High Performance Molecular Visualization: In-Situ and Parallel Rendering with EGL

John E. Stone*, Peter Messmer†, Robert Sisneros‡, and Klaus Schulten§
*Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, IL
†NVIDIA, Developer Technology Group, Zurich, Switzerland
*National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana, IL
§Department of Physics, University of Illinois at Urbana-Champaign, Urbana, IL
Email: *johns@ks.uiuc.edu, †pmessmer@nvidia.com, ‡sisneros@illinois.edu, §kschulte@ks.uiuc.edu

***Abstract*—Large scale molecular dynamics simulations produce terabytes of data that is impractical to transfer to remote facilities. It is therefore necessary to perform visualization tasks in-situ as the data are generated, or by running interactive remote visualization sessions and batch analyses co-located with direct access to high performance storage systems. A significant challenge for deploying visualization software within clouds, clusters, and supercomputers involves the operating system software required to initialize and manage graphics acceleration hardware. Recently, it has become possible for applications to use the Embedded-system Graphics Library (EGL) to eliminate the requirement for windowing system software on compute nodes, thereby eliminating a significant obstacle to broader use of high performance visualization applications. We outline the potential benefits of this approach in the context of visualization applications used in the cloud, on commodity clusters, and supercomputers. We discuss the implementation of EGL support in VMD, a widely used molecular visualization application, and we outline benefits of the approach for molecular visualization tasks on petascale computers, clouds, and remote visualization servers. We then provide a brief evaluation of the use of EGL in VMD, with tests using developmental graphics drivers on conventional workstations and on Amazon EC2 G2 GPU-accelerated cloud instance types. We expect that the techniques described here will be of broad benefit to many other visualization applications.**

***Keywords*-in-situ visualization; remote visualization; parallel rendering; molecular visualization;**

I. Introduction

Continuing advances in experimental imaging, structure refinement, and simulation of large biomolecular complexes such as viruses and photosynthetic organelles have created an increasing demand for high performance visualization. State-of-the-art molecular dynamics (MD) simulations of large viruses such as HIV [1], [2] and photosynthetic membranes [3], [4] contain tens to hundreds of millions of atoms, and produce terabytes of trajectory output, as do small-size but long-timescale (tens of microseconds) protein folding simulations [5], [6]. Routine visualization of the structure and dynamics of such biomolecular systems poses a formidable challenge, requiring parallel rendering with high performance graphics hardware and software. Next-generation atomic-detail simulations of cellular complexes are expected to encompass billions of atoms, with attendant increases in visualization and analysis requirements.

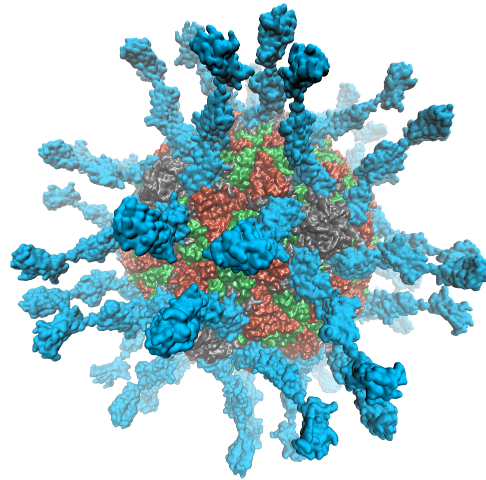

Fig. 1. Example VMD EGL-based off-screen rendering of poliovirus using OpenGL programmable shaders for shading of the virus capsid's molecular surface with pixel-rate lighting, depth cueing, and multisample antialiasing.

VMD is a widely used application for preparation, analysis, steering, and visualization of molecular dynamics simulations [7]–[9]. In the context of visualization on data center compute nodes, VMD supports in-situ visualization and computational steering of running MD simulations using a lightweight communication protocol originally developed for interactive haptic steering [8], [9], which is now supported by the popular NAMD, LAMMPS, GROMACS, and Terachem MD and quantum chemistry simulation tools. This capability allows VMD to render and analyze MD trajectory data on-the-fly as it is produced. VMD can be run as a conventional single-node interactive visualization application on remote visualization servers and clouds, but it also supports large scale parallel rendering on clusters and petascale supercomputers [2]–[4], [10]–[12].

Until now it was not possible to develop vendor-neutral visualization applications that could take advantage of high performance OpenGL implementations except through windowing system-based software interfaces. In 2004, Sun Microsystems proposed glP, an OpenGL extension for hardware accelerated rendering without the need for a windowing system [13]. Unfortunately, glP remained a proprietary solution and it was orphaned when Sun exited the high

performance graphics market. VMD has previously relied on the OpenGL Extension to the X-Window System (GLX) to manage OpenGL rasterization on GPUs and with software-based rasterizers such as Mesa and OpenSWR. Recently, it has become possible to eliminate the need for a running windowing system for visualization applications, through the use of the Embedded-system Graphics Library (EGL) in conjunction with the Vendor Neutral GL Dispatch Library (GLVND) that is part of current vendor-provided OpenGL drivers. We have adapted VMD to support EGL, permitting its use for parallel visualization without any dependency on windowing system software. The VMD EGL renderer efficiently supports multi-GPU compute node configurations without the need for any additional software libraries other than MPI, making it easy to deploy at end-user sites.

Cloud computing environments are rapidly becoming an interesting target for deployment of advanced visualization software due to the opportunity to provide a broader community of molecular scientists with access to visualization and analysis capabilities formerly available only at supercomputer centers, and perhaps most importantly, eliminating the requirement for expertise in installation and maintenance of molecular visualization and analysis tools on parallel computers. The new VMD EGL implementation was tested using Amazon Elastic Compute Cloud (Amazon EC2) "G2" GPU-accelerated compute nodes with single-GPU and quad-GPU instance types, with large host and GPU memory capacities required to load test visualizations from petascale MD simulations. The use of Amazon cloud instances permitted evaluation of developmental, i.e. so-called "beta", EGL implementations in a data center environment. We report our early experiences testing the EGL-based VMD renderer on Amazon EC2 and comment on its potential suitability for large scale visualization workloads.

Prior to the availability of EGL, the only widely available method for using OpenGL without a windowing system was via the Mesa open source OpenGL implementation, using software rasterization. One of the major potential advantages of EGL as compared with the use of Mesa software rasterization is that EGL now provides direct access to the same robust full-functionality OpenGL implementations widely used by commercial software on conventional desktop computers. For example, Mesa does not support the complete range of OpenGL extensions available in commercial OpenGL implementations. Furthermore, to date, Mesa exhibits compatibility problems with certain VMD OpenGL shaders that otherwise function properly with the commercial OpenGL implementations provided by AMD, Intel, and NVIDIA (see Fig. 6).

In this paper we describe the benefits of EGL for parallel rendering as well as remote and in-situ visualization. However, EGL offers many other benefits including simplified system configuration, management, and job scheduling for support of visualization, and simplified application deployment. Below we discuss technical and software engineering considerations for the use of EGL-based parallel rendering and remote and in-situ visualization through our early experience incorporating EGL into VMD. We report, and evaluate when possible, other potential benefits such as reduced node memory footprint, reduced propensity for operating system (OS) jitter, improved parallel job launch and cleanup performance, and reduced filesystem I/O. Multiple vendors currently support EGL and GLVND-based OpenGL libraries, and we feel that EGL is poised to become the best route for portable and efficient parallel OpenGL rendering on high performance computing systems.

II. Windowing System Considerations

Windowing systems are a necessary component for many interactive visualization and analysis scenarios. They are needed when a fully interactive remote desktop is required to allow users to work with a large number of computing tools with disparate graphical interfaces under a cohesive environment, as they would on a conventional desktop workstation. Full remote desktop type compute node usage is a valuable means for presenting a user with a familiar and convenient interactive computing environment, and it allows direct access to high performance storage systems that contain simulation inputs and both intermediate and final results. Remote desktop use can also facilitate direct communication with other jobs, including large scale parallel visualization back-ends, through direct access to high performance message passing hardware and job management systems.

At present, windowing systems are not well supported by compute nodes found in data centers, clouds, and supercomputers. In most cases the work of configuring and supporting windowing systems for these platforms falls onto the shoulders of system administration staff at customer sites rather than being provided by system vendors. The complexity of this process can be daunting in the case of large scale supercomputer systems due to the degree of specialization (and often minimalism) of their compute node OS software [14].

The creation of graphics on supercomputers poses many challenges. First, hardware configurations routinely partition the resources that users have direct access to from those that perform computations, and sometimes at multiple levels. Moreover, any significant network latency for remote users makes interacting with visualizations disagreeable, regardless of back-end rendering performance. Early researchers, in their designs of VisIt [15] and ParaView [16], [17], utilized componentized structures on top of VTK [18] consisting of at least a front-end and back-end operating in client-server mode; this model persists as state-of-the-art.

The client-server model does not resolve all issues however. Graphics accelerators have only been recently incorporated into supercomputers. Previously, visualizations performed on supercomputer compute nodes were implemented entirely in software, e.g. with Mesa. This ecosystem created a significant challenge for visualization applications that sought to support HPC clusters and supercomputers, while maintaining support for conventional laptops and desktop workstations. While supercomputers could compensate for a lack of software-based graphics performance through the use of larger node counts, support for visualization applications on laptops and

TABLE I
Summary of resources used by an X-Windows server on Cray XK7 nodes. The X-Windows server process resident set size (RSS) includes the size of code, data, and stack segments, but doesn't count some kernel structures.

Resource	X11 Server	No X11 Server
Tesla K20X GPU mem use	46 MB	32 MB
X11 process RSS	52 MB	-
X11 I/O, rchar	3.4 MB	-
X11 I/O, wchar	1.6 MB	-
kernel threads	171	170
user-mode procs	21	16

workstations was entirely dependent on graphics accelerator hardware and associated vendor-provided OpenGL implementations. There was therefore a necessity for visualization software components to handle rendering in multiple ways. This was accomplished with a variety of unwieldy and undesirable workarounds, ranging from management of multiple OpenGL-implementation-specific builds of visualization software and component libraries, to "hot swapping" of the required OpenGL libraries on-the-fly at runtime. Coupling this with the inherent difficulties of building and deploying software on supercomputers resulted in the requirement for significant time and expertise. It is clear how valuable EGL would have been for the development of visualization tools in days past; we believe the timely incorporation of EGL into ParaView version 5.0.0 (January, 2016) and VTK version 7.0.0 (February, 2016) are further evidence of its value moving forward.

By eliminating the need for a running windowing system, a significant application deployment obstacle is removed, but there are further benefits. HPC centers invest great effort on efficient execution of target applications. A high priority is therefore placed on the elimination of non-essential server-side daemons or processes. The minute possibility of such a process affecting application jitter, run-time variation, or power consumption leads to a general reluctance to add it to the list of software essential to operations. The windowing system process and associated OS services consume compute node resources, occupying a small amount of system memory and increasing the number of active kernel threads and user-mode processes. Launch-time loading of windowing system-associated shared libraries and configuration files creates additional I/O activity during parallel job launch, slowing job launch and potentially causing disruption to other running jobs. Such considerations, in conjunction with specific hardware configurations, further slowed windowing system adoption. One such example is Blue Waters, where message passing and I/O share the same communication fabric [10] which could intensify any performance degradation.

At the time when support for full graphics capabilities was first implemented on Blue Waters, its system monitoring capabilities were not as complete as they are now [14]. While decisions were supported through specialized testing, to our knowledge nothing comprehensive and reproducible has been reported to date. With the deployment of updated and additional data collection methods on Blue Waters, particularly the OVIS framework [19], we had the available resources to evaluate system effects. We first measured the compute node resource usage associated with running an (unused) X-Windows server on the Blue Waters Cray XK7 compute nodes, summarized in Table I. The resources used by the X-Windows server represent an insignificant fraction of the total host and GPU memory on the XK7 compute nodes. We also verified the additional resources to have no effect on GPU power consumption.

To evaluate the impact of a running X-Window server we performed a battery of performance tests incorporating cross-node synchronization via parallel reductions, i.e. having XK7 node CPUs coordinating the completion of simultaneous GPU benchmarks. Each benchmark included bidirectional host-GPU data transfers and several iterations of runs were completed before synchronizing to highlight any run-time variation. The tests were run on the Blue Waters system under a normal production workload period using up to 100 XK7 nodes. The tests showed no measurable loss in performance when the X-Windows server was active. In fact, aside from an approximately ten second initialization overhead to start the XK7 nodes in the correct mode, we were only able to measure an additional two seconds of overhead we may associate with the different configurations. These two seconds were across one session that launched twelve separate tests running over 24.2 minutes, a run-time variation well within the range attributable to typical system noise.

In principle, reducing the number of user-mode processes and kernel threads reduces contributions to OS jitter, which can be very harmful to the performance of large scale collective operations such as synchronizations and reductions [20]–[23]. While the above tests also showed no measurable loss in the performance of MPI collective operations which would be associated with increases in OS jitter, our tests were not optimal for detecting detrimental effects thereof. We nonetheless believe that at least on Blue Waters, a running X-Window server is unlikely to impact OS jitter. While there are X-Windows configurations that exhibit low-rate ongoing background activity such that an unused server could foreseeably add to OS jitter, we verified (using `strace`) that on Blue Waters an unused server makes no further system calls after initialization. This observation in concert with the fact that the X-Windows server is only active when requested by a job suggests that further large-scale jitter measurements are unnecessary.

While there are many potential benefits from the elimination of the windowing system when it is not needed, there can be a few obstacles that an application may need to overcome to remove accumulated dependencies on the windowing system. Sophisticated visualization applications often rely on other software libraries that may themselves include hard-coded assumptions of a running windowing system, availability of windowing system-provided fonts or color definitions, or they may require linkage with windowing system libraries even if they are not used at runtime. Early versions of the OptiX [24] GPU-accelerated ray tracing framework required linkage with the X-Windows client libraries. To overcome this issue, VMD

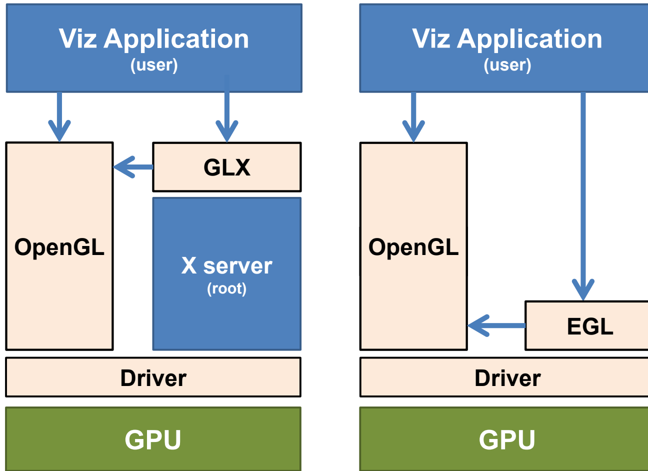

Fig. 2. Comparison of the software stack without EGL (left) and with EGL (right). Prior to EGL, an application obtained an OpenGL context by calling the X server via the GLX library. Depending on the version of X server, X had to be run as a privileged process. EGL simplifies this architecture, by allowing applications to obtain an OpenGL context without an X server.

previously incorporated stub functions that allowed compilation and linkage with OptiX [11]. While recent versions of OptiX no longer contain the windowing system dependency, this issue is somewhat pervasive in other graphics related software libraries such as OpenGL extension loaders, font rendering libraries, and others, and is therefore an important consideration in the process of implementing EGL off-screen rendering support for an application.

III. OpenGL Context Management with EGL

For the past two decades, OpenGL has been the standard interface to access graphics acceleration hardware. Being a state based API, OpenGL requires a memory object, the OpenGL context, to maintain the state between multiple API calls. The details of the OpenGL context creation are deliberately not part of the OpenGL standard. This is motivated by the need for context creation routines to interface with system specific resources like windowing systems and OS drivers. While this leads to a graphics API standard largely free of OS dependencies, it has led to a plethora of different methods for creating this context object, including the GLX library on Unix-like OSes, or WGL on Microsoft Windows.

In the case of GLX, the API is not directly responsible for managing the OpenGL hardware, but it rather acts as an interface to an X-Windows server running on the system and managing the graphics acceleration hardware. As a result, the use of OpenGL on Unix-like systems has required a running X-Windows server. For workstation settings, this has not been a significant hurdle, but in the embedded space, this extra process requires resources that could be better used by applications. Likewise in the HPC space, where the extra X-Windows server software introduces additional complexities for queuing systems' prologs and epilogs, and may introduce OS jitter or other system effects at a level unacceptable to the system operators.

Despite these shortcomings, HPC centers such as NCSA, CSCS, and ORNL have enabled X-Windows servers for their users [14]. While this approach enables the use of the graphics hardware in these GPU-accelerated supercomputers for graphics purposes, the launch of an X-Windows sever on each compute node is far from elegant. Over the years, the requirements of the embedded space have led to the development of an alternative context management mechanism, EGL. EGL is a software interface that provides application access to platform-native OpenGL rasterization hardware and software. Unlike GLX or WGL, EGL was specifically designed to support embedded systems platforms that use custom windowing systems or no windowing system at all.

EGL provides application programming interfaces for managing graphics contexts and resources such as memory buffers and output surfaces, and for synchronizing rendering operations. In conventional use, applications typically call EGL APIs to obtain contexts for OpenGL ES, a specialized variant of OpenGL for embedded systems. Since high performance visualization applications typically use the full, performance-oriented OpenGL rather than the more limited subset OpenGL ES, conventional usage of EGL is inadequate. In addition, many scientific visualization systems depend on OpenGL standards predating OpenGL ES, making it a labor intensive effort to switch to configurations without an X-Windows server. EGL has therefore been extended to enable the use of the full OpenGL standard in conjunction with contexts managed by EGL. This means that the rendering code in legacy applications can be migrated to EGL with changes to the context management code only, rather than requiring modifications to the core OpenGL rendering code. Figure 2 shows a comparison of a visualization application and the graphics software stacks associated with conventional GLX- and X-Windows-based OpenGL contexts and with the use of EGL-based OpenGL contexts.

Management of resources like the OpenGL context is accomplished by a so-called EGL platform. In the case of a workstation, this platform could be a windowing system, e.g. X-Windows or Wayland. In the case of a headless system in data center, a platform could be a generic buffer manager independent of a windowing system. In particular, each EGL client can determine which platforms are available on a specific system and select one that is most appropriate for a particular task. While this offers a great level of flexibility, an application is not privy to which OpenGL library nor which platform to link against until it inspects the available options at runtime. Relying on finding the appropriate OpenGL library first in the system library search path therefore is no longer a suitable approach. Instead, applications should now link against a vendor-neutral dispatch library, `libOpenGL.so`, that redirects the OpenGL API calls made through a given EGL context to the correct vendor-specific OpenGL implementation that is associated with the EGL platform managing the context.

IV. NUMA AND MULTI-GPU COMPUTE NODES

Over the past twenty years, HPC compute nodes have evolved toward shared memory multiprocessors based on non-uniform memory access (NUMA) architecture. NUMA compute nodes present challenges for application software, due to the need for HPC system software and applications to carefully manage placement and migration of memory allocations and application CPU threads to maintain high performance. The incorporation of GPUs into NUMA compute nodes adds further complexity. As observed by Li et al. [25], the term NUMA has become slightly misleading in common usage, since the performance asymmetries arising from modern NUMA architectures affect not only CPU memory accesses, but also data transfers to and among devices for disk and network I/O, and accelerators such as GPUs and FPGAs. To maximize performance, applications must ensure that threads managing GPU computation and rendering are bound to CPU sockets with the most direct connectivity to their associated GPUs. One of the exciting benefits of EGL in the context of multi-GPU compute nodes is that it provides APIs that enable applications to take simple and direct control over the association of a host CPU thread with a particular physical GPU, thereby ensuring optimal use of the compute node NUMA topology.

The compute node block diagrams shown in Figs. 3 and 4 exemplify common commodity-based multi-GPU compute nodes, with CPU and PCIe bus topologies that a visualization application would need to contend with in the field. In ideal scenarios where CPU threads are bound to the CPU (socket) that drives the PCIe I/O hub (IOH) for their corresponding GPU and host-side memory buffers are "pinned" in advance, host-GPU transfers would traverse only a single QuickPath (QPI) or HyperTransport (HT) CPU link, and all GPUs and network interfaces could communicate at maximum bandwidth to their host CPU threads. Conversely, if CPU threads and GPUs are mismatched, or host-side memory buffers are not pinned, a worst case scenario could result in each host-GPU transfer traversing two or more QPI/HT links, and the CPU bus linking the two CPU sockets could become a bottleneck limiting communication performance to roughly half (or less) of what would be achieved in the best case. The degree of application performance variation associated with optimal vs. worst-case NUMA mapping can vary significantly depending on many application-specific factors in combination with NUMA topology of the target hardware [25]–[28].

We have previously demonstrated the importance of correct use of NUMA topology when mapping host CPU threads to CPU sockets and GPUs for several molecular and cellular simulation applications [26], [28], [29]. Spafford et al. and Meredith et al. report findings for several other HPC applications [27], [30]. Li et al. report similar findings for data-intensive applications that access other classes of PCIe-attached hardware such as SSDs and NIC interfaces [25]. Eileman et al. reported worthwhile parallel rendering performance increases achieved by extending the Equalizer framework

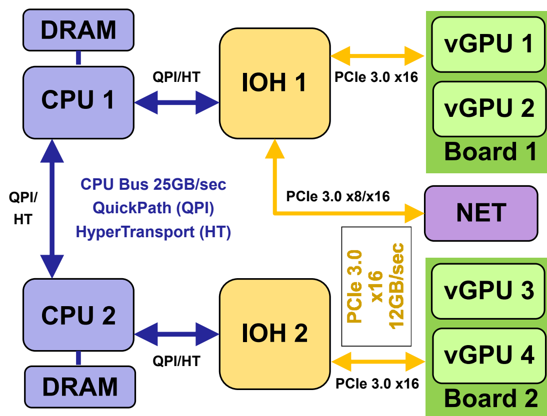

Fig. 3. Compute node block diagram for cloud-oriented hardware configurations with commodity 10 Gbit/s ethernet networking, such as Amazon EC2 CG or G2 instance types. The diagram shows NUMA CPU and PCIe bus topology and bandwidths for each point-to-point link interconnect the CPUs, PCIe I/O hubs (IOH), GPUs, and networking hardware.

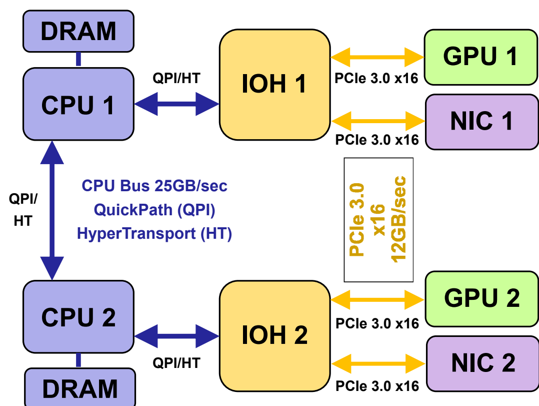

Fig. 4. Compute node block diagram representing hardware configurations found in clusters and supercomputers, where NICs and GPUs are grouped together to permit high performance RDMA transfers directly between the on-board memory of GPUs on different compute nodes. The diagram shows NUMA CPU and PCIe bus topology and bandwidths for each point-to-point link interconnecting the CPUs, PCIe I/O hubs (IOH), GPUs, and high-bandwidth low-latency interconnect adapters (NIC).

for NUMA-aware thread affinity [31]. Recently, Wang et al. reported on the impact of NUMA link contention for multi-GPU sort-last parallel image compositing [32], which is of particular interest for HPC visualization applications [33]. At present, most applications make only limited use of NUMA topology information due the complexity of doing so correctly. Our early experience with NUMA performance issues and lack of NUMA-aware applications on the "QP" and "AC" GPU clusters at NCSA led to the development of a NUMA-aware CUDA interposer library that ensures the best mapping of host threads to GPUs by intercepting CUDA context creation calls and assigning them to the optimal GPU [26]. A similar approach could conceivably be implemented for EGL by intercepting EGL calls for enumerating and binding to specific GPU devices.

The examples shown in Figs. 3 and 4 highlight different

NUMA topologies that must be used effectively for best performance. The diagram shown in Fig. 3 is closely modeled after the Amazon EC2 G2 (g2.2xlarge and g2.8xlarge) cloud instance types used for development and testing of the EGL features of VMD described further below. In Amazon EC2 G2 cloud compute nodes the application is bound to one or more virtualized GPUs which are distributed among add-in GPU boards that each contain two complete GPUs linked via an on-board PCIe switch. The cloud compute node configuration allows full-bandwidth peer-to-peer transfers between the pairs of GPUs on the same add-in-board, bypassing host CPUs entirely, e.g. for parallel image compositing [32], [33], or exchange of ghost cells or boundary elements [28].

An unusual challenge associated with cloud virtualization of CPUs and GPUs is that the virtualization layer often obscures hardware occupancy details required for an application to determine correct mapping of threads to GPUs. We note that Amazon only allows allocation of G2 instance types in one of two configurations, g2.2xlarge (one GPU), and g2.8xlarge (four GPUs). By eliminating the possibility of a two-GPU configuration, Amazon has effectively eliminated the most problematic scenario since hardware resources not used by one instance are provided to other instances running on the same compute node. Limiting instance allocations to only units of one or four GPUs eliminates the need for the virtualization system to provide a transparent view of the physical CPU and GPU allocations to applications.

The compute node shown in Fig. 4 is representative of existing and future multi-GPU cluster nodes that closely couple high performance low-latency interconnect adapters (NIC) with one or more GPUs. Although the configuration is shown based on commodity x86 hardware, it is similar in some respects to the proposed node architecture of the upcoming DOE Summit and Sierra machines, albeit with only one GPU per-socket, and a PCIe-based host-GPU interconnect rather than NVLINK. The key issue for compute node designs such as the one shown in Fig. 4 involves use of so-called zero-copy message passing approaches that allow inter-node messages to be passed directly from the memory of a GPU on one node to the memory of GPU on another, e.g. using InfiniBand RDMA operations between GPUs, bypassing the host CPUs [34].

Applications can determine NUMA topology with hardware introspection, using APIs for enumerating CPUs, PCIe buses, and PCIe devices. Ensuring that GPU mappings are consistent across multiple libraries can sometimes be a minor challenge. As an example, GPUs are often indexed differently by CUDA, OptiX, and EGL for reasons that are unique to each library. To ensure that efficient zero-copy exchange of GPU-resident data structures can be performed among several libraries, an application must ensure that GPU device indexing is done properly observing both hardware topology and the indexing scheme used by a particular library.

One way to ensure consistent device enumeration across multiple libraries is to use hardware-based enumeration. This is the default enumeration scheme used by EGL to report devices when queried via `eglQueryDevicesEXT()`. However, CUDA's default performance-based enumeration ordering assigns GPUs with higher performance to lower indices. This default behavior can be overridden by setting the environment variable CUDA_DEVICE_ODER=PCI_BUS_ID, resulting in an enumeration scheme identical to EGL. Alternatively, an application can query EGL to provide the CUDA device index for a given EGL device by querying the EGL_CUDA_DEVICE_NV attribute for a given device. Other libraries such as OptiX provide APIs for correlating CUDA device indices with their own device index.

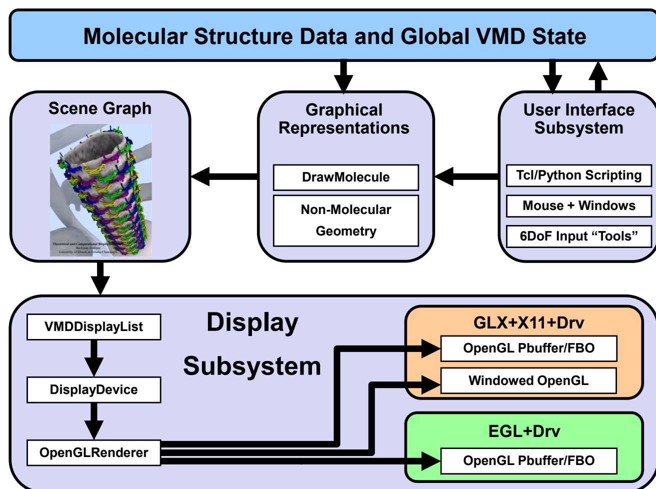

Fig. 5. Block diagram of VMD display subsystem class hiearchy with EGL. VMD can be compiled to support either a windowing system based OpenGL implementation, e.g. by linking against GLX and X-Windows libraries, or it can be compiled for EGL with no dependency on a windowing system or its associated shared libraries.

V. ADAPTATION OF VMD TO EGL

We have adapted VMD to support EGL-based OpenGL contexts as an alternative to its existing GLX and WGL renderer interfaces. The VMD display subsystem is designed as a set of C++ classes and subclasses that collectively implement all of the functionality required to render molecular scenes for diverse display hardware ranging from conventional planar displays to immersive displays including the CAVE and tiled display walls [35], [36], and more recently for VR headsets [37]. The same display class hierarchy is also used for large scale parallel OpenGL rendering [10]. Figure 5 shows a block diagram of the major VMD display subsystem components related to OpenGL rendering.

In the simplest case, the only application code that needs modification is associated with OpenGL context creation and management, thereby using EGL in much the same way that GLX or WGL would normally be used. In practice, complex software may need further modification, in particular if libraries used by the application have their own dependencies on the windowing system, GLX, or WGL APIs. Previous VMD versions used bitmap fonts provided by windowing system APIs for the display of text labels and numerical data in molecular scenes. We have redesigned VMD to use a compiled-in Hershey font library, which is OS-independent

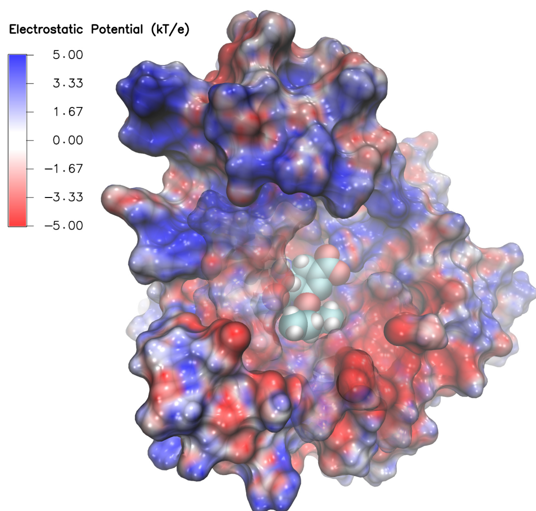

Fig. 6. Swine Flu A/H1N1 neuraminidase bound to Tamiflu, with molecular surfaces colored by electrostatic potential [38]–[40]. This EGL rendering demonstrates the full function of the most complex VMD OpenGL shaders for ray-cast spheres and 3-D volumetric texturing of potential maps, with pixel-rate lighting, depth cueing, transparency, and multisample antialiasing (8 samples per-pixel). VMD text labels, e.g. use for the electrostatic potential color scale bar at top left, use built-in Hershey fonts, so that VMD does not require windowing system-provided fonts.

and has no windowing system dependency. Some versions of the OptiX ray tracing framework contain a dependency on Xlib, which has required VMD to provide stub functions when it is linked without X11. We eliminated the last dependencies on the OpenGL utility library `libGLU` which VMD previously used as a fall-back for rendering of quadric primitives when OpenGL programmable shading was not available. Finally, VMD uses its own internal mechanisms for managing OpenGL extensions, compiling OpenGL shaders, and other functionality that in many other applications tends to be provided by external libraries, that may not be compatible with EGL yet.

As highlighted in the previous section, EGL allows multiple general windowing systems, so-called platforms, to coexist at the same time. In order for EGL to obtain the appropriate context, it is therefore necessary to select the correct platform. Given that a system could contain different graphics devices with each of the devices supporting different platforms, an application must therefore first obtain the list of available graphics devices in a given system, followed by the selection of an appropriate platform on the system. Once the platform has been initialized, one can then initialize EGL and select the desired graphics API, and then the OpenGL context can be created.

Before the OpenGL context can be used for rendering operations, rendering resources must be created. In the simplest form, this requires a so-called EGL surface. A surface can either be a (visible) window resource managed by the selected platform, or it can be an off-screen pixel buffer. This off-screen pixel buffer is particularly useful in the case of headless compute nodes typical in HPC environments. Once the surface is created, the newly created OpenGL context can be made current and the actual rendering activities can proceed.

While this approach is straightforward to follow, a drawback is that all of the rendering objects are transparently managed by EGL. If an application needs to access one of the buffer objects, e.g. for interoperability with CUDA, an additional copy is required from the EGL-managed surface to the buffer resource. Performance critical applications can therefore choose to bypass the EGL surfaces and create an OpenGL context without any EGL-managed buffers associated with them. It is then the application's responsibility to create the appropriate framebuffer, textures, and renderbuffer objects to enable correct rendering activities. On the other hand, this enables the most efficient path to sharing data between OpenGL and CUDA.

VMD currently allows the use of POSIX CPU thread affinity APIs to bind threads to the CPU sockets most closely connected to their associated GPUs, as a means of ensuring good performance on contemporary NUMA compute node architectures [10]. Some batch queuing systems and MPI implementations also enforce their own CPU affinities. At present VMD assigns EGL devices to MPI ranks in a round-robin fashion consistent with its assignment of CUDA and OptiX devices, but this scheme is not yet flexible enough to handle scenarios where CPU affinities are externally assigned and enforced by job schedulers, runtime software, cloud hardware virtualization software, or the like.

VI. Evaluation of VMD EGL Renderer

At the time of writing, support for EGL in combination with full-capability-OpenGL (as opposed to OpenGL ES) is available only in the most recent development-oriented "beta" NVIDIA driver releases. As such, it is not yet possible to run large-scale tests on production GPU-accelerated supercomputers such as the Blue Waters or Titan Cray XK7 systems. Accordingly, we have developed and tested the EGL-based OpenGL rendering path in VMD using conventional desktop workstations and with GPU-accelerated Amazon EC2 G2 cloud instances.

Using a workstation, we verified that the VMD EGL renderer produces an output that is pixel-for-pixel identical to that produced by the conventional GLX-based renderer in VMD. We verified that EGL-based rendering produced correct images, including scenes that made extensive use of OpenGL extensions and VMD-specific OpenGL shaders for pixel-rate lighting, transparent materials, and efficient sphere rendering [36], as shown in Figs. 1 and 6.

VMD EGL parallel rendering tests were performed on Amazon EC2 cloud virtual machines running on g2.8xlarge instance types containing two Intel Xeon E5-2670 CPUs with 8-cores @ 2.6 GHz, 60 GB RAM, and two NVIDIA GRID K520 GPUs, each containing two on-board virtualizable GPUs with 8 GB RAM in each on-board GPU (4 GPUs in total). Amazon-provided AMIs are very similar to the environment found on supercomputer compute nodes — they are extremely minimalistic. We prepared a GPU-capable Amazon Machine Image (AMI), replacing the Amazon-provided NVIDIA GPU drivers

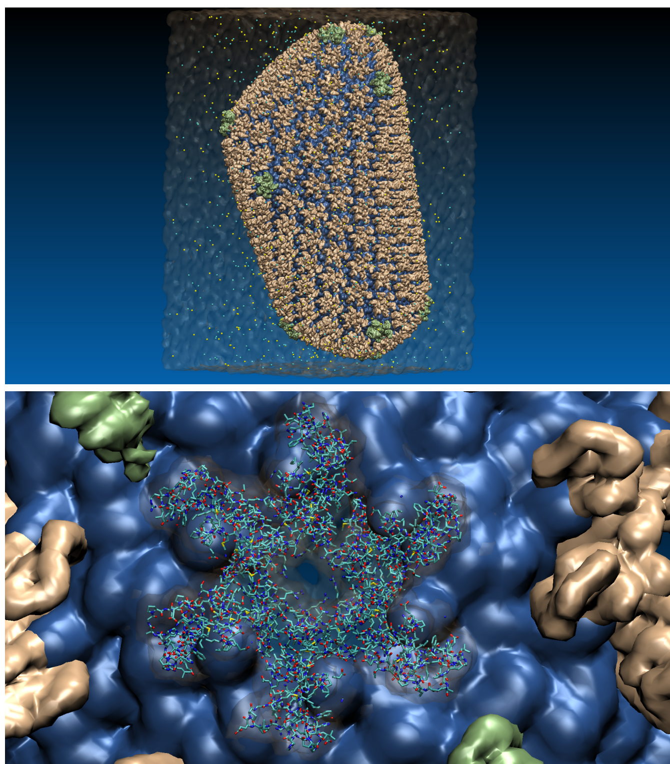

Fig. 7. VMD EGL-based off-screen OpenGL renderings of a 64 M-atom HIV-1 capsid simulation [1] selected from a 1,079 frame movie rendered on an Amazon EC2 g2.8xlarge GPU-accelerated instance. The top image shows the spatial distribution of hexameric (tan) and pentameric (green) subunits responsible for capsid morphology. The bottom image shows an atomic-detail close-up rendering of one of the capsid's hexameric subunits. The VMD HIV-1 EGL renderings used OpenGL shaders for ray-cast spheres, transparent surface materials, pixel-rate lighting, and multisample antialiasing.

with the developmental NVIDIA 358.16 beta driver required for EGL. The minimal software environment provided by Amazon's AMIs served as an excellent stand-in for a typical HPC compute node, and it allowed us to ensure that no X-Windows libraries or header files were on the system. In order to test the VMD EGL renderer for a simple parallel rendering workload, the Amazon-provided `mpich-devel` package for MPI was installed, which was the only noteworthy extra package needed beyond the standard Amazon-provided GPU AMI.

To gauge the correct parallel operation of the VMD EGL renderer we created an HIV-1 capsid visualization [11] using the new EGL renderer in an MPI-enabled build of VMD (see Fig. 7). Table II summarizes the movie rendering runtime with up to 32 MPI ranks, with one virtual GPU and eight CPU cores assigned to each MPI rank, and with each MPI rank assigned to its own separate EGL GPU device. The overall movie rendering runtimes shown in Table II include between 63 s and 68 s of non-parallelizable I/O that is associated with initial molecular structure loading and analysis that must be performed by all MPI ranks, representing an increasing fraction of overall runtime with larger numbers of MPI ranks. Through repeated test runs on one and two MPI ranks with all file I/O directed to Amazon Elastic Block Storage (EBS), we observed

TABLE II
VMD EGL HIV-1 movie rendering performance on Amazon EC2 g2.8xlarge cloud instances. ‘

MPI Ranks and vGPUs	g2.8xlarge instances	Render Time (sec), (I/O%)	
		1920×1080	$3840x2160$
1	1	619 s (10%)	626 s (10%)
2	1	331 s (19%)	347 s (19%)
4	1	209 s (32%)	221 s (31%)
8	2	140 s (47%)	141 s (46%)
16	4	107 s (64%)	107 s (64%)
32	8	90 s (75%)	90 s (76%)

significant run-to-run performance variations averaging 10%, which is attributable to significant variation in I/O performance when molecular data is first loaded and during rendering. In the worst cases recorded, we observed the fastest and slowest single-rank runtimes differ by 40%. To minimize the impact of the observed EBS I/O rate variation on parallel performance tests, we directed all disk I/O to instance-local (node-internal) SSDs, thereby eliminating the most significant external system noise from our test results, implementing a common sense performance optimization that is the very reason that the high-performance Amazon G2 instances incorporate instance-local SSD storage hardware. The overall performance trend shown in Table II indicates roughly linear parallel scaling for the rendering-specific (post-initialization-I/O) parts of the workload, with the 32-rank results representing a speedup of $26\times$ and a parallel scaling efficiency of 81%. We note that the runtime shown for the 32-rank results is dominated by initial startup I/O and that if the initial I/O runtime component were removed (68 s), the achieved movie frame rate (49 FPS) for the 1,079 frame test movie is roughly $2\times$ faster than the real-time playback rate (24 FPS) of the original movie [11].

We intend to investigate strategies for improving I/O performance during initialization and to run tests with larger numbers of MPI ranks on Amazon EC2 g2.8xlarge GPU instances. We found that the overall convenience and low cost of the Amazon EC2 GPU instances were very well suited to the development needs of this project. Although we operate and maintain significant local GPU clusters that could have been used for the work described here, it was much more convenient to use the cloud instances with the developmental "beta" GPU drivers; our local GPU clusters could continue running production science jobs with stable system software. Amazon charges a price premium for on-demand use of the highest-performance instances such as g2.8xlarge ($2.60/hour), however we were able to run all of our performance tests for less than 15% of the on-demand price using so-called spot pricing during off-peak hours. By running the tests during off-peak hours, the 32-rank tests cost roughly the same as a 4-rank test would when run on-demand. Overall we have been pleasantly surprised at how easy it was to prepare a VMD-specific AMI supporting parallel rendering with EGL. Cloud computing and EGL offer great potential for enabling a broader community of molecular scientists to benefit from parallel rendering features of VMD that were previously only available on large supercomputers.

VII. CONCLUSION

We have implemented a new EGL-based rendering system that enables the molecular visualization program VMD to support high performance OpenGL rasterization on clouds, clusters, and supercomputers that previously did not support high performance visualization. While our EGL renderer implementation is specifically tailored for the design of VMD and the needs of molecular visualization, we feel that similar use of EGL for in-situ and parallel rendering will be valuable for many other science and engineering visualization applications.

ACKNOWLEDGMENTS

The authors acknowledge support from NIH grants 9P41GM104601 and 5R01GM098243-02, the CUDA Center of Excellence at the University of Illinois, and the Blue Waters sustained-petascale computing project supported by NSF awards OCI-0725070 and ACI-1238993, the state of Illinois, and "The Computational Microscope" NSF PRAC awards OCI-0832673 and ACI-1440026.

REFERENCES

[1] G. Zhao, J. R. Perilla, E. L. Yufenyuy, X. Meng, B. Chen, J. Ning, J. Ahn, A. M. Gronenborn, K. Schulten, C. Aiken, and P. Zhang, "Mature HIV-1 capsid structure by cryo-electron microscopy and all-atom molecular dynamics," *Nature*, vol. 497, pp. 643–646, 2013.

[2] J. R. Perilla, B. C. Goh, J. Stone, and K. Schulten, "Chemical visualization of human pathogens: the Retroviral Capsids," *Proceedings of the 2015 ACM/IEEE Conference on Supercomputing*, 2015, (4 pages).

[3] M. Sener, J. E. Stone, A. Barragan, A. Singharoy, I. Teo, K. L. Vandivort, B. Isralewitz, B. Liu, B. C. Goh, J. C. Phillips, L. F. Kourkoutis, C. N. Hunter, and K. Schulten, "Visualization of energy conversion processes in a light harvesting organelle at atomic detail," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Press, 2014, (4 pages).

[4] J. E. Stone, M. Sener, K. L. Vandivort, A. Barragan, A. Singharoy, I. Teo, J. V. Ribeiro, B. Isralewitz, B. Liu, B. C. Goh, J. C. Phillips, C. MacGregor-Chatwin, M. P. Johnson, L. F. Kourkoutis, C. N. Hunter, and K. Schulten, "Atomic detail visualization of photoynthetic membranes with GPU-accelerated ray tracing," *Parallel Computing*, 2016.

[5] P. L. Freddolino, F. Liu, M. Gruebele, and K. Schulten, "Ten-microsecond molecular dynamics simulation of a fast-folding WW domain," *Biophys. J.*, vol. 94, pp. L75–L77, 2008.

[6] P. L. Freddolino and K. Schulten, "Common structural transitions in explicit-solvent simulations of villin headpiece folding," *Biophys. J.*, vol. 97, pp. 2338–2347, 2009.

[7] W. Humphrey, A. Dalke, and K. Schulten, "VMD – Visual Molecular Dynamics," *J. Mol. Graphics*, vol. 14, pp. 33–38, 1996.

[8] M. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L. Kalé, R. Skeel, K. Schulten, and R. Kufrin, "MDScope – A visual computing environment for structural biology," in *Computational Mechanics 95*, S. Atluri, G. Yagawa, and T. Cruse, Eds., vol. 1, 1995, pp. 476–481.

[9] J. E. Stone, J. Gullingsrud, P. Grayson, and K. Schulten, "A system for interactive molecular dynamics simulation," in *2001 ACM Symposium on Interactive 3D Graphics*, J. F. Hughes and C. H. Séquin, Eds. New York: ACM SIGGRAPH, 2001, pp. 191–194.

[10] J. E. Stone, B. Isralewitz, and K. Schulten, "Early experiences scaling VMD molecular visualization and analysis jobs on Blue Waters," in *Extreme Scaling Workshop (XSW), 2013*, Aug. 2013, pp. 43–50.

[11] J. E. Stone, K. L. Vandivort, and K. Schulten, "GPU-accelerated molecular visualization on petascale supercomputing platforms," in *Proceedings of the 8th International Workshop on Ultrascale Visualization*, ser. UltraVis '13. New York, NY, USA: ACM, 2013, pp. 6:1–6:8.

[12] J. E. Stone, R. McGreevy, B. Isralewitz, and K. Schulten, "GPU-accelerated analysis and visualization of large structures solved by molecular dynamics flexible fitting," *Faraday Discuss.*, vol. 169, pp. 265–283, 2014.

[13] P. Ramsey, "glP an OpenGL extension for graphics without a windowing system," Presentation by Sun Microsystems to the OpenGL ARB standardization body, Sep. 23, 2004.

[14] M. D. Klein and J. E. Stone, "Unlocking the full potential of the Cray XK7 accelerator," in *Cray User Group Conf.* Cray, May 2014.

[15] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, K. Bonnell, M. Miller, G. H. Weber, C. Harrison, D. Pugmire *et al.*, "Visit: An end-user tool for visualizing and analyzing very large data."

[16] J. Ahrens, B. Geveci, and C. Law, *ParaView: An End-User Tool for Large Data Visualization*. Elsevier, 2005, ISBN 978-0123875822.

[17] U. Ayachit, *The ParaView Guide: A Parallel Visualization Application*. Kitware, 2015, ISBN 978-1930934306.

[18] W. J. Schroeder, B. Lorensen, and K. Martin, *The visualization toolkit*. Kitware, 2004.

[19] M. Showerman, J. Enos, J. Fullop, P. Cassella, N. Naksinehaboon, N. Taerat, T. Tucker, J. Brandt, A. Gentile, B. Allan *et al.*, "Large scale system monitoring and analysis on Blue Waters using OVIS," *Proceedings of the 2014 Cray User's Group, CUG*, 2014.

[20] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 55–.

[21] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," *2006 IEEE International Conference on Cluster Computing (CLUSTER)*, vol. 0, pp. 1–12, 2006.

[22] P. De, R. Kothari, and V. Mann, "Identifying sources of operating system jitter through fine-grained kernel instrumentation," in *Cluster Computing, 2007 IEEE International Conference on*, Sept 2007, pp. 331–340.

[23] P. De, V. Mann, and U. Mittaly, "Handling OS jitter on multicore multithreaded systems," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–12.

[24] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: a general purpose ray tracing engine," in *ACM SIGGRAPH 2010 papers*, ser. SIGGRAPH '10. New York, NY, USA: ACM, 2010, pp. 66:1–66:13.

[25] T. Li, Y. Ren, D. Yu, S. Jin, and T. Robertazzi, "Characterization of input/output bandwidth performance models in NUMA architecture for data intensive applications," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 369–378.

[26] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. E. Stone, J. Phillips, and W. Hwu, "GPU clusters for high performance computing," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 2009, pp. 1–8.

[27] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying NUMA and contention effects in multi-GPU systems," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 11:1–11:7.

[28] M. J. Hallock, J. E. Stone, E. R. C. Fry, and Z. Luthey-Schulten, "Simulation of reaction diffusion processes over biologically relevant size and time scales using multi-GPU workstations," *J. Paral. Comp.*, vol. 40, pp. 86–99, 2014.

[29] J. Cabezas, I. Gelado, J. E. Stone, N. Navarro, D. B. Kirk, and W. Hwu, "Runtime and architecture support for efficient data exchange in multi-accelerator applications," *IEEE Trans. Parallel Distrib. Systems*, vol. 26, pp. 1405–1418, May 2015.

[30] J. S. Meredith, P. C. Roth, K. L. Spafford, and J. S. Vetter, "Performance implications of nonuniform device topologies in scalable heterogeneous architectures," *IEEE Micro*, vol. 31, no. 5, pp. 66–75, 2011.

[31] S. Eilemann, A. Bilgili, M. Abdellah, J. Hernando, M. Makhinya, R. Pajarola, and F. Schrmann, "Parallel rendering on hybrid multi-GPU clusters," in *Eurographics Symposium on Parallel Graphics and Visualization*, H. Childs, T. Kuhlen, and F. Marton, Eds. The Eurographics Association, 2012.

[32] P. Wang, Z. Cheng, R. Martin, H. Liu, X. Cai, and S. Li, "NUMA-aware image compositing on multi-GPU platform," *Vis. Comput.*, vol. 29, no. 6-8, pp. 639–649, Jun. 2013.

[33] J. Xie, H. Yu, and K.-L. Ma, "Visualizing large 3D geodesic grid data with massively distributed GPUs," in *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*, Nov 2014, pp. 3–10.

[34] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 80–89.

[35] J. E. Stone, A. Kohlmeyer, K. L. Vandivort, and K. Schulten, "Immersive molecular visualization and interactive modeling with commodity hardware," *Lect. Notes in Comp. Sci.*, vol. 6454, pp. 382–393, 2010.

[36] J. E. Stone, K. L. Vandivort, and K. Schulten, "Immersive out-of-core visualization of large-size and long-timescale molecular dynamics trajectories," *Lect. Notes in Comp. Sci.*, vol. 6939, pp. 1–12, 2011.

[37] J. E. Stone, W. R. Sherman, and K. Schulten, "Immersive molecular visualization with omnidirectional stereoscopic ray tracing and remote rendering," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2016 IEEE International*. IEEE, May 2016.

[38] L. Le, E. H. Lee, K. Schulten, and T. Truong, "Molecular modeling of swine influenza A/H1N1, Spanish H1N1, and avian H5N1 flu N1 neuraminidases bound to Tamiflu and Relenza," *PLoS Currents: Influenza*, vol. 2009 Aug 27:RRN1015, 2010, (9 pages).

[39] L. Le, E. H. Lee, D. J. Hardy, T. N. Truong, and K. Schulten, "Molecular dynamics simulations suggest that electrostatic funnel directs binding of Tamiflu to influenza N1 neuraminidases," *PLoS Comput. Biol.*, vol. 6:e1000939, 2010, (13 pages).

[40] A. Vergara-Jaque, H. Poblete, E. Lee, K. Schulten, F. González-Nilo, and C. Chipot, "Molecular basis of drug resistance in A/H1N1 virus," *J. Chem. Inf. Model.*, vol. 52, pp. 2650–2656, 2012.