Exploring Live Volumetric Video Compression with Parallelization and Hardware Acceleration in Mind

Jack Danner
University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana, IL, USA
CS 598 KN – Advanced Multimedia Systems
jackld2@illinois.edu

ABSTRACT

Volumetric video is a capture technique that has been around for quite a while, but interest in volumetric video capture and transmission has been growly rapidly in the past few years. Despite this growing popularity and demand, there are many bottlenecks and problems to be solved surrounding volumetric video applications. One of these applications is real time volumetric video compression and transmission for tele-immersion or other live-streamed events. Point cloud based volumetric video is the most favored representation currently due to its capacity for higher detail and ability to handle non-manifold geometry. However, this voxel-based representation requires significantly more data than a traditional pixel-based video. There is currently no standardized way to compress point-clouds, though a point cloud compression standard is in the works by MPEG. Until this standard is finished, sending a live dynamic point cloud to a client is unrealistic due to bandwidth limitations. Another approach is to compress and send the unprocessed camera sensor data to the client. This is the approach taken by Yang et al. [1] in their 2006 project which aimed to explore new possibilities in tele-immersion. Using their proposed 3D data model compression technique, I aim to show that leveraging modern hardware acceleration and parallelization yields a dramatic speed-up in compression and decompression times. I also aim to explore the optimal server/client setup and volumetric video transmission pipeline for dynamic point cloud live streaming applications. Using a simple dataset, a consumer grade GPU, and accessible CUDA-based compression tools, I show that motion JPEG video compression can be over 28 times faster than shown by Yang et al. [1] in 2006. Based upon the experimental results, I conclude by proposing a point-cloud based video transmission pipeline worth testing once a dynamic point cloud compression standard is solidified.

1. INTRODUCTION

Volumetric video represented as a point cloud first requires simultaneous sensor and video capture from many angles. This data is then converted to a point cloud using photogrammetry or other methods depending on the sensor data collected. This results in massive amounts of computational overhead and data to be streamed or stored. Due to this, there is a clear need for a method to quickly compress volumetric video data if it is to be streamed in real time without unrealistic bandwidth and hardware demand.

One approach to stream live volumetric video is to generate point clouds from your capture and compress them server-side in real time. The compressed dynamic point cloud is then sent to the client, where it is decompressed and displayed to a user. Another approach is to compress the capture data client-side in real time using common image or video compression algorithms accompanied by other techniques. This compressed capture data is then sent to the client where it is decompressed, and the point cloud is constructed in real time client-side. This is the method Yang et al. [1] use in their paper.

Yang et al. use multiple sensors to capture different viewpoint images accompanied by depth data. The propose two schemes to compress this capture data in their framework. Using their framework and results as a baseline, I explore how hardware acceleration and parallelization can be used to remove the computational bottleneck in their work from 2006, a time when platforms like CUDA were non-existent. I show that CUDA based compression libraries on even modest consumer grade GPUs show a dramatic speed-up when it comes to real time compression, leaving bandwidth as the only bottleneck. With this bandwidth bottleneck in mind, I propose the next steps in exploring real time volumetric video streaming once MPEG point cloud compression standardization is complete.

2. RELATED WORK

2.1 Real-Time 3D Video Compression for Tele-Immersive Environments

The Tele-immersive Environments for Everybody (TEEVE) research project at University of Illinois Urbana-Champaign was an experiment between UIUC and UC Berkeley to explore and experiment with tele-immersion. With this established system in mind Yang et al. set out to capture, transmit, and display volumetric video for the purpose of tele-immersion. Their testbench involved 10 camera sensors, each 640x480 captures (spatial resolution), and 10 frames captured and transmitted each second (temporal resolution). The captures included both color and depth information (RGBD). The goal of TEEVE was to compress this color and depth information with relatively high compression ratios with the prioritization of speed due to the 10 FPS requirement. The TEEVE framework utilized a middleware layer for traffic shaping, coordination, and synchronization to deal with the complexity of real-time environment transmission. Their framework did not utilize any significant hardware acceleration, and their compression and data processing were CPU-based. Without compression, the bandwidth requirement for their testbench would have been around 1.23Gbps [1].

With this need for less bandwidth utilization, they propose and test two different compression schemes.

The first scheme utilized color reduction every *n* frames of the transmission. This is a lossy compression method with high ratios, and fortunately the human eye is not particularly sensitive to color loss [1]. In essence a color reduction builds a color description tree of the observed colors, prunes the tree, and then color values are assigned based on this reduced tree. Even though color reduction takes longer than other compression methods, the reduction tree is only built every *n* frames making it faster in practice on image sequences. The reduced color data along with the depth data was then compressed using *zlib*[1].

The second compression scheme is introduced as JPEG-RH. This scheme uses motion JPEG (M-JPEG) to compress the color data. M-JPEG is a video compression format that simply compresses each frame of a video as a JPEG. This yields lower compression ratios when compared to something like H.264 or H.265, but has several benefits. M-JPEG does not propagate errors across frames and retains most detail, which is critical when it comes to volumetric video. The depth data is separately compressed with run length encoding, and then that itself is compressed with Huffman coding.

[1] *zlib* is a compression library used for general purpose data compression. It uses DEFLATE, a lossless compression algorithm that is a combination of LZSS and Huffman coding.

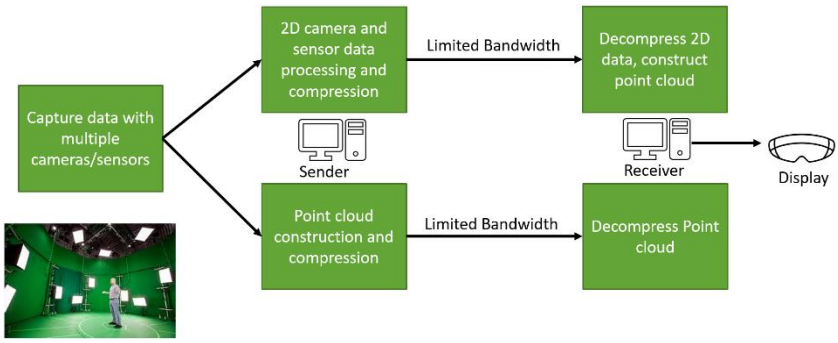

Figure 1: Two approaches to a real-time volumetric video client/server relationship

Both compression schemes hit their goal of 10 frames per second, but by narrow margins. A testing comparison is made later in the paper.

2.2 Other Works

One work worth mentioning by Gül et al. [6] takes a unique cloud-based approach to volumetric video streaming. Unlike the two client/server relationships shown in Figure 1, the cloud-based approach moves the rendering process to a cloud server. They implement a head motion prediction system on the cloud to reduce the latency added by the cloud interaction. A two-dimensional view of the volumetric video is then sent from the cloud to the client/viewer. This eliminates high performance hardware demands on both sender and receiver machines caused by the high cost of volumetric video rendering. Compression between the cloud server and the client is also no longer an issue with this approach. However, the raw capture data at the capture site still must be compressed and sent in large amounts. Therefore, bandwidth is still a bottleneck in this system. This paper focuses on streaming from a database, so the issues raised by live capture are not discussed.

Hardware accelerated compression is not completely standardized yet but there are many papers and resources that discuss the topic. Arnebäck [4] displays a parallelized CUDA implementation of run length encoding that is upwards of 35 times faster than sequential compression despite using modest hardware by today's standards (GTX 960). This performance difference grows with the amount of data being encoded, which is perfect for a volumetric video use case.

A paper by Rahmani et al. [5] presents a parallelized CUDA implementation of Huffman coding that performs up to 22 times faster than sequential Huffman coding with no constraints or tradeoffs. Their testbench uses a GTX 480 which is quite outdated, and we can expect a modern GPU to be capable of much more.

A main takeaway of this paper is the importance of the release of the Point Cloud Compression (PCC) standard by the Moving Picture Experts Group (MPEG). MPEG is responsible for many of the two-dimensional video

compression standards used today, and they took notice to the growing demand for such a standard. PCC can be thought of as the next step for video compression, and it is an important step towards transmitting point clouds for many different applications. Schwarz et al. [2] outline this upcoming standard and how it will affect the multimedia landscape.

3. HARDWARE ACCELERATION

3.1 Background on GPU Acceleration

CUDA (Compute Unified Device Architecture) is a parallel programming platform developed by Nvidia to leverage the massive parallelism of a GPU for general purpose applications (commonly referred to as GPGPU). Many tasks traditionally performed by CPUs can be parallelized and executed on GPUs. This has caused huge paradigm shifts in many fields. For example, during the 2012 ImageNet Challenge[2] a team from the University of Toronto achieved a top-5 error rate of 17.0% by using a convolutional neural network implemented with CUDA [7]. Their model's accuracy was over 10% better than the runner-up, a complex CPU-based model. GPGPU made the vast amounts of simple computation required by this model feasible. A SIMD (Single Instruction Multiple Data) parallel programming approach can be applied to certain algorithms and techniques using platforms like CUDA resulting in massive performance increases.

3.2 GPU Accelerated Compression

GPU-based video encoding has been widely adapted for common video encoding formats such as h.264 and h.265. In 2012, Nvidia introduced NVENC and NVDEC which are accelerator engines for video encoding and decoding respectfully [3]. Nvidia GPUs contain hardware-based decoders and encoders separated from the CUDA cores meaning that compression and decompression can be offloaded from the GPU. This means that a GPU is free to live-stream video without hinderance to other tasks such as rendering. FFMPEG is an open-source media streaming library that leverages this API and hardware. This technology is utilized in game streaming software such as Open Broadcaster Software (OBS) as well as media processing and compression software such as Handbrake. Nvidia's line of GPGPU and stream processing focused GPUs such as the Tesla series can encode or decode upwards of 20 video streams at a time (Figure 2).

[2] ImageNet is a large image database used to train object recognition models. The ImageNet challenge is a competition to create the most accurate classification model. The results of the 2012 competition changed the trajectory of machine learning and AI research and applications forever.

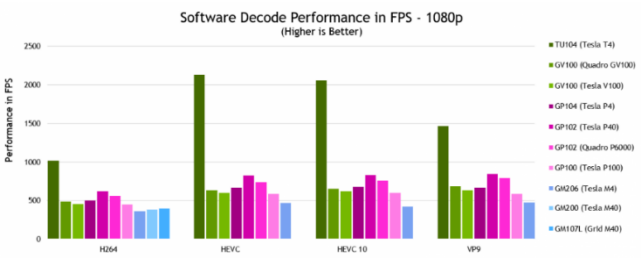

Figure 2: NVDEC Hardware-Accelerated Video Decoding performance of GPGPU and stream processing focused Nvidia GPUs [3].

3.3 Limitations

Despite this solid foundation of GPU accelerated media encoding and paradigm shift towards GPGPU, there is a huge lack of support and software for volumetric video and tele-immersive media streaming. The coding and decoding standards supported by NVEC and NVDEC are lossy and intended for two-dimensional video. This means that if one were to utilize Nvidia's technology for the purpose of reconstructed a point cloud, much detail would be lost, and error propagation would be unavoidable. Furthermore, Nvidia's acceleration engines only support color data meaning depth data or output from LIDAR (Light Detection and Ranging) cannot currently be compressed on these separate hardware encoders and decoders. This is most likely because there is currently no standard for point cloud compression, but that will change soon. As of now, a framework designed to stream volumetric video must use experimental point cloud compression algorithms that most likely have not been adapted for GPGPU or use open-source hardware accelerated coding software to create a CUDA-based compression and decompression scheme.

4. COMPRESSION TESTBENCH

4.1 Motivation

The goal of my compression testbench is to show that the real-time 3D video compression schemes introduced by Yang et al. [1] in 2006 are no longer heavily limited by computational power but mainly bandwidth. It would be ideal to implement both compression schemes using CUDA and other open-source libraries that leverage hardware acceleration, but due to time constraints and limited resources this was not within the scope of this research paper. Furthermore, it makes sense to implement and examine one piece of a compression scheme at a time in case there is a bottleneck that makes the whole scheme unviable.

Due to the use of *zlib* and color reduction, I turned to the JPEG-RH scheme. The use of the CPU-based software *zlib* complicates a making a true comparison, and a SIMD parallelized color reduction algorithm is non-trivial and would likely warrant a paper on its own. It may be worth exploring this color reduction compression scheme in the future, however. The color reduction introduced desirable

asymmetric compression times for a client/server model in their experimentation.

JPEG-RH uses M-JPEG, run length encoding, and Huffman coding in its scheme. All three of these compression techniques are simple and widely used. CUDA-based implementations of these algorithms exist, and their performance compared to a CPU-based version has been documented.

4.2 Compression Scheme

The original JPEG-RH compression scheme first compresses each frame of each camera angle serially using M-JPEG. Next, the depth data of each camera angle is compressed serially. The depth data is compressed with run length encoding, and then again with Huffman coding. In this scheme, every frame is encoded serially.

There are quite a few things to consider when designing a largely parallelized JPEG-RH scheme. First, we are no longer limited to performing color compression and depth compression serially. Depending on the computational power of the GPU being used and the size of the camera sensor data, it is possible to compress/decompress both color and depth data for each frame in parallel while using separate CUDA streams. With both happening in parallel, compression time per frame is not the sum of color and depth coding runtime, but the maximum of the two. We could go so far as to process multiple frames in parallel with a reasonable CUDA thread block and grid mapping. Designing and implementing such a scheme is a complex task that requires synchronization, a working 3D video live-streaming system with a capture studio, non-trivial CUDA implementation, and vigorous testing. This paper aims to show whether such a testbench is worth creating by scouting what is possible with today's hardware and software.

For M-JPEG compression, the tools available are proprietary or costly. However, the company FastVideo offers a non-commercial use CUDA-based JPEG compression tool called *fast jpeg*. Since M-JPEG video compression simply compresses each frame as a JPEG, *fast jpeg* can be used to simulate M-JPEG by passing it a sequence of JPEG frames. Though this approach is not tailored for maximum parallelism, the tool offers great parameters for experimental use cases such as mine.

Regarding CUDA-based run length encoding we can see that Arnebäck [4] achieved a kernel[3] 5 times faster than a CPU which reportedly can be increased with further optimizations to 35 times faster. Arnebäck's implementation is simplistic in that it compresses a string

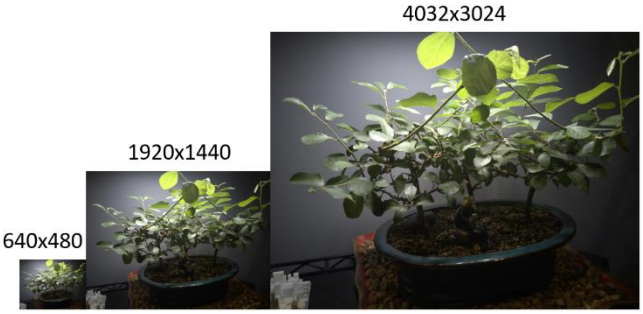

Figure 3: An example of captured test data with a visualized spatial resolution comparison.

of integers. Similarly, Rahmani et al. [5] show that Huffman coding can be sped up 22 times without tradeoff when leveraging CUDA hardware acceleration. A modern GPU would likely outperform these numbers. To effectively test depth data compression, a custom RLE/Huffman kernel would need to be created since these works are conceptual in nature. Instead, these CUDA-based lossless encoding metrics are kept in mind for testing but not relied heavily upon.

4.3 Test Data

The test bench requires an array of raw images surrounding an object of interesting at various angles. Teleconferencing usually involves a uniform background, so the images captured involve a flat and single-color background. This limits the colors in the scene and will lead to higher color data compression ratios. Yang et al. [1] use 640x480 spatial resolution in their experimentation, but the standard for image quality is higher than it was in 2006 and by today's hardware standards we can aim higher. Tele-immersion has taken the leap to virtual reality and with increasingly high-resolution screens a considerable amount of detail must be captured to generate a point cloud and or mesh that provides a good user experience. The test data contains arrays of 10 images from different angles of objects of interest. Each image has a 4032x3024, 1920x1440, and 640x480 BMP file variant.

4.4 Testbench Parameters and Hardware

FastVideo's JPEG codec takes various raw image formats as input, but the most reliable format upon testing was BMP. The codec outputs a standard jpeg file which can be decompressed by setting it as input. The user can adjust the quality and subsampling parameters, but they are left as default in final testing. The codec also allows CPU multi-threading. This is not to be confused the parallelism that a CUDA kernel utilizes on the GPU. Instead, CPU-based multithreading in this case means multiple device/host relationships[4] at once. In other words, multiple images can be compressed on the GPU at a time.

[3] CUDA kernels are functions written in C or C++ that are made to be passed to the GPU to execute on a large amount threads that are split up into thread blocks. These thread blocks are split up into grids. A CUDA kernel is designed with parallelism and thread coherence in mind.

[4] In CUDA terms, the host is the CPU and its memory, while the device is the GPU and its memory. A simple kernel execution is as follows: The host sets the thread block dimensions, copies

Motion JPEG decompression using CUDA

Input JPG	# Threads	Decomp. / s	Decode time	Theor. FPS
640x480	1	90	124 ms	726
1920x1440	1	90	202 ms	445
4032x3024	1	90	555 ms	162
640x480	4	360	373 ms	241
1920x1440	4	360	543 ms	165
4032x3024	4	360	1795 ms	50

Figure 4 Decompression of various image sizes at different decompressions per second.

Motion JPEG compression using CUDA

Input BMP	# Threads	Comp. / s	Comp. Ratio	Encode time	Theor. FPS	Bandwidth
640x480	1	90	17.44	42 ms	2131	4.59 MB/s
1920x1440	1	90	26.13	140 ms	640	27.9 MB/s
4032x3024	1	90	30.43	511 ms	176	102.6 MB/s
640x480	4	360	17.4	71 ms	1259	18.36 MB/s
1920x1440	4	360	26.13	340 ms	158	111.6 MB/s
4032x3024	4	360	30.43	1326 ms	68	410.4 MB/s

Figure 5: Compression of various image sizes at different compressions per second.

Method	24 bit color	# Streams	Target FPS	Encode time	Theor. FPS	Bandwidth
2006 Paper	640x480	10	10	599 ms	17	5.162 MB/s
CUDA GTX1070	640x480	10	10	21 ms	476	5.162 MB/s
CUDA GTX1070	640x480	10	90	179 ms	502	46.08 MB/s
CUDA GTX1070	1920x1440	10	90	766 ms	118	278.946 MB/s
CUDA GTX1070	4032x3024	10	90	3329 ms	27.035	1.06 GB/s

Figure 6: Comparison of Yang et al. JPEG-RH color compression test results to the CUDA variant. Number of streams refers to number of cameras.

Originally Yang et al. target 10 frames per second as their tele-immersion spatial resolution. In order to test whether this compression scheme can be used on a virtual reality device, I provide two test cases with the original target of 10 fps as well as 90 fps which is the current standard for a comfortable virtual reality experience.

The performance metrics used are compression ratio, bandwidth, encode/decode time in milliseconds, and theoretical fps. Theoretical fps is a measure of how many 3D video frames per second can be compressed or decompressed given the encode/decode time results of the test. The number of streams refers to how many images are being compressed/decompressed per frame.

All testing was performed on a Windows 10 machine with a i5-6600K Intel processor and 16 GB of RAM, and a GTX 1070 GPU with 1920 CUDA cores, 8GB of RAM, and a memory bandwidth of 256GB/s. A Google Pixel 3 camera was used to collect test data.

5. EXPERIMENTAL EVALUATION

5.1 M-JPEG Using CUDA

Figure 4 and 5 show the first set of results of the M-JPEG compression and decompression results tests. Each test case is split up by resolution and compressions per second. The number of threads can be likened to the number of cameras angles being captured in this test, with each thread compressing 90 JPEGs per second.

The compression test results show extremely low encode times for the 640x480 frames with low bandwidth requirements. In contrast, the 4032x3024 frames have much higher encode time by comparison. Even though the theoretical framerates are usable by today's standards, the bandwidth requirements for 4K streams are unrealistic. We start to see how the low compression ratios of M-PEG create a bandwidth bottleneck for higher resolutions. The 1920x1440 frames show a realistic middle ground and would most likely be the best candidate for a modern JPEG-RH compression scheme.

The decompression results in figure 4 show an interesting relationship between decompression time and image size. At 90 decompressions per second at a spatial resolution of 640x480, we see that it takes around 3 times longer than compression. With 4 threads and 360 images we see that it is an even bigger decrease in performance. This relationship is less dramatic at higher resolutions, but with this test it becomes apparent that decompression time will be the limiting factor when it comes to framerate in this color compression scheme.

5.2 Comparison to Yang et al.

In the second test, I adjusted the test bench to resemble the test bench of Yang et al. as closely as possible. They used 10 camera streams at 640x480 with a goal of 10 fps. Using FastVideo's CUDA-based JPEG codec, I show that color compression can be over 28 times faster than shown in their 2006 paper which uses a CPU-based M-JPEG compression tool. This effectively demonstrates the new possibilities as a result of GPGPU programming. Different resolution 90 fps variants of the setup are tested and shown in figure 6 as well. The results of these higher framerates and resolutions show that M-JPEG compression costs way too much in terms of bandwidth.

6. CONCLUSION

Despite the vast compression time improvements shown when using hardware acceleration, JPEG-RH is not a compression scheme that will scale well when high volumetric video detail and framerate is desired. The alternative color reduction tree compression scheme may be worth exploring but this requires a novel parallelized

---

data required for computation to the device global memory, then runs the kernel. The kernel executes on the GPU, writing to its output in global memory. Once the kernel finishes, the host copies the output back to its own memory.

color reduction algorithm, and the compression ratio is not notably better than M-JPEG. It is shown by the tests that modern volumetric video requires higher compression ratios and highly parallel and computationally heavy approach.

At higher resolutions, the compression of an array of images at different angles becomes wasteful. In order to construct a point cloud, the images must remain as detailed as possible. This is the inherent problem with a client/server relationship where 2D camera sensor data is being compressed and transmitted. The subject only takes up a fraction of the image, but the unneeded background color data is compressed and transmitted regardless. The images cannot be compressed tightly, so there is no possible way to achieve realistic bandwidth requirements.

This leads back to the alternative client/server relationship discussed before. The point cloud must be generated and compressed before being sent. Though a point cloud is considerably more data than a 2D image, the collection of images needed to construct it are much larger and cannot be tightly compressed. The next steps of volumetric video compression depend on a fast dynamic point cloud compression standard.

Since MPEG is working on exactly this, I posit that live volumetric video streaming will become much more feasible very soon. A much simpler live volumetric video pipeline can be proposed with PCC. If APIs like NVENC and NVDEC add support to MPEG's new PCC standard, point cloud generation will be able to take place unhindered on a GPU while compression is handled on a separate hardware encoder. In an ideal setting, capture, point cloud generation, and point cloud compression will take place on the server/sender side. A tightly compressed point cloud can then be sent to a client with less computational power such as a cell phone for decompression.

PCC is the next logical step in volumetric video and multimedia as a whole, and the ability to compress 3D representations of objects and locations in real time will open many different areas of research in the future.

7. REFERENCES

[1] Zhenyu Yang, Yi Cui, Zahid Anwar, Robert Bocchino, Nadir Kiyanclar, Klara Nahrstedt, Roy H. Campbell, William Yurcik, "Real-time 3D video compression for tele-immersive environments," Proc. SPIE 6071, Multimedia Computing and Networking 2006, 607102 (16 January 2006); https://doi.org/10.1117/12.642513

[2] S. Schwarz *et al*., "Emerging MPEG Standards for Point Cloud Compression," in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 1, pp. 133-148, March 2019, doi: 10.1109/JETCAS.2018.2885981.

[3] Nvidia Corporation. 2020. NVIDIA VIDEO CODEC SDK. (October 2020). Retrieved December 13, 2020 from https://developer.nvidia.com/nvidia-video-codec-sdk

[4] Eric Arnebäck. 2020. Implementing Run-length encoding in CUDA. (2020). Retrieved December 13, 2020 from https://erkaman.github.io/posts/cuda_rle.html

[5] H. Rahmani, C. Topal and C. Akinlar, "A parallel Huffman coder on the CUDA architecture," 2014 IEEE Visual Communications and Image Processing Conference, Valletta, 2014, pp. 311-314, doi: 10.1109/VCIP.2014.7051566.

[6] Serhan Gül, Dimitri Podborski, Thomas Buchholz, Thomas Schierl, and Cornelius Hellge. 2020. Low-latency cloud-based volumetric video streaming using head motion prediction. In Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '20). Association for Computing Machinery, New York, NY, USA, 27–33. DOI:https://doi.org/10.1145/3386290.3396933

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. Commun. ACM 60, 6 (June 2017), 84–90. DOI:https://doi.org/10.1145/3065386