Real-Time Rendering of Molecular Dynamics Simulation Data: A Tutorial

N.Alharbi[1] , M. Chavent[2] and RS. Laramee[1]

[1]Department of computer science, Swansea University, UK
[2] Institute of Pharmacology and Structural Biology (IPBS), Toulouse, France
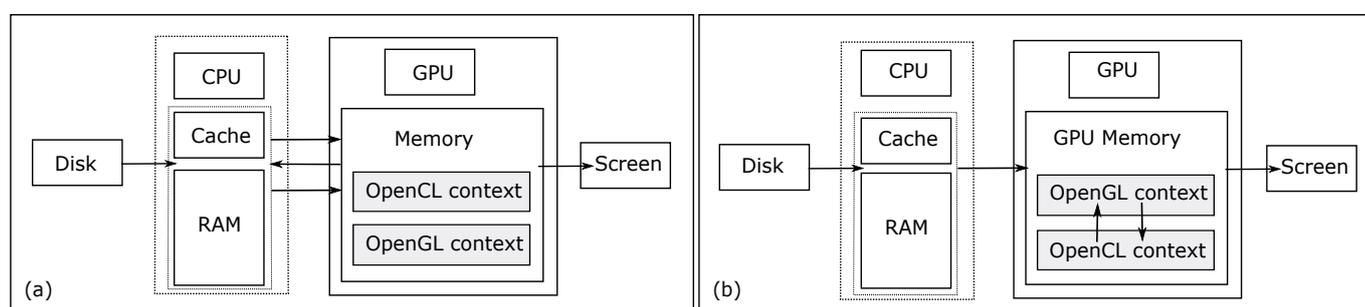
Figure 1: A traditional approach vs. an advanced approach. The arrows indicate the flow of the data throughout the visualization pipeline. a) the traditional approach requires copying the data four times per computation. The first copy is from disk to the RAM, and the remaining to exchange the data between the CPU and GPU. b) the advanced approach utilizes a mapping technique and OpenGL interoperability. The data is copied only twice: from disk to RAM and from RAM to the GPU.

Abstract

Achieving real-time molecular dynamics rendering is a challenge, especially when the rendering requires intensive computation involving a large simulation data-set. The task becomes even more challenging when the size of the data is too large to fit into random access memory (RAM) and the final imagery depends on the input and output (I/O) performance. The large data size and the complex computation processing per frame pose a number of challenges. i.e. the I/O performance bottleneck, the computational processing performance costs, and the fast rendering challenge. Handling these challenges separately consumes a significant portion of the total processing time which may result in low frame rates. We address these challenges by proposing an approach utilizing advanced memory management and bridging the Open Computing Language (OpenCL) and Open Graphics Library (OpenGL) drivers to optimize the final rendering frame rate. We illustrate the concept of the memory mapping technique and the hybrid OpenCL and OpenGL combination through a real molecular dynamics simulation example. The simulation data-set specifies the evolution of 336,260 particles over 1981 time steps occupying 8 Gigabyte of memory. The dynamics of the system including the lipid-protein interactions can be rendered at up to 40 FPS.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—...

1. Introduction and Motivation

In computational biology, Molecular Dynamics Simulation (MDS) is used to simulate the dynamics of lipids and proteins. MDS is capable of producing a high volume of MDS data resulting from simulating the motion and interaction of billions or even trillions of particles. The MDS data can be investigated utilizing various visualization techniques. The ever-increasing size of MDS output poses a number of challenges and requires data visualization scientists employing new technologies and techniques to address these challenges. For a recent MDS visualization literature review we refer to Alharbi *et al.* [AAM*17]. Thankfully, advances in commodity Computer Graphics hardware including Graphics Processing Unit (GPU) technology result in a significant acceleration of graphics applications including scientific visualization. Commodity graphics hardware is capable of processing millions of textured triangles per second [THO02]. However, even though the new hardware is

designed to ensure high rendering quality and performance, the size and nature of the data, the required processing per frame, and the rendering approaches affect the final rendering frame rate and pose three main challenges, the I/O, the computation, and the rendering challenge respectively.

In terms of the size of data, which forms the basis of the first challenge, we consider the data to be big data when it does not fit into the RAM. Data not held in RAM means it requires more frequent file I/O access which is very expensive compared to RAM access and may turn to a rendering bottleneck. The second challenge is posed by the computation requirements. Regardless of the complexity of the computation, the computation challenge is tightly coupled with the size of the data and the methods that are used to handle it. The third challenge is tightly related to the (CPU versus GPU) computation location, and the data exchange approach by which data is passed between the computational and the rendering stages. In terms of location, the computation might be performed on the CPU or on the GPU. In both cases, the data transfer method plays an essential role in determining the visualization performance.

A real-time visualization can be achieved by increasing the rendering frame rate. The higher this number, the better the user's perception of the fluidity of the scene. An optimal frame rate is around 30 fps, as the human visual system is only perceives up to about 25 images per second (called the persistence of vision) [Gol14]. If such high frame rates are achieved, the rendering is considered real-time [CLK*11].

This article addresses the big data I/O challenge, computation challenge and fast rendering by introducing a memory mapping technique and an advanced GPU approach utilizing the OpenCL interoperability feature. The simulation is performed previously by GROMACS [VDSLH*05]. We choose OpenCL because it is free, cross-platform, well documented and fast. See figure 2. The article is targeted at developers who would like to render MDS data and guides the user on how to achieve a real-time rendering. Our contributions are:

- A memory mapping technique to enhance the I/O performance.
- Illustrating the concept of Memory-Mapped Files and buffer mapping.
- Exploiting the GPU to accelerate the big data computation.
- Integrating OpenCL and OpenGL by utilizing the OpenCL interoperability feature.
- A case study using a real example involving the dynamic of lipids and proteins from a molecular dynamics simulation.
- A performance comparison of a traditional approach vs. our approach.

This article focuses on *OpenGL 4.3* and *OpenCL 1.1*, and it does not cover data visualization techniques such as filtering, sampling, Level of Detail (LoD) etc, as these techniques are beyond the scope.

The rest of the paper is organized as follows: in section 2, we provide an overview of related work. Section 3 describes the requirements of our real-time molecular dynamics rendering. In section 4 we describe the challenges and their solution, and provide a comparison of our solution with a traditional solution followed by the conclusion in section 5.
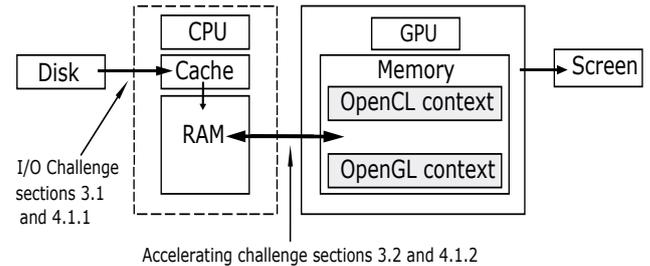
Figure 2: An overview of the challenges throughout the visualization pipeline.

2. Related Work

GPU technology plays a significant role in enhancing the performance of MDS data computation and visualization. Molecular modeling algorithms that exploit the GPU have been largely successfully compared to alogorithms that use CPU alone. See Stone *et al* [SHUS10] for an overview. Recent developments utilizing GPUs address performance limitations and herald the next generation of molecular visualization solutions according to Chavent *et al* [CLK*11].

In general, a number of tutorials have been published illustrating GPU technology and how to code GPU-based programs. Stone *et al.* [SGS10] provide an overview of the key architectural features of recent microprocessor designs and describe the programming model and abstractions provided by OpenCL. Shreiner *et al* [SSKLK13] and Wright *et al.* [WJHSL10] can be considered OpenGL specification-based references. The tutorials, of Shreiner *et al* and Wright *et al*, are accompanied by a collection of C++ code examples. Shreiner *et al.* [SSKLK13] focus on the latest methods and techniques for OpenGL 4.3 application development. They describe every stage of the programmable rendering pipeline and cover shading techniques found in the OpenGL Shading Language (GLSL) including the compute shader which is introduced in OpenGL 4.3. The compute shader runs in a separate stage of the GPU and allows an application to make use of the power of the GPU for general purpose work that may or may not be related to graphics. Wright *et al.* [WJHSL10] is designed for readers who are learning computer graphics through OpenGL and readers who may already be familiar with graphics but want to learn about OpenGL, however, readers are required to understand computer programming in C++. In addition to the essential OpenGL tutorials, each book involves a number of various topics that cover some advanced concepts including high-performance rendering techniques and GPU analysis and OpenGL debugging tools. These tutorials cover the compute shader which can be used to fulfil computational requirements. There are a number of efficient frameworks that are essentially designed to harness GPU to perform computational tasks.

Weiskopf [Wei06] consists of 5 chapters and it is designed to be a starting point to understand the GPU-based visualization. The main parts of the book focus on efficient GPU-based visualization techniques for interactive exploration of 3D scalar and vector fields, and for enhancing visual perception of non-photorealistic rendering. A number of useful topics are discussed throughout the book

including parallelization on clusters with several GPUs, adaptive rendering methods and multi-resolution methods.

Gaster *et al.* [GHK*12] guide readers, by example, on how to program heterogeneous environments with OpenCL and define the concepts that the readers need to understand before starting to program any heterogeneous system. They concentrate on illustrating the OpenCL framework in a disconnected context. However, Gaster *et al.* [GHK*12] briefly discuss the concept of sharing a context between OpenCL and OpenGL in some of their examples. Munshi *et al.* [MGMG11] explain how OpenCL 1.1 can be used to express a wide range of parallel algorithms. They cover the entire OpenCL 1.1 specification including advanced OpenCL features i.e. OpenCL interoperation which enables developers to access and manipulate OpenGL buffers from a shared context. Scarpino [Sca11] involves 16 chapters illustrating the OpenCL language by example. The first 10 chapters explore the OpenCL language following by four chapters that show how OpenCL can be used to perform large-scale tasks. The last two chapters focus on the OpenCL interoperation and show how OpenCL can be used to accelerate OpenGL applications. We utilize the OpenCL interoperation approach in the same manner used in [MGMG11, Sca11], however, we also integrate the *glMapBuffer()* and the MMPs in the solution to enhance the final rendering performance.

Schatz *et al.* [SMK*16] present a novel method that enables users to explore one trillion particles. Their method is based on an advanced focus and context technique and a camera space visualization i.e. only particles that surround the camera are selected. They utilize a dual-GPU configuration that splits the workload between the GPUs based on the type of data. Schatz *et al.* decode and visualize different attributes of the MDS data and allow users to explore the data-set by moving the camera throughout the scene. However, even though they apply their method to a dynamic data-set, the current version of the method handles the data-set as a volume.

Hrabcak and Masserann [HM12] illustrate the concept of asynchronous buffer transfers to enhance the performance of two way data traffic i.e uploading and downloading data to and from the graphics card. The proposed approach focuses on optimizing data traffic per frame by utilizing two techniques: i) a map buffer, and ii) a swapping buffer technique. This approach is useful for applications that render a static data-set, or applications that separate computation from rendering by performing computations on the CPU. We utilize the map buffer technique to stream data from the CPU to the GPU.

Movania and Feng [MF12] describe a method for implementing and visualizing real-time deformation. They utilize a modern GPU transform feedback mechanism. The transform feedback mechanism enables developers to feed an OpenGL buffer, the so-called transform feedback buffer (TFBB), via either the vertex or geometry shader which means the calculation must be done in the vertex (or geometry) shader. One of the most significant advantages of the transform feedback is that the TFBB content can be rendered directly from the buffer which eliminates unnecessary traffic between the CPU and GPU. However, this approach is not feasible for intensive computations or computing big data and the vertex and geometry shaders tend to be used for light computations as the
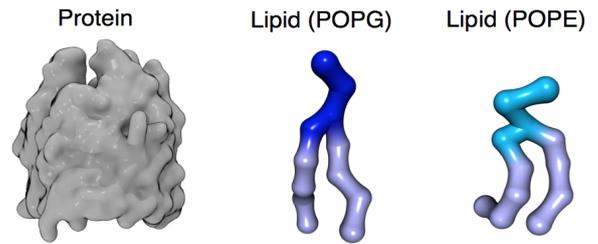
Figure 3: Structure of molecules: Protein, POPG and POPE types (left to right). The hallow of protein particles is used to construct the protein surface. An advance ball and stick is used to represent the POPG and POPE types. The Protein image is generated with VMD [HDS96], and the Lipid type images are generated with UnityMol [LTDS*13] based on the hyperballs representation [CLK*11].

vertex shader and the geometry shader are limited to a single vertex and primitive respectively.

In this article, we focus on the performance aspect of real-time MDS rendering. We cover a number of techniques that can enhance the performance throughout the visualization pipeline. These techniques are Memory-Mapped Files (MMFs), a Mapped buffer, and integrating OpenGL and OpenCL by allocating a shared context.

3. Real-Time Visualization Requirements

Scientific visualization research utilizes a well-defined pipeline to create the final representation of the input data-set. For simplicity, we use a basic definition for the visualization pipeline. We consider the visualization pipeline consisting of three main stages (Moreland [Mor13]): i) the data I/O stage, ii) the computation stage, and iii) the rendering stage.

In this section, we briefly describe our data set, a programming environment for developing a solution, and three popular frameworks that can be utilized through the visualization pipeline to achieve the real-time rendering.

Data Description Our data-set represents biological dynamics of lipids and proteins at high resolution. The system's dimensions are 116.01860 x 116.01860 x 10.13590 nano meters (x, y, and z respectively) and the individual trajectories reflect the evolution of 336,260 particles over 1,981 nanoseconds (ns). The system consists of three molecule types: one protein type, and two lipid types (POPE type and POPG type). The protein type involves 256 protein molecules while the lipid POPE and the POPG type consist of 14,354 and 4,738 molecules respectively. The hierarchy of the structures is described below:

- The protein structure consists of 256 proteins. Each protein has 171 residues. Each residue consists of 1 to 3 particles. A single protein consists of 344 particles which results in (256x344) 88,064 particles in total.
- The lipid structure consists of 19,092 lipids. Each lipid contains 3 groups: i) a head group (2 particles), ii) a tail group (5 particles), and iii) a second tail group (6 particles). Each lipid molecule has 13 particles which results in (19,092x13) 248,196 lipid particles in total.

Figure 3 provides a depiction of the structure of a protein, POPG type and POPE type. In terms of the size of the data, the data-set contains more than 666 million vertices that occupy 8 Gigabytes of memory.

Our requirement is to visualize the interaction between lipids and proteins on-the-fly. A protein's body is represented by a sphere glyphs while shaded spheres are used to represent lipid particles. The interaction between lipids and proteins is represented by color-mapping the interacting particles (Figure 4).
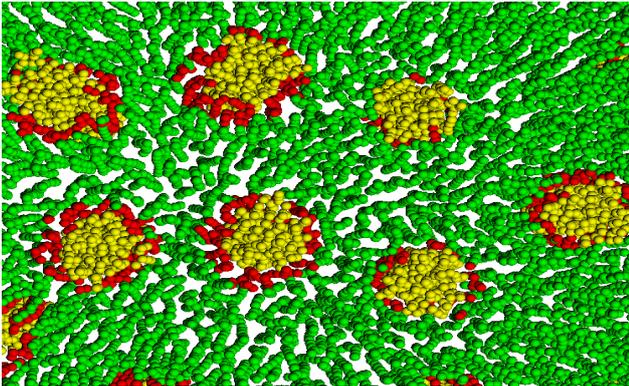
Figure 4: Protein-Lipid interaction. The protein particles are represented by gold spheres and the lipid particles are represented in green. The red spheres represent the lipid particles that interact with the protein particles within 0.6 angstrom. The image is generated with the tutorial accommodated example code.

Development Framework There are a number of a powerful programming languages that can be utilized to implement the visualization pipeline. In this article, the implementation of these stages is done in C++ utilizing the Qt Framework (v. 5.7) [Qt17a]. Qt is chosen for a number of reasons:

- Qt is cross-platform.
- Qt provides developers with a free development GUI called Qt Creator [Qt17b].
- In addition to the QOpenGlWidget class, Qt provides developers with a set of QOpenGL* classes that interface with most of the OpenGL objects including the shader.
- Qt has up-to-date on-line documentation.
- Qt for application development is available under two licence agreements: commercial and open source licenses.

The tutorial code is implemented following Laramee's [Lar10] concise coding conventions which considerably enhance the code readability and result in well organized source code. The full source code and the supplementary materials can be downloaded from the following **URL** http://cs-svr1.swan.ac.uk/~csnai/RealTimeRendering

3.1. File I/O and Memory Management

In general, a standard C or C++ library is utilized for the implementation of file I/O through the FILE class and iostream library. In C++, a developer can open a file when instantiating a stream. The stream is utilized to perform the I/O operations and finally the file is closed automatically on destruction of the

stream [LK00]. However, the standard C++ iostream has some limitations which must be taken into account. See section 4. Even though Qt addresses this limitation with the QFileDevice class, for learning purposes we decide to implemet the file I/O solution using the Boost library [Boo17a]. The Boost library involves a collection of useful libraries [DD10, DD12] that can be used as an alternative to the C++ Standard Template Library (STL). The Boost iostream Library address the STL iostream limitation via the Memory-Mapped Files classes: 1) *mapped_file_params* 2) *mapped_file_source*, 3) *mapped_file_sink* and 4) *mapped_file*. They provide access to memory-mapped files on Windows and the Portable Operating System Interface (POSIX) systems [Boo17a]. The *mapped_file_params* class is responsible for the parameters used to open a memory-mapped file. The *mapped_file_source*, *mapped_file_sink* and *mapped_file* classes provide read, write and read-write access to memory-mapped files respectively. We propose these classes to enhance the file I/O performance and reduce the memory management limitations.

3.2. GPU Accelerator

The GPU is essentially designed to accelerate the rendering process, however, it is widely used for general purpose computation as well. There are a number of Application Programming Interfaces (APIs), such as CUDA [CUD17], OpenCL [Gro17a], OpenGL [Gro17b], and the promising graphics and computing API Vulkan [Gro17c], that can be utilized to program a commodity GPU. They enable developers to harness GPU parallelism through straightforward C code that runs in thousands or millions of parallel invocations. Even though all these frameworks provide the developer with an interface to communicate with graphics cards, each one is designed for a particular goal. CUDA and OpenCL are designed to accelerate the computation process. However, they can access OpenGL buffers and manipulate buffer content before it is rasterized. OpenGL concentrates on the graphics card rendering pipeline. However, since OpenGL 4.3 developers are able to perform computation tasks utilizing the shader. Vulkan is known as the new generation of OpenGL. It provides a comprehensive computation and rendering framework. Our solution relies on OpenGL and the OpenCL API as they are supported by all commodity graphics cards. The proposed solution benefits from the concept of sharing data. See Figure 5.

4. Rendering Big MDS Data: A Proposed Solution

As mentioned, rendering big MDS data involves three main challenges. These challenges are distributed throughout the visualization pipeline and are tightly related to each other. Our solution utilizes an advanced memory techniques that reduces the data traffic during the visualization and utilizes a GPU feature that enables OpenCL to access OpenGL buffers. See Figure 1. In this section, we describe these challenges and illustrate how can they be addressed by our proposed solutions.

Data Size Challenge Even though the size of the data can be reduced before it is sent to graphics card by applying filtering and LoD for example, the essential issue, before hand, is finding an optimal approach to extract the data from big files. The standard C++ library provides developers with many file I/O operations such as open, read, seek, write etc. via the iostream class. However, consider a sequential read of a file on disk using the standard system
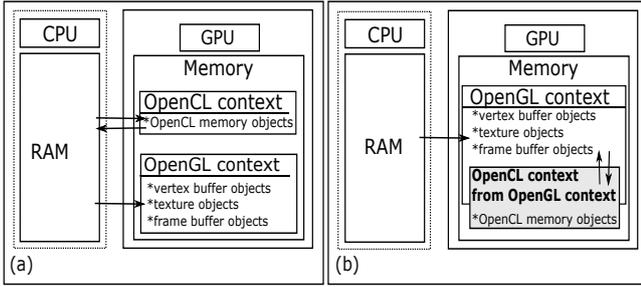
Figure 5: OpenCL and OpenGL integrating approaches. a) OpenCL context and OpenGL context are separated. This approach requires exchanging data between CPU and GPU. b) OpenCL creates its own context from an existing OpenGL context. This shared context allows OpenCL to access the shared OpenGL objects in GPU memory. The data is computed and visualized solely on the GPU.

calls open, read, and write. Each file access requires a system call and disk access which is considered one of the iostream limitations. Alternatively, we can use a virtual memory techniques to treat file I/O as a routine memory access. This approach, known as memory mapping a file, allows a part of the virtual address space to be logically associated with the file. Memory mapping a file is accomplished by mapping a disk block to a page (or pages) in virtual memory [SGGS98]. Another limitation of the iostream is that it requires buffering the data from files (the buffer is allocated in RAM then it is passed as a parameter to the read function) before it can be used, whereas memory mapped files provide us with a pointer to the required data in the virtual memory space (Figure 6). We decide to use a Memory Mapped file as it has two advantages: first it reduces the number of system calls, second it provides us with a pointer to a process's address space instead of copying all of the data into RAM.
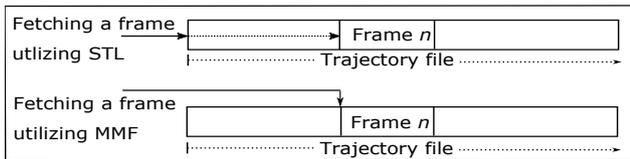
Figure 6: Data I/O stage. The process of fetching data from harddisk utilizing standard C++ library, and fetching data utilizing MMFs. The MMF communicates directly with the OS and provides a random access. STL copies all of the data from file to RAM before it can be used. MMF maps all or part of a file to virtual address space, that can be accessed by CPU, and provides developer with a pointer to that space. Unnecessary data may be skipped rather than read into RAM.

Separate Memory Challenge OpenCL and OpenGL are widely used for heterogeneous computing and graphics development. Both *OpenCL* and *OpenGL* require a valid *context* in order to perform their functionalities. The OpenGL *context* is an object that stores information (e.g. buffer binding) about an OpenGL state. An application may have one or more OpenGL contexts [SAL14]. A *context* can be created by invoking *clCreateContext()* and *glCreate-Contex()* for *OpenCL* and *OpenGL* respectively which results in

two disconnected contexts. Typically the results produced by the OpenCL computation are used as input by OpenGL to render the data. The data computed using OpenCL is located on the GPU and cannot be directly accessed by the OpenGL runtime as they are created in a disconnected mode. As a result, the data has to be explicitly copied from the GPU to the CPU memory. OpenGL can then use the data as an OpenGL buffer for rendering on the GPU. This approach requires data to be transferred between the CPU and GPU each time the application switches from OpenCL compute to OpenGL rendering. Exchanging data between the CPU and GPU adds a significant execution overhead and affects the performance of the application [UGK14]. To handle this issue OpenCL provides an optional extension to share memory objects between OpenCL and OpenGL [MGMG11]. The extension enables OpenCL to create a context from an existing OpenGL context. This approach generates a bridge between OpenCL and OpenGL where the OpenCL can access the OpenGL objects in GPU memory. Leveraging the OpenCL-GL shared context results in a 2.2X performance increase [UGK14]. Figure 5 illustrates the concept of OpenCL and OpenGL shared context.

4.1. Mapped Memory and GPU framework interoperability: Illustration by Example

Decompressing the XTC file MDS systems produce trajectory files that contain the atomistic behavior of molecular systems. Each system adopts its own formats for storage of trajectory data [LAS*14]. GROMACS [HKVDSL08] defines different formats such as TRR and XTC format. XTC is a lossy compression format [TGCC95] and it can be read by special libraries like *xdrfile* [GRO09].

The XTC format is optimized for reducing the size of the XTC files. However, it is not practical to read the trajectories from the XTC file as it requires decompressing the trajectory each time we access the file. The first step in this tutorial is to decompress the trajectory file and save the result in a new file via *xdrfile* and the Boost library. The *xdrfile* library consists of three classes: xdrfile, xdrfile_xtc and xdrfile_trr. The xdrfile_xtc class encapsulates the decompressing processes and provides a straightforward function for reading the trajectory data frame by frame (See the supplementary materials). The output is written to a new file after calculating the decompressed frame size by utilizing the MMF functionality.

4.1.1. Reading and Writing Data Using MMF

Logically, we should start by illustrating how to utilize the Boost library to read files, however, reading and writing using Boost is transferable and the concept is applicable for general reading and writing. As mentioned MMFs maps a file, in reading and writing mode, to the virtual memory space. The Boost Library supports the writing mode via two classes: mapped_sink and mapped_file. The former is designed for writing mode only whereas the latter can be utilized to simultaneously read and write a file. If the file is entirely opened, these classes return a pointer of const char type that points to the first byte of the file in virtual memory. However, it is not practical to open the entire file to access a portion of it. Instead of reading the entire file, the required part can be mapped to a virtual memory space, however, the developer is responsible for writing an algorithm that calculates the right position of the required data in the virtual memory space based on the operating system's virtual memory allocation granularity. i.e. the returned pointer always

Figure 7: Calculating the position of the second frame in virtual memory space.

points to a value that must be a multiple of the operating system's virtual memory allocation granularity. The developer might need to handle this issue by adding an offset to the pointer as in Figure 7. The implementation of this functionality requires the size of the decompressed frame and the value of the operating system alignment. In our case, each frame represents the position of 336,260 particles in 3D space. As all the frames hold the same amount of data, the frame size is calculated only once. The frame size can be obtained by multiplying the number of particles by two variables: the number of dimensions of the domain space, and the size of the float data type.

$$Frame\ Size = p \text{ x } c \text{ x } FLOAT\_SIZE$$

Where p and c are the particle number and the number of components to represent the particle position respectively. The third parameter $FLOAT\_SIZE$ is the size of the float type with respect to the specification of the operating system. Applying the above, the decompressed frame occupies 4,035,120 bytes of memory. The alignment value can be obtained directly from the Boost class by which the file is opened. Our operating system (macOS Sierra) aligns data in a blocks of 4,096 bytes.

Calculating The First Byte of a Frame (a time-step) Operating systems do not allow random access to file data. They provide developers with a pointer that points to the first byte of the required block or blocks. Each block must be a multiple of the operating system's virtual memory allocation granularity which means we cannot ensure that the first byte of a frame will be directly returned. In order to get the correct position of a frame in the file, for either reading or writing, we need to calculate the position. First, we find the index of a block that involves the first byte of a frame. Then we calculate the correct position of the frame. We use the above values to calculate the correct position to write the frame data to the new file. To illustrate, in our case, the first frame occupies 985 blocks and 560 bytes from block number 986. To write the second frame we need to access the block number 986 at byte 560. See Figure 7. However, the operating system can not return a pointer to a random position. Alternatively, we open the file at the block 986 (which involves 560 bytes of the first frame) then we write the next frame starting at the returned pointer plus 560 bytes.

Paging the File to a Virtual Memory Space Paging the file means mapping a particular part of the file to a virtual memory space. The paging requires two parameters: the starting position (must be a multiple of the alignment), and the size of the required data (in bytes). The bigger the page size the smaller the number of mapping operations. The performance can be optimized by measur-

ing the system performance against various page sizes. Our optimal page size is 14,857,600 bytes. Each page involves three frames, the following is the number of frames per page:

$$Number\ of\ Frames\ Per\ Page = PS/FS$$

Where PS indicates the page size, and FS indicates the frame size.

Writing Data to File First, a file for storing the decompressed data must be opened via either mapped_file or the mapped_file_sink class. These classes obtain the properties of the new file from the mapped_file_params class through its members: 1) path, 2) flags, 3) offset, 4) length and 5) new_file_size [Boo17b]. The path holds the file name and location on the hard-disk. The flags identify the access mode (read, write, and read/write). The offset identifies where the mapping begins. The length identifies the size of the mapped data (the page size). The new_file_size identifies the size of the entire file. The entire size of the file can be derived by multiplying a frame size by the total number of frames.

If the file does not exist, a new file is created automatically and a pointer to the first byte of the file in the virtual memory space is returned. If the file already exists a pointer to the first byte of the mapped part is returned. The compressed data can be copied to virtual memory via the *memcopy* function or by casting the pointer type and assigning the data to it directly.

Reading Data from File Mapping a file for reading only follows the same manner described above, however, we utilize the mapped_file_source class which is designed to map a file for reading only. The mapped_file_source returns a read only const char pointer. Reading the pointer value does not require any *memcopy* overhead, the returned pointer can be converted into any other type including user defined types.

4.1.2. Mapping OpenGL Buffers

An OpenGL buffer is a data container that belongs to the graphics card. A graphics card has its own memory which is used to store all the OpenGL buffer types and the other OpenGL objects. Traditionally, after creating the OpenGL buffer the data must be transferred or uploaded from RAM to the OpenGL buffer in the GPU memory. There are two ways to upload and download data to the GPU memory. See Figure 8. The first way is to use the *glBufferData* and *glBufferSubData* functions [HM12]. The other way to upload data to the GPU is to get a pointer to the internal drivers' memory with the functions *glMapBuffer* and *glUnmapBuffer*. This pointer can be used to fill the buffer directly which means we save one copy per memory transfer. *OpenGL 4.3* provides three functions to map and unmap buffers: 1) *glMapBuffer()*, 2) *glMapBufferRange()*, and 3) *glUnmapBuffer()*. The *glMapBuffer()* function maps the entire buffer data to a memory whereas the *glMapBufferRange()* maps only a subset of the buffer which means there is no need to re-upload the buffer completely. Finally, after filling the buffer the mapping can be released via the *glUnmapBuffer()*. The *glMapBuffer()* and *glMapBufferRange()* require a valid and binded OpenGL buffer. A valid buffer means that the buffer has a valid name/id that is obtained via *glGenBuffers()*. A Binded buffer means that the buffer is linked with the current *context* by calling *glBindBuffer()*. We note that the buffer size must be defined by allocating some memory via *glBufferData()* before using *glMapBuffer()* and *glMapBufferRange()*. When utilizing *glMapBuffer()* there may be

significant performance advantages if a NULL pointer is passed to *glBufferData()* instead of using an initial data [SSKLK13].

Figure 8: Data flow via *glBufferData()* and *glMap*()*. Copying is indicated by a solid arrow and mapping is represented by a dashed arrow. a) *glBufferData()* copies a data from a program space to a buffer or a subset of it. b) *glMap*()* returns a pointer to a memory space that is accessible by the GPU.

4.1.3. OpenGL shared context

On the GPU side, our solution relies on the concept of objects sharing. By default the OpenGL framework and OpenCL framework generate two unique *contexts*, the OpenGL *context* and the OpenCL *context*. The OpenGL context is responsible for maintaining the OpenGL objects state and the OpenCL context is responsible for managing the OpenCL memory objects. They are isolated from each other. i.e the OpenGL context is not accessible by the OpenCL framework and the OpenCL context is invisible to the OpenGL framework which means the computation process and the rendered process cannot be synchronized on the GPU. See Figure 5. An OpenCL extension addresses this limitation via the so-called OpenCL interoperability [MGMG11] . This extension enables the OpenCL framework to access the OpenGL *context*. At a high level, the OpenGL interoperability can be achieved by, first, creating an OpenGL *context* and initializing an OpenCL *context* from it. As not all devices support this feature. A device needs to be queried to determine whether it supports OpenGL interoperability. The *clGetDeviceInfo()* function is used to obtain variety of information about the device (a CPU, a GPU, a Digital Signal Processor (DSP)). *clGetDeviceInfo()* can return the supported extensions name in a string for the CL_DEVICE_EXTENSIONS property. The cl_APPLE_gl_sharing extension name is used for Mac OS and the cl_khr_gl_sharing name is used for other systems [Gro10]. A proper name, cl_APPLE_gl_sharing or cl_khr_gl_sharing, must appear in the returned string with respect to the host OS. If the OS supports the OpenGL interoperability, then we can utilize it.

To benefit from this feature we need to create two things: 1) create an OpenCL *context* from an OpenGL *context*, and 2) create an OpenCL object from an existing OpenGL object. Creating an OpenCL *context* from an OpenGL *context* requires two steps: First, the OpenGL *context* must be initialized and current before creating the OpenCL *context* [Sca11]. Second, the OpenCL *context* must be configured in order to enable the OpenGL framework to access the OpenGL objects. The configuration is achieved by setting the cl_context_properties structure with the proper values with respect to the OS. On Mac OS, the cl_context_properties structure requires only one property: CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE. However, the value associated with this property must have the data type CGLShareGroupObj, and it can be acquired via the function

CGLGetShareGroup(). This function requires a *CGLContextObj* structure, which can be obtained by calling *CGLGetCurrent-Context()*. The following code shows how these functions work together:

```
CGLContextObj glContext = CGLGetCurrentContext();
CGLShareGroupObj shGroup = CGLGetShareGroup(glContext);
cl_context_properties prop[] = {
    CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE,
    (cl_context_properties)shGroup,
    0};
```

After the cl_context_properties structure is set, an OpenCL context that is capable of accessing OpenGL data can be created via *clCreateContext()* (for more detail and for Linux and Windows configuration see the supplementary material). Hence the OpenCL context is created from an OpenGL context. The OpenCL framework is able to access number of the OpenGL objects and modify them. OpenCL can access and modify vertex buffer objects (VBOs), texture data (texture objects) and pixel data (renderbuffer objects) by creating the corresponding OpenCL memory object via one of four functions: *clCreateFromGLBuffer()*, *clCreateFromGLTexture2D()*, *clCreateFromGLTexture3D()*, or *clCreateFromGLRenderbuffer()*. These functions require the OpenGL object to be already initialized before they can create an OpenCL object from it. Hence an OpenCL memory object is created from an OpenGL object. Any change in its content affects the OpenGL object content.

Synchronization Sharing data between OpenGL and OpenCL doesn't mean they can access the shared data at the same time. i.e. if OpenCL is manipulating a memory object data created from an OpenGL vertex buffer, then OpenGL can't access the vertex buffer. OpenCL supports synchronization through two functions: 1) *clEnqueueAcquireGLObjects()*, and 2) *clEnqueueReleaseGLObjects()*. The main task for the two functions is to lock and unlock access to the OpenCL memory objects. The first function ensures that the OpenCL will have exclusive access to the data. The second function releases the data and enables OpenGL and other processes to access it. Before acquiring a lock we have to ensure that all OpenGL routines have completed their operation by invoking *glFinish()*. And after releasing the data from the lock we need to ensure that all OpenCL commands have completed their operation by calling *clFinish()*.

4.2. Performance Test

Performance measurement is an important step that helps in optimizing code and decide between available methods. We provide three performance tests and show how do they result in the frame rate : i) file I/O approaches (STL vs. MMFs), ii) OpenGL interoperability (OpenCL-GL separated context vs. shared context), and iii) uploading data to OpenGL buffers (Buffer data vs. Map Buffer). The user can choose between these approaches in order to investigate how do they work under different configurations. See figure 9. Due to the nature of the data which is dynamic data-set, and in order to provide a smooth transition of the dynamics of the data more than two time-steps per second must be rendered. We render ~ 4 time-steps per second. The process of rendering each time-step involves the following: reading particles position from the file, computing the interaction between the particles, updating the OpenGL buffer based on the computation result, and finally rendering the result.

I/O		CL-GL context		GL		Total
STL	MMF	Default	interoperability	Buffer Data	Map Buffer	rendering rate FPS
✓		✓		✓		21 fps
✓		✓			✓	24 fps
✓			✓	✓		42 fps
✓			✓		✓	42 fps
	✓	✓		✓		19 fps
	✓	✓			✓	21 fps
	✓		✓	✓		41 fps
	✓		✓		✓	41 fps

Table 1: An overview of the performance test result based on different options. Utilizing shared context results in no overhead of uploading data to the GPU for rendering.

The frame rate is measured per second, and it has been averaged by total time-step. All performances are measured in millisecond (ms)
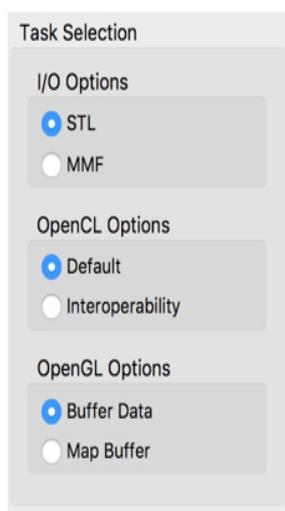
Figure 9: Task selection. Each task has two options. The user can switch between these to see how do they affect the total rendering rate in FPS.

except the final rendering rate which is measured in frames per second (FPS). The performance tests are performed on MacBook Pro Retina, 13-inch, Early 2015 with a 2.7 GHz Intel Core i5 processor and 8 GB of memory (1867 MHz DDR3). The graphics card is Intel Iris Graphics 6100 (1.5 GB of memory). Our MDS data sample involves a trajectory of 336260 particles over 1981 time-steps. Each position is represented by a 3D floating point vector. The data occupies 8 Gigabyte and it is stored in a binary file. For each time-step the program performs three main tasks: 1) reading the data of the current time-step (so called frame) from the file. 2) for each protein we perform a Protein-Lipids interaction test. 3) visualizing the interaction result and the global molecular dynamics. The performance overhead is associated with the I/O and the computation tasks. The average performance of the I/O task is 17 ms per time step. The computation consumes ∼35 ms of the total rendering time per time step. The rendering frame rate is steady at ∼20 FPS by utilizing the OpenCL-GL default configuration while it almost doubled by utilizing the OpenCL-GL interoperability option.

STL vs. MMFs The program requires fetching the data from the file frame by frame. The performance of fetching the frames varies depending on the position of the frame in the file. STL shows a stable performance (17 ms) for extracting a single frame from the file at any position. MMF requires 5 ms to 20 ms to fetch one frame from the file.

OpenCL-GL shared context vs. separated context The OpenCL-GL context test includes time required to transfer data between the CPU and GPU. The separated context approach requires 40 ms to perform the computations and to transfer the data between the CPU and GPU per time step. The OpenCL-GL shared context results in slight enhancement by completing this task in ∼30 ms. This enhancement comes from the fact that, in the shared context approach, the data is sent from the CPU to the GPU once where as the separated approach requires the data to be uploaded from the CPU to the GPU twice: the first time to perform the computation and the second time to render the data.

Buffer data vs. Map data The final performance test is associated with the rendering. The new particles need to be rendered with respect to the interaction result. Based on the selected computation approach (Open CL-GL separated context or shared context), this task will require uploading the data from the CPU to the GPU. The Map buffer option requires 1 ms per time step whereas the buffer data requires 3 ms.

5. Conclusion

Visualizing Proteins-lipid interaction of large data-sets of molecular dynamics simulation is a challenge. A number of articles have illustrated different techniques in order to enhance the performance of rendering large data set. OpenCL is utilized to exploit the GPU for the computation purposes and OpenGL is widely used for rendering the data-set including the interaction result. Harnessing the interoperability feature of OpenCL and the OpenGL is a key point in boosting the performance of rendering molecular dynamics. We illustrated the concept of memory mapping by proposing the MMF method from Boost library and the *glMap*()* functions from OpenGL API. The illustration covers the usage of OpenCL as a means to perform computation tasks in GPU and we provide and example of utilizing the OpenCL-GL interoperability to enhance the data traffic performance between the CPU and GPU. The OpenCL-GL interoperability and the *glMap*()* functions have a great impact in the final rendering frame rate.

Future Work In this work we discuss a possible approach that can enhance dynamic data rendering frame rate by integrating the MMF, OpenGL-CL interoperability, and OpenGL Map buffer. The Memory Mapped File versus standard buffered I/O and random access I/O requires further investigating. Utilizing the 16-bit half-float to address the data size and to see how it could affect the performance is also future work.

References

[AAM*17] ALHARBI N., ALHARBI M., MARTINEZ X., KRONE M., ROSE A., BAADEN M., LARAMEE R. S., CHAVENT M.: Molecular visualization of computational biology data: A survey of surveys. Eurovis short papers, 2017 forthcoming. 1

[Boo17a] BOOST: Boost library, 2017. URL: http://www.boost.org/. 4

[Boo17b] BOOST: Boost library, 2017. URL: http://www.boost.org/doc/libs/1_50_0/libs/iostreams/doc/classes/mapped_file.html. 6

[CLK*11] CHAVENT M., LÉVY B., KRONE M., BIDMON K., NOMINÉ J.-P., ERTL T., BAADEN M.: Gpu-powered tools boost molecular visualization. *Briefings in Bioinformatics* (2011), bbq089. 2, 3

[CUD17] CUDA N.: Nvidia cuda, 2017. URL: https://developer.nvidia.com/cuda-toolkit. 4

[DD10] DEMMING R., DUFFY D. J.: *Introduction to the Boost C++ Libraries; Volume I-Foundations*. Datasim Education BV, 2010. 4

[DD12] DEMMING R., DUFFY D. J.: *Introduction to the Boost C++ Libraries - Volume II - Advanced Libraries*. Datasim Education BV, 2012. 4

[GHK*12] GASTER B., HOWES L., KAELI D. R., MISTRY P., SCHAA D.: *Heterogeneous Computing with OpenCL: Revised OpenCL 1*. Newnes, 2012. 3

[Gol14] GOLDSTEIN E. B.: *Cognitive Psychology: Connecting Mind, Research and Everyday Experience*. Nelson Education, 2014. 2

[GRO09] GROMACS: Xtc library, 2009. URL: http://www.gromacs.org/Developer_Zone/Programming_Guide/XTC_Library. 5

[Gro10] GROUP K.: Opengl, 2010. URL: https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/EXTENSION.html. 7

[Gro17a] GROUP K.: Opencl, 2017. URL: https://www.khronos.org/opencl/. 4

[Gro17b] GROUP K.: Opengl, 2017. URL: https://www.khronos.org/opengl/. 4

[Gro17c] GROUP K.: Vulkan, 2017. URL: https://www.khronos.org/vulkan/. 4

[HDS96] HUMPHREY W., DALKE A., SCHULTEN K.: VMD: visual molecular dynamics. *Journal of Molecular Graphics 14*, 1 (1996), 33–38. 3

[HKVDSL08] HESS B., KUTZNER C., VAN DER SPOEL D., LINDAHL E.: Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation 4*, 3 (2008), 435–447. 5

[HM12] HRABCAK L., MASSERANN A.: Asynchronous buffer transfers. In *OpenGL Insights*. AK Peters/CRC Press, 2012, pp. 391–414. 3, 6

[Lar10] LARAMEE R. S.: Bob's concise coding conventions (c3). *Advances in Computer Science and Engineering (ACSE) 4*, 1 (2010), 23–26. 4

[LAS*14] LUNDBORG M., APOSTOLOV R., SPÅNGBERG D., GÄRDENÄS A., SPOEL D., LINDAHL E.: An efficient and extensible format, library, and api for binary trajectory data from molecular simulations. *Journal of computational chemistry 35*, 3 (2014), 260–269. 5

[LK00] LANGER A., KREFT K.: *Standard C++ IOStreams and locales: advanced programmer's guide and reference*. Addison-Wesley Professional, 2000. 4

[LTDS*13] LV Z., TEK A., DA SILVA F., EMPEREUR-MOT C., CHAVENT M., BAADEN M.: Game on, science-how video game technology may help biologists tackle visualization challenges. *PloS one 8*, 3 (2013), e57990. 3

[MF12] MOVANIA M. M., FENG L.: Real-time physically-based deformation using transform feedback. In *OpenGL Insights*. AK Peters/CRC Press, 2012, pp. 231–246. 3

[MGMG11] MUNSHI A., GASTER B., MATTSON T. G., GINSBURG D.: *OpenCL programming guide*. Pearson Education, 2011. 3, 5, 7

[Mor13] MORELAND K.: A survey of visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics 19*, 3 (2013), 367–378. 3

[Qt17a] QT: Licensing, 2017. URL: https://www.qt.io/licensing/. 4

[Qt17b] QT: Qt creator ide, 2017. URL: https://www.qt.io/ide/. 4

[SAL14] SEGAL M., AKELY K., LEECH J.: The opengl r graphics system: A specification. the kronos group, 2014. 5

[Sca11] SCARPINO M.: *OpenCL in Action: How to Accelerate Graphics and Computations*. Manning Publications, Nov. 2011. URL: http://amazon.com/o/ASIN/1617290173/. 3, 7

[SGGS98] SILBERSCHATZ A., GALVIN P. B., GAGNE G., SILBERSCHATZ A.: *Operating System Concepts*, vol. 4. Addison-wesley Reading, 1998. 5

[SGS10] STONE J. E., GOHARA D., SHI G.: Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering 12*, 3 (2010), 66–73. 2

[SHUS10] STONE J. E., HARDY D. J., UFIMTSEV I. S., SCHULTEN K.: Gpu-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling 29*, 2 (2010), 116–125. 2

[SMK*16] SCHATZ K., MÜLLER C., KRONE M., SCHNEIDER J., REINA G., ERTL T.: Interactive visual exploration of a trillion particles. In *Large Data Analysis and Visualization (LDAV), 2016 IEEE 6th Symposium on* (2016), IEEE, pp. 56–64. 3

[SSKLK13] SHREINER D., SELLERS G., KESSENICH J., LICEA-KANE B.: *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013. 2, 7

[TGCC95] THIEL W., GREEN D., CLEMENTI E., CORONGIU G.: In methods and techniques in computational chemistry: Metecc-95. *Eds. E. Clementi, G. Corongiu, STEF, Cagliari* (1995), 141. 5

[THO02] THOMPSON C. J., HAHN S., OSKIN M.: Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on* (2002), IEEE, pp. 306–317. 1

[UGK14] UKIDAVE Y., GONG X., KAELI D.: Performance evaluation and optimization mechanisms for inter-operable graphics and computation on gpus. In *Proceedings of Workshop on General Purpose Processing Using GPUs* (2014), ACM, p. 37. 5

[VDSLH*05] VAN DER SPOEL D., LINDAHL E., HESS B., GROENHOF G., MARK A. E., BERENDSEN H. J.: Gromacs: fast, flexible, and free. *Journal of computational chemistry 26*, 16 (2005), 1701–1718. 2

[Wei06] WEISKOPF D.: *GPU-Based Interactive Visualization Techniques (Mathematics and Visualization)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 2

[WJHSL10] WRIGHT JR R. S., HAEMEL N., SELLERS G. M., LIPCHAK B.: *OpenGL SuperBible: comprehensive tutorial and reference*. Pearson Education, 2010. 2