International Journal of Computer Science and Applications

Vol. XX No. XX, pp. XXX - XXX, 20XX

Advanced in-home streaming to mobile devices and wearables

DANIEL POHL

Intel Corporation, Campus Saarbruecken E2.1,
66123 Saarbruecken, Germany
daniel.pohl@intel.com

BARTOSZ TAUDUL

Huuuge Games, Mickiewicza 53,
70-385 Szczecin, Poland
wolf.pld@gmail.com

RICHARD MEMBARTH

DFKI, Campus Saarbruecken D3.2,
66123 Saarbruecken, Germany
richard.membarth@dfki.de

STEFAN NICKELS

Intel Visual Computing Institute, Campus Saarbruecken E2.1,
66123 Saarbruecken, Germany
nickels.stefan@gmail.com

OLIVER GRAU

Intel Corporation, Campus Saarbruecken E2.1,
66123 Saarbruecken, Germany
oliver.grau@intel.com

The quality of real-time graphics on mobile devices has improved continuously. However, a large visual gap remains between images produced on e.g. a smartphone versus images created on high-end PCs or a group of servers. To enable such high-fidelity applications, in-home streaming can be used to make server-side rendered content available, interactively, on mobile devices. For a high *Quality of Experience* it is important to have high image quality and low latency. Our proposed new framework, which fully utilizes the new 802.11ac wireless network standard, offers better image quality at half the latency of other existing solutions. Further, we extend this concept to wearables like smartwatches.

Keywords: in-home streaming; streaming; wearables; virtual reality; smartwatch; low latency; HPC; ray tracing; big data visualization; gaming.

1. Introduction

The advent of powerful handheld devices like smartphones and tablets offers the ability for users to access and consume media content almost everywhere without the need for wired connections. Video and audio streaming technologies have dramatically evolved and have become common technologies. However, in scenarios where users need to be able to interact with the displayed content and where high image quality is desired, video streaming derived technologies are typically not suitable since they introduce latency and image artifacts due to high video compression. Remote desktop applications fail when it comes to using 3D graphics applications like computer games or real-time visualization of big data in scientific High-Performance Computing (HPC) applications. Enabling these applications over Internet connections suffers significantly due to restrictions induced by the limits of today's Internet bandwidth and latency. Streaming those in local networks, commonly referred to as *in-home streaming*, still remains a very challenging task in particular when targeted at small devices like tablets, smartphones or wearables that rely on Wi-Fi connections.

In this article, we present a novel lightweight framework for in-home streaming of interactive applications to small devices, utilizing the latest developments in wireless computer networking standards, IEEE 802.11ac [Wireless LAN Working Group, 2013], for mobile devices and wearables. Further, we use a distributed rendering architecture [Chalmers and Reinhard, 1998] in combination with a high-quality, hardware-accelerated decompression scheme utilizing the capabilities of modern handheld devices, resulting in much higher image quality while requiring only half the latency compared to other streaming solutions.

The main novelties compared to our previous version [Pohl et al., 2014] are (1) additional lossless LZ4 compression that enables streaming at higher screen resolution and/or additional power savings on the client (e.g. 725 mA instead of 850 mA), (2) extending our concept towards wearables, e.g. a smartwatch, running at 85 fps, (3) using front buffer rendering to save one frame of latency, (4) additional dithering step to reduce compression artifacts. We release our framework as open source.

The setup is shown in Figure 1. The main application is running on a server or even a group of servers. Via network connection, the graphical output of the server application is streamed to a client application running on a mobile device. In addition, a back channel connection is present that collects user input events on the client and sends it back to the server. The server reacts to this input and produces an updated image, which is then transferred back and displayed at the client. The *Quality of Experience* is determined mainly by two factors: firstly, the delay between a user input issued on the client and the server-provided graphics refresh displayed at the client should be as low as possible. Secondly, the graphics quality of the streamed application on the client side should be as high as possible, even during scenarios with high motion.

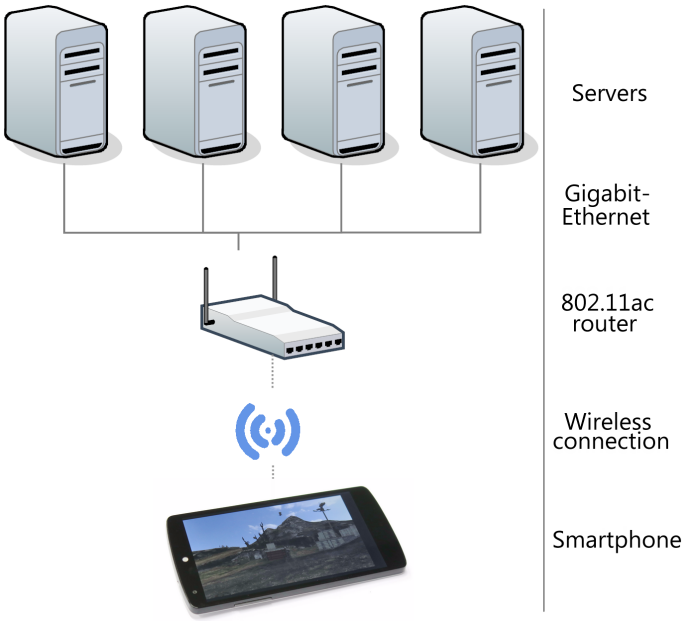

Figure 1. *Distributed rendering, in-home streaming setup targeted at mobile devices. The four servers are rendering an image. Over Gigabit-Ethernet they transport it to a router with support for IEEE 802.11ac. The router sends the image data wirelessly to the client device (smartphone) which displays it.*

2. Related Work

In this section we give an overview of known streaming technologies and applications, which we separate into three classes.

Classical desktop sharing and terminal applications: Examples are Microsoft's Remote Desktop Connection or VNC (Virtual Network Computing) [Richardson et al., 1998]. These are optimized for typical 2D applications like text processing or spreadsheet calculations. 3D support is typically very limited and, if supported, not capable to cope with the demands of real-time 3D games and visualizations.

Cloud gaming: The second class of streaming technologies has emerged from the field of computer gaming. Popular commercial solutions like Gaikai or OnLive aim at streaming applications, mainly games, via Internet connection from a cloud gaming server to a user's desktop. There are also open source approaches like Gaming Anywhere [Huang et al., 2013]. All of these are specifically optimized for usage with Internet connections and typically rely on the H.264/MPEG-4 AVC [Wiegand et al., 2003] video codec for streaming graphics. Gaikai and OnLive require a 3-5 Mbps Internet connection at minimum and end-to-end latency values are at best 150 ms under optimal conditions (c.f. [Huang et al., 2013]). OnLive uses a proprietary hardware compression chip in dedicated gaming servers hosted by OnLive. Gaikai's approach has meanwhile been integrated into the PlayStation 4 console after Sony

acquired the company and has been renamed to "PlayStation Now". The service is limited on both the client and server side to dedicated hardware. In general, cloud gaming approaches are not optimized for in-home streaming, sacrificing image quality for lowering network traffic and reducing processing latency.

Dedicated in-home streaming: The third category are approaches designed for delivering applications to other devices in a local network using wired or wireless connections. Recently, Valve's game distribution platform Steam introduced in-home streaming support. Nvidia released a portable game console named Shield in 2013, based on hardware for mobile devices running an Android operating system. Games can be streamed from a PC to the console. Both approaches rely again on the H.264 codec and are exclusively capable of streaming games. In addition, Nvidia Shield is bound to Nvidia graphics cards and mobile platform architectures. A further streaming approach, which also handles in-home usage, is Splashtop. It can stream any application, game or the complete desktop from a single machine using H.264 compression.

We are not considering solutions like Miracast, Intel's WiDi or Display as a Service [Löffler et al., 2012] as these are aimed at just replicating pixels to another display, not at interactively using the streamed application on a client device. Further, we exclude approaches like Games@Large [Nave et al., 2008], or GLX [McReynolds, 1996] that stream hardware-specific 3D function calls that are usually not compatible with mobile devices.

3. System

In this section, we propose a new framework for streaming applications from one or many high-end machines to a mobile device or wearable. Using a hardware-enabled decompression scheme in combination with a distributed rendering approach, we fully utilize the potential of recent progress in wireless computer networking standards on mobile devices. With this setup we achieve higher image quality and significantly lower latency than other established in-home streaming approaches. We first give an overview of the hardware setup used in our approach. Then, we explain our decision on the compression scheme we used and after that we talk about the details of our software framework and application setup.

3.1. *Hardware Setup*

Our distributed rendering setup consists of four dual-socket workstations using the Intel Xeon X5690 CPUs (6 cores, 12 threads, 3.46 GHz) and the Intel 82575EB Gigabit Ethernet NIC. The client is a LG Nexus 5 smartphone which uses the Snapdragon 800 CPU (4 cores, 2.3 GHz) with the Adreno 330 GPU (450 MHz) and the Broadcom BCM4339 802.11ac wireless chip. The devices are connected together through a Netgear R6300 WLAN Gigabit Router. The servers use wired Gigabit Ethernet to connect to the four Ethernet ports of the router. The smartphone connects wirelessly over 802.11ac (1-antenna setup).

3.2. *Compression Setup*

First, we have a look at how displaying of streamed content is usually handled on the client side. Using the popular video library FFmpeg and the H.264 codec an arriving stream at the client needs to be decoded. Using a CPU-based pipeline, the decoding result is an image in the YUV420 color format. As this format is usually not natively supported for displaying, the data is converted into the RGB or RGBA format. From there, the uncompressed image data will be uploaded to the graphics chip to be displayed.

If a hardware H.264 decoder is available, then the arriving stream needs to be converted into packets, suited for that hardware unit and uploaded to it. The decoding process is started over a proprietary API and usually acts as a black box. Some decoders only handle parts of the decompression procedure; others do the full work and offer an option for either directly displaying the content or sending it back into CPU memory. Hardware H.264 decoders are usually optimized to enable good video playback, but not specifically for low-latency.

Next, we have a look at our approach on displaying streamed content. An important feature of modern mobile device GPUs is, that they have native support for displaying ETC1 [Ström and Akenine-Möller, 2005] textures (mandatory since OpenGL ES 3.0, but widely supported as an extension in previous versions). Therefore, once an ETC1 compressed image arrives at the client, we can directly upload it to the graphics chip where decoding to RGB values happens directly during rendering through support in the texture units. Given the fixed compression ratio of ETC1 of 1:6 for RGB data, the required transfer to the graphics chip is lower compared to uploading uncompressed RGB or RGBA data as described in the CPU-based pipeline for H.264.

ETC1 does an image by image (intra-frame) compression instead of using information across multiple frames (inter-frame). Therefore, even if there is a lot of motion between frames, a robust image quality is guaranteed. The video codec MJPEG [Vo and Nguyen, 2008] also has this characteristic, but as it lacks hardware decompression support on mobile devices, it is not suited as it still requires non-accelerated decompression and the more bandwidth-intensive upload of uncompressed RGB/RGBA pixels to the GPU. Nevertheless, when comparing the image quality of an intra-frame with an inter-frame approach (like H.264) at the same bit rate, the latter will usually be of higher quality. However, codecs with inter-frame compression usually have higher latency.

To get better results, we are using an additional on-top, lossless compression scheme. ETC1 compression is designed for a fixed data size reduction in GPU memory, where image reconstruction speed of randomly accessed pixels is of the utmost importance. This is in a stark contrast to the more traditional approaches using dictionary coders or Huffman trees, which target tight packing of data and rely on stream decoding. Combining those two techniques together would allow us to transfer more raw image data within the limited data rate of the connection.

For our needs, we had to use a very fast compression algorithm, both on the server and especially on the client, which is much more limited in terms of computational power. The best currently available solution is the LZ4 library, offering very high speed with 422 MB/s compression and 1820 MB/s decompression on an Intel Core i5-3340M processor, which beats the industry standard DEFLATE algorithm used in the zlib library by an order of magnitude. Despite being a relatively new compression algorithm, LZ4 is already well spread and used in software such as the Linux kernel. We also performed tests with the HC (High Compression) variant of LZ4, which offers even higher decompression speed, but is on par with zlib concerning compression speed. The resulting frame rate drop convinced us that currently it makes no sense to explore other compression schemes.

3.3. *Software Setup*

The Microsoft Windows 7, 64-bit servers are running our custom written HPC ray tracing software, partly accelerated by Intel Embree [Wald et al., 2014] and multi-threaded through Intel Cilk Plus [Madhusood, A., 2013]. The ray tracer can be given the task to only render certain regions of the complete image. The ray tracer hands over the image section to the streaming module of our framework. This module can compress image data into the ETC1 format by using the etcpak library which we multi-threaded using Intel Cilk Plus. LZ4-compressed data can be sent to the client using TCP/IP, supported by libSDL_net 2.0. Further, the server listens on a socket for updates that the client sends.

The client runs Android 4.4.4 and executes an app that we wrote using libSDL 2.03, libSDL_net 2.0, LZ4 r123 and OpenGL ES 2.0. All relevant logic has been implemented using the Android Native SDK (NDK r10d).

In the initialization phase the client informs the servers about the rendering resolution and which parts the render server should handle, parameters for loading content, and initial camera settings for rendering. Then the client will receive an image (or part of it) from the server. After the initialization steps the following procedure as described in Figure 2 will be processed every frame. The client checks for user input (like touch or accelerometer and magnetometer readings) and interprets this into changes in the camera setup (this step can also be done on the server side to stay application-independent). Those new settings, an unique time stamp and other application relevant data (total of 192 bytes) are then sent to the server. The server receives this over the network and updates its internal state. An image (or part of it) is rendered, then compressed to ETC1, compressed (lossless) using LZ4 and sent to the client. There, the compressed image data is LZ4-decompressed to ETC1 format and uploaded as an OpenGL ES texture. Next, a quad is drawn on the screen using that texture at the representing areas that have been assigned earlier to the rendering server.

The rendering algorithm used here, ray tracing, is known as an "embarrassingly parallel" problem [Fox et al., 1994] with very high scalability across the number of cores, CPUs and servers, because rendering the image can be split into smaller,

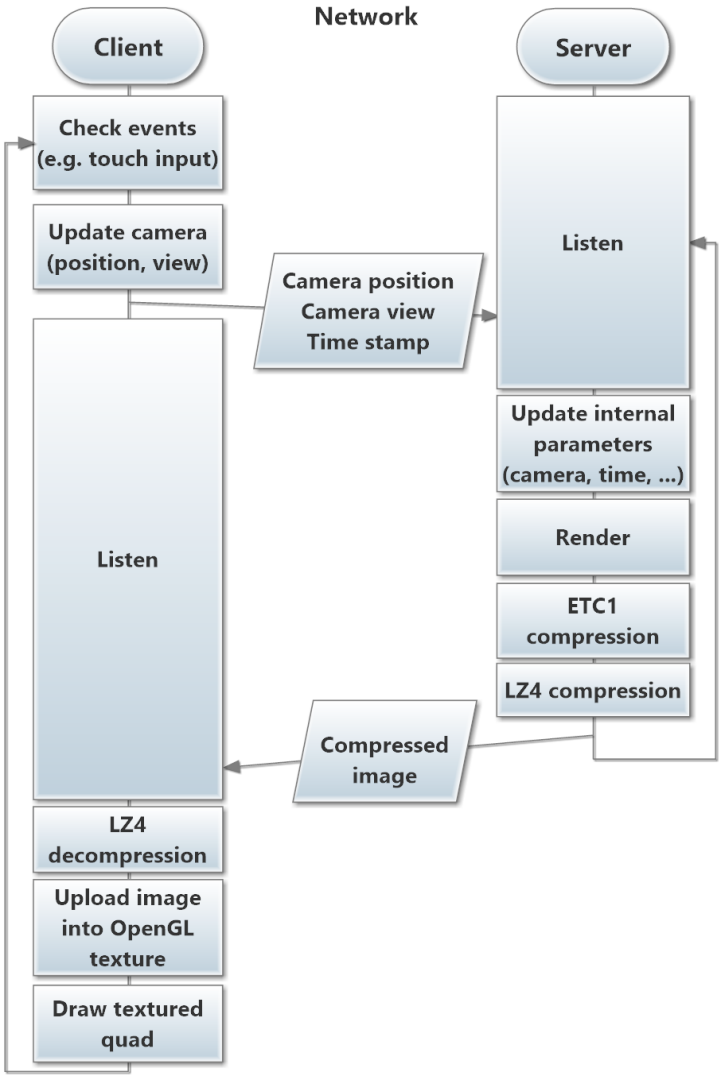

Figure 2. *Tasks of the client and server architecture.*

independent tasks without extra effort. Therefore, for the multi-server setup we naïvely split the image into four parts (one for each server), dividing the horizontal resolution by 4 and keeping the full vertical resolution. In order to achieve good scaling for a high number of servers, we recommend instead using smaller tiles and to smartly schedule them over e.g. task stealing [Singh et al., 1994]. In addition to Figure 2, the client will now send one packet (192 bytes) to each server regarding the updates. The client will get the first part of the image from the first server, upload it as OpenGL ES texture, receive the second part of the image and so on.

As exemplary rendering content the "island" map from the game *Enemy Terri-*

tory: Quake Wars (id Software and Splash Damage) and a sample scene containing the Stanford bunny are used.

It is our goal to make a very smooth experience by fully utilizing the display refresh rate of the smartphone (60 Hz). Given the properties of our hardware setup we choose to render at 1280×720 pixels instead of the full physical display resolution of the Nexus 5 (1920×1080 pixels) as it was in the current setup not possible to guarantee higher resolutions with 60 Hz due to limitations of the data rate of a 1-antenna 802.11ac setup in the smartphone.

4. Results

Here, we evaluate our and other in-home streaming approaches in terms of compression rate, data rate, latency, image quality and power consumption. We compare our implementation with Nvidia Shield (Android 4.3, System Update 68) and Splashtop (Splashtop Personal 2.4.5.8 and Splashtop Streamer 2.5.0.1 on the Nexus 5). While using Steam's in-home streaming, we encountered a high amount of frame drops during our tests, so we will not further compare it here.

4.1. *Compression Rate*

While ETC1 compression has a fixed ratio of 1:6 for compressing RGB image data, the ratio for using lossless LZ4 compression on the ETC1 blocks varies. We tested the compression of various screenshots from five games to see how efficient the additional LZ4 compression works. The average additional gains from using LZ4 on ETC1 blocks are:

Title	**Compression Ratio LZ4 : ETC1**
Angry Birds	1 : 3.8
BioShock Infinite	1 : 1.7
Far Cry 3	1 : 1.3
Max Payne 3	1 : 1.9
Enemy Territory: Quake Wars	1 : 1.5

4.2. *Data Rate*

The wireless networking standard 802.11ac allows a maximum data rate of 433 Mbit/s (for a 1-antenna setup). Hardware tests show an effective throughput of 310 Mbit/s (38.75 MB/s) for our router [Ahlers, 2014]. As our rendering resolution is 1280×720 pixels and has 8 bit per color channel this makes about 2.64 MB per image for uncompressed RGB data. Using ETC1 with the fixed compression ratio of 1:6 leads to 0.44 MB per image. Assuming the effective throughput, this results in a maximum of about 88 frames per second (fps), not including additional savings through the lossless LZ4 compression.

4.3. *Latency*

The total latency from a user input to an update on the screen (motion-to-photons time) can have various causes of lag in an interactive streaming setup. The user input takes time to get recognized by the operating system of the client. Next, the client application needs to react on it. However, especially in a single-threaded application, the program might be busy doing other tasks like receiving image data. Afterwards, it takes time to transfer that input (or its interpretation) to the rendering server over network. There, the server can process the new data and start calculating the new image. Then compression takes place and the data is sent to the client, which might be delayed if the client is still busy drawing the previous frame. Once the image data is received, it will be uploaded to the GPU for displaying. Fixed refresh rates through VSync might add another delay before the frame can be shown. Some displays have *input lag*, which describes the time difference between sending the signal to the screen and seeing the actual content there.

For the following measurements of our implementation we took the setup with four servers. As the distributed rendering approach does not work with the solutions we are comparing to, we modified the setup to use only one server and a very simple scene that achieves the same frame rate on a single machine as our four servers in the more complex rendering scenario. That way we have a fair comparison of the latency across the approaches. To get accurate motion-to-photons time we captured videos of user input and waiting for the update on the screen. Those videos are sampled at 480 frames per second using the Casio Exilim EX-ZR100 camera. In a video editing tool we analyzed the sequence of frames to calculate the total latency.

Using our approach led to a motion-to-photons latency of 60 to 80 ms. On Nvidia Shield, which uses H.264 video streaming, we measured 120 to 140 ms. The Splashtop streaming solution, also relying on H.264, shows 330 to 360 ms of lag. We visualized this in Figure 3.

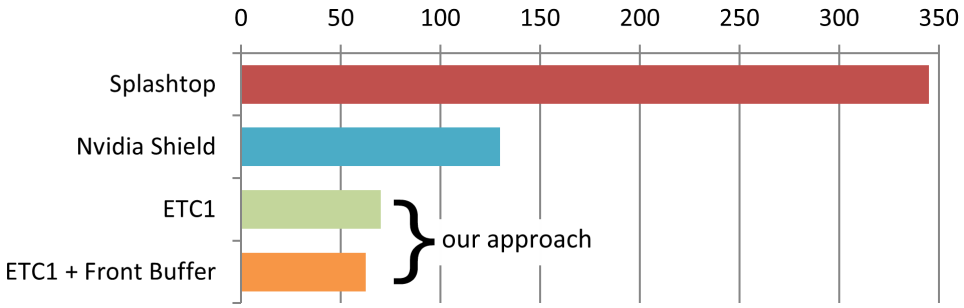

Figure 3. *Latency of in-home streaming solutions in ms. Lower is better.*

One of the causes that increase latency even more is the usage of multiple buffers for graphics. For example, when displaying the image on the screen of the client, OpenGL uses double (or even triple) buffering. In one buffer the current image is

drawn, while the previous buffer is displayed. At a display refresh rate of 60 Hz this adds additional 16.6 ms of latency. Double buffering is the default in all modern architectures and can usually not be changed to a single buffer, with one exception. The Samsung Galaxy Note 4, which is also used for the virtual reality device Samsung Gear VR, has a special interface that allows enabling single buffering to reduce latency. This is enabled through the egl_GVR_FrontBuffer() function and a call to enable GL_WRITEONLY_RENDERING_QCOM. In portrait mode the screen will be updated from left top to left bottom, then continuing right from it, again from top to bottom. To avoid artifacts, it needs to be precisely timed when new pixels are put into the buffer. For a stereoscopic image, this means that after the screen updated the left eye view and starts working on displaying the new right eye view, the left area can be updated. We enabled the front buffer rendering mode, but for test purposes did not implement the necessary timing, therefore we saw some artifacts. Running on the Note 4 with front buffering enabled, we measured a latency of about 55 to 70 ms, down from 70 to 85 ms that we measured on that device without front buffering. The last number is a bit higher than on the Nexus 5, presumably due to higher touch screen latency on the Note 4.

4.4. *Image Quality*

To analyze the difference in image quality we chose one image of a sequence in which a lot of camera movement is happening as shown in Figure 4. We quantify the image quality using the metrics of Peak Signal-to-Noise Ratio (PSNR) [Wang et al., 2002] and Structural Similarity (SSIM) [Wang et al., 2004] index, which takes human visual perception into account. For Nvidia Shield and Splashtop we were not able to test the distributed rendering setup, so we precalculated the ray traced frames offline and played them back from a single machine at the same speed that they would have been generated using four servers. That way a fair comparison of the image quality happens across all approaches. In Table 1, one can see that our approach has better image quality compared to Nvidia Shield, Splashtop and H.264 encoding at 5 Mbit/s. As expected, higher bit rate inter-frame encoding offers even higher image quality: going to 50 Mbit/s using H.264 succeeds the quality delivered by ETC1. Using an even higher bit rate than 50 Mbit/s during H.264 encoding does practically result in the same image quality for our setup.

As an additional option to reduce ETC1 compression artifacts, we added support for Floyd-Steinberg dithering [Floyd and Steinberg, 1976]. This step needs to be done before the ETC1 compression, to reduce the source image color depth from RGB888 to RGB565. Since ETC1 blocks encode the average color in RGB444 or RGB555 color space, the fewer colors there are in the source data, the easier it is to find a good fit for all pixels in a block. The results can be seen in Figure 5.

4.5. *Performance*

Even without the lossless LZ4 compression, we are able to achieve 60 frames per second at 1280×720 pixels. Given the ETC1 compression ratio of 1:6, this corre-

Figure 4. *Left: Previous frame. Right: Frame for analysis with marked red area.*

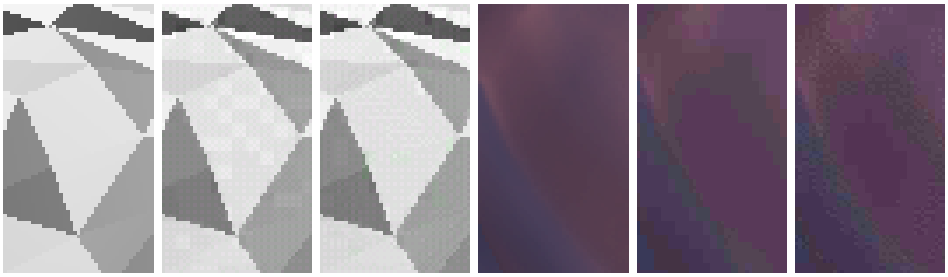

Figure 5. *The first three images show a close-up of the bunny: original, ETC1-compressed without dithering and ETC1-compressed with dithering. The last three images are analogue from a screenshot of BioShock Infinite.*

sponds to an effective throughput of 210.93 Mbit/s (26.36 MB/s). We first look at the time spent without using LZ4 compression. The relevant components on the server side are rendering of the image region (∼7 ms), network transfer (∼3 ms) and ETC1 compression (∼2 ms). On the client side, network transfer (∼12 ms) and OpenGL ES commands including swapping the display buffer and waiting on VSync (∼4 ms) are the most time consuming tasks. Once we enable additional LZ4 compression, the server needs less than 0.1 ms for that step. The client takes about 1 ms for decompression. But, because now less data is transferred, the time for the network transfer goes down. This means, that as long as the compression rate (see Section 4.1) is about 1 : 1.1 or better, LZ4 compression should be enabled. Depending on the compression rate of the used content, our in-home streaming approach can be used for higher resolutions than 1280 × 720 pixels, despite the data rate limitations of the 1-antenna 802.11ac setup.

4.6. *Battery Drain*

For the Nexus 5 we use the "CurrentWidget" app. In our approach, we observed an average battery usage of 850 mA, 605 mA for Splashtop and 874 mA for the locally rendered 3D game "Dead Trigger 2" (Madfinger Games). The higher battery usage of our approach compared to Splashtop can be explained by the fact that we are handling much more data. The Nvidia Shield console uses different hardware; there-

	Original	H.264 211 Mbit/s	H.264 50 Mbit/s	ETC1	H.264 5 Mbit/s	Splashtop	Nvidia Shield
PSNR	-	47.0	46.9	37.3	32.9	31.0	29.8
SSIM	1.0	0.997	0.997	0.978	0.877	0.861	0.779

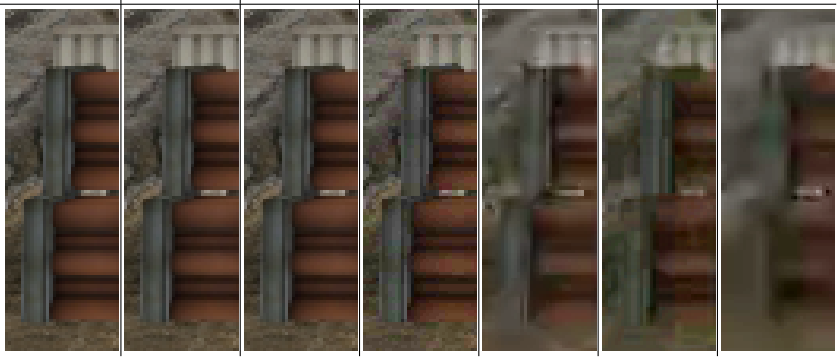

Table 1. *PSNR and SSIM values for different codecs and platforms, exemplified by respective image sections (contrast-enhanced) as marked in Figure 4. Higher values are better. While with a high bit rate the image quality of H.264 surpasses ETC1, the later approach has significantly lower latency, which we show in section 4.3.*

fore, the architectural difference has impact on the result and cannot be compared directly. Nevertheless, we report the number for completeness. The "CurrentWidget" app does not work on Nvidia Shield, so we measured the drop in percentage of the available battery power for an hour and by knowing the total battery capacity we got a value of 880 mA.

Our approach has been measured without LZ4 compression above. Once enabled, and if a high enough compression can be reached, then less battery usage can be observed. With a compression ratio of LZ4 to ETC1 of 1 : 1.8, the average battery usage was 725 mA.

5. Enhanced Applications

In this section, we have a look on various application scenarios that are enhanced by our high-quality, low latency in-home streaming solution. These can vary widely as the content of the displayed image is independent of the internals of the used compression, transportation and displaying method.

High-quality Gaming

As there are already products evolving for in-home streaming for games, like Nvidia Shield and Steam's in-home streaming, this could potentially be an area of growth. The benefits are e.g. to play on the couch instead of sitting in front of a monitor or to play in another room where an older device is located that would not be able to render the game in the desired high quality. In fast-paced action games like first person shooters, it is important to be able to react as fast as possible, therefore, our reduced latency setup enriches the gaming experience. The commercially available solutions for in-home streaming of games are typically limited to using the rendering power of only one machine. Through the distributed rendering approach

we potentially enable closer to photo-realism games by combining the rendering power of multiple machines.

Virtual Reality for Smartphones

For virtual reality there are projects like FOV2GO, Durovis Dive (see Figure 6) and Samsung Gear VR that developed cases for smartphones with wide-angle lenses attached to it. Once this is strapped on the head of a user, mobile virtual reality can be experienced. For a good *Quality of Experience* high-quality stereo images need to be rendered that have pre-warped optical distortion compensation to cancel out spatial and chromatic distortions of the lenses. While this works well on desktop PCs [Pohl et al., 2013], the performance and quality that smartphones can achieve today is not very compelling for high-end, immersive virtual reality. To achieve higher image quality, these applications have to switch from a local to a server-based rendering approach. As latency is an even more important issue in virtual reality, our latency-optimized approach is in particular suitable for this scenario. Solutions with a latency of 120 to 140 ms (Nvidia Shield) would lead to much more motion sickness compared to a latency of 60 to 80 ms. Nevertheless, optimizing virtual reality streaming applications for even lower latency might become more relevant in the future.

Figure 6. *A mobile virtual reality platform that can be strapped on the head. In front of the case a smartphone is plugged in. Lenses bring the image into focus for the viewer. To compensate for optical distortions a high-quality, pre-warped stereoscopic image is used and streamed with low latency using our framework.*

Wearables

One of the emerging trends in the space of wearables are smartwatches. E.g. the Simvalley AW-421.RX is a fully independent Android device, equipped with its own CPU and GPU, 802.11n Wi-Fi and a 240x240 resolution display running at 120 Hz. Size, battery life and cost are limiting factors, so these devices are usually

equipped with less capable processing units compared to other mobile devices. Using our in-home streaming approach, we were able to interact with server-generated content to unlock the full power of smartwatches—independent of their weak internal components. We achieved a frame rate of 85 frames per second on this device (see Figure 7).

Figure 7. *Smartwatch running an interactive, server-calculated app at 85 frames per second*

HPC and Big Data

A scenario where our streaming solution is also well suited for is real-time visualization of data-intensive computations like in the big data and HPC domain. Here, specialized applications either run analyses on huge amounts of data or computationally intensive calculations, typically relying on powerful back ends with a high amount of system memory. Typical application domains are health sciences, simulations in engineering, geographic information systems or marketing and business research. Being able to control, monitor and visualize these computations running on big server farms from small handheld devices is a very convenient benefit. Our solution, in comparison to other in-home streaming approaches, enhances graphics streaming for HPC applications as it supports a distributed scheme for rendering natively at high-quality and low latency. A testbed where we are currently integrating our streaming solution into is the molecular modeling and visualization toolkit BALL/BALLView [Hildebrandt et al., 2010; Moll et al., 2006], see Figure 8. In BALLView, running computationally demanding molecular dynamics simulations in combination with real-time ray tracing visualization on complex molecular data sets [Marsalek et al., 2010] requires a powerful compute server.

6. Conclusion and Outlook

We have shown a new approach to in-home streaming that fully leverages the latest development in wireless network standards and utilizes a hardware-accelerated

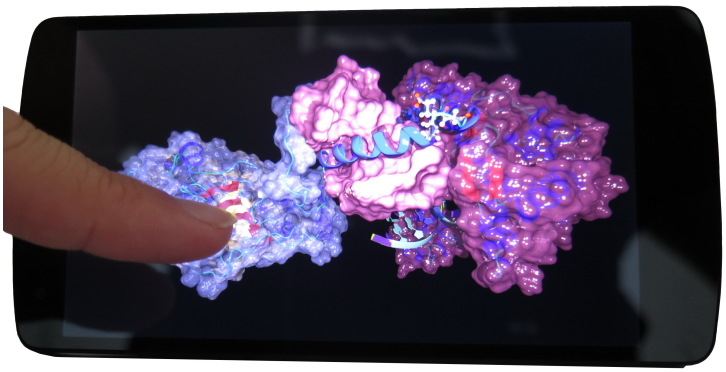

Figure 8. *Molecular visualization from BALLView displayed on a mobile device.*

intra-frame decompression scheme supported by modern mobile devices and wearables. The approach is well suited for streaming of interactive real-time applications and offers significantly higher image quality and half the latency in comparison to other recent solutions targeting this space.

Further optimizations like hardware encoders for ETC, switching to ETC2 compression and using a 2×2 antenna network connection setup, as supported by IEEE 802.11ac, will lead to even higher image quality, faster performance and will enable 1080p at 60 frames per second.

7. Appendix

7.1. *Open Source*

We include the source code of a minimal ray tracer (minrt) and a minimal in-home streaming client (minclient), which implements the described framework from this article. It comes with a simple test scene that includes the Stanford bunny. minrt runs under current versions of Microsoft Windows 32 and 64 bit, Linux and Mac OS X. The client runs on Android 4.x or higher. The source code can be found at https://github.com/ihsf.

7.2. *Further Information*

Additional information on used products and libraries can be found here:

- www.splashtop.com
- shield.nvidia.com
- bitbucket.org/wolfpld/etcpak
- code.google.com/p/lz4
- www.samsung.com/global/microsite/gearvr
- code.google.com/p/currentwidget
- www.durovis.com
- www.simvalley-mobile.de/Android-Watch-silver-PX-1794-919.shtml

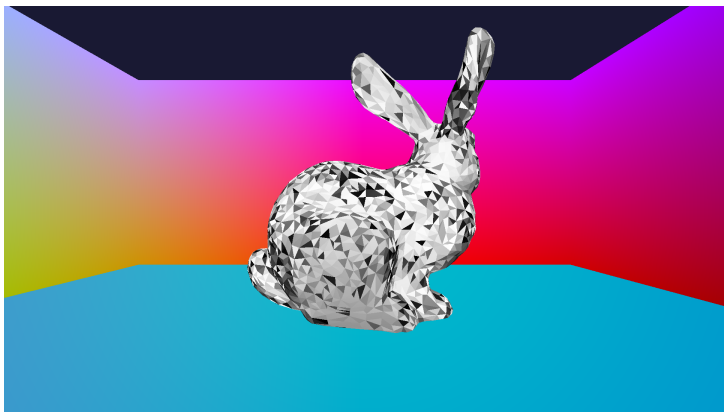

Figure 9. *Sample scene in our open source release of the minimal ray tracer (minrt).*

References

E. Ahlers. Rasante Datenjongleure. *c't Magazin*, 1:80–89, 2014.

A. Chalmers and E. Reinhard. Parallel and distributed photo-realistic rendering. In *Philosophy of Mind: Classical and Contemporary Readings. Oxford and*, pages 608–633. University Press, 1998.

R. W. Floyd and L. Steinberg. An adaptive algorithm for spatial grey scale. In *Proceedings of the Society of Information Display*, volume 17, pages 75–77, 1976.

G. Fox, R. Williams, and G. Messina. *Parallel Computing works!* Morgan Kaufmann, 1st edition, 1994.

A. Hildebrandt, A. Dehof, A. Rurainski, A. Bertsch, M. Schumann, N. Toussaint, A. Moll, D. Stockel, S. Nickels, S. Mueller, and O. Lenhof, H.-P.and Kohlbacher. BALL - Biochemical Algorithms Library 1.3. *BMC Bioinformatics*, 11(1):531, 2010.

C.-Y. Huang, C.-H. Hsu, Y.-C. Chang, and K.-T. Chen. GamingAnywhere: An open cloud gaming system. *Proceedings of the 4th ACM Multimedia Systems Conference, MMSys 2013*, pages 36–47, 2013.

A. Löffler, L. Pica, H. Hoffmann, and P. Slusallek. Networked displays for VR applications: Display as a Service (DaaS). In *Virtual Environments 2012: Proceedings of Joint Virtual Reality Conference of ICAT, EuroVR and EGVE (JVRC)*, 10 2012.

Madhusood, A. Best practices for using Intel Cilk Plus. White Paper, Intel Corporation, 7 2013. `http://software.intel.com/sites/default/files/article/402486/intel-cilk-plus-white-paper.pdf`.

L. Marsalek, A. Dehof, I. Georgiev, H.-P. Lenhof, P. Slusallek, and A. Hildebrandt. Real-time ray tracing of complex molecular scenes. In *Information Visualisation (IV), 2010 14th International Conference*, pages 239–245. IEEE, 2010.

T. McReynolds. Programming with OpenGL: An Introduction. `http://www.inf.ed.ac.uk/teaching/courses/cg/Web/intro_ogl.pdf`, 1996.

A. Moll, A. Hildebrandt, H.-P. Lenhof, and K. O. BALLView: a tool for research and education in molecular modeling. *Bioinformatics*, 22(3):365–366, 2006.

I. Nave, H. David, A. Shani, Y. Tzruya, A. Laikari, P. Eisen, and P. Fechteler. Games@Large graphics streaming architecture. *Proceedings of the International Symposium on Consumer Electronics, ISCE*, 2008.

D. Pohl, G. Johnson, and T. Bolkart. Improved pre-warping for wide angle, head mounted displays. *Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST*, pages 259–262, 2013.

D. Pohl, S. Nickels, R. Nalla, and O. Grau. High quality, low latency in-home streaming of multimedia applications for mobile devices. In *Federated Conference on Computer Science and Information Systems (FedCSIS), 2014*, pages 687–694, Sept 2014.

T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

J. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: performance and architectural implications. *Computer*, 27(7):45–55, 1994.

J. Ström and T. Akenine-Möller. ipackman: High-quality, low-complexity texture compression for mobile phones. *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2005:63–70, 2005.

D. Vo and T. Nguyen. Quality enhancement for motion JPEG using temporal redundancies. *IEEE Transactions on Circuits and Systems for Video Technology*, 18(5):609–619, 2008.

I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph.*, 33(4), July 2014.

Y. Wang, J. Ostermann, and Y. Zhang. *Video Processing and Communications*, page 29. Prentice Hall, 2002.

Z. Wang, L. Lu, and A. Bovik. Video quality assessment based on structural distortion measurement. *Signal Processing: Image Communication*, 19(2):121–132, 2004.

T. Wiegand, G. Sullivan, G. Bjøntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

Wireless LAN Working Group. IEEE Standard 802.11ac-2013 (Amendment to IEEE Std 802.11-2012), 12 2013.