University of Nevada
Reno

An Extensible Component-based Approach to Simulation Systems on Heterogeneous Clusters

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in Computer Science and Engineering

by

Roger Viet Hoang

Dr. Frederick C. Harris, Jr., Dissertation Advisor

May, 2014

THE GRADUATE SCHOOL

We recommend that the dissertation
prepared under our supervision by

ROGER V HOANG

entitled

An Extensible Component-Based Approach To Simulation Systems On Heterogeneous Clusters

be accepted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

Dr. Frederick C. Harris, Jr., Advisor

Dr. Sergiu M. Dascalu, Committee Member

Dr. Monica Nicolescu, Committee Member

Dr. Bobby Bryant, Committee Member

Dr. James Kenyon, Graduate School Representative

Marsha H. Read, Ph. D., Dean, Graduate School

May, 2014

Abstract

There is an abundance of computing power sitting in computer labs waiting to be harnessed. Previous research in this area has shown promising results networking clusters of workstations together in order to solve bigger problems faster at a fraction of the cost for supercomputer time. There are, of course, challenges to using these sorts of clusters: the communication fabrics linking these machines are not necessarily high-performance, and the differences between individual machines in the cluster require careful load balancing in order to efficiently use them. These problems have only become greater with the introduction of acceleration hardware such as GPUs and FPGAs; however, that hardware also provides even greater computing power at an even lower price point for those that can work around their idiosyncrasies. This dissertation presents an approach to designing software to effectively utilize these heterogeneous computing clusters in a modular, extensible manner. I apply it to the development of a large-scale NeoCortical Simulator(NCS) as well as the engineering of a virtual reality library, caVR.

Dedication

For Allison. Yeah buddy.

Acknowledgments

It's been eight years since I applied to graduate school on a whim. I've come across quite a few characters since then, but needless to say, I wouldn't be where I am today without them. First, thanks to Dr. Sergiu Dascalu, Dr. Bobby Bryant, Dr. Monica Nicolescu, and Dr. James Kenyon for reading my ramblings and serving on my committee. Thanks to my advisor Dr. Fred Harris for hiring me way back when and putting up with my maddening overengineering of things.

To everyone I've seen come and go in the many labs I've been in, my extended stay has made that list way too long, but thanks for listening to all my ludicrous ideas about non-scrolling virtual reality Mario games and Karaoke machines. I've made a lot of hopefully life-long friends out of you all.

To my family, thank you for all the support and the delicious dinners I already miss. To Allison, thanks for all the wonderful times that have been and will be.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Despite the ever shrinking size of the transistor, heat and power consumption problems have stymied chip manufacturers' attempts to make a single processing core faster and more powerful. As a result, manufacturers have shifted towards developing processors with multiple slower but more energy-efficient cores. With these architectural changes comes a set of algorithmic challenges in order to fully utilize these multicore chips [48]. A similar trend can be seen in the evolution of the graphics processing unit (GPU) because of their original purpose: massive throughput of highly data-parallel computations [96]. Due to the nature of this task, GPU designers are able to use extra transistors gained from their increased density for more computation, resulting in specialized chips whose annual performance increases outstrip gains made by the more generalized CPU. Furthermore, the introduction of programmability on the GPU opened the doors to its use in other applications; however, similar to multicore CPUs, harnessing a GPU's full potential comes with its own slew of algorithmic challenges [58]. Even greater computational capacity can be gained by networking multiple machines equipped with multicore CPUs and GPUs. This is visible in the latest supercomputers being built today. That these devices are designed and priced at commodity levels allows for a significant amount of computation to be relatively affordable for anyone with a few hundred dollars [72]. Networking cheaply-built computers into Beowulf clusters has been done since the 1990s [14], and the addition of affordable multicore CPUs and GPUs allow for a great deal of computational power to applications that are capable of harnessing it. Again, though, effective utilization

of these resources requires algorithmic designs that overcome the additional layers of inter-node communication and load balancing.

Designing software that runs efficiently on these sorts of systems often comes at the cost of flexibility. For efficiency's sake, algorithms are finely tuned to the accelerators that they run on. For example, the memory access and branching patterns on CUDA devices can greatly affect the resulting performance [92]. Further complicating matters is the fact that while builders of the world's largest supercomputers can afford to homogenize their hardware – the Titan supercomputer has 18,688 of the same CPU and the same number of the same GPU [108] – finding such a homogeneous system in a common research lab is unlikely. Different accelerators have different characteristics and APIs, even between the same class of hardware. An an example, different versions of CUDA hardware have different compute capability levels which can limit, for example, the type of atomic operations that are available [93]. To make software accessible to the largest audience, it must be designed to account for all of these idiosyncrasies.

This dissertation presents an approach to designing software that is both efficient and extensible in light of these challenges. The methodology decomposes a problem into graph nodes that can be executed in parallel with data being passed via a publisher-subscriber mechanism. Within each node, a subgraph is formed using a number of plugin-based extensions in a manner that allows each node of the subgraph to also execute in parallel. The overall graphs are then linked first across multiple different compute devices on the same machine, and then further linked across multiple machines. The end result is a highly parallelized piece of software that efficiently uses whatever resources are available within any given cluster of heterogeneous hardware.

I demonstrate the methodology on two different applications. The first is the latest version of the NeoCortical Simulator (NCS), a large-scale brain simulator. I present improvements over the previous version include the utilization of any arbitrary mix of CPUs and GPUs to accelerate brain computations as well as a set of interfaces that allow for different neuronal, synaptic, and input models to be used together with

minimal added computational cost. The second application is a virtual reality toolkit, caVR. Similar to NCS, the design of caVR allows for arbitrary input and rendering methodologies to be mixed and matched based on availability and the needs of both the developer and user.

There are several contributions from this work. First, an extensible approach to simulation systems on heterogeneous hardware is presented. Second, that approach is demonstrated on the development of a brain simulator. As an added effect, I show how to efficiently map certain brain computations to CUDA devices, in particular, those of the previously CPU-specific NCS models. I also show how my design can allow for models of different levels of biological fidelity and computational load can be mixed with one another. Finally, I demonstrate the methodology on a much different application, a VR library.

The rest of this document follows these contributions. Chapter 2 begins by giving a history of parallel computing, the introduction of accelerators, and developments in software design that take advantage of these developments. Chapter 3 outlines our approach to dealing with heterogeneous hardware clusters that allow for extensibility without sacrificing performance. Chapter 4 illustrates how we apply this approach to the design and implementation of the most recent version of NCS, the Neo-Cortical Simulator, while Chapter 5 applies the same process to caVR, a virtual reality library. Related work and results specific to each of these applications is presented within their respective chapters. Chapter 6 ends this document with some closing thoughts.

Chapter 2

Background and Related Work

Though the two applications we are targeting have rather disparate purposes, the targeted hardware is similar: clusters of computers with potentially heterogeneous hardware. This section outlines the evolution of such systems both from the hardware side and the supporting software side. We take a bottom up approach, beginning with the development of various accelerators that have found themselves as the workhorses in the modern computing cluster and ending with advances on cluster computing in general.

2.1 Accelerators

Like the math coprocessors that preceded them, a number of different pieces of add-on hardware have been designed to offload expensive computations from the CPU. While they have been designed for a multitude of purposes, such as graphics, audio, and physics [49], the GPU has been the most prevalent source of modern computational offloading. We also discuss another offloading solution, reconfigurable computing, that serves as an intermediate between fast hardware-specific solutions and slower software-specific ones.

2.1.1 The Graphics Processing Unit

Rendering computer graphics using entails the transformation of geometry into pixels on the screen. For raster-based graphics, geometry is usually represented as trian-

gles. The vertices of these triangles are transformed from the reference frame they are specified in to a world-space reference frame by multiplying each vertex by a transformation matrix. They are then moved into a view-space reference frame by another matrix. Finally, the three-dimensional view of the geometry is flattened into a two-dimensional image plane by another matrix multiplication. The pixels within the resulting triangles on the image plane are filled in a process called rasterization, with colors based on the values interpolated from per-vertex attributes. These pixels, called fragments, may be blended with existing pixels or discarded entirely based on the desires of the programmer. Fragment colors could be further augmented through the use of textures, usually one-, two-, or three-dimensional arrays of color values.

Originally designed to handle rendering tasks instead of the CPU, the graphics processing unit (GPU) employs a parallel pipeline architecture [6] in order to transform large numbers of vertices and fragments. Figure 2.1 shows such a pipeline. Vertices of polygons are transformed based on the desired perspective by the vertex processor. The results are clipped to the boundaries of the viewport before they are rasterized, converting geometry into actual pixel fragments located appropriately on the display. The final color of the fragments are computed in the fragment processor before they are potentially displayed on the screen. It should be noted that in addition to being pipelined, GPU architectures generally parallelize across each stage of the pipeline. That is, there may be multiple vertex processors working in parallel on different vertices in a Single-Instruction Multiple-Data (SIMD) scheme.

Programmer control over the graphics card was handled through a number of APIs, including OpenGL [102], a cross-platform API, and Direct3D [17], an API specific to Windows and other Microsoft platforms. Both allowed for certain parts of the pipeline to be altered, but beyond that, computation through the pipeline was fixed. For example, lighting could be specified as per vertex or per fragment, and blending of overlapping fragments could be specified by the programmer; however, vertex positions would always uniformly be modified by a set of user-specified transformation matrices to move vertices from object space to the image space.

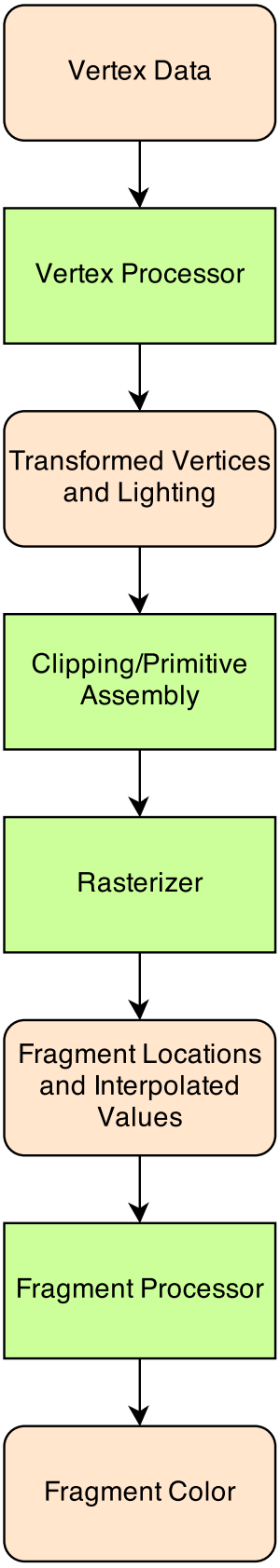

Figure 2.1: The Fixed-Functionality Graphics Pipeline

This deficiency would be somewhat addressed with the introduction of programmable shaders. Originally written in assembly [123], shaders wholly replace parts of the fixed-functionality pipeline, in particular the vertex processor and the fragment processor. For example, one could offset some vertices based on the current time and a sine wave in addition to or instead of the matrices. Usability of shaders would improve over time as each respective API introduced higher level languages to compose them, with OpenGL adding the OpenGL Shading Language (GLSL) [56], Direct3D adding the High-Level Shader Language (HLSL) [32], and Nvidia introducing Cg [83], which could generate GLSL or HLSL based on the platform.

Other developments in GPUs would only further increase their flexibility with the addition of features such as geometry shaders [90] that allow customized manipulation of whole geometric primitives rather than single vertices, but the addition of feedback mechanisms such as framebuffer objects [55] and transform feedback [91], which allowed for pixels to be rendered into readable textures and transformed vertices to be stored into readable buffers, respectively, would pave the way for a whole set of different problems to be offloaded onto the GPU.

2.1.2 General Purpose Computation on the GPU

With the previously discussed hardware-accelerated ways of retrieving the results of the now programmable shaders, researchers began experimenting with the graphics card as a stream processor and general coprocessor [125]. This practice eventually became known as general purpose computation on GPUs (GPGPU) [58]. From the more graphical side, simulations of significantly larger numbers of particles could be done on the GPU – where they would later have to be sent for rendering anyway – by disguising individual pieces of particle data as colors, storing them in texture memory, and updating them by reading in that texture memory in a fragment shader and rendering the updated values into a different texture [79].

Earlier research in this domain typically accelerated solutions to problems that could be easily mapped to graphics concepts. For example, Liu et al. [82] compute

a fluid simulation on a discretized 3D grid that can be mapped to 2D textures. Crane et al. [33] simulate fluids in a similar fashion, albeit by using then available 3D texture rendering capabilities to more closely match the problem domain. In the latter case, the fluid simulation was directly rendered as it was updated; as such, the GPU solution provides two advantages: not only does the simulation get accelerated, but the rendering throughput is also increased by removing the need to transfer data from the CPU to the GPU. A similar boon could be found in the development of VFire [65], an interactive virtual reality application where wildfire is simulated [67] and visualized. The spatial domain of the wildfire simulation is easily mapped to textures which can be quickly visualized as the simulation runs.

While the results of GPU computing were relatively impressive, harnessing it was cumbersome. Developers needed to not only adapt their algorithms and code to graphics constructs but also have knowledge of how to use graphics APIs to actually utilize these constructs. As a result, other APIs, languages, and extensions were created that tried to abstract away these details. One such work, Brook [26], extended C to allow for constructs such as data streams and the kernels that operated on them. Uses of Brook include N-body simulations [43] as well as the computational side of a ray tracer [69].

GPGPU did not go unnoticed by the hardware manufacturers themselves. Nvidia would eventually release its first version of its Compute Unified Device Architecture (CUDA) in 2007 alongside the G80 series of GPUs. CUDA presents the user with a programming model that can better express the data parallelism inherent to GPGPU. In such a model, a kernel function can be executed by a large number of threads (on the order of thousands) concurrently. Threads differentiate themselves and the data they operate on through a system of assigned IDs. Additional advantages over then-traditional GPGPU was the ability to access memory in more familiar array primitives rather than textures as well as the ability for threads within a block to communicate with one another through shared memory [75]. The G80 series of GPUs also marked a change in GPU architecture. While the same type of feed-forward

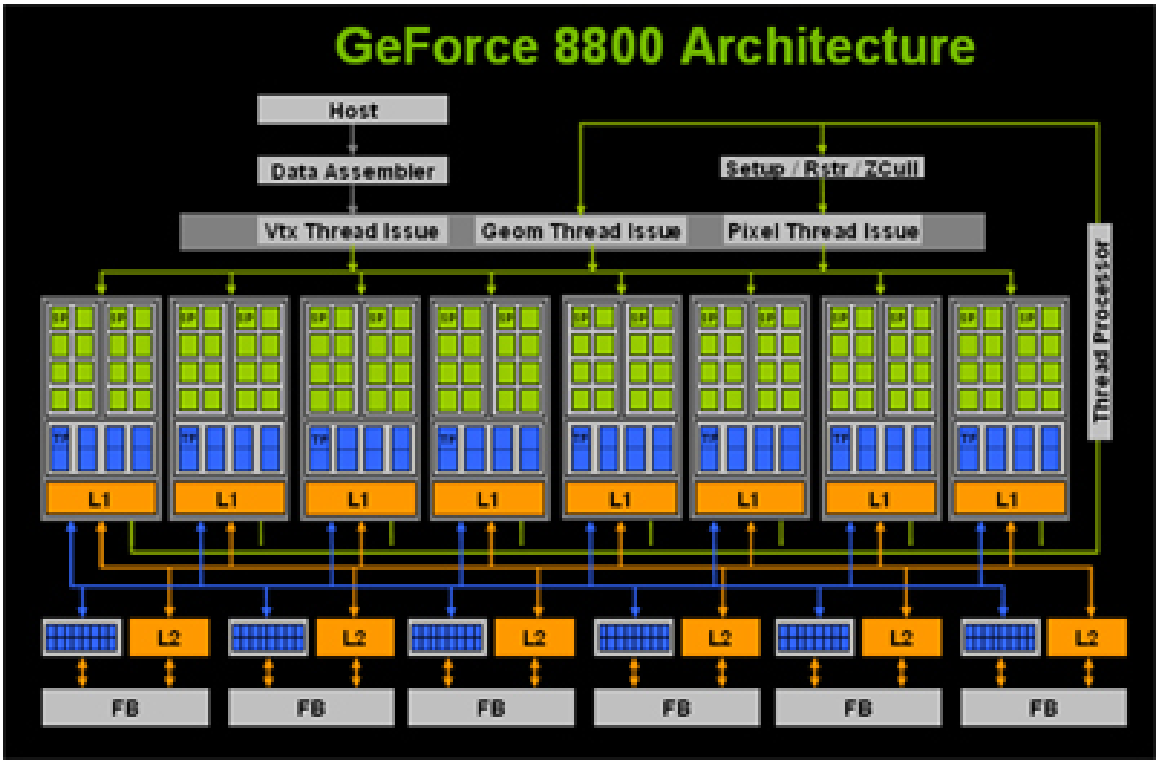

Figure 2.2: Architecture of the Nvidia GeForce 8800 GPU from Nvidia's technical brief [50].

pipeline is employed, the actual processing architecture was unified: instead of specific circuitry to handle vertices and other circuitry to handle fragments, a set of generic processors are able to handle all types of shaders. Figure 2.2 shows this architecture, itself composed of 128 processing cores divided into 16 streaming multiprocessors. Later improvements to GPU architecture would generally increase the number of processors, with the latest GTX780 cards containing 2304 processing cores [89].

While programs written in CUDA look more akin to standard C and C++ programs, programmers must still take care with their programs' behavior in order to maximize performance. For example, memory should be accessed in aligned contiguous sections within blocks in order to coalesce them into single memory accesses, and branching within a warp (a cluster of threads executing in lockstep) would cause a portion of the warp to stall while the branch was executed [93].

While CUDA primarily presents a high-performance GPU programming model, it was designed for Nvidia hardware. Akin to OpenGL, OpenCL was designed as an open standard alternative to CUDA for parallel programming on not only GPUs but also CPUs and other architectures. The standard does not promise any sort of optimality in terms of performance; rather, it guarantees correctness across all supported device types [115].

2.1.3 Field-Programmable Gate Arrays and Reconfigurable Computing

Solutions for various applications could be placed on a spectrum. On one end, dedicated hardware could be designed to very efficiently perform a specific task. On the other end, a very general processor could be used, and software would dictate which parts of the generalized hardware would be used in which order to accomplish the same task. Early GPUs could be viewed as belonging to the the hardware end while CPUs could be placed in the software end. GPUs and other hardware solutions are unmatched to CPUs in terms of performance due to their very specialized nature; however, they are not usually applicable to other tasks. CPUs tend to be slower but more versatile due to their general purpose design.

An intermediate solution to problems exist in the form of reconfigurable computing, where hardware can be altered after fabrication to tailor it to the task at hand. One such piece of hardware is the field-programmable gate array(FPGA). Here, parts of the hardware are controlled by configurable hardware bits; additionally, the routing circuitry itself is programmable, allowing a customized circuit to be constructed. Similar to other hardware acceleration solutions, these devices tend to be coupled with a CPU to handle other tasks such as control of the device itself [31]. The use of an FPGA can be viewed has moving computation from the temporal domain of having a linear set of instructions that must be performed to the spatial domain where computation is performed by some cluster of circuitry before proceeding to the next cluster of circuitry. The advantage of this transition is the pipelining and

thus parallellization of computation throughout the hardware, increasing throughput. Results show improvements ranging from 10 times to 100 times compared to CPU solutions [39]. A potential drawback to using these types of devices is the large variety of device types and lack of standardized design methods. Todman et al. [122] give a survey of many of these architectures and methods while Hartenstein [59] summarizes a number of more coarse-grain reconfigurable computing projects.

2.2 Parallel and Cluster Computing

There are limits to the amount of computation a single CPU core can do, limiting the types of problems that can be solved in any reasonable time frame. Fabrication advancements raised these limits, and for a while, with each new processor generation, programs written for single-core CPUs grew faster and faster without any modifications. In 2005, Sutter [117] would declare that "[t]he free lunch is over." Chip manufacturers were shifting to multicore designs, with two or more cores on the same die. Clock speeds would not increase as drastically as before, so those unaltered single-threaded programs would gain nothing from the addition of a whole extra computing core. Programs that wanted to take advantage of these developments would have to be structured with concurrency and parallellism in mind.

The transition to multicore was not the harbinger of the concurrent computing age; it merely brought it into the limelight. Multiple processors, albeit not on the same die, could be placed on a single motherboard and communicate via shared memory long before this. Early research with these systems focused on algorithms [73], memory consistency [3], and synchronization [84]. Later work would introduce tools to aid developers in productively utilizing such systems. OpenMP [36] is an API designed to simplify parallel programming on shared memory systems. Code could be annotated with directives that would allow OpenMP to parallellize constructs such as loops across a pool of available processors.

Though building shared memory systems is and was certainly possible, it required more esoteric hardware to facilitate multiple processors sharing that same memory.

An alternative is cluster computing [8], where multiple independent computers are networked together and communicate through messages over that fabric. Such systems became an attractive option for several reasons, including the performance to price ratio for standalone computers in addition to improvements in networking technology. Modern supercomputers tend to follow the same format, albeit with more sophisticated communication fabrics; unfortunately, the design of these systems requires significantly greater engineering effort compared to using an already connected network of workstations [5].

As OpenMP facilitated parallelization across shared memory systems, the Message-Passing Interface (MPI) [128] was designed to facilitate communication between multiple computers. MPI allows nodes in a cluster to communicate with one another by sending each other point-to-point messages. Additionally, a set of collective operations, such as broadcasting, where a single computer replicates a piece of data to all other computers, and reducing, where the results from multiple computers are combined, are also defined. MPI-2 extends this feature set with one-sided communication methods, such as reading and writing to another machine's memory without that machine's involvement, and dynamic process generation, which allows more processes to be generated during run-time rather that configuration time [51].

Concurrent and parallel programming across a cluster of computers via message-passing is arguably more complicated than implementing the same task across a shared memory system. A middle-ground exists for programs designed for the latter: distributed shared memory [101]. Such systems tend to physically be structured as a cluster; however, from the programmer's point of view, all resources across the entire cluster appear as a single powerful system image. Memory consistency across the cluster is a significant problem in these sorts of setups. What to do when two different nodes read and write to the same piece of memory in the same period of time depends on the implementation; the choice of consistency model affects the amount of data that must be passed around as well as assumptions that the programmer can make [88].

2.3 Heterogeneous Cluster Computing

The term "heterogeneous" in this domain is somewhat overloaded but generally refers to computing clusters constructed out of a diverse set of hardware. In some cases, that heterogeneity manifests itself as different processor speeds across different nodes in the cluster. In other cases, the diversity stems from the introduction of accelerators. In the most extreme cases, both the preceding conditions exist: clusters are composed of processors with varying performance characteristics along with variety of different accelerator types. Research has been done across all of these types.

Beaumont et al. [12] provide work that is an example of the first case. The authors modify the data distribution components of the ScaLAPACK library in order to load balance linear algebra computations across a heterogeneous cluster of CPUs. They show that load-balancing matrix operations can be a difficult problem: determining the optimal grid configuration for a group of heterogeneous processors is NP-complete. They do, however, provide a heuristic solution. Barbosa et al. [10] also solve linear algebra problems with a heterogeneous set of CPUs connected together with the Windows Parallel Virtual Machine.

In both of the preceding examples, matrix operations such as LU decomposition required solutions that took into account that the problem size would reduce as the algorithm progressed; in this case, the way in which the problem shrinks can be determined a priori and handled accordingly. For other problems, such a luxury cannot be afforded. Teresco et al. [119] develop a system that is generally used for adaptively refined meshes where the initial distribution of data may become unpredictably unbalanced during each iteration depending on where more refinement is needed. In such a system, a load-balancing suite redistributes data after each iteration based on properties collected about each machine in the cluster.

For the second definition of heterogeneous computing, an example can be found in the Keeneland project [126]. Architecturally, the cluster resembles a supercomputer with its high-performance communication fabric. The GPUs used across the entire

cluster are also all of the same model. To help with programmer productivity, a number of tools are provided, including Ocelot [40], a framework that allows CUDA programs to be executed on non-CUDA compatible devices as well as CPUs. Efforts similar to Ocelot can be found in research presented by Lee et al. [80], where the parallellization through OpenMP is modified to run on CUDA devices instead.

At the more extreme end of heterogeneous computing, projects such as Axel [124] can be found. Axel itself is a cluster composed of an array of identical nodes; each node, however, is composed of a CPU, a GPU, and an FPGA. To utilize all the processing elements in the cluster, a map-reduce framework was employed. A similar system of identical nodes can be found in the QP cluster [109], where each node is composed of two dual-core CPUs, four GPUs, and an FPGA and are connected using Infiniband.

In almost all the discussed systems, tools were developed to accelerate development of applications on their respective clusters. In the same vein as Ocelot but more akin to distributed shared memory systems, Barak et al. [9] implemented an OpenCL abstraction layer that allows an OpenCL program to use all available devices in a cluster without knowing it.

Chapter 3

An Extensible Component-based Approach to Simulation Systems on Heterogeneous Clusters

The commonplace computer lab can potentially be rife with computational power; they can also be rife with diversity as only subsets of these computers get upgraded at any given point in time. Designing software that performs well on one such heterogeneous cluster can be a difficult task, with specially designed software and frameworks developed just for that architecture. Making that software efficient on other systems only makes it more challenging.

I propose an approach to engineering simulation systems that is easily extensible to different types of heterogeneous clusters. These systems are also extensible in terms of adding new functionality without the need to recompile the simulation core or expose proprietary code.

The approach can be summarized as follows:

- Decompose the simulation into a graph of computational segments, their inputs, and their outputs.
- Decompose each computation into a subgraph to support extensibility.
- Replicate this whole resulting graph across all devices within the cluster.
- Connect these graphs with minimal communication.

- Distribute data based on available resources.

I now go into each of these steps into detail. To illustrate each step, I apply the approach to a gravitational n-body simulation.

3.1 Flow Decomposition

The approach begins by decomposing the simulation flow into a directed graph, where nodes represent segments of computation and connections present the flow of data between computational segments. We allow computation in each node to happen concurrently with data passed along connections using a publisher-subscriber mechanism developed by the author, where a node will subscribe to its necessary inputs and blindly publish its outputs to any subscribing buffers. We constrain the publishing mechanism by limiting the number of published buffers that any given node can have in circulation at any point in time, somewhat similar to many double- and triple-buffering schemes. This constraint is imposed for two reasons. First, it prevents cannibalization of computing resources on nodes that have no dependencies and can truly run freely, and second, it provides a natural "resource pool" mechanism, where memory is not constantly freed and reallocated but simply reused when marked available. To facilitate this, published buffers must be released by each of their subscribers; upon release from all subscribers, the buffer is automatically added back to the publisher's pool of available blank buffers. The basic outline for the classes that comprises this publisher-subscriber mechanism is shown in Figure 3.1.

As an example, Figure 3.2 shows a graph decomposition of an n-body simulation. There are two primary computations in such a system: the updating of velocities and the updating of positions. The system in this case is tightly coupled. In order to update a body's velocity, both its previous velocity as well as the position of every body in the system are needed. To update a body's position, the body's current position is required. To illustrate the need for constraining the publisher system, an additional computation node is added where external forces are applied to the system.

```
template<typename PublicationType>
class SpecificPublisher : public Publisher {
public:
  unsigned int publish(PublicationType* pub);
  Subscription<PublicationType>* subscribe();
  void addBlank(Publication* blank);
  Publication* getBlank();
  bool init(unsigned int num_blanks = 3); // triple-buffer
};
private:
  std::vector<Subscription<PublicationType>*> subscriptions_;
};

template<typename PublicationType>
class Subscription {
public:
  PublicationType* pull();
  void pull(PublicationType** location, Mailbox* mailbox);
};

class Publication {
public:
  void release(); // signals that we are done with this pub, free it
private:
  int ref_count_; // how many subscribers still need to release this pub
};
```

Figure 3.1: Basic class definitions for the publisher-subscriber system in NCS.

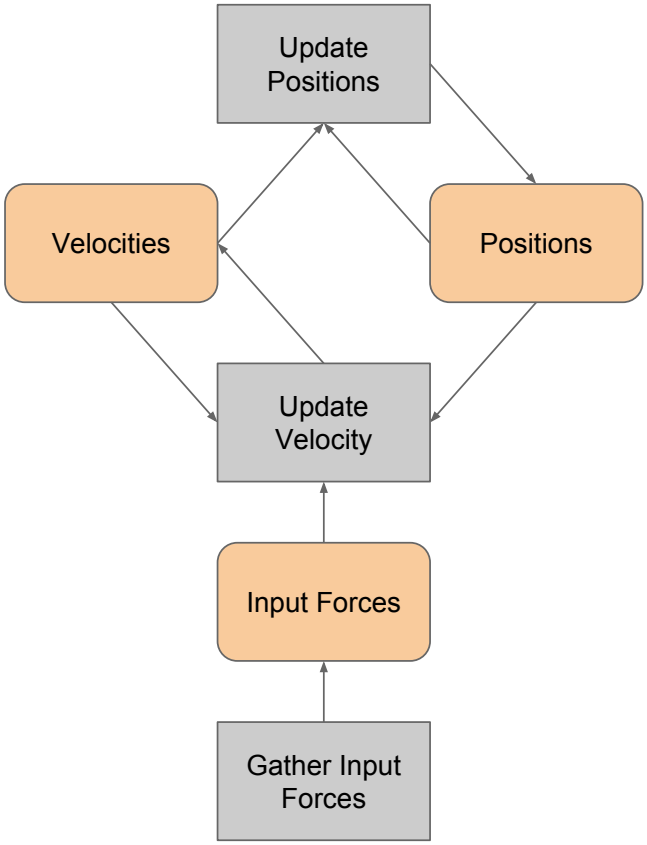

Figure 3.2: Example graph decomposition of an n-body simulation. The gray boxes represent computational segments while the orange boxes represent data being passed.

Without the constraints in place, this node is allowed to run as fast as it can since it does not require any data from another node. In doing so, the node could allocate and publish buffers faster than subscribing nodes could them, resulting in memory overuse. Additionally, because nothing can block the node, it would be allowed to consume all allocated processing time given to it rather than relinquishing that time to subscribing nodes.

3.2 Extensibility Decomposition

To support extensibility, we decompose each desired computational node into a subgraph where each subnode represents a single extension, all of which again run concurrently using the same described publisher-subscriber mechanism. These extensions

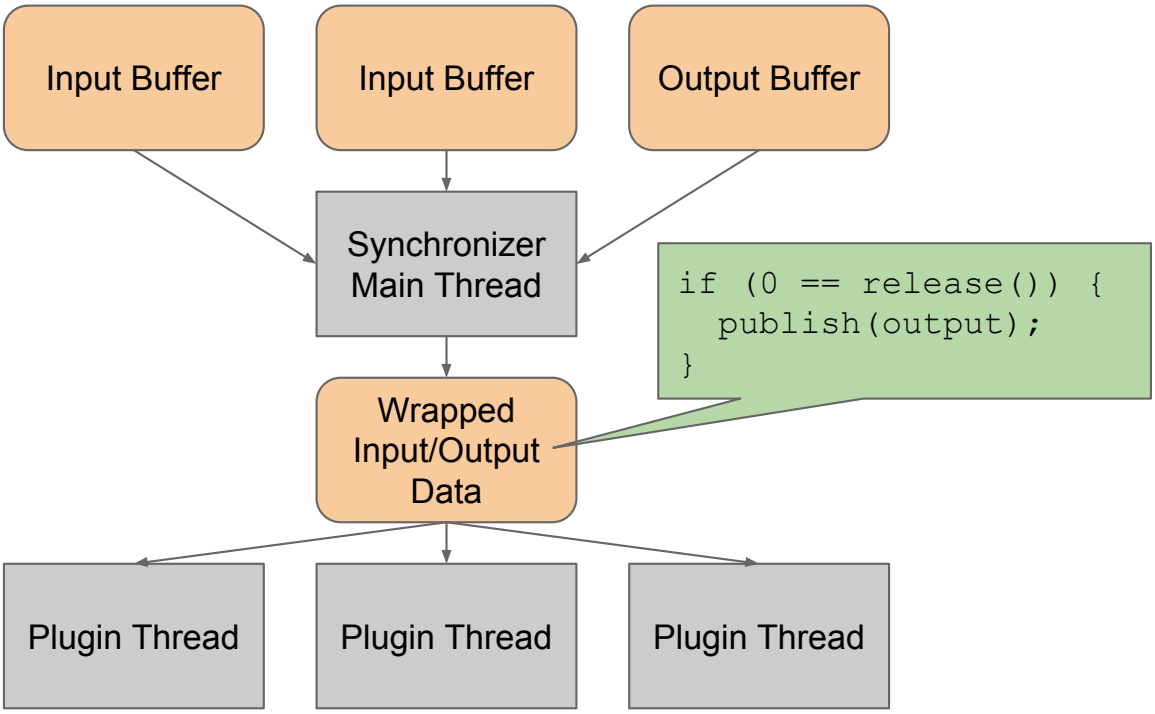

Figure 3.3: Generalized decomposition of a computation node to achieve extensibility. The gray boxes represent computational threads while the orange boxes represent the data passed between them.

take the form of plugins (shared libraries). This decision was made in order to prevent the need to recompile the simulation core while also allowing third parties to develop proprietary modules without the risk of exposing secret information. Figure 3.3 shows a generalized decomposition of a node. The execution of the wrapped publishing command is achieved with the introduction of a prerelease function that is also executed when a buffer is completely released.

Suppose that input forces in the n-body could be gathered from a number of different sources, such as a file, a network socket, or random number generation. In such a case, decomposing the input node into the aforementioned subgraph would allow all sources to be gathered in parallel.

3.3 Graph Replication and Communication

A complete, concurrent, and extensible description of the entire simulation in terms of behavior and data flow is available at this point. We then simply replicate the graph to all available devices in the cluster, allowing each device to perform a full simulation of all elements it would be responsible for if all the available data is there. To facilitate this, we identify any global data that is necessary and connect all of these replicated graphs with a data exchange node that pushes all local data to other devices while simultaneously pulling data from every other device. This node itself can be represented as a subgraph that shrinks or grows based on the number of machines and devices in the system.

For an n-body simulation, we require the position of every body in the system in order to gather the total forces acting on any given body. As such, we connect the replicated graphs with a node that exchanges all body positions with one another across the cluster. Figure 3.4 shows this modification.

3.4 Distribution

Although a distributed simulation system exists at this point, care must be taken in distributing simulation elements to devices given potential differences in performance characteristics between devices. To achieve this, two things are required: a description of each device's capabilities and a description of each simulation element's computational requirements.

In the case of the n-body problem, a reasonable description of performance per device could be estimated or measured computing throughput. The amount of calculation required per body is roughly the same; thus, a reasonable distribution methodology would be to dole out bodies to each device based on their relative performance estimations during initialization before the processes are allowed to run.

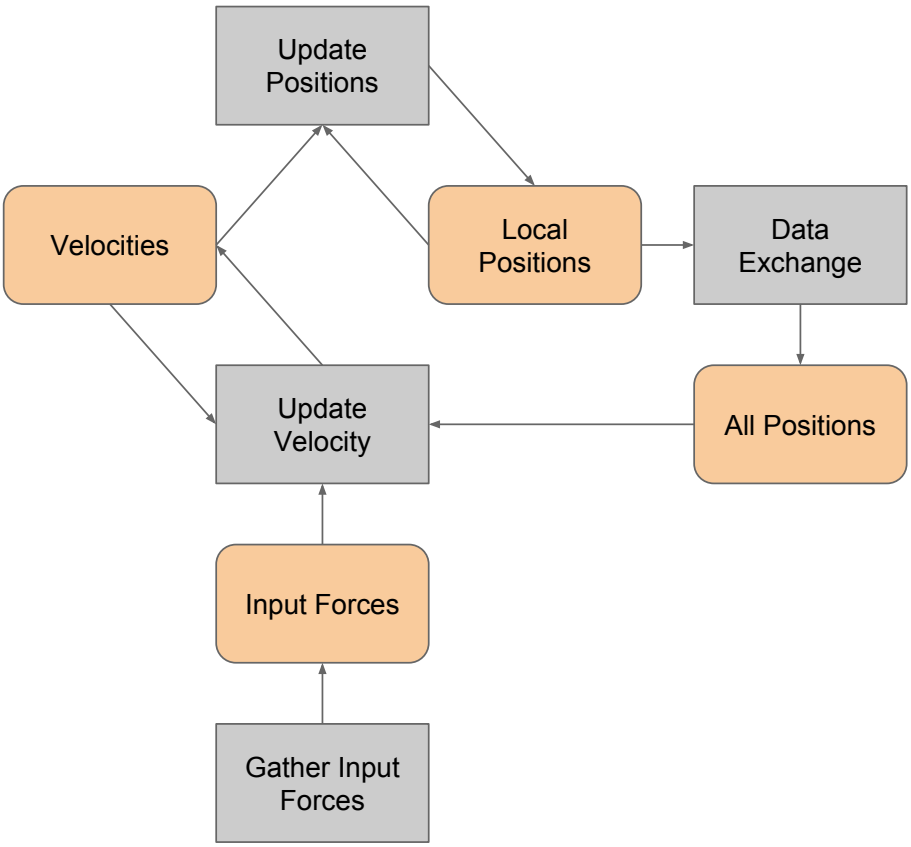

Figure 3.4: Modification of n-body graph decomposition to facilitate communication. Gray boxes represent computational processes while orange boxes represent data.

3.5 The Rest of this Dissertation

The rest of this dissertation is dedicated to demonstrating this approach on two applications. In the first, we apply this methodology on the latest iteration of the NeoCortical Simulator (NCS), a large-scale brain simulator. Though previous versions had already been parallelized across CPU clusters, the alterations in this incarnation not only allow multiple types of neurons, synapses, and inputs to be modeled in tandem with one another, but also allow for these elements to be simulated on both CPUs and GPUs simultaneously. The design of the system also allows for inputs and reports to be specified at execution time, presenting a robust solution for more real-time applications.

The second application, the virtual reality library caVR, solidifies the real-time constraint as a requirement. Here, the heterogeneity stems not only from the variety of computing hardware but also from the input and output methodologies. While rendering may need to support different graphics libraries, such as OpenGL and DirectX, this only encapsulates one sense that can be "rendered." The system must be designed to support various output libraries from the beginning. In addition, virtual reality in particular is a field of considerable hardware innovation and experimentation, so the system must be easily extensible enough to facilitate these needs. Further confounding matters is the need to also support a wide variety of input devices. In spite of the hardware heterogeneity, the library must be robust enough such that an application developer need only specify a small set of functions for a large amount of hardware to be made functional.

Each of the subsequent two chapters are structured as follows for each respective application. The problem will be introduced, and then background and previous work related to that problem will be presented. We then show the design of the system based on this approach and discuss results and conclusions before ending with some avenues for future work.

Chapter 4

NCS

4.1 Introduction

Understanding the mechanisms that drive the human body is an essential prerequisite to effectively addressing the myriad of possible maladies that can plague it. Unfortunately, acquiring this knowledge can often times prove impractical or even unethical with existing technology. This is particularly true in the field of neuroscience. In these cases, modeling stands out as a potential alternative. However, its application to neuroscience has its own set of computational hurdles depending on the size and complexity of the system being modeled.

In this chapter, we present the latest version of the NeoCortical Simulator, (NCS), an application designed to simulate large-scale models of spiking neurons. In order to accommodate larger simulations in a reasonable time frame, we have redesigned NCS to exploit the processing power of GPUs along with CPUs. The remainder of this chapter is structured as follows: Section 4.2 gives an overview of spiking neural simulations and related solutions while Section 4.3 describes the design of the new system. Section 4.4 gives some performance results before Section 4.5 draws some conclusions and offers some room for future work.

4.2 Background

The subject of neuroscience is a field of study on its own and its complete review beyond the scope of this dissertation. Instead, I review enough biology for the reader

to understand the problem domain and the various modeling strategies.

4.2.1 Biological Neurons

The brain is composed of a large number of cells called neurons. In a human brain, the number of neurons is estimated to be about 100 billion. Figure 4.1 illustrates the typical structure of a neuron. Branching away from the cell body are a number of dendrites that can receive signals from other neurons. Also branching away from the cell body is a generally longer axon through which electro-chemical signals flow towards the axon endings. These endings connect to the aforementioned dendrites of other neurons, though not physically; instead, a small gap called a synapse exists. The number of these gaps is roughly 1000 to 10000 per neuron [18].

The dynamics of this mass of neurons generally operates as follows: Ion channels alter the amount of electrical charge both inside as well as outside the cell membrane, the difference of which is called a membrane potential. When the membrane potential exceeds a threshold voltage, a signal called an action potential is transmitted from the body through the axon to the axon endings. The axon endings then release chemicals called neurotransmitters which cross the synaptic gap to another neuron's dendrites, where they affect the ion channels of the subsequent cell, altering the voltage of that body [18].

4.2.2 Modeling Neurons

A number of models have been developed regarding the dynamics of neurons and their associated ion channels. One of the earliest and most used is the one derived from experimental results by Hodgkin and Huxley [68]. In this model, both the behavior of ion channels as well as the membrane potential are governed by a system of differential equations. Depending on the type of ion channel, the equations controlling it can be dependent on gating variables.

The previously described equations can be modeled as an electrical circuit. Figure 4.2 shows one such circuit. The neuron's membrane potential is the voltage from

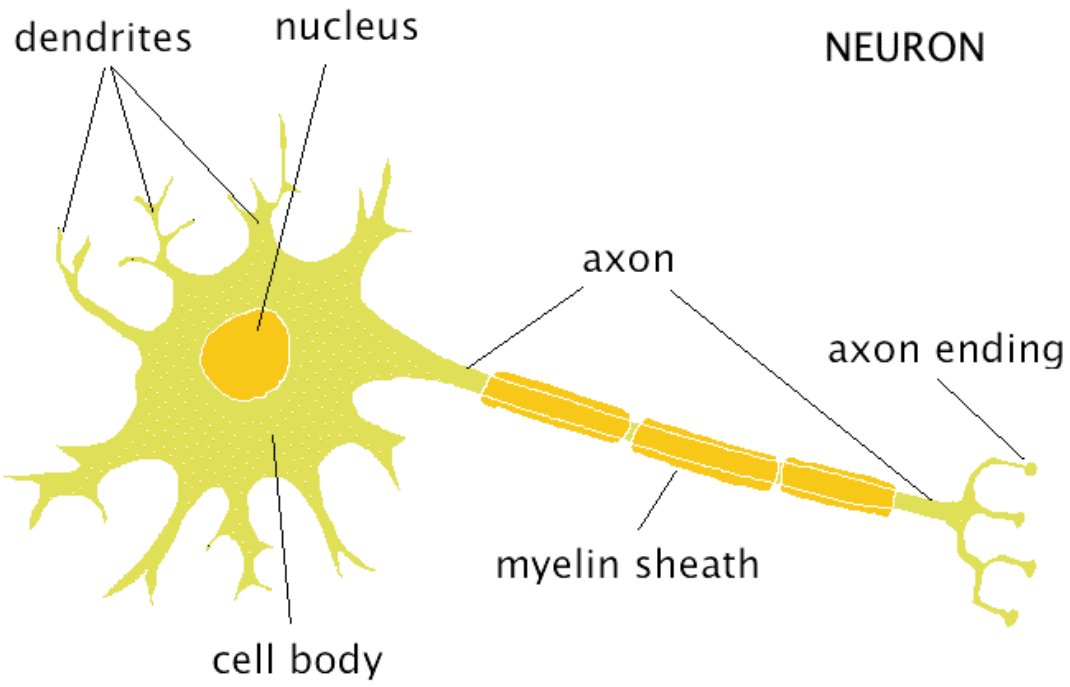

Figure 4.1: Structural illustration of a neuron by Boeree [18].

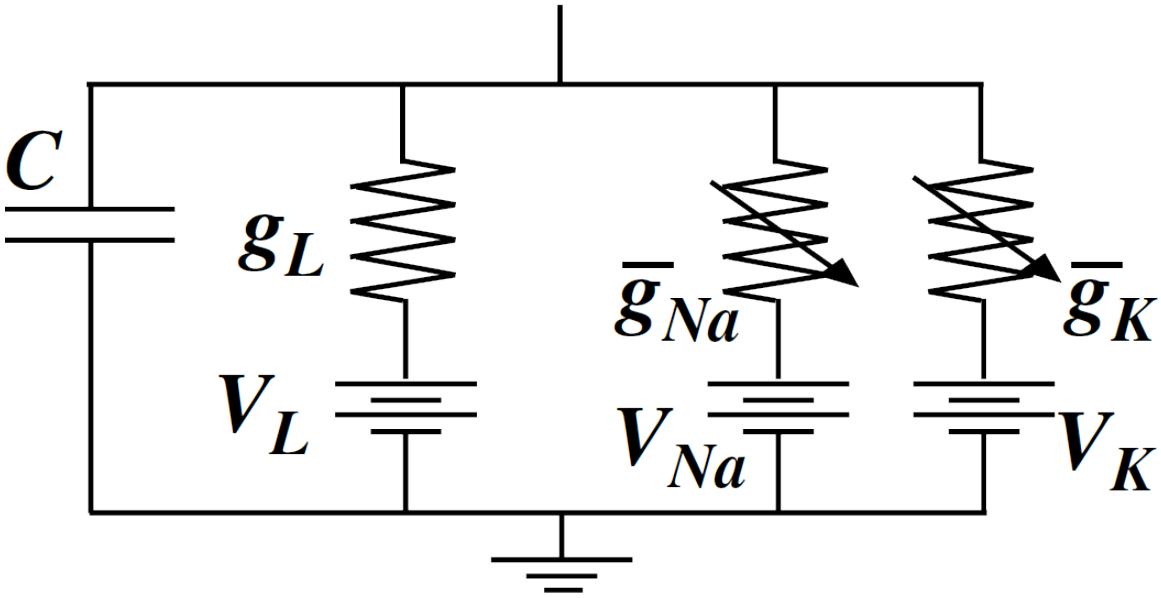

Figure 4.2: Equivalent circuit for a neuron by Gutkin, Pinto, and Ermentrout [57]. C is the capitance, while g_L, $\bar{g}_{Na}$, and $\bar{g}_K$ are conductances due to leakage channels, sodium channels, and potassium channels, respectively. V_L, V_{Na}, and V_K are the reversal potentials of their respective channels.

the top of the circuit to the ground. Stored charge is represented by a capacitor while various ion channels are represented as a voltage source that corresponds to a reversal potential and a potentially variable resistor that affects the amount of generated current [57].

While the Hodgkin-Huxley model provides an elegant model of neuron behavior, its complexity does not lend itself to timely simulations of large numbers of neurons. For these larger networks, researchers may be more interested in the neurons' interactions rather than the precise dynamics of a singular neuron. In such cases, functional approximations may be appropriate. Two popular options are the leaky integrate-and-fire (LIF) neuron and the Izhikevich neuron. For the former type, the neuron essentially accrues ("integrates") charge until it reaches a threshold voltage, upon which it fires, resetting the amount of accumulated charge. Charge integrates as a result of both synaptic currents as well as external input current. Charge is also dissipated as a result of a leakage current [28].

The latter approximation, the Izhikevich neuron [71], goes even further in eschewing biological accuracy in favor of performance. The neuron has only two variables that evolve over time: a membrane potential v and a recovery variable u. Four constants determine the behavior of the neuron in a set of equations that can be very quickly computed. Based on these constants, a neuron can be modeled to exhibit a variety of different firing characteristics. The flexibility of these neurons along with their computational simplicity makes them an attractive option for large scale simulations of networks of neurons.

The discussed neuron models are only a subset of models that have been proposed; a number of other hybrid solutions exist, such as the one used by previous versions of NCS, where the subthreshold dynamics are governed by Hodgkin-Huxley style equations while the firing mechanism simply follows a templated waveform in order to save computation [41].

It should also be noted that the models mentioned so far only discuss the dynamics of the membrane potential; the dynamics of synaptic transmission have their own set of models. Some models assume that when an action potential arrives at a synapse, a constant amount of synaptic current is injected into the postsynaptic neuron; however, other models may incorporate learning through mechanisms such as spike-timing dependent plasticity (STDP), where the amount of synaptic current or weight that is injected is modified over time based on the spike timings of both the presynaptic and postsynaptic neurons [111].

4.2.3 Simulation Strategies and Tools

A more in-depth survey of simulation strategies and tools can be found in Brette et al. [24]. The spectrum of approaches is broad, each with its own advantages and disadvantages. For example, the use of a discrete time grid simplifies the nature of the computation: for every time step, update every neuron and check for firings. This sort of algorithm lends itself well to vectorization and parallelization. That simplicity also has some notable drawbacks: firings are locked to the resolution of the time

grid, and computation must be done on every neuron at every timestep regardless of whether they are firing or not. The situation is reversed when using an asynchronous or event-based simulation strategy. Firings occur precisely where they are calculated to occur at the cost of a more complicated algorithm that must constantly maintain a time-sorted list of events in order to figure out which neuron to update next. Further complicating matters is the possibility that new events can preempt or even cancel other future events.

Several pieces of software have been developed over the years for the purpose of simulating networks of neurons, including NEURON [60], GENESIS [20], and NEST [52]. An in-depth comparison of these tools, along with NCS, is given by Brette et al. [24]. Though efforts have been mounted on parallellizing each of these simulators for the sake of improving performance [85, 53, 99], these efforts have focused primarily on using multiple CPUs, whether in the same machine or networked together, to reduce processing times and increase problem sizes.

With the advent of CUDA and GPGPU, a slew of research projects cropped up that mapped neural simulations to the GPU. A number of these projects focus on the use of the Izhikevich neuron [86, 45, 15, 46]. In many of these cases, the simulation was also performed on a single GPU.

4.2.4 NCS and Related Work

With the success found in simulating networks of neurons on single GPUs, it was in our interest to update NCS in order to use GPUs in order to speed up simulations. This work was built upon a bit of previously published research, including our attempts to simulate Izhikevich neurons on multiple homogeneous GPUs in the same machine [120], where we passed a bit vector representing the firing state of every neuron in the system as the primary form of communication between GPUs. One of the authors of that previous work, Corey Thibeault, would later expand upon this with a hybrid communication scheme based on firing rates [121].

The bulk of work done on the previous version of the NeoCortical Simulator,

NCS5, was done by James Frye [47]. The work focused on the optimization of its predecessor, NCS3, in order to obtain an order of magnitude performance increase in terms of speed on a cluster of CPUs.

The confluence of these two strands of research is the latest version, NCS6, a simulator designed to not only exploit the parallellism of both CPUs and arbitrary GPUs but also allow for the addition and combination of different neuronal models. These features are all new and unavailable in NCS5. Additionally, NCS6 uses the parallel graph structure discussed in Chapter 3, employs a different communication scheme, and allows for reporting to be dynamically specified at run-time. The rest of this chapter has been published in *Frontiers in Neuroinformatics* [66]. Despite publication, improvements have been added and the text modified to reflect these alterations.

4.3 Design

4.3.1 Goals

NCS6 was designed with three qualities in mind: extensibility, efficiency, and approachability. The first, extensibility, is a requirement in order to enable mixed modeling, where parts of the simulation could be computed one way while other parts are computed another. Enabling this capability would allow more expensive precise computations to be done in regions of interest while other regions could be approximated with simpler computations. Additionally, it would allow different submodels created using different components to be combined without the need to convert the computational models used in one submodel to the computational models used in another.

The second quality, efficiency, simply stems from a desire to maximize resource utilization and minimize communication in order to maximize throughput. By maximizing resource utilization, we seek to use all available computational devices, not only the CPUs alone or the GPUs alone but rather all device types working in tan-

dem. We minimize communication in hopes of ameliorating the necessity of a high-performance network connecting compute nodes. Though efficiency can be at odds with extensibility, we arrange our data structures in such a manner as to minimize the loss of the former while gaining the boons of the latter.

Approachability deals with providing an effective user experience. That is, running the simulator should be a simple task of executing the simulator program, perhaps with a few input arguments. Furthermore, the modeling aspect should be streamlined in a way such that repetitive tasks are reduced without sacrificing the desired expressions of the user.

4.3.2 Simulation Composition

At its core, every simulation is composed of four elementary parts: Neurons, Synapses, Stimuli, and Reports. Each of these components is also regarded as a subtype of a more generic Element type. Neurons and Synapses in this scope are not exact analogs to their biological counterparts. Neurons are the cell bodies that receive stimulus currents and clamp voltages and, under some defined circumstance, fire, transmitting spike signals to their synapses. Synapses represent unidirectional connections from one neuron to another. When the presynaptic Neuron fires, the Synapse injects current into the postsynaptic Neuron after some specified delay. Stimuli represent external inputs that affect Neurons. These can have the effect of either augmenting the amount of current received by a neuron or clamping the voltage of the Neuron body to some precise value. The final element, Reports, specify the output component of the system. Reports take either a collection of neurons or a collection of synapses and output some desired value to some type of data sink.

The descriptions of the aforementioned elements were intentionally left vague. In reality, the behavior of each element is governed by a selected computational model, though some constraints are enforced. For example, one neuron could be simulated following an Izhikevich model while another could be simulated using an integrate-and-fire model. While the internal behavior of the two cell types can diverge, both

are still required to provide two bits of information at each time step: whether the cell fired and the cell voltage. They are both also provided with the same set of input data: Stimuli, total synaptic current, and the previous neuron voltage value.

4.3.3 Simulation Environment and Distribution

The initial targeted computing environment for NCS6 was any, potentially heterogeneous, cluster of one or more computers composed of some mix of CPUs and CUDA-capable GPUs. Due to the way NCS is designed, expansion to OpenCL devices would only require the implementation of certain stub functions. Since all computing devices can potentially have different performance characteristics, we first assign a relative computational power rating to each device. The current method for estimating these values is to multiple each device's clock speed by its number of compute cores.

Given the relative power of each device, we distribute simulation elements across them. The rationale behind our distribution method can be traced back to the expected behavior of Synapses. When a presynaptic Neuron fires, a Synapse will, after some delay, inject current into the postsynaptic Neuron purely based on the state of the Synapse itself and the voltage of the postsynaptic Neuron. As such, every Synapse is distributed with its corresponding postsynaptic neuron in order to minimize the amount of data that must be passed between devices. With such a scheme, the only data that must be passed across the cluster during simulation is the firing state of every Neuron. This state is a boolean true/false value; thus, a single Neuron's firing state can be represented by a single bit. Stimuli are similarly distributed. Since each Stimulus can only affect a single Neuron, stimuli are distributed on the same devices as their associated Neurons.

As for the distribution method itself, we first estimate the computational cost of a Neuron and all of the Synapses and Stimuli that affect it. Since the number of Synapses generally greatly outnumbers the number of Neurons by several orders of magnitude, we use the number of Synapses that affect a given Neuron as the Neuron's computational cost. Neurons are sorted in order of decreasing cost and

then distributed across all devices in the cluster such that the device with the lowest load (total cost / device power) receives the next Neuron. Once all Neurons are distributed, their associated Synapses and Stimuli are placed on the same devices. All compute Elements on each device are then reordered so that elements of the same subtype (Izhikevich, LIF, etc.) form contiguous blocks in memory that can later be consumed by plugins.

4.3.4 Data Scopes and Structures

Due to the distributed nature of NCS6, Elements may be referenced in a number of scopes that mirror the environment's hierarchy: plugin, device, machine, and global (cluster). After the distribution is finished, every Element is assigned a monotonically increasing ID for each scope. IDs are padded between plugins so that data words for structures allocated in other scopes are related to only one plugin. In general, this means that IDs are padded to a factor of 32 (the number of bits in a word) between plugins. It is important to note that IDs are only unique within the same Element type; that is, there can be both a Neuron and a Synapse with a global ID of 0. Figure 4.3 shows an example distribution.

Depending on which elements need access to other elements, certain key data structures are allocated and accessed using different scopes. Data that is specific to an Element subtype is stored at the plugin scope. Because Synapses may need to access the membrane voltage from their postsynaptic Neurons in order to determine their synaptic current contributions, membrane voltages are stored and accessed using device level IDs. The reason is all postsynaptic Neurons and the Synapses that affect them reside on the same device due to the way they are distributed. However, because the spiking state of a synapse depends on the spiking state of the presynaptic Neuron, the spiking state of Neurons is accessed using a global level ID when updating synaptic spiking states.

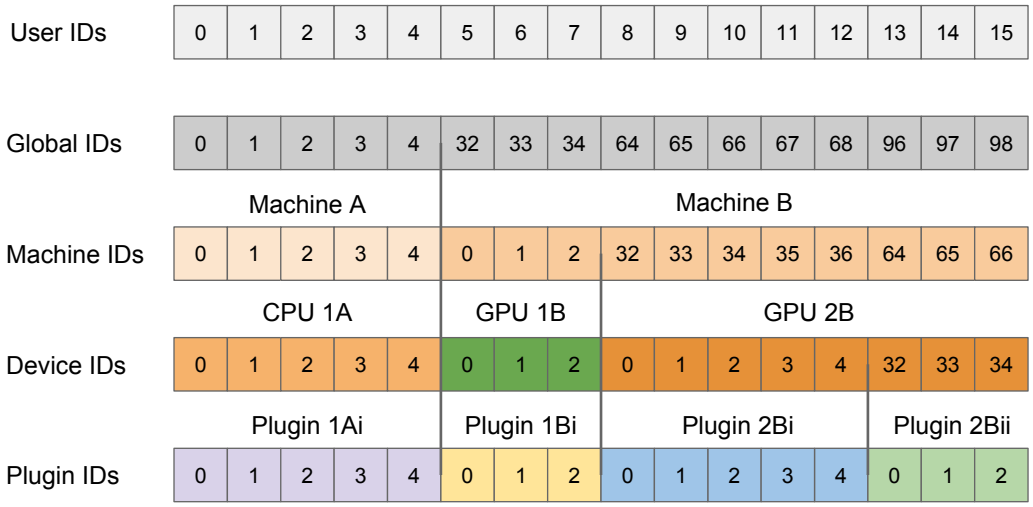

Figure 4.3: An example of how IDs would be distributed across a cluster for a single element type. Vertically aligned boxes denote the IDs at different scopes for the same element. To allow processes to work on wholly separated sections of memory even in the case of bit-vectors, padding is used at every level.

4.3.5 Simulation Flow

The basic flow of a simulation is as follows: for each time-step, the current from Stimuli and Synapses is computed and used to update the state of every neuron. The resulting spiking state of each Neuron is then used to determine the spiking state of their associated Synapses in later time-steps.

To facilitate maximum utilization of computing devices, the simulation is partitioned into several stages that can be executed in parallel as long as the requisite data for a given stage is ready. Figure 4.4 illustrates this division of work along with the required data needed to simulate a particular stage and the data that is produced once that stage has been updated. A publisher-subscriber system is used to pass data buffers from one stage to the next. During the simulation, a stage attempts to pull all necessary data buffers from their associated publishing stages. The stage is blocked until all the data is ready. Once it obtains all the required data buffers, it advances the simulation by a single time-step and publishes its own data buffer while releasing all the others that it no longer needs. The stage then attempts to grab the data

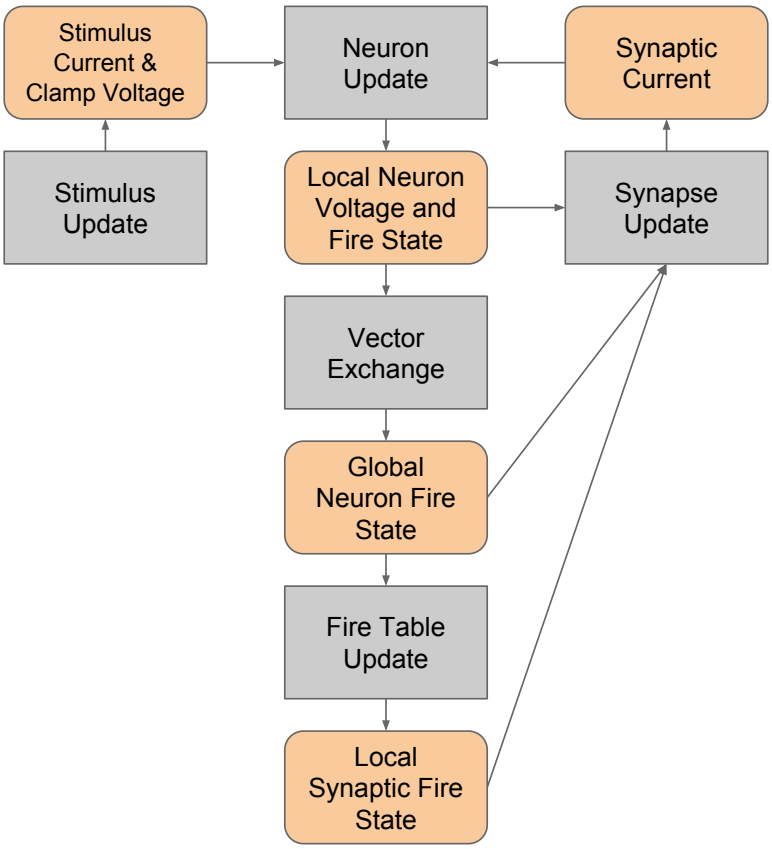

Figure 4.4: Graph decomposition of an NCS simulation. Gray boxes represent computing processes while the orange boxes represent the data that is passed between them.

needed for the next time step and the process begins again. When all subscribers to a data buffer release it, the data buffer is added back to its publisher's resource pool for reuse. For any given stage, a limited number of publishable buffers are used to prevent a stage from consuming all computational resources and getting unnecessarily ahead of any other stages. For example, without limiting the buffer count, because the stimulus update stage requires no data from any other sources, the stage could generate buffers at a rate faster than the neuron updater could consume them, which would waste extra memory and add latency if the stimulus updating does not relinquish processing time to the Neuron Update stage.

Within a single stage, further granularity is gained by parallellizing across subtypes. As an example, if a device simulates both LIF Neurons and Izhikevich Neurons,

the plugins updating each can be executed in parallel. Due to padding from the ID assignments, updates should affect completely separate regions of memory, including operations on bit vectors. Exceptions to this, such as when an stimulus writes to a device-indexed stimulus current for its target neuron, are handled by using atomic operations or by aggregating partial buffers generated by each plugin. The method chosen depends on the type of device and its memory characteristics. While plugins are allowed to update ahead of one another, the results for from a stage at a given time-step will not be published to subscribers until all plugins (in that stage) have updated up to that time-step. This is accomplished by counting the number of plugins operating on a single data buffer. When that number is decremented to zero, the thread responsible for causing that condition publishes the data buffer out to the next stage.

Stimulus Update

The purpose of the stimulus update stage is to compute the total stimulus current to each neuron on the device as well as any voltage clamping that should be done. The stimulus current is represented by an array of floating point values, one for each Neuron (including padding) on the device, initialized to zero at the beginning of each time step. The to which voltage neurons are clamped are stored in a similar fashion where a bit vector is used to select which Neurons khould actually be clamped.

Stimuli are expected to be updated by stimulus plugins designed to handle their subtype. Other than the device-level Neuron ID for each Stimulus that is statically determined at the beginning of the simulation, stimulus plugins rely on no other data from any other stage of the simulation. As such, they are allowed to simulate ahead of the rest of the system as long as they have empty buffers that can be written to and published.

Neuron Update

Unlike the stimulus update stage, the neuron update stage has two dependencies: the stimulus current per neuron published from the stimulus update stage and the synaptic current per Neuron published by the synapse update stage. Given these two pieces of information, this stage is expected to produce the membrane voltage and spiking state of every Neuron on the device. Like the stimulus current, the membrane voltage is represented by an array of floating point values with one value for each Neuron. On the other hand, the spiking state is represented by a bit vector.

Similar to Stimuli, Neurons are expected to be updated by neuron plugins designed to handle their subtypes. Despite receiving and writing data out into device-level structures, neuron plugins operate purely in plugin space. This is possible due to the fact that each plugin is given a contiguous set of device-level IDs during the distribution. As a result, device-level data passed into each plugin is simply offset accordingly to yield the appropriate plugin-level representation.

Vector Exchange

The result of the neuron update stage is the firing state of every Neuron residing on the device. However, synapses are distributed purely based on the postsynaptic neurons and as such the presynaptic neurons could possibly reside on a different device. Thus, to determine synaptic spiking, the state of every neuron in the simulation must be gathered first. Again, the publisher-subscriber scheme is used to pass data asynchronously. However, rather than passing data between stages, it is used to pass data between different data scopes.

Figure 4.5 shows the flow of the neuron spiking information across a cluster. When the device-level vector exchanger receives a local firing vector, the data is published to the machine-level vector exchanger. Within this exchanger, the local vector is copied into a global vector allocated in the system memory. Once all local device vectors are copied for a single time step, the complete machine-level vector is broadcast using MPI to all the other machines in the cluster. This condition is

detected by the broadcaster receiving a signal from each copier signifying that its copy is complete. After all machines in the cluster finish broadcasting, the complete global firing vector is published back to the device-level vector exchangers where it is copied back into the appropriate type of device memory before being published out to any subscribing stages.

Firing Table Update

With the firing state of every Neuron in the simulation, a device can determine when all of its Synapses will receive the firing based on a per-Synapse delay value. Given the potential range of delays, this information is stored within a synaptic firing table. A row of the table is a bit vector representing the firing state of every Synapse on the device. The number of rows in the table depends on the maximum delay of all local synapses. If M is the maximum delay in time steps, then the table must contain at least M + 1 rows in order to record all potential future firings from the current time step. When this stage receives the global neuron fire vector, each Synapse checks its associated presynaptic Neuron for a firing state. If it is firing, the Synapse adds its delay to the current time-step to determine the appropriate vector which is then modified by setting its bit to 1. Figure 4.6 illustrates this dynamic.

After updating the table for a single time-step, the table row associated to that step can be published. However, up to N time-steps ahead of the current time can be published, where N is the minimum delay across all local synapses, since all firings up to that point are guaranteed to have been propagated. This allows devices to simulate ahead of one another to a point rather than being completely locked in step. Additionally, the publication of these extra buffers at the beginning of the simulation allows the data to start flowing through the simulation. As rows are released back to the updater, they are repurposed as future rows in a circular indexing fashion.

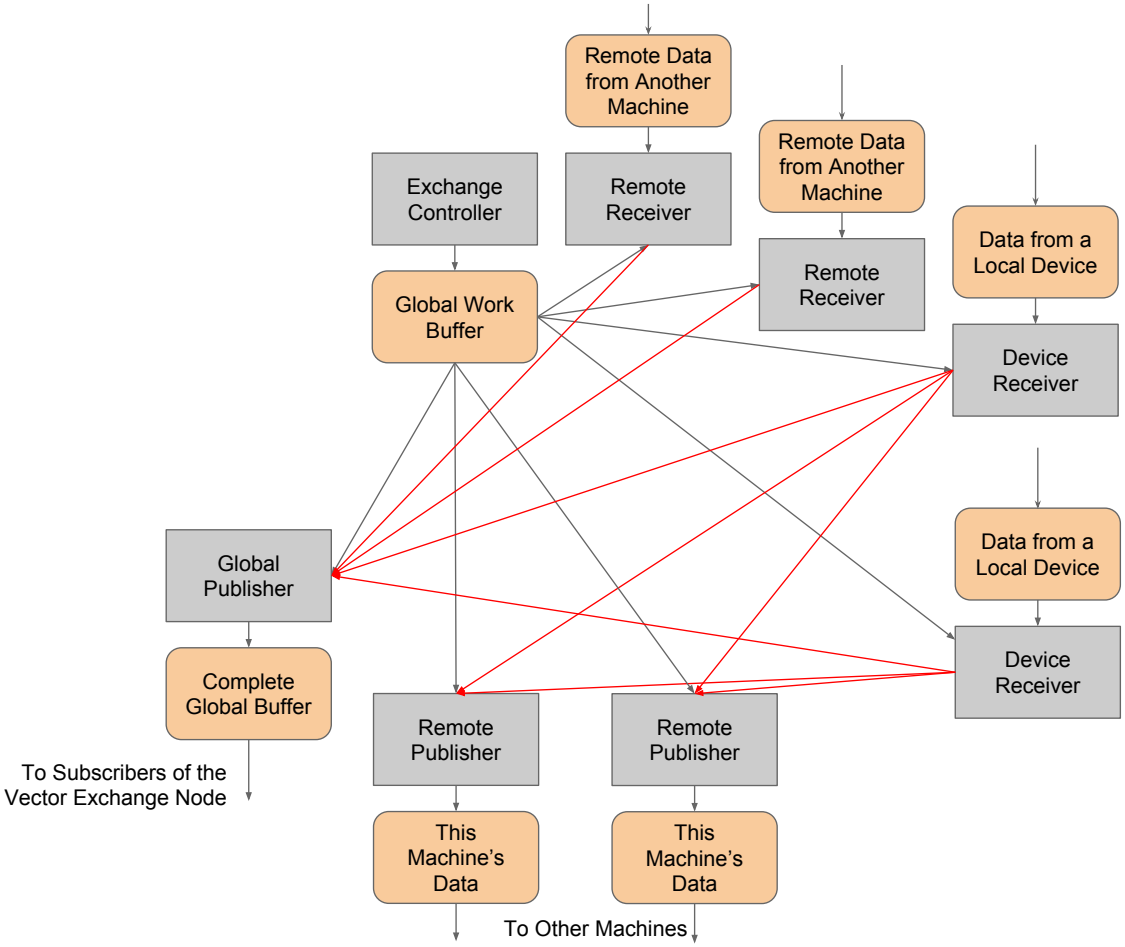

Figure 4.5: NCS communication graph. Gray boxes represent processes while orange boxes represent data. The black arrows indicate the flow of data from process to process while the red arrows indicate the flow of an empty buffer used as a signaling mechanism.

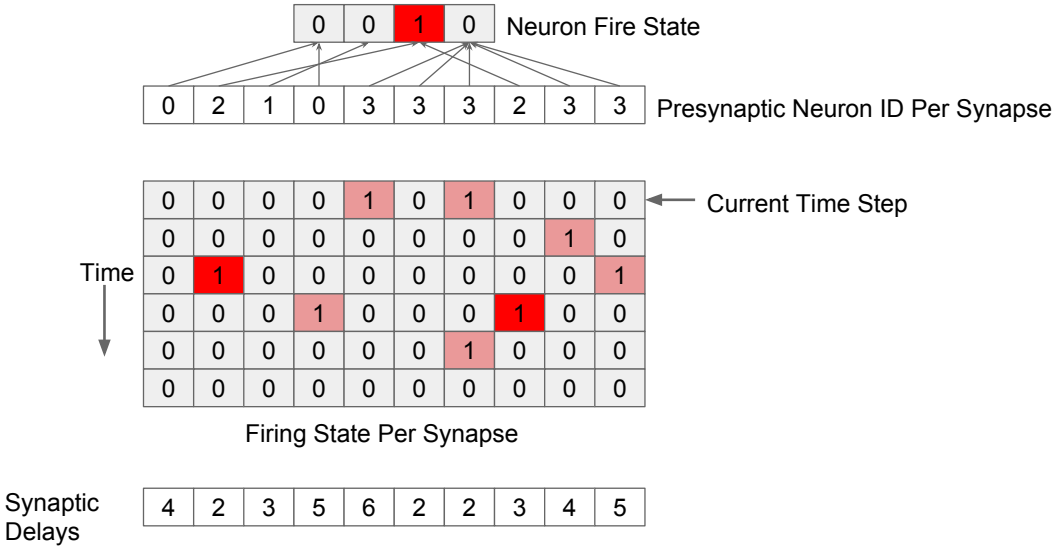

Figure 4.6: An illustration of how the firing table works in NCS6. Data highlighted in bright red denote changes that occur due to neuron firings during the current time step.

Synapse Update

Given the firing state of each Synapse on the device, the Synapses themselves can be updated. Like the stimulus update stage, the synapse update stage produces the total synaptic current per device-level Neuron also represented by an array of floating point values. In terms of operating spaces, synapse plugins update Synapses that operate at both the plugin and device levels, reading from the synaptic fire vector while writing to the synaptic current vector.

4.3.6 Reporting

Reports gather information regarding some aspect of the simulation. They are specified by the user as a set of Elements of the same subtype along with the value that should be extracted from them as the simulation progresses. Because these Elements can be scattered across multiple devices across different machines and because the data required can reside on one of several different scopes, every machine, device, and plugin are given a unique identifier. Following distribution, every Element that must

be reported on can be located by the appropriate ID based on data scope as well as the identifier of the data source.

With these two values, the appropriate data can be extracted during simulation. To accomplish this, a single Reporter is instantiated on each machine that contains at least one Element that should be collected. A Reporter then subscribes to each publisher of the data that it is interested in through a more generalized publisher-subscriber interface. This interface allows a Reporter to access data arrays along with the memory type using a string identifier. At each time step, the Reporter extracts data from all of its subscriptions and aggregates them as necessary. A separate MPI communication group is then used to further aggregate this data across the entire cluster asynchronously before being written out to a file or some other data sink.

In previous versions of NCS, desired reports needed to be specified in the configuration files. In NCS6, reports are instead specified during runtime. The rationale behind this change was to allow users to arbitrarily connect to a running simulation, select a desired set of elements to report on, receive those reports, and disconnect from the simulation while allowing it to continue.

The first iteration of NCS6 implemented a plugin-type interface that was devised in order to provide flexibility in terms of data extraction, aggregation, communication, and output techniques without overly complicating the resulting code. For example, a Reporter that counts the number of Neuron firings may choose to minimize data bus traffic on CUDA devices by implementing the count directly on the device and retrieving the single value rather than by downloading the entire buffer to system memory first before operating on it. Implementations of the Reporter interface are given access to an MPI communication group along with the Element IDs and source identifiers with which to accomplish the aforementioned tasks.

Upon further inspection, it was noted that there were very few realistic extraction and aggregation tasks; users would usually want to extract the precise values of a set of elements or find some count or aggregation of a set of elements. Internode communication of this data was similar for both tasks. What did differ, however, was

the sink to where all of this data was being delivered. Thus, we removed the plugin type for Reporters and instead created a single Reporter built-in to NCS6. For other data sinks, it is up to the developer to implement the appropriate DataSink interface only.

The structure of a Reporter connected to the simulation looks like a tree connected to the requisite computational nodes of the cluster. Tear-down of the tree occurs when one of two events occur: either the simulation is destroyed, or the data sink is destroyed. In the former case, the extractor nodes will receive null pointers when they request data from their publishers. This in turn causes the extractor nodes to delete themselves, signaling nodes closer to the root of the tree to also terminate. This process repeats itself until the data sink itself receives a null pointer, at which point it does any necessary I/O cleanup before destroying itself. The latter case for termination begins with the destruction of the data sink. Nodes publishing to it will realize that nothing is subscribed to its messages and as such will destroy themselves, resulting in a reversed cascading effect.

4.3.7 CUDA Details

Every CUDA plugin in any stage of the simulation flow uses a separate CUDA stream to enqueue work for the GPU, sleeps while waiting for kernel execution to finish, and publishes the results to subscribing stages when the results are ready. Each stream operates independently on separate pieces of data, allowing the CUDA scheduler to execute kernels from different streams concurrently in order to maximize hardware utilization.

Implementation of some model plugins for NCS were rather trivial; for example, Izhikevich neurons as described in Subsection 4.2.2 could be simulated using a simple array of data for each of the six variables that specify each neuron. A similar practice allows for the implementation of an impulse style synaptic connection with STDP learning rules as described by Song, Miller, and Abbott [111]. There are, however, some non-trivial techniques that were used in implementing some other models, most

notably the NCS rendition of an integrate-and-fire neuron, its associated synapse type, and the classical Hodgkin-Huxley neuron. I detail any novel details regarding each in the following subsections.

The NCS LIF Neuron and Synapse in CUDA

Unlike the computationally-straightforward Izhikevich model, the LIF model as specified by NCS [66] presents a number of challenges when implementing it in CUDA. To begin with, LIF neurons can be composed of multiple compartments that affect one another and have different synaptic connections. To maintain minimal data transfer, all compartments of a single LIF neuron are decomposed into neuron-like objects that must be distributed to the same device, localizing cross-compartment interactions to that device. Since each compartment is modeled like a neuron, synaptic connections to specific compartments are realized as well.

An additional complexity of the LIF neuron comes from the ability for a compartment to have one or more channels that alter its current based on a number of different attributes such as the membrane voltage or the calcium content of the cell. The solution to this comes from applying the simulation flow breakdown to this smaller subproblem. Each unique channel type is implemented as a plugin to the larger LIF plugin in order to minimize branching within a single kernel. At each time step, the channel plugins concurrently modify an ion channel current buffer. This buffer is then published to the compartment updater, which in turn publishes the compartments newly updated state for use by the channel plugins in the subsequent time step. Figure 4.7 illustrates this dynamic.

A final challenge to modeling NCS neurons is due to the behavior of firings. Rather than sending a single impulse across a synapse when the neuron fires, a waveform is sent over a potentially large number of time steps. Repeated firings over a short time period produce multiple waveforms that are summed together. To enable this memory of firings in CUDA, the synaptic update plugin behavior is decomposed into a few steps. A synapse begins by checking the fire table to see if a firing has

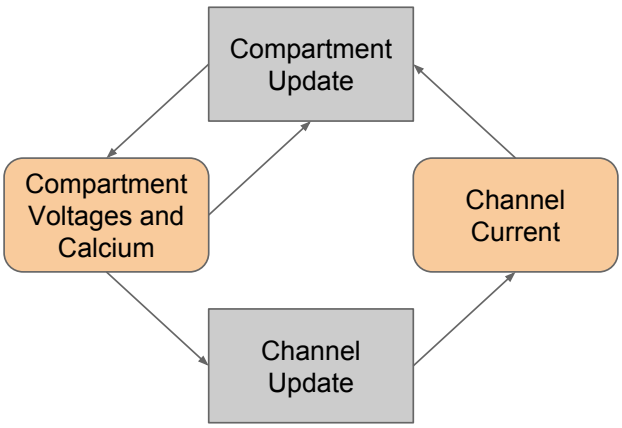

Figure 4.7: Graphical breakdown of NCS neuron update.

been received. If so, it pushes the event composed of a waveform iterator onto a list. A waveform iterator consists of a pointer to a waveform and the current position of the iterator along that waveform. Both the newly generated iterators as well as the iterators generated from previous time steps are then updated, computing the total synaptic current for a single neuron at the same time. If an iterator has not yet iterated across its entire waveform, it is pushed onto a new list that is published for the next time step.

The Hodgkin-Huxley Neuron in CUDA

While the NCS LIF neuron uses Hodgkin-Huxley-like mechanisms for its subthreshold dynamics, it simply follows a templatized spike shape upon crossing that threshold. Iterating through potentially arbitrary spike shapes stored on the GPU requires a level of pointer indirection and extra variables for each neuron; thus, in hopes of improving performance by reducing the number of memory accesses, we also implemented a pure Hodgkin-Huxley model, which inherently accounts for the hyperpolarization dynamics.

Implementation of the Hodgkin-Huxley neuron [68] follows a similar structure as that of the NCS LIF neuron with the cell updating and channel updating structured as two separate subnodes within the plugin. We generalize the channel currents to

be based off of a single generic equation whose variables are specified by the user:

$$y = \frac{A + Bx}{C + exp((x + D)/F)} \tag{4.1}$$

This equation can also be found in GENESIS [1]. Depending on which values are chosen for A, B, C, D, and F, a variety of behaviors can be generated including the parameters of a Hodgkin-Huxley neuron.

A drawback of the classical Hodgkin-Huxley model is its numerical instability when using a forward Euler integration scheme. As such, a much smaller time step on the order of 0.01ms must be used instead of the 1ms time step that can be used with other models. Its integration into a system that often uses a 1ms time step for NCS LIF neurons is simplified by allowing the subgraph decomposition of the Hodgkin-Huxley neuron to run a far larger number of iterations (100 in this case) before the cell updating node actually publishes a result out to the higher neuron updater node. It should be noted for future developers that in these cases, it would be important to weight the computational cost of a Neuron based on its type in order to reflect the amount of computation needed. An alternative method to addressing this instability is to use the exponential Euler method for integration, which is the default integration method for GENESIS [19]. This method is implemented in NCS6 as a different plugin.

4.3.8 pyNCS: Improving Quality of Life for Configuration

While simulation tools are handy for experimentation, they are often difficult to wield with domain-specific languages or configuration files. There have been efforts on several fronts to deal with this problem, many of which provide Python interfaces as a more user-friendly way to specify models [44, 37, 54]. NCS is no stranger to error-prone configuration files; the configuration file used in NCS5 could get quite large depending on the size of the model; moreover, large portions of those files were simply copies of one another with some minor change in some parameter value. Any mistake could potentially be replicated a number of times, making maintenance of

larger models untenable. As such, NCS6 also takes an exodus from configuration files to Python.

NCS6's approach to providing simpler interfaces focuses on decoupling components and allowing users to connect them as they see fit. The simulation core of NCS6 is written using C++11. though not as a program but rather as a library. Developers instead write their own programs that call library functions to run a simulation. A Python interface was automatically generated using SWIG [13], and a more user-friendly interface was wrapped around that. A built-in configuration file was done away with. Users can instead implement models directly in Python or C++, or they can read externally-designed configuration files if a converter exists. An example of a simple brain specification is shown in Figure 4.8.

An example of this workflow is an ongoing project that stores models in JavaScript Object Notation (JSON) [34]. Models are constructed in a custom-built program and then exported into JSON. The JSON files are read in via a Python script, which then converts the input data into the appropriate structures for NCS6, which then simulates them.

4.4 Results

Two types of experiments were run to gauge the performance of the simulator. In the first, several sets of Izhikevich neurons were modeled. The size of these models ranged from 100k to 1 million cells in increments of 100k cells. In each case, 80% of the cells were modeled as excitatory cells while the remaining 20% were modeled as inhibitory cells. Excitatory cells are configured such that the result of a excitatory presynaptic cell firing will raise the voltage of the postsynaptic cell while inhibitory cells are configured to lower the voltage of the postsynaptic cell when the presynaptic cell fires. Each cell has 100 outgoing synaptic connections that use a simple synapse which implements STDP, resulting in 10 million to 100 million synaptic connections total. For the second experimental setup, a model of NCS LIF cells developed by neuroscientists was used. To explore the effects of simulation size on performance,

```
import sys
import ncs

def run(argv):
  sim = ncs.Simulation()
  bursting_parameters = sim.addModelParameters("bursting",
                                               "izhikevich",
                                               {
                                                 "a": 0.02,
                                                 "b": 0.3,
                                                 "c": -50.0,
                                                 "d": 4.0,
                                                 "u": -12.0,
                                                 "v": -65.0,
                                                 "threshold": 30,
                                               })
  group_1 = sim.addCellGroup("group_1",1,bursting_parameters,None)
  if not sim.init(argv):
    print "failed to initialize simulation."
    return

  input_params = {
    "amplitude": 18,
    "width": 1,
    "frequency": 1
  }
  sim.addInput("rectangular_current", input_params, group_1, 1, 0.01, 1.0)
  current_report=sim.addReport("group_1","neuron","synaptic_current",1.0)
  current_report.toStdOut()
  voltage_report=sim.addReport("group_1","neuron","neuron_voltage",1.0)
  voltage_report.toAsciiFile("./bursting_izh.txt")
  sim.step(1000)

if __name__ == "__main__":
  run(sys.argv)
```

Figure 4.8: An example NCS6 configuration written in Python.

the number of neurons in the model were scaled to various levels while maintaining the same level of connectivity; that is, the number of synapses affecting a single neuron remain constant despite scaling. For all experiments, a simulation timestep of 1 millisecond was used.

Experiments were run on a cluster of eight computers in a graphics lab, each equipped with two CUDA-capable graphics cards as listed in Table 4.1. It should be noted that the cards are of varying computational capability in terms of core count and clock speed. The machines are interconnected using gigabit Ethernet, though jumbo frames were not used due to administrative constraints. For each combination of machine count and synaptic count, ten runs were timed and averaged for both 1 second and 10 second simulations.

Table 4.1: Simulation environment.

Machine	**Device 0**	**Device 1**
slurms	GTX 480	GTX 480
kif	GTX 480	GTX 460
nibbler	GTX 480	GTX 480
hypnotoad	GTX 480	GTX 480
clamps	GTX 460	GTX 570
bender	GTX 480	Tesla C2050
robotDevil	GTX 460	GTX 570
wernstrom	GTX 480	GTX 480

The results of the Izhikevich model tests are shown in Figure 4.9 and Figure 4.10 for 1 and 10 second simulations respectively. For a single machine, models failed to run for synapse counts greater than 60 million as a result of memory limits; thus, for failed runs, a value larger than the maximum recorded time was used to denote this. This can be seen as the sharp points in the upper left corner of each plot. The flattened region on lower synaptic counts despite the increase in machine count can be explained by nodes becoming starved for work, resulting in internode communication consuming most of the execution time.

For the NCS model tests, results are shown in Figure 4.11 and Figure 4.12. Due

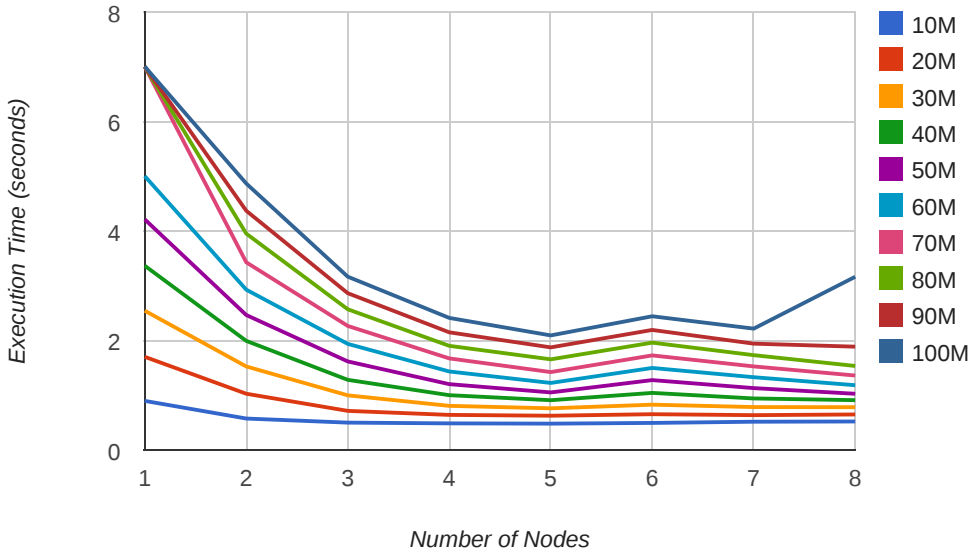

Figure 4.9: Execution time vs number of nodes for a 1 second simulation of Izhikevich neurons. Each line uses a different number of synapses.

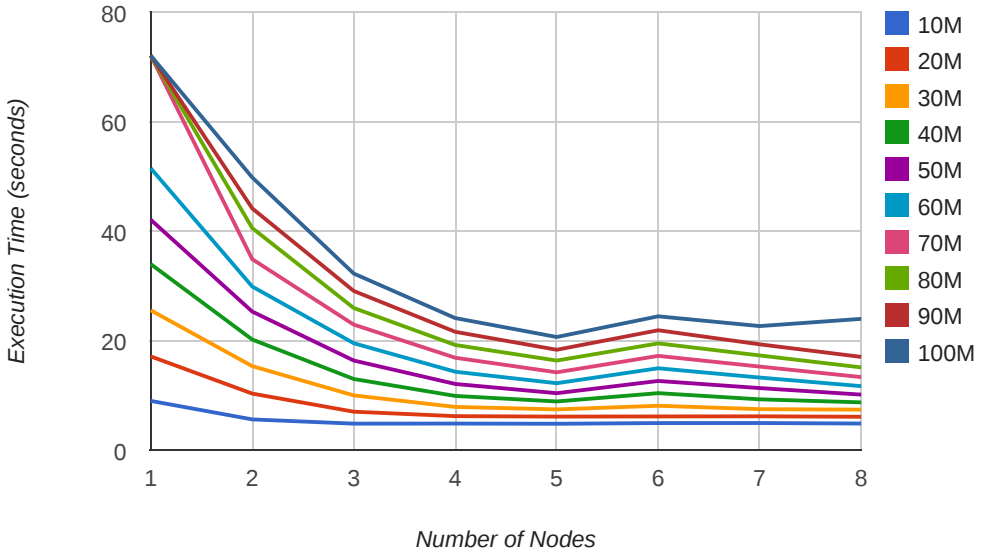

Figure 4.10: Execution time vs number of nodes for a 10 second simulation of Izhikevich neurons. Each line uses a different number of synapses.

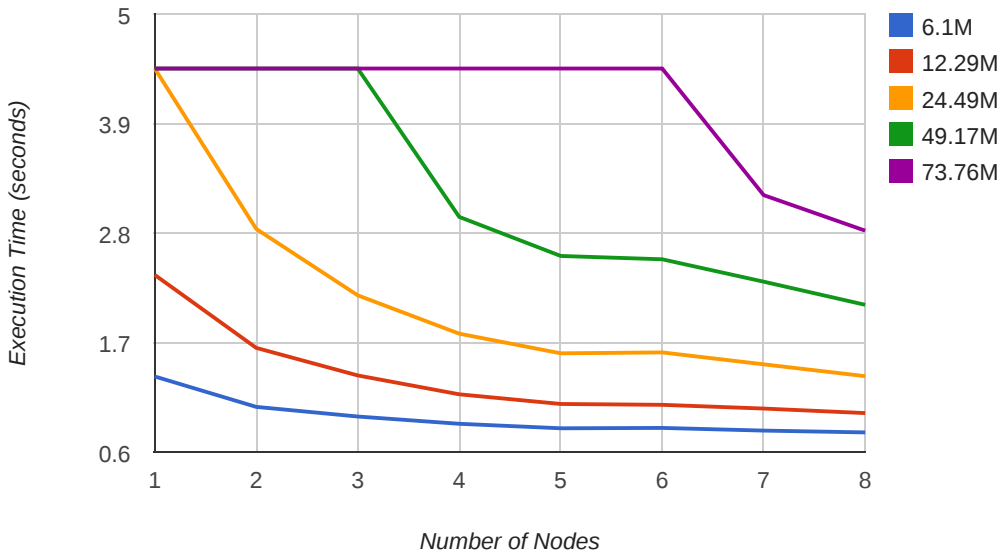

Figure 4.11: Execution time vs number of nodes for a 1 second simulation of NCS LIF neurons. Each line uses a different number of synapses.

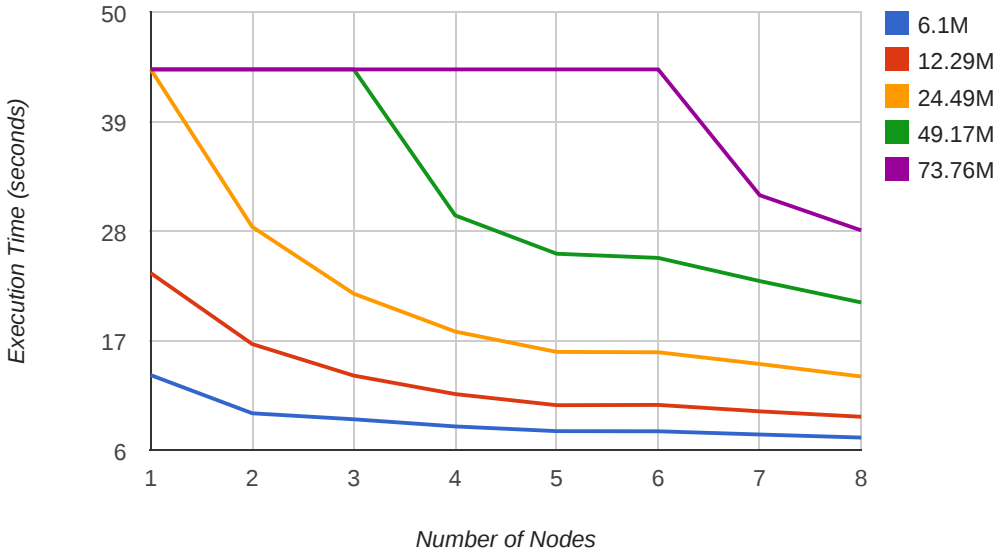

Figure 4.12: Execution time vs number of nodes for a 10 second simulation of NCS LIF neurons. Each line uses a different number of synapses.

to the large amount of memory a single NCS synapse consumes compared to the ones used for the Izhikevich model, fewer synapses could be allocated on a single card, resulting in the larger plateau area.

The 10 second results appear similar to the 1 second results, showing that execution time scales linearly with simulation time. Also of notable interest are the upward trends on execution times with the Izhikevich neurons as the number of machines is increased, particularly when an eighth one is added. These trends can be explained as a limitation on the message-passing technique used in conjunction with the selected network fabric. For gigabit Ethernet, a theoretical 1 billion bits can be transferred per second to and from a machine. Using a 1 ms time step, the amount of data that can be transferred for a single time step in real-time is reduced to 1 million bits that correspond to the firing states of 1 million neurons; however, the number of nodes that any individual node must broadcast to must also be considered. In the case of two nodes, each node can possibly simulate up 1 million neurons and broadcast that data to the other node without oversaturating the network. That number is cut in half when dealing with three nodes as each node must send twice as much data out. Table 4.2 shows the theoretical limits for simulation sizes for up to eight machines with gigabit Ethernet. These numbers scale up or down depending on the network fabric. Note that this is not a problem exclusive to Izhikevich neurons; rather, it is a limitation of the bit-vector messaging scheme that was used.

Table 4.2: Theoretical limits for the number of neurons per machine for real-time simulation.

Machines	**Neurons Per Machine**	**Total Neurons**
2	1M	2M
3	500k	1.5M
4	333k	1.333M
5	250k	1.25M
6	200k	1.2M
7	166k	1.162M
8	143k	1.114M

4.5 Conclusions and Future Work

In this chapter, I presented the latest version of the NeoCortical Simulator, NCS6. Capable of producing results at near real-time for large spiking neural networks on the order to ten to a hundred million synapses on a relatively small cluster of machines as exemplified by the dark blue line in Figure 4.9, NCS6 can provide a simulation solution using readily-available computing resources. It allows for submodels to be repurposed and reused with minimal effort by allowing for different simulation models to be combined and is extensible enough that new simulation models can be added into the system without the need to rebuild the rest of the system.

Though designed for increased user productivity, NCS6 is not without its shortcomings, many of which should be addressed in future work. One that has been addressed by other research is the message-passing scheme whose drawbacks were made apparent in the results in Section 4.4. Thibeault et al. [121] uses a hybrid scheme that switches between the currently-implemented bit vector scheme and a more traditional address event representation depending on the amount of firing that is actually occurring. Implementation of this in NCS6 would require the replacement of the communication node of the simulation flow graph with a similar one that accounts for the firing rates of all the neurons in the system.

While the implementation of more models is an obvious direction for future work, the workflow for implementing these plugins can prove to be tedious. The general structure for many neuron models is a set of equations that are used to update their states. To implement this relatively small set of equations, however, requires a large deal of boilerplate code to initialize arrays, execute CUDA kernels, and parse input parameters. Creating tools that can interpret these equations and automatically generate the necessary plugin code would enhance NCS6's use as an experimental platform.

In general, more tools to aid users would be beneficial. There has been some work done already on visualization tools [29] as well as ongoing efforts to create web-

based model repositories and model builders. Finally, while the results presented here were performed on a small heterogeneous cluster of GPUs, more performance characteristics would be interesting to collect on much larger clusters with more diverse hardware characteristics.

Chapter 5

caVR

5.1 Introduction

As modeling allows users to study things that would be impractical or even impossible to study otherwise, virtual reality brings the promise of granting users experiences otherwise burdened by similar impracticalities. Though the concept has been around for quite a while, with Ivan Sutherland in 1965 describing systems that "[w]ith appropriate programming... could literally be the Wonderland into which Alice walked" [116], we are not there yet. In fact, the area is an active playground for experimentation as both researchers and commercial companies alike search for that "ultimate display." Recent advances in rendering and computing technology has only expanded that playground.

caVR is a library designed to allow developers to quickly design virtual reality applications by providing a common interface for both input and output methodologies. It is extensible enough that new input and output modalities can be added to the library without need to rebuild the core of caVR or programs that use it in most cases. The rest of this chapter is structured as follows: Section 5.2 gives some background on virtual reality both in terms of hardware and software. Section 5.3 discusses the design of caVR while Section 5.4 discusses a few projects and applications that use it or its predecessor Hydra. I conclude in Section 5.5 with closing remarks and future work.

5.2 Background

Virtual reality is somewhat of a nebulous term, often defined by the presence of certain pieces of hardware. A more appropriate definition may be found by Steuer: "A 'virtual reality' is defined as a real or simulated environment in which a perceiver experiences telepresence," where telepresence "is the experience of presence in an environment by means of a communication medium" [113]. I can divide the subject into three intertwining components: the communication medium, applications that try to expose telepresence, and the software toolkits that allow those applications to run on all of this medium. I now delve into each of these topics individually.

5.2.1 Communication Medium

A common misconception about virtual reality is that it is primarily about hardware [113]; however, there is truth in it: a great of hardware has been developed and experimented with in order to increase a user's sense of telepresence. Hardware is usually purposed for either outputting information to the user or receiving input from that user, though that line is occasionally blurred.

Arguably the most developed space in terms of virtual reality hardware would be visual rendering technology. In addition to advancing graphics technology, various display technologies have been developed to better immerse the user. Amongst them are large screen displays and head-mounted displays. Examples of the former include literal large screens [30] and walk-in environments such as the CAVE [35]. Examples of each of these are shown in Figure 5.1 and 5.2 respectively. These devices increase immersion by attempting to maximize their presence in the user's field of view. Additionally, many of these devices use passive or active stereoscopy in order for users to perceive depth. With passive stereo systems, the left and right eyes are displayed on a surface simultaneously; however, each image is polarized in a different manner. Glasses with corresponding polarities are used to force each eye to see only their appropriate images. On the other hand, active stereo systems work by alternating

Figure 5.1: An example of a large screen display [87].

between the two eyes. A different pair of specialized glasses shutters out the correct image in a synchronized manner [30]. Meanwhile, head-mounted displays, like the one shown in Figure 5.3, are worn on the user's head. While earlier versions were plagued by weight limitations and other factors [42], recent developments such as the Occulus Rift [127] are introducing this technology to a much wider audience.

While visual hardware development has been in the limelight, the other senses have not gone completely ignored. In the audio domain, 3D sound can be presented using earphones or loudspeakers. Each has its own advantages and disadvantages. Earphones present sound directly to the user and can block out outside interference but cannot simulate vibrations in the body that can be felt when low frequency sounds are generated. Loudspeakers, on the other hand, have difficulty handling interference as well as conveying spatial information as a cost for being able to generate the

Figure 5.2: A CAVE-like environment [38].

Figure 5.3: A user with a head-mounted display [105].

aforementioned vibrations [42]. On the topic of vibrations, difficulties in accurate audio output also include the problem of generating the correct sound base on the virtual environment being conveyed; real-world effects such as echoes can become computationally expensive to simulate [107].

Other senses, such as touch and smell, have also received some attention. The former is provided via haptic devices such as exoskeletal gloves or vibrating motors. Like head-mounted displays, haptic devices suffer from causing fatigue due to the additional weight [27]. Olfactory displays can generate smells, though this form of hardware appears to be custom-built for research purposes [132, 133].

The devices discussed so far only facilitate communication in one direction: from the application to the user. Equally important is communication from the user back to the application. Input devices for virtual environments go beyond the standard keyboard and mouse. Given the 3D nature of many output devices, input devices must also be usable in a 3D space. These devices involve tracking parts of the body including the hands, the head, and the eyes. A number of solutions exist that balance cost with accuracy and precision. Ultrasonic and inertial trackers, such as the Intersense IS-900 [131], provide information on all six degrees of freedom for any given tracker. The wand input held by the hand additionally provides button and joystick inputs. The downside to these trackers is their prohibitive cost; however, developments in the videogame industry may be ameliorating this problem with more affordable options such as a Nintendo Wii controller [130] and the Razer Hydra [11], both of which provide wand-style inputs and can be augmented to only provide 6-DOF tracking. Tracking can also be achieved via computer vision, and like the previous two examples, new products like the Microsoft Kinect [94] and Leap Motion Controller [129] mark an increase in affordability and availability. increased availability.

5.2.2 Applications

There is a large variety of virtual reality applications that have been created over the years. The purpose of these applications ranges from testing new interaction methods

to training users in otherwise impractical environments to testing the effects of virtual environments themselves. I present a few examples of each of these application types.

In the domain of interaction methodologies, the extra dimension leads to potentially more intuitive or more cumbersome interfaces. Stoakley et al. [114] present a method for navigation in which the users have access to a miniature version of the virtual world that they are able to manipulate in order to perform tasks or even move themselves by picking up their avatars and placing them elsewhere. Poupyrev et al. [100] present an interaction method in which the user's hand is tracked. In order to manipulate nearby objects, users can simply grab or touch them; however, to manipulate objects out of their physical reach, virtual hands are rendered based on the direction of their actual hands.

Virtual reality training applications are in no short supply, with a tall order found just within the medical domain. Ahlberg et al. [4] researched the use of virtual reality to train residents to perform laparoscopic cholecystectomies. Their findings show a reduction in error rate when residents were trained in a simulator before performing a set of real cholecystectomies. Seymour et al. [103] found similar results for the same task.

There are numerous risks and design considerations that must be addressed with virtual reality technology. Systems must be designed given the limitations of the individual senses. A review of these concerns can be found by Stanney et al. [112]. A very real concern is simulator sickness, a phenomenon similar to motion sickness, though it can occur without actually moving the user [77]. Applications have been developed to measure the extent of these problems as a function of other variables such as field of view [81].

5.2.3 Toolkits

Facilitating the development of applications without the need for information about the underlying hardware are virtual reality toolkits and libraries. A review of many of these tools can be found by Bierbaum et al. [16].

For shared memory machines, FreeVR [104] is an open-source VR library written in C. Designed to be a low level library, FreeVR abstracts away input and output details but little else in terms of managing or providing content, though a number of other libraries have been used in conjunction with it to fill in this gap [106]. The deficiencies found by Bierbaum et al. [16] led to the creation of VRJuggler. Similar to FreeVR, it is designed to be low-level and non-intrusive. Additionally, later versions are able to run on clusters [2].

Unlike the previous two examples, VRUI [78] was designed as a high-level VR library. A consequence of this is that VRUI applications tend to have the same look and feel. Higher level concepts like navigation are automatically handled by VRUI. VRUI is also capable of running in clustered environments.

While many toolkits handle both input and output abstraction, VRPN [118] was deliberately designed to only handle inputs. The design of VRPN employs a server daemon that abstracts away input details. Applications then connect over a network pipe to the server in order to query for updates regarding any particular input.

5.3 Design

The author's first attempt at a VR library, Hydra, came about from the limitations brought on by FreeVR. The low-level design allowed for rapid development of applications; however, the lack of cluster support reduced portability as hardware was being upgraded from shared memory machines to distributed systems.

caVR is considered to be an upgraded version of Hydra. The goals when designing both were to create a VR library that was easy to use for developers and users, extensible to new devices and input paradigms, and could be used on a large variety of hardware configurations. Several quality-of-life improvements were made in order to allow developers to more rapidly write programs and users to more quickly configure their environments. Structurally, the two systems are very similar otherwise; I now delve into the design of that structure.

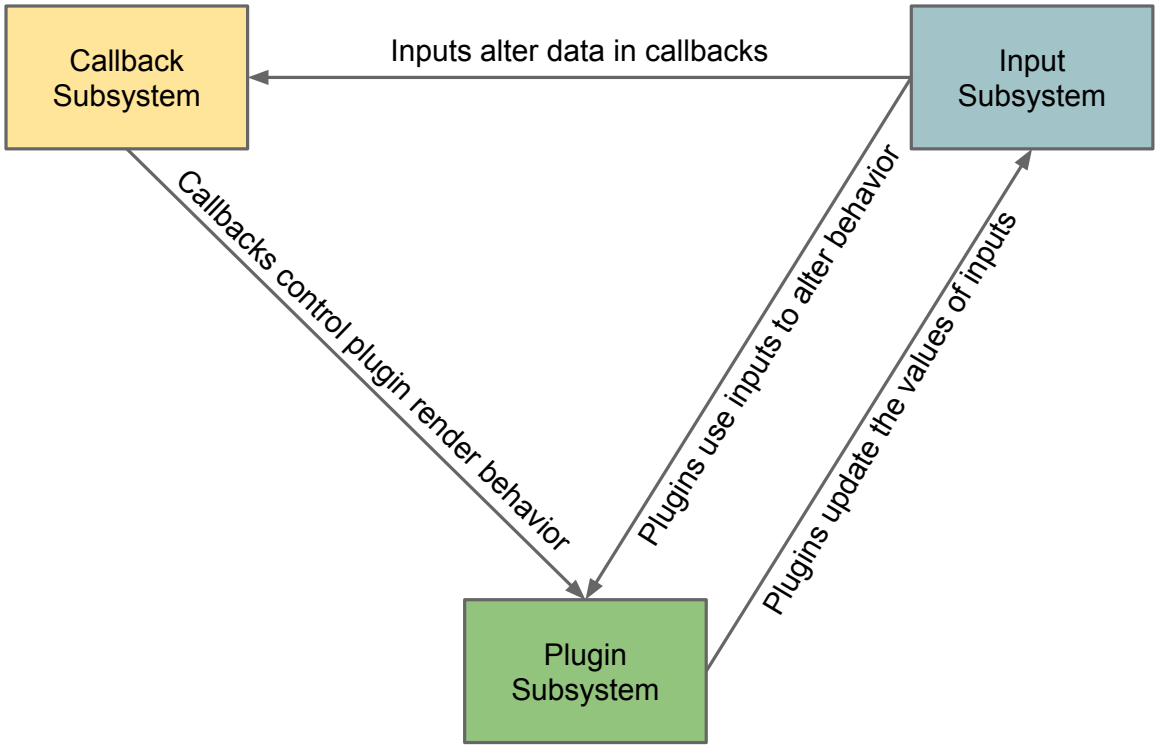

Figure 5.4: The three core subsystems of caVR and their interactions.

5.3.1 Subsystems

At the most basic level, a VR program collects inputs, updates itself based on those inputs, and outputs information back to the user. To facilitate all of these activities, caVR provides three core subsystems: a callback subsystem, an input subsystem, and a plugin subsystem. The relationship between the three subsystems are visualized in Figure 5.4.

The callback subsystem allows application developers to customize the behavior of their programs. Developers register a set of callbacks to a set of strings; caVR will then call the appropriate callbacks if and when they are needed. Some callbacks, such as the "update" callback, will most certainly be used by caVR itself; others such as "gl_render" will only be called if an OpenGL renderer is configured to call it. Developers need only specify callbacks for the types of inputs and outputs that they wish to support. The design is reminiscent of GLUT [74].

Within any of these callbacks, developers have access to the input subsystem that provides details on the current state of every registered input such as whether a button is pressed or where in physical space a 6-DOF tracker is positioned at a given point in time. caVR abstracts all inputs into three categories: Buttons, Analogs, and SixDOFs. Buttons as an input mirror their physical world counterparts; they behave as a two-state toggle that can be pressed, held, released, or left open. Analogs are used for inputs whose value resides within a continuous scale; they return a normalized value between -1 and 1. Finally, SixDOFs represent the position and orientation of a tracker in space. It should be noted that developers access inputs using alias strings such as "exit" or "head"; the rationale behind this decision is to further separate the developer from the underlying devices that may be driving the system.

The last component, the plugin subsystem, allows devices to interact with the program. Plugins have access to both the input and callback subsystems. Its access to the former subsystem allows plugins to alter the state of any input at any point in time, facilitating the use of input devices. In these cases, plugins access these inputs via their device names rather than their functional aliases. The plugin subsystem's access to the callback system allows plugins to call the appropriate callbacks to drive output devices. For example, an OpenGL renderer would call a registered "gl_init_context" to initialize OpenGL-specific data, "gl_update_context" to update that data before each frame, and "gl_render" to render to each surface that it is responsible for. Two different plugins might use the same set of callbacks; in the case of OpenGL callbacks, suppose that a rendering plugin was developed for X11 windows when running under Linux and a different plugin was developed using WGL to open windows under the Windows operating system.

5.3.2 Execution Flow

At the start of a program that uses caVR, a configuration file is read that specifies the machines involved in the system along with a set of plugins that should be loaded on each machine. A separate input mapping file is also loaded that binds a device

name that an input plugin would access to a functional alias that is accessed by the application developer. The machine that originally executes the application is deemed the master and is responsible for collecting all inputs. The master machine forks a copy of the application to every other machine involved in the system via ssh; these worker machines in return form a master-slave configuration by subscribing to updates published by the master. Plugins on every machine are then configured, and each begins executing in an endless loop in its own thread.

A main loop also begins to execute, repeatedly calling the registered "update" callback. Before each update call, the current state of every input is locked by the master; this locked state is the state that is actually visible to the developer during the execution of the "update" callback. This state is published to every other machine in the system along with a time delta. Workers take this state and force the locked state of their individual input subsystems to be exactly the same as the master's. The overall effect of this scheme is that every machine in the system is updated the same way, resulting in the same application state to be stored on every machine in order to prevent desynchronization.

An additional step found in caVR but not Hydra is the execution of a "publish_data" callback by the master after each call of "update" and the execution of a "receive_data" callback before the "update" call on each worker. The addition of this behavior allows for programs to take a split-middle approach to application design; instead of recomputing information on every machine or perhaps receiving that information from some external source for every machine, only the master is allowed to compute this data and then publish it out to subscribing worker nodes to use. The benefit of this approach is that worker nodes need not be as computationally powerful as the master node.

Execution of the program stops when the developer calls a specially-defined "shutdown" function within the "update" callback, at which point all threads will break out of their loops and rejoin the main thread. Plugins on each machine are torn down before the remote processes themselves are destroyed. Finally, the master

process terminates.

5.3.3 Extensions

The design of caVR leaves the core incredibly lean and portable; however, it comes with the drawback of providing little functionality on its own. For example, for an OpenGL render callback to function properly, the developer needs access to the transformation matrices required to render the scene properly. Incorporating access to these structures in the caVR core means that caVR now has a direct dependency on OpenGL. Instead, a middle ground approach is taken by incorporating extensions. The caVR GL extension provides this functionality through its own set of thread-local variables and function calls. Plugin developers who need to expose GL properties to application developers use this interface to communicate these details. Both the plugin and application link to the extension's library to harness this functionality. This approach allows caVR to be extensible without being weighed down by more and more dependencies as new APIs start being used.

5.3.4 Implementation Details and Differences from Hydra

Unlike Hydra which was written in C++, caVR is written in C++11. The primary boon gained from the language update is simplification of the callback subsystem as constructs such as lambdas can be used. Threading constructs are also built-in, increasing ease of portability. A templatized mathematics library is also provided with swizzle semantics similar to that of GLSL, and some basic geometry and graphics libraries were created to replace the legacy fixed functionality lost in recent versions of OpenGL.

While networking in Hydra was also specified using plugins that could allow it to form networks using anything from TCP to Morse code, it was deemed cumbersome and superfluous for the user to configure for very little gain. The networking structure was replaced with a single networking library, ZeroMQ [61], that makes the underlying communication transparent to the user.

```
import("defaults.cfg");
self = {
  hostname = HOSTNAME;
  ssh = HOSTNAME;
  address = "tcp://" .. HOSTNAME .. ":8888";
  plugins = {
    x11_renderer = x11_renderer;
    vrpn = vrpn;
  };
};
machines = { self; };

self.hostname =  OVERRIDE_NAME ;
```

Figure 5.5: An example configuration of a caVR system in Lua.

Further simplifying the configuration of systems is the new configuration file scheme. Hydra used a custom-designed configuration language that would most likely be described akin to JSON. While simple enough for users to understand, it lacked the ability for users to embed logic within those files, resulting in the use of multiple sets of files based on whether the user wanted a certain parameter to be enabled or not. To ameliorate this situation, the configuration files now use Lua [70] instead. While remaining readable, the Lua configuration files allow for logic to be employed and precise parameters to be overrode later on through table accesses (overrides in Hydra usually required respecification of entire sections). Configuration files are also separated into two sets of files for similar reasons. The first file specifies only the machines (which can be seen in Figure 5.5) and their plugins while the second file specifies the input mapping between device names and functional names as the second file represents input bindings that are usually highly specific to the application while the first file can be reused for all applications running on a given system.

The same style of Lua files are also used to define the configuration file schemas themselves. The advantage of this is that configuration files can be validated without running the program itself. This is particularly advantageous for the configuration of plugins; a validated configuration file will be guaranteed to successfully configure

```
import("plugin.lua")
machine = {
  hostname = {
    type = "string";
    required = true;
    description = "hostname as specified by the environment variable
        HOSTNAME";
  }; -- hostname
  ssh = {
    type = "string";
    required = true;
    description = "ssh address to machine; assumes shared keys";
  }; -- ssh
  address = {
    type = "string";
    required = true;
    description = "zeromq-style address of the machine for communications";
  }; -- address
  plugins = {
    type = "list";
    required = true;
    subtype = plugin;
  }; -- plugins
}; -- machine
```

Figure 5.6: A caVR schema for the specification of a "machine" in Lua.

a plugin, which reduces the amount of boilerplate error checking that must be done. Additionally, the schema files are available outside of the program, allowing configuration tools to be more easily developed. An example of a schema file is listed in Figure 5.6.

Both versions of the VR library come with some simulator tools to aid in development. Particularly, all X11 rendering plugins come with the capability to capture keyboard input from the window itself. Additionally, a flag can be set that forces the window to entire a simulator state where developers have access to a set of dummy Analogs and SixDOFs. The ability to open multiple windows allows developers to check whether their rendering algorithms are correct or not. Figure 5.7 shows the simulator running on a very simple test application.

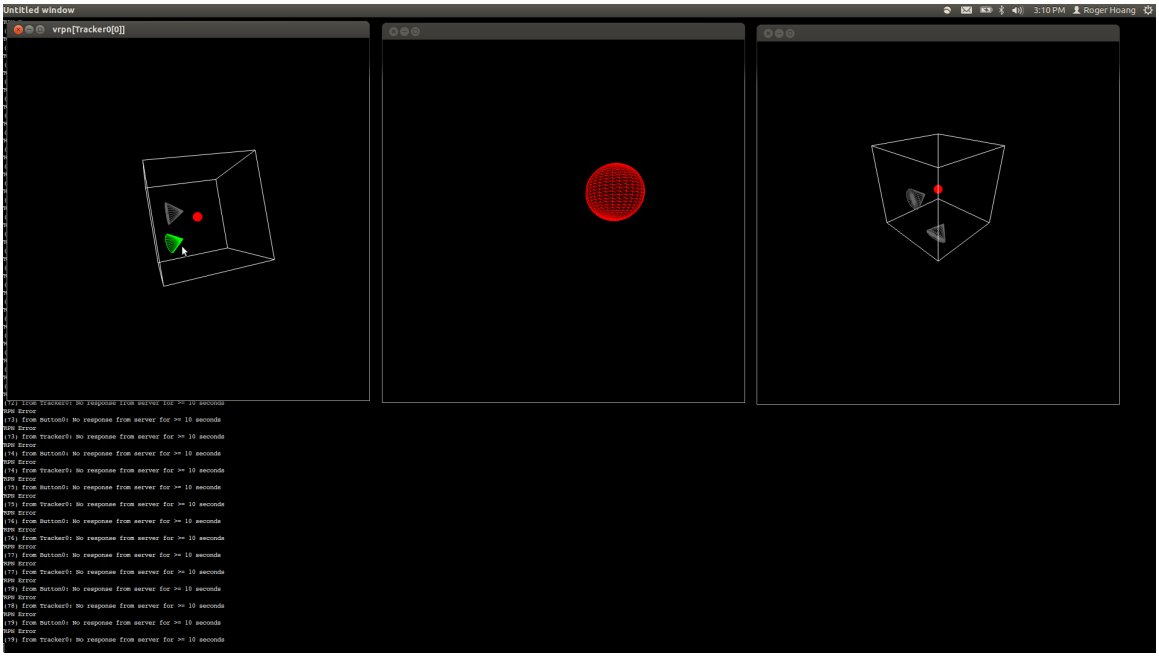

Figure 5.7: A caVR application running with simulator windows to test behavior.

5.4 Results

caVR is the latest iteration on the development of a VR library that began with Hydra. Over the years, a number of projects have successful used Hydra to create VR applications, the first of which was VFire [65], a wildfire simulation and visualization tool. The application allows users to load terrain information about an area including vegetation and moisture details and simulate wildfire spread using that data. Parameters such as wind speed and direction can be altered on the fly, and the effects of adding barriers can also be visualized. Figure 5.8 shows VFire running in a CAVE-like environment.

Another application developed in the same environment was the Radiological Immersive Survey Training (RIST) tool [76]. RIST places Civil Support Team members into a virtual environment where some radioactive source has been placed. The system mimics the real-world task of CST members, where they must havigate the environment and locate the source without exposing themselves to too much radi-

Figure 5.8: VFire, a Hydra application, running in a CAVE-like environment [65].

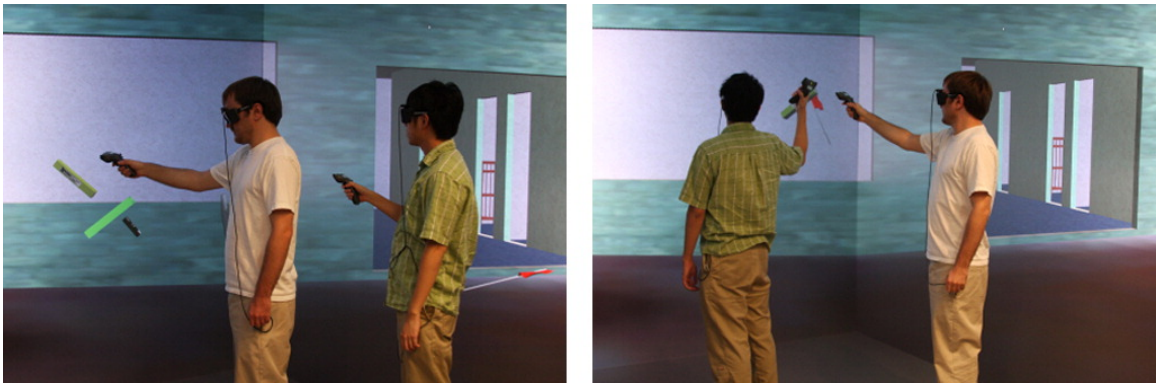

Figure 5.9: RIST, a Hydra application, running in a CAVE-like environment [76].

ation in the process. Shielding models are provided, and the system also employs a unique multiperspective rendering technique since survey teams generally work in pairs. Figure 5.9 shows RIST running in a CAVE-like environment.

For the purposes of improving immersion in virtual environments, Hydra was also used in researching global illumination performance in those environments. While real-time performance is already a difficult problem in a regular desktop setting, finding an adequate solution for a clustered, multi-screen environment only complicates matters. Two techniques were implemented with markedly different performance and image quality characteristics [62]. An example of one of the techniques running in a CAVE-like environment is seen in Figure 5.10.

On the plugin developer front with experimental outputs, remote streaming plugin was successfully added that allowed for images to be rendered by a more powerful server machine and then streamed to less powerful devices such as tablets and smartphones. This allows any device with a screen to be used as an output device regardless of whether the application itself is installed or whether it has adequate rendering power to render images. As an example, Figure 5.11 shows VFire running on a smartphone.

Beyond its use for research project, Hydra has been employed in teaching a class about virtual reality. Students successfully created applications such as immersive paint programs, planetary simulators, molecular visualizers, and games.

Figure 5.10: Experimental global illumination techniques being tested in a CAVE-like environment [62].

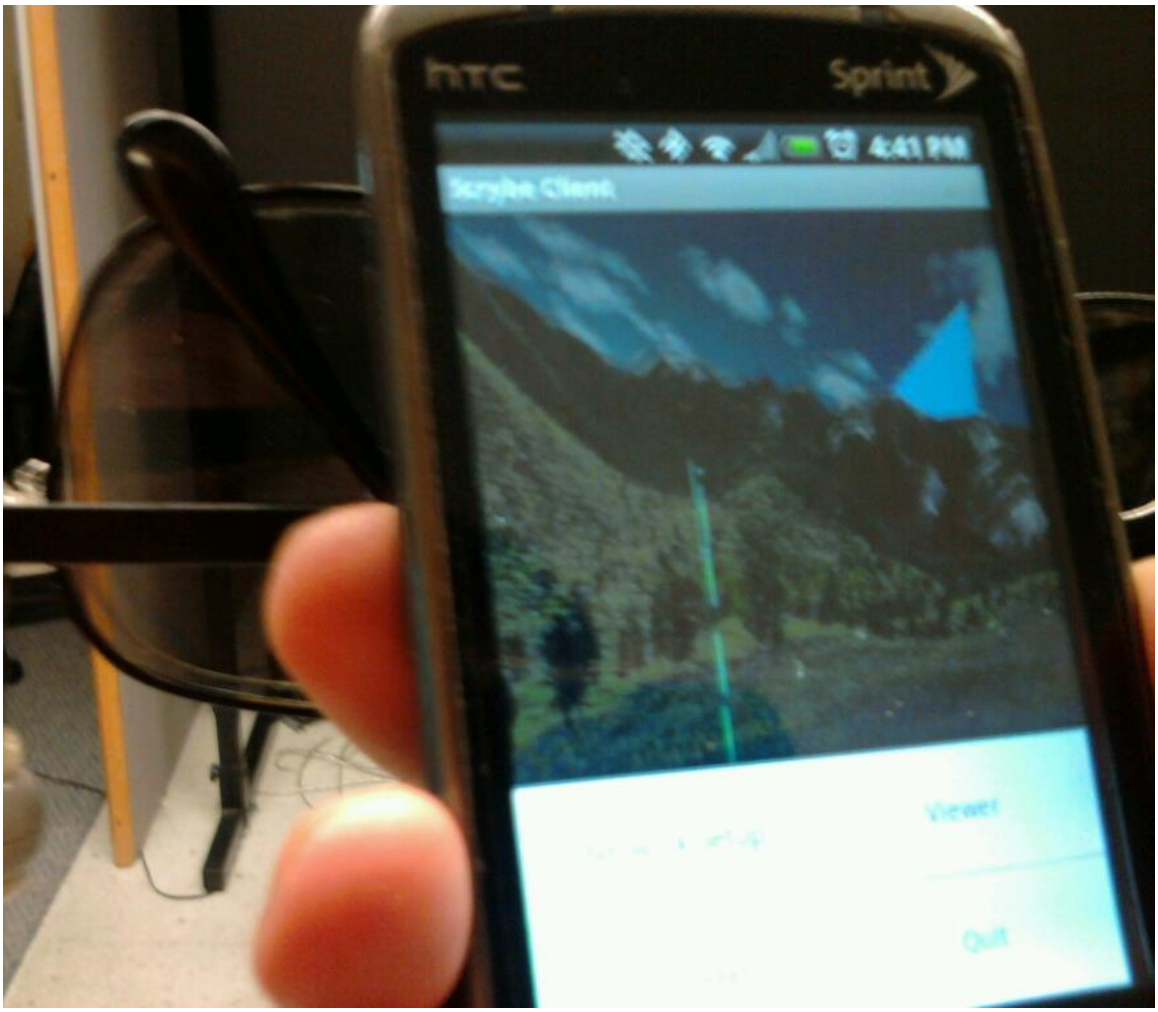

Figure 5.11: An Android phone being used as a rendering surface with Hydra [63].

5.5 Conclusions and Future Work

In this chapter, I presented caVR and its predecessor Hydra. The design of caVR allows it to be flexible for the application developer, the plugin developer, and the end users. The flexibility also extends to the hardware on which applications using caVR can run, from shared memory machines, to Beowulf clusters made from ordinary computers in graphics labs, to powerful dedicated clusters driving CAVE-like systems. I discussed a variety of applications that have used Hydra in order to create new virtual environments, explore rendering techniques, and experiment with new output devices.

While these results are promising, there is nevertheless room for improvement. Beyond the obvious need for more applications and plugins, there are other less apparent areas for future research. For example, while the input subsystem allows for inputs of three specific types, it could be restructured to support any arbitrary combination of input types by combining the concept of extensions with the new callbacks that allow for user data to be broadcast across all nodes in the system. Explained in depth, each input type could be an extension that registers itself with caVR. The input synchronization step would then have each input extension serialize its own data to be published to the worker nodes, where those same extensions would be responsible for deserializing and updating its own data on the remote machines.

Along the same vein, a layer of software could be built on top of caVR to accelerate development. In particular, one could imagine an API that allows parameters of various types to be specified by the developer along with the names, descriptions, and default values of these parameters. These same parameters would then be available to plugins that could automatically generate the correct interfaces, such as an HTML page accessible by any device with a web browser. Analog values could be represented by sliders, buttons by checkboxes, and string inputs by text boxes. Changing these values would automatically be reflected in the application as well as any other automatically generated interfaces. Forseen difficulties include automatic generation of a

reasonable mapping given the characteristics of a parameter. For example, a floating point value that scales from 1 to 10000 may not be effectively presented as a small linear slider between the two values; it may be more advantageous to provide a slider with a logarithmic mapping.

Finally, on the topic of inputs, the newly designed schema files allow for configurations to be validated externally. And while the switch to Lua goes a long way in increasing the power and simplicity of configuration files, they still require an end user to understand the programming language and all its idiosyncrasies. A more reasonable approach would be to provide configuration tools through some sort of graphical user interface as valid values for each parameter are also specified within the schema. Once the user configures their system in this more intuitive interface, the configuration tool can automatically generate the appropriate configuration files.

Chapter 6

Conclusions

This dissertation presents a method for dealing with clusters of heterogeneous hardware in an efficient, extensible fashion. I apply this approach to two very different applications: a throughput-critical neuron simulator and a latency-critical virtual reality library. In the former application, I developed NCS6, a neocortical simulator capable of harnessing all available hardware in a cluster. It allows multiple models to be connected and simulated together, and I show how certain non-trivial models to CUDA devices. For the latter application, I designed a library that accounts for the ever-shifting input and output devices and paradigms found in virtual reality research. I discussed improvements made to the library and presented various projects that have successfully applied the library.

Bibliography

[1] GENESIS Modeling Tutorial. `http://www.genesis-sim.org/GENESIS/Tutorials/genprog/chantut.html`. Accessed April 22nd, 2014.

[2] The VR Juggler Suite. `http://vrjuggler.org/features.php`. Accessed April 28, 2014.

[3] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[4] Gunnar Ahlberg, Lars Enochsson, Anthony G Gallagher, Leif Hedman, Christian Hogman, David A McClusky III, Stig Ramel, C Daniel Smith, and Dag Arvidsson. Proficiency-based virtual reality training significantly reduces the error rate for residents during their first 10 laparoscopic cholecystectomies. *The American journal of surgery*, 193(6):797–804, 2007.

[5] Thomas E Anderson, David E Culler, and David Patterson. A case for NOW (networks of workstations). *Micro, IEEE*, 15(1):54–64, 1995.

[6] Edward Angel. *Interactive computer graphics*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[7] Sridhar Reddy Anumandla, Laurence C Jayet Bray, Corey M Thibeault, Roger V Hoang, Sergiu M Dascalu, FC Harris, and Philip H Goodman. Modeling oxytocin induced neurorobotic trust and intent recognition in human-robot interaction. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 3213–3219. IEEE, 2011.

[8] Mark Bakery and Rajkumar Buyyaz. Cluster computing at a glance. *High Performance Cluster Computing: Architectures and System. Upper Saddle River, NJ: Prentice-Hall*, pages 3–47, 1999.

[9] Amnon Barak, Tal Ben-Nun, Ely Levy, and Amnon Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–7. IEEE, 2010.

[10] Jorge Barbosa, João Tavares, and Armando J Padilha. Linear algebra algorithms in a heterogeneous cluster of personal computers. In *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*, pages 147–159. IEEE, 2000.

[11] Aryabrata Basu, Christian Saupe, Eric Refour, Andrew Raij, and Kyle Johnsen. Immersive 3DUI on one dollar a day. In *3D User Interfaces (3DUI), 2012 IEEE Symposium on*, pages 97–100. IEEE, 2012.

[12] Olivier Beaumont, Vincent Boudet, Antoine Petitet, Fabrice Rastello, and Yves Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). *Computers, IEEE Transactions on*, 50(10):1052–1070, 2001.

[13] David M Beazley. Swig: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk workshop*, pages 129–139, 1996.

[14] Donald J Becker, Thomas Sterling, Daniel Savarese, John E Dorband, Udaya A Ranawak, and Charles V Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, volume 95, 1995.

[15] Mohammad A Bhuiyan, Vivek K Pallipuram, and Melissa C Smith. Acceleration of spiking neural networks in emerging multi-core and GPU architectures. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[16] Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker, and Carolina Cruz-Neira. VR juggler: A virtual platform for virtual reality application development. In *Virtual Reality, 2001. Proceedings. IEEE*, pages 89–96. IEEE, 2001.

[17] David Blythe. The Direct3D 10 system. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 724–734, New York, NY, USA, 2006. ACM.

[18] C. George Boeree. General Psychology. `http://webspace.ship.edu/cgboer/genpsy.html`. Accessed April 20th, 2014.

[19] James M Bower and David Beeman. *The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System.* Electronic Library of Science, The, 1995.

[20] James M Bower, David Beeman, and Michael Hucka. The GENESIS simulation system. `http://authors.library.caltech.edu/36220/1/bower_2003.pdf`, 2003. Accessed May 14th, 2014.

[21] Laurence C Jayet Bray, Sridhar R Anumandla, Corey M Thibeault, Roger V Hoang, Philip H Goodman, Sergiu M Dascalu, Bobby D Bryant, and Frederick C Harris Jr. Real-time human–robot interaction underlying neurorobotic trust and intent recognition. *Neural Networks*, 32:130–137, 2012.

[22] Laurence C Jayet Bray, Emily R Barker, Gareth B Ferneyhough, Roger V Hoang, Bobby D Bryant, Sergiu M Dascalu, and Frederick C Harris. Goal-related navigation of a neuromorphic virtual robot. *BMC Neuroscience*, 13(Suppl 1):O3, 2012.

[23] Adrienne Breland, Harpreet Singh, Omid Tutakhil, Mike Needham, Dickson Luong, Grant Hennig, Roger Hoang, Torbjorn Loken, Sergiu M Dascalu, and Frederick C Harris Jr. A GPU algorithm for comparing nucleotide histograms. *Proceedings of the 2012 ISCA International Conference on Advanced Computing and Communication*, pages 13–18, 2012.

[24] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M. Bower, Markus Diesmann, Abigail Morrison, Philip H. Goodman, Frederick C. Harris Jr., Milind Zirpe, Thomas Natschlger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P. Davison, Sami El Boustani, and Alain Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of computational neuroscience*, 23(3):349–398, 2007.

[25] David T Brown, Roger V Hoang, Matthew R Sgambati, Timothy J Brown, Sergiu M Dascalu, and Frederick C Harris, Jr. An application for tree detection using satellite imagery and vegetation data. *Journal of Computational Methods in Science and Engineering*, 10:13–25, 2010.

[26] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[27] Grigore C Burdea. Haptics issues in virtual environments. In *Computer Graphics International, 2000. Proceedings*, pages 295–302. IEEE, 2000.

[28] Anthony N Burkitt. A review of the integrate-and-fire neuron model: I. homogeneous synaptic input. *Biological cybernetics*, 95(1):1–19, 2006.

[29] Justin E Cardoza, Alexander K Jones, Denver J Liu, Roger V Hoang, Devyani Tanna, Laurence C Jayet Bray, Sergiu M Dascalu, and Frederick C Harris Jr. Design and implementation of a graphical visualization tool for NCS. In *Proceedings of the 2013 International Conference on Software Engineering and Data Engineering*, pages 37–43, 2013.

[30] Daniel C Cliburn. Virtual reality for small colleges. *Journal of Computing Sciences in Colleges*, 19(4):28–38, 2004.

[31] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[32] Microsoft Corporation. HLSL. `http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx`. Accessed January 16th, 2014.

[33] Keenan Crane, Ignacio Llamas, and Sarah Tariq. Real-time simulation and rendering of 3D fluids. *GPU Gems*, 3(1), 2007.

[34] Douglas Crockford. The application/json media type for Javascript Object Notation (JSON). `http://tools.ietf.org/html/rfc4627`, 2006. Accessed May 8th, 2014.

[35] Carolina Cruz-Neira, Daniel J Sandin, and Thomas A DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 135–142. ACM, 1993.

[36] Leonardo Dagum and Ramesh Menon. OpenMP: An industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[37] Andrew P Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: A common interface for neuronal network simulators. *Frontiers in neuroinformatics*, 2, 2008.

[38] Thomas A DeFanti, Gregory Dawe, Daniel J Sandin, Jurgen P Schulze, Peter Otto, Javier Girado, Falko Kuester, Larry Smarr, and Ramesh Rao. The StarCAVE, a third-generation CAVE and virtual reality OptIPortal. *Future Generation Computer Systems*, 25(2):169–178, 2009.

[39] André DeHon and John Wawrzynek. Reconfigurable computing: What, why, and implications for design automation. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 610–615. ACM, 1999.

[40] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364. ACM, 2010.

[41] Rich Drewes, Quan Zou, and Philip H Goodman. Brainlab: A Python toolkit to aid in the design, simulation, and analysis of spiking neural networks with the neocortical simulator. *Frontiers in neuroinformatics*, 3, 2009.

[42] Nathaniel I Durlach and Anne S Mavor. *Virtual Reality: Scientific and Technological Challenges*. National Academies Press, 1994.

[43] Erich Elsen, Vaidyanathan Vishal, Mike Houston, Vijay Pande, Pat Hanrahan, and Eric Darve. N-body simulations on GPUs. *arXiv preprint arXiv:0706.3060*, 2007.

[44] Jochen Martin Eppler, Moritz Helias, Eilif Muller, Markus Diesmann, and Marc-Oliver Gewaltig. PyNEST: A convenient interface to the NEST simulator. *Frontiers in neuroinformatics*, 2, 2008.

[45] Andreas K Fidjeland, Etienne B Roesch, Murray P Shanahan, and Wayne Luk. Nemo: A platform for neural modelling of spiking neurons using GPUs. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 137–144. IEEE, 2009.

[46] Andreas K Fidjeland and Murray P Shanahan. Accelerated simulation of spiking neural networks using GPUs. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1–8. IEEE, 2010.

[47] James Frye. Parallel optimization of a neocortical simulation program. Master's thesis, Citeseer, 2003.

[48] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[49] David Geer. Vendors upgrade their physics processing to improve gaming. *Computer*, 39(8):22–24, 2006.

[50] NVIDIA GeForce. 8800 GPU architecture overview. *Technical Brief TB-02787-001 v0*, 9, 2006.

[51] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing*, pages 128–135. Springer, 1996.

[52] Marc-Oliver Gewaltig and Markus Diesmann. NEST(neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[53] Nigel H Goddard and Greg Hood. Large-scale simulation using parallel GENESIS. In *The Book of GENESIS*, pages 349–379. Springer, 1998.

[54] Dan FM Goodman and Romain Brette. The Brian Simulator. *Frontiers in neuroscience*, 3(2):192, 2009.

[55] Simon Green. The OpenGL framebuffer object extension. In *Game Developers Conference*, volume 2005, 2005.

[56] Gold Standard Group. OpenGL Shading Language. `http://www.opengl.org/documentation/glsl/`. Accessed January 16th, 2014.

[57] Boris Gutkin, David Pinto, and Bard Ermentrout. Mathematical neuroscience: from neurons to circuits to systems. *Journal of Physiology-Paris*, 97(2):209–219, 2003.

[58] Mark Harris. GPGPU: General-purpose computation on GPUs. *SIGGRAPH 2005 GPGPU COURSE*, 2005.

[59] Reiner Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.

[60] Michael L Hines and Nicholas T Carnevale. The NEURON simulation environment. *Neural computation*, 9(6):1179–1209, 1997.

[61] Pieter Hintjens. Ømq-the guide. *Online: http://zguide.zeromq.org/page:all, Accessed on April 27, 2014*, 23, 2011.

[62] Roger Hoang, Steve Koepnick, Joseph D Mahsman, Matthew Sgambati, Cody J White, and Daniel S Coming. Exploring global illumination for virtual reality. *ACM SIGGRAPH 2010 Posters*, page 125, 2010.

[63] Roger V Hoang, Joshua Hegie, and Frederick C Harris Jr. Scrybe: A tablet interface for virtual environments. In *CAINE*, pages 105–110, 2010.

[64] Roger V Hoang, Joseph D Mahsman, David T Brown, Michael A Penick, Frederick C Harris Jr., and Timothy J Brown. Vfire: Virtual fire in realistic environments. In *Virtual Reality Conference, 2008. VR'08. IEEE*, pages 261–262. IEEE, 2008.

[65] Roger V Hoang, Matthew R Sgambati, Timothy J Brown, Daniel S Coming, and Frederick C Harris Jr. Vfire: Immersive wildfire simulation and visualization. *Computers & Graphics*, 34(6):655–664, 2010.

[66] Roger V Hoang, Devyani Tanna, Laurence C Jayet Bray, Sergiu M Dascalu, and Frederick C Harris Jr. A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling. *Frontiers in neuroinformatics*, 7, 2013.

[67] Roger Viet Hoang. Wildfire simulation on the GPU. Master's thesis, University of Nevada, Reno, 2008.

[68] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.

[69] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive kd tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174. ACM, 2007.

[70] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.

[71] Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

[72] Dana A Jacobsen, Julien C Thibault, and Inanc Senocak. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *48th AIAA Aerospace Sciences Meeting and Exhibit*, volume 16, 2010.

[73] Richard M Karp and Vijaya Ramachandran. A survey of parallel algorithms for shared-memory machines. *Handbook of Theoretical Computer Science. North-Holland*, 1988.

[74] Mark J Kilgard. The OpenGL utility toolkit (GLUT) programming interface API version 3. `http://users.informatik.uni-halle.de/~schenzel/ws02/opengl/spec3.pdf`, 1996. Accessed May 8th, 2014.

[75] David Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[76] Steven Koepnick, Roger V Hoang, Matthew R Sgambati, Daniel S Coming, Evan A Suma, and William R Sherman. RIST: Radiological immersive survey training for two simultaneous users. *Computers & Graphics*, 34(6):665–676, 2010.

[77] Eugenia M Kolasinski. Simulator sickness in virtual environments. Technical report, DTIC Document, 1995.

[78] Oliver Kreylos. Environment-independent VR development. In *Advances in Visual Computing*, pages 901–912. Springer, 2008.

[79] Lutz Lata. Building a million particle system. In *Proceedings of the Game Developers Conference 2004*, pages 54–60, 2004.

[80] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[81] JJ-W Lin, Henry Been-Lirn Duh, Donald E Parker, Habib Abi-Rached, and Thomas A Furness. Effects of field of view on presence, enjoyment, memory, and simulator sickness in a virtual environment. In *Virtual Reality, 2002. Proceedings. IEEE*, pages 164–171. IEEE, 2002.

[82] Youquan Liu, Xuehui Liu, and Enhua Wu. Real-time 3D fluid simulation on GPU with complex obstacles. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*, pages 247–256. IEEE, 2004.

[83] William R Mark, R Steven Glanville, Kurt Akeley, and Mark J Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM, 2003.

[84] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[85] Michele Migliore, C Cannia, William W Lytton, Henry Markram, and Michael L Hines. Parallel network simulations with NEURON. *Journal of computational neuroscience*, 21(2):119–129, 2006.

[86] Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeffrey L Krichmar, Alex Nicolau, and Alexander V Veidenbaum. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks*, 22(5):791–800, 2009.

[87] Tao Ni, Greg S Schmidt, Oliver G Staadt, Mark A Livingston, Robert Ball, and Richard May. A survey of large high-resolution display technologies, techniques, and applications. In *Virtual Reality Conference, 2006*, pages 223–236. IEEE, 2006.

[88] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[89] NVIDIA. GeForce GTX 780. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780/specifications. Accessed January 21st, 2014.

[90] NVIDIA. OpenGL geometry shader extension. http://developer.download.nvidia.com/opengl/specs/GL_EXT_geometry_shader4.txt. Accessed December 16th, 2014.

[91] NVIDIA. OpenGL transform feedback extension. http://developer.download.nvidia.com/opengl/specs/GL_NV_transform_feedback.txt. Accessed December 17th, 2014.

[92] NVIDIA. CUDA C best practices guide, 2013. http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf.

[93] NVIDIA. CUDA C programming guide, 2013. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[94] Iason Oikonomidis, Nikolaos Kyriazis, and Antonis A Argyros. Efficient model-based 3D tracking of hand articulations using Kinect. In *BMVC*, pages 1–11, 2011.

[95] Sohei Okamoto, Roger V Hoang, Sergiu M Dascalu, Frederick C Harris, and Noureddine Belkhatir. SUNPRISM: An approach and software tools for collaborative climate change research. In *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, pages 583–590. IEEE, 2012.

[96] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[97] Michael A Penick, Roger V Hoang, Frederick C Harris Jr, Sergiu M Dascalu, Timothy J Brown, William R Sherman, and Philip A McDonald. Managing data and computational complexity for immersive wildfire visualization. *Proceedings of High Performance Computing Systems (HPCS07)*, 2007.

[98] Jesse D. Phillips, Roger V. Hoang, Joseph D. Mahsman, Matthew R. Sgambati, Xiaolu Zhang, Sergiu M. Dascalu, and Frederick C. Harris Jr. Scripted artificially intelligent basic online tactical simulation. In *CAINE*, pages 292–297, 2008.

[99] Hans E Plesser, Jochen M Eppler, Abigail Morrison, Markus Diesmann, and Marc-Oliver Gewaltig. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In *Euro-Par 2007 parallel processing*, pages 672–681. Springer, 2007.

[100] Ivan Poupyrev, Mark Billinghurst, Suzanne Weghorst, and Tadao Ichikawa. The go-go interaction technique: Non-linear mapping for direct manipulation in vr. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 79–80. ACM, 1996.

[101] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 4(2):63–71, 1996.

[102] Mark Segal and Kurt Akeley. The design of the OpenGL graphics interface. Technical report, Silicon Graphics Computer Systems, 1994.

[103] Neal E Seymour, Anthony G Gallagher, Sanziana A Roman, Michael K OBrien, Vipin K Bansal, Dana K Andersen, and Richard M Satava. Virtual reality training improves operating room performance: results of a randomized, double-blinded study. *Annals of surgery*, 236(4):458, 2002.

[104] William R Sherman. FreeVR. `http://freevr.org/`, 2013. Accessed May 8th, 2014.

[105] William R Sherman and Alan B Craig. *Understanding virtual reality: Interface, application, and design*. Elsevier, 2002.

[106] William R Sherman, Simon Su, Philip A McDonald, Yi Mu, and Frederick C Harris Jr. Open-source tools for immersive environmental visualization. *Computer Graphics and Applications, IEEE*, 27(2):88–91, 2007.

[107] Russell D Shilling and Barbara Shinn-Cunningham. Virtual auditory displays. *Handbook of virtual environment technology*, pages 65–92, 2002.

[108] Anand Lal Shimpi. Inside the titan supercomputer: 299k AMD x86 cores and 18.6k NVIDIA GPUs. *AnandTech online computer hardware magazine, October*, 2012.

[109] Michael Showerman, Jeremy Enos, Avneesh Pant, Volodymyr Kindratenko, Craig Steffen, Robert Pennington, and Wen-mei Hwu. QP: A heterogeneous multi-accelerator cluster. In *Proc. 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[110] Michael J Smith, Roger V Hoang, Matthew R Sgambati, Sergiu Dascalu, and Frederick C Harris Jr. A dynamic multi-contextual GPU-based particle system using vector fields for particle propagation. In *CAINE*, pages 203–208, 2008.

[111] Sen Song, Kenneth D Miller, and Larry F Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, 3(9):919–926, 2000.

[112] Kay M Stanney, Ronald R Mourant, and Robert S Kennedy. Human factors issues in virtual environments: A review of the literature. *Presence: Teleoperators and Virtual Environments*, 7(4):327–351, 1998.

[113] Jonathan Steuer. Defining virtual reality: Dimensions determining telepresence. *Journal of communication*, 42(4):73–93, 1992.

[114] Richard Stoakley, Matthew J Conway, and Randy Pausch. Virtual reality on a WIM: Interactive worlds in miniature. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 265–272. ACM Press/Addison-Wesley Publishing Co., 1995.

[115] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[116] Ivan E Sutherland. The ultimate display. *Multimedia: From Wagner to virtual reality*, 1965.

[117] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[118] Russell M Taylor II, Thomas C Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T Helser. VRPN: A device-independent, network-transparent VR peripheral system. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 55–61. ACM, 2001.

[119] James D Teresco, J Fair, and Joseph E Flaherty. Resource-aware scientific computation on a heterogeneous cluster. *Computing in science & engineering*, 7(2):40–50, 2005.

[120] Corey M Thibeault, Roger V Hoang, and Frederick C Harris Jr. A novel multi-GPU neural simulator. In *Bioinformatics and Computational Biology*, pages 146–151, 2011.

[121] Corey M Thibeault, Kirill Minkovich, Michael J O'Brien, Frederick C Harris Jr, and Narayan Srinivasa. Efficiently passing messages in distributed spiking neural network simulation. *Frontiers in Computational Neuroscience*, 7, 2013.

[122] Timothy J Todman, George A Constantinides, Steven JE Wilton, Oskar Mencer, Wayne Luk, and Peter YK Cheung. Reconfigurable computing: Architectures and design methods. *IEE Proceedings-Computers and Digital Techniques*, 152(2):193–207, 2005.

[123] Stanimire Tomov, Michael McGuigan, Robert Bennett, Gordon Smith, and John Spiletic. Benchmarking and implementation of probability-based simulations on programmable graphics cards. *Computers & Graphics*, 29(1):71–80, 2005.

[124] Kuen Hung Tsoi and Wayne Luk. Axel: A heterogeneous cluster with FPGAs and GPUs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 115–124. ACM, 2010.

[125] Suresh Venkatasubramanian. The graphics card as a stream computer. In *SIGMOD-DIMACS workshop on management and processing of data streams*, volume 101, page 102, 2003.

[126] Jeffrey S Vetter, Richard Glassbrook, Jack Dongarra, Karsten Schwan, Bruce Loftis, Stephen McNally, Jeremy Meredith, James Rogers, Philip Roth, Kyle Spafford, et al. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *Computing in Science & Engineering*, pages 90–95, 2011.

[127] Oculus VR. Oculus rift-virtual reality headset for 3D gaming. `http://www.oculusvr.com`, 2014. Accessed May 8th, 2014.

[128] David W Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, 1994.

[129] Frank Weichert, Daniel Bachmann, Bartholomäus Rudak, and Denis Fisseler. Analysis of the accuracy and robustness of the leap motion controller. *Sensors (Basel, Switzerland)*, 13(5):6380, 2013.

[130] Chadwick A Wingrave, Brian Williamson, Paul D Varcholik, Jeremy Rose, Andrew Miller, Emiko Charbonneau, Jared Bott, and JJ LaViola. The wiimote and beyond: Spatially convenient devices for 3D user interfaces. *Computer Graphics and Applications, IEEE*, 30(2):71–85, 2010.

[131] Dean Wormell and Eric Foxlin. Advancements in 3D interactive devices for virtual environments. In *Proceedings of the workshop on Virtual environments 2003*, pages 47–56. ACM, 2003.

[132] Tomoya Yamada, Satoshi Yokoyama, Tomohiro Tanikawa, Koichi Hirota, and Michitaka Hirose. Wearable olfactory display: Using odor in outdoor environment. In *Virtual Reality Conference, 2006*, pages 199–206. IEEE, 2006.

[133] Yasuyuki Yanagida, Shinjiro Kawato, Haruo Noma, Nobuji Tetsutani, and Akira Tomono. A nose-tracked, personal olfactory display. In *ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1. ACM, 2003.

Appendices

Appendix A

Publications

This appendix details the various publications and projects with which the author has been involved. Though there is a bit of overlap on a few publications, they can be categorized into four particular areas: virtual reality, wildfire visualization and simulation, GPU computing, and neural simulation. Table A.1 provides these publications in chronological order along with the areas that they are categorized in.

A.1 Virtual Reality

Work in virtual reality actually began with work in wildfire visualization in the original VFire project [97], which visualized precomputed wildfire simulations in a CAVE environment using FreeVR with a shared memory computer driving the rendering. Early work focused on improving the rendering quality [64] and incorporating real-world data such as tree positions derived using supervised tools [25].

The installation of a six-sided CAVE-like environment served as impetus to the development of Hydra as the number of graphics cards required to drive up to twelve displays simultaneously made a shared-memory system impractical. VFire was eventually modified to run using Hydra [65] with run-time simulation capabilities incorporated from some interim research [67], marking its transition from solely a visualization tool to an interactive simulator. Hydra was also used for the RIST [76], a project designed to train civil support teams to perform radiological surveys in virtual environments. In order to test the performance of various global illumination

techniques in virtual reality, Hydra was also employed [62].

During this time, the author was also involved in a set of side projects, one developing a GPU-based particle system that could run in an environment with multiple rendering contexts such as the CAVE [110], and another developing Scrybe, an extension to Hydra that allowed remote devices to become interfaces for a corresponding Hydra application [63]. SUNPRISM [95] is a project that employs Hydra in order to visualize climate change data in virtual environments such as the CAVE.

A.2 Wildfire Visualization and Simulation

The VFire project originally attempted to provide realistic visualizations of wildfires in a CAVE [97, 64]. Tree positioning using a developed computer-vision tool was later added [25]. VFire's sole purpose as a visualization tool that could only show data that took too long to process made it less effective as a tool for experimentation and real-time use; as such, research was done to create a wildfire simulator that could execute on the GPU; the advantage of this approach was the exploitation of the GPU's data parallellism as well as the instant availability of that data for rendering afterwards [67], a system that would be later incorporated into the newly Hydra-driven VFire [65], which provided interactive tools to users to allow them to start fires, manipulate weather conditions, and test the effects of various countermeasures such as fire breaks.

A.3 GPU Computing

The majority of publications discussed in this appendix are related in some way to GPU computing. In addition to the previously described GPU implementations of a fire simulator [67] and a particle system [110], a GPU algorithm was developed for comparing nucleotide sequences was also developed using CUDA [23]. The global illumination application also employed some GPU ray-tracing as a component of one of its algorithms [62]. The author's involvement in all of the publications discussed

in the next section about neural simulation can be attributed to some form of GPU computing used in each one.

A.4 Neural Simulation

The version of NCS6 [66] described in Chapter 4 was the capstone to a body of work in neural simulation. The idea for using bit vectors as the primary communication message between devices was originally developed in a prototype Izhikevich simulator that could run on multiple GPUs [120]. A visualization application for the output of NCS6 was developed by another group of students [29].

Before work began on NCS6, a CUDA implementation of a gabor filter was developed to assist with research using NCS5, particularly in modeling trust and the neurotransmitter oxytocin [7, 21]. Additionally, it was used in the simulation of a brain architecture that could successfully navigate virtual environments [22].

Table A.1: Publications sorted chronologically with subject areas.

Publication	**Virtual Reality**	**Wildfire Simulation**	**GPU Computing**	**Neural Simulation**
Managing data and computational complexity for immersive wildfire visualization[97]	X	X		
VFire: virtual fire in realistic environments [64]	X	X		
Wildfire simulation on the GPU [67]		X	X	
A dynamic multi-contextual GPU-based particle system using vector fields for particle propagation [110]	X		X	
Scripted Artificially Intelligent Basic Online Tactical Simulation [98]	X			
Scrybe: a tablet interface for virtual environments [63]	X			
An application for tree detection using satellite imagery and vegetation data [25]		X		
Exploring global illumination for virtual reality [62]	X		X	
VFire: Immersive wildfire simulation and visualization [65]	X	X	X	
RIST: Radiological immersive survey training for two simultaneous users [76]	X			
Modeling oxytocin induced neurorobotic trust and intent recognition in human-robot interaction [7]			X	X
A novel multi-GPU neural simulator[120]			X	X
Real-time humanrobot interaction underlying neurorobotic trust and intent recognition [21]			X	X
Goal-related navigation of a neuromorphic virtual robot [22]			X	X
A GPU algorithm for comparing nucleotide histograms [23]			X	
SUNPRISM: An approach and software tools for collaborative climate change research [95]	X			
Design and implementation of a graphical visualization tool for NCS [29]				X
A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling [66]			X	X