUs. It manages everything related to the device such as data transfer, memory allocation, memory release, device dependent optimization, and function execution. This process always waits for the next task, and then executes it on an actual device. Therefore, the process has different implementations for each device because an implementation for a CPU cannot manage a GPU. Currently, it uses Python for the CPU implementation and PyCUDA for the GPU implementation.

All execution units can execute a worker function without an executable object because they have a built-in translator and compiler to execute the worker function. When the task arrives, the execution unit translates and compiles it in just-in-time. The compiled context remains until the function is no longer necessary, hence the compilation overhead only occurs once per device. In this manner, there are other execution units for other devices, and to these devices, Vivaldi can provide tasks.

Data transfer between different types of execution units is always possible because the units have access to CPU memory, regardless of their implementation. For example, a GPU execution unit can transfer data from the GPU to CPU memory for transfer to the CPU execution unit. In addition, Vivaldi has accelerated data transfer between GPUs, called GPU direct memory access, implemented using OpenMPI and PyCUDA.

4.1.2 Data Reader

The data reader is an I/O dedicated process and can handle three types of task: data read, data write, and memory copy. These types of executable tasks are restricted because networks can easily become bottlenecks of the entire system. Only this process has access to the hard disk, hence, all reading and writing occur in this process. For example, if an execution unit finishes a computation, the resulting data is transferred to the data reader. In addition, it has functions to partially read big data, hence Vivaldi can provide out-of-core rendering through the data reader.

4.1.3 Scheduler

The scheduler uses a priority queue for task management that is modified for high utilization. The priority of the task is determined by the task type. Data write has the highest priority, because this frees memory for further data. Memory copy has second priority, and the priority between memory copy and function execution is determined using heuristics for system performance. Data read has the lowest priority because Vivaldi often needs to handle out-of-core data, but the physical memory is not large enough to load it. Usually, a priority queue does not execute low-priority tasks if there are unexecuted high-priority tasks. However, in this model, the scheduler can execute low-priority tasks if the high-priority tasks are not executable for some reason, such as dependency issues, to achieve high-utilization.

In addition to tasks, all MPI user functions are executed by the scheduler. For example, the synchronization function asks the scheduler if all tasks are finished and waits for the answer. Another case is related to dependency, and if dependencies are found, then the main process sends data request to the scheduler and waits the data arrive.

Figure 7 shows the connection of the modules. The main process generates tasks and registers them with the scheduler. The scheduler maps the tasks to execution units, and the execution units run their tasks. The data reader provides input data from the hard disk and also writes result data.

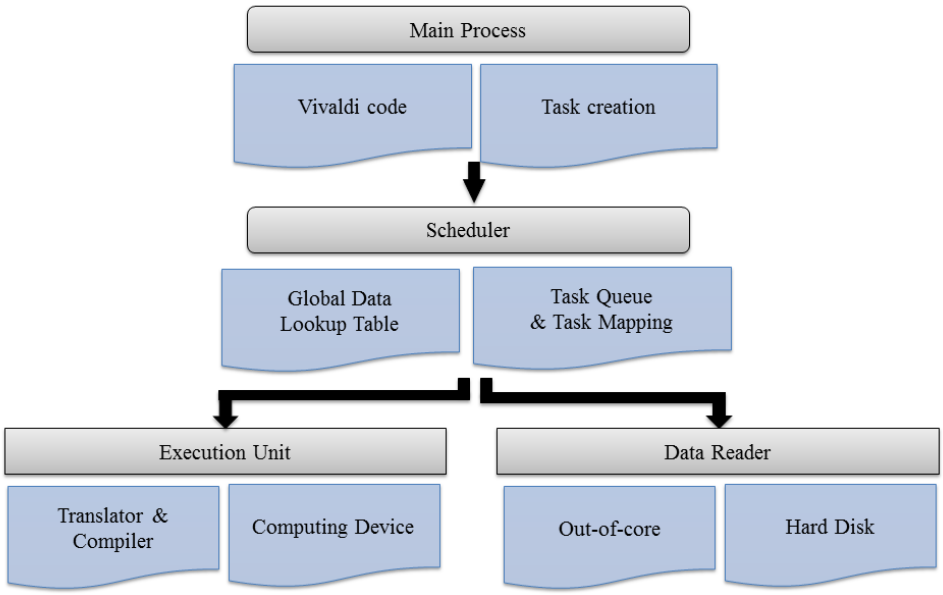


Figure 7: Overview of Vivaldi system architecture.

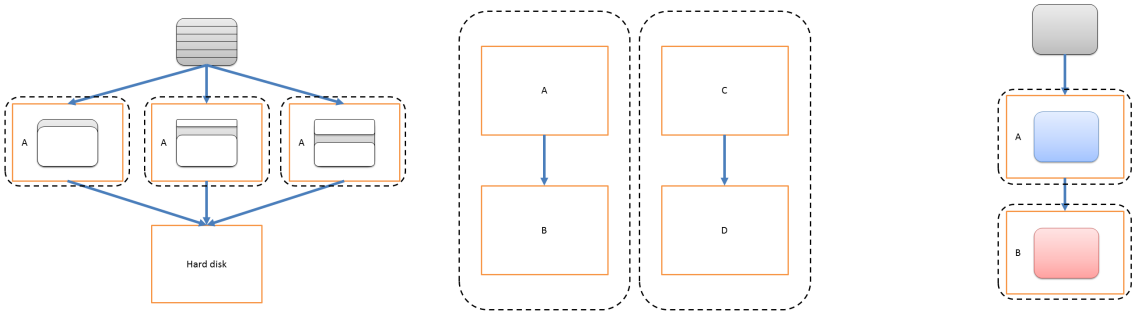
4.2 Parallelism

Vivaldi system provides three kinds of parallelism: task, data, and pipeline. The runtime system of Vivaldi works in a task-parallel fashion. However, the execution modifier, virtual shared memory system, and non-blocking execution of the task generator can provide data and pipeline parallelism as well.

Data parallelism can be supported by the **split** modifier that splits data into several pieces. The divided data can be processed separately by multiple execution units. Figure 8a shows how data parallelism works. Each execution unit processes a different part of the data using the same function "A." In this way, out-of-core processing can be managed.

Task parallelism is the default mode of the system if there are no dependencies between functions. In Figure 8b, there are A to B and C to D dependencies. Therefore, A and C lines can be run concurrently.

Pipeline parallelism is available using the **execid** modifier. The user can specify where a function can be executed. Therefore, if the user assigns function A to GPU1 and function B to GPU2, the scheduler will map the tasks accordingly and the memory system automatically transfers the data.



(a) Data parallelism (b) Task parallelism (c) Pipeline parallelism

Figure 8: Example of Vivaldi parallelism. The dotted boundary lines indicate concurrent execution. The arrows indicate dependency between functions.

5 Implementation

5.1 Data Management

5.1.1 Data_package

The data_package is a set of information about a piece of data. However it can be created without real data, so it is possible to asynchronously execute the main function. The scheduler creates it per each subdata, and the system partially allocates and release part of the data. This characteristic is used for out-of-core rendering and processing. The scheduler uses it instead of the actual contents of the buffer for data management.

```
class Data_package():
  def __init__(self):
    # data id
    self.unique_id
    self.split_shape
    self.split_position
    . . .

    # data information
    self.data_range
    self.data_halo
    self.data_dtype
    self.data_bytes
    . . .

    # full data information
    self.full_data_range
    . . .

    # file
    self.file_name
    self.extension
    . . .
# etc functions
. . .
```

Figure 9: Data package class that includes information about real data.

There are four types of information in the data package: the ID that indicates the actual data, the buffer divisions, information about the overall data, and information about the file on the hard disk.

5.1.2 Retain Count

The retain count is mapped to each data package ID. The retain counts are managed in the scheduler as a retain count dictionary because every event in the system is reported to the scheduler, hence the scheduler can instantaneously change the retain count value. For example, when a task finishes, then the execution unit reports that it no longer needs the data, and the scheduler decreases the retain count of the data.

5.1.3 Data Creation

When a task requests non-existing data, the runtime system creates it using existing data. For example, a half-size of the data can be created by two quarters of the data. This is implemented using a dictionary called the "remaining writing count," where the key is data ID and the value is how many writes are necessary to complete the data buffer. This value is always one when creating smaller data from larger data. The value is greater than two when creating bigger data. The remaining write count decreases every time a write is reported to the scheduler.

```
1  tid = .. # ID of the data content
2  tshape = .. # Split shape of the data
3  tposition = .. # Data position of divided data
4
5  R = retain_count_list
6  for split_shape in R[tid]:
7    # Find minimum set of buffer with split shape to create the target shape
8    sub_data_list = get_sub_buffers_list(tid, split_shape, tposition)
9    if sub_data_list == None: continue
10   # Check the sub data set exists
11   flag = check_exist(sub_data_list)
12   if flag == None: continue
13   # Increase retain\_count of sub buffer
14   increase_retain_count(sub_data_list)
15   remaining_writing_count(tid,tshape,tposition)
16   return
17
18 # There is no way to create the data
19 # Invalid state
20 assert(False)
```

Figure 10: Algorithm for creating required data using existing data.

Figure10 is an algorithm for data creation. Lines 1 to 3 are the data information that we want to create. Line 5, retain_count_list, has information about the existing data. Line 6 iterates the existing data shapes.

Lines 7 to 12 check that all necessary data exist. The program progresses to the next line if the data set is available. Otherwise, it returns to the start of iteration if the data set is not available. In lines 14 to 15, if the data set is defined, then we increase the retain count of the source data and increase the remaining write count of the target buffer.

5.1.4 Data Access

Memory access necessary domain-specific functions are called xxx_query instead of array[]. The query functions use full_data_range and buffer_data_range from data_package for correct data access. Assume that there is a one dimensional data set of size 10, and it is divided into two pieces. Then, the full_data_range is 0 to 10 and buffer_data_range is divided to 0 to 5 and 6 to 10 for each data. Hence, if the user is trying to access the right buffer using point_query(,x=8,), then the function changes to data[8-buffer_data_range.x_start].

5.2 Task

Tasks have a high level of abstraction and describe the behaviour of processes because different devices require different implementations. In Vivaldi, tasks are saved using two types of class: function_package and memcpy_package.

5.2.1 Function_package

The function_package contains information about a worker function execution. It is created by the task generator in the main function and scheduled in the scheduler.

Figure 11 is the implementation of the function_package. The function_name is the name of a worker function, and function_arg is a list of the data_packages. The work_range records how many times the function will be executed, and execid is a list of execution units that will be used for the function. Matrix mmtx is a model view matrix used for the built-in GUI viewer.

The task generator creates several tasks depending on the parallel processing model. For example,

```
1  class Function_package():
2    def __init__(self):
3      self.function_name
4      self.function_args
5      self.work_range
6      self.execid_list
7      self.mmtx
8      . . .
9    # etc functions
10   . . .
```

Figure 11: The function package contains information about a task.

when the user uses the output split model, then each function package has a different work_range. In the input split model, every function has the same work_range, but each data_package in the function_arg has a different data range. For the in-and-out split model with identical shapes, work_ranges and data_packages have the same work_range and data_range.

5.2.2 Memcpy_package

The memcpy_package contains information about the data copy: source, data, function, and destination. One difference with the function package is that it uses a built-in data transfer function for non-axis-aligned data copies. Another difference is that it is an inter-process function, in contrast to the function package, which is an intra-process function.

The memcpy_package has an additional flag related to hard disk I/O. If the data source is a hard disk, then the task is assigned to the data reader for hard disk access. If the data destination is a hard disk, then it also assigned to the data reader. In addition, it has an out-of-core flag. If this flag is true, then the data reader saves the data immediately as they arrive. If the flag is false, then the data reader waits until every part of the data has arrived and saves it at once.

5.3 Process Design

All processes in Vivaldi share one basic structure, as shown in Figure 12. First, they initialize the environments and wait for a signal from any process. In line 7, the first signal is always a process number

```
1  # Initialization
2  ...
3
4  # Infinity loop
5  while True:
6    # Waiting any signal from the other process
7    source = comm.recv(MPI.ANY_SOURCE, tag=..)
8    flag = comm.recv(source=source, tag=..)
9
10   # Do assigned work at the flag
11   if flag == ..:
12     work() #
```

Figure 12: Basic structure of Vivaldi processes.

called the **source**, and the second signal is the **flag** from the source that indicates the kind of work. After line 11, there are many *if* statements for checking the flag and the functions for it.

This structure is a free to low-level implementation. Therefore, it is possible to optimize or change an implementation while correctly maintaining input and output.

5.3.1 Execution Unit

The current execution unit has two implementations: PyCUDA and Python. In the Python case, it is possible to run the worker function with additional outer loops for the worker range because the worker function has a Python-like syntax. However, in the PyCUDA case, the worker function is not directly executable in the GPU, hence it needs a translation step that converts it to CUDA code. In addition, it also requires a significant amount of programming effort for compilation, data transfer, synchronization, and so on.

The translator is implemented using Python. It consists of two steps: parsing and translation. In the parsing step, it determines the data types of the input, intermediate, and output and the variables for all scopes. For the next step, it translates the worker function code to target programming code such as CUDA using information from the parsing step.

Parsing is done-by-block, recursively. It splits the worker function into block levels that share the same scope. Furthermore, it recursively looks into small blocks to determine the data types for every scope. Inside a block, data type inference is done by Python operator precedence, arithmetic conversions, and the

pre-defined function in/output data type list. For example, if a function is found, then the output data types are inferred by the function dictionary and the data types of the input variables.

Translation uses the result of the parsing step and worker function code. This step changes the worker function code into the target language style. For example, GPU code requires a strict data type declaration for each data, so it adds data type declarations inside every scope. In addition, this translation step must consider the restrictions of the target architecture. For example, GPU kernel code does not allow function name overloading, therefore a different kernel name is necessary for every data type. This kind of restriction purely depends on the compiler. For instance, this example was valid for the Nvidia SDK 6.0, but it may not be a problem in later versions.

5.4 Pipeline Optimization

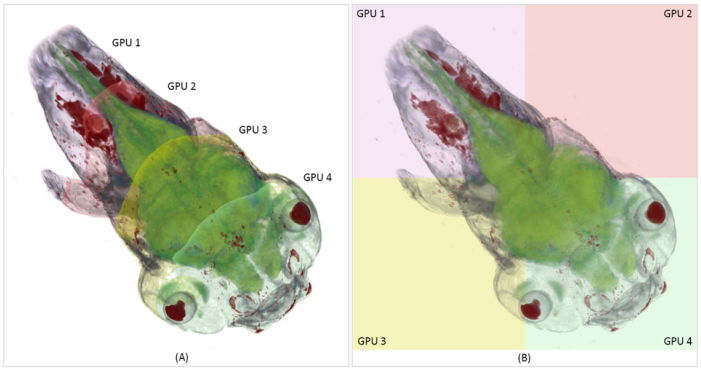
The scheduler has a simple pipeline optimization implemented through retain count and the function package. For example, if a pipeline has no sink, then the computation of the pipeline does not affect the final result. Hence, it is better to skip that computation.

The optimization is started when user removes a data pointer in the main function. If a user removes data, then the runtime system sends a signal to the scheduler. The scheduler then decreases the retain count of the data. If the retain count is zero, then the scheduler frees the data and checks a function list to determine if there is a function that has lost a sink. If a function package has lost a sink, then the retain count of the function input argument is also reduced. This is repeated until there are no function packages that have lost a sink.

6 Example

Vivaldi was tested on several large-scale visualization and computing applications, as shown below.

6.1 Distributed Parallel Volume Rendering




```
1 def main():
2   volume = load_data_3d(DATA_PATH+'/Zebra.dat')
3   enable_viewer(render(volume,x,y)
4                              .range(x=-512:512,y=-512:512)
5                              .split(volume, x=4)
6                              .merge(composite,'front-to-back')
7                              .halo(volume,1) ,'TFF2', '3D', 256)
```

(a) Sort-Last Parallel Rendering – Input Split

```
1 def main():
2   volume = load_data_3d(DATA_PATH+'/Zebra.dat')
3   enable_viewer(render(volume,x,y)
4                              .range(x=-512:512,y=-512:512)
5                              .split(volume, x=4)
6                              .merge(composite,'front-to-back')
7                              .halo(volume,1) ,'TFF2', '3D', 256)
```

(b) Sort-First Parallel Rendering – Output Split

Figure 13: Examples of distributed parallel volume rendering.

In this experiment, two different parallel renderings were tested: sort-first rendering and sort-last rendering. The data set used was the light sheet fluorescent microscopy (LSFM) of a juvenile zebrafish image with two different fluorescent color channels. The volume size is about 4.8 GB. In Vivaldi, parallel

processing model was selected using the **split** modifier. If the split modifier splits output data, it will be a sort-last rendering. If it splits the input data, then it will be a sort-first rendering. Figure 13 lists the programming code of the sort-first and sort-last rendering.

6.2 Distributed Numerical Computation

Many numerical computing algorithms can be easily implemented and parallelized using the in-and-out split and halo functions in Vivaldi. An iterative solver was implemented using a finite-difference method for 3D heat equations in Vivaldi and compared with a C++ version to assess the performance and usability of Vivaldi. As shown in Code 3, the 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 supplemental material, required at roughly 160 lines of code (counting the CUDA- and MPI-related lines) and in addition, it required knowledge of CUDA and MPI to write.

```
1  def heatflow(vol,x,y,z):
2    a = laplacian(vol,x,y,z)
3    b = point_query_3d(vol,x,y,z)
4    dt = 1.0/6.0
5    ret = b + dt*a
6    return ret
7
8  def main():
9    volume = load_data_3d(DATA_PATH+'data.raw')
10   for i in range(n):
11     vol = heatflow(vol,x,y,z).range(vol)
12                              .split(vol,x=2,y=2,z=2)
13                              .halo(vol,1)
14                              .output_halo(1)
```

Figure 14: Iterative 3D heat equation solver.

6.3 Streaming Out-of-Core Processing

Vivaldi can process large data in a streaming fashion. A segmentation algorithm was implemented in Vivaldi to extract cell body regions in electron microscopy brain images. The segmentation processing pipeline consisted 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 (4455 $\times$ 3408

× 512, float32). Vivaldi's disk I/O function provides an out-of-core mode such that a large file can be loaded as a stream of blocks and distributed to execution units. As shown in Figure, the user enables the out-of-core mode using the load_data _3D() and save_image() functions. The remaining code is the same as the in-core code. The halo and split modes can be used in out-of-core processing. The streaming I/O will attach halo data to each stream block when loading the data, and the number of tasks will be generated based on the parameters to split. Note that this implementation uses the in-and-out split and in-and-out halo to prevent halo communication between function executions (i.e., any task can be processed from median to threshold functions in order without needing to communicate with other tasks).

```
1  def main():
2    vol = load_data_3d('em.dat',out_of_core=True)
3    vol = median(vol,x,y,z).range(vol).split(vol,x=8,y=4).
4                            .halo(vol,18).out_halo(15)
5    vol = stddev(vol,x,y,z).range(vol).split(vol,x=8,y=4).
6                            .halo(vol,15).out_halo(12)
7    vol = bilateral(vol,x,y,z).range(vol).split(vol,x=8,y=4).
8                              .halo(vol,12).out_halo(7)
9    vol = minimum(vol,x,y,z).range(vol).split(vol,x=8,y=4).
10                            .halo(vol,7).out_halo(5)
11   vol = threshold(vol,x,y,z).range(vol).split(vol,x=8,y=4).
12                              .halo(vol,5)
13   save_image(vol,'result.raw',out_of_core=True)
```

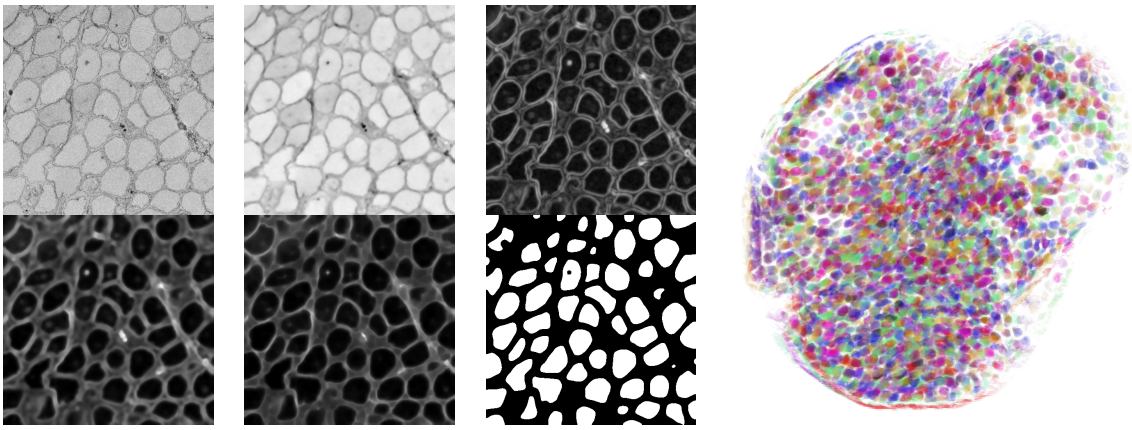


Figure 15: Streaming out-of-core processing for cell body segmentation in a 3D electron microscopy zebrafish brain image. Gray images are images in the pipeline. The colored image is a volume rendering of segmented cell bodies.

7 Result

7.1 Performance Evaluation

Vivaldi performance was evaluated using three benchmark tests written in Vivaldi and C++, MPI, and CUDA, as there are no domain-specific languages that work on cluster systems.

Volren is an isosurface volume rendering using the Phong shading model on a distributed GPU. The test used a 2 GB input volume and the output image was 1920 $\times$ 1080 full High-Definition (HD) resolution. The ray marching size was two for each step. **Bilateral** is a 3D bilateral filter processing of a 512 $\times$ 512 $\times$ 1512 floating-point (4 byte) 3D volume. The comparison bilateral filter used a C++, CUDA, and MPI implementation. The test comprised a single iteration of a bilateral filter of size 113. 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** is a 3D iterative heat flow simulation on a 512 $\times$ 512 $\times$ 1512 floating point 3D volume using the finite-difference method. Similar to the bilateral benchmark comparison code, a comparison iterative solver was implemented in C++, CUDA, and MPI. The benchmark used an in-and-out halo of size 10 and the total number of iterations was 50, hence the halo communication was performed once every 10 iterations. This benchmarks demonstrates the scalability of the system when halo communication is involved.

Program	Version	Lines	1 GPU	2 GPUs	4 GPUs	8 GPUs	12 GPUs
Volren	Vivaldi Sort-First	33	2.44	1.37(1.8x)	0.86(2.8x)	0.47(5.1x)	0.37(6.6x)
Bilateral	Vivaldi	35	114.54	57.52(2.0x)	28.9(4.0x)	14.41(7.9x)	9.61(11.9x)
	C++	160˜	92.38	47.36(1.9x)	24.30(3.8x)	12.46(7.4x)	8.53(10.8x)
Heatflow	Vivaldi input halo	12	11.19	5.84(1.9x)	3.47(3.2x)	2.09(5.3x)	1.65(6.7x)
	Vivaldi in-and-out halo	12	11.17	5.72(1.9x)	3.02(3.6x)	1.68(6.6x)	1.25(8.9x)
	C++ input halo	160˜	11.32	5.72(1.9x)	2.63(4.3x)	1.47(7.6x)	1.05(10.7x)
	C++ in-and-out halo	160˜	11.33	5.77(1.9x)	3.02(3.7x)	1.58(7.1x)	1.19(9.4x)

Table 1: Result of performance tests using three benchmarks for 1, 2, 4, 8, and 12 GPUs. The numbers in the cells are the essential number of lines, time in seconds, and scalability as the number of GPUs increase.

The number of GPUs was varied from 1 to 12 and the number of essential lines, performance in seconds, and scalability with the number of GPUs were measured .

The results for Volren in the Table 1 show that it has good scalability with two GPUs (1.8×) but only 6.6× scalability with 12 GPUs. The Volren problem does not have good scalability, but this is typical of volume rendering algorithms. In the parallel processing, if each GPU has one work range for rendering, then the total running time is decided by the last GPU. Furthermore, the work load in the Voren test is not well balanced because usually the data is in the middle of the screen, so GPUs mapped to the bottom of the image finish earlier than those mapped to the middle of the screen. Even though the scalability of Volren is not good, it can provide custom volume rendering with only 33 lines of programming code. In addition, the load unbalancing problem can be solved using a smaller grid size. If Volren divides the screen into much smaller pieces, then a GPU that is mapped to an empty space can join another part after it finishes its job. However, smaller block size can incur scheduler overhead, so balancing the load and scheduler overhead is necessary.

The Bilateral test shows good scalability because the GPU kernel is the dominant factor, the load is well balanced, and there is no data communication between the GPUs. In the benchmarks, Vivaldi shows comparable scalability with the manual C++ version, but the absolute computation time is slower. This is because the level of CUDA code optimization is different for the Vivaldi and hand-written code. The way Vivaldi executes the algorithm is to translate the user code to CUDA and execute it using PyCUDA. Therefore, if the CUDA code is the same, then the performance of the algorithm should be the same. However, the C++ code was optimized by a programmer. However, if the translator includes this optimization step, then Vivaldi could show better performance without any optimization on the part of the user.

Heatflow shows non-linear scalability because of the halo communication overhead. Halo communication does not exist with one GPU, but it increases as the number of GPUs increases. The decrease in scalability caused by halo communication overhead can be checked using the C++ Heatflow implementation. If we compare the input and in-and-out halo, we can see that the Vivaldi and C++ implementations show different tendencies. The C++ version performs better with an input halo, but the Vivaldi version performs better with an in-and-out halo. This difference come from the scheduler. In the C++ implementation, there is no scheduler because the programmer already knows how and when the functions and memory copies have to start, but Vivaldi uses a scheduler to map the task to an execution

unit, and the scheduler overhead increases as the number of tasks increases. In detail, an input halo has significantly more tasks than an in-and-out halo because there are more halo communications that have to occur, and each halo communication is considered one task. In the benchmark, the in-and-out halo communicates every 10 iterations, so there are 10 times more halo communication tasks. Even though the scheduler created overhead, this cannot be removed. Therefore, we need better scheduler algorithms to solve this performance problem.

7.2 Limitations and Future Work

The Vivaldi benchmark tests show slower execution times than C++. However, this is because there is no performance optimization. Basically, Vivaldi's back end is implemented by Python, which is 40 times slower than C++ in a simple loop statement test. This contributed to the creation of a huge scheduler overhead in the Heatflow input halo benchmarks because a 40$\times$ faster back end results in 40 times less overhead. This scalability decrease could be solved by a C++ back end implementation.

However, Vivaldi has room for performance optimization. First, the current Vivaldi translator directly translates Vivaldi code to Python or CUDA, but this programming code can be optimized during the translation step. For example, we can replace global memory to increase data read speed, and shared memory or loop unrolling can lead to performance increases. Second, performance can be improved by a better scheduling algorithm. Currently, the scheduler considers locality, but if it also considers the bandwidth and computation power of each device, then it can finish entire tests more quickly. Third, Vivaldi provides only regular grids, but unstructured grids could also be provided by new a new modifier in the future. Even though Vivaldi has an extendable design for various accelerators, currently the execution unit only supports CPUs and Nvidia GPUs because there are only two translator implementations, one for a CPU and the other for a Nvidia GPU. However, more translator implementations will be added in the future such as OpenCL, and Vivaldi can support AMD GPUs or Xeon Phis. In addition, the Vivaldi language is designed for volume rendering and processing, but the target domain can be extended in the future for other visualization purposes such as meshes, vectors, and so on.

8 Conclusion

In this work, Vivaldi, a Python-like domain-specific language for volume rendering and processing on distributed heterogeneous system was proposed. In addition, its system architecture, algorithm for system management, and parallelism were explained. The evaluation showed that Vivaldi has comparable scalability to a C++ implementation, but requires only 8–25% of the programming code. Currently, Vivaldi has much room for improvement such as in the scheduler, translator, compiler, and target domain.

References

[1] M. P. Forum, "Mpi: A message-passing interface standard," tech. rep., Knoxville, TN, USA, 1994.

[2] "Voreen: Volume rendering engine." `http://www.voreen.org/`.

[3] "Osirix." `http://www.osirix-viewer.com/`.

[4] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer, "Diderot: A parallel dsl for image analysis and visualization," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 111–120, ACM, 2012.

[5] 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 Trans. Graph.*, vol. 31, pp. 32:1–32:12, July 2012.

[6] "Volume ray casting." `http://en.wikipedia.org/wiki/Volume_ray_casting`. Accessed: 2015-01-05.

[7] M. Ament, F. Sadlo, and D. Weiskopf, "Ambient volume scattering," *IEEE Trans. Vis. Comput. Graph.*, pp. 2936–2945, 2013.

[8] S. Frey, F. Sadlo, K.-L. Ma, , and T. Ertl, "Interactive progressive visualization with space-time error control," in *Proceedings of IEEE SciVis 2014 (also IEEE TVCG 20(12))*, November 2014.

[9] M. Hasan, J. Wolfgang, G. Chen, and H. Pfister, "Shadie: A domain-specific language for volume visualization.," draft paper, 2010.

[10] H. T. Vo, D. K. Osmari, J. Comba, P. Lindstrom, and C. T. Silva, "HyperFlow: A Heterogeneous Dataflow Architecture," in *Eurographics Symposium on Parallel Graphics and Visualization* (H. Childs, T. Kuhlen, and F. Marton, eds.), The Eurographics Association, 2012.