SCI INSTITUTE
TECHNICAL REPORT

Multi-Threaded Streaming Pipeline For VTK

Huy T. Vo and Cláudio T. Silva

UUSCI-2009-005

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

May 27, 2009

Abstract:

In this document, we describe the implementation details of our proposal on how to modify the VTK pipeline execution framework to support improved streaming and multi-threaded capabilities. We believe the functionality reported here is the best way to start adding this functionality to VTK. (See [1] for a higher-level description of the work that includes more functionality). The plan would be to first settle on the basic functionality; after that, it should not be hard to continue adding the rest of the framework to VTK (e.g., support for streaming unstructured data structures and GPUs).

THE UNIVERSITY OF UTAH

Multi-Threaded Streaming Pipeline For VTK

Huy T. Vo and Cláudio T. Silva

SCI Institute
University of Utah

May 27, 2009

1 Introduction

In this document, we describe the implementation details of our proposal on how to modify the VTK pipeline execution framework to support improved streaming and multi-threaded capabilities. We believe the functionality reported here is the best way to start adding this functionality to VTK. (See [1] for a higher-level description of the work that includes more functionality). The plan would be to first settle on the basic functionality; after that, it should not be hard to continue adding the rest of the framework to VTK (e.g., support for streaming unstructured data structures and GPUs).

2 Pipeline Architecture

In the most current implementation (version 5.4), VTK uses `vtkStreamingDemandDrivenPipeline` as its default executive. This executive supports streaming by performing multiple updates with smaller regions of the input data. This is done by allowing algorithms to modify information of a request to notify the executive of continuous execution as well as update extents. An example is the image streaming class `vtkImageDataStreamer`, which is usually connected at the end of an image streaming network. When using this class, instead of passing the whole extents of the output image upstream once for update, the pipeline breaks this extent into multiple regions and pass them upstream in multiple passes. In particular, for an update request, it puts the current update extent into the `UPDATE_EXTENT` block of the request, and use the flag `CONTINUE_EXECUTING` to trigger new update passes from the executive if there are more pieces needed to be processed.

This design can be used to increase memory locality and pipeline performance, however, it is not efficient on multi-core machines. The main reason is that calling updates on algorithms causes a lack of parallelism during execution. When an update is called on an algorithm, with a `REQUEST_DATA` request, upstream modules are always called to update sequentially, even though in many cases, they can all be updated concurrently. Similarly, with streaming,

updates are also being called sequentially, thus, at any particular time, there is only one algorithm actively processing a piece of data. This results in an under-utilization of resources when other modules could process other pieces of data at the same time.

The goal of our Streaming and Multi-Threaded framework (for more details, see [1]) is to resolve the above issues by deriving a new executive, `vtkThreadedStreamingDemandDrivenPipeline`. We add two parallel updating mechanism to VTK:

Multi-Threaded Update For a pipeline, when an update requests is passed upstream, upstream modules will be updated simultaneously in separate threads. The number of threads is by default limited by the number of cores on the running machine. To avoid the case of an algorithm is requested to update by more than one downstream module, which would cause code failure due to the non-reentrant implementation of most algorithms, modules with more than one immediate downstream modules will be explicitly updated before their children. The multi-threaded capability is turned on by default for all VTK modules using our executive, but can be turned off by a global call to `vtkThreadedStreamingDemandDrivenPipeline::SetMultiThreadedEnable`.

Parallel Streaming Computation For a streaming pipeline in our framework, modules are allowed to process concurrently with different pieces of data. To ensure the efficiency in resource usage, we also implement a scheduler that is responsible for queuing module executions as well as how many threads they should run (if they can utilize more than one thread). Each module is running in their own thread(s) and will become active only when the data of its upstream modules are ready. Since, in VTK, the data object output from one module is passed directly downstream, modules with immediate relation (direct child or parent), are not allowed to run simultaneously, to avoid overlapping data accesses. In order to use this new streaming pipeline, a streaming network must begins with *Stream Generator*(s) and ends with a *Stream Merger* respectively. Stream generators are responsible for producing pieces data and the mergers will collect the data and merge/write to the final output. A merger can have multiple generators and there can also be more than one mergers in a single pipeline. The actual implementations of these generators and mergers can greatly differ from one network to another, depending on the computation.

3 Implementation Details

Here is the list of files that have been modified and/or added to VTK to support our multi-threaded streaming framework and a simple implementation of image data streaming.

3.1 List of Files

Added

`Filtering/vtkThreadedStreamingDemandDrivenPipeline.h`
`Filtering/vtkThreadedStreamingDemandDrivenPipeline.cxx`

Filtering/vtkStreamInterface.h
Filtering/vtkStreamInterface.cxx
Filtering/vtkComputingResources.cxx
Filtering/vtkComputingResources.h
Filtering/vtkExecutionScheduler.cxx
Filtering/vtkExecutionScheduler.h

Implementation of Image Data Streaming
Imaging/vtkImageDataStreamGenerator.cxx
Imaging/vtkImageDataStreamGenerator.h
Imaging/vtkImageDataStreamMerger.cxx
Imaging/vtkImageDataStreamMerger.h

3.2 Details

Below, we describe the implementation details of each file or changes to existing files.

Filtering/vtkStreamInterface.{h,cxx}

The streaming interface consists of a set of functions such as `IsStreamGenerator` and `IsStreamMerger` to indicate whether an algorithm is a stream generator or merger. A stream generator can then further implement `IsEndOfStream`, `StreamRestart`, `StreamNext` and `StreamRetrieve` to control how data should be generated. Likewise, a stream merger also needs to implement `StreamMergeCurrentInput` to collect data from upstream pieces. Though the streaming interface can be separated into a separate interface class, it is more convenient for algorithm developers if it is incorporated with `vtkAlgorithm`, since subclasses from VTK's default algorithms such as `vtkImageAlgorithm`, `vtkPolyDataAlgorithm`, etc. can also support streaming.

An algorithm can define a streaming module by setting either `IS_STREAM_GENERATOR` or `IS_STREAM_MERGER` key of `vtkThreadedStreamingDemandDrivenPipeline` in their `Information` property. If any of these two keys are defined, the algorithm is also required to define a `vtkStreamInterface` object in their `Information` property under the key `STREAM_INTERFACE`. This object should implement the above interface methods so that the executive can interact with the module.

Filtering/vtkThreadedStreamingDemandDrivenPipeline.{h,cxx}

This is the default executive of our framework. This inherits directly from `vtkStreamingDemandDrivenPipeline`, with changes to 2 functions `ForwardUpstream`, `ProcessRequest`, and `CallAlgorithm`.

`ForwardUpstream` interrupts the default forwarding requests when the request is `REQUEST_DATA` and uses a thread-pool to perform updates simultaneously. The maximum number of thread of this thread pool is also obtained from `vtkMultiThreader:: GetGlobalDefaultNumberOfThreads()`. `ForwardUpstream` also goes up more than one level of

immediate upstream modules to search for modules with more than one output connections and update those first before spawning threads.

`ProcessRequest`, `CallAlgorithm` and the rest of the code in these two files are for performing the general streaming framework. The change in `ProcessRequest` is simply for logging the computation time of all `REQUEST_DATA` requests. These statistics are used for the scheduling strategy. `CallAlgorithm` is the actual control flow of the streaming framework. Its general algorithm can be briefly expressed in an event-driven fashion as below, however, all the modules are updated concurrently with the help of the scheduler.

```
StreamGenerator->StreamRestart()
while not StreamGenerator->IsEndOfStream() do
    StreamGenerator->StreamRetrieve()
    <Pass data down to Streaming Modules>
    StreamMerger->StreamMergeCurrentInput()
    Scheduler->Reschedule()
    StreamGenerator->StreamNext()
```

This algorithm is triggered whenever an update is request on any stream merger. From the merger, a simple traverse upstream is performed to collect all corresponding stream generators. All the modules that stay between those stream generators and merger will be mark as streaming modules. Thus, each block of stream merger and generators are associated with its own scheduler and control flow.

Filtering/vtkComputingResources.{h,cxx}

This file holds the definition of a computing resource, which can be either CPU (threads) or GPU (CUDA kernels). The interface implements a few operation on resources to be used by the scheduler. One of them is `IncreaseByRatio`. This basically tells the scheduler how to divide a resource based on a ratio (its updated time). The current implementation for VTK only uses the CPU threads as its resources. In order to manage resources for subclasses of `vtkThreadedImageAlgorithm` without changing the default code, the CPU resource class also knows how to setup the resource to an algorithm (by checking the class name and call `SetNumberOfThreads`).

Filtering/vtkExecutionScheduler.{h,cxx}

This class is responsible for scheduling resources as well as executions for a network of modules. Resources can be distributed by calling `RescheduleNetwork`. This function will start from a stream merger, and based on its computation time and how long it took for its upstream modules to deliver inputs in the previous request, it will divide the resources, the number of threads, across all of them. Then the distribution process are recursively propagated upstream until it reaches the stream generators. Currently, we uses a greedy strategy to split the number of threads evenly with the time ratio.

The scheduler also manages the execution of modules such that resources are fully utilized without being overflow. When the executive requests to update through `Run`, it will be put

to sleep on the scheduler queue and wait until there are enough resources to run. Once a module is done executing, `Finish` will be called to suspend the module and put a lock on itself while releasing all the locks from its immediate modules to prevent overlapping data accesses.

Imaging/vtkImageDataStreamMerger.{h,cxx}

This is a concrete implementation of a stream merger for image data. Its role is similar to `vtkImageDataStreamer`, and should be connected at the end of a streaming network to collect and merge pieces of data after processing. Besides using the same `NumberOfStreamDivisions` property as `vtkImageDataStreamer` to control how many pieces to break the stream. It also implements the following 4 functions:

```
virtual bool IsStreamMerger() { return true; }
virtual int  ProcessRequest(vtkInformation*,
                            vtkInformationVector**,
                            vtkInformationVector*);
virtual void StreamMergeCurrentInput(vtkInformationVector*);
```

Since the streaming algorithm is executing in a top-down, event-driven manner, the overlapping region of the input pieces must be computed beforehand. This is done by the re-implementation of the handler of `REQUEST_UPDATE_EXTENT` in `ProcessRequest`. It will pass a sample extent, which is equivalent to the size of a piece extent, in `STREAM_SAMPLE_EXTENT` all the way up to the generators and perform a difference computation of the overlapping region. `StreamMergeCurrentInput` will concatenate pieces together by appending later pieces to the bottom of the image.

Imaging/vtkImageDataStreamGenerator.{h,cxx}

This is a concrete implementation of a stream generator for image data. It should take inputs from any image reader or could be subclassed to act as a stream image reader. The following functions are implemented:

```
virtual bool IsStreamGenerator() { return true; }
virtual int  ProcessRequest(vtkInformation*,
                            vtkInformationVector**,
                            vtkInformationVector*);
virtual void StreamRestart();
virtual int  StreamNext();
virtual void StreamRetrieve(vtkInformationVector* outputVector);
virtual bool IsEndOfStream();
```

In working with `vtkImageDataStreamMerger`, the `ProcessRequest` function in `vtkImageDataStreamGenerator` will take the `STREAM_SAMPLE_EXTENT` information to setup its piece size as well as how large its overlapped region should be by subtracting sample extent from `UPDATE_EXTENT`. `StreamRestart` is just for resetting the current piece number to

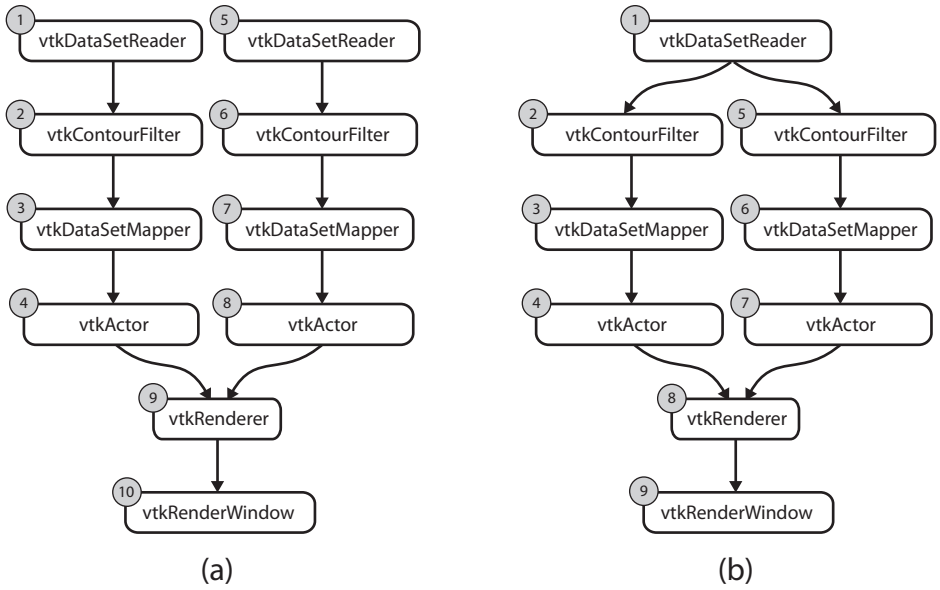

Figure 1: Two simple iso-contour pipelines extracting two surfaces from (a) two different datasets and (b) a single dataset.

0. The piece number is used in `StreamNext` to construct the piece extent and pass upstream for requesting data. `StreamRetrieve` will retrieve the current piece data of the stream generator and pass it to the module output. Finally, `IsEndOfStream` will return true when the current piece extent is falling outside the whole image extent.

COMPATIBILITY ISSUES

To preserve the ability of being cross-platformed of VTK, we use `vtkMultiThreader` and `vtkMutexLock` for multi threaded implementation as well as `vtkTimerLog::GetUniversalTime()` to measure the performance time of module executions. Moreover, our new executive is also backward-compatible to all existing VTK's pipelines, however, it can achieve higher performances especially with new streaming pipelines.

4 Illustrative Examples

4.1 Multi-Threaded Update

Figure 1 illustrates two simple pipelines that could benefit from our multi-threaded update mechanism. The number on the top left corner indicate the order of update using the default VTK execution engine. In the new framework, all requests, except `REQUEST_DATA`, will also be processed in the same order. For the case of Figure 1a, when `REQUEST_DATA` was requested from `vtkRenderWindow` to `vtkRenderer`, which has two upstream modules, the thread pool will spawn 2 threads, one to go up `vtkActor` (4) and the other to (8). This results in independent executions of two sets of modules (1,2,3,4) and (5,6,7,8).

Similarly, in Figure 1b, a pipeline of rendering two iso-surfaces out of a single dataset, however, if upstream modules of `vtkRenderer` were to be updated simultaneously in two threads, the module `vtkDataSetReader` may be re-entered from both threads at the same time. *This will crash since VTK classes are not thread-safe.* With the current implementation of `vtkThreadedStreamingDemandDrivenPipeline`, when forwarding `REQUEST_DATA` from `vtkRenderer`, it will go upstream and check for modules more than one output, which is `vtkDataSetReader` in this case, and explicitly update it first. This will cause later requests from (2) or (5) will be ignored since it is already up-to-date. Thus, the execution of this pipeline is (1), then concurrent execution of two sets (2,3,4) and (5,6,7), then (8) and (9).

4.2 Parallel Streaming Computation

Figure 2 shows how an edge detection pipeline can be setup in VTK using our framework. In Figure 2a, there is no streaming, each modules is updated sequentially from (1) to (8). Because, modules (2,3,4) are inherited from `vtkThreadedImageAlgorithm`, it can actually run in multiple threaded. For example, on a machine with 4-core, (2) will be executed using 4 threads, then (3) will execute in 4 threads, and so as (4). Figure 2b shows a typical VTK image streaming pipeline, where `vtkImageDataStreamer` is plugged into the end of an image network. This class will separate output extent into smaller chunks and perform multiple update requests. For example, if `vtkImageDataStreamer` has the property `NumberOfStreamDivisions` set to 3, then the image will be divided into 3 portions. Then, it will sequentially apply modules (2,3,4,5) on each portion. Briefly, if each module takes 1 unit of time to update, the time line of execution for Figure 2b is {1, 2,3,4,5, 2,3,4,5, 2,3,4,5, 6, 7, 8, 9}.

Figure 2c is how the same streaming pipeline can be implemented in our framework. Besides `vtkImageDataStreamMerger` which is similar to `vtkImageDataStreamer`, `vtkImageDataStreamGenerator` is also required to put at the beginning of a streaming network. When the module (6), `vtkImageDataStreamMerger`, is requested to update, it will go upstream and search for `vtkImageDataStreamGenerator` and find (2). All modules inside this block will be updated parallel using the scheduler of (6). Given the data locking mechanism, there are two sets of modules (2,4,6) and (3,5) can be updated simultaneously. Again, for a division of 3 pieces and if each module takes exactly 1 unit of time to update, the corresponding execution time line is {1, 2, 3, (4,2), (5,3), (6,4,2), (5,3) (6,4), 5, 6, 7, 8, 9, 10}, with concurrently updated modules placed inside the parenthesises.

Figure 2d shows a more complicated example as one merger may have multiple generators. The example is a blend of edge detection with a background image. In order to work in our framework, it requires that the two generators can split to the same number of pieces as well as their pieces can be computed in pairs. The whole block between `vtkImageDataStreamMerger` and two `vtkImageDataStreamGenerator` also share a single scheduler and controlled by the `vtkImageDataStreamMerger`'s executive.

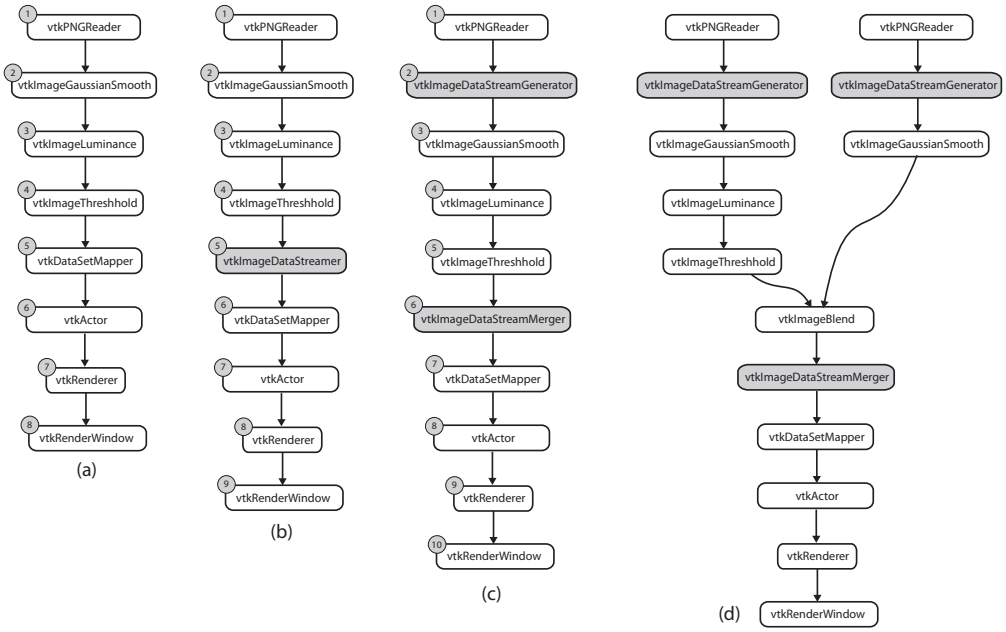

Figure 2: Difference in VTK (a), VTK's streaming (b) and our streaming framework (c) in an edge detection pipeline. (d) illustrates a pipeline with one stream merger against more than one stream generator in our framework.

References

[1] H. T. Vo, D. K. Osmari, B. Summa, J. L. Comba, V. Pascucci, and C. T. Silva. Parallel dataflow scheme for streaming (un)structured data. Technical report, SCI Institute, University of Utah, 2009. unpublished.