Transparent Checkpointing for OpenGL Applications on GPUs

David Hou
MemVerge, Inc.
Milpitas, USA
david.hou@memverge.com

Jun Gan
MemVerge, Inc.
Milpitas, USA
jun.gan@memverge.com

Yue Li
MemVerge, Inc.
Milpitas, USA
yue.li@memverge.com

Younes El Idrissi Yazami
Northeastern University
Boston, USA
elidrissiyazami.y@northeastern.edu

Twinkle Jain
Northeastern University
Boston, USA
jain.t@northeastern.edu

***Abstract*—This work presents transparent checkpointing of OpenGL applications, refining the split-process technique [1] for application in GPU-based 3D graphics. The split-process technique was earlier applied to checkpointing MPI and CUDA programs, enabling reinitialization of driver libraries. The presented design targets practical, checkpoint-package agnostic checkpointing of OpenGL applications. An early prototype is demonstrated on Autodesk Maya. Maya is a complex proprietary media-creation software suite used with large-scale rendering hardware for CGI (Computer-Generated Animation). Transparent checkpointing of Maya provides critically-needed fault tolerance, since Maya is prone to crash when artists use some of its bleeding-edge components. Artists then lose hours of work in re-creating their complex environment.**

Keywords*—*Checkpoint-Restart, OpenGL, GPU, DMTCP, CRIU, Maya

I. Introduction

In complex media-creation programs such as Autodesk Maya [2], artists are faced with a crucial dilemma. They can stay with the core software and plugins, which are relatively robust. But eventually, for superior work, they are forced to use some third-party plugins that can be prone to crash during normal operation.

A crash causes artists to lose hours of work and forces them to wait multiple minutes to reload their project and continue working. These interruptions put artists out of their flow at the most inopportune times, and result in significant loss of productivity. Transparent checkpoint/restart (C/R) technology is a promising tool for dealing with these occurrences: by taking periodic snapshots of the editor program in the background, the artist will be able to restore from a recent checkpoint and quickly resume working.

Many media creation programs, including Maya, make extensive use of GPUs to render 3D graphics using OpenGL [3] and to perform heavier computations using CUDA and OpenCL. This typically occurs in the context of rendering farms (analogous to traditional HPC clusters) for CGI (Computer-Generated Animation). We will be focusing on OpenGL in this paper, but the techniques described are reasonably general and can be applied to any subsystem of this form. These APIs are usually implemented by a vendor-specific library, which talks to the hardware through various means.

The OpenGL API is structured as a state machine. The consumer (of the API) can allocate and load various resources, such as shaders and textures, and operate on the resources they have created. In handling these operations, the OpenGL drivers will load some state into the GPU device. We must maintain this state across checkpoint and restart. Unfortunately, current driver implementations do not provide a convenient way to do this directly. To make matters worse, driver-device communications are often closed-source, opaque, and unstable.

Luckily, the OpenGL API is well-defined and provides a clean, deterministic interface. We can leverage this to capture a program's entire OpenGL state by logging all OpenGL calls made by the program. This provides a promising idea to support C/R of OpenGL programs:

- *While the user program is running*: intercept and log all OpenGL API calls to encapsulate the state of the OpenGL state machine.
- *When checkpointing*: drop all resources (VMAs, FDs, etc) related to OpenGL drivers.
- *On restore*: recreate OpenGL drivers, and replay the logs to restore driver state.

We are able to maintain a representation of the OpenGL drivers' state and reset the drivers at will. One could say that we are checkpointing the OpenGL drivers' state separately from the rest of the program.

We present two implementations of our system, one based on CRIU and the other based on DMTCP. These two implementations share all core functionalities, demonstrating the checkpoint-package agnostic nature of our solution. These two implementations differ in their interaction with their respective checkpointing packages in order to take advantage of their different architectural properties.

There are two major problems to consider in order to enable checkpoint-restart an OpenGL application: a) reinitializing a fresh OpenGL library on restart from a checkpoint image (Section II); and b) restoring the earlier state of that reinitialized OpenGL library (Section III). Following that discussion, an experimental evaluation for the Autodesk Maya suite (Section IV) is presented. Sections III and IV are based primarily on the implementation using CRIU and VNC/VirtualGL. Section V presents a second implementation using DMTCP with native handling of X instead of VNC, but only for the GLX demo glxgears. Lastly, related work (Section VI) is presented.

II. Split processes for OpenGL and its kernel drivers

Naively, one would like to simply save the user-space memory of the OpenGL library and restore it on restart. This cannot work, since on restart, the kernel drivers will be in a state inconsistent with that of OpenGL. And for natural reasons, there is no library call in OpenGL to re-initialize the library and kernel drivers to a fresh state. So, the preferred solution is to load a fresh copy of OpenGL during restart. Any constructor functions in OpenGL will use knowledge of internals to reset the kernel drivers at that time.

The approach taken is that of a *split process*. Jain et al. [4] refined the split process concept of MANA for MPI [1] to apply to GPUs and CUDA. That package, CRAC for CUDA [4], splits the memory of a process into two regions: application code and system libraries (e.g., network, MPI, CUDA, etc.). All memory regions are tagged as upper half (application code) or lower half (system libraries). At the time

of checkpoint, only the upper half memory regions are saved. At the time of restart, a trivial lower-half application (with system libraries) is launched, and the trivial application then restores the upper half memory that was saved earlier.

The split-process approach performs better than the well-known use of proxy or helper processes (e.g., see Kazemi et al. [5], [6]). In the proxy approach, any OpenGL calls that use pointers require copying to the proxy process the buffer referenced by the pointer. With split processes, a pointer is passed directly between upper and lower half, since they share the same memory space. Further efficiencies apply when managing OpenGL resource ids.

While the OpenGL library is thread-safe, managing threads is non-trivial. If the upper-half application has two threads, then there must be two corresponding threads in the lower half. Any use of thread-local variables by the OpenGL library will be sensitive to this.

Finally, the split process approach is C/R-package agnostic. All OpenGL resource creation and deletion operations by the driver-half libraries are captured and tracked independently of the C/R package. Further, the management of upper-/lower-halves is independent of the C/R package. To demonstrate this, the work has been implemented twice (using the CRIU [7] and DMTCP [8] C/R packages).

III. Log-replay to restore OpenGL state

Section II described how to reinitialize the OpenGL library and kernel drivers on restart. Broadly, there are two important libraries, OpenGL [9] and GLX [10], which are responsible, respectively, for: (i) state machines for rendering; and (ii) managing their interaction with X-Windows. We are interested in providing the user program this interface in a way that is consistent across unload/reload operations.

To restore OpenGL state correctly during log-replay, we implement virtualization of graphics IDs. The graphic IDs are not assigned deterministically, and can change between checkpoint and restart. For example, `glCreateShader(...)` returns a value with type `GLuint`, which is an ID pointing to a graphic shader object. We may log the ID saved in the user code, but on replay, we may receive a different ID. The solution is to maintain a translation table between a virtual ID and the real ID returned by the current OpenGL library. The virtual ID is saved in the user code. Any OpenGL call using a virtual ID is automatically translated to the real ID of the current OpenGL. On restart, the virtual-to-real table is updated to use the real ID returned by the newly loaded OpenGL library.

GLX is primarily responsible for setting up an X11 Window that is responsive to X11 events. The OpenGL library creates the graphics image within a frame buffer provided by X11. We are able to save the entire state of these libraries by a log-and-replay system for capturing calls made to the two libraries. In addition, we need to handle the connection to the X server, since user program will hold coordinated state (e.g., created windows) with the X server. The standard method for dealing with this for non-OpenGL programs is to checkpoint the X server along with the user program, so that both the client and server state are captured together [7], [8]. However, these X servers generally do not support the GLX extension, so we cannot normally use GPU-backed OpenGL rendering with them. Luckily, there exists an off-the-shelf solution to this, VirtualGL. VirtualGL separates rendering from the X server implementation one by using an alternative X server to do the rendering and passing the resultant bitmap images to the desired X server. Therefore, we just need to shut down the alternative X server connection before checkpoint and restore it on restart along with the OpenGL drivers.

IV. Experimental Evaluation

Our prototype has been applied to Autodesk Maya 2020, a complex computer graphics design software widely deployed in the movie and entertainment industry. One of our goals is to demonstrate the fast restart capability that our approach brings to Maya for better crash-recovery. All the tests ran on a 2-socket compute server running CentOS 7.6 with Intel Xeon Gold 5220 CPU (2.2 GHz, 18 cores/socket), 192 GB of RAM, two 1.6 TB NVMe SSDs and one Nvidia RTX 4000 graphics card. We use CRIU as the C/R package, a VNC session to support checkpointing of the X windows, and VirtualGL is used to support GPU-accelerated OpenGL with VNC.

We use Maya to first load a moderately-sized model from the Disney's Moana Island Scene dataset [11]. We measure the loading time for both baseline (without our system), and using log-replay for OpenGL calls (with our system). In our system, we checkpoint (using the CRIU variant of our software), kill the Maya process, and restart. We show that our checkpoint-restart time is actually *shorter* than the time for the baseline to: relaunch Maya, wait for the initialization to finish and reload the same model from storage.

Figure 1 shows Autodesk Maya loaded with the model `isBayCedarA1.obj` (121 MB).

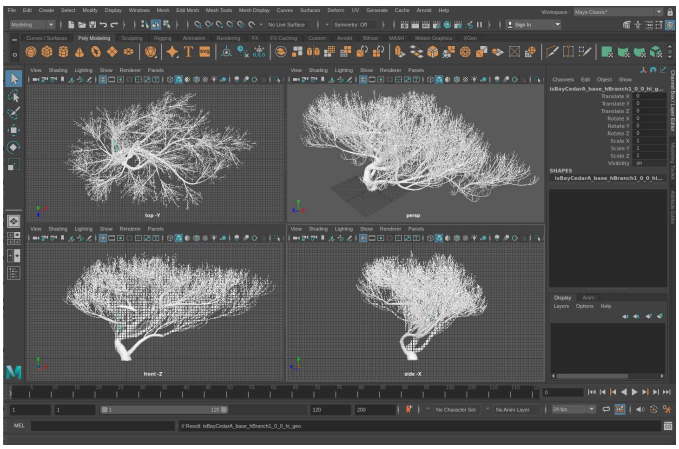

Fig. 1. Maya after loading a model from the Moana Island Scene dataset.

For the baseline, launching Maya and this model takes 60 s. We are able to restart Maya at this state from a checkpoint image on disk in just 4 s. Figure 2 shows baseline vs. C/R for various models from the dataset, and shows a clear advantage of our approach on restart speed.

The library logging, use of VNC and VirtualGL [12] during the normal operation of the OpenGL program necessarily incur performance overhead. Preliminary results show up to 10% increase on cold start time (Figure 2) and a noticeable viewport frames-per-second (FPS) penalty when interacting with the model. However, we are still able to achieve a very usable FPS on models that were tested. We plan to improve upon this in the future by reducing the number of extra transport layers introduced by the facilities above, and by introducing log pruning [5].

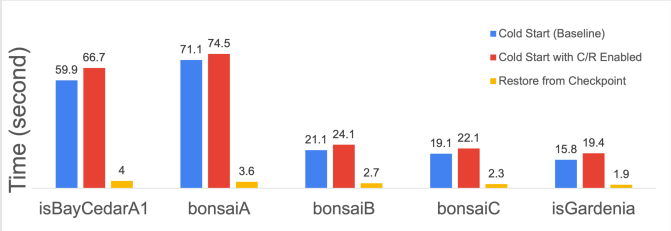

Fig. 2. Maya baseline cold start time, cold start time with C/R enabled, and checkpoint-restart time on different models.

V. Second Implementation using DMTCP

As explained in the introduction, two implementations were created using two different checkpointing packages: CRIU and DMTCP. Both implementations use the same split-process approach described in Section II. This shows the generality of the approach.

The DMTCP design does not require VNC or VirtualGL for its operation. However, the DMTCP implementation is not as far advanced, and so it is demonstrated for the well-known GLX demo glxgears, instead of for Maya. The two implementations share code for the log-and-replay that was described in Section III.

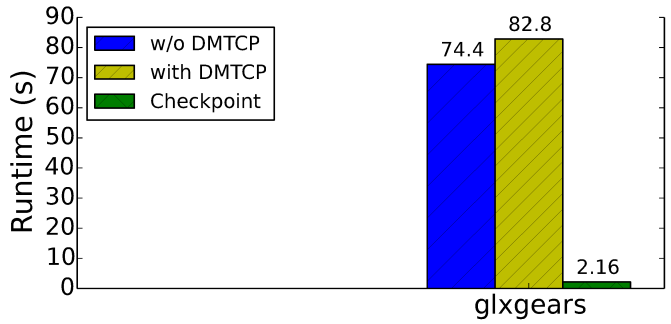

Fig. 3. glxgears runtime overhead (with and w/o DMTCP) and checkpoint time.

Glxgears normally runs infinitely and prints back out the frames per-second (FPS) information. However, we modified glxgears to loop for $10,000$ times for these experiments. The experiments were run inside a virtual machine using version 3.0 of the mesa OpenGL implementation. Mesa emulates the graphics in software without the use of the GPU.

Glxgears was run without DMTCP to establish a baseline (w/o DMTCP), and then under DMTCP's control (with DMTCP). In Figure 3 we can see that glxgears incurred an overhead of 8% when running under DMTCP's control. This is similar to the Maya experiments and is due to the many OpenGL calls made by glxgears.

The 8% overhead should be either due to the startup time of DMTCP or due to switching between the application and driver's half. A quick experiment of doubling the glxgears loop's number of iterations showed that the overhead remained at 8%, showing that DMTCP startup time was insignificant.

Hence, switching back and forth between application and driver's half is responsible for the 8% overhead. When glxgears makes an OpenGL call, the call is redirected from the application half to the driver's half. The 8% overhead can be improved (lowered) by using Linux's upcoming FSGSBASE as described in [4].

VI. Related Work

There is a surprisingly long history of checkpointing OpenGL (3D graphics) for X-Windows. In 2007, Lagar-Cavilla et al. presented VMGL [13], demonstrating vendor-independent checkpoint-restart for OpenGL version 1.5. This landmark result employed a shadow device driver to model the OpenGL state and restore it on restart. This work was VMM-independent (independent of the virtual machine). In 2010, Lin et al. [14] showed live migration of GPU-based 3D graphics between machines. In 2013, Kazemi et al. [5], [6] showed checkpoint-restart for OpenGL version 3, using a record-prune-replay technique. In future work, we intend to integrate record-prune-replay.

Acknowledgment

This work was partially supported by National Science Foundation Grant OAC-1740218 and a grant from Intel Corporation.

References

[1] R. Garg, G. Price, and G. Cooperman, "MANA for MPI: MPI-agnostic network-agnostic transparent checkpointing," in *Proc. of the 28th Int. Symp. on High-Performance Parallel and Distributed Computing*. ACM, 2019, pp. 49–60.

[2] "Autodesk Maya," [accessed Dec-2020]. [Online]. Available: https://www.autodesk.com/products/maya

[3] "OpenGL," [accessed Dec-2020]. [Online]. Available: https://www.khronos.org/opengl/

[4] T. Jain and G. Cooperman, "CRAC: Checkpoint-restart architecture for CUDA with Streams and UVM," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2020, pp. 1083–1097.

[5] S. Kazemi, R. Garg, and G. Cooperman, "Transparent checkpoint-restart for hardware-accelerated 3D graphics," http://arxiv.org/abs/1312.6650, arxiv.org, Tech. Rep., 2013.

[6] ——, "Transparent checkpoint-restart for hardware-accelerated 3D graphics (version 2)," http://arxiv.org/abs/1312.6650v2, arxiv.org, Tech. Rep., 2014.

[7] "CRIU team," [accessed Dec-2020]. [Online]. Available: https://criu.org/Main_Page/

[8] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[9] D. Shreiner, G. Sellers, J. Kessenich, and B. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley, 2013.

[10] G. Humphreys and P. Hanrahan, "A distributed graphics system for large tiled displays," in *Proceedings Visualization'99 (Cat. No. 99CB37067)*. IEEE, 1999, pp. 215–527.

[11] "Moana Island scene," [accessed Dec-2020]. [Online]. Available: https://www.disneyanimation.com/resources/moana-island-scene

[12] "VirtualGL," [accessed Dec-2020]. [Online]. Available: https://www.virtualgl.org/

[13] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. De Lara, "VMM-independent graphics acceleration," in *Proc. of 3rd Int. Conf. on Virtual execution environments*, 2007, pp. 33–43.

[14] Y. Lin, W. Wang, and K. Gui, "OpenGL application live migration with GPU acceleration in personal cloud," in *Proc. of 19th ACM Int. Symp. on High Performance Distributed Computing (HPDC'10)*, 2010, pp. 280–283.