# Abstractions for Code Migration from CPU to GPU in Simulation Domain

Avikant Wadhwa

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2025

Committee:

Michael Stiber

Munehiro Fukuda

Annuska Zolyomi

Program Authorized to Offer Degree:
Computing and Software Systems

University of Washington

**Abstract**

Abstractions for Code Migration from CPU to GPU in Simulation Domain

Avikant Wadhwa

Chair of the Supervisory Committee:
Michael Stiber
Department of Computing and Software Systems

Simulations are crucial in science, enabling the modeling of complex phenomena that are difficult to study experimentally. As they scale, they demand greater performance and efficiency. To meet this need, computing has shifted toward heterogeneous architectures that combine CPUs and GPUs. While effective, this shift introduces software engineering challenges, making abstraction an increasingly important tool for improving programmability. Abstractions hide low-level implementation details behind clean interfaces, improving clarity and reducing complexity.

This thesis reviews existing abstractions for heterogeneous architectures, analyzing their integration effort, performance trade-offs, and limitations. It uses the insights from that review to present the design and implementation of DeviceVector, a lightweight abstraction that unifies host and device memory management in Graphitti, a high-performance graph-based simulation platform. DeviceVector enhances programmability by reducing code duplication, introducing a clear CPU–GPU data relationship, and abstracting CUDA boilerplate through an interface that closely mirrors a standard C++ container. It also discusses design approaches for extending support in the future to object hierarchies and general function-level abstractions, further minimizing logic duplication between host and device code. Overall, this work highlights how thoughtful abstraction design can bridge the usability-performance gap in heterogeneous computing systems.

# TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

Table Number Page

# LIST OF CODE SECTIONS

# GLOSSARY

SIMULATION: Simulation involves creating and analyzing virtual models of real-world systems to understand their behavior and optimize outcomes. It allows for experimentation and analysis without the constraints and risks associated with physical experiments.

Host/CPU: Refers to the central processing unit (CPU) of a computer system, which is used as a compute device and also coordinates with external devices, including GPUs. In GPU programming, the host is responsible for initiating data transfers and kernel launches on the GPU.

Device/GPU: Refers to the Graphics Processing Unit (GPU), which is used as a compute device for accelerating parallel workloads. In GPU programming, the device executes kernels and operates on data in its own memory space.

CUDA: Compute Unified Device Architecture (CUDA) is NVIDIA's parallel computing platform and programming model that allows developers to use NVIDIA GPUs for general-purpose computing. It accelerates processing tasks by enabling high-performance parallel computation.

## ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my chair, Dr. Michael Stiber, for his unwavering support, patience, and insightful guidance throughout this research. His enthusiasm and deep knowledge were instrumental in shaping both the direction of this research and my growth as a researcher. It has been a privilege to work under his mentorship. I would also like to thank my committee members, Dr. Munehiro Fukuda and Dr. Annuska Zolyomi, for their valuable time, thoughtful feedback, and encouraging support during the course of this work.

I am grateful to my fellow members of the Intelligent Networks Laboratory for their contributions to the development and maintenance of Graphitti, which played a foundational role in this research. Lastly, I would like to acknowledge my parents, family, friends, and faculty, who have supported me in various ways throughout this academic journey.

# Chapter 1

# INTRODUCTION

Simulations play a central role in scientific research, engineering, and industrial applications. By enabling the analysis, modeling, and forecasting of complex physical and computational systems, simulations facilitate the investigation of phenomena that are otherwise impractical or infeasible to study empirically. Applications span a wide range of domains, including neural simulation [1], climate modeling, high-fidelity physics-based computation [2], and large-scale graph analytics [3]. These simulation workloads are often computationally intensive, requiring the resolution of high-dimensional numerical systems or the orchestration of interactions among a large number of agents or components. Consequently, as simulations grow in scale and fidelity, they impose stringent demands on computational throughput, increasing performance and efficiency to remain practical and responsive.

To address these growing performance requirements, modern computing has transitioned toward heterogeneous architectures, which integrate general-purpose CPUs with high-throughput GPUs within a unified system [4][5]. This shift to a heterogeneous architectural paradigm leverages the strengths of both processor types: CPUs offer efficient control flow and complex branching due to their rich instruction sets and hierarchical caches, while GPUs are optimized for high-throughput handling massive parallel workloads due to their many-core architecture execution model. By distributing computational tasks appropriately between CPU and GPU, simulation frameworks can achieve significant speedups and scale to previously infeasible problem sizes. One of the most influential tools enabling this transition is CUDA (Compute Unified Device Architecture), introduced by NVIDIA in 2006 [6]. CUDA is a parallel computing platform and programming model that provides an interface that allows developers to write general-purpose code for GPUs in a familiar C/C++ syntax, significantly lowering the barrier to entry for scientific and high-performance computing applications. CUDA's impact on scientific domains such as molecular dynamics

(e.g., AMBER, LAMMPS), neural modeling (e.g., NEST, Arbor) and real-time physical simulations is well documented [7].

However, GPU programming is inherently different from traditional CPU programming and introduces a unique set of challenges. Unlike CPU code, which operates under a single-threaded or multithreaded model, GPU code follows a parallel execution model, separate memory management, and implementation approach that diverges significantly from traditional software design. Tasks that are straightforward in CPU programming, such as using recursive data structures, iterative processing, dynamic memory allocation, or maintaining complex object hierarchies, often need to be carefully refactored or redesigned for performance and correctness when implemented for GPU execution [8]. GPU kernels operate across thousands of threads, which must be carefully synchronized and optimized to prevent bottlenecks due to memory access patterns, thread divergence, or bank conflicts. Additionally, explicit memory management between the host and the device introduces additional complexity, as data must be explicitly transferred, incurring performance costs if not optimized [9]. As a result, migrating code from CPU to GPU or maintaining a hybrid CPU-GPU codebase often becomes complex and error-prone, reducing overall programmability.

This growing complexity in GPU programming has prompted increased interest in abstraction mechanisms as one of the ways to improve programmability—enabling developers to manage performance while maintaining code readability, modularity, and correctness. Abstractions help by hiding low-level implementation details behind intuitive interfaces, allowing domain experts to focus on algorithm design rather than hardware intricacies [10]. In the context of CUDA development, abstractions can encapsulate device memory allocation, streamline kernel invocation patterns, and provide tools for safe synchronization. It can reduce boilerplate code, enforce compile-time constraints, and ensure performance portability across different hardware configurations [11].

### *1.1 Research Motivation*

To address the complexity of developing heterogeneous applications, a variety of libraries and frameworks have emerged that offer high-level abstractions for GPU programming. These

tools typically provide portable data structures and interfaces that encapsulate hardware-specific details and manage low-level API calls internally. By abstracting memory management, algorithmic patterns, and parallel execution models, they aim to make GPU development more accessible and less error-prone. Some advanced solutions even enable seamless integration across host and device environments, supporting code reuse and improving developer productivity.

While high-level abstractions simplify GPU development, they often introduce challenges in integration, maintenance, and performance. These tools evolve independently of application code, leading to compatibility issues, increased testing complexity, and limited extensibility. Their general-purpose nature can incur performance overhead compared to hand-optimized solutions, making them less suitable for high-performance or domain-specific needs. Moreover, there is a lack of design guidance for these tools, with limited discussion on categorizing them by the problems they address—such as memory management or kernel launch—and the trade-offs involved. This gap makes it difficult for developers to design or implement suitable abstractions.

The goal of this thesis is to design abstractions specifically targeted to Graphitti, a high-performance graph-based heterogeneous simulation platform. To achieve this, the thesis reviews existing abstractions and categorizes them to analyze how different types of abstractions address various aspects of heterogeneous application development. It uses insights from that review to present the design and implementation of DeviceVector, a lightweight abstraction that unifies host and device memory management in Graphitti. This work lays out the design approach and choices behind DeviceVector, which is designed to improve the programmability of Graphitti so that it can be easily extended by researchers to implement additional simulation domains. Rather than offering yet another general-purpose library, the intention is to provide a starting point or offer insights and guidance for designing targeted, maintainable abstractions.

### *1.2 Overview*

Chapter 2 describes Graphitti's architecture, design, and motivation from the perspective of heterogeneous application development. Chapter 3 presents a literature review of exist-

ing abstractions for heterogeneous development, highlighting their core features. It briefly outlines their integration ease, performance, compatibility, offer to developers—alongside their limitations.

Chapter 4 categorizes different abstraction mechanisms aimed at simplifying and optimizing the development of heterogeneous applications. It then describes the methodology behind the design and implementation of the DeviceVector class—an abstraction that streamlines memory management and GPU data access in the Graphitti. The chapter also outlines key design decisions and integration steps.

Chapter 5 presents an evaluation of the DeviceVector integration into Graphitti, demonstrating how it reduces development complexity and addresses existing programming challenges. Lastly, Chapter 6 summarizes the key findings of this study and outlines potential directions for future work.

# Chapter 2

# BACKGROUND: HETEROGENEOUS DEVELOPMENT OF GRAPHITTI

Graphitti is a high-performance graph-based simulation platform that aims to aid scientists and researchers by providing pre-built code that can easily be modified to fit different simulation models. It is derived from the architectural principles of BrainGrid, a neural network simulator designed to model large-scale neuronal dynamics. Neural network simulators provide a complementary computational approach for investigating neuronal population dynamics. They enable large-scale modeling and analysis using hybrid CPU–GPU infrastructures.

The BrainGrid simulator was capable of modeling neural networks with varying sizes, numbers of synapses, and simulation durations etc., depending on the experimental setup. In one example configuration, it modeled a network of 10,000 neurons arranged in a $100 \times 100$ grid, with up to 500,000 synapses forming dynamically over the course of the simulation. Executed across 600 million time steps—each representing a 0.1 millisecond interval—this fine-grained temporal resolution enabled BrainGrid to capture emergent behaviors and self-organizing dynamics characteristic of biologically inspired systems. Graphitti generalizes the BrainGrid architecture to support a broader class of graph-based simulations beyond biologically motivated networks. It introduces an additional abstraction layer that facilitates the modeling of diverse real-world systems as complex, self-organizing graphs. In this framework, vertices represent entities with state, while edges facilitate communication and influence among them and can also possess internal state [12]. The system supports both directed and weighted graphs and enables mixed continuous/discrete simulation, in which vertex and edge states and outputs can be determined by both discrete message passing and the integration of differential equations as well.

Implemented in C++ and CUDA C++, Graphitti features a modular design compris-

ing both CPU and GPU execution pathways. A notable aspect of its implementation is that the CPU-only version does not require the NVIDIA CUDA compiler (nvcc); it can be compiled using the standard g++ compiler found on most host systems. This enables users to develop and test serial simulations using standard C++, and then seamlessly migrate to parallel GPU execution with minimal modifications [12]. The framework provides clearly defined CUDA integration steps, making it accessible to researchers seeking high-performance simulations of large-scale, graph-based systems.

All subsystems in Graphitti follow a graph-based modeling abstraction, enabling flexible initialization of objects and their associated behaviors. While the next section focuses on the architecture of the Neural Vertices class for managing both CPU and GPU operations, this structural pattern is consistently applied across other subsystems as well.

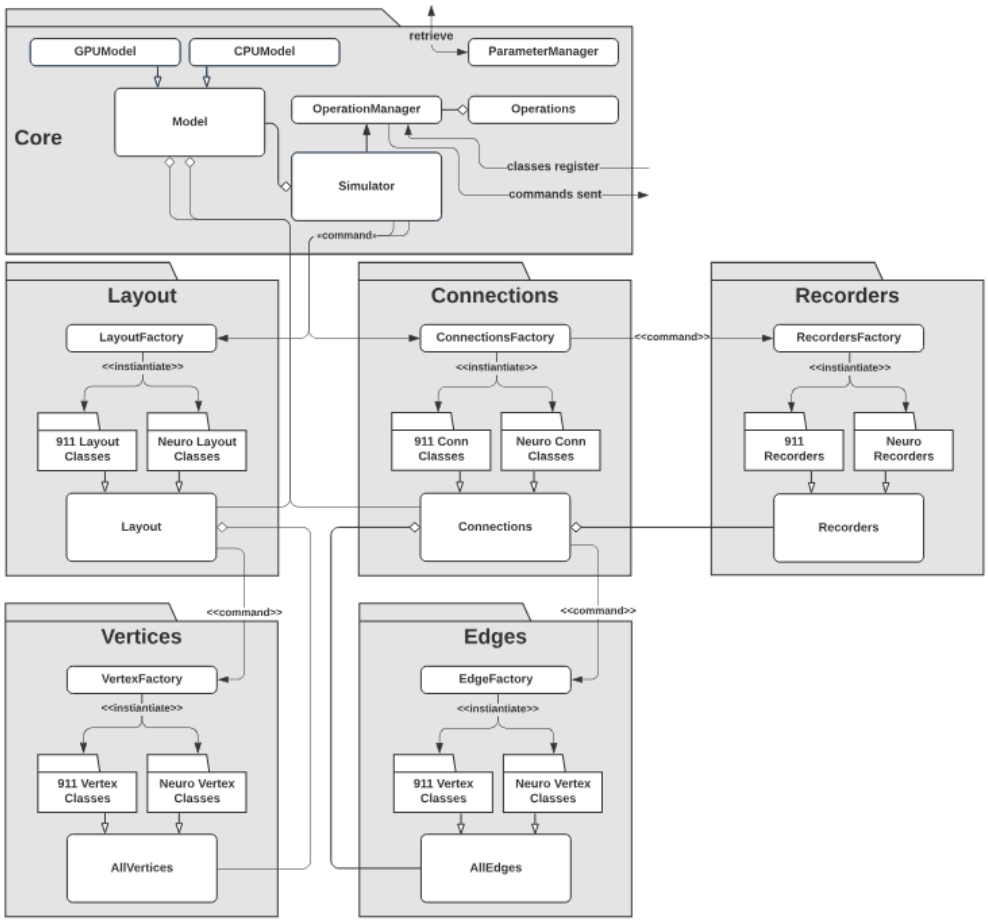


Figure 2.1: Graphitti UML diagram showing the six main subsystems: Core, Layouts, Connections, Vertices, Edges, and Recorders and their respective subsystems, reused from [13]

### *2.1 Host and Device Separation for Entity: Vertices*

The vertices subsystem encapsulates the data structures, interface classes, and subclasses related to vertex management. All vertex-related objects are initialized prior to the start of the simulation. Based on the simulation type specified in the configuration file—such as Growth, STDP, or NG911—Graphitti uses polymorphism to initialize the appropriate vertex classes [14]. As illustrated in Figure 2.1, the AllSpikingNeurons class serves as the base class for all neural implementations. While this section focuses on the neural do-

main—specifically the structure of AllSpikingNeurons—a similar design approach is applied across the implementation of other classes within Graphitti that involve GPU code.

### *2.1.1 AllSpikingNeurons.h file*

On the host side, the `AllSpikingNeurons` class uses standard C++ containers such as `std::vector<bool> hasFired` (which manages per-neuron state). These data members exist in both host and device builds, as data is initialized on the host, copied to the GPU for simulation, and later transferred back to the host for persistence (e.g., writing to files). Some functions—such as `setupVertices()` (which initializes the vertices with initial data)—are declared without any conditional compilation since they are build-independent. In contrast, build-specific functions are conditionally compiled using the `USE_GPU` macro. For example, `advanceVertices()` (which updates the state of all vertices for one simulation time step) is excluded from device-side compilation, while `clearDeviceSpikeCounts()` (which clears the spike counts for all neurons) is excluded from host-side compilation.

Listing 2.1: AllSpikingNeurons class in AllSpikingNeurons.h file

```
class AllSpikingNeurons : public AllVertices {
public:
    void setupVertices();
    ...
#if defined(USE_GPU)
public:
    clearDeviceSpikeCounts(...);
    ...
#else
public:
    advanceVertices(...);
    ...
#endif
private:
    vector<bool> hasFired_;
    ...
}
```

For the device side, the same file defines an `AllSpikingNeuronsDeviceProperties` struct, which mirrors the data layout of the host but uses raw device pointers such as `bool* hasFired` to manage memory on the GPU. This structure is conditionally compiled using the `USE_GPU` macro and is excluded when compiling for the host, thereby ensuring a clear separation between host and device data.

Listing 2.2: Struct for GPU data in AllSpikingNeurons.h file

```
#if defined(USE_GPU)
struct AllSpikingNeuronsDeviceProperties : public AllVerticesDeviceProperties {
    bool *hasFired_;
    ...
};
#endif
```

### *2.1.2 AllSpikingNeurons.cpp file*

This file contains all the member function definitions of the `AllSpikingNeurons` class and uses the `USE_GPU` preprocessor directive to exclude host-only implementations during GPU builds. This separation ensures that host-specific functionality remains isolated and does not interfere with device-side compilation.

Listing 2.3: Methods in AllSpikingNeurons.cpp file

```
void AllSpikingNeurons::setupVertices() {
....
}
#if !defined(USE_GPU)
void AllSpikingNeurons::advanceVertices(....) {
...
}
#endif
```

### *2.1.3 AllSpikingNeurons_d.cpp file*

This file contains all the device-side logic, including CUDA kernels and device-only functions. It is responsible for defining all GPU-specific operations, such as the `calcSummationPointDevice` kernel (which adds psr of all incoming synapses to summation points) and other device-level functionality associated with this class. The file is compiled using `nvcc` and is excluded from host-only builds via CMake to prevent incompatibility issues. This separation ensures that the CPU implementation remains lightweight and free of CUDA dependencies when targeting non-GPU platforms. By isolating CUDA code in device-specific files and using conditional compilation, the architecture remains modular and portable, thereby simplifying both development and testing.

Listing 2.4: kernels and device methods in AllSpikingNeurons_d.cpp file

```
//declaration
__global__ void calcSummationPointDevice(int totalVertices,..);
...
//definition
__global__ void calcSummationPointDevice (...) {
...
}
void AllSpikingNeurons::integrateVertexInputs (..) {
    //actual call from the host
    calcSummationPointDevice<<<blocksPerGrid, threadsPerBlock>>>(...);
}
```

# Chapter 3

# EXISTING ABSTRACTIONS FOR HETEROGENEOUS COMPUTING

The increasing prevalence of heterogeneous computing systems, combining general-purpose CPUs with specialized accelerators such as GPUs, has significantly impacted software development practices. Programming models like CUDA and OpenCL enable explicit GPU utilization but often impose considerable complexity due to manual memory management, kernel invocation details, and device synchronization requirements [6]. Consequently, various abstraction frameworks have emerged to simplify heterogeneous programming and enhance productivity while preserving performance and portability.

Directive-based abstractions like OpenACC simplify GPU programming by using compiler pragmas to offload computations and manage data movement automatically [15]. While OpenACC can achieve performance comparable to low-level CUDA programming with minimal programming effort in certain scenarios [16], its reliance on compiler optimizations may limit fine-grained control and effectiveness in irregular workloads. Algorithmic skeleton frameworks, including SkelCL and SkePU, simplify GPU programming by offering predefined parallel patterns such as map, reduce, and scan, combined with automated memory management and multi-GPU support [17] [18]. By abstracting low-level GPU details, these libraries significantly enhance developer productivity, enabling concise and efficient parallel implementations. They have successfully demonstrated competitive performance in applications like medical imaging, numerical simulations, and data-intensive computations [18]. However, their applicability is typically restricted to computations that naturally align with provided skeleton patterns. C++ template-based libraries such as Kokkos and RAJA offer portable parallelism by abstracting execution patterns via templates, allowing single-source code to compile across multiple backends (CUDA, HIP, OpenMP) [19] [20]. Both frameworks have successfully demonstrated significant productivity gains and high performance in large-scale scientific applications [21]. However, their template-heavy programming style

introduces complexity, steep learning curves, and potential difficulty in debugging and maintaining codebases, particularly for developers unfamiliar with advanced C++ concepts.

Thrust simplifies GPU programming with an STL-like interface for common parallel algorithms, such as sorting, reductions, and transformations, lowering the barrier to GPU acceleration [22]. However, it primarily supports NVIDIA GPUs, limiting cross-platform portability, and its high-level approach may obscure low-level optimizations. Cross-platform extensions like HIP and SYCL enhance portability across architectures. HIP offers source compatibility with CUDA for AMD GPUs (Liu et al., 2016), while SYCL supports heterogeneous execution on CPUs, GPUs, and FPGAs from standard C++ [23]. Nevertheless, HIP remains limited mostly to AMD/NVIDIA hardware, and SYCL's runtime may introduce performance overhead. The Alpaka framework provides backend-agnostic parallelism through C++ templates, enabling unified coding across accelerators like CUDA, HIP, and OpenMP [24]. Alpaka emphasizes performance portability and control but introduces complexity, a steep learning curve, and increased compilation times due to extensive template use.

NVIDIA's NVC++ compiler abstracts GPU programming by automatically offloading standard C++17 parallel algorithms (e.g., std::reduce) to GPUs, simplifying development and achieving strong performance with minimal code changes. However, its implicit offloading may hinder precise performance tuning and efficient memory management. FastFlow supports efficient CPU-GPU orchestration through pipeline-based parallel patterns suited to streaming applications like multimedia processing and financial analytics [25]. Despite its strengths in structured workflows, FastFlow requires explicit pipeline structuring, limiting its applicability to irregular or less structured parallelism. PHAST provides single-source, STL-inspired C++ abstractions, enabling portable parallel programming across CPUs and GPUs, demonstrated effectively in cryptography and machine learning domains [26]. However, PHAST's abstraction may constrain fine-grained optimizations and flexibility in irregular computational scenarios.

While high-level abstraction frameworks have made heterogeneous computing more accessible by simplifying parallelism, memory management, and hardware targeting, their reliance on external libraries introduces a host of development and maintenance challenges.

These tools often follow independent lifecycles, with updates and feature changes outside the control of the application developer. This can lead to integration issues, unexpected behavior, and long-term maintainability concerns, especially in large or tightly coupled systems. Testing becomes more complex due to dependency variability, and reproducing behavior across versions may be difficult. Moreover, these libraries are rarely accompanied by guidance on designing or extending abstractions, making it harder for developers to adapt them to evolving requirements. They may fail to integrate into systems that are designed to address novel or specific problems, which can have critical implications for software architectural decisions.

Furthermore, the general-purpose nature of these tools often introduces performance overhead compared to specifically targeted and optimized implementations, making them less suitable for systems where maximum efficiency and fine-grained control are essential. A notable example is Graphitti, where integrating such abstractions may not only require substantial effort—potentially necessitating a complete architectural rewrite—but also result in reduced control and suboptimal performance compared to the existing low-level CUDA implementation. As a result, while intended to reduce effort, such abstractions can inadvertently increase development complexity and compromise the long-term adaptability of software in domain-specific or high-performance computing environments.

# Chapter 4

# METHODOLOGY

## *4.1 Abstraction Categorization*

This section categorizes various abstraction mechanisms designed to simplify and optimize the development of heterogeneous applications using CUDA. The objective is to analyze how different types of abstractions either support or hinder development in systems that integrate both CPU and GPU components, and to identify the specific problems they aim to solve. For each abstraction category, the section outlines its purpose and benefits, and illustrates its practical use by referencing an existing library or framework that exhibits the corresponding characteristics. As a result, a single abstraction may appear in multiple categories if it embodies features relevant to each. Insights from this can help in designing abstractions targeted at specific problem areas.

- **Memory Management Abstractions**

  **Purpose:** Simplify and automate memory allocation, deallocation, and data transfers between host (CPU) and device (GPU).

  **Description:** In native CUDA or similar frameworks, memory operations must be explicitly managed using routines such as `cudaMalloc`, `cudaMemcpy`, and `cudaFree`. These low-level operations introduce verbosity and distract from the core computational logic. Abstractions such as memory managers, buffer wrappers, or container-based memory holders encapsulate low-level memory operations, simplifying development by providing clean interfaces for allocation, transfer, and deallocation. While these abstractions may support lifecycle management through destructors, enable deep-copy semantics, and integrate with unified memory models, their primary purpose is to offer a structured way to hide implementation details—without completely relinquishing control over memory behavior.

Listing 4.1: Example of Memory Management Abstraction

```
DeviceVector<float> d_vec(100);
d_vec.copyToDevice(); // Abstracts cudaMalloc and cudaMemcpy
```

**Representative Tools:** `DeviceVector`, `Kokkos::View`, `thrust::device_vector`, `PHAST::gpu_vector`.

**Benefits:** Reduces boilerplate code, improves code manageability, may provide safer memory access patterns, and is particularly beneficial for large data structures or frequent memory transfers.

- **Execution and Kernel Launch Abstractions**

  **Purpose:** Simplify the process of configuring and launching kernels by abstracting thread block/grid setup and execution control.

  **Description:** Kernel launch configuration using the traditional `<<<gridDim, blockDim>>>` syntax is replaced with abstractions like `parallel_for`, lambda-based dispatch, or policy-driven execution. These interfaces allow developers to define computation declaratively, separating algorithm logic from hardware configuration. Libraries like Kokkos and RAJA provide structured interfaces where kernel behavior is parameterized by policy.

Listing 4.2: Example of Execution and Kernel Launch Abstraction

```
Kokkos::parallel_for("scale", 100, KOKKOS_LAMBDA(int i) {
    data[i] *= 2.0f;
});
```

**Representative Tools:** `Kokkos::parallel_for`, `RAJA::forall`, custom `launchKernel()` templates.

**Benefits:** Increases readability, reduces launch-time errors. These abstractions improve code maintainability and reduce the likelihood of misconfigurations related to launch parameters.

- **Data and Container Abstractions**

  **Purpose:** Provide high-level data structures that abstract GPU memory access and layout using STL-like interfaces.

  **Description:** Containers such as vectors, matrices, or multidimensional views abstract raw device pointers and promote safer memory access by enforcing structured access patterns through familiar interfaces (e.g., iterators, accessors). These GPU-aware abstractions often manage memory implicitly, reduce the likelihood of errors such as invalid access, and support parallel algorithms (e.g., map, reduce, transform), contributing to more robust and maintainable code.

Listing 4.3: Example of Data and Container Abstraction

```
thrust::device_vector<int> vec(100, 1);
thrust::transform(vec.begin(), vec.end(), vec.begin(),
                  [] __device__ (int x) { return x * 3; });
```

  **Representative Tools:** `thrust::device_vector`, `Kokkos::View`, `PHAST::gpu_matrix`.

  **Benefits:** Provides safer data handling, integrates with parallel algorithms, and enables implicit memory optimization (e.g., coalesced access, alignment). These abstractions allow developers to manipulate data using familiar paradigms without dealing directly with device pointers or raw memory.

- **Execution Policy Abstractions**

  **Purpose:** Decouple algorithm logic from execution hardware by parameterizing parallel execution strategies.

  **Description:** Execution policies allow the same algorithm to run on different hardware (e.g., CPU sequential, OpenMP, CUDA) simply by changing the policy parameter without modifying the logic itself. This technique aligns with performance-portability goals and allows for architecture-aware tuning.

Listing 4.4: Example of Execution Policy Abstraction

```
RAJA::forall<RAJA::cuda_exec<256>>(RAJA::RangeSegment(0, N), [=] __device__ (
    int i) {
    output[i] = input[i] * 4;
});
```

**Representative Tools:** `RAJA::forall`, `Kokkos::RangePolicy`, execution traits in Alpaka/SYCL.

**Benefits:** Encourages reuse of algorithms across backends, minimizes platform-specific code branches, and supports fine-grained tuning.

- **Single-Source Programming Model**

  **Purpose:** Enable writing a unified codebase that runs on both CPUs and GPUs using shared function definitions and compilation paths.

  **Description:** Using qualifiers like `__host__ __device__` (in CUDA), developers write functions that can be executed on both CPU and GPU. This model is further enhanced in frameworks like SYCL or `stdpar` with device-agnostic lambdas. Single-source models improve code maintainability and allow unified logic flow across architectures.

Listing 4.5: Example of SingleSource Programming Model

```
__host__ __device__ float square(float x) {
    return x * x;
}
```

**Representative Tools:** `CUDA dual qualifiers`, `SYCL device lambdas.`

**Benefits:** Reduces code duplication, supports conditional compilation, and promotes architectural symmetry.

- **Compiler-Based and Directive-Based Abstractions**

  **Purpose:** Allow incremental GPU parallelism via compiler hints without requiring full kernel rewrites.

**Description:** Compiler- or directive-based models such as OpenACC use directives to annotate loops or code regions for parallel execution, allowing the compiler to automatically generate kernels targeting CUDA or similar frameworks. This model is well-suited for large, monolithic legacy codebases where gradual GPU enablement is preferred, as it requires minimal modifications to the existing code structure.

Listing 4.6: Example of Compiler-Based and Directive-Based Abstraction

```
#pragma acc parallel loop
for (int i = 0; i < N; ++i) {
    data[i] *= 2;
}
```

**Representative Tools:** `OpenACC`, `PGI/NVHPC`, `Intel oneAPI DPC++ extensions`.

**Benefits:** Low entry barrier for GPU adoption, promotes rapid prototyping, and preserves serial logic.

- **Backend-Agnostic Abstractions**

**Purpose:** Provide a unified interface for deploying to CUDA, HIP, OpenCL, SYCL, or other accelerators without source-level changes.

**Description:** TThese frameworks abstract hardware-specific APIs behind a common runtime by relying on backend mapping and toolchain support to target the appropriate hardware. Using configuration-based dispatch, they select the correct implementation at compile time or runtime based on the compiler and platform.

Listing 4.7: Example of Cross-Platform and Backend-Agnostic Abstraction

```
q.submit([&](sycl::handler& h) {
    h.parallel_for(N, [=](int i) {
        data[i] += 10;
    });
});
```

**Representative Tools:**

`SYCL (DPC++)`, `Alpaka`, `Kokkos`, `HIP`, `OneAPI`, `OpenCL`.

**Benefits:** Codebase unification, vendor independence, and increased code reuse across systems (NVIDIA, AMD, Intel etc.).

- **Template and Metaprogramming Abstractions**

  **Purpose:** Use C++ templates and compile-time techniques to specialize code for host and device execution without duplication.

  **Description:** These abstractions use C++ metaprogramming techniques to generate specialized code paths for CPU and GPU execution. By resolving the execution context at compile time, they enable performance-portable designs and are often structured as header-only or lightweight template-based libraries.

Listing 4.8: Example of Template and Metaprogramming Abstraction

```
template<typename ExecSpace>
void compute(ExecSpace&& exec) {
    if constexpr (is_gpu_exec<ExecSpace>) {
        launch_gpu_kernel<<<...>>>(...);
    } else {
        run_cpu_loop();
    }
}
```

**Representative Tools:** `__CUDA_ARCH__` macro for device-specific compilation or branching, Kuricheti's CUDA metaprogramming constructs [27].

**Benefits:** Enables compile-time branching, unifies host/device logic, and maximizes code reuse in templated libraries.

### *4.2 Device Vector*

This section presents the methodology behind the design and implementation of the DeviceVector class—an abstraction we developed to streamline memory management and GPU data access in Graphitti. It falls into the hybrid category of memory management and data & container abstractions, as the primary motivation behind this design is to simplify GPU memory handling in Graphitti. As explained in Chapter 2, Graphitti currently relies on verbose and difficult-to-maintain code structures to manage device and host memory separately, making it difficult to maintain and limiting overall code manageability in its heterogeneous architecture. The design of DeviceVector can serve as a foundation for other heterogeneous simulation frameworks, particularly those used in large-scale graph-based and neural simulations. Such simulations may involve frequent and precise memory operations—such as copying data between host and device at each epoch—where the accuracy of state transitions is critical. Proper memory handling ensures that simulation results remain valid and interpretable, which is essential for analyzing, understanding, and drawing meaningful conclusions from the generated data. In this context, data movement becomes a core part of the simulation logic and must be handled in a controlled and explicit manner by the developer. This chapter also outlines the key design decisions and integration steps involved in the development of DeviceVector.

The `DeviceVector` class encapsulates core responsibilities such as device memory allocation, host-device data transfer, and safe GPU memory access, all within a reusable, STL-compatible container interface. By abstracting low-level CUDA operations (e.g., `cudaMalloc`, `cudaMemcpy`, `cudaFree`), it reduces boilerplate code and developer burden, promoting a higher-level programming model that mirrors standard CPU-side container usage. This unified interface streamlines memory management across CPU and GPU, minimizes code duplication, and enhances code maintainability by adopting familiar host-side development patterns within GPU contexts.

Crucially, DeviceVector also reinforces some degree of logical relationship between CPU and GPU data. Although physically separated in memory, host and device data often correspond to the same logical entity within a simulation (e.g.,weight vector of a vertex).

Maintaining this correspondence is essential for program correctness, especially in hybrid CPU-GPU execution contexts where some computations may be offloaded to the GPU and later synchronized back with the CPU. The abstraction ensures that these logically linked data structures can be synchronized easily and maintain consistent meaning throughout the simulation. Technically it offer a clean separation between host and device data internals representations—while maintaining compatibility with raw CUDA device pointers and kernel calls.

From an architectural standpoint, the abstraction follows general software engineering principles such as modularity, encapsulation, and extensibility. It is implemented using templates to support generic types and can be extended or specialized for complex data layouts. While broadly applicable to various CUDA applications, the design of DeviceVector is specifically tailored with the requirements of Graphitti in mind, ensuring compatibility with its simulation kernels and dataflow mechanisms.

### *4.2.1 Design Approach*

The `DeviceVector<T>` class was implemented using C++ templates to support arbitrary data types. The class follows a composition-based design, internally managing:

- `std::vector<T>` object for host-side storage.
- raw `T*` pointer (`d_ptr_`) representing device memory.

This approach allows DeviceVector to support all STL vector-like operations for host-side logic, while explicitly providing GPU memory operations via member functions explained below. All these function must be explicitly called by the developer, so it is the caller's responsibility to manage the memory lifecycle in coordination. By requiring explicit allocation, the design provides developers with precise control over GPU memory management, which is essential for performance tuning and avoiding unnecessary overhead in high-performance simulation workloads.

- `allocateDeviceMemory()` - This function is responsible for allocating memory on the GPU corresponding to the current size of the host container. Internally, it invokes

`cudaMalloc()` to allocate a contiguous block of device memory large enough to store all elements held in the host-side vector. The size of the allocation is derived from the number of elements in the vector and the size of the type `T`. It ensures that the device memory is properly allocated and ready to be used in CUDA kernels.

- `copyToDevice()` - This function transfers data from the host-side container to the previously allocated device memory. Internally, it uses `cudaMemcpy()` with the `cudaMemcpyHostToDevice` flag to perform the memory copy. This function assumes that device memory has already been allocated via `allocateDeviceMemory()` and that the size of the host container matches the expected size. This explicit transfer operation gives developers control over when and how data is synchronized between host and device and avoids hidden or implicit memory transfers that can degrade performance or obscure correctness, which is particularly important in large-scale simulations or tightly-tuned GPU applications.

- `copyToHost()` - It performs the reverse of `copyToDevice()`—it transfers data from the device memory back into the host-side container. It uses `cudaMemcpy()` with the `cudaMemcpyDeviceToHost` flag and assumes that the device memory has been allocated and populated with valid data. This function is typically called after CUDA kernel execution when results or updated data need to be retrieved for further processing on the CPU.

- `freeDeviceMemory()` - It deallocates memory on the GPU previously allocated with `allocateDeviceMemory()`. It internally calls `cudaFree()` on the device pointer and resets the internal device memory state to prevent accidental reuse or access. This function must be called manually to avoid GPU memory leaks, particularly in long-running simulations or iterative workloads and it reinforces disciplined memory management and encourages developers to treat GPU memory as a critical, limited resource.

### *4.2.2 Design Choices*

The implementation of the `DeviceVector<T>` abstraction was guided by deliberate design choices aimed at balancing usability, performance, and compatibility within heterogeneous CUDA applications. These choices prioritize clarity, composability, and explicit control over memory, while remaining easy to integrate into existing C++ workflows.

- **Favor Composition Over Inheritance**

  The class design favors composition by containing a `std::vector<T>` internally, rather than inheriting from it. This choice provides better encapsulation, enabling the class to manage GPU memory independently of the host container's interface. It also avoids the complexities of exposing and maintaining base class internals, and allows greater flexibility in controlling how and when specific functions are forwarded or overloaded. Composition ensures a clear separation between host-side logic and GPU-side control.

- **Default Host Behavior**

  The `DeviceVector` is designed to behave like a `std::vector<T>` by default when used on the host. This eliminates the need for developers to explicitly extract or access the internal container. Through implicit conversion and function forwarding, host code can perform standard operations (e.g., indexing, resizing, appending) as with a normal STL vector. This improves usability and ensures the abstraction is intuitive for developers familiar with standard C++ containers.

- **Device-Side Simplicity via Raw Pointer Access**

  The `DeviceVector` class is intentionally designed to expose the device memory as a raw `T*` pointer when passed into CUDA kernels. This reflects the underlying GPU programming model, where device addresses are raw pointers by nature. The abstraction does not attempt to wrap device-side behavior, perform internal bounds checking, or handle invalid memory access at runtime. Instead, it mirrors the expectations of writing raw CUDA applications—developers are still responsible for ensuring proper memory allocation, synchronization, and safe access patterns.

- **Selective Forwarding of STL Member Functions**

  Only a minimal subset of commonly used functions such as `push_back()`, `resize()`, and `operator[]` are explicitly forwarded to the internal `std::vector<T>`. This approach balances convenience with control, exposing only essential operations needed for typical use cases while keeping the interface clean. It avoids the need to re-implement the full `std::vector` API, which would add unnecessary complexity and maintenance cost.

- **Explicit Memory Management and Synchronization Model**

  The class does not use CUDA's unified memory or other unified dynamic memory operators. Instead, it relies on manual memory allocation and explicit synchronization, requiring developers to call functions like `allocateDeviceMemory()`, `copyToDevice()`, and `copyToHost()` as needed. This approach ensures transparency, fine-grained control, and predictable performance, especially in applications where memory usage and synchronization patterns are critical to correctness and efficiency.

- **Implicit Conversion to `T*` for Kernel Calls**

  To enable seamless integration with CUDA kernels, the class overloads the cast operator to allow implicit conversion from `DeviceVector<T>` to a raw `T*` (i.e., device pointer). This eliminates the need for explicit calls to `.dataDevice()` or `.getPointer()`, allowing `DeviceVector` objects to be passed directly to kernel launch arguments in a syntactically clean and intuitive way.

- **Implicit Conversion to `std::vector<T>&` for Host Compatibility**

  The class also provides an implicit conversion to `std::vector<T>&`, allowing it to interoperate with existing host code that expects a standard container. This ensures backward compatibility and simplifies integration into legacy or external codebases without requiring major refactoring. Developers can use `DeviceVector` in algorithms, iterators, or STL-compatible functions without breaking existing logic.

- **Explicit Type Set**

  Unlike general-purpose containers that support arbitrary template types, the `DeviceVector` class deliberately restricts its supported types through a compile time type checking. By explicitly enumerating supported types—such as `int`, `float`, and `bool`—this abstraction avoids undefined behavior and ensures that memory management and device access remain correct at all points. This choice was made in consideration of the Graphitti simulation platform, which already operates using a fixed and well-defined set of data types. Moreover, this constraint helps safeguard the abstraction from being misused with unsupported or complex object types, such as classes with dynamic memory or virtual functions, which are incompatible with raw GPU memory access, thereby preventing silent errors or subtle bugs from arising in future extensions. At the same time, this approach preserves backward compatibility with existing code and provides a clear path for extending the type set in a controlled and predictable manner, should additional types need to be supported in the future.

- **Conditional Compilation**

  Since `DeviceVector` includes both host and device-side operations, its implementation is conditionally compiled using the `__CUDACC__` macro, which is defined by `nvcc` during device-side compilation. This ensures that device-specific logic and pointers are excluded from host-only builds, avoiding compilation errors. Unlike application-defined macros `USE_GPU`, which cannot differentiate between host and device translation units, `__CUDACC__` provides a reliable mechanism for this. This distinction is essential because `DeviceVector` is included in both host headers (`.h`) and device implementation files (`_d.cpp`), and headers in C++ are not compiled independently. Instead, compilation context is determined by the source file including them. Using `__CUDACC__` ensures that CUDA constructs are only processed by `nvcc` when compiling device files, preserving compatibility and correctness across heterogeneous build configurations.

### *4.2.3 Integration*

- **Replacing std::vector with DeviceVector**

  Each data member of std::vector type within all neural vertex classes was systematically refactored to use DeviceVector instead. This transformation was applied across classes such as AllSpikingNeurons, AllIFNeurons, and other derived vertex types.

Listing 4.9: Replacement of vector by DeviceVector

```
class AllIFNeurons {
    ....
    vector<BGFloat> Trefract_;
};
```

```
class AllIFNeurons {
    ....
    DeviceVector<BGFloat> Trefract_;
};
```

- **Removing Device Properties Structures**

  The structures that held raw device pointers for each data member, as mentioned in Section 2.1, were removed. With the adoption of DeviceVector, this layer became unnecessary, as each instance now encapsulates both host and device memory.

  **Example:**

Listing 4.10: Removal of DeviceProperties

```
struct AllIFNeuronsDeviceProperties {
    ....
    BGFloat *Trefract_;
};
```

- **Replacing Device Memory Operations with method call**

  All CUDA memory management operations such as cudaMalloc, cudaMemcpy, and cudaFree were replaced with method calls.

  **Example:**

Listing 4.11: Replacment of CUDA operations with method calls

```
HANDLE_ERROR(cudaMalloc(Trefract_,...));
HANDLE_ERROR(cudaMemcpy(Trefract_, ..., cudaMemcpyHostToDevice));
HANDLE_ERROR(cudaMemcpy(..,Trefract_, ...cudaMemcpyDeviceToHost));
HANDLE_ERROR(cudaFree(Trefract_));
```

```
Trefract_.allocateDeviceMemory();
Trefract_.copyToDevice();
Trefract_.copyToHost()
Trefract_.freeDeviceMemory();
```

- **Modifying CUDA Kernels**

  All CUDA kernels in neural vertices classes were modified to accept raw pointers as argument encapsulated by device vector itself instead of struct holding device pointers. Moreover, within the kernel, data was accessed directly rather than having indirection using device struct.

  **Example:**

Listing 4.12: Modifying CUDA Kernels

```
__global__ void advanceLIFNeuronsDevice(....,
    AllIFNeuronsDeviceProperties *allVerticesDevice) {
    ...allVerticesDevice->Trefract_[i];
}
....
advanceLIFNeuronsDevice<<<blocksPerGrid, threadsPerBlock>>>(
    allIFNeuronsDeviceProperties);
```

```
__global__ void advanceLIFNeuronsDevice(..., BGFLOAT *Trefract_,
    BGFLOAT *Vthresh_) {
    ...Trefract_[i];
}
....
advanceLIFNeuronsDevice<<<blocksPerGrid, threadsPerBlock>>>(Trefract_,
     Vthresh_);
```

# Chapter 5

# RESULTS

## *5.1 Quantative Evaluation*

This section presents the results of integrating DeviceVector into Graphitti. Given the limited availability of well-defined metrics for evaluating programming abstractions, a set of standard software engineering metrics is employed to assess its effectiveness. In addition, we evaluate the performance overhead, with the expectation that DeviceVector introduces minimal to no additional computational overhead.

### *5.1.1 Code Reduction*

Code reduction refers to the potential decrease in lines of code (LOC) resulting from the integration of `DeviceVector`. A lower LOC often correlates with simpler, more maintainable, and faster-to-develop implementations.

**Lines of Code (LOC) Comparison**

**Before Integration:**

$\text{LOC}_{\text{Neuro}}$ $= 1514$

**After Integration:**

$\text{LOC}_{\text{Neuro}}$ $= 1428$

$\text{LOC}_{\text{DeviceVector}}$ $= 341$

$\text{LOC}_{\text{Total}}$ $= 1428 + 341 = 1769$

**Reduction:**

$\text{LOC}_{\text{Neuro}}$ $= 1514 - 1428 = \mathbf{86}$

Percentage Reduction $= \left(\frac{86}{1514}\right) \times 100 \approx \mathbf{5.68\%}$

It shows that although the introduction of DeviceVector increases the lines of code due to its own definition and implementation, it significantly reduced the code for neural vertices. This reduction was expected, as it enabled the removal of all device-side structures and replaced low-level CUDA operations with a single method call. This means that as DeviceVector is progressively adopted throughout the codebase, the overall lines of code should steadily decrease, ultimately resulting in a net reduction.

### *5.1.2* ***Boilerplate Ratio***

Boilerplate refers to repetitive code written to handle tasks such as device memory management—code that is necessary for execution but not part of the core simulation logic. The Boilerplate Ratio quantifies the proportion of such auxiliary code relative to the total lines of code, serving as an indicator of structural or syntactic overhead. High boilerplate often results from low-level implementations that lack abstraction. `DeviceVector` addresses this by encapsulating common memory and synchronization patterns into reusable components.

**Before Integration**

| | |
|---|---|
| $\text{LOC}_{\text{Neuro}}$ | $= 1514$ |
| $\text{LOC}_{\text{Boilerplate}}$ | $= 139$ |
| Boilerplate Ratio | $\approx 9\%$ |

**After Integration**

| | |
|---|---|
| $\text{LOC}_{\text{Neuro}}$ | $= 1428$ |
| $\text{LOC}_{\text{Boilerplate}}$ | $= 39$ |
| Boilerplate Ratio | $\approx 2.7\%$ |

The reduction in boilerplate ratio was expected, as DeviceVector now encapsulates memory management on the device. This eliminates the need to repeatedly write boilerplate CUDA code, thereby reducing development effort and minimizing the risk of human error.

In fact, during integration process a bug was identified in one of the classes, where GPU operations—including allocation and memory transfers—were mistakenly applied to the wrong data object multiple times. This type of error, caused by redundant code, should now be reduced.

### *5.1.3* ***Code Affected***

Code Affected refers to the set of code locations exposed to change during the integration process. It includes all functions, classes, or modules that had to be touched, examined, or reviewed—regardless of whether they were directly modified. By capturing both actual modifications and reviewed code, this measure reflects the breadth of the integration effort and provides insight into the extent to which the new abstraction actually impacts the existing codebase. It serves as an indicator of the mental overhead and review burden imposed on developers, offering a proxy for the invasiveness of the change.

$$\text{Classes Affected (replacing vector with DeviceVector etc.)} = 4$$
$$\text{Total Classes} = 4$$

$$\text{Host Methods Affected} = 0$$
$$\text{Total Host Methods} = 35$$

$$\text{Device Methods Affected (Kernels etc.)} = 25$$
$$\text{Total Device Methods} = 25$$

This shows on the host side, integrating DeviceVector requires minimal changes, as it internally overloads the operators and methods of std::vector to preserve interface compatibility. As a result, once the substitution is made, the core logic remains unchanged. This makes the process low risk, particularly given that most modern code editors support reference tracking, allowing std::vector instances to be reliably and efficiently replaced with DeviceVector.

In contrast, on the device side, all methods and kernels are restructured to accommo-

date GPU-specific behavior. Specifically, all kernel signatures are updated to accept data parameters explicitly. Consequently, all references to these parameters within the kernel and methods must also be updated, eliminating the need for indirection through structures. Despite the breadth of these changes, the integration remains low risk, as inconsistencies are reliably caught at compile time. Legacy kernel signatures or continued use of structure-based indirection result in compiler errors, thereby minimizing the likelihood of subtle runtime bugs and facilitating early detection during development.

Moreover, modifying memory management operations is relatively less error-prone, as the changes primarily affect how data is handled rather than how it is referenced. For each data member, the original code using explicit CUDA operations and the new code using method calls both reference the same variable names same number of times, reducing the likelihood of introducing errors during refactoring.

### *5.1.4* ***Symbolic Complexity***

Symbolic Complexity quantifies the total number of distinct symbols or variables involved in individual code statements that are modified during the integration of an abstraction or refactoring process. A lower symbol count indicates cleaner abstraction and reduced cognitive load, while a higher count may suggest verbose, error-prone operations. This metric is useful for evaluating how well an abstraction simplifies common operations

**Before Integration**

Symbols in LOC Modified $= 957$

**After Integration**

Symbols in LOC Modified $= 518$

**Reduction**

Change $= 957 - 518 = \mathbf{439}$

Percentage Reduction $= \left(\frac{439}{957}\right) \times 100 \approx \mathbf{45.87\%}$

This demonstrates a significant reduction in the number of symbols used in the modified

statements. This outcome was expected, as `DeviceVector` abstracts away verbose and complex CUDA operations that typically require multiple parameters per call, replacing them with concise method calls. While kernel signatures saw an increase in parameters due to the unwrapping of device structures, this increase did not substantially impact the overall symbol count. In total, the change resulted in a net reduction in symbolic complexity.

### *5.1.5* ***Cyclomatic Complexity***

Cyclomatic Complexity quantifies the number of independent control flow paths within a function or code unit. It reflects the structural decision complexity introduced by conditional constructs such as if, for, while, and switch. A higher cyclomatic complexity indicates more branching logic, which can increase testing effort and reduce maintainability. This metric is useful for evaluating how well an abstraction reduces conditional logic or improves control flow simplicity.

**Before Integration**

Cyclomatic Complexity $= 48$

**After Integration**

Cyclomatic Complexity $= 46$

This demonstrates that there is no significant change in cyclomatic complexity following the integration of `DeviceVector`. This outcome was expected, as `DeviceVector` neither introduces nor eliminates branching logic within the existing code. The slight reduction observed may be attributed to the removal of conditional compilation statements previously used for managing device-specific structures.

### *5.1.6* ***Runtime Performance Overhead***

Performance overhead refers to the additional time or computational resources introduced by an abstraction or system component compared to a baseline implementation. It reflects the trade-off between ease of use and efficiency, often arising when higher-level abstractions simplify development at the cost of lower-level performance optimizations. Minimizing per-

formance overhead is critical in performance-sensitive domains such as GPU programming, where indirect control over memory and execution can lead to measurable slowdowns.

Below is the runtime execution time for one of the simulation test cases, averaged over five runs.

**Execution Time (in seconds):**

Before Integration

CPU = 650.4

GPU = 30.4

After Integration

CPU = 657.6

GPU = 29.6

The runtime results for both CPU and GPU show minimal or no change in performance after integrating DeviceVector. This aligns with expectations, as DeviceVector merely encapsulates details, while the underlying GPU memory operations remain unchanged.

## *5.2 Qualitative Evaluation*

This section presents a qualitative comparison of `DeviceVector` against existing abstractions in terms of ease of integration, usability, control, and portability. It first defines the levels—high, medium, and low—along with their associated indicators, and then provides a comparison table that highlights each abstraction's characteristics, assigning levels based on those characteristics.

### *5.2.1 Ease of Integration*

Ease of integration refers to how smoothly an abstraction can be adopted within an existing codebase or workflow. It considers factors such as the amount of refactoring required, compatibility with existing components, and toolchain dependencies. A high ease of integration

minimizes disruption and accelerates adoption without requiring significant architectural changes. `DeviceVector` is designed to closely mirror the interface of `std::vector`, enabling developers to incorporate it into the existing Graphitti codebase with minimal modifications. In most cases, replacing `std::vector` with `DeviceVector`—along with updating memory management and kernel/function parameter handling—is sufficient, without the need to alter core logic.

Table 5.5: Ease of Integration Levels

| **Level** | **Indicators** |
|---|---|
| **High** | Easily integrates with existing code; minimal changes required; Localized refactoring and no toolchain changes. |
| **Medium** | Requires moderate effort; may need some refactoring or tool changes; Signature changes etc. |
| **Low** | Major refactoring required; disrupts architecture or needs new build systems; new programming model and incompatibility with existing logic. |

Table 5.6: Ease of Integration Comparison

| Framework / Tool(s) | Ease of Integration | Integration Characteristics |
|---|---|---|
| **DeviceVector** | High | Directly replaceable for `std::vector`; minimal restructuring for CUDA (application specific). |
| **Kokkos / RAJA** | Medium | Requires `parallel_for` (loops), policy setup and replacing data with `Views` (no support for methods). |
| **Thrust** | Low | Easy for simple operations, but harder to integrate in large codebases due to lack of custom kernel support. |
| **SYCL** | Low | Demands significant restructuring to adopt buffers, accessors, and explicit device queues;kernel launches must follow SYCL model. |
| **OpenACC / NVC++** | High | Minimal changes needed; annotate existing loops with pragmas. |
| **OpenCL** | Low | Requires explicit management of platforms, contexts, and command queues, increasing boilerplate code. |
| **SkelCL / SkePU** | Low | Requires adaptation to skeleton patterns; limited general-purpose operations. |
| **FastFlow** | Low | Straightforward for stream-based CPU pipelines; minimal native GPU support; |
| **PhAST** | Low | Needs external code generation and configuration file handling; integration is non-trivial; lacks general-purpose kernel support. |

### *5.2.2* ***Usability***

Usability refers to how easily a developer can learn, understand, and effectively use a programming abstraction or tool. It encompasses the clarity of the interface, learning curve, error proneness, and how well the design aligns with developer expectations. High usability reduces cognitive load and accelerates development without sacrificing correctness. `DeviceVector` supports this by providing intuitive methods such as `.copyToDevice()`, `.copyToHost()`, and `.getDevicePtr()` for accessing and synchronizing memory. These methods follow conventions familiar to C++ developers, while abstracting away boilerplate CUDA operations like `cudaMalloc`, `cudaMemcpy`, and `cudaFree`.

Table 5.7: Usability Levels

| **Level** | **Indicators** |
|---|---|
| **High** | Intuitive and easy to use; Familiar syntax; resembles existing containers, minimal boilerplate, clear naming, fast developer onboarding. |
| **Medium** | Understandable with some learning; needs familiarity with templates, requires reading documentation, moderate boilerplate or control flow understanding. |
| **Low** | Verbose with steep learning curve; complex low-level constructs, unintuitive design or unfamiliar programming paradigm. |

Table 5.8: Usability Comparison

| Framework / Tool(s) | Usability | Usability Characteristics |
|---|---|---|
| **DeviceVector** | High | Intuitive methods like `.copyToDevice()` etc. |
| **Kokkos / RAJA** | Medium | Requires policy understanding; flexible but has a learning curve. |
| **Thrust** | High | STL-like containers; easy to use |
| **SYCL** | Low | Verbose with complex low-level control. |
| **OpenACC / NVC++** | High | Directive-based; integrates with minimal code changes. |
| **OpenCL** | Medium | Low-level and verbose; kernels written as strings etc. |
| **SkelCL / SkePU** | Low | Skeleton-templates based; verbose and only supports specific patterns. |
| **FastFlow** | Low | Intuitive for Stream-based and multicore CPUs, minimal for GPU applications. |
| **PhAST** | Low | Niche, stencil-specific tool; relies on code generation and unfamiliar configuration file syntax. |

### *5.2.3* ***Portability***

Portability refers to the ease with which a program, abstraction, or framework can run across different hardware architectures, platforms, or compilers with little to no modification. High portability allows code to adapt to various environments—such as CPUs, GPUs from different vendors without requiring significant rewrites.

Table 5.9: Portability Levels

| Level | Indicators |
|---|---|
| **High** | Runs across multiple hardware platforms; vendor-neutral design; supports both CPU and GPU targets without platform-specific logic. |
| **Medium** | Portable with moderate effort; may require backend-specific tuning or conditional compilations; |
| **Low** | Tied to a specific vendor; designed for specific application or strong dependence on platform specific APIs drivers. |

Table 5.10: Portability Comparison

| Framework / Tool(s) | Portability | Portability Characteristics |
|---|---|---|
| **DeviceVector** | Low | CUDA-specific and targeted for Graphitti. |
| **Kokkos / RAJA** | High | Supports multiple backends (CUDA, HIP, OpenMP); portable across NVIDIA, AMD, and CPUs. |
| **Thrust** | Low | CUDA-based; not portable outside NVIDIA GPUs. |
| **SYCL** | High | Cross-platform C++ abstraction (via DPC++) supporting NVIDIA, AMD, Intel, and CPU targets. |
| **OpenACC / NVC++** | Medium–High | Directive-based model supported by multiple compilers; portable across CPU and GPU with compatible toolchains. |
| **OpenCL** | High | Vendor-neutral; works across NVIDIA, AMD, Intel GPUs, and CPUs. |
| **SkelCL / SkePU** | High | Built on OpenCL backend, offering portability. |
| **FastFlow** | Low | Optimized for multicore CPUs; limited GPU support. |
| **PhAST** | Low | Stencil-specific tool with niche applicability; limited portability and requires code generation. |

### *5.2.4* ***Control***

Control refers to the degree of programmability a framework provides over execution behavior, including memory management, kernel launch configuration, and synchronization. It also encompasses the nature of control—whether it is user-defined (explicit) or fixed and managed by the compiler or runtime. `DeviceVector` offers a balanced level of control by abstracting routine operations like memory allocation and data transfer, while still allowing developers to explicitly manage when and how data is moved between host and device using methods like `.copyToDevice()` and `.copyToHost()`. This enables fine-grained performance tuning without fully exposing the developer to low-level CUDA APIs, achieving a compromise between flexibility and simplicity.

Table 5.11: Control Levels

| **Level** | **Indicators** |
|---|---|
| **High** | Full control over memory, execution, or synchronization; launch configuration (threads, blocks) control etc. |
| **Medium** | Partial control with some abstractions; tuning possible but constrained. |
| **Low** | Minimal or no direct control; no manual memory handling or kernel tuning |

Table 5.12: Control Comparison

| Framework / Tool(s) | Control Level | Control Characteristics |
|---|---|---|
| **DeviceVector** | High | Explicit control over GPU memory operations; no change in kernel launch. |
| **Kokkos / RAJA** | Medium | Uses execution/memory policies to offer control; allows backend selection, tuning, and memory spaces. |
| **Thrust** | Low | Offers STL-like operations with less control over memory and kernel launches. |
| **SYCL** | Medium | Explicit control over memory and execution; but limits tuning of launch configuration and shared memory. |
| **OpenACC / NVC++** | Low | Abstracts memory and kernel management via directives. |
| **OpenCL** | High | Complete low-level control over memory, device selection etc. |
| **SkelCL / SkePU** | Low | Hides memory and execution details behind skeletons; limited ability to control memory or customize execution patterns. |
| **FastFlow** | Low | Stream-based CPU abstraction with limited control over GPU memory. |
| **PhAST** | Low | Focused on stencil automation; memory layout and kernel control are mostly internal or auto-generated. |

### *5.2.5* ***Extensibility***

`DeviceVector` is designed with extensibility in mind, allowing future enhancements without disrupting existing functionality. Its templated architecture can be extended to accommodate new data types or memory access patterns as simulation requirements evolve. Because it encapsulates host and device memory logic in a unified interface, developers can introduce advanced features—like RAII synchronization strategies, or device-side views for complex

object heirarchies—while preserving compatibility with current code. This modularity ensures that `DeviceVector` can grow alongside Graphitti's evolving GPU capabilities.

### *5.2.6* ***Logical Relationship between device and host data***

In traditional CUDA codebases, it is common to manage host-side and device-side data separately, often using distinct variables that merely share the same name or rely on comment-based conventions to indicate their relationship. However, these variables are logically disconnected in the actual implementation—there is no formal structure binding them together. This loose association can lead to bugs, mismanaged memory, or accidental inconsistencies in data movement between the host and the GPU.

In contrast, the `DeviceVector` abstraction explicitly encapsulates both host and device representations within a single object.The host-side data (typically stored in a `std::vector<T>`) and the corresponding device-side memory (represented by a `T*` pointer) are bound together within a unified structure. This design enforces a clear and consistent logical relationship between the two: they are not merely linked by convention, but physically encapsulated, promoting conceptual integrity and reducing opportunities for error.

# Chapter 6

# CONCLUSION

## *6.1 Summary*

The primary objective of this thesis was to design abstractions specifically targeted to Graphitti that simplify GPU development and enhance programmability. To achieve this, the thesis reviews existing abstractions and categorizes them to analyze how different types of abstractions address various aspects of heterogeneous application development. It uses insights from that review to present the design and implementation of DeviceVector, a lightweight abstraction that unifies host and device memory management in Graphitti.

`DeviceVector` enhances programmability by reducing code duplication and abstracting CUDA boilerplate through an interface that closely mirrors a standard C++ container. Specifically designed to address the programmability challenges in Graphitti's GPU-based simulation, `DeviceVector` lowers development complexity, improves maintainability, and increases developer productivity. It achieves these benefits with minimal disruption to the existing codebase and a low risk of integration errors. Importantly, the abstraction preserves fine-grained control, making it well-suited for performance-critical applications such as large-scale simulations.

The integration of `DeviceVector` was quantitatively evaluated using standard software engineering metrics, including code reduction and boilerplate ratio. This work also provides a qualitative evaluation of `DeviceVector` in terms of ease of integration, usability, and other criteria by comparing its characteristics with those of existing abstractions. These comparisons offer insights into the trade-offs and limitations of current solutions and inform future design directions.

This work lays out the design approach and choices behind DeviceVector, which is designed to improve the programmability of Graphitti so that it can be easily extended by researchers to implement additional simulation domains. Rather than offering yet another

general-purpose library, the intention is to provide a starting point or offer insights and guidance for designing targeted, maintainable abstractions.

### *6.2 Future Work*

In the context of future research, several avenues merit further exploration to enhance the programmability of GPU development within heterogeneous applications.

Foremost, a key direction for future work is to integrate DeviceVector with other subsystems within Graphitti (e.g., Edges). Given the documentation and this thesis as a reference, this integration should now require relatively minimal effort. One promising enhancement is the incorporation of RAII principles into the `DeviceVector` class—specifically, by relocating the `freeDeviceMemory()` operation into the class destructor. Since GPU-resident data is meaningful only when its corresponding host container exists (as host-side retrieval is essential for storage, analysis, and further computation), binding the device memory lifecycle to the host object aligns naturally with RAII. This would relieve developers of the manual burden of memory deallocation and reduce the likelihood of memory leaks. However, such a modification should be implemented cautiously and only after thoroughly evaluating Graphitti's data flow and memory management requirements.

Another important direction involves introducing function-level abstractions to further reduce logic duplication and enhance maintainability. The computational logic in many Graphitti components remains largely consistent across host and device contexts, with the primary differences lying in parameter handling. By leveraging unified templates and CUDA's `__host__ __device__` function specifiers, it becomes feasible to consolidate these logic paths. This concept is discussed in Section A.2. Although this approach has not yet been integrated into Graphitti due to time constraints, it holds significant potential for future improvements.

Furthermore, while `DeviceVector` currently supports only primitive data types, extending it to support user-defined object types could enhance its generality and applicability. Implementing such support would require a careful study of relevant use cases, the development of clear design guidelines, and rigorous testing to ensure that the abstraction remains effective without introducing unnecessary complexity. A conceptual design for integrating

`DeviceVector` with object types is presented in section A.1, demonstrating the feasibility of this idea within the context of Graphitti.

Finally, while this work primarily focuses on memory management, other dimensions of GPU programmability could similarly benefit from lightweight, modular solutions. Exploring these aspects in future work may lead to more holistic and generalizable improvements across heterogeneous computing. The central idea behind this thesis is that, through iterative enhancements to Graphitti and the insights gained along the way, it will be possible to identify recurring challenges and derive design patterns that go beyond abstraction alone. Just as traditional design patterns have long provided reusable solutions to recurring problems in software engineering, the patterns emerging from this work can offer structured guidance for addressing common issues in heterogeneous systems—including, but not limited to, memory management. These patterns are intended to guide the development of maintainable and well-structured code that effectively addresses the specific challenges of an application.

Overall, this work lays the foundation not only for future research and development of application-specific GPU abstractions, but also for the formulation of generalizable design patterns that can consistently address recurring challenges in heterogeneous applications.

# BIBLIOGRAPHY

[1] H. Markram and et al., "Reconstruction And Simulation Of Neocortical Microcircuitry," *Cell*, vol. 163, no. 2, pp. 456–492, Oct. 2015. DOI: 10.1016/j.cell.2015.09.029.

[2] J. A. Anderson, J. Glaser, and S. C. Glotzer, "HOOMD-blue: A Python Package For High-performance Molecular Dynamics And Hard Particle Monte Carlo Simulations," *Computational Materials Science*, vol. 173, p. 109 363, 2020. DOI: 10.1016/j.commatsci.2019.109363.

[3] J. Leskovec and R. Sosič, "SNAP: A General-Purpose Network Analysis And Graph-Mining Library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, Jul. 2016, ISSN: 2157-6904. DOI: 10.1145/2898361.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008. DOI: 10.1109/JPROC.2008.917757.

[5] S. Mittal and J. S. Vetter, "A Survey Of Methods For Analyzing And Improving GPU Energy Efficiency," vol. 47, no. 2, Aug. 2014, ISSN: 0360-0300. DOI: 10.1145/2636342.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming With CUDA," *Queue*, vol. 6, pp. 40–53, Mar. 2008. DOI: 10.1145/1401132.1401152.

[7] J. Stone, D. Hardy, I. Ufimtsev, and K. Schulten, "GPU-Accelerated Molecular Modeling Coming Of Age," *Journal of molecular graphics  modelling*, vol. 29, pp. 116–25, Sep. 2010. DOI: 10.1016/j.jmgm.2010.06.010.

[8] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010, ISBN: 0123814723.

[9] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles And Application Performance Evaluation Of A Multithreaded GPU Using CUDA," PPoPP '08, pp. 73–82, 2008. DOI: 10.1145/1345206.1345220.

[10] G. Juckeland and S. Chandrasekaran, *OpenACC For Programmers: Concepts And Strategies*. Sep. 2017, ISBN: 978-0134694283.

[11] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[12] A. A. Rudrawar, "Evaluating Impact Of GPU API Evolution On Software Development And Application Performance.," M.S. thesis, UW Bothell, 2022.

[13] V. J. Salvatore, "Demonstrating Software Reusability: Simulating Emergency Response Network Agility With A Graph-Based Simulator," M.S. thesis, UW Bothell, 2021.

[14] S. Singh, "Graph Analysis For Simulated Neural Networks With STDP.," M.S. thesis, UW Bothell, 2021.

[15] A. Marowka, "On The Performance Portability Of OpenACC, OpenMP, Kokkos And RAJA," in *International Conference On High Performance Computing In Asia-Pacific Region*, ser. HPCAsia '22, Virtual Event, Japan: Association for Computing Machinery, 2022, pp. 103–114, ISBN: 9781450384988. DOI: 10.1145/3492805.3492806.

[16] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC — First Experiences With Real-World Applications," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 859–870, ISBN: 978-3-642-32820-6.

[17] M. Steuwer, P. Kegel, and S. Gorlatch, *SkelCL - A Portable Skeleton Library For High-Level GPU Programming*, Jun. 2011. DOI: 10.1109/IPDPS.2011.269.

[18] S. Ernsting and H. Kuchen, "Algorithmic Skeletons For Multi-core, Multi-GPU Systems And Clusters," *International Journal of High Performance Computing and Networking*, vol. 7, pp. 129–138, Apr. 2012. DOI: 10.1504/IJHPCN.2012.046370.

[19] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003.

[20] R. D. Hornung and J. A. Keasler, "The RAJA Portability Layer: Overview And Status," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep., Sep. 2014. DOI: 10.2172/1169830.

[21] C. R. Trott, D. Lebrun-Grandié, D. Arndt, *et al.*, "Kokkos 3: Programming Model Extensions For The Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022. DOI: 10.1109/TPDS.2021.3097283.

[22] N. Bell and J. Hoberock, "Chapter 26 - Thrust: A Productivity-Oriented Library For CUDA," in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W.-m. W. Hwu, Ed., Boston: Morgan Kaufmann, 2012, pp. 359–371, ISBN: 978-0-12-385963-1. DOI: https://doi.org/10.1016/B978-0-12-385963-1.00026-5.

[23] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ For Programming Of Heterogeneous Systems Using C++ And SYCL*. Jan. 2021, ISBN: 978-1-4842-5573-5. DOI: 10.1007/978-1-4842-5574-2.

[24] E. Zenker, B. Worpitz, R. Widera, *et al.*, "Alpaka - An Abstraction Library For Parallel Kernel Acceleration," Feb. 2016. DOI: 10.48550/arXiv.1602.08477.

[25] M. Goli and H. González-Vélez, "Heterogeneous Algorithmic Skeletons For Fast Flow With Seamless Coordination Over Hybrid Architectures," in *2013 21st Euromicro International Conference On Parallel, Distributed, And Network-Based Processing*, 2013, pp. 148–156. DOI: 10.1109/PDP.2013.29.

[26] B. Peccerillo and S. Bartolini, "PHAST - A Portable High-Level Modern C++ Programming Library For GPUs And Multi-Cores," *IEEE Transactions on Parallel and*

*Distributed Systems*, vol. 30, no. 1, pp. 174–189, 2019. DOI: 10.1109/TPDS.2018.2855182.

[27] M. Kuricheti, "Using Modern C To Improve CUDA Programs," University of California, Davis, Master's thesis, 2024, Discusses modern C++ abstractions (iterators, buffer views, bounds checking) in CUDA kernel code.

[28] G. E. G. David J. Skudra, "C++ Resource Intelligent Compilation For GPU Enabled Applications," *NASA STI Program, NASA/TM–2018–219897*, 2018.

[29] J. M. Jordan and M. Stiber, "Graph-based Modeling And Simulation Of Emergency Services Communication Systems," in *2024 32nd International Conference On Modeling, Analysis And Simulation Of Computer And Telecommunication Systems (MASCOTS)*, 2024, pp. 1–4. DOI: 10.1109/MASCOTS64422.2024.10786343.

[30] OpenAI, *ChatGPT (May - June 2025 Version)*, Accessed: 2025-05 and 2025-06, 2025.

# Appendix A

# OTHER DESIGN INSIGHTS

## *A.1 GPU-Safe Views for Integration of DeviceVector with RecordableVector and EventBuffer (Object Types)*

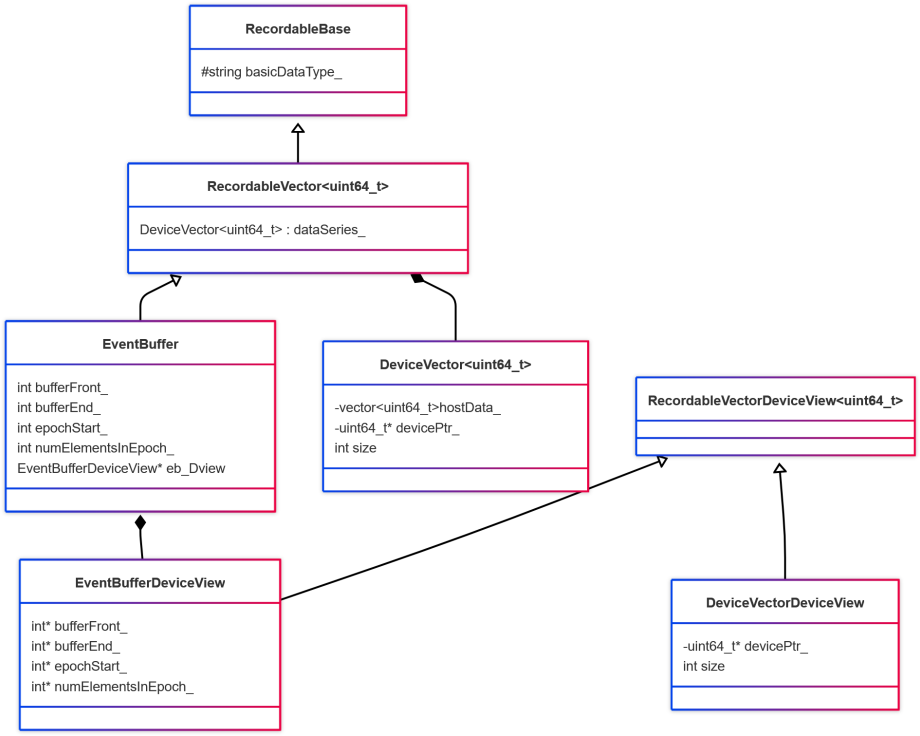


Figure A.1: Integration design of `DeviceVector` with `RecordableVector` and `EventBuffer`.

This presents a conceptual design for how `DeviceVector` integrates with the existing `RecordableVector` and `EventBuffer`, which are used to record data in Graphitti.

The key idea behind supporting complex objects is the introduction of a **device view** — a plain-old-data (POD) struct that mirrors the layout of the original object using only GPU-compatible types. This device view explicitly holds raw device pointers to data members

and excludes any host-only constructs, such as virtual functions (which introduce vtables), STL containers like `std::vector`, or even abstractions like `DeviceVector` itself.

Because not all host objects are trivially transferrable to the device, the responsibility of defining a suitable device view lies with the user. As shown in A.1, to support GPU-safe integration between `DeviceVector` and `EventBuffer`—a complex object that internally uses host-only types—we introduce a `EventBufferDeviceView`. This view inherits from `RecordableVectorDeviceView` and holds device pointers to the data required on the GPU. The original `EventBuffer` instance maintains a reference to its device view, enabling kernels to access GPU-allocated data safely.

While this design allows safe GPU usage, it does introduce limitations. First, data duplication is present since both `dataSeries` (inherited from `RecordableVector`) and the device view (from d_ptr) is referencing the same memory. Second, this approach does not fundamentally solve the problem of making `DeviceVector` compatible with object-oriented constructs like `EventBuffer`; rather, it shifts the burden of compatibility to a user-defined translation layer.

Despite these issues, this design serves as a reference point for what approaches may not scale or generalize. By identifying the pain points in abstraction, we can better inform future iterations that strive for a more balanced trade-off between usability and performance.

### *A.2 Function Abstractions for Graphitti*

The core idea behind this abstraction is that the functional logic required on both the host and device sides is mostly similar, with differences primarily in parameter handling and execution context. CUDA provides execution space specifiers such as `__device__` and `__host__` to distinguish where a function is executed and from where it can be called. Specifically, `__device__` marks a function that is executed on the device and callable only from device code, while `__host__` marks functions executed and callable solely from the host. When both specifiers are used together, the function is compiled for both host and device execution, enabling shared logic across execution spaces. This construct has been widely adopted and represents a well-established technique in CUDA development for unifying host and device implementations while minimizing code duplication.

```
// CUDA/C++ host OR device code

// Necessary to prevent qualifiers from getting in host pipeline
#ifdef __CUDACC__
  #define HOSTDEVICEQUALIFIER __host__ __device__
#else
  #define HOSTDEVICEQUALIFIER
#endif

HOSTDEVICEQUALIFIER void someModel(int particles) {
#ifdef __CUDA_ARCH__
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < particles; i += stride) {
#else
  for (int i = 0; i < particles; i++) {
#endif
  // 50 lines of common computation here
  }
}
```

Figure A.2: Example of usage of host and device qualifier and iterator conditional compilation reused from [28]

```
// CUDA/C++ host OR device code

#ifdef __CUDACC__
  #define KERNEL __global__
#else
  #define KERNEL
#endif

KERNEL void modelCallKern (int particles) {
  someModel(particles);
}

// Again, only exists in device compilation pipeline. Transfers program
// from host to device.
#ifdef __CUDACC__
void someModelGPU(int particles, int threadsPerBlock, int blocksPerThread)
    {
  modelCallKern<<<blocksPerGrid, threadsPerBlock>>>(particles);
}
#endif

void someModelCPU(int particles) {
  someModel(particles);
}
```

Figure A.3: Example of kernel and host method passing necessary arguments to same host-device qualified code, reused from [28]

Building on this idea, combining execution space specifiers with function templates allows flexible parameter passing while maintaining a unified implementation for both host and device contexts. An illustrative example of such a function is provided below, demonstrating how templated functions can be annotated for dual execution environments.

Listing A.1: Example showing current Methods in Graphitti

```
//host-function
bool isSpikeQueue(BGSIZE iEdg) {
  ...
}
//device-function
__device__ isSpikingSynapsesSpikeQueueDevice(allEdgesDevice, BGSIZE iEdg) {
  ....
}
```

Listing A.2: Example showing unified logic

```
template<typename T>
static __device__ __host__ bool isSpikeQueue(T* props, BGSIZE iEdg) {
    uint32_t& delayQueue = props->delayQueue_[iEdg];
    int& delayIdx = props->delayIndex_[iEdg];
    ....
}
isSpikeQueue(this, iEdg); //host-call
isSpikeQueue(edges, iEdg); //device-call
```

Although this approach has not yet been integrated into Graphitti as part of this project due to time constraints, it is presented here as a potential future enhancement. If adopted, this technique could further reduce code duplication and enhance maintainability by enabling unified host-device implementations through templated, execution-aware functions.