Free Viewpoint Video using Open Compute Language (OpenCL)

A G Megharaj,
Sakshi verma,
Amit Gupta
Advanced System Technology,
STMicroelectronics
Greater Noida, India
agmegharaj@gmail.com,
v.sakshi15@gmail.com,
amit.gupta@st.com

Giovanni Visentini,
Davide Aliprandi
Advanced System Technology,
STMicroelectronics
Agrate, Italy
giovanni.visentini@st.com,
davide.aliprandi@st.com

Abstract

Free Viewpoint Video (FVV) is the ability provided to the final user to interactively control the watching position of a scene through a dedicated rendering engine.
This paper aims at describing how this application has been accelerated, parallelizing a reference proprietary C-code implementing the FVV [1] through OpenCL (Open Compute Language), in order to exploit the potentialities of modern GP-GPU systems (General Purpose processor – Graphics Processing Unit).
Specific and customized kernels have been identified and implemented, so that to keep the application' behaviour unchanged in terms of quality, but also achieving a further speed up of about 28%. Performances obtained on GPUs with different complexity profiles have been compared, thus obtaining useful figures in term of speedup and behaviour of kernels requiring large global memory access and strong data dependencies; finally a comparison between cycle counts for the execution on CPU versus GPU have been reported, showing how GPUs can be used for general computation too

Keywords: Free Viewpoint Video, 3D video, OpenCL, GP-GPU, parallel computing, image processing, Multi-viewpoint video

1. Introduction

Free Viewpoint Video gives flexibility in the hands of the viewer to interactively choose the viewpoint in the real time, typically through set top box. To meet such hard requirements only through CPU is quite challenging, hence we use GP-GPU processing which includes both CPU and GPU together.

In our work we use Open Computing Language (OpenCL or OCL) an open, royalty free standard for parallel programming on modern processors, and serial code presented in [1].

This paper is organized as follows. We'll briefly describe the Free Viewpoint Video chain in section II, followed by an introduction to OpenCL in section III. In section IV we describe acceleration achieved using OpenCL and speedup in terms of cycles consumed by algorithms on CPU and GPU. Finally we analyse results and report our conclusions in section V. Section VI gives the direction for future work.

2. Free Viewpoint Video Pipeline

The Block diagram for Free Viewpoint Video rendering chain is shown in Fig. 1 and described below.
Inputs to our view synthesis pipeline are the 2 views (left and right) of the same scene with theirs depth maps and we have to generate the intermediate view at the location specified by the user. The input is YUV: 420 format video stream

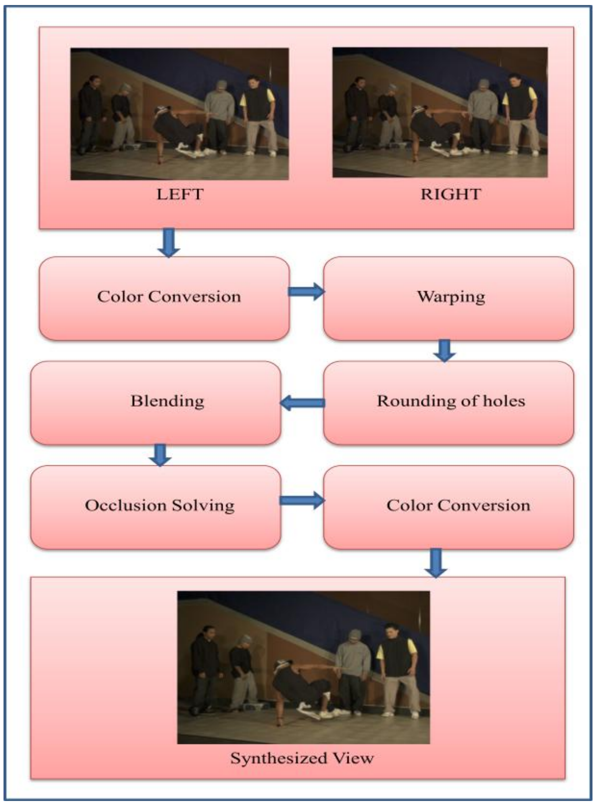

Figure 1 Free Viewpoint Video chain

2.1 Color conversion

YUV color space is chosen in our pipeline as it encodes video taking human perception into account, allowing reduced bandwidth for chrominance components, thereby enabling transmission errors or compression artifacts to be more efficiently masked by human perception than using RGB- representation.

Since the human visual system is more sensitive to variation in brightness than color, a video system can be optimized by devoting more bandwidth to *luma* component than to color (*chroma*) components.
Initially we take the compressed input image YUV: 420 and convert the same to YUV: 444 as the intermediate stages of Free Viewpoint Video pipeline are performed in this format and finally converting the same to YUV: 420 format as output.

2.2 Warping

Warping includes two main functions they are boundary detection and projection.

Before projecting each pixels to target view we need to find foreground and background boundary to enhance the final result by avoiding warping of pixels that can potentially produce artifacts along boundaries (patented solution [2]). Boundary detection is performed by identifying discontinuities in the depth values in the source frame.

3D warping is performed by projecting pixels from source viewpoints to the target viewpoints which includes the following steps [1].

- Project the pixel location into world reference system by using the inverse of source camera's projection matrix.
- Project the world coordinates into the coordinate system of the target camera by using the target camera's projection matrix.
- Convert to homogeneous form to obtain the pixel location in the target frame.
- Finally assign the source intensity to the target pixel location in the synthesized frame.

2.3 Rounding of holes

After projection there will be great quantity of small unfilled areas, often just one or two pixel sized. These gaps are due to integer rounding operation that assigns to the projected pixel position (u,v) within the frame. When the resulting location is at least two units greater than the previous one (located in the same row of the frame), a one pixel sized area remains unfilled [1].
We have implemented a method that, uses certain thresholds of hole size and pixel colour, sets the hole pixels to the average of the pixels located at two extremities of the hole [1].
Once we have projected left and right views we perform blending operation.

2.4 Blending

Blending is the consequence of projection, aiming at merging the contribution of the source views in a unique frame. In general, the unfilled areas in the left projected views are filled in the right projected views, and vice versa.

In our work we are using uniform blending where the projected views are uniformly weighted, but it does not perform well when the number of source views is big [1]. Even after blending there will be unfilled pixels which will be solved within the Occlusion Solving step.

2.5 Occlusion Solving

Even after blending if found unfilled pixel, that pixel is defined as occluded pixel.
In occlusion solving function the basic idea is to fill up the unfilled areas with the background colour.
Given an occlusion, it is imprinted with the colour of the pixel with smallest depth among those surrounding the occlusion itself [1].

3. OpenCL Introduction

Applications today demands very high computing power, but even if modern CPUs support higher frequencies, this is generally not scalable in terms of area and power consumption [3] therefore multi core architectures are becoming very popular especially in the embedded domain. Efficiently exploitation of the processing capabilities of today's multicore devices is not only tricky but also difficult task, there comes the importance of high-level APIs like OpenCL, which API's to easily program and utilize parallel processing capability on multi core CPUs.
GPUs initially were meant for only graphics, but with the latest advances in GPUs and proliferation of APIs like OpenCL GPUs can be used even for general purpose processing other than graphics[4].
OpenCL is so powerful it allows many devices to plug in to the same host, exporting to the user their processing power as a whole. OpenCL provides access to all the resource of the device to extract the inbuilt hardware power [4].

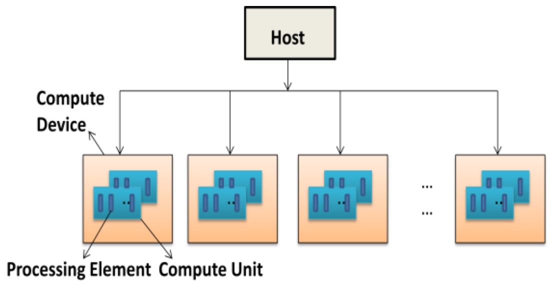

Figure 2 OpenCL view on the hardware [5]

The Typical OpenCL platform is shown in Figure 2, where one host is connected to multiple devices, each device has multiple compute units and each compute unit has multiple processing elements (PE). An OpenCL application runs on the host and submits commands to execute computation on the device. PEs execute single stream of instructions, thereby exploiting data-level parallelism [4].
Kernels can be executed on OCL devices in 1, 2 or 3-dimensional index space called NDRange, with one kernel instance (work item) per index (ID). Each work item executes the same code in SIMD (single instruction multiple data) or SPMD (single process multiple data) fashion. Work items are organized in work-groups and are uniquely identified by Global-ID or Global workgroup ID +Local ID [4].

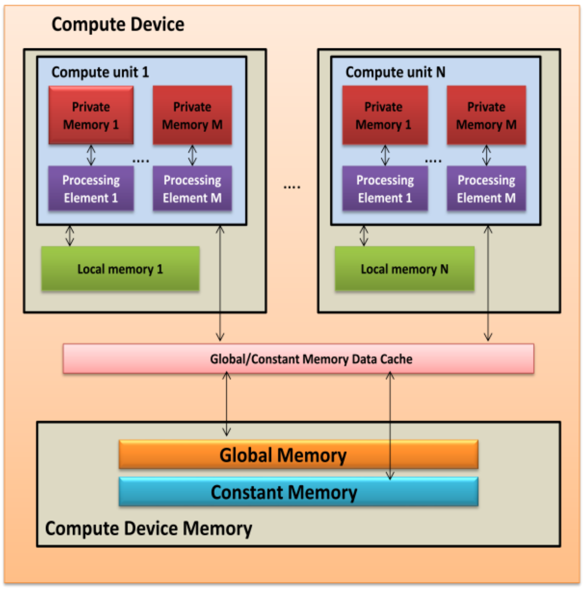

Figure 3 OpenCL view on the Memory [5].

The memory model of an OpenCL system is shown in Figure 3. Here, work items have access to global memory (permits read/write access to all work-items in all work-groups), constant memory (section of global memory that remains constant during execution of kernel), local memory of work group (memory region local to work-group) and private memory (private to work item).Thus OpenCL framework allows applications to use a host and *n* OpenCL devices as single heterogeneous parallel computer system [4].

4. Free Viewpoint Video chain using OpenCL

We profiled the C-reference code on the following CPU: Intel(R) Xeon(R) CPU E5504 @ 2.00GHz. We accelerated the algorithms by using OpenCL, on couple of GPUs to show the performance comparison between different devices: NVIDIA Quadro NVS 295 from now we will be referring to it as GPU1 and GeForce GTX 470 which is a high end GPU, from now we will be referring to it as GPU2. Platform version: OpenCL 1.0.The following table 1 shows the time taken by the various steps (in seconds) by the reference and optimized code.

Algorithm	**Reference (s)**	**Optimized GPU 1 (s)**	**Spee d up**
Color conversion	3.19	3.96	0.80
Warping	10.16	5.66	1.79
Blending	3.36	1.47	2.28
Occlusion solving	0.41	1.05	0.39
Rounding of Holes	0.6	1.39	0.43

Table 1 Free Viewpoint Video – C reference Vs OpenCL (Video: 1024x768, 100 frames) on GPU1

Algorithm	**Reference (s)**	**Optimized GPU2 (s)**	**Spee d up**
Color conversion	3.19	0.0465	68.60
Warping	10.16	0.0719	141.3
Blending	3.36	0.0193	174.0
Occlusion solving	0.41	/	/
Rounding of Holes	0.6	0.55	1.09

Table 2 Free Viewpoint Video – C reference Vs OpenCL (Video: 1024x768, 100 frames) on GPU2

As shown in table there is an overall speedup in FVV chain. The optimizations, algorithmic improvements and results are discussed in the sections that follow.

4.1 Colour conversion

Kernel	**Speed up** (GPU1 Vs CPU)
yuv 420->422	1.54
yuv 422->444	1.30
yuv 444->422	1.08
yuv 422->420	0.96

Table 3 Acquisition kernels GPU1 Vs CPU

The conversions YUV: 420 to the YUV: 444, is bit complicated, as it involves six global memory access per output pixel and also each pixel is read six times.

The conversions YUV: 444 to the YUV: 420 include twelve global memory accesses per output pixel and also each pixel is read twelve times.

This is a big overhead because the access to the global memory cost many number of GPU clock cycles [6], and in a parallel environment we can have collision while different kernels are reading the same location.

Using global memory was two times slower than that of CPU so to overcome this problem local memory was used, which came to be faster. The pattern of loading global memory to local is described below.

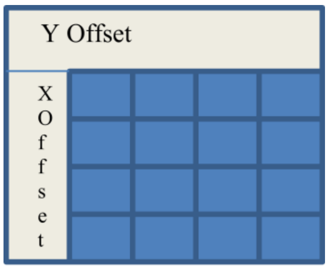

Figure 4 Loading into local memory

The image data was loaded to local memory in general the workgroup size is in powers of two, so we may come across a situation haven't the image size that is a perfect multiple, so we have an offset. This can be calculated as follows, let's take X for rows and Y for number of Columns than

```
Xoffset = X % Workgroup_size
Yoffset = Y % Workgroup_size
```

The blue boxes represent the global data loaded into work group in GPU. If offset is equal to zero there will be normal loading but if the offset is not equal to zero than we skip the offset number of pixels this will be achieved by adding offset to *get_global_id* and *get_local_id* (OpenCL API functions), after the calculation being done on the local memory. In the next phase we load only the offset represented pixels, which were not loaded previously into local memory to first work groups to complete the calculation.

Loading data from global memory to local memory provided speed up but also had an overhead of setting the border in each work group.

The speed of the kernels depends entirely on the local work size, making it less compatible on different devices with different local memory. In fact, if local work size is less, than threads don't share data so using local memory gives no improvement. On the other side, using large local work size reduces the number of parallel threads [7]).

Even after using local memory we were not able to achieve much speed up than that of CPU as these kernels had much memory access and overheads of setting border, but provided very good speed up on GPU2.

Kernel	**Speed up** (GPU2 Vs CPU)
yuv 420->422	49.40
yuv 422->444	205.24
yuv 444->422	178.65
yuv 422->420	164.61

Table 4 Acquisition kernels GPU2 Vs CPU

4.2 Warping

This part of the pipeline includes boundary detection, background pixel and foreground pixel projection.

Kernel	**Speed up** (GPU1 Vs CPU)
Boundary detection	2.53
Background projection	2.90
Foreground projection	12.69

Table 5 Warping kernels GPU1 vs CPU

Boundary detection kernel is a complex algorithm from OpenCL implementation point of view. It involves taking the decision whether the pixel is to be classified as foreground or background. This decision was taken on the basis of surrounding 8 pixels comparison to find the discontinuity in the depth values: if the discontinuity is higher than the threshold values, then the pixel with higher value is a foreground pixel and the pixel with lower value is a background pixel [1].

We were not finding the boundary for skip pixels (pixels which are not projected) as in the original C-reference code, but these pixels were within the global work size range so threads were invoked but they were not performing any operation. Hence, they were not writing anything to the output buffer resulting in random values while reading data at the host side. For this reason, each thread must initialize output buffer to zero which adds up to the global memory access, causing an overhead.

An API responsible for clearing the output buffer and setting it to some value would really increase the performance.

The number of Global Access per output Pixel was 32 resulting in a reduced performance [6]. This was brought up to 9 global accesses per pixel by loading the input pixel and surrounding pixels into the local memory which gave an improved performance of 44% w.r.t the previous version kernel on GPU1.

Moreover, instead of loading 9 pixels from global memory to local memory in each thread, we are loading more efficiently, as shown in Fig 4, then loading surrounding pixels from local memory to private memory for each thread and perform the operation. This gave a

good speed up of 25% w.r.t the previous version kernel on GPU1
Kernel with large memory access does not perform well on low profile GPUs like GPU1 but efficient local memory usage does provide performance. We are able to achieve speed up of 2.42, and moreover these type of kernels can be executed on OpenCL enabled CPUs to extract the advantage of out of order execution since the constant, local and private memory are larger on CPUs[7].

In Background projection, data is not reused by the threads in the same block, so local memory is not required. We tried to use float4(OpenCL data type) so as to use address bus efficiently[8], also efficient access of global memory but our algorithm had many conditional checks also not necessarily 4 consecutive input resulted in 4 consecutive output , these limitations pulled us from using float4. We are getting good speed up on GPU1. We also tried using image object so as to use texture memory of the device which is faster than that of global memory[6], but image buffers in OpenCL have limitations as image buffers can be read only or write only which was not much suitable for some part of the algorithm. After replacing seven data buffer objects by three images buffer objects we got speed up from 2.6 seconds to 2.3 seconds on GPU1, which is not much a speed up.
But image buffer objects do help in the future work of the where we would be generating OpenCL buffers from OpenGL so as to extract interoperability properties of OpenCL.

For foreground pixel projection, there was not necessity of local memory also float4 was not possible. Unlike the background pixel, foreground pixel are very less in number in a frame so, instead of taking the global work size for entire frame, we are identifying the number of foreground pixels and passing them through a buffer, also setting global work size for the same gave very good speed up but off course, there was cost of data transfer.

The same kernels gave a very good speed up in GPU2 as shown in the table 6.

Kernel	**Speed up** **(**GPU2 Vs CPU)
Boundary detection	214.94
Background projection	256.61
Foreground projection	222.08

Table 6 Warping kernels GPU2 vs CPU

4.3 Rounding Of Holes.

Implementation of the same in OpenCL involves filling holes by taking the average of two extremities of the hole. This means, there is dependency in the each row, but each row is independent of the other rows. Hence in the kernel one thread must operate on entire row. In our example MSR Break dancers sequence the resolution is 1024x768, so just 768 threads and each thread operates on 1024 pixels. This is a serial algorithm hence we cannot get good performance, i.e. the speed up is only 0.66 in terms of GPU1 vs. CPU clock cycles.
The kernel performance on GPU2 was very disappointing for us because the speed up was just 1.82, while other kernels even with large global access have been accelerated in hundred's.
Even with less performance we are performing the operation on GPU, because the output from the warping stage will be kept on the device without reading to the host, it will used by the blending stage hence reduces eight memory reads and eight memory writes. Again even memory read and writes are very costly on both GPU1 and GPU2 [8].

4.4 Blending

This step involves blending projected colour frames and extended colour frames.

Kernel	**Speed up** **(**GPU1 Vs CPU)
Blend proj color frames	4.08
Blend ext color frames	1.90

Table 7 Blending Kernels GPU1 vs CPU

Blending of projected colour frames was found to be very good as each thread was independent of each other and also memory transfer is less as the output from warping is on the device.

Blending of extended colour frames is paralell but we couldn't get maximum performance from the same. The output of blending of extended colour frames are three buffers: foreground, background and extended foreground. Only background buffer will be filled by the maximum number of threads but not the other two buffer hence chances that they take random values is more.

The kernel was taking 0.11 seconds on 100 run on GPU1, in CPU it was taking 0.55 the speed up was very good , but two output buffers must be initialized to zero in each thread. This adds to more global access hence the kernel speed was reduced to 0.37, which made huge difference. Basicaly the algorithm does less calculation, so in an such a algorithm addition of global access makes huge overhead, this is not the same in complex algorithm with large calculations involved.

We also executed the kernels on GPU2 and the table 8 below gives the speed up intems of GPU2 vs. CPU cycles.

Kernel	**Speed up** (GPU2 Vs CPU)
Blend proj color frames	403.49
Blend ext color frames	102.32

Table 8 Blending Kernels GPU2 vs CPU

4.5 Occlusion Solving

Even after blending some pixels will be left unfilled due to occluded areas of the scene in 3D. Those pixels will be filled by this step. Clearly the number of unfilled pixels will be very less in number, just 2 % [1] (two source index) in the entire frame so the kernel is doing few processing. The speed up achieved is just 0.6 times comparing the GPU1 vs. CPU clock cycles.
Even after using local memory the performance was not good, and we also modified the code. The unfilled pixels will be filled by the neighbour pixels with lowest depth values, so that each thread can run independent on GPU, which reduced the quality. While doing occlusion solving on the CPU we actually added three memory reads and three memories writes to the host, which is also not acceptable.

It is clearly evident that when we have situation where kernel with dependencies and memory transfers both are more time consuming, it is difficult to decide whether to go with less performing kernels or high memory transfer cost, both are not encouraging.
We will strive in future to solve occlusion and rounding of holes by good quality and parallel algorithms.

5. Results and Conclusions

It is our observation that the kernels even with large global memory access gave good speed up on high profile GPU (GeForce GTX 470), w.r.t low profile GPU (Quadro NVS 295), but this was not the same with kernels having dependencies and memory transfers.
To get maximum benefit out of GPU it is essential that the work is parallelized as much as possible. Typically in video/imaging algorithms frame (pixels) are the inputs and processed frame as outputs, if each pixel of o/p can be computed independently that is considered to be good level of parallelism.
On the other hand, there are memory transactions between CPU to GPU and vice versa which are costly in terms of time w.r.t. CPU-CPU or GPU-GPU transactions. Therefore these must be avoided as much as possible [8].
Ideally the buffers which are to be reused by later stages of the pipeline should remain in the GPU memory and reused/rewritten and only the final output should be given back to the CPU.
If data is being reused in the kernel or there is more global access per thread efficient usage of local memory and private memory (using more private memory degrades the performance) gives speed up to 50-55% (reducing the global access cycles [6]) than without using local memory kernels.
The buffers transferred from CPU-GPU and vice versa have to be of fixed size known apriori, therefore algorithms involving buffers of variable size like hash tables, histograms, linked lists etc have to be avoided or the code has to be modified thereby bringing inefficiency in the operation [5].

Kernels with large global access are found to be better performing on OpenCL enabled CPUs than on low profile GPUs like GPU1. This also provides scope for out of order execution [7].
Different local work size gives different performance on different device. If less local work size very less data will be shared among threads, if more local work size forms less parallel threads hence this must be handled carefully [7].
Loading small buffers directly into constant memory space gives good performance.
One dimensional range is preferable, due to cache locality and saving index computation.
We strongly recommend Khronos group to add features so that the buffers can be initialized to zero also set to some values which makes the API powerful.
Although still far from real time, our solution for Free Viewpoint Video chain is reliable and good-quality chain. We can clearly see that GPU's can be employed for general purpose computing and can actually speed up the execution significantly. This is a key learning, especially in context of embedded world where the processing power is limited. If we take example of set-top-box, which consists of a CPU (host processor), video-accelerator and GPU for graphics and rendering subtitles etc. the GPU which is mostly idle can be put to good use by such programming models thereby moving complex algorithms from PC domain to embedded domain .

6. Future Work

We can see that the programs can achieve significant speed up if GP-GPU model is used. In future for us this would involve evolution in 3 main directions.

1. Search for parallel and scalable algorithms.
2. CUDA implementation of the present OpenCL code to check the performance comparison between OpenCL and CUDA.
3. Since we have OpenGL code for FVV chain, we would also like to explore the possibility of further speedup through mixed use of OpenCL and OpenGL code thereby exploiting the hardware (GPU) to the maximum.

Acknowledgment

We are indebted to Emiliano PICCINELLI and GP-GPU team at STMicroelectronics for technical and infrastructure support. Without which this work would not have been possible.

References

[1] D. Aliprandi, E.M. Piccinelli. *Image-based three-dimensional free viewpoint video synthesis*. 3DTV-CON 2009

[2] D. Aliprandi and E.M.Piccineli, Patent application 08-AG-035 for free view point Telivision field titled "METHOD AND SYSTEM FOR VIDEO RENDERING, COMPUTE PROGRAM PRODUCT THEREFORE"

[3] Bryan Schauer, *Multicore Processors - A Necessity*, 2008.

[4] Khronos Group,"*OpenCL Specification v 1.0*" http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf, October 2009.

[5] G. Visentini, A Gupta, "*Depth Estimation using Open Compute Language (OpenCL)*",International Conference on Latest Computational Technologies, 2012

[6] AMD, "*ATI Stream Computing OpenCL Programming Guide*", http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf, June 2010

[7] Intel," Writing Optimal OpenCL Code with Intel OpenCL SDK". http://software.intel.com/file/37171/

[8] NVIDIA, "*OpenCL Programming guide for CUDA architecture*. Version 2.3", http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingOverview.pdf, August 2009