Modular parallelization framework for multi-stream video processing

Tim Lenertz
Université Libre de Bruxelles, LISA department
Avenue Franklin D. Roosevelt 50
1050 Brussels, Belgium
+32 2 650 30 82
tim.lenertz@ulb.ac.be

Gauthier Lafruit
Université Libre de Bruxelles, LISA department
Avenue Franklin D. Roosevelt 50
1050 Brussels, Belgium
+32 2 650 30 89
gauthier.lafruit@ulb.ac.be

SUBMITTED to ACM MULTIMEDIA 2016 OPEN SOURCE SOFTWARE COMPETITION

ABSTRACT

A flow-based software framework specialized for 3D and video is presented, which in particular handles automatic parallelization of multi-stream video processing. The workflow is decomposed into a set of filter units which become nodes in a graph. Execution with support for time windows on node inputs is automatically handled by the framework, as well as low-level manipulation of multi-dimensional data.

CCS Concepts

• **Computer systems organization~Data flow architectures**
• *Computing methodologies~Image manipulation*
• Computing methodologies~Computer vision
• *Software and its engineering~Middleware*

Keywords

video; view synthesis; FTV; parallelization; multi-stream; 3D

1. INTRODUCTION

In video and image processing, algorithms are often developed in an ad hoc manner. An existing solution is used as test bed, and incremental improvements are made to it in order to improve the visual quality of the output.

This is for instance the case in the MPEG-FTV (free viewpoint TV) standardization activities [6], where the goal is to synthesize virtual-viewpoint images of a scene that is captured by multiple real cameras, and possibly depth sensors which output point cloud data.

Such an exploratory development approach often lacks a software architecture and eventually causes maintenance problems. To remedy this problem, we propose a neat software framework providing a generic implementation of elements common to FTV-related algorithms, including a data flow structure into which *filter* modules that process video frames are inserted. A major feature is the support for automatic parallelization of the workflow, and output-driven partial activation of multi-stream input.

Figure 1 shows an example of such a flow graph for FTV which will be further elaborated in section 4. The diamond-shaped nodes are filters implemented as independent modules. Also shown are thumbnails of image frames passing through the graph.

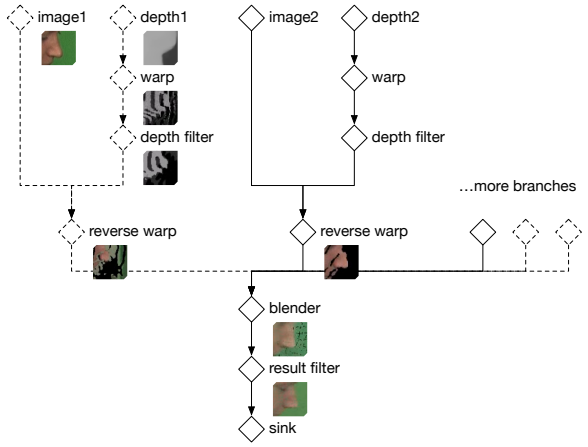

Figure 1. Example flow graph of FTV virtual view synthesis with multiple input streams

2. GOALS

One aim in the design of the framework is to clearly separate incrementally developed *experimental* code, and the *definite* code of the generic, reusable framework components.

It should also be *modular*, in the sense that external code, which does not reference any of the framework functionality, can easily be fitted into a plug-in and used as a node in the flow graph.

The framework remains as versatile as possible, and provides a greatest common denominator of the functionality required for multi-stream FTV applications. For instance, *filter* nodes may have many inputs which get dynamically activated or deactivated, and may receive bidirectional time windows of their inputs. For example the "blender" node in Figure 1 activates its input branches in function of a virtual camera position that varies in time. The flow graph framework should be versatile enough to make it possible for complex algorithms to be decomposed and remodeled into this structure.

Parallelization is handled by the framework itself, both on an *inner* (code used by the filters) and *outer* (coordination of nodes) level.

Also, the framework is written in modern C++, using techniques like RAII[1] and static polymorphism, and is extensively tested.

[1] Resource acquisition is initialization

3. FEATURES

The framework is currently built following a bottom-up approach with the goal of getting the required components to eventually develop FTV view synthesis. It consists of low-level *core* components for multi-dimensional data manipulation and queuing, the *flow graph system* which is built on top of this, and additionally a *media toolkit*, which is a loose collection of useful components for 3D and video applications.

3.1 N-d array

The data processed in DIBR[2] and 3D point cloud algorithms often has the form of *n*-dimensional homogeneous arrays, whereby types can be scalars, vectors or tuples and may be null-able. For instance, a masked image is a 2D array of null-able RGB 3-vectors, and a point cloud is a 1D array of XYZRGB tuples.

A strided *n*-d array type is therefore provided at the core level of the framework, with the basic functionality like slicing, sectioning and iteration.

Copy-less type casts make it possible to pass for example an `ndarray<1,xyzrgb>` object to a function which expects an `ndarray<1,xyz>`, or possibly an `ndarray<2,float>`, without copying data or breaking the encapsulation, or unnecessarily templatizing the function.

A wrapper `masked_elem<T>` may add a binary mask flag that makes the type null-able (unless it already is so), and it can also be cast away similarly. Having this support at the core level removes difficulties in application code caused by differing conventions for marking null values, e.g. separate binary mask images or special "background color" values.

Type erasure of lower dimensions is also supported. For instance components which work on full video frames may safely receive video data in the form of an `ndarray<1,frame>` instead of `ndarray<3,rgb>`. Code complexity is reduced, but the framework still assures type safety and proper memory alignment.

Moreover, a kind of `reinterpret_cast` operation is provided which allows for in-place processing on an `ndarray` with different input and output element types.

3.2 Queues

On top of the type-erased `ndarrays`, FIFO ring buffer classes are implemented. Features include absolute time indices on frames, and dual-thread support (one reader and one writer) with mutual waiting for readable/writable frames and deadlock prevention. These rings allow to access past and future frames for temporal processing in the flow graph, cf. next section.

Two variants exist, a *seekable* ring buffer where the reader may seek to another absolute time in the stream and the writer responds accordingly, and a *non-seekable* ring buffer where the stream duration does not need to be known at construction and the writer marks the end after it has written the last frame.

The circular buffer wrap-around is implemented using the operating system's virtual memory mapping functionality. This way, no special handling is required for views that cross the buffer's boundary.

The flow graph system makes use of these classes to implement the data flow with both seekable files, as well as non-seekable real-time sources.

3.3 Flow graph

The flow graph is the tree structure through which frames of the processed media stream flow. All data in the flow graph has an *n*-d array format, described by a dimension, element type, and frame shape[3]. Nodes in the graph perform a concrete frame-wise media processing step.

The flow graph always contains exactly one *sink* node, and one or multiple *source* nodes. Frames are recursively *pulled* from the sink, back through the preceding nodes, up to the sources. Nodes may have more than one input, and (currently) always have one output. Inputs are connected to outputs of other nodes in a one-by-one manner; the graph can have no loops.

Each node contains a *filter* which does the concrete frame processing and may be implemented as a module external to the framework.

3.3.1 Features

The data format of a frame is described by a *n*-d array dimension, element type, and frame shape. Input and output ports of nodes can each have different formats. Time is represented as an integral index value. The nodes operate frame-wise and invariably write one frame to their output at each step.

Nodes can receive a *bidirectional time window* on their inputs: on construction, the node specifies, for each input, the number of past and future frames that it needs to receive, in addition to the current frame. The system ensures that nodes receive this time window for each frame. It gets truncated only near the beginning and end of the stream. Previous frames are retained, and no unnecessary copies are made. This allows, for example, the implementation of a node which applies an image kernel filter over both time and space, on a 3D *n*-d array.

Nodes may *activate* and *deactivate* their inputs at runtime. No frames are pulled from graph branches connected to deactivated inputs. When an input is reactivated after having been deactivated (e.g. a change of virtual view position, modifying the processing flow), the intermediary frames are skipped. If the source node at the end of the branch is seekable, a *seek* request gets propagated towards it, and intermediary frames are never loaded or processed. With this it is possible to implement nodes with a large number of inputs, out of which only a small number are active at a time.

Finally, inter-node parallelism is supported with *asynchronous nodes*. These nodes have the same features as *synchronous nodes*, but instead of processing frames when pulled from the output, they run a separate thread and independently process frames in advance. Pull requests wait until the requested frames become available. For example, while the sink is processing frame t, a preceding asynchronous node may be processing frame $t + k$ at the same time. The maximal number of frames an asynchronous node may be in advance of the frame pulled from its output is called its *prefetch duration* and can also be configured at runtime.

Any *filter* can be run on a synchronous or asynchronous node, and this can be specified at runtime. Additional node types may be added in the future, such as one that processes multiple frames in parallel. Possibly, *OpenCL* can also be integrated at the framework level.

Nodes can also have parameters which are either set to a constant value, to a function of time, or to mirror a parameter of another node.

---

[2] Depth image-based rendering

[3] e.g. for 2D images, its width and height

3.4 Media toolkit

Currently the media toolkit mainly consists of an encapsulation of extrinsic and intrinsic camera matrices that are often used in FTV and point cloud applications. In a *space object* hierarchy 3D objects have poses relative to each other, and classes for pin-hole camera models are implemented. Several depth projection conventions are implemented, providing direct support for different depth image formats, and for *OpenGL* integration.

Also included are an interface to *OpenCV* for image processing tasks, generic *n*-dimensional kernel filter functions, and point cloud support. The next section provides a concrete example of some of its functions used in a multi-stream FTV application.

4. DEMONSTRATION PROGRAM

As stated in the introduction, the goal of free-viewpoint video (FTV) is to synthesize virtual-viewpoint images of a scene that is captured by multiple real cameras and possibly depth sensors. This typically involves a reconstruction step where scene geometry is estimated from the captured images, followed by a view synthesis step which generates the virtual view.

Numerous approaches to this have been developed [4][8], and a call for evidence for new FTV technologies was recently published by MPEG, end of 2015 [6]. MPEG uses the programs DERS[4] and VSRS[5] as reference software to evaluate the results of candidate algorithms proposed by MPEG committee members. They use an algorithm based on stereo matching and image warping.

A small demonstration program was written using the framework which generates a video with a moving virtual camera based on the *Poznan Blocks* sequence [5] (see Figure 2). The input data consists of 9 fixed position camera feeds, along with depth sensor data. These depth images are of low quality and have blurred edges and other artifacts.

The algorithm replicates the one employed by VSRS in its most basic form, except that now more than two input views are taken. Here, the flow graph system is used for automatically activating or deactivating the contributing or idle nodes. The goal of the demonstration program is not to reach high-quality output (much better FTV results exist already), but to have a program which is written from scratch and uses almost all of the framework's features, for validation.

Figure 2. Demonstration program output

[4] Depth Estimation Reference Software [6]

[5] View Synthesis Reference Software

4.1 Algorithm

Figure 1 shown earlier depicts the flow graph and the filters that this demonstration program consists of. The *main* function of the program only constructs and runs the graph. All of the concrete processing steps are implemented as filter modules. Note that the "blender" node has 9 inputs; only two of the branches are shown on the figure.

4.1.1 Method

In two camera images depicting the same scene from different viewpoints, 2D points from one *source* image can be mapped to points from the other *destination* image, knowing its depth in either image, and the homography matrix. The process of copying each pixel in an image into its corresponding location on a new destination image is called *image warping*, and is an elementary method for view synthesis.

The algorithm first warps the input depth image into the virtual camera view. The resulting depth image (see Figure 3) contains virtual shadows for regions occluded from the original viewpoint, and small holes due to the limited resolution of the input images. The depth image is refined using a kernel filter. Then a *reverse warping* is performed to transform the input image into the virtual camera view, using the new *destination* depth image. (Figure 3)

The resulting images for each input view depict the scene from the same viewpoint, but have different occlusions. The three whose original camera positions are closest to the virtual camera are picked out and blended together.

The final image is again refined using a 3D kernel filter over an image and time window. It removes flickering pixels for example.

4.1.2 Implementation

The different steps are performed by connected nodes in a flow graph. Holes in the transformed images and depth maps are represented using the `masked_elem` wrapper. The 4×4 homographic matrices used for warping are obtained using the camera model classes from the media toolkit, and also include the depth reprojection. No low level matrix operations are performed by the filter code.

The "blender" node activates at each frame 3 of its 9 inputs, and deactivates the others. When – due to a new virtual viewpoint position – an input is reactivated after having been inactive, the branches need to jump to the new time index. This *seek* operation is implicitly propagated through the branch, up to the source nodes which then change the file position. Intermediary frames are never loaded or processed by the "warp" or "depth filter" nodes.

The "reverse warp" nodes are set to be *asynchronous*. Thanks to this, the program gets automatically parallelized into 4 threads, with three separate threads for the branches connected to the active blender inputs. The point of synchronization of two threads is restricted to the ring buffer between two nodes. This improves runtime on multiprocessor systems, as shown in the next section.

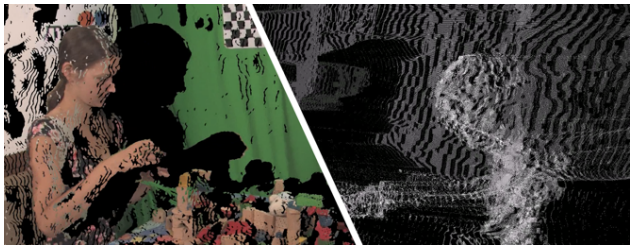

Figure 3. Image after reverse warping \ after depth warping

5. BENCHMARK

In order to measure the performance gains reached with the parallelization, the processing durations and CPU usage of the demonstration program were compared without parallelization (Figure 4) and with the 4 thread configuration described before (Figure 5).

The X-axis represents real time in seconds. The black curve (with the markers) is the real time between subsequent frames arriving at the sink. The blue curve indicates the CPU usage of the program at that time. An 8-core machine is used, and a value of 800% means that all cores are fully used. The full stream consists of 200 frames.

With parallelization, the total time it takes to process the streams is lowered from 170 seconds to 65 seconds.

CPU usage averages at around 350%, for the 4 threads. The spike at the beginning occurs because the "blender" deactivates its input branches only when it starts processing the first frame. Without parallelization, only one CPU core is used at 100%.

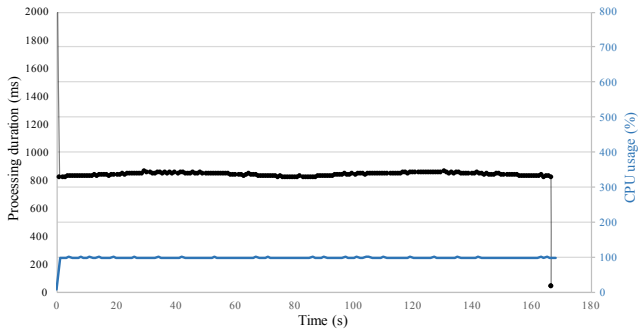

Figure 4. Sequential processing times and CPU usage

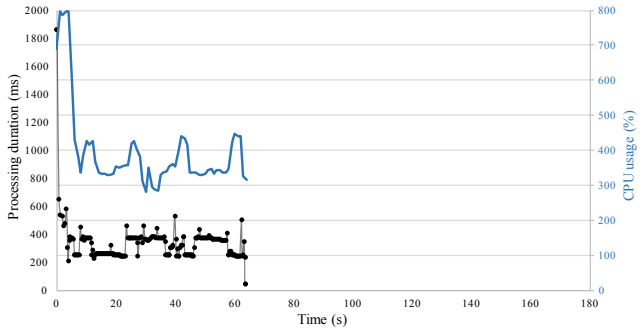

Figure 5. Parallelized processing times and CPU usage

6. USAGE

The current version of the framework is available for download at http://timlenertz.github.io/mf/. The site also contains source code documentation, the test sequence, build and usage instructions.

The framework currently only supports Linux and Darwin (OS X). Some low-level components (virtual memory mapping, thread synchronization) depend on the operating system and are only implemented for these targets. A compiler with full support for C++14 is required. The framework is developed and tested on *LLVM/Clang* 3.8. Standard GNU *Makefiles* are used for building.

7. SIMILAR SYSTEMS

RabbitFire [7] is a similar data-flow framework with the primary purpose of handling parallelization at the framework level, and it is also intended for video processing. It reaches very high efficiency by using lockless thread synchronization, also supports intra-node parallelism and input time windows. However, it runs only on Windows, and currently does not support asynchronous nodes. These allow for high speed gains in our framework.

Intel Thread Building Blocks [3] is a more general-purpose solution for data flow graphs. It is based on a notion of message passing instead of a stream of frames, making it less useful for the workflow used here.

GStreamer [2] is an open-source modular data-flow system with heterogeneous media formats. It is mostly intended for conversion and playback of different video and audio formats, and not specific for frame-wise processing.

On the Apple platform, *Quartz Composer* [1] is a graph-based system which is integrated into the operating system, and mainly used for visual GUI effects and screensavers. It is not primarily intended for use with custom application-specific plugins.

8. CONCLUSION

A C++ software framework specialized for multi-stream FTV was presented. It handles low-level data manipulation and implements a high-level data flow structure, providing support for application modules that concentrate on the specific image processing. In particular, multi-stream input, parallelization, time windows and implicit seek operations are supported. Its main advantage against similar systems is its targeted development towards FTV applications.

9. REFERENCES

[1] Apple Inc. Quartz Composer. https://developer.apple.com/osx/

[2] GStreamer Team. GStreamer. https://gstreamer.freedesktop.org

[3] Intel®. Thread Building Blocks. https://www.threadingbuildingblocks.org

[4] Lee, C., Tabatabai A. and Tashiro K. 2015. Free viewpoint video (FVV) survey and future research direction. *APSIPA Transactions on Signal and Information Processing.* 15, 4 (Oct 2015).

[5] Domański, M., Dziembowski, A., Kuehn, A., Kurc, M., Łuczak, A., Mieloch, D., Siast, J., Stankiewicz, O., Wegner, K. Poznan Blocks - a multiview video test sequence and camera parameters for Free Viewpoint Television. *ISO/IEC JTC1/SC29/WG11 MPEG 2014/M32243*. San Jose, USA, (Jan 2014)

[6] MPEG. Call for Evidence of Free-Viewpoint Television: Super-Multiview and Free Navigation – update. *ISO, Coding of Moving Pictures and Audio. N15733*. (Oct 2015) http://mpeg.chiariglione.org/standards/exploration/free-viewpoint-television-ftv/call-evidence-free-viewpoint-television-super-0

[7] NEXTWave Technologies SPRL. RabbitFire™. http://www.nextwave-techs.com/rabbitfire.html

[8] Smolic, A. 2011. 3D video and free viewpoint video - From capture to display. *Pattern Recognition*. 44 (2011), 1958-1968