Development of a GPGPU Video Encoding Server Application in a Multi-GPU environment

B.Eng. Christian Kehl * **B.Eng. Christian Froh** **

* *Uinversity of Applied Sciences Technology, Business and Design, Wismar, 23952 Germany (e-mail: Christian.Kehl@gmx.net).*
** *Uinversity of Applied Sciences Technology, Business and Design, Wismar, 23952 Germany*

Abstract: Since the release of the first OpenCL-version, an emerging interest porting highly parallel, highly complex tasks to the GPU exist in many computing branches. One of the most profitable branches within GPGPU computing is image- and video processing. While being used for development of new desktop software based on OpenCL, present online video services are not using this technology so far. In times of low-performance, small-format computers like netbooks, display workstations and handhelds, the integration of GPGPU computing in multimedia server applications can be a significant push for web startups, gathering new users and markets. Therefore, the department of multimedia engineering at the University of Wismar has formed a small group to create a GPU-based video processing service on prototype level. It has been created a C-written console server application based on OpenCL and OpenCV for fast video encoding and manipulation. The server application is controlled by a Silverlight RIA with modern layout as well as client-side video player and key frame extractor. Measurements have shown the significant performance advantages that prove this application to be a pointer to the right direction of server-side video processing.

Keywords: Multi-GPU, GPU Video Processing, OpenCL

Fig. 1. Badaboom: NVIDIA desktop example application for GPGPU Video processing

1. INTRODUCTION

In present time, videos are commonly encoded by programs that are using the CPU for computation and data reduction. Because each frame can be processed independently, video processing can be parallized very good. Since the development of the GPGPU technology, it is possible to let the GPU do all the work. This results in a massive speed-up, which can be seen in various example applications (NVIDIA [2010]). The video software itself is inspired by the approaches of commercial client-side applications, i.e. NVIDIA Badaboom (Fig. 1) and AVIVIO Video Converter.

These applications offer desktop users great benefits of their hardware set. Besides this, the market for small, energy-efficient devices as well as workstation solutions is emerging. These users are not able to benefit from the GPGPU technology because their hardware has too less performance. The majority of them take their software out of `"the cloud"`. This is possible due to RIA [1] and their broad capability set. The significant component of RIA is the corresponding server application. Empowering video encoding server applications with GPGPU technology will give their clients the ability to encode videos faster.

The speed advantage getting by this technology is used in the new application to provide different image processing filters for picture quality enhancement. Methods applied for picture quality enhancement are presented in `"Image Manipulations"` (1.3).

1.1 Video Processing on the GPU

GPGPU technology is implemented in a variety of different APIs and libraries. OpenCL has been chosen for the development of the new server application. Since version 1.1, OpenCL comes with ICD support (Christiopher Cameron [2010]) enabling most functions and applications to run both on NVIDIA-based as well as ATI-based systems. This is advantageous concerning portability and scalability issues. For access to the video image data, OpenCV has been chosen as a first, simple interface for regulated video access. The extracted data are sent to the GPU, where

[1] RIA - **Rich Internet Applications** use internet technologies and provide an intuitive graphical user interface.

B	B	B	B	....
G	G	G	G	....
R	R	R	R	....

Fig. 4. original OpenCV image data pixel byte order

several filters described in `"Image Manipulations"` (1.3) are applied on the video. It is planned to give a choice whether to apply a filter or not in order to let the user give the choice how long a conversion will take.

The software as a whole is controlled by the RIA frontend application. It offers a variety of interaction with the video conversion and encoding backend, as described in figure 3.

1.2 Video Recording with OpenCV

OpenCV is a popular, open image processing library based on C. An extension enables the user to open video files of pre-defined codecs, grab frame by frame and push result images into corresponding containers with OpenCV. Because of the long experience and the clear structure, OpenCV has been used as a first approach to access video data.

Within OpenCV, the frame access is organized sequentially. Pre-defined requirements for each video are:

- no resolution change of pictures within a video
- access to pixel colors results in RGB values
- pixels are ordered in main column order

The memory values are originally stored as 8 bit character values in BGR order. The memory organization is as shown in figure 4.

For flexibility reasons, this arrangement needs to be changed. During the conversion, it is advantageous to do a key frame extraction of the original and converted video.

Internally, OpenCV uses some parts of the FFMPEG C-library. For some reasons not all available FFMPEG-codecs are possible to use in OpenCV, which needs to be concerned in the design of the system.

1.3 Image Manipulations

Regarding the further presented software system, the GPU performance speed-up is used to compute functions for image manipulations and correction for each frame. These effects are separated into the two categories of image enhancement and artistic filters. The computation queue is ordered so that point-wise operations are done before sharpening or smoothening the image. After enhancing the picture quality, the artistic filters are applied.

The majority of filters either operates on luminance values or can be applied only on luminance values to get the desired effect. Therefore, a color space transformation is done to extract the luminance values. The YIQ model has been chosen as temporary destination color space. After applying all effects it needs to be re-transformed. The color space transformation is done on the GPU. The transformation function is:

$$\begin{aligned} Y &= 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\ I &= 0.595716 \cdot R - 0.274453 \cdot G - 0.321263 \cdot B \\ Q &= 0.211456 \cdot R - 0.522591 \cdot G + 0.311135 \cdot B \end{aligned} \tag{1}$$

The first filter applied on a frame is the linear gray scale transformation. It is used to modify overall luminance and contrast. The disadvantage of linearity in this case is the possibility to over-brighten or over-darken the frames. The linear formula is as following (Equation 2):

$$\begin{aligned} &Y' = c2 \cdot (c1 + Y) \\ &-Y_{max} < c1 < Y_{max} \\ &0 < c2 < Z \\ &\text{inside the application:} Z = 10.0 \\ &\text{standard value for c1:} Z = 10.0 \\ &\text{standard value for c2:} Z = 10.0 \end{aligned} \tag{2}$$

The second filter is a non-linear gray scale transformation implemented as Gamma-correction. It modifies contrast and luminance without over-brighten or darken the frame. The following equation (3, Wilhelm Burger [2005]) describes this modification.

$$\begin{aligned} &y = f_\gamma(x) = x^\gamma \\ &for: x = [0...1]; y > 0 \\ \\ &x = \left(\tfrac{Y}{Y_{max}}\right) \\ &for: Y, Y_{max} = [0...255] \\ \\ &y = \left(\tfrac{Y}{Y_{max}}\right)^\gamma \\ &for: y = [0...1] \\ \\ &Y' = y \cdot Y_{max} \\ &for: Y', Y_{max} = [0...255] \\ \\ &Y' = Y_{max} \cdot \left(\tfrac{Y}{Y_{max}}\right)^\gamma \end{aligned} \tag{3}$$

The gray scale transformations is followed by a Gaussian smooth filter. This consists of a 3x3 Matrix with a variable weight of the filter box. The implementation is adapted from an example OpenCL Gaussian Filter. It uses the parallel processors for column-wise parallel execution. Each Kernel computes the filter row-wise in a loop. Therefore, each pixel is read out of the global memory. Then, the present filter output is computed for each pixel of an m-by-m field downwards. Subsequently, each temporary result is taken and computed for the total horizontal output value upwards. Afterwards, each m-by-m field is transposed and the computation kernel starts again. Because of the switch, the horizontal output is taken to compute the total output vertically with the same kernel. Finally, the picture is transposed one last time and the picture is filtered. Figure 5 shows this algorithm.

The execution continues with an edge sharpening filter. This is done by a 3x3 LaPlace-Kernel with variable epsilon. This defines the LaPlace ratio of the resulting image and therefore the strength of sharpen. The implementation is done by modifying an OpenCL example of the 3x3

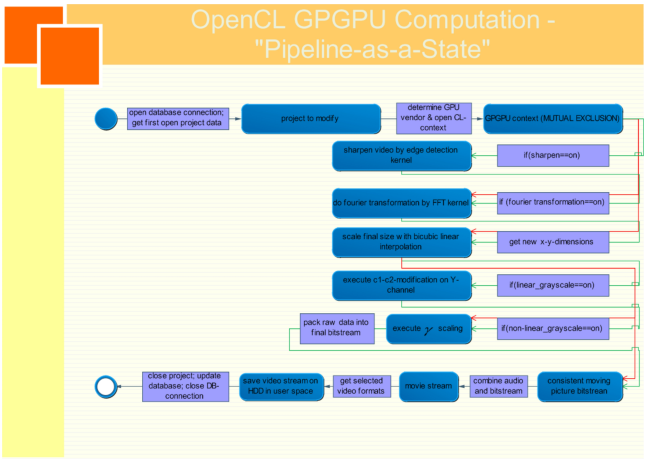

Fig. 2. OpenCL GPGPU Computation - "Pipeline-as-a-State": Shows the general process of video modification on the GPU.

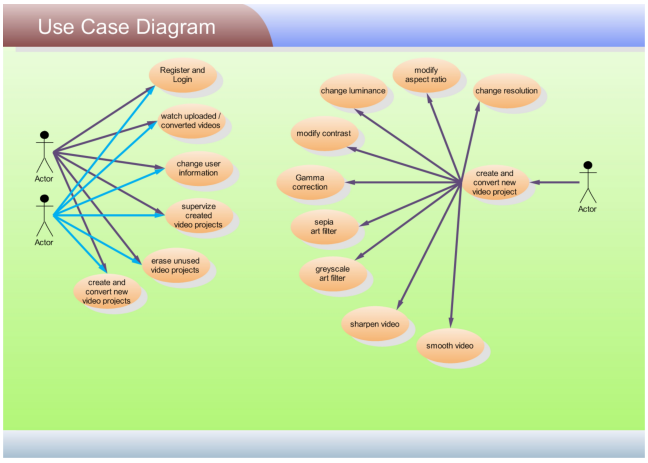

Fig. 3. General use case diagram showing interaction possibilities with the new video software

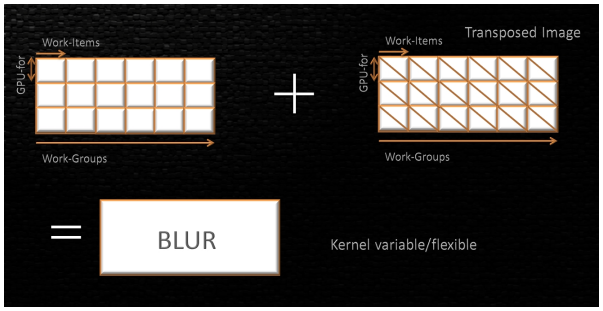

Fig. 5. algorithm description of the GPU Gaussian smooth filter

Gaussian Smooth filter. The filter matrix presented in equation 4 is used.

$$K = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4+e & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{4}$$

After these image correction and enhancement filters, artistic effects are applied to the video. This is done under mutual exclusion. Implemented artistic effects are the transformation into gray scale picture and the sepia transformation. For gray scale video frames the I- and Q-Channel are deleted so that the Y-Channel with luminance information remains. Concerning the Sepia-Transformation, following Matrix is applied to the frames.

$$\begin{aligned} R' &= 0.393 \cdot R + 0.769 \cdot G + 0.189 \cdot B \\ G' &= 0.349 \cdot R + 0.686 \cdot G + 0.168 \cdot B \\ B' &= 0.272 \cdot R + 0.534 \cdot G + 0.131 \cdot B \end{aligned} \tag{5}$$

2. DESIGN

The software is divided into four main groups: database access, main function and video services, GPGPU API and one group for the OpenCL background.

The database access is kept to a minimum. This is possible because the only things to do with the database are getting the clustered information of a project, update a project and register new key frames. The main interaction with the Frontend is done by a `C#` super-server, the database class and the database itself. The main function includes the algorithm shown in figure 6. The GPGPU API is an in-between class taken from Kehl [2010]. It clusters initialization and connection methods for a variety of GPGPU libraries and APIs. For this project only OpenCL is activated.

The OpenCL group is one class. Regarding the complexity of these operations its design fundamentals are presented in further subchapter.

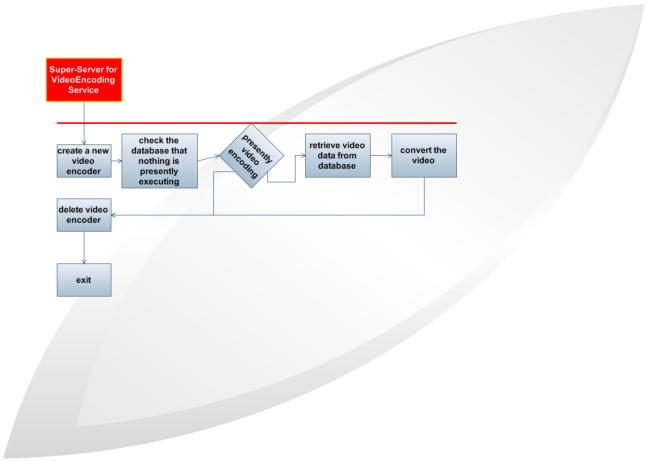

Fig. 6. main-function algorithm

2.1 Combination of OpenCV and OpenCL

The aim of the GPGPU processing is the picture-wise modification with listed effects. Therefore, the image data need to be transferred to the OpenCL class. The original image data are pointers in the main memory, but for processing different side parameters like width, height, spacing and bit depth also need to be present. On purpose to create a flexible, portable software package that can also be compiled as library, there should be introduced a new class or structure for these data. Instead, it has been decided to integrate the IplImage structure in OpenCL. Therefore, a pointer to the present frame and a pointer to the destination frame are forwarded through the GPGPU API to the OpenCL class. The destination frame is an image with the same parameters as the original frame but with empty image data.

The programming interface therefore is designed simple. The GPGPU API has an open interface function to start the encoding process. All necessary parameters for images and effects are given to the interface. The clamping and processing of standard values is done in the OpenCL class. The whole collaboration process is documented in the following diagram (Fig. 7).

2.2 Multi-GPU support

As described in Kehl [2010] combining multiple GPUs at once to fulfill a task is a possible way to speed up computation. In order doing this, the OpenCL system has to be adapted to some extent.

We decided to use multiple cards for rendering each frame as fast as possible. Therefore, the frame needs to be separated into areas of same size. Each part frame of the original is transferred to the corresponding GPU. For easy split-up, the H-LFR[2] mode is suited very well for this task. The algorithm is described in detail by figure 8. A special point is that the resolution also defines the work-size dimension of the GPU task. This should be a power-of-two number. Odd resolution numbers for part frames are leading to unused resources. It can also happen that local and global work-group size doesn't match. This results in uneven divisibility, which is forbidden kernel behavior [Reference OpenCL spec]. For avoiding any conflicts, the video frame size needs to be scaled to a power-of-two dimension in either direction. In addition, if the number of available GPUs is non-power-of-two, the number of actually used GPUs is clamped to the next lower power-of-two number.

After separating the original frame, each part frame is transferred to the GPU. Multiple temporary part frame memory objects need to be created and initialized for effect computation. Each used GPU gets has its own function pipeline which is controlled by a corresponding thread. These are started after GPU memory allocation and compute the effects on the GPU. The main thread is synchronized with these worker threads and starts frame composition after last thread's finish. The workers are dismissed and the temporary memory is cleaned.

2.3 Successive Kernel Strategy

The input for OpenCL class is the image data byte stream. The byte stream has to be transformed, as discussed before (Chapter 1.2). This byte transformation is done before GPU execution by OpenCV. Afterwards, the byte stream is prepared for decomposition or transferred into a GPU texture map.

On execution of the worker thread the effects computation begins with the color space transformation (Chapter 1.3) which is the only operation that cannot be skipped out. Afterwards, the effects are computed in order as presented before (Chapter 1.3). Each filter can be controlled by a minimum of one parameter and each parameter has standard values indicating that no change is done to the frame. If all parameters of a particular kernel are set to standard values, the kernel is skipped and the part frame is processed by a GPU-internal copy operation to the next destination temporary memory. After effect processing a color space transformation back to RGB is performed. The data flow is described in the figure 9.

Each kernel is created and prepared on object initialization. This will be discussed in (Chapter 3.3).

[2] H-LFR - **horizontal line frame rendering** separates the part frame in lines of the same height and the total width

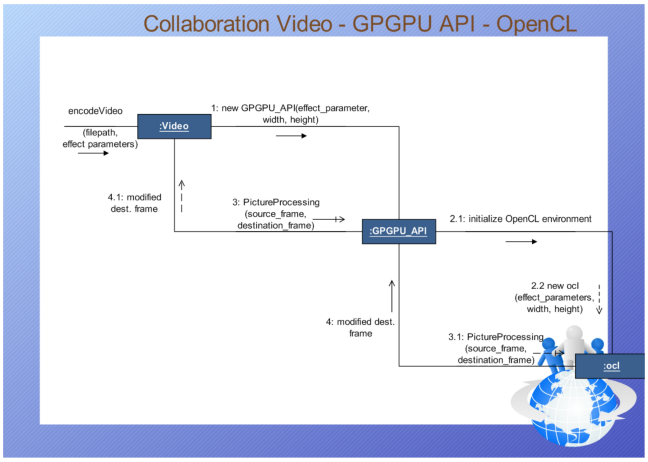

Fig. 7. Collaboration diagram of video, GPGPU API and OpenCL

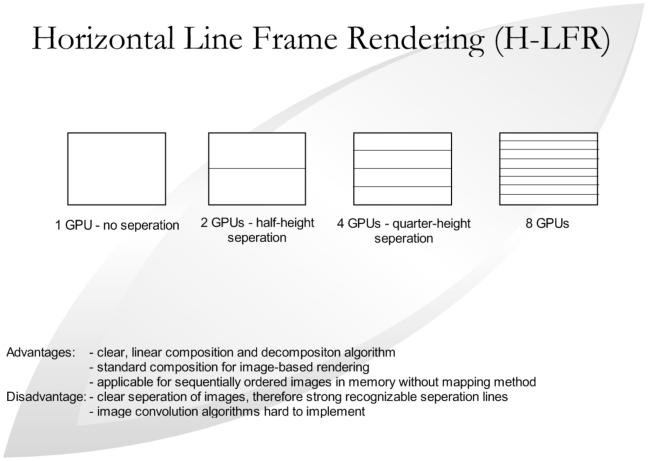

Fig. 8. H-LFR - (de)-composition algorithm

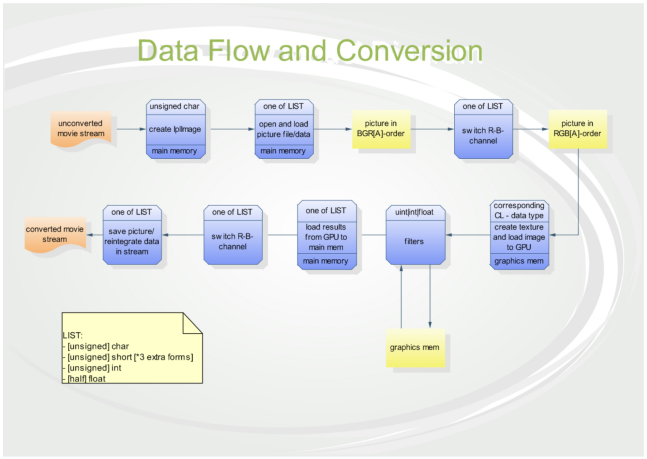

Fig. 9. Data Flow and Byte order Conversion

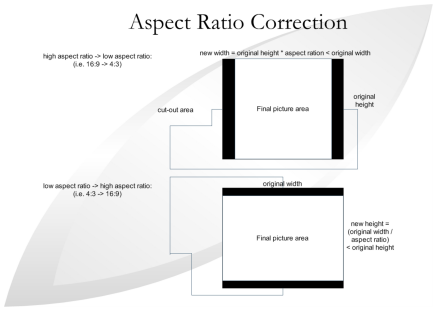

Fig. 10. Data Flow and Byte order Conversion

3. IMPLEMENTATION

The software has been developed stepwise in the order of groups as presented. A pre-implemented MFC database class has been used and adapted for the database connection in the corresponding video database class. For using MFC CDatabase the ODBC driver of the specific database needs to be installed and configured. The video database class is the only proprietary class using MFC. In case of porting the software to a different system it needs to be replaced. This is nevertheless the case because the database itself is the proprietary MS SQL Server 2008.

According to the video class two main functions are implemented. One function is for general GPGPU code testing. It performs image processing of one picture thirty times in a row, take the time and compare the results. The seconds is the video encoding function. Several helper functions have been created to efficiently choose the right codec, perform the key frame extraction of destination videos and modify the aspect ratio in the way described in figure 10.

The GPGPU API class has been modified to forward parameters as shown in figure 7. In addition, the only supported parallel technology is OpenCL, so the other technologies have been disabled.

The main focus is on the OpenCL class, which will be focused in the following subchapters.

3.1 Textures and Byte Streams

As presented in Chapter 2.3 the input data for the OpenCL class is an ordered RGB byte stream which is decomposed into part frames. For further processing, the byte stream can be taken as it is, it can be formed into a pinned / mapped buffer or transformed into a texture.

Textures are complicate to set-up. In addition, textures in the memory can't be swapped with each other or be replaced. Another disadvantage is the limited, supported atomic data types for GPU textures. Usage of textures for the video processing task has been tested. Although the simplest filter on RGB textures works, the system uses one byte stream technique. The disadvantage of hard conversion of two 32 bit type textures and the left-out possibility of fast texture data replacement are the main reasons for this decision. If GPGPU texture techniques are developing like in 3D Shaders, so that the function area is automatically interpolated, textures would be an option in order to perform fast geometric transformations.

Consequently, the byte stream solution is taken. It's used as non-pinned / non-mapped buffer because the memory regions are changing from frame to frame. Pinned Buffers are of use if the buffer address space is not changing because it cannot be moved or swapped. Therefore, speed advantage can be gained by using the maximum transfer rate of the PCI-Express Bus. In contrast, the video frames are dynamically loaded into the main memory, processed, saved on the HDD and then dismissed, so each frame needs individually to be pinned / mapped and de - pinned / demapped. The speed advantage gaining by that is not sure. That's why, pinned buffer technology was skipped.

The part frame image data are transferred to the GPU memory as signed character values. Beginning from the initial- until the end color space transformation, the temporary data are processed as single-precision floating point values.

For skipping effects with standard values, the GPU-internal function for copying buffers is applied.

3.2 Video Memory Structure

Due to the successive kernel strategy, the computation time for each part frame is significantly high. Vectorization of rows of matrices has been taken into account as optimization method to decrease computation time. Vectorization can be used in the following functions:

- color space transformation from RGB to YIQ
- color space transformation from YIQ to RGB
- linear gray scale transformation
- sepia transformation

It has been examined if this technique leads to optimization and, if that's the case, how efficient the speed-up is. As an example RGB-to-YIQ-transformation is described in the following. Vectorization in that case is meant to replace row-wise constant factors with OpenCL-integrated memory vector types (Munshi [2010]). For the color space transformation, these are the transformation factors as seen in then following formula.

$$\begin{aligned} Y &= 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\ I &= 0.595716 \cdot R - 0.274453 \cdot G - 0.321263 \cdot B \\ Q &= 0.211456 \cdot R - 0.522591 \cdot G + 0.311135 \cdot B \end{aligned} \tag{6}$$

The software is developed with OpenCL 1.0 which only supports vector field lengths of one, two and four. Consequently, a vector field length of four was chosen while setting the last component to zero. These vector fields are created in the main memory and on execution transferred onto the GPU as constant vector fields. On the GPU, these values are saved sequentially in GDDR memory, which should lead to faster access times. In addition, color values are also vectorized because these values are accessed lots of times in one kernel. The formula is then replaced with the following one.

$$RC = \begin{bmatrix} Y \\ I \\ Q \\ 0 \end{bmatrix}$$

$$OC = \begin{bmatrix} R \\ G \\ B \\ 0 \end{bmatrix} \tag{7}$$

$$D = \begin{bmatrix} 0.299 & 0.587 & 0.114 & 0 \\ 0.595716 & -0.274453 & -0.321263 & 0 \\ 0.211456 & -0.522591 & 0.311135 & 0 \end{bmatrix}$$

$$RC_{11} = D_{11} \cdot OC_{11} + D_{12} \cdot OC_{12} + D_{13} \cdot OC_{13}$$
$$RC_{12} = D_{21} \cdot OC_{21} + D_{22} \cdot OC_{22} + D_{23} \cdot OC_{23}$$
$$RC_{13} = D_{31} \cdot OC_{31} + D_{32} \cdot OC_{32} + D_{33} \cdot OC_{33}$$

In contrast to the expected performance increase, a slight decrease of ten percent has been measured in a series of six different original pictures with ten passes per picture. Therefore, the present software version uses non-vectorized kernels.

3.3 Constant Kernels

In previous GPGPU software systems the GPU kernels were compiled and build "On-the-Fly", meaning at the point of usage. This, in particular, is a bottleneck of the application. For GPU software with very dynamic kernel execution this is an unavoidable problem, but for the effect processing computation steps and kernel execution order are static. It only needs to be differentiated if a kernel at one positions needs to be executed or not (resulting in memory copy operation). This leads to the possibility of pre-compiled kernels executing on their specific position. This is done by creating, compiling and building them in the initialization routine for the OpenCL class. The kernels and their execution order stay constant for the whole execution cycle. They are dismissed with the destruction operation of the OpenCL class. Their execution call is done for each part frame in the corresponding worker thread.

4. PERFORMANCE MEASUREMENTS

In this chapter a listing of time measurements regarding the video conversion and effect computation can be found.

Generally, considering the time consumption affected by the input video time, the conversion of small videos is relatively high in comparison to long videos. This is the OpenCL initialization trade-off. The OpenCL initialization and Multi-GPU context creation takes 25-50 seconds, depending on overall system load, present graphics load and number of GPUs. Measurement two to four have been taken in times without constant kernels, therefore taking up much more time. The overall conversion time rises dramatically by higher resolution encoding. This is because scaling of frame is done on the CPU with next-neighbor interpolation. This can be reduces with GPGPU technology when scaling and interpolations can be done inside OpenCL. The number of effects computed by the GPU, and therefore the kernel execution time, has significant influence on the total conversion time. Consequently, the effect-skipping possibility strategy is a good way to control the conversion time.

5. FRONTEND CONNECTION

The Silverlight-based RIA Frontend is interconnected by 3 different subsystems. The first subsystem is the backend database connection class. Data are transferred over the database making the implicit communication obvious. The second subsystem, the database management system itself, is responsible for controlled updates of the database entries. For synchronization the status and configuration information are vital. The backend application triggers status changes and determine, by connecting status and configuration, the present conversion status, as described below.

- if there are configurations of one project ready for processing, take the oldest configuration created and convert this
- if there is one configuration of one video project converting at the moment, the GPU is blocked by another instance of the server application; therefore skip momentary execution
- if not all configurations of one project are done, complete the conversion of one project
- if all configurations of all projects are converted be idle

The presented database tables are very important for synchronization and stability, so the database triggers need to react fast for interaction. Taking the GPU as conversion component, the main processor is idle so that the database has sufficient resources to complete the tasks.

The last subsystem is a self-created video service super-server, starting the backend application on demand. This prevents implementing the backend application as polling service using up lots of resources. The super-server also triggers database changes.

6. FUTURE DEVELOPMENT

Although the server is running and converting videos there are some issues to fix, some compatibility and version problems to resolve and some extensions to implement for a first final version.

Concerning the issues database triggers needed to be implemented with work-around methods due to the complex, inconsistent user management within the Microsoft Windows 7 operation system and its system software. Another issue is the codec management and undefined behavior in OpenCV. Although OpenCV can convert only a few codecs (Agam [2006]), the encoding itself has a high failure rate for particular combinations of extension and codec. Moreover, there is an undefined behavior of video encoding in high-resolution frames. This can be fixed by switching to the original FFMPEG library as it will be proposed in further passages.

In the application there exists a whole list of incompatibilities due to the predefined versions of binaries of used APIs and libraries. Due to the underlying 64-Bit technology and the 64-Bit operation system the video encoding software should be a 64-Bit application too. OpenCV as image

Table 1. Performance Measurements

Start Configuration	End Configuration	Time Consumption
MPEG-4; 720x720; 24 fps; 2.3 Mbps; 00:00:33 h	MPEG-4; 1024x768; 24 fps; 50.3 Mbps; all effects; gray	00:01:12 h
MPEG-4; 720x480; 59 fps; 3.6 Mbps; 00:00:09 h	MJPEG; 1280x720; 59 fps; 75.4 Mbps; all effects; color	00:03:14 h
MPEG-4; 720x480; 59 fps; 3.6 Mbps; 00:00:09 h	MPEG-4; 1280x720; 59 fps; 62.1 Mbps; all effects; color	00:00:57 h
MPEG-4; 1280x720; 29 fps; 4.0 Mbps; 00:01:40 h	MPEG-4; 960x720; 29 fps; 44.2 Mbps; LGST[1]; color	00:08:32 h
MPEG-4; 640x480; 25 fps; 1.1 Mbps; 00:21:11 h	MPEG-4; 1280x720; 25 fps; 51.4 Mbps; no effects, gray	00:25:53 h

[1] LGST - linear gray scale transformation

software library is shipped as 32-Bit library only. The used OpenCL NVIDIA Computing SDK is available with 64-Bit libraries. Within Visual C++ it's originally not possible to combine them for several reasons:

- An application compiled for MACHINE:X64 can only inherit 64-Bit libraries, therefore no OpenCV
- An application compiled for MACHINE:X86 can only inherit 32-Bit libraries, therefore not the available, installed NVIDIA Computing SDK libraries

As a solution, the NVIDIA Computing SDK was compiled as a 32-Bit compatibility version. Nevertheless, the 64-Bit C# super-server needs special privileges to run a 32-Bit application as a 64-Bit father process. As possible solution OpenCV can be replaced with another library that is 64-Bit compliant. In addition, a 64-Bit application can also be started by the SQL Server 2008 64-Bit version within a trigger, which opens up the possibility to leave out the super-server subsystem.

Another version incompatibility point is the NVIDIA Computing SDK version 3.0. Some logging- and debugging functions like shrCheckError and shrLogBot have been dismissed in the new SDK version or have been moved to the ocl-library. Therefore, the basic GPGPU API and Multi-GPU OpenCL code is not working with the new version. This led to the usage of the NVIDIA Computing SDK version of February 2010. The code needs to be modified to replace the main server version and to use new capabilities.

This change opens up new extension possibilities. Due to the cloud computing-character of the application, it should be developed a super-server distributing created video project configurations to parallel graphics servers. According to the present OpenCL version, supporting OpenCL ICD, this can also be done in a heterogeneous environment of graphics servers, leading to large scalability.

The usage of FFMPEG would allow the transcoding of videos into nearly every existing video codec, skipping one big limitation of the present application. In addition, it will then be possible to encode and modify the audio channel, giving new use cases and opening up new capabilities for the user. For FFMPEG integration it can be used the C-based library as long as the memory alignment of data is partly compliant with the existing memory management inside the application.

REFERENCES

Gady Agam. Introduction to programming with opencv. electronic, January 2006. URL `http://www.cs.iit.edu/~agam/cs512/lect-notes/opencv-intro/opencv-intro.html`.

Micheal Houston John Kessenich Christopher Lamb Laurent Morichetti Aaftab Munshi Ofer Rosenberg Christiopher Cameron, Benedict Gaster. Opencl khronos icd. electronic, March 2010. URL `http://www.khronos.org/registry/cl/extensions/khr/cl_khr_icd.txt`.

Christian Kehl. Research on the optimization of graphical data processing systems in multi-gpu environments. Bachelorthesis, University of Wismar, March 2010.

Aaftab Munshi. Opencl specification. electrionical, Octobre 2010. URL `www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf`.

Corporation NVIDIA. Cuda zone - the resource for cuda developers, 2010. URL `http://www.nvidia.de/object/cuda_apps_flash_new_de.html`.

Mark James Burge Wilhelm Burger. *Digitale Bildverarbeitung*. Springer, 2005.