VideoMR: A *Map and Reduce* Framework for Real-time Video Processing

Benjamin-Heinz Meier
Hasso-Plattner-Institute
University of Potsdam, Germany
benjamin-heinz.meier@student.hpi.de

Matthias Trapp
Hasso-Plattner-Institute
University of Potsdam, Germany
matthias.trapp@hpi.de

Jürgen Döllner
Hasso-Plattner-Institute
University of Potsdam, Germany
juergen.doellner@hpi.de

ABSTRACT

This paper presents VideoMR: a novel map and reduce framework for real-time video processing on graphic processing units (GPUs). Using the advantages of implicit parallelism and bounded memory allocation, our approach enables developers to focus on implementing video operations without taking care of GPU memory handling or the details of code parallelization. Therefore, a new concept for map and reduce is introduced, redefining both operations to fit to the specific requirements of video processing. A prototypic implementation using OpenGL facilitates various operating platforms, including mobile development, and will be widely interoperable with other state-of-the-art video processing frameworks.

Keywords
map and reduce, video processing, real-time, bounded memory

1 INTRODUCTION

Modern video processing has been shifted from post-processing to real-time systems, applications, and techniques. The reason for this development is the application of graphic processing units (GPUs) for massive data parallel tasks besides rendering. Video processing in the context of this paper is defined as part of signal processing, which applies filters and transformation on video sequences and frames [15].

While current video processing frameworks offer the opportunity to program own GPU-based plug-ins for real-time video processing [12] or implement own GPU-based frameworks [16], a developer still has to deal with the constraints of the GPU memory management and its limitations. Therefore, a developer is required to have a deep understanding of the functionality of the GPU to program own real-time filters. Still, for general GPU programming tasks there are frameworks which have a much more suitable level of abstraction, such as the map and reduce framework *MARS* from He et al. [4]. But those focus on general map and reduce tasks and not on video processing itself. Thus, this paper introduces a redefinition of the map and reduce concept, which is more suitable for video processing using implicit GPU programming. This facilitates GPU programming without a deep technical understanding of the underlying hardware with less lines of code. Our approach is a new interpretation of the map and reduce implementation for *implicit parallelism* on cluster based systems presented by Dean and Ghemawat [3].

Moreover, to ensure efficient usage of memory, which is bounded to certain device-specific limits, such as the memory in embedded or mobile devices, this paper introduces the concept of pre-allocated *ringbuffers* for the video data handling. Avoiding dynamic reallocation of memory and inspired by the disruptor implementation in the *LMAX Disruptor* [13]. To summarize, this work makes the following contributions:

1. it presents a concept for bounded and transparent handling of GPU memory for video processing,
2. a novel concept of a map and reduce framework for real-time video processing redefining both map and reduce to be more suitable for video processing and a bounded memory concept,
3. an interoperable implementation of the presented concept based on OpenGL [10].

The remainder of this paper is structured as follows. Section 2 discusses the fundamental ideas of the map and reduce concept as well as implicit parallelism in functional programming for data parallel tasks in video processing. The idea of using a *ringbuffer* for bounded memory allocation and the combination of both concepts into an efficient framework for real-time video processing called *VideoMR* is presented (Section 3). Section 4 briefly describes a prototypical implementation based on OpenGL [10] which is widely used on all platforms including mobile devices. Furthermore, Section 5 demonstrates and explains the framework by means of a move detection filter. Section 6 discusses the performance, code reduction, and limitations of the prototypical implementation (Section 6), followed by the benefits of bounded memory allocation and implicit parallelism for video processing tasks and a generalization of the introduced concept for different types of data streams (Section 7).

2 RELATED WORK

The idea of using *map* and *reduce* in functional programming languages such as Haskell [5] is not new and has been used in large-scale systems for almost a decade. Loidl et al. [6] have shown that an implicit parallelization of functional code in languages such as Glasgow Parallel Haskell (GpH) [14], as extension of Haskell98 [5], can be efficient while reducing programming overhead for the parallelization task.

This concept was further developed by Dean and Ghemawat [3], focusing on a definition of the *map* and *reduce* principle known from the functional paradigm for cluster systems, hiding the details of parallelization, load balancing, and other tasks of cluster-based systems. The computation is therefore divided in a *map* and *reduce* step, which have to be defined by the developer:

$$\begin{aligned} map(k_1, v_1) &\rightarrow \quad list(k_2, v_2) \\ reduce(k_2, list(v_2)) &\rightarrow \quad list(v_2) \end{aligned}$$

The *map* function receives an input *key/value* pair (k_1, v_1) and emits a list of *intermediate key/value* pairs $list(k_2, v_2)$. Those will be sorted into lists of values related to a specific key $(k_2, list(v_2))$ and passed on to a *reduce* function. According to the key, the *reduce* function merges the list of values $list(v_2)$ to a smaller list. The size of the output is typically one or zero. Functions defined this way can be automatically parallelized by the library. Compared to explicit parallel computing models [7], the overhead in programming time has been significantly reduced.

A GPU-based implementation is presented by He et al. [4]. GPUs, originally developed for rendering purposes only, are used since 2002 for general computing tasks [9] as well. The idea behind this is that GPUs perform very well on data parallel tasks, because

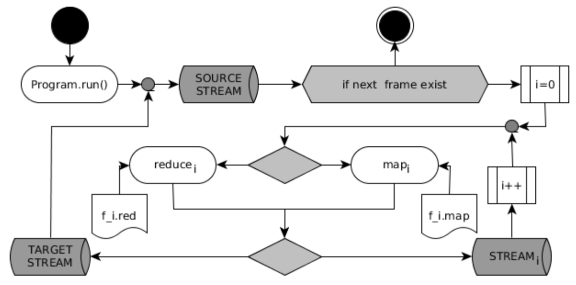


Figure 1: Overview of the work flow of *VideoMR*. As long as frames can be loaded from a source stream they are processed by the map and reduce operations, defined by the corresponding program, and finally pushed to an output stream.

they have been designed to render high amounts of vertices in parallel. Therefore, they have multiple computation units, which can perform the same task on different data in parallel. Nevertheless, porting *map* and *reduce* to a GPU can still be improved, by taking into account that the dynamic memory model, required for the *intermediate key/value* pairs, is not preferable for GPUs. Furthermore, the specifics of video processing can be used to define an own *map* and *reduce* concept for real-time video processing using the pixel coordinates of each frame as implicit key.

To support a static bounded memory concept, a data structure that can handle massive data is preferred. An efficient approach has been introduced by Thompson et al. with a *ringbuffer* concept called *disruptor* as an alternative for queues in the *LMAX Disruptor* framework [13]. This framework addresses the problems occurring in massive financial data exchange. Because of the *ringbuffer* structure, memory has to be allocated only once and is bounded by the total size of all *ringbuffers*. Moreover, memory can be allocated in advance and reused during processing.

3 CONCEPTUAL OVERVIEW

VideoMR combines the ringbuffer concept for bounded memory allocation and a redefinition of the map and reduce paradigm, to fulfill both: the requirements of modern GPUs and the requirements of real-time video processing. In contrast to existing map and reduce frameworks for GPUs, VideoMR does not focus on transferring map and reduce implementation to the GPUs, but instead on implementing a novel interpretation for video processing.

With respect to both concepts, a *VideoMR program* can be defined using a combination of ringbuffers denoted as *streams*, connected through map and reduce *operations*. A specific *source stream*, e.g., a video loader or generator, passes frames to the first map or reduce operation as long as new frames are available. The cor-

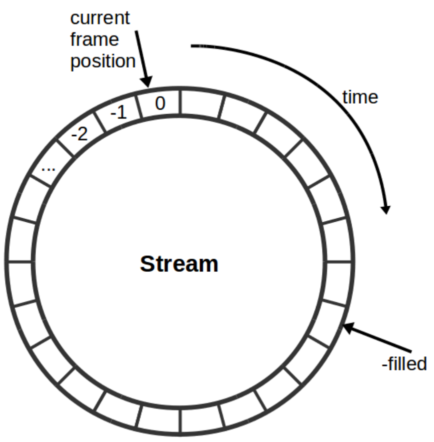


Figure 2: This figure shows the implementation of streams using a ringbuffer. Continuously, the ringbuffer is filled with input video frames. The index indicates the time steps that have passed with respect to the current frame. One pointer is indexing the current frame. Another indicates whether the ringbuffer is full and therefore the next frame will overwrite the oldest one in the stream.

responding operations are processed and executed until new streams are connected. If a target stream, such as a *display* or *writer* stream, is reached, the program restarts for the next frame. This process is summarized in Figure 1 (previous page).

Memory-bounded Streams

The basic data structure, implemented based on the disruptor concept [13], is the *stream*. A stream is a finite sequence of frames. For efficient memory usage for every stream a ring-buffer of k frames is allocated in video memory. Further, a *pointer* to the current frame is initialized and the *counter* for the number of frames in the stream is set to zero. A stream, therefore, represents the current sub-sequence of frames of a video (Figure 2). This temporal information of the streams distinguish video processing from image processing.

Map and Reduce for Video Processing

Initialized *streams* instances can be connected by defining *map* or *reduce* operations (Figure 3) with an input stream s_{in}, and output stream s_{out}, as well as a set of side streams $\{s_0, s_1, \ldots, s_{n-1}\}$. Both functions are defined in the same way to be easily combinable:

$$\begin{aligned} map(s_{in}[, s_0, s_1, \ldots, s_{n-1}], func.map) &\quad \rightarrow s_{out} \\ reduce(s_{in}[, s_0, s_1, \ldots, s_{n-1}], func.red) &\quad \rightarrow s_{out} \end{aligned}$$

The developer can now implement own map and reduce functions by passing definitions in a `func.map` or `func.red` file, denoted as code *snippet*, to the operation. Examples of such code snippets are shown in Section 5.

By implementing these operations, a developer can access the input stream s_{in} and has to write the result to the output stream s_{out}. The side streams $s_0, s_1, \ldots, s_{n-1}$ can be used to pass information from earlier processing steps or other constant application data to the specific operation. The concept of a key/value pair of the original map and reduce concept is replaced by the fundamental approach that a pixel position itself serves directly as key for the respective pixel value. Therefore, a key/value pair consists of the tuple $(position\,x, position\,y)$ as the key, and the emitted value would represent the result of an operation on those pixel in time. Still, this approach is not directly comparable to the original idea, because the introduced map and reduce concept is derived from the general concept of map and reduce in functional programming and therefore does not require any keys. However, the general concept, which both approaches share, is that the map function is applied several times to different input data and the reduce function is aggregating those data.

Map Operation Interface

For the *map operation* (Figure 3 and 5), a developer has to define a function that is applied to every pixel of a frame of the output stream s_{out}. The frame is subsequently pushed as first frame to s_{out}. Every stream can be accessed using a *stream structure* representing meta data of the streams such as width, height, and size. To retrieve the position of the pixel to process (serving as key in the operation), the developer has to call $vmr\_getPosition()$.

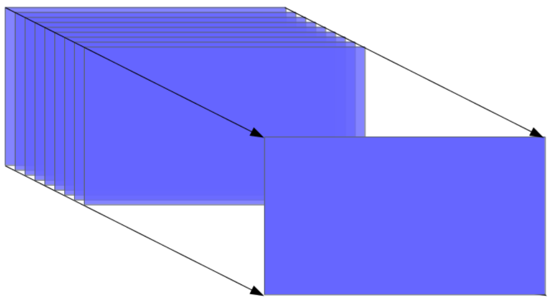
The data at the respective position in the input stream or a side stream can be accessed by the function $vmr\_getStreamDataX(pos,\ time)$. Here, X represents a placeholder for 'In' for s_{in} or the number$i$ of the side stream $(s_0, s_1, \ldots, s_i, \ldots, s_{n-1})$. The function parameter *pos* defines the pixel position and *time* the temporal difference between the frame and the current frame.

To write to an output position, the developer has to call $vmr\_emitToStreamOut(pos,\ pixel)$ explicitly, or, to write to a side stream: $vmr\_emitToStreamX(pos,\ pixel)$ respectively.

Reduce Operation Interface

To implement a *reduce operation* (Figure 3 and 5), the developer is required to define a function that is called for an intermediate frame. This frame is half the width and height of the preceding intermediate frame. During processing, this function is repeatedly called as long as the resulting frame is larger than the least basic resolution of 1×1, or $vmr\_emitToStreamOutAndBreak(pos, pixel)$ has been called.

To retrieve the data from a preceding intermediate frame, the developer calls the $vmr\_getOldData(pos)$ function. To emit the resulting pixel information, the function $vmr\_emitNewData(pos, pixel)$ has to be called. For an efficient and bounded memory allocation, two intermediate frames are allocated per reduce operation in advanced and used alternated as target or source – so called *ping-pong processing*. Especially for the computation of image or video metrics (e.g., the mean color of a frame), the reduce function is repeated until a minimal frame size of 1×1 is reached. Nevertheless, to reduce the resolution of a frame to a desired size (resolution), the developer can call $vmr\_emitToStreamOutAndBreak(pos, pixel)$.


Map: (Stream$_{in}$ [, Stream$_0$, ... , Stream$_{n-1}$]) → Stream$_{out}$

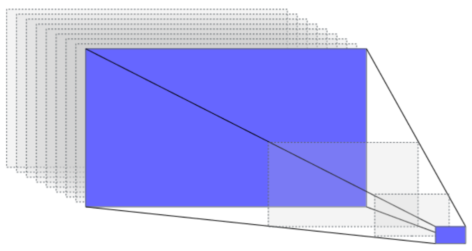

Reduce: (Stream$_{in}$ [, Stream$_0$, ... , Stream$_{n-1}$]) → Stream$_{out}$

Figure 3: Illustration of the general concept of map and reduce operations for video data. The top figure illustrate a map operation that executes its corresponding function for every pixel of the output frame. The bottom figure shows a reduce operation, where a neighborhood of four pixels is merged until the loop is terminated. Afterwards, the resulting frame is emitted to the output stream.

4 IMPLEMENTATION

The presented concept is prototypical implemented using OpenGL [10]. This application programming interface (API) is platform-independent, open source, and supported on most mobile devices. Moreover, OpenGL offers *compute shader*, a generalized interface to GPU programming and is, therefore, the ideal choice for a prototypical implementation.

The stream data structure is implemented using *shader storage buffer objects (SSBOs)*. These have two major advantages: (1) they can be as large as the GPU memory and (2) they are writable at random access. For the implementation of the map and reduce operations compute shader are used. This is a shader type introduced to implement general purpose GPU (GPGPU) operations. We rely on, but are not limited to, the OpenGL Shading Language (GLSL) for operation implementation.

The map and reduce operation are implemented using wrapped compute shader. The discussed API functions are automatically generated for every shader and the streams are automatically passed by the program. The *SSBOs* will be bound to the shader in the beginning of the program run-time.

The basic data structures of VideoMR are implemented using object-oriented design based on GLObjects [1], an object-oriented wrapper for OpenGL. This reduces the code size of the library and helps to extend the framework later on by applying the concepts of the object-orientated paradigm. The framework comprises *core* and *optional* classes (Figure 4), because libraries used for loading or displaying may not be supported or required in every run-time environment. An overview of the functions accessible in a map or reduce operation is shown in Figure 5.

5 APPLICATION EXAMPLE

Discussing a simple move detection program will facilitate an understanding of our implementation of VideoMR and the usage of map and reduce operations for video processing in general.

Assuming a constant camera position, movement in a video can be detected by finding a peak in the first order derivation of the video stream. Thus, a discrete approximation of the derivation is the difference of a pixel over time. If this difference is larger than the mean color of the current frame, movement can be assumed, otherwise not. To reduce noise, this movement has to be detected in at least two subsequent frames at the same position.

Listing 1: Map and reduce main program.

```
// init program
auto prog = std::make_shared<vmr::GlfwProgram
    >();
```

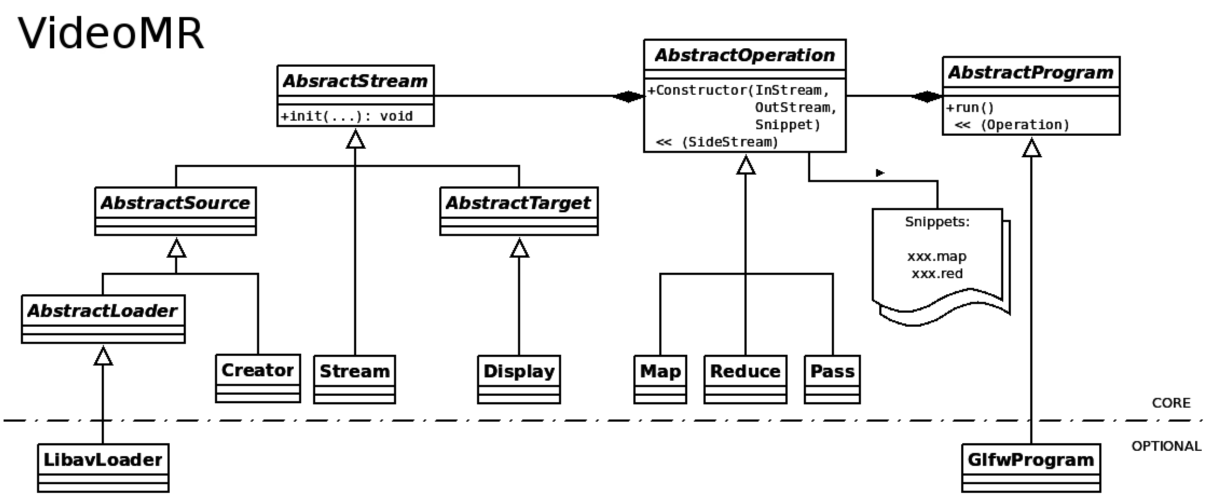


Figure 4: The overview shows the Unified Modeling Language (UML) class structure of the VideoMR framework. Every abstract operation is defined on input, output, and side streams. The snippets contain the source code for a particular operation. Therefore, a program contains multiple operations.

```
// init video loader stream
auto source = std::make_shared<vmr::
    LibavLoader>("./example.mov");
// set the size of the stream to 3
source->init(3);

// mean
// init the result stream for the mean,
// without an explicit init call
// the size will be the size of the
// input stream of the operation
auto mean = std::make_shared<vmr::Stream>();
// init the mean reduce operation and
// add the source stream as input and
// the mean stream as output
auto meanRed = std::make_shared<vmr::Reduce>(
    source,mean,"./mean.red");

// move detection
// init the result stream for the move
// detection, without an explicit init call
// the size will be the size of the
// input stream of the operation
auto move = std::make_shared<vmr::Display>();
// init the move map operation and add
// the source stream as input and the
// move stream as output
auto moveMap = std::make_shared<vmr::Map>(
    source,move,"./move.map");
// add the mean as side stream
*moveMap<<mean;

// setup program and add the operations to it
*prog<< meanRed
     << moveMap;

// run program
prog->run();
```

The code shown in Listing 1 initializes the program. Afterwards, a video loader is initialized and connected with a *reduce* operation to compute the average color of each frame. The output stream of this operation is, together with the loader, connected to a *map* operation computing the actual movement detection, receiving the mean as side stream. The resulting stream is displayed subsequently. Both operations, *map* and *reduce*, receive a file with the concrete implementation as argument.

Listing 2: Exemplary implementation of a mean value computation (mean.red)

```
// get position of current thread
ivec2 pos = vmr_getPosition()*2;

// get four pixel neighbourhood
vec3 c = vmr_getOldData(pos);
pos.x += 1;
c += vmr_getOldData(pos);
pos.y += 1;
c += vmr_getOldData(pos);
pos.x -= 1;
c += vmr_getOldData(pos);
c/=4;

// write result to current position
vmr_emitNewData(vmr_getPosition(), c);
```

The details of the *reduce* operation file is shown in Listing 2. In every reduce step, a four pixel neighborhood is summarized and divided by four. This computes the local average until only one pixel is left, which then serves as the global mean. If the intermediate size equals a frame resolution resolution of one, the result is automatically pushed to the output stream.

Listing 3: move.map

```
// get position of current thread
ivec2 pos = vmr_getPosition();

// get data from last three frames
vec3 c;
```

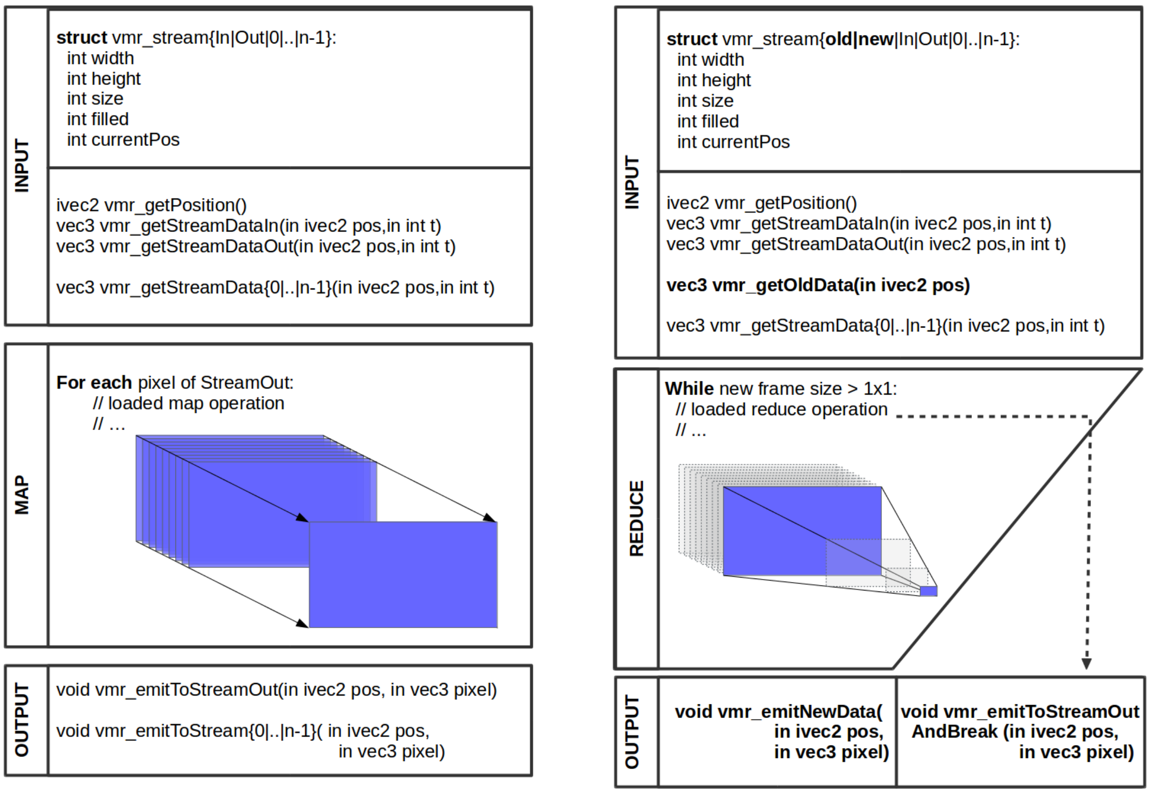

INPUT
struct vmr_stream{In|Out|0|..|n-1}:
int width
int height
int size
int filled
int currentPos
ivec2 vmr_getPosition()
vec3 vmr_getStreamDataIn(in ivec2 pos,in int t)
vec3 vmr_getStreamDataOut(in ivec2 pos,in int t)
vec3 vmr_getStreamData{0|..|n-1}(in ivec2 pos,in int t)
MAP
For each pixel of StreamOut:
// loaded map operation
// ...
OUTPUT
void vmr_emitToStreamOut(in ivec2 pos, in vec3 pixel)
void vmr_emitToStream{0|..|n-1}(in ivec2 pos, in vec3 pixel)
INPUT
struct vmr_stream{old|new|In|Out|0|..|n-1}:
int width
int height
int size
int filled
int currentPos
ivec2 vmr_getPosition()
vec3 vmr_getStreamDataIn(in ivec2 pos,in int t)
vec3 vmr_getStreamDataOut(in ivec2 pos,in int t)
vec3 vmr_getOldData(in ivec2 pos)
vec3 vmr_getStreamData{0|..|n-1}(in ivec2 pos,in int t)
REDUCE
While new frame size > 1x1:
// loaded reduce operation
// ...
OUTPUT
void vmr_emitNewData(in ivec2 pos, in vec3 pixel)
void vmr_emitToStreamOut AndBreak (in ivec2 pos, in vec3 pixel)

Figure 5: Overview of map and reduce operations and their interfaces. The left figure illustrates how to program an own map operation, access the input data, and emit to the output stream. The right figure explains the same sequence for the reduce operation.

```
vec3 c1 = vmr_getStreamDataIn(pos,  0);
vec3 c2 = vmr_getStreamDataIn(pos, -1);
vec3 c3 = vmr_getStreamDataIn(pos, -2);

// compute difference
float diff1 = abs(c1.r-c2.r)+abs(c1.g-c2.g)+
     abs(c1.b-c2.b);
float diff2 = abs(c2.r-c3.r)+abs(c2.g-c3.g)+
     abs(c2.b-c3.b);

// get mean from side stream
ivec2 pos2 = ivec2 (0,0);
vec3 meanC = vmr_getStreamData0(pos2,0);
float aveMean=(meanC.r+meanC.g+meanC.b)/3;

// compare with mean
if (diff1>aveMean && diff2>aveMean){
    c = vec3(255, 255, 255);
} else {
    c = vec3(0, 0, 0);
}

// write result to current position
vmr_emitToStreamOut(pos,c);
```

The algorithm for the move detection used in the *map* operation shown in Listing 3 is a special version of *differential images*, described by Collins et al. [2]. First, the pixel value of the current frame and the two preceding frames are computed. Afterwards, the difference between the first and the second frame as well as the difference between the second and the third frame is computed. If both differences are larger than the mean color of the step before, the emitted pixel is set to white, otherwise to black. The basic idea is, that noise in two different frames is visible in two different positions and therefore filtered using this computation.

The result of this operation for a single frame of a video of an animated fractal set is shown in Figure 6. While the right side of the figure shows the frame source, the left part shows the movement-detection filter applied. Both, the moving borders and also the details of the fast changing inner structure of the fractal can be detected.

6 RESULTS AND DISCUSSION

The prototypical implementation of VideoMR and the example shown in Section 5 demonstrate that efficient GPU programming is possible without explicitly taking care of memory handling or parallelization using the presented map and reduce concept.

Code Reduction

Furthermore, the code can be reduced in comparison with a pure implementation using GLobjects

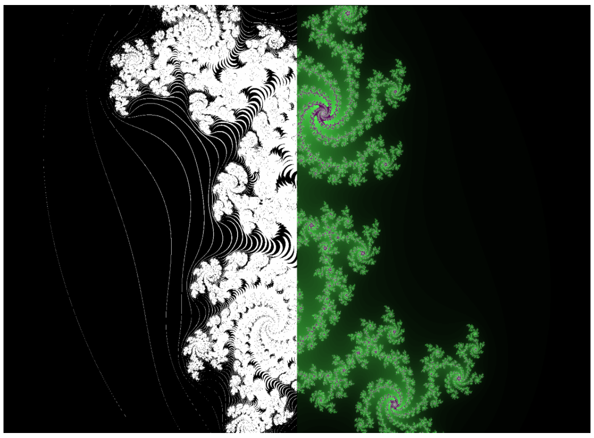


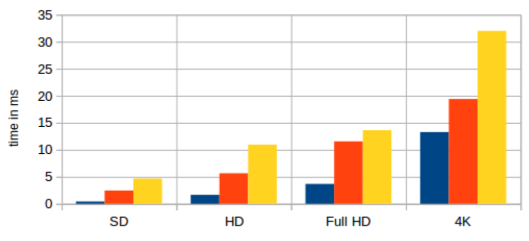
Figure 6: This frame shows on the right side an animated fractal set and on the left side the move detection filter applied for that fractal.

and OpenGL. Referring to the movement detection example in Section 5, the memory management and program setup requires 392 lines of code, the reduce operation 109, and map operation 70 (571 in total). In this example, less than 50 lines are required using VideoMR, a reduction by a factor of 10.

Performance Evaluation

The performance of the prototypical implementation has been computed for four different resolutions: SD (720×576), HD (1280×720), Full HD (1920×1080), and 4K (3840×2160). Figure 7 shows the runtime performance according to these resolutions for a map and reduce program using a single source and target stream with a single map operation, a single reduce operation, as well as a complex example with two map and a single reduce operation.

In video processing, real-time processing can be defined by achieving more than 24 frames-per-second



	SD	HD	Full HD	4K
🟦 map	0.4426 ms	1.6692 ms	3.6844 ms	13.2744 ms
🟥 reduce	2.4658 ms	5.6576 ms	11.563 ms	19.4006 ms
🟨 complex	4.6846 ms	10.958 ms	13.6282 ms	31.9996 ms

Figure 7: Performance for different video resolutions in milliseconds. Displayed are different resolutions, applied to a single map operation, a single reduce operation, and a complex example containing two map and a single reduce operation.

(FPS). This is approximately the sampling rate of the human eye, i.e., maximal 40 ms per frame. Therefore, real-time processing can be achieved on a Quadro K1000M GPU total (dedicated) video memory 2048 MB in the complex example running at 4K resolution (Figure 7). Still – if not required for displaying the result – the OpenGL rendering context can slow the processing down.

Memory Boundaries

To compute the GPU video-memory usage the following equations can be used:

$$\begin{aligned} usage(Stream) &= Stream.size \cdot Stream.width \\ &\quad \cdot Stream.height \cdot 3Byte \\ usage(Map) &= (count(Streams) \cdot 5 + 2) \cdot 4Byte \\ usage(Reduce) &= (count(Streams) \cdot 5 + 4) \cdot 4Byte \\ &\quad + Stream_{in}.width \cdot Stream_{in}.height \\ &\quad \cdot 3Byte \cdot 2 \end{aligned}$$

Limitations and Improvements

However, some operations in video processing are not parallelizable and therefore not programmable with map and reduce. These classes of problems can be approached within VideoMR by introducing a specific implementation of an explicit operation that copies the data transparently to the main memory and can then be programmed with a serial approach.

Also the concept of frames as an array of red, green, and blue values is a limitation for modern 2.5D or 3D video data. Thus, in future work a general concept of n-dimensional buffers instead of frames will be used. Frames then can be handled defining a $frame[width][height]$ as a $buffer[width][height][3]$. This makes it also possible to handle keyboard input or the sound of a video and other data as streams.

Moreover, the current limitation to $8Bit$ values per channel is not preferable for using other data than images or videos. Thus, the extension with template classes to decide which main data type to use should be added in future work. Still OpenCL [11] and CUDA [8] are rarely supported in embedded and mobile environments, but future implementation should also consider using them, because they are not depending on a render context that possibly impacts runtime performance, which is, for read and write operations, not required.

While this paper shows the suitability for real-time video processing, comparison with other map and reduce frameworks will be part of future research. The reason for that is that current benchmarks for map and reduce frameworks are not focusing on video processing tasks and therefore a suitable one has to be developed before.

7 CONCLUSION

This paper presents a concept for transferring existing map and reduce processing metaphors to the domain of video processing using GPUs. It describes a prototypical implementation based on OpenGL and the OpenGL Shading Language. This proof-of-concept demonstrates the efficiency of map and reduce for modern real-time video processing applications. Implementations can be performed using less lines of code with transparent memory handling. Furthermore, the disruptor concept of ringbuffers offers the opportunity to implement video processing on systems such as mobile devices, where memory is a limited resource or the access is restricted by the operating system itself.

To summarize, current video processing frameworks such as *Gstreamer* [12] focus on transparently using video filters and effects. This enables developers to use existing filters, but gives less support for programming own filters based on GPU programming languages. In contrast, *VideoMR* focuses on a programming paradigm based on redefined *map* and *reduce* strategies, allowing to develop own filters for GPUs that are fast to write using implicit parallelism concepts designed for video processing. A lightweight implementation with *OpenGL* [10] ensures the portability and interoperability with existing frameworks. Moreover, the bounded memory handling fulfills the requirements of mobile development with limited or bounded memory resources.

8 REFERENCES

[1] GLObjects. `https://github.com/hpicgs/globjects`. Accessed: 2015-02-04.

[2] Robert Collins, Alan Lipton, Takeo Kanade, Hironobu Fujiyoshi, David Duggins, Yanghai Tsin, David Tolliver, Nobuyoshi Enomoto, and Osamu Hasegawa. A system for video surveillance and monitoring. Technical Report CMU-RI-TR-00-12, Robotics Institute, Pittsburgh, PA, May 2000.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[4] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[5] Simon P. Jones, John Hughes, and Lennart Augustsson. *Haskell 98: A Non-strict, Purely Functional Language*. 1999.

[6] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, Á J. Rebón, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16(3):203–251, September 2003.

[7] Kato Mivule, Benjamin Harvey, Crystal Cobb, and Hoda El-Sayed. A review of cuda, mapreduce, and pthreads parallel computing models. *CoRR*, abs/1410.4453, 2014.

[8] NVIDIA. *NVIDIA CUDA Programming Guide*. NVIDIA, The address of the publisher, 2.3 edition, 2009.

[9] Matt Pharr. *Part IV - General-Purpose Computation on GPUs: A Primer*. GPU Gems 2. Addison-Wesley Publishing Company, 2005.

[10] M. Segal and K Akeley. *The OpenGL Graphics System: A Specification*. Silicon Graphics Inc., 4.4 edition, 2014.

[11] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[12] W. Taymans, Baker S., A. Wingo, S. Bultje, and S. Kost. *GStreamer Manual*, 2014.

[13] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. *DISRUPTOR: High performance alternative to bounded queues for exchanging data between concurrent threads*. LMAX, 2011.

[14] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[15] Yao Wang, Joern Ostermann, and Ya-Qin Zhang. *Video Processing and Communication*. Prentice Hall, 2002.

[16] Rubin Xu. A gpu-enabled real-time video processing library, part ii, computer science tripos, trinity college, 2010.

ACKNOWLEDGEMENTS

This work was funded by the German Federal Ministry of Education and Research (BMBF), as part of the InnoProfile Transfer research group "4D-nD-Vis" (`www.4dndvis.de`).