LANGUAGE DESIGNS FOR GEOMETRY AND HETEROGENEOUS REASONING IN GRAPHICS PROGRAMMING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Dietrich Geisler

August 2024

LANGUAGE DESIGNS FOR GEOMETRY AND HETEROGENEOUS REASONING IN GRAPHICS PROGRAMMING

Dietrich Geisler, Ph.D.

Cornell University 2024

There has been growing demand for 3D image rendering in the past several decades, primarily from video games, but also from fields as broad as film, art, architecture, and scientific simulation. A major challenge with expanding use of rendering, however, is that the programming is difficult, requiring significant field expertise when abstractions break down.

In this dissertation, we will examine how we may be able to design programming languages to ameliorate some of these challenges. Our goal will be to examine two specific challenges in graphics programming reasoning: geometric correctness and performance in heterogeneous device communication.

In the first part of this talk, we will discuss Gator, a language which provides type-level reasoning for a class of bugs we describe as "geometry bugs", as well as a lightweight mechanism to reason about operations on geometry. In the second part of this talk, we will discuss Caiman, a language which typechecks heterogeneous implementations against a fixed specification. We will also examine how Caiman's type-level restrictions allow for separating performance and correctness, as well as providing a mechanism for restricted synthesis of heterogeneous programs.

BIOGRAPHICAL SKETCH

Your biosketch goes here. Make sure it sits inside the brackets.

This document is dedicated to all Cornell graduate students.

ACKNOWLEDGEMENTS

Your acknowledgements go here. Make sure it sits inside the brackets.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer graphics has long been a core field of study within computer science. In the past decade alone, computer graphics has seen application in video games, animated film, scientific simulation, data visualization, and medical devices. The term computer graphics itself has become so broad as to be fuzzy; we refer here specifically to the study of modeling and rendering snapshots of 2 or 3-dimensional spaces onto a screen.

Despite the number of applications and domains that make use of computer graphics, maintaining or using software systems for rendering (such as a *rendering engine*) can be extremely difficult and time–consuming. Manipulating a rendering engine can require specialized learning in topics as diverse as light physics, geometry, visual design, and material science. Additionally, the history of these specialized topics can often find themselves at odds with the practical realities of building a performant computer system, resulting in complex engineering constraints and implicit rules for manipulating code.

As a consequence of this complexity, computer graphics has many domain–specific challenges that have been solved through sheer engineering prowess and, to put it bluntly, hacks on the tools available. For instance, game engines are frequently specialized to C++ engineering, relying on macros to control performance characteristics and providing highly specialized program behavior. Similarly GPUs come equipped with a rendering pipeline originally meant for the usual case of computer graphics, but as programmer specialization has outpaced hardware design, the GPU rendering pipeline has been taken apart and pieced back together to squeeze more performance or a specific behavior out of this hardware.

These challenges in graphics programming have real cost: large–scale rendering engines can be difficult to update for new technologies (such as Unreal Engine lumbering towards supporting raytracing), non–experts can be forced to rely on black–box implementations without any realistic mechanism to customize these implementations, and

performance can be left on the table in critical applications (there are some interesting examples of this, I need to find “good references" though).

The sheer breadth and complexity of these computer graphics and rendering systems, however, poses a unique opportunity for programming language designers. Improvements in languages for graphics specifically could provide mechanisms that keep up with graphics programmer needs faster than hardware design, and can expose domain–specific design challenges with interesting consequences for language research as a whole (I have a couple of citations here related to sampling theory).

Despite there being both potential and real need for graphics–specific programming language design, this focus of study has remained largely untapped. Notable efforts in this direction include languages for static and dynamic reasoning about the rendering pipeline and specialized languages for automatic and symbolic differentiation in geometric spaces. There are, however, few other high–profile efforts, despite anecdotally there remaining many potentially interesting problems (which will be further discussed in Section 1.0.2).

1.0.1 Summary of Work

In this dissertation, we present work that both identifies and provides language–level solutions for two specific challenges in graphics programming: geometric reasoning and performance exploration. In both cases, we identify properties of graphics programs which are either stated informally or otherwise known to the programmer, but are not communicated to the typechecker and compiler. Without this necessary context for the intended program semantics, the programmer loses the benefits of static typechecking and compiler optimizations, and may be forced to introduce the various C++ hacks described earlier.

Concretely, we describe three pieces of work:

- Gator, a language for providing semantics and typechecking for graphics–style geometry
- A paper on commutative diagram verification needed to solve a technical problem within Gator
- Caiman, a language for providing type–level support for heterogeneous performance exploration

Both Gator and the commutative diagram paper have been previously published.

Chapter 2 explores the relationship between the geometry of a scene being rendered and the code used to calculate properties of that scene through the lens of the Gator language. By identifying and naming three commonly–needed pieces of geometric information, Gator is able to provide geometry-aware types and semantics for several core graphics algorithms. We also show how introducing these types enables the Gator compiler to safely synthesize light–weight geometric transformations. The guarantees Gator aims to provide, however, resulted in needing a solution for online verification of commutative diagrams, a technical layer described further in Chapter 3.

Chapter 4 changes our focus to the performance concerns of graphics programmers, and specifically the narrow problem of inter–device communication, or heterogeneous programming. We introduce a language, Caiman, which provides a type system and compiler implementation for separating semantic and performance concerns in heterogeneous settings. We additionally develop and implement an algorithm for synthesizing these (otherwise complex) transformations in a type–directed and decomposable way, allowing a programmer to explore performance characteristics of a compiled program while maintaining control over the details of that program.

1.0.2 Open Directions

I plan to think about this section more, I'll defer on writing it for now.

CHAPTER 2

GATOR: GEOMETRY TYPES FOR GRAPHICS PROGRAMMING

CHAPTER 3

ONLINE VERIFICATION OF COMMUTATIVITY

CHAPTER 4

CAIMAN: DSL FOR OPTIMIZING HETEROGENEOUS PROGRAM COMMUNICATION (CAIMAN)

4.1 Introduction

Accelerators have become of great interest to the computing community in the last few years [16] [5]. For specific applications or computations, specialized accelerators can provide significant performance improvements. Using an accelerator, however, can require significant engineering effort and programmer expertise.

A major reason for the complexity of accelerator programming comes from the significant performance concerns that often appear when designing accelerator code. Intuitively, when an engineer finds that an accelerator is needed for some task, performance is likely a consideration. Since accelerators are often multithreaded, with specific requirements on data layouts or algorithm design. These constraints and performance needs have led to years of research and industrial work to optimize accelerator *kernels* (accelerator–only units of computation). However, there is a significant cost in the communication between devices, and specifically when this data communication occurs within the logic of the program.

This cost has been explored in several works in the form of automatic compiler optimizations [18] and code generation [11], which rely on either a compiler or learning model to generate communications or kernels based on some heuristic. Despite these advances, anecdotally programmers are still optimizing performance–critical communication by hand. This is (at least partially) due to the need for experimentation and profiling in performance–critical code, where a given solution may not perform optimally in all

heterogeneous arrangements and even within a given codebase.

We argue, without any meaningful way to prove or refute this claim, that heterogeneous programming is diverse enough and performance–critical enough, that such hand–optimized solutions will continue to be important for kernel and shader engineers. As a result, we propose a solution that raises the level of abstraction for performance engineering, but engineered so that the layer of abstraction being constructed is both transparent and decomposable. We start by making more concrete the specific challenges we have observed while hand–optimizing kernels, specifically separating performance and correctness, and managing a combinatorial explosion of kernel interactions.

4.1.1 Performance Experimentation

Our running example throughout this paper will be used to illustrate the performance trade–offs inherent in conditional logic and device synchronization. Specifically, we will be building the `select_sum` function, which takes in three arrays ‘v1‘, ‘v2‘, and ‘v3‘, and returns the sum of either ‘v2‘ or ‘v3‘ depending on the sign of ‘v1‘.

A naive `Rust` solution for this might look something like the following (for some fixed size of array):

```
select_sum(v1 : [i64; N1], v2 : [i64; N2], v3 : [i64; N3]) -> i64 {
  if sum(v1) < 0 {
    sum(v2)
  }
  else {
    sum(v3)
  }
}
```

Note that, in Rust, an expression without a `;` is equivalent to adding a `return`, so this code can be read as either returning the sum of `v2` or `v3`.

We might find that, for large arrays, this code is insufficiently performant in our setting. We might hope to be able to take an approach of the following Rust-like psuedocode, where we simply move our expensive `sum` operations onto the GPU:

```
select_sum_2(v1 : [i64; N1], v2 : [i64; N2], v3 : [i64; N3]) -> i64 {
  if (sum_gpu(v1) < 0) {
    sum_gpu(v2)
  }
  else {
    sum_gpu(v3)
  }
}
```

Upon measuring the performance of this solution, however, we may find that this code isn't "fast enough", even if we are using a highly–optimized GPU implementation of `sum`. If we want to improve the performance of this code further, we have several options. We could move this entire code onto the GPU (noting, crucially, that the GPU tends to perform poorly with conditions), but another reasonable approach would be to calculate each `sum` in advance, to avoid bottlenecking the next GPU calculation behind the CPU condition. Concretely, in psuedocode, this implementation may look something like:

```
select_sum_3(v1 : [i64; N1], v2 : [i64; N2], v3 : [i64; N3]) -> i64 {
  sum1 = sum_gpu(v1);
  sum2 = sum_gpu(v2);
  sum3 = sum_gpu(v3);
  if (sum1 < 0) {
    sum2
  }
  else {
    sum3
  }
}
```

To complicate this function even more, however, we may have a case where `v1` is a smaller array than `v2` or `v3`, and so sending `v1` to the GPU is introducing unnecessary overhead in our computation. For this special case, we may need to consider the following function variation (even within our same program):

```
// For N1 << N2,N3
select_sum_4(v1 : [i64; N1], v2 : [i64; N2], v3 : [i64; N3]) -> i64 {
  if (sum(v1) < 0) {
    sum_gpu(v2)
  }
```

```
  else {
    sum_gpu(v3)
  }
}
```

Note that, in this variation, we may find it optimal to revert to the original idea of having `sum_gpu` inside of the condition rather than before, since we are no longer blocking on communication between devices.

It is important to note that every one of these steps we described may depend on measuring performance in a specific setting, with specific array sizes, and even within specific functions. The particular functions we wrote may or may not be optimal – an analysis of our timing each variation of this function can be found in 4.8 (including blindly moving the entire `select_sum` function to the GPU), but we can only show these results for a particular setting. The takeaway, then, should be that a programmer may need to try each of these variations to find a performant solution, and may need to maintain multiple such functions (such as both `select_sum_3` and `select_sum_4`) in the same program, despite these functions doing the same computation!

To make matters worse, we are ignoring even more decisions that can be made between these black–box calls to `sum_gpu`. The decision space here will be expanded on in **??**, which will also expand on why these functions may be much harder to write than what we have shown here. Before we continue, however, we need to explore another problem that arises from maintaining multiple definitions of `select_sum`.

4.1.2 Combinatoric Explosion

We have shown how to end up with several implementations of `select_sum`, where optimal performance may depend on setting-specific experimentation. Another key aspect

to optimizing heterogeneous performance, however, is optimizing for the context of surrounding functions. This is suggested by our reliance on the condition `N1 << N2,N3`, which implies that we ought to be careful about which "version" of `select_sum` to call based on information about the function calling `select_sum`. We can extend this logic the other direction, and find cases where we may prefer one implementation of `sum_gpu` over another for a particular choice of `select_sum`, which we can name `sum_gpu2`.

Concretely, let us define some `complicated_function`, where we have three calls (at various points) to `select_sum`. In–code, we can summarize this assumption as follows:

```
fn complicated_function(...) {
  select_sum(...);
  // ...
  select_sum(...);
  // ...
  select_sum(...);
}
```

We may find, however, that `complicated_function` is optimized for a particular arrangement of `select_sum` implementations; for instance, we might experiment with the following arrangement:

```
fn complicated_function(...) {
  select_sum_1(...);
  // ...
  select_sum_3(...);
  // ...
  select_sum_4(...);
}
```

This sort of experimentation can lead to an explosion of variations to try, where we may find that a given implementation of `sum_gpu` is preferable in a particular implementation of `select_sum` within this particular `complicated_function`, necessitating writing yet another variation of `select_sum`, which may require adjusting our other decisions, and so on and so forth. In this manner, we can quickly end up writing an exponential number of variations of functions, or, practically, write a complex system of macros,

comments, and/or implicit dependencies between code variants, with little in the way to help manage this design overhead.

4.1.3 Caiman Languages

To help address these problems with performance experimentation in heterogeneous programming, we introduce the Caiman language. The core goal of Caiman is to provide a separation between the definition and the implementation of a function, and to allow these function implementations to be used interchangeably in implementation (with some important type–level protections). In doing so, we make the following contributions in this work:

- We provide formal guarantees on multiple implementations for the same function *specification*
- We describe functions that are decomposable, in that a function implementation can be broken apart so that parts can be used by other implementations without duplicating the entire function, all without breaking Caiman's type–level guarantees of matching implementation to definition.
- We describe and implement a system within Caiman to harness type–directed program synthesis to help reduce the overhead of performance experimentation and to allow transparency into compiler optimizations.
- We provide a syntax and implementation for both human–writeable Caiman with the above guarantees, and a more precise Caiman IR for analysis and precise operational control.
- We demonstrate the performance tradeoffs for several synthetic WebGPU kernels, and show that a Caiman implementation of these kernels allows a similar

performance exploration.

4.2 Related Work

Scheduling Languages The philosophy of Caiman has overlap with scheduling languages such as Halide [17], Taco [10], and Exo [7]. The Caiman strategy of separating performance and implementation concerns is inspired by these sorts of scheduling languages, and indeed the direction for Caiman was initially written to be a scheduling language for heterogeneous interactions.

Despite this core inspiration, Caiman navigates the specific challenges of heterogeneous programming differently than the scheduling languages we are aware of. Most notably, Caiman emphasizes design for the combinatoric explosion of kernel design described in 4.1.2 and introduces type requirements for the multi–threading and condition logic present in heterogeneous decision making.

Heterogeneous Languages There have been a variety of heterogeneous programming languages in the last few years. We highlight three well–known languages/APIs for comparison, but acknowledge there are many more projects that we will not specifically address.

CUDA [4] is a computational language for the GPU that allows for compiling C++ code with a GPU program while providing and API “to use C++ as a high–level–language" [15]. A key feature of this approach, however, is to make the interface between CPU and GPU very thin (CUDA code strongly resembles a C++ function). This design makes writing code fairly straightforward, but can make extracting lower–level details for performance difficult.

OpenCL [8] is a widely used heterogeneous language solution, providing a language and API for a variety of heterogeneous settings. OpenCL provides a similar toolset to CUDA, but is designed to be more general–purpose and more controllable than GPU compute, at the expense of being somewhat difficult to program and require domain–specific expertise. SYCL [9] is an API built on top of (but separate to) OpenCL, and is meant to provide some of the higher–level benefits that can be seen with CUDA.

Vulkan [1] is a graphics–specific API for the GPU that is meant to give more control over exact GPU implementation and communication details. As with OpenCL, however, Vulkan can be difficult to use at scale due to the amount of domain–specific knowledge and optimization work needed. Despite this, the recent interest in Vulkan gives an indication of the value that exposing detailed interfaces can have for developing performant systems.

Caiman is designed as an alternative approach to these heterogeneous APIs, with an emphasis on transparency in the boundary between abstractions and performance–critical implementation details. A Caiman implementation could be compiled to any of these targets, or used as an alternative language/API.

4.3 Background

Before discussing the Caiman implementation, we first must examine some practical concerns when dealing with CPU–GPU communication. While Caiman could be used for many heterogeneous systems, our particular implementation is for CPU–GPU, and so we embed some of the terminology and concepts from the GPU in our Caiman discussion to make ideas concrete.

Our implementation of Caiman is specifically targeting WebGPU Shading Lan-

guage [14] (WGSL) through the Rust API WGPU [2]. Specific properties of our implementation are discussed in Section 4.7, but we will need to summarize WebGPU data movement to motivate details of the overall Caiman language design.

Our goal in this section is to provide an overview of how data is managed in WGPU as it relates to Caiman. We will be intentionally eliding many technical details of data movement in WGPU and more general graphics programming that exist in Caiman, but are unnecessary to understanding the Caiman typesystem and language design.

4.3.1 WGPU Data Submission

To move data from the CPU to the GPU, we distinguish four stages of work that need be done:

1. Setup: Request a memory location for writing to on the GPU, along with an `encoder` to manage data copying.
2. Encoding: Using the `encoder`, copy data from the CPU as needed, and set the program on the GPU that will actually be run.
3. Submission: submit that the data movement is finished and the GPU can begin processing work.
4. Await: Once the GPU is finished processing, the output data stored in memory can be safely read.

We name these steps as corresponding to the operations that Caiman will introduce, though similar naming schemes are used by WGPU and other GPU APIs. Each step must

be performed in sequence for a given operation, though data inputs during encoding and outputs after submission can be modified up until a signal is received for the next step.

A flag specifying when the next data transfer step or operation can occur is called a *fence*. Additionally, each of these operations are asynchronous between the CPU and the GPU, as the device waiting for data may be able to continue a previously defined operation until ready to read from a piece of memory. Such a location is called a *future*, indicating that the referred data can be read at some point in the future (if the computation terminates). It is common for futures to be implemented such that a device will *await* the future (hence our use of the Await step) until the data is written to the future and a flag is set, thus synchronizing data movement.

4.4 Practical Caiman

To explain how Caiman works, we will work through the example shown in the introduction, namely `select_sum`. For this illustration, we will be using code written the user-facing Caiman frontend, as opposed to the Caiman IR described in more detail in 4.7.

A Caiman program consists of a series of functions written across several languages: the three *specification* languages and the one *implementation* language. Informally, a function written in the specification languages describe semantic behaviors of the program, while a function written in the implementation language describes how that program is implemented on the host machine. This is the core mechanism by which Caiman separates semantic and performance concerns.

Additionally, a particular Caiman specification must deal with one of three distinct properties of the operation it is specifying: the value calculations, the timeline of the

events and synchronizations, and manipulations of existing memory. We give each of these “kinds" of specification functions a unique name, respectively they are the *value*, *timeline*, and *spatial* specifications. Any implementation must implement one of each kind of specification, though specifications may be used by multiple implementation functions.

We will start by showing how to implement `select_sum` in Caiman's specification language(s), before moving onto describing the choice of implementations Caiman provides. We will also examine some intuition of why the Caiman typechecker is able to validate an implementation against a given specification, though the formal model and proof will be deferred to section 4.5.

4.4.1 Value Specification

Caiman specifications are written as functions with a Rust-like header and declarative bodies. We emphasize declarative here to mean that declaration order does not matter (as we will see, we are essentially building a dependency graph). The complete syntax can be summarized as follows:

```
// function header, any number of arguments
spec_kind name(arg : type) -> return_type {
  var :- expression
  returns var
}
```

The specific expressions allowed for each variable declaration depend on the kind of specification. All specifications, however, share expressions for function calls and ternary conditional expression:

```
// function call
fn_name(expr)
// the usual notion of conditions
```

```
if cond then expr1 else expr2
```

The most intuitive Caiman specification to start with is often the *value* function, which describes what calculations are needed for each data value in the program. We will take a deep dive into describing the value specification for `select_sum` before returning to definitions for Caiman's other specification languages in section 4.4.3.

Our value specification for `select_sum` is as follows:

```
val select_sum(v1: [i64; N], v2: [i64; N], v3: [i64; N]) -> out: i64 {
    sum1 :- sum(v1)
    sum2 :- sum(v2)
    sum3 :- sum(v3)
    condition :- sum1 < 0
    result :- if condition then sum2 else sum3
    returns result
}
```

We have written this specification to be more verbose than needed for the sake of providing a more detailed examination of the semantics being used here. Many of these lines can be condensed or combined (we can just write `if sum(v1) < 0 ...`, for example).

Nevertheless, this declaration more-or-less mirrors our naive C code, notably replacing conditional `if/else` blocks with the ternary expression `if-then-else`. This distinction is more than just syntactic, Caiman's specification language is designed to have no internal control flow, as hinted at by the syntax.

Additionally, since Caiman's operations are unordered (we can freely move or combine each declaration here without changing the specification meaning), we do not have any requirement that these operations must be executed in the order written.

More precisely, this specification gives us constraints on what values this function must produce, but leaves the details up to the actual implementation. We can exactly enumerate these requirements as follows:

- `v1`, `v2`, and `v3` are all arrays of type `i64`, and this function produces data of type `i64` (this is the usual type constraint placed by a function header).
- The specification variables `sum1`, `sum2`, and `sum3` depend on `v1`, `v2`, and `v3`, respectively. For each of these, we must apply `sum` to produce our respective value.
- `condition` depends on having calculated `sum1` already *at some point*, and also have the constant `0` as a value.
- Similarly, `result` depends on having calculated `condition`, `sum1`, and `sum2`, thus requiring the calculations of all of their dependencies to have been done.
- Finally, `returns result` informs us that this function will return the `result` value.

Importantly, however, this specification provides formal requirements; as we will shortly in subsection 4.4.2, Caiman implementations of this specification which do not produce the specified values will fail to typecheck and will be rejected at compile time. Indeed, we can safely say that each specification variable becomes a type we can refer to while typechecking an implementation function. We must first take a short diversion into examining Caiman function definitions.

Function Equivalence

We need to take a short dive into how the `sum` function is being used in this `select_sum` specification. In Caiman, every function used in a specification must be a *function equivalence class*. Function equivalence classes consist of a name associated with one or more function definitions (which may be defined in Caiman or externally) that the programmer considers equivalent.

The most immediate use for function classes can be seen with the following definition of `sum` that we include in the `select_sum` file:

```
feq sum {
    extern(cpu) pure sum_cpu([i64; N]) -> i64
    extern(gpu) pure sum_gpu([i64; N]) -> i64
}
```

Here we are simply saying that we consider the `cpu` and `gpu` definitions of `sum` to be equivalent. The key reason for introducing these equivalence classes is to allow for multiple implementations of the same function to coexist in the same program, and to allow the user to fearlessly call any particular version in their implementation of some specification.

Note that `select_sum` must also be in such a function equivalence class – syntactically we name this equivalence class to be the same as our defined value function, in this case just `select_sum`. In this way, Caiman specification functions can call other specification functions without introducing assumptions about the implementations of those functions.

Importantly, however, Caiman does not attempt to prove this assertion or require any more annotation than exactly the type of the function provided here (the external implementations of `sum` could be buggy, and Caiman makes no claim about preventing this). Similarly, we also make no attempt to verify that an external function implementing a Caiman value specification will match the semantics of that specification, and leave this work to the user. Stylistically, this means that Caiman's guarantees and utility are strongest when a majority of the code being used in a Caiman program is written in Caiman rather than made external.

4.4.2 Implementation Language

With our value specification and function equivalence classes in hand, we can now implement a Caiman program for `select_sum`. As a reminder, all the code we have written so far provides typing information for our actual implementation here, and is not compiled into an executable program without this implementation.

Frontend Caiman implementations more-or-less resemble Rust code, with the addition of specifying which particular specification(s) they are implementing. As noted above, we will only focus on implementing our value specification for now. Our first implementation of `select_sum` is thus as follows:

```
fn select_sum_impl(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> i64 impls select_sum,... {
    if sum_cpu(v1) < 0 {
        sum_cpu(v2)
    }
    else {
        sum_cpu(v3)
    }
}
```

This is a valid Caiman implementation of this program, meant to show that in many cases, the programmer can essentially implement code similar to standard languages, where information can mostly be inferred. For understanding the typechecking work Caiman does on this program, however, it is perhaps more informative to show the explicit Caiman type annotations we can provide for such a program. When we hand-write these annotations, we produce the following (equivalent) program:

```
fn select_sum_impl(
  v1: [i64; N] @ val.v1,
  v2: [i64; N] @ val.v2,
  v3: [i64; N] @ val.v3)
  -> i64 @ val.out
  impls select_sum,... {
    let s1 : i64 @ val.sum1 = sum_cpu(v1);
    let cond : bool @ val.condition = s1 < 0;
    let res : @ val.result = if cond {
        let s2 : i64 @ val.sum2 = sum_cpu(v2);
```

```
        s2
    }
    else {
        let s3 : i64 @ val.sum3 = sum_cpu(v3);
        s3
    };
    res
}
```

Our focus in this rewrite is to expose the (baseline) types used by a Caiman implementation. Each variable in an implementation has 2 components: an raw datatype and three specification types (we are only showing the value specification type for now for simplicity). `s1`, for example, has a raw datatype of `i64`, and a (value) specification type of `val.sum1`.

The part of the type `node(val.sum1)` has two pieces, `val`, and `sum1`, with the following meanings:

- `val` states that the given type is a part of the value specification this function is implementing, `select_sum` in this case
- `sum1` states that we are working with the specific node within our value specification named `sum1`.

Crucially, this is part of the type of the implementation variable as much as the datatype `i64`, though the specification type can be erased when compiling Caiman code. In other words, if we instead wrote the (incorrect) annotation:

```
let s1 : i64 @ node(val.sum2) = sum(v1);
```

Then our code would fail to compile. Interestingly, this line alone would be enough to fail to compile, as we defined `val.sum2:-sum(v2)` in the specification, and since our implementation variable `v1` has type `input(val.v1)`, the types of `v2` and `v1` don't "match up".

An important observation is the type of `res`, which is derived from the `if-else` condition logic being applied. In essence, we consider control flow in Caiman's implementation language to have a return type, which can only be a `unit` if the associated specification type is a `unit`. We can show that this particular conditional logic works out, because the true case returns a result associated with the value type `v2`, while the false case returns a result associated with the value type `v3`. If we swapped the logic to instead produce `v3` in the true case and `v2` in the false case, we would have a type error.

With our types in hand, we can now rewrite our original code to reorder the sum operations to before the conditional logic, as we originally desired. The (type-inferred) code in Caiman can be simply written as follows:

```
fn select_sum_impl_2(v1, v2, v3)
  -> i64 impls select_sum,... {
    let s2 = sum_cpu(v2);
    let s3 = sum_cpu(v2);
    if sum_cpu(v1) < 0 {
        s2
    }
    else {
        s3
    }
}
```

A visibly typed version of this code can be found in the appendix. This implementation still typechecks and so maintains the value semantics of the `select_sum` specification – a proof of this claim will be given in 4.5. Importantly, by providing this function definition, we now have two implementations in the `select_sum` function equivalence class – these can be called elsewhere in Caiman by name with `select_sum_impl` or `select_sum_impl_2`.

We can now address the notion of actually using function equivalence in our implementation. Unfortunately, we are not able to yet write our call to `sum_gpu` (as alluded to in 4.4.1), since we will first need to specify more details about synchronizing to another

device (the GPU), and we have so far assumed that the CPU is our local host and so does not need synchronization.

To avoid introducing unnecessary complications to our straightforward equivalence relation, we will instead introduce another implementation of sum on the CPU, extending our equivalence class of `sum`:

```
feq sum {
    extern(cpu) pure sum_cpu_1([i64; N]) -> i64
    extern(cpu) pure sum_cpu_2([i64; N]) -> i64
    extern(gpu) pure sum_gpu([i64; N]) -> i64
}
```

This example is clearly synthetic and meant to be illustrative, but even in this case we could imagine a second definition of `sum_cpu`: specialized to be multi-threaded while our first definition is single-threaded.

We can freely use either definition within our `select_sum` function; for example, we could now write the following code, and the types of each value are identical to what we had before:

```
fn select_sum_impl_3(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> i64 impls select_sum,... {
    if sum_cpu_2(v1) < 0 {
        sum_cpu_1(v2)
    }
    else {
        sum_cpu_2(v3)
    }
}
```

This mechanism to freely interchange function definitions is core to Caiman's goal of separating performance decisions and specification, as discussed in Section **??**. Note that here we have introduced yet another definition to the `select_sum` function equivalence class to explore swapping out specific definitions. In this way, we can now avoid the usual combinatorial explosion of logic being checked by relying on the value specification to

maintain static consistency.

It is important now to address the limitations of Caiman's function equivalence classes, since they are designed to be very simple (and we hope transparent). Equivalence classes only apply to exactly function calls, and must either be filled by external functions (which are not checked by Caiman), or by implementations in Caiman.

This means that basic properties like arithmetic or logical equivalence are not reasoned about by Caiman unless explicitly declared. Concretely, for example, we consider `1+1` to be a distinct value from the constant `2`. We will examine potential extensions to Caiman's minimal equivalence system when we discuss future work in Section 4.9.1.

4.4.3 Timeline and Spatial Specifications

We have thus far focused on a vertical slice through Caiman with the value specification language. We can now take the intuition and ideas from this explanation to work through Caiman's other two specification languages.

It is worth noting up–front that Caiman's particular choice of specification languages is not rooted in any sort of experimentation, but instead an intuition of what is needed for the examples we care about – you could extend the ideas presented here to write an entirely different set of specification languages. Detailing this more abstract notion of what a specification language is, however, is out of scope of this write–up, and will be left to another, more theoretical paper.

We capture the intent of the value specification language in Caiman as reasoning about the data produced and used by our computation. When we are working with another device (such as a GPU), however, it is also important to be precise about what threading

and memory resources we interact with. More specifically, we would like to be able to break apart synchronization and memory resources across control flow, in much the same way as the value language allows us to (safely) reuse data and decompose computation on that data into carefully designed pieces.

To achieve this goal, we also introduce the *timeline* and *spatial* specification languages to Caiman. These languages follow the syntax and declarative approach used by the value specification, but with operations intended to capture the intent of synchronization and memory primitives used when scheduling GPU operations.

A Caiman synchronization mirrors that of the GLSL and WGPU submission processes, described in Section **??**. This process consists of exactly 3 events, described as a host managing the devices A and B, where device A is providing the data to run and device B is providing the computation and result:

1. The host requests an *encoding* location from device B, which is then given to device A.
2. After device A has finished encoding data, the host *submit*s that device B can begin computation.
3. Once device B has finished the computation, the host allows device A to *synchronize* and access the written data.

Caiman employs classic promise semantics for this model, described in 4.3.1. For the timeline specification, then, we introduce an `Event` type, which informally describes a point in time. Note that we implicitly also introduce subtypes for each step of this process to keep track of the logical flow, where each requirement is maintained structurally as being in dependency order. We also introduce the following three operations to match the stages described:

```
// Given an Event, starts an encoding on device B,
//   also produces a 'local' event associated with device A
//   along with a 'remote' event associated with device B
local, remote :- encode_event(e)
// Given a 'remote' event, begins a submission on device B
//   produces a 'submission' event
sub :- submission_event(remote)
// Given a 'submission' and 'local event,
//  produces a 'synchronization' event on device A
snc :- synchronization_event(local, sub)
```

We will explore a more detailed example of using the timeline language shortly in Section 4.4.4. First, however, we will summarize the (relatively simple) spatial specification language.

The spatial language is used primarily to specify memory behaviors to help with guarantees related to implemented loops or recursion. To this end, the spatial language provides a `BufferSpace` type, which defines a cluster of memory, along with exactly one operation, used to divide up this buffer space:

```
// splits up a given buffer space into n>=1 evenly sized pieces:
separate_buffer_space(buff, n);
// for example, we can split a buffer in half:
buff1, buff2 :- separate_buffer_space(buff, 2);
```

We will not being working through any examples that define a non-trivial spatial specification.

Having defined our specification functions, we can now fully state that an implementation in Caiman must be associated with exactly one of each kind of specification. A given implementation need not implement the entire specification of each, so long as the input and output types of that implementation are correct as stated. Note that this means that a call into a particular Caiman implementation of a function equivalence class may result in a series of calls (some of which may be also used by other Caiman implementations of that specification).

Further, each specification could be implemented by any number of Caiman implementation functions. Additionally, each specification function has its own equivalence class, which refers to any implementation of that specification (it is rare to use a function equivalence class other than a value specification or for external functions).

Note that a specification can always be trivial (simply taking in a single input and returning it), and so we can syntactically elide a specification in both an implementation header and a variable type. In these cases, we assume that the type of the implementation variable associated with the missing specification function refers to the trivial specification.

4.4.4 Select Sum on the GPU

With our timeline and spatial specification functions more carefully defined, we can now construct an implementation of `select_sum` that calls into the GPU to calculate either `sum(v2)` or `sum(v3)` (there are many other variations we can write, but we will start with this approach). In Rust-like psuedocode, what we are looking to implement resembles the following logic:

```
if sum_cpu(v1) < 0 {
    sum_gpu(v2)
} else {
    sum_gpu(v2)
}
```

First, we reiterate our value specification for ease of readability:

```
val select_sum(v1: [i64; N], v2: [i64; N], v3: [i64; N]) -> out: i64 {
    sum1 :- sum(v1)
    sum2 :- sum(v2)
    sum3 :- sum(v3)
    condition :- sum1 < 0
    result :- if condition then sum2 else sum3
    returns result
}
```

Second, we provide a timeline specification for a *single* synchronization (since we only call the gpu once in each branch, we need only to synchronize once). As a result, our specification will simply lay out our four stages in sequence:

```
tmln single_sync(in : Event) -> out : Event {
    local, remote :- encoding_event(in)
    sub :- submission_event(remote)
    sync :- synchronization_event(local)
    returns sync
}
```

Third, we provide a (trivial) spatial specification:

```
sptl trivial_spatial(b : BufferSpace) -> BufferSpace {
    returns b;
}
```

Now we can provide an exact implementation for this entire function. We will write this implementation without explicit value types, though an explicitly typed version can be found in Appendix B.1.1:

```
fn select_sum_impl_gpu(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> i64 impls select_sum,single_sync,trivial_spatial {
    if sum_cpu(v1) < 0 {
        let enc = encode-begin @ node(tmln.(local, remote)) { v2 } gpu;
        // copy the data from v2 into a gpu variable `v2_gpu`
        encode enc.copy[v2_gpu <- v2];
        // schedule the gpu to write the result
        //   of sum_gpu(v2) to a variable named s2_gpu
        encode enc.call[s2_gpu <- sum_gpu(v2) @ val.sum2];

        // submit the calculation to the gpu
        let fence = submit @ tmln.sub enc

        // await the result
        let gpu_data = await @ tmln.sync

        // return s2_gpu from the bundled result
        gpu_data.s2_gpu
    }
    else {
        // we apply similar logic here
        let enc = encode-begin @ node(tmln.(local, remote)) { v3 } gpu;
        encode enc.copy[v3_gpu <- v3];

        encode enc.call[s3_gpu <- sum_gpu(v3) @ val.sum3];
        let fence = submit @ tmln.sub enc
        let gpu_data = await @ tmln.sync
```

```
        gpu_data.s3_gpu
        sum_cpu(v3)
    }
}
```

This implementation more-or-less mirrors the first approach we examined, namely calculating the sum of `v2` and `v3` inside of the condition. Now that we have the logic for using the GPU, this approach of first calculating both before the condition will (hopefully) be more immediately appealing.

Interestingly, in this case, we can apply both operations sequentially on the GPU with only a single synchronization, but this requires an unsatisfying black–box solution where we write a `sum_2_arrays` type function. Alternatively, we could write a new timeline specification which allows for two synchronizations, and if our goal were to interleave our encodings and synchronizations, then we would write a new specification.

However, if we are comfortable synchronizing twice, Caiman does provide a clean solution in the form of breaking up our implementation to use the timeline specification twice, without needing to modify any of our specifications:

```
fn select_sum_impl_gpu_2(
  v1: [i64; N] @ val.v1,
  v2: [i64; N] @ val.v2,
  v3: [i64; N] @ val.v3)
  -> i64 @ val.result
  impls select_sum,single_sync,trivial_spatial {
    let enc = encode-begin @ node(tmln.(local, remote)) { v2 } gpu;
    encode enc.copy[v2_gpu <- v2];
    encode enc.call[s2_gpu <- sum_gpu(v2) @ val.sum2];
    let fence = submit @ tmln.sub enc
    let gpu_data = await @ tmln.sync
    let s2 = gpu_data.s2_gpu;

    select_sum_impl_gpu_2_inter()
}
fn select_sum_impl_gpu_2_inter(
  v1: [i64; N] @ val.v1,
  s2: i64 @ val.sum2,
  v3: [i64; N] @ val.v3)
  -> i64 @ val.result
  impls select_sum,single_sync,trivial_spatial {
    // similar logic to compute s3
```

```
    select_sum_impl_gpu_2_ret(v1, s2, s3)
}
fn select_sum_impl_gpu_2_ret(
  v1: [i64; N],
  s2: i64 @ val.sum2,
  s3: i64 @ val.sum2)
  -> i64 @ val.result
  impls select_sum,trivial_timeline,trivial_spatial {
    if sum_cpu(v1) < 0 {
        s2
    }
    else {
        s3
    }
}
```

This idea of breaking up a function to implement only part of a specification is crucial to using Caiman, as code written in this way can represent unfinished and can avoid duplicating computation. The reason this code is able to typecheck is precisely because of our explicit annotations on the arguments and return type of each function – otherwise, the program analysis needed often becomes intractable. Since Caiman can prove that the type boundaries of each function resolve, we need not worry about the potential complexity introduced between function calls.

Another important observation is that we can use a similar idea to break apart encoding and synchroniziation, allowing a programmer to first encode an operation, then continue doing work on the device, and finally synchronize. Note that Caiman's restrictions are such that we could not encode an operation and then never synchronize that operation (at least, within a Caiman program where we manipulate the timeline between pieces of control flow), since the types associated with the timeline and spatial specifications must match between each possible branch we could take.

Finally, it is worth acknowledging that this Caiman implementation has become quite complicated even for this simple operation. These implementations are, by design, rather detailed about each operation that is needed to synchronize information with the GPU (or

$$
\begin{aligned}
&\psi \in \mathbf{S} \\
&\phi \in \mathbf{C} \\
&\tau \in \mathbf{T} \\
&d ::= d_1.d_2 \mid \tau\ \psi \text{ :- } \phi(\psi_1) \mid \tau\ \psi \text{ :- if } \psi_1 \text{ then } \psi_2 \text{ else } \psi_3 \\
&p ::= d.\text{returns}\ \psi
\end{aligned}
$$

Figure 4.1: Spawnling Specification Syntax

$$
\begin{aligned}
&\psi \in \Psi \\
&\tau \in \mathbf{T} \\
&x \in \mathbf{V} \\
&f \in \mathbf{F} \\
&c ::= c_1; c_2 \mid x \leftarrow f_\psi(x_1) \mid \tau\ x \mid x \leftarrow \text{select}_\psi(x_1, x_2, x_3) \\
&p ::= c; \text{return}\ x
\end{aligned}
$$

Figure 4.2: Spawnling Implementation Syntax

just any other device). The intention behind this approach is to allow the user of Caiman to be as precise as possible with defining program implementation. That being said, having implementations be this dense seemingly defeats the purpose of the guarantees Caiman can make – if we require the programmer to both write careful specifications and implement those specifications to exacting detail, the effort to maintain this system can quickly get out of hand.

We will address this concern with a core piece of the Caiman design when we discuss in Section 4.6. First, however, we need to take a brief detour into examining a formal model for Caiman's type system, to provide formal backing and proof behind the core guarantee of Caiman's implementations actually being checked against the specification.

4.5 Formal Model

In this section, we will be describing a formal model of the Caiman language. We will use this to prove our assertion that the Caiman typechecker will ensure a given (typed) implementation has the same (observational) semantics as a specification it implements. We will also be more formally specifying these terms to precisely narrow our claim, and addressing the limitations of this presented approach.

To do this, we will start by describing a subset of the Caiman IR, a language called *Spawnling*. Spawnling has operations similar to the types and control flow of the Caiman IR, with the goal of describing exactly the type guarantees made by Caiman.

As with the Caiman IR, Spawnling has a specification and an implementation. Our primary goal will be to setup infrastructure for, but not prove, the following theorem:

Theorem 1. *Any well-typed Spawnling implementation program with types matching those of a well-typed Spawnling specification program must, for all inputs, either fail to terminate or produce equivalent values as that specification program.*

As an observation about this proof, the term *equivalent outputs* relies on a precise notion of equivalence and values in this context. Specifically, equivalence is defined exactly as any function that is a member of its function class (as described in subsection 4.4.1), while value refers to the series of operations that define some result. In this sense, we observe that the operation 1 + 1 is not considered to be the same value as the constant 2, unless an equivalence is explicitly stated.

The syntax for each of these Spawnling sub-languages are defined in figures 4.1 and 4.2, respectively. These syntax rely on external sets of distinct symbols, namely **T**, **S**, **P**, **V**, and **F**. These sets have no explicit meaning, but intuitively represent the

sets of types, specification variables, function classes, implementation variables, and function names, respectively. Function calls in Spawnling only allow a single argument – the extension of these proofs to multiple arguments is straightforward but mechanically irksome. We assume that the set of types **T** includes the type unit, which we use in the typechecking semantics. We also assume the set of specification variable symbols **S** and the set of function classes **C** includes **0**, a special character that cannot be assigned to and indicates a lack of dependence.

A Spawnling specification program additionally requires 2 contexts, 1 dynamic and 1 static:

- $\Psi : \mathbf{S} \mapsto \mathbf{T} \bigtimes (\mathbf{C} \bigtimes (\mathbf{S} \bigtimes \mathbf{S} \bigtimes \mathbf{S}))$, which is dynamic and maps from a specification variable to both its type and its dependencies. A variable depends either on a function or no function (semantically represented with $\top$), and has exactly 0, 1, or 3 dependencies (0 for an input, 1 for a function call, or 3 for a condition).
- $\Phi : \mathbf{C} \mapsto \mathbf{T} \bigtimes \mathbf{T}$, which is static and maps from function classes to the input type and the output type of that function.

A Spawnling implementation program, on the other hand, similarly requires only 1 dynamic context, but requires 3 constant global contexts (where Δ and Ψ are built to be derived from the specification associated with the current implementation). These contexts are as follows:

- $\Gamma : \mathbf{V} \mapsto \mathbf{T} \bigtimes (\mathbf{S} \cup \top)$, the sole dynamic context, which maps from an implementation pointer to the type of the data being referred to by that pointer. This context also includes the associated value if one has been written to the location pointed at by this variable.

$$\frac{\Psi, \Psi' \vdash d_1 \qquad \Psi', \Psi'' \vdash d_2}{\Psi, \Psi'' \vdash d_1.d_2} \qquad\qquad \frac{\Psi, \Psi' \vdash d_2 \qquad \Psi', \Psi'' \vdash d_1}{\Psi, \Psi'' \vdash d_1.d_2}$$

$$\frac{\psi \notin \Psi \qquad \Psi(\psi_1) = (\tau_1, \_) \qquad \Phi(\phi) = (\tau_1, \tau_2)}{\Psi, \Psi \sqcup [\psi \mapsto (\tau_2, (\phi, (\psi_1, \mathbf{0}, \mathbf{0})))] \vdash \tau\, \psi :\text{-}\, \phi(\psi_1)}$$

$$\frac{\psi \notin \Psi \qquad \psi_1 \in \Psi \qquad \Psi(\psi_2) = (\tau, \_) \qquad \Psi(\psi_3) = (\tau, \_)}{\Psi, \Psi \sqcup [\psi \mapsto (\tau, (\top, (\psi_1, \psi_2, \psi_3)))] \vdash \text{if}\, \psi_1\, \text{then}\, \psi_2\, \text{else}\, \psi_3}$$

$$\frac{\Psi, \Psi' \vdash d \qquad \Psi'(\psi) = (\tau_{\text{out}}, \_)}{\Psi, \Psi' \vdash d.\text{return}\, \psi}$$

Figure 4.3: Spawnling Specification Typing Judgment

- $\Psi : \mathbf{S} \mapsto \mathbf{T} \bigtimes (\mathbf{C} \bigtimes (\mathbf{S} \bigtimes \mathbf{S} \bigtimes \mathbf{S}))$, which must be derived from the result of applying the typechecking rules to a well-typed Spawnling specification. When typechecking an implementation, however, Ψ is global and constant.
- $\Phi : \mathbf{C} \mapsto \mathbf{T} \bigtimes \mathbf{T}$, which is the same as with the specification program
- $\Xi : \mathbf{F} \mapsto \mathbf{C}$, which maps from functions to their owning function class

Finally, both Spawnling programs must specify an output type. For the Spawnling specification program, we denote this as τ_{out}, while for the Spawnling implementation program, we denote this as ψ_{out}. Note that we assume that a function being a member of Φ implies that it is a well–typed function.

4.5.1 Typing Semantics

We describe the type semantics for a Spawnling specification in Figure 4.3. The first two rules, applied to $.$, indicate that Spawnling specification operations are unordered; we can either apply the typing judgment to the left or the right of a $.$ operation and use the

$$\frac{\Gamma(x) = (\tau, \psi)}{\Gamma, \Gamma[x/(\tau, \top)]} \qquad \frac{\Gamma, \Gamma' \vdash c_1 \qquad \Gamma', \Gamma'' \vdash c_2}{\Gamma, \Gamma'' \vdash c_1; c_2}$$

$$\frac{\Psi(\psi) = (\_, (\phi, (\psi_1, \mathbf{0}, \mathbf{0}))) \qquad \Xi(f) = \phi \qquad \Gamma(x_1) = (\tau, \psi_1)}{\Gamma, \Gamma[x/(\tau, \psi)] \vdash x \leftarrow f_\psi(x_1)}$$

$$\frac{x \notin \Gamma}{\Gamma, \Gamma[x/(\tau, \top)] \vdash \tau\ x} \qquad \frac{\Gamma(x_i) = (\_, \psi_i) \qquad \Psi(\psi) = (\_, (\top, (\psi_1, \psi_2, \psi_3)))}{\Gamma, \Gamma[x/(\tau, \psi)] \vdash x \leftarrow \text{select}_\psi(x_1, x_2, x_3)}$$

$$\frac{\Gamma, \Gamma' \vdash c, \Gamma'(x) = \psi_{\text{out}}}{\Gamma, \Gamma' \vdash c; \text{return}\, x}$$

Figure 4.4: Spawnling Implementation Typing Judgment

result in the other. Similarly, we terminate a specification program with `return`, and require that the expected type τ_{out}. Note also that we do not allow multiple definitions of a specification variable through requiring they are not already included in Ψ.

The most interesting (if dense) operations in this semantics are the types to the operations $\phi(\psi_1)$ and $\text{if}\, \psi_1\, \text{then}\, \psi_2\, \text{else}\, \psi_3$. In both of these cases, we require that types "match up", where we use _ to indicate that the remainder of our stored type is irrelevant. The type of what is added to the context Ψ is necessarily a bit messy, requiring that we retain information about the structure of the specification type for the implementation. Specifically, for function calls, we produce the returned type, the function being called, and the input to the function being called. Similarly, for conditions, we produce the (matching) type of each branch, no function being called, and the set of three variable used in the condition.

We then describe the type semantics for the Spawnling implementation in Figure 4.4. The semantics for these implementations match those of an imperative language in most cases, including allowing reassignment of variables, sequential operations, and

declarations of variables to fix a type. Since a Spawnling implementation must terminate with a `return`, a Spawnling implementation can only typecheck if we produce a result of type ψ_{out}.

A Spawnling implementation is not explicitly tied to a particular specification outside of the definition of Ψ being used. We note that the context Ψ being used is fixed throughout our typechecking of an implementation. We use the specification typechecking results stored in Ψ to validate that the argument x_1 to the function call f_ψ matches the type of our specification requirement, namely ψ_1. The rule for conditions is similar, instead requiring that each branch match with the associated branch of the specification.

4.6 Explication

As observed at the end of Section 4.4, hand–writing Caiman implementations are painful and can practically lead to significant barriers to experimentation. Specifically, the code itself is dense and can be hard to navigate, but we have found it to be anecdotally difficult to write even with compiler support. Indeed, it feels as though the work done when implementing a Caiman program is both essentially the same work as writing a Caiman specification and can obscure the performance characteristics being explored.

The entire design of Caiman, however, is such that this task of writing an implementation from a specification can be automated with annotation. More specifically, we can apply a form of type–directed program synthesis, as seen in several works .

We use explication to describe the task of synthesizing a Caiman implementation from a Caiman specification in the presence of programmer–written implementation requirements. To make this statement more concrete, we will start by working from

example of how this looks in the Caiman frontend.

The author notes that at the time of this writing, the programs shown in this section cannot be written as–is in the Caiman frontend, due entirely to minor missing engineering work on AST transformations. Approximately equivalent programs can be written explicitly in the Caiman IR (described in 4.6.2, and the programs are included in Appendix B.1.3), but I believe it is important to note this limitation in our actual implementation. Consequently, more annotations may need to be included in the finished Caiman frontend (to help guide the explicator) than are shown here.

4.6.1 Using Explication

We will start by reiterating our usual value specification for `select_sum`:

```
val select_sum(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> [out : i64] {
    sum1 :- sum(v1)
    sum2 :- sum(v2)
    sum3 :- sum(v3)
    condition :- sum1 < 0
    result :- if condition then sum2 else sum3
    returns result
}
```

Along with a trivial timeline and spatial specification, this is enough to write a fully explicated implementation of `select_sum`:

```
fn select_sum_expl(
  v1: [i64; N] @ val.v1,
  v2: [i64; N] @ val.v2,
  v3: [i64; N] @ val.v3
) -> i64 : val.out
  impls select_sum,trivial_time,trivial_space{
  ???
}
```

With explication, we still need to provide a header (arguments and return types) and the

specifications to implement. When we write `???`, however, we are semantically saying that any type-safe code can be inserted to replace this *multi–line hole* in the program.

The Caiman *explicator* can use then types we have given this function to deduce some program that matches these types. The explicator will not, however, be able to use any other specifications than those given, meaning that it is often the case that no such program exists. For instance, if we had excluded `v1` from our arguments, the `select_sum` specification provides no way to construct `v1`, and so the explicator would fail to produce a solution.

Importantly, however, the explicator is somewhat possible to control outside of this strictly open `???` hole we just defined. More precisely, we can summarize our explicator requirements with three rules:

1. The synthesized code must typecheck according to our given inputs and outputs, and as defined by our specification.
2. For a fixed implementation and fixed specifications if the explicator produces a program, the explicator must always produce the same program. That is, the explicator must be deterministic for the resulting program.
3. Any operations provided by the programmer must be used in the order they were written. Additionally, new operations can only be added where we have a multiline hole (`???`)

The last of these rules, the ordering of operations, is the key piece of the Caiman explicator design that allows for performance manipulation without adjusting the specification (or writing many implementation functions). We will describe precisely what this rule entails after exposing more of Caiman's implementation in Section 4.6.2. For now, as a concrete

example, let us restrict the explicator for `select_sum` to ensure that the sum of `v2` and `v3` must be calculated before the condition:

```
fn select_sum_expl_2(
  v1: [i64; N] @ val.v1,
  v2: [i64; N] @ val.v2,
  v3: [i64; N] @ val.v3
) -> i64 : val.out
  impls select_sum,trivial_time,trivial_space{
  .. = sum(v2);
  .. = sum(v3);
  ???;
}
```

The key addition to this code is that we have stated that both `sum(v2)` and `sum(v3)` must be computed and stored (in the unnamed variable `..`) before any other computation can be done. With respect to our explication rules, the order here is important – by having the `???` after these computations, we've required that the explicator can't insert code that we don't expect before these operations.

Of course, the more important reason to introduce to explicator is to help with exploring the more detailed (and complicated) device communication space. Fortunately, we have introduced enough of Caiman design to use the explicator to help us with writing more complicated with the GPU, taking advantage of the bounds imposed by the timeline specification along with our current value specification. We first write the timeline specification as shown in Section 4.4.4 (restated here to help with readability):

```
tmln single_sync(in : Event) -> out : Event {
    local, remote :- encoding_event(in)
    sub :- submission_event(remote)
    sync :- synchronization_event(local)
    returns sync
}
```

And now we can write our implementation, providing some structure to ensure we achieve our desired behavior of only encoding and syncing with the GPU once depending on our condition:

```
fn select_sum_expl_gpu(
  v1: [i64; N] @ val.v1,
  v2: [i64; N] @ [val.v2, tmln.in],
  v3: [i64; N] @ [val.v3, tmln.in])
  -> i64 @ [val.out, tlmn.out]
  impls select_sum,single_sync,trivial_spatial {
    if sum(v1) < 0 {
        ???
    }
    else {
        ???
    }
}
```

Since we have only allowed work within each branch of the condition, we can be confident that we will compute `v2` and `v3` respectively. Additionally, our restriction that our inputs are associated with `tmln.in` and our outputs are associated with `tmln.out` means that we must have synced with a device to produce these results. Strictly speaking, Caiman does not guarantee which device is used, as in our current explicator implementation we only allow synchronization with the GPU; this would, however, not be difficult to change.

A slightly more interesting observation is that eliding the condition here (`if sum(v1) < 0`) will result in the same program in this particular case, due to the way we have written our timeline specification. By both specifying we can only synchronize with the GPU once, only providing a definition for a single `sum` kernel, and requiring that we produce a result with `tmln.out`, we have restricted our implementation sufficiently that there is only one viable solution. This sort of constraint can be difficult to observe in practice, however, motivating the use of partial implementations such as the condition shown here in cases where the programmer explicitly desires this structure.

The other important consequence of observing the extent by which Caiman specifications restrict implementation is how many specification and partial implementations arrangements result in an overconstrained and unsolvable system. This can be seen as advantageous, in that attempting an explication can save a programmer significant effort

after reaching an impossible–to–proceed state, but the explicator may instead time out without resolution. Consequently, transparent messaging is important when working with the explicator, both information about what the explicator was able to produce and where it may have become stuck. The specific engineering around the explicator and user messaging will be explored in Section 4.7.1.

For the remainder of this section, we will examine the algorithm used by the Caiman explicator. Before we can do so, however, we need to expose a lower–level representation of Caiman code, the Caiman IR.

4.6.2 Caiman IR

So far, we have focused on the human–usable Caiman frontend, used to illustrate the intention of Caiman design and how this design can be realistically used by a programmer. When dealing with explication and precise performance characterization, however, it is helpful to expose the precise operations evaluated by Caiman in the form of an exact (dense) intermediate representation. It is worth noting that the Caiman IR is implemented to be interfaced with directly as needed, though ideally a programmer is able to avoid writing such code explicitly.

Caiman IR is written to be syntactically distinct from the frontend to avoid confusion, and has the following properties meant for analysis:

- A Caiman IR program consists of “funclets" (similar to basic blocks with named inputs and outputs) to contain operations. There is no control flow in a funclet, and calls between funclets use a style similar to CPS, or more precisely, to compiling without continuations [13].

- Caiman IR uses Static Single Assignment, so each name is unique. For ease of use, we allow `\%_` to be used to represent an anonymous variable (which may still be used by the Caiman explicator).
- Each operation consists of exactly instruction, and produces either nothing or exactly one result.
- There are minimal operations in Caiman IR (around 30 with our current implementation), with much of the work being done on external operations.

These properties are why we need to expose Caiman IR for explication. Specifically, we need to have a sense of Caiman's internal representation to describe explication concretely.

For the sake of examples, we will expose 7 operations in a Caiman IR implementation function. Note that there are distinct operations used in the Caiman IR specifications; these are similar enough to the specifications we have already seen that we will not need to show them here (complete Caiman IR examples can be found in Appendix B.1.2). For these operations we are exposing, we are ignoring some implementation cruft in the Caiman IR that is unnecessary for understanding the programs, but will appear in the examples provided. These operations are described as follows:

- `alloc-temporary`, which takes in a place (local, cpu, or gpu) and an exposed datatype (such as `i64`), and produces a memory location that can be written to. Caiman IR does not distinguish between stack and heap allocations. We expose `alloc-temporary` as the only allocation mechanism for simplicity, in cases where we need to return the reference rather than just data, we would use another operation instead.

- `read-ref`, which takes in a reference and returns the data referred to by this reference.
- `begin-encoding`, which takes in a place to encode to (cpu or gpu), an associated timeline specification node, and a list of memory locations to request for encoding, then returns a reference to an encoder.
- `local-do/encode-do`, which takes in an external function (such a `sum`), a specification node to match with, arguments to the function, and, in the case of `encode-copy`, an encoder to operate with. With this, this operation writes the result of this function to call to a given allocation slot. Returns nothing.
- `local-copy/encode-copy`, which takes in a source reference and a destination reference (and an encoder in the case of encode–copy), and copies the data referred to by the source to the location referred to by the destination. Returns nothing.
- `submit`, which takes in an encoder and a timeline specification node to associate with, and submits the encoded jobs to the associated device. Returns a reference to the resulting fence to later synchronize on.
- `sync-fence`, which takes in a fence and a timeline specification node, and requires synchronizing on that fence before continuing. Returns nothing.

Additionally, the Caiman IR defines several possible tail edges for funclet implementations to provide type information and to manage continuations when interacting with control flow. We will expose 2 of these tail edges, though to avoid excessive detail, we will defer discussion of the continuation logic to Section 4.7:

- `return` takes a single argument as the value to be returned from this funclet. Note that, to pass typechecking, the given data must match all of the specifications required by the return type of this funclet.

- `schedule-select` takes in a `i32` or `i64` value to use to select a function, an ordered array of funclets (with matching arguments and return types) to select, the specification node(s) this selection is associated with, the arguments to the selected function, and a function to join with per continuation–passing style. The return type of each selected funclet must match arguments of the joined funclet, and the return type of the join funclet must match the return type of this funclet.

Worked Example

With our definitions in mind, we can now show an example assembly implementation for `select_sum`. We will be showing the hand-translated equivalent to the implementation first shown in 4.4:

```
fn select_sum_impl(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> i64 impls select_sum,... {
    if sum_cpu(v1) < 0 {
        sum_cpu(v2)
    }
    else {
        sum_cpu(v3)
    }
}
```

Since we will need to refer to specification nodes for our assembly implementation substantially, we will also rewrite our usual value specification to make these names concrete:

```
val select_sum(v1: [i64; N], v2: [i64; N], v3: [i64; N])
  -> [out : i64] {
    sum1 :- sum(v1)
    sum2 :- sum(v2)
    sum3 :- sum(v3)
    condition :- sum1 < 0
    result :- if condition then sum2 else sum3
    returns result
}
```

Since Caiman IR funclets do not have control flow, we must start by defining a function

to compute the condition of the `if` statement, and then calling funclets that represent the left and right branches with `schedule-select`. Note that we will be using `...` to indicate elided syntax cruft that is irrelevant for this example (as opposed to the `???` or `?` Caiman syntax for an explication hole):

```
schedule[...] %select_sum_main<...>
(%v1 : val.%v1 ...,
 %v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%out ...] {
    // We are eliding irrelevent allocation details
    // Note we allocate locally rather than on the CPU
    //   in doing so, we can avoid synchronizing on the CPU
    //   which may, in general, be another device
    %ref = alloc-temporary local [...] i64;

    // write the result of sum(v1) to ref
    local-do-external %sum val.%sum1(%v1) -> %ref;

    // read the result into a local variable
    %sum1 = read-ref i64 %ref;

    // We will not show how to get the constant 0
    %zero = ...

    // write the result of sum1 < 0 to ref
    local-do-external %lt val.%condition(%sum1, %zero) -> %ref;

    // read the result into a local variable
    %cond = read-ref i64 %ref;

    // we will not show how we are computing join
    %join = ...

    // we will be writing the left and right branches shortly
    // note that we must pass %v2 and %v3 to both
    //   since their input types must "match up"
    // note also that we specify %result rather than %out
    //   since the if/then/else in the specification gives %result
    schedule-select %cond [%select-sum-left, %select-sum-right]
      [val.%result, ...] (%v2, %v3) %join;
}
```

Syntactically, it is worth noting that we use % to indicate a Caiman IR defined variable to help with distinguishing from the frontend and to visually distinguish between operations and variables. We have been careful to note each detail we are eliding, which are either irrelevant or out of scope for this writing – most notably the calculation of the join operation, the complete type syntax Caiman IR uses, and additional qualifiers needed

for allocation.

This code provides a precise representation of the line-by-line operations required for Caiman's analysis and compilation. Fortunately, the left and right branches of this condition are simpler than this selection program, at least when we run locally:

```
schedule[...] %select_sum_left<...>
(%v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%sum2 ...] {
    %ref = alloc-temporary local [...] i64;
    local-do-external %sum val.%sum2(%v2) -> %ref;
    %res = read-ref i64 %ref;
    return %res;
}

schedule[...] %select_sum_right<...>
(%v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%sum3 ...] {
    %ref = alloc-temporary local [...] i64;
    local-do-external %sum val.%sum3(%v3) -> %ref;
    %res = read-ref i64 %ref;
    return %res;
}
```

These funclets are nearly identical, except for their return type and computation. We observe that these functions can be used in selection because we have annotated our selection with `val.%result`, which is sufficient for the typechecker to agree so long as each branch return type "lines up" with its respective branch in the specification. Finally, then, we can write the (trivial) funclet to join the original selection:

```
schedule[...] %select_sum_ret<...>
(%v1 : val.%v1 ...,
 %v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%out ...] {
    return %res;
}
```

Morally, this function mirrors the return that we have in the specification, and explicitly provides the connection between `val.%result` and `val.%out`.

Extending to the GPU

With our overview of Caiman IR in hand, we will briefly examine how to interface with the GPU. This code will be quite similar to our high–level Caiman frontend example shown in 4.4.4, but is useful for demonstrating our introduced Caiman IR GPU operations.

Specifically, we can modify our funclet `select_sum_left`, (note that similarly we could modify `select_sum_right`). Concretely, we will be implementing the usual single–synchronization timeline function:

```
tmln single_sync(in : Event) -> out : Event {
    local, remote :- encoding_event(in)
    sub :- submission_event(remote)
    sync :- synchronization_event(local)
    returns sync
}
```

Concretely, our Caiman IR implementation is as follows:

```
schedule[...] %select_sum_left<...>
(%v2 : val.%v2 time.%in ...,
 %v3 : val.%v3 time.%in ...)
  -> [%out : val.%sum2 time.%out ...] {
    // Create a reference to the gpu memory
    //   we will be able to write to
    // Note that this memory is not available
    //   until we have an encoding
    %v2_gpu = alloc-temporary gpu [...] [i64; N];
    %res_gpu = alloc-temporary gpu [...] i64;
    %res_local = alloc-temporary local [...] i64;

    // Begin an encoding with references
    //   for our input (v2_gpu) and output (res_gpu)
    %enc = begin-encoding gpu time.%enc [%v2_gpu, %res_gpu] [];

    // We can now write to this memory location
    // Note that this copy does not have
    //   an associated specification operation
    // In other words, if we had some
    //   other mechanism to have a value
    //   of type v2 in the right place at the
    //   right time, we could use that instead
    encode-copy %enc %v2 -> %v2_gpu;

    // Encode running the `sum` operation after our copy
    // Writes the result to our earlier encoded memory location
    encode-do %enc %sum_gpu val.%sum2(%v2_gpu) -> %res_gpu;
```

```
    // Submite the operations and recover a fence
    %fnc = submit %enc time.%sub;

    // Synchronize locally against the GPU result
    sync-fence %fnc time.%snc;

    // Copy the result back to the local allocation
    // We can do this because we have synchronized
    local-copy %res_gpu -> %res_local;
    %res = read-ref i64 %res_local;
    return %res;
}
```

In this GPU implementation, we hint at type rules to manage memory allocations when encoding, submitting, and synchronizing. We will include engineering for these interactions in our explicator implementation, but the type–safety guarantees we would intuitively aim for are left to a future work.

4.6.3 Explication of Caiman IR

Before diving into this session, we note that, at the time of this writing, our actual explicator implementation for this example at this time of writing is missing a couple of implementation details that are described here. Specifically `???` is not carefully tested, and `begin_encoding` is not fully implemented.

With our more-exact representation of Caiman IR in hand, we can provide a precise definition of Caiman's explication syntax. Caiman allows exactly two synctatic "holes", defined as follows:

- `?`, which indicates a specific component of a specific operation can be explicated
- `???`, which indicates that the explicator can introduce any number of operations (including introducing new funclets for control flow).

As before, `???` also means that the explicator can only introduce operations within the block outlined by operations surrounding this multi–line hole. We will summarize these requirements through a partial explication for `select_sum_main`:

```
schedule[...] %select_sum_main<...>
(%v1 : val.%v1 ...,
 %v2 : val.%v2 ...,
 %v3 : val.%v3 ...)
  -> [%out : val.%out ...] {
    // do any setup needed to work out the condition
    ???;

    // require that we use exactly %lt,
    // and that we must work out val.%condition in this funclet
    // note that we explicitly state
    //  a requirement of two arguments here
    //  though we do not say what these arguments are
    local-do-external %lt val.%condition(?, ?) -> ?;

    // do any setup neeeded for the schedule-select
    ???;

    // We must branch into the funclets
    //   select-sum-left and select-sum-right
    // We also must have this select be associated with val.%result
    // otherwise, the explicator may do as it would please
    schedule-select ? [%select-sum-left, %select-sum-right]
      [val.%result, ...] (?, ?) ?;
}
```

There are many ways we could write this partial implementation such that no solution exists, including providing the incorrect number of required arguments to `%lt`. Given how large the space of impossibilities are, the explicator makes no attempt to prove that no program solution exists, but instead will fail to terminate. Additionally, the explicator does not fully typecheck existing code, and so only ensures that the code produced is type–safe assuming a correctly typed partial implementation. Details of how our engineered explicator with the Caiman typechecker are expanded more in 4.7.1.

The Caiman explicator also works with the operations needed to encode and synchronize on another device. For example, the following funclet header is sufficient to produce the implementation described in 4.6.2:

```
schedule[...] %select_sum_left<...>
```

```
(%v2 : val.%v2 time.%in ...,
 %v3 : val.%v3 time.%in ...)
  -> [%out : val.%sum2 time.%out ...] {
    ???;
    encode-do ? %sum_gpu val.%sum2(?) -> ?;
    ???;
}
```

Note that we must include this specific `encode-do` to be sure that we are actually running the operation on the GPU. Otherwise, the explicator could very reasonably choose to run this operation locally, and simply apply a no–op to the GPU to progress to `time.%out`. In other words, the particular operation we expect to apply to compute `sum2` would otherwise be left as an implementation detail of the explicator.

4.6.4 Core Algorithm

We now have the context to define an explication algorithm. Generally speaking, our approach will be to make arbitrary (type–safe) decisions for the first node of the funclet, recurse to see if we produce a valid explication result, and then try the next decision we could make. Conceptually this algorithm resembles a depth first search through the decisions we could make, and if we find a type–safe result, we simply return this result.

The algorithm used to explicate a node of a Caiman IR funclet is more precisely described in Algorithm 1. We start this recursion with an empty list of solved nodes, all of our funclet inputs in the set of unused nodes, and our funclet definition. Note that the funclet definition we provide is intended as a static reference, and is not modified as we recurse.

If *NodeExplication* returns **NONE**, then no solution was found. If this function returns a pair that includes a list of any requests for multiline explication, then we continue the recursion with the next solution for the first node. Otherwise, we accept the valid

explication solution as the list of explicated nodes (along with an explicated tailedge, which is elided in the result for readability).

unusedNodes deserves special mention, as this set holds information about all programmer–given nodes (starting with the funclet inputs). Per our requirements on explication, we must use all programmer–given nodes at some point in the function, so we use this set to track if our current explication attempt fulfills this constraint. Multi–line explication nodes are exempt, so we need not add them to the set as we recurse.

We now need to specify several of the functions used in this algorithmic approach:

- *TailEdgeExplication* works out if, for the given funclet types and current explicated nodes, we are able to both produce a result of the correct type from the funclet and use all remaining unused nodes. If we can fulfill both requirements, this function returns the resulting explicated tail edge and any multi–line explication requests. Otherwise, this function returns **NONE**.
- *ValidSolutions* takes in our current node selections and our current node index (along with our usual funclet information), and determines all the ways this node could be explicated given what we already have, called solutions. Note that, abstractly, this function is exhaustive – for example, if we have an allocation, this function will iterate through every possible type that could be allocated. To maintain algorithmic determinism, the order we iterate through solutions must be deterministic. In practice, we provide heuristics on this order of solutions to make it more likely we will find an explication result quickly.
- *GetUsedNodes* takes in our current solution, and returns the set of nodes it uses.
- *GetMultilineRequests* takes in our current solution and list of nodes we have already found, and returns a list of nodes that must be added to a multiline explication.

Note that, since we must use every programmer node to complete explication, these newly generated nodes will not “conflict" with existing user choices. We will expand what requests are needed for a given node shortly.

A key piece of the Caiman explication algorithm is writing down what a given explication solution will need to request for multiline explication. This detail is why Caiman's type system is setup such that the specification(s) provided are sufficient to narrow down such a request. Indeed, in practice, there is often only one operation (or chain of operations) that will satisfy a given operation under our specification type constraints.

To make this decision making more concrete, we will specify the exact operations required for a few of our defined Caiman IR operations. Note that the precision of Caiman IR is crucial here, allowing us to have a precise (finite) number of holes we can fill for a given operation. Note also that we use the term *selects* as a stand in for iterating through every possible fill choice and fixing that decision for the remaining requirements:

- `alloc-temporary` has no requirements, and, as presented, iterates through every place and every supported datatype.
- `local-do-external` first selects a value specification node (say `val.\%sum1`) and an implementation in that class. This operation then requires all arguments to that specification value be already calculated, and that there is an allocation for our return type as a write target.
- `read-ref` selects a datatype, and then requires that there be an allocation of that datatype.
- `submit` selects a timeline specification node, and requires that we have an encoder at a time where we can submit.

While we have not enumerated every implementation operation we have presented, it should be straightforward to derive the remaining operations. One operation that bears special mention, however, is `begin-encoding`, which allows us to encode any number of operations as an argument. The algorithmic solution is to consider the powerset of all allocations and iterate through this set. In practice, we treat `begin-encoding` as a special case of multiline explication, where instead of allowing arbitrary operations, we can request that an allocation be included in the encoding.

While Caiman does currently use the algorithm we have described for explication, the critical takeaway is that there *exists* an algorithm that satisfies our requirements from earlier. Practically, we expect that pushing parts of this explication algorithm to a solver or synthesis tool may help with explicator performance and scalability. This being said, a qualitative analysis of using Caiman's explication approach will be discussed in Section 4.8.1

4.7 Caiman Engineering

We have built an implementation of the Caiman language and Caiman IR as a toolchain we will refer to as the Caiman compiler. For our implementation[1], we take in Caiman or Caiman IR code, apply explication and semantic analysis, and emit Rust code to connect Rust (on the CPU) and WebGPU (on the GPU) through the WGPU interface.

Our choice to emit Rust and WGPU is somewhat arbitrary – indeed, practically we would recommend that future work on a Caiman compiler instead target an IR rather than Rust directly, and a lower–level interface like Vulkan. Nevertheless, our implementation of Caiman shows how we can construct graphics–focused and dynamic CPU/GPU code,

[1]https://github.com/cucapra/caiman

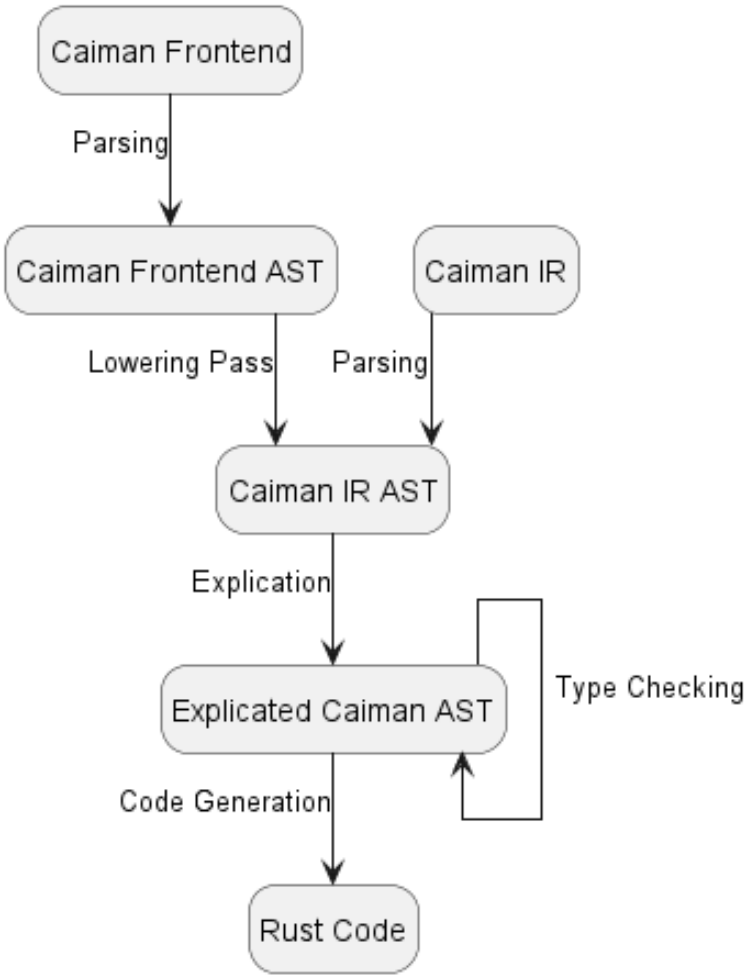

Figure 4.5: Structure of the Caiman Compiler

more concretely described in Section 4.8.

4.7.1 Compiler Infrastructure

The stages of work done by the Caiman compiler can be seen in Figure **??**. For the remainder of this section, we will examine some of the technical details for each stage of compilation, working from the top of the compiler (parsing) to the bottom (code generation).

Caiman to Caiman IR

Caiman has two distinct representations in the frontend and the IR, resulting in two distinct Abstract Syntax Trees (ASTs). The transformation from the Caiman frontend AST to the Caiman IR AST is straightforward (if non–trivial to build), needing to transform the specification and implementation logic to the single-operation and SSA style of the Caiman IR. Specifications (value, timeline, and spatial) in Caiman have no control flow, so translating to the similar representation in Caiman IR requires only (simply) reducing expressions.

Translating a Caiman implementation to Caiman IR is somewhat more involved, however, as we need to account for the control flow of the function (with respect to the specification) as well as provide information about the allocation of resources in computation. For conditions, function calls, and recursion, we translated directly for the call site based on the types of the values being used. We do not currently attempt to implement loops in Caiman. Allocations are calculated based on value use – in principle, it would improve explicator flexibility to move allocation logic to the explication search, but we have not practically engineered this approach.

Additionally, both specifications and implementations include standard Hindley-Milner type inference [6]. For the implementation language, this type inference is primarily useful for control flow headers (which cannot be explicated), and values not inferred are instead pushed to the explicator for solving.

Caiman IR explication and typechecking

To lower the Caiman IR for typechecking, we erase variable and function names and validate program structural assumptions. We then explicate the program as described

in Section 4.6. To help make debugging explicated Caiman code manageable, the explicator maintains names throughout, requiring the explicator to retain metadata about the transformed program. Additionally, at this stage, the explicator performs minimal typechecking to perform explication. Importantly, however, the explicator makes no guarantee that existing code typechecks, and only intends to produce explicated code that will typecheck if existing code will typecheck. We do not validate this claim, and instead leave formalizing this connection to future work.

After explication, we guarantee that the explicator either failed to terminate (due to a type error or due to timing out), or every hole in the AST representation of the Caiman IR is filled. Practically, we represent this state with a duplicate AST with no hole variants. The typechecker is then able to walk through each function and validate our requirement that each implementation function does the work promised by the implemented specifications, as stated in Section 4.5. Additionally, the typechecking pass here validates properties of the allocations and data between funclets, a significant detail of Caiman engineering that is out of scope of this writing.

Code Generation

Finally, the resulting Caiman IR program is emitted as Rust code, which can then be linked to, compiled, and run. This Rust code notably includes calls to externally-defined Rust functions and setting up WebGPU API allocations and function calls. At this stage, code generation makes no attempt to optimize the resulting code (such as inlining code), leaving these additional optimizations to the underlying compiler.

Caiman code generation currently does not support directly emitting WebGPU code, and as a result treats each WebGPU function as a black–box call. It is possible to fuse

these calls, or even to emit WebGPU directly in Caiman, but these additions are not yet implemented. This detail can lead to significant performance loss, and certainly would require an implementation to use Caiman in a practical setting.

4.8 Results

At the time of this writing, quantitative results of Caiman are minimal and not worth reporting. We have, however, implemented around 100 synthetic examples of Caiman programs, which can give qualitative insight into the current implementation of Caiman described. Intended future performance measurements will be summarized in Section 4.9.1.

The programs implemented in Caiman are intended primarily for testing, but provide some idea of the flexibility of our produced implementation. Notable examples of Caiman programs include the following, the full code of which is included in Appendix B.1.4:

- Nested conditional logic
- Recursive functions
- WebGPU function calls, including conditional logic and recursive calls
- Fixed-sized arrays
- `select_sum`, as described in Section 4.4

In developing these examples, we have found that Caiman's type system is remarkably flexible, though it takes time to become comfortable with implementing Caiman code. Specifically, implementing a Caiman specification requires some amount of care when breaking up the specification and managing types of each operation in cases where type inference is insufficient.

This being said, the Caiman type system has caught dozens of errors we have made while implementing these functions, including mistakes in managing control flow when calling into the GPU. We have also found that, practically, Caiman programs can be written essentially as we might expect despite the specificity of Caiman's type requirements.

The main issue we have had implementing programs is that some implementation rewrites are restricted under Caiman's type system without changing the specification, which is precisely what we would like to avoid. A notable example of this weakness is the technical detail described in Section 4.7.1, where the Caiman semantics is insufficient for merging two calls that we know to be equivalent to a specific single function call.

We have not yet implemented functions that significantly and scientifically test Caiman's resulting runtime performance. Qualitatively, all Caiman programs we have written run nearly instantly, but without comparison to directly written WebGPU programs, this result is, bluntly, insufficient to draw any real conclusions.

4.8.1 Explication

Explication is the most likely stage of Caiman compilation to produce interesting timing results, both in terms of the time of compilation and the runtime of these programs. We have implemented synthetic examples of explication, with at most 30 lines of code for a given explication implementation. Notably, however, at the time of this writing, we have not fully implemented the `???` statement described in Section 4.6.

At the time of this writing, qualitatively, the explication examples we have written compile and run nearly instantly. We expect this to be similarly instant for an implementation written entirely with a `???` statement. Where we expect to see the pathologically

slow examples are in functions that use a mix of `???` statements and a small number of restricted expressions, where the greedy search done by the explicator becomes less likely to quickly find a solution. We expect these cases to scale exponentially with the number of lines in a given funclet.

4.9 Conclusion

We have argued that writing performant heterogeneous operations often requires experimentation with multiple function implementations of the same underlying operation. We have seen how this gap between intent and implementation can lead to programmer mistakes and maintaining a combinatorial explosion of function implementations.

With Caiman, we have shown a mechanism by which we can separate the specification and implementation of such heterogeneous operations. We have shown how we can represent control flow and recursive operations in a specification and how to verify that an implementation associated with a specification will calculate the data as we expect.

We have also examined how we can use explication to automate generation of these implementations, and specifically in such a way that this automation can be broken down. More concretely, we have examined how automating the generation of programs can be controlled through providing both a specification and precise line–by–line generation requirements.

Finally, we have described our specific implementation for CPU/GPU code to generate Rust code to interface with WGPU. We have also shown the structure of the Caiman compiler, and have hinted how code generation can be replaced to work in a variety of heterogeneous settings.

4.9.1 Future Work

Most immediately, timing information about explication and generated Caiman code will be gathered and reported. Such data will allow for more concrete and narrow arguments as to the efficacy of our Caiman implementation and an examination of the potential scalability of explication.

An important observation about the explication results we have observed is that the explicator does not attempt to reason about the interactions between multiple funclets. Introducing general control flow to the explicator would *likely* reduce performance (and expand engineering needs) substantially. This being said, implementing explication in such a way as to generate functions to fill in control flow seems relatively achievable, and would help substantially in implementation details that currently need to be written by hand.

The remainder of this section will discuss potential directions for Caiman that have been discussed, but not implemented.

Caiman's implementation in Rust with WebGPU is fairly narrow and can be difficult to manage. Implementing Caiman to directly generate code for an IR (such as Cranelift [3] or LLVM [12]) would improve Caiman's usability and transparency tremendously. Additionally, targeting Vulkan would make more practical sense in exposing more details in the Caiman communication model.

It would be very interesting to explore an implementation of Caiman targeting either compute (with CUDA or OpenCL), or another device entirely. Caiman seems directly useful for multiple–device code architectures, or for using another non–CPU host (such as GPU–driven logic). Caiman was also designed with FPGAs and network chips in mind, though without an implementation, how effective this design transfers is completely

untested.

Rewriting kernels entirely in Caiman seems impractical in many cases, but it also seems conceptually straightforward to write a translator from a subset of C or CUDA to a Caiman specification. Such tooling could have clear practical advantages, allowing a user to harness the type–level power of Caiman without needing to rewrite every specification by hand. The untested downside of this approach could be that Caiman's type system would be too rigid to allow such a straightforward translation, and this would require practical experimentation.

Finally, Caiman explication code can be difficult to debug, particularly in cases where the explicator is directed incorrectly and generates a bunch of strangely–named nonsense that exposes some mistake in the specification. The explication step we have described and implemented is transparent to the compiler, however, and there is an interesting potential HCI challenge in exploring an approach to improve Caiman error messages or provide some sort of visual or IDE tooling to examine explicated Caiman code.

```
Data: A list of solved Caiman IR Nodes, a set of nodes we have yet to use, and
      the current Caiman IR funclet.
Result: Either (a list of explicated nodes, a list of requests for multi–line
        explication to be done) or NONE if no solution was found
[Node] solvedNodes
{Node} unusedNodes
Funclet funclet
begin
    index ← Length(filledNodes)
    nodes ← Nodes(funclet)
                          // base case (we assume 0-indexing)
    if index = Length(nodes) then
        return TailEdgeExplication(filledNodes, unusedNodes, funclet)
    end
    currentNode ← nodes[index]
    if IsMultilineHole(currentNode) then
        result ← NodeExplication(filledNodes + currentNode, unusedNodes,
         funclet)
        if IsNone(result) then
            return NONE
        end
        else
            (explicated, requests) ← result
            return (requests + explicated, [])
        end
    end
    for solution ∈ ValidSolutions(nodes, index, funclet) do
        usedNodes ← GetUsedNodes(solution)
        solutionRequests ← GetMultilineRequests(solution, solvedNodes)
                    // copy nodes, with our solution added
        newNodes ← nodes + solution
        newUnusedNodes ← (unusedNodes ∪ currentNode) \ usedNodes
        result ← NodeExplication(newNodes, unusedNodes, funclet)
        if IsNone(result) then
            (explicated, requests) ← result
                        // prepend our solution and requests
            return (solution + explicated, solutionRequests + requests)
        end
    end
    return NONE
end
```

Algorithm 1: NodeExplication

CHAPTER 5

CONCLUSION AND FUTURE DIRECTIONS

We have examined how domain–specific language design for graphics programming can help programmers with managing correctness and performance considerations. With Gator, we have shown how we can define a specialized type system for reasoning about geometric types, and how these type definitions can provide information for multi–operation transformations. Additionally, we have shown how Gator exposed a surprising challenge in reasoning about equivalence of these transformations through the work on commutative diagrams.

With the Caiman language, on the other hand, we examine how definitions of correctness and equality with respect to a specification can help explore performant solutions. Additionally, we have shown how we can provide decomposable program implementations for heterogeneous code from these fixed specifications through explication.

APPENDIX A

GATOR APPENDIX

A.1 Gator's Syntax

A.1.1 Type System

$$c \in \text{constants}$$

$$x \in \text{variables}$$

$$p \in \text{primitives}$$

$$t \in \text{types}$$

$$\tau ::= \text{unit} \mid \top_p \mid \bot_p \mid t$$

$$f \in \text{function names}$$

$$e ::= x \mid c \mid f(e_1, e_2) \mid x \text{ as! } \tau \mid x \text{ in } \tau$$

$$C ::= \tau\, x = e \mid e$$

$$P = C; P \mid \epsilon$$

A.2 Gator's static rules

$$\frac{\tau_1 \leq \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2} \qquad \frac{\mathrm{X}(c) = p}{\Gamma \vdash c : \bot_p} \qquad \frac{\Gamma(v) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma(v) = \tau}{\Gamma \vdash \tau\ x = e : \text{unit}}$$

$$\frac{\Gamma \vdash C : \tau_1 \qquad \Gamma \vdash P : \tau_2}{\Gamma \vdash C; P : \text{unit}} \qquad \frac{}{\Gamma \vdash \epsilon : \text{unit}} \qquad \frac{\Gamma \vdash e : \top_p \qquad \tau \leq \top_p}{\Gamma \vdash e \text{ as! } \tau : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \mathrm{P}(\tau_1, \tau_2) = f}{\Gamma \vdash e \text{ in } \tau_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, \vdash e_2 : \tau_2 \qquad \Phi(f, \tau_1, \tau_2) = \tau_3}{\Gamma \vdash f(e_1, e_2) : \tau_3}$$

A.3 Ordering rules

$$\frac{}{\tau \leq \tau} \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \qquad \frac{}{\tau \leq \text{unit}} \qquad \frac{}{\bot_p \leq \top_p} \qquad \frac{\tau \leq \top_p}{\bot_p \leq \tau}$$

A.4 Target language grammar

Hatchling is an abstraction over sound imperative languages.

It has an *operation context* Ξ that maps operator names and types to their implementation. Ξ : (operator names), $\tau_1, \tau_2 \rightarrow$ type.

We use the symbol $||\vdash$ for judgment in the target language

A.4.1 Hatchling's syntax

$$c \in \text{constants}$$

$$x \in \text{variables}$$

$$p \in \text{primitives}$$

$$t \in \text{types}$$

$$\tau ::= \text{unit} \mid \top_p \mid \bot_p$$

$$o \in \text{operation names}$$

$$e ::= x \mid c \mid o(e_1, e_2)$$

$$c ::= \tau\, x = e \mid e$$

$$P = c; P \mid \epsilon$$

A.4.2 Hatchling's static semantics

Subsumption

$$\frac{\Gamma \mid\!\vdash e : \tau, \Gamma}{\Gamma \mid\!\vdash e : \text{unit}, \Gamma}\ \text{SUBSUMPTION}$$

Constants and variable declarations

$$\frac{}{\Gamma \mid\!\vdash () : \text{unit}}\ UNIT$$

$$\frac{\Gamma \vdash \Gamma(v) = \tau}{\Gamma \vdash v : \tau}\ VAR$$

$$\frac{\Gamma \vdash X(c) = p}{\Gamma \vdash c : \bot_p}\ PRIMITIVE$$

$$\frac{\Gamma \mid\!\vdash e : \tau}{\Gamma \mid\!\vdash \tau x := e : \text{unit}, x \mapsto \tau}\ DECL$$

Operations

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, \vdash e_2 : \tau_2 \qquad \Gamma \vdash \Xi(o, \tau_1, \tau_2) = \tau_3}{\Gamma \vdash o(e_1, e_2) : \tau_3}\ OP$$

A.5 Translation

A.5.1 Literals

$$[\![c]\!] \triangleq c$$

A.5.2 Types

$$[\![T]\!] \triangleq \top_p$$

Where we know $\top_n$ is always defined because of the structure imposed on a well formed Δ.

$$[\![\top_p]\!] \triangleq \top_p$$

$$[\![\bot_p]\!] \triangleq \top_p$$

$$[\![\text{unit}]\!] \triangleq \text{unit}$$

A.5.3 Expressions

$$
\begin{aligned}
[\![c]\!]_\Gamma &\triangleq c \\
[\![x]\!]_\Gamma &\triangleq x \\
[\![\tau\ x := e]\!]_\Gamma &\triangleq [\![\tau]\!]\ x := [\![e]\!]_\Gamma \\
[\![e \text{ as! } \tau]\!]_\Gamma &\triangleq [\![e]\!]_\Gamma \\
[\![e \text{ in } \tau_2]\!]_\Gamma &\triangleq [\![f(e)]\!]_\Gamma && \text{where } \Gamma \vdash e : \tau_1 \text{ and } f = \mathrm{P}(e, \tau_1, \tau_2) \\
[\![f(e_1, e_2)]\!]_\Gamma &\triangleq f'(e_1, e_2) && \text{where } \Gamma \vdash e : \tau_1, \Gamma \vdash e : \tau_2, \text{ and } f' = \Psi(f, e_1, e_2, \tau_1, \tau_2) \\
[\![\epsilon]\!]_\Gamma &\triangleq \epsilon \\
[\![C; P]\!]_\Gamma &\triangleq [\![C]\!]_\Gamma; [\![P]\!]_\Gamma
\end{aligned}
$$

A.5.4

Additionally, we define the translation of Γ and Φ; in the target language, we replace every τ in the range of Γ and Φ with its translation to get $[\![\Gamma]\!]$ and $[\![\Phi]\!]$. This is a different translation function than the one previously discussed, but for convenience we will use the same notation $[\![\Gamma]\!]$ and $[\![\Phi]\!]$.

Lemma 1.

$$
\frac{[\![\Gamma \vdash \Gamma(x) : \tau]\!]_\Gamma}{[\![\Gamma]\!] \mid\!\vdash [\![\Gamma]\!]([\![x]\!]_\Gamma) : [\![\tau]\!]}
$$

$$
\frac{[\![\Gamma \vdash \Phi(f, \tau_1, \tau_2) : \tau_3]\!]_\Gamma}{[\![\Gamma]\!] \mid\!\vdash [\![\Gamma]\!]([\![\Phi(f, \tau_1, \tau_2)]\!]_\Gamma) : [\![\tau]\!]}
$$

Proof. Follows from the definitions of $[\![\Gamma]\!]$ and $[\![\Phi]\!]$. □

A.6 Proof that if source type checks then translation type checks

We use structural induction over Gator commands to show that any translation under a valid Δ type checks. In the process of the induction we use the theorem that if a Gator expression type checks then its translation does too. This is also shown using structural induction over all Gator expressions.

Theorem 2. *Given any Gator expression e that type checks under some* Γ *to produce* τ*, we prove that its translation* $[\![e]\!]$ *also type checks under* $[\![\Gamma]\!]$ *to produce* $[\![\tau]\!]$.

Theorem 3. *Given any Gator program P that type checks under some* Γ *to produce* τ*, we prove that its translation* $[\![e]\!]$ *also type checks under* $[\![\Gamma]\!]$ *to produce* $[\![\tau]\!]$.

Below, I.H. = Inductive Hypothesis. Sometimes the inductive hypothesis is used over a general τ. This implicitly excludes the special case of the polymorphic type.
We use the inversion of the type checking rules in Gator; if a statement in Gator type checks then we know that the premise of the corresponding type checking rule is true. Using this rule will be marked as T.C.

We split the proof into cases, one for each of the typing judgement rules in Gator.

A.6.1 Expressions Type Check

We present the most interesting case, for primitives. The other cases follow vry similarly.

Primitive rule

$$\dfrac{\dfrac{}{\llbracket\Gamma\rrbracket \mid\!\vdash c : \top_p, \llbracket\Gamma\rrbracket}\ \text{PRIM} \qquad \llbracket\top_p\rrbracket \triangleq \top_p \qquad \llbracket c\rrbracket \triangleq c}{\llbracket\Gamma\rrbracket \mid\!\vdash \llbracket c\rrbracket : \llbracket\top_p\rrbracket}\ \text{SUBST}$$

Functions type check because of the constraints on a well formed Ψ.

$$\dfrac{\dfrac{\Gamma \vdash f(e_1 : \tau_1, e_2 : \tau_2) : \tau_3}{\Phi(f, \tau_1, \tau_2) = \tau_3} \qquad \llbracket\Gamma\rrbracket \mid\!\vdash \Psi(f, e_1, e_2, \tau_1, \tau_2) = \llbracket\Phi(f, \tau_1, \tau_2)\rrbracket}{\llbracket\Gamma\rrbracket \mid\!\vdash \llbracket f(e_1 : \tau_1, e_2 : \tau_2)\rrbracket : \llbracket\tau_3\rrbracket}\ \text{SUBST}$$

A.6.2 Commands Type Check

A.6.3 Declaration

One case is presented here, and other follows suit along the same lines.

Declaration

$$\dfrac{\dfrac{\dfrac{\dfrac{\llbracket\Gamma \vdash \tau x := e : \text{unit}, x \mapsto \tau\rrbracket}{\llbracket\Gamma \vdash e : \tau\rrbracket}\ \text{T.C.}}{\llbracket\Gamma\rrbracket \mid\!\vdash \llbracket e\rrbracket : \llbracket\tau\rrbracket}\ \text{I.H.}}{\llbracket\Gamma\rrbracket \mid\!\vdash \llbracket\tau\rrbracket\llbracket x\rrbracket := \llbracket e\rrbracket : \text{unit}, \llbracket x\rrbracket \mapsto \llbracket\tau\rrbracket}\ \text{DECL} \qquad \llbracket\tau x := e\rrbracket \triangleq \ldots}{\llbracket\Gamma\rrbracket \mid\!\vdash \llbracket\tau x := e\rrbracket : \text{unit}, \llbracket\Gamma\rrbracket, \llbracket x\rrbracket \mapsto \llbracket\tau\rrbracket \qquad \llbracket\Gamma, x \mapsto \tau\rrbracket \triangleq \llbracket\Gamma\rrbracket, \llbracket x\rrbracket \mapsto \llbracket\tau\rrbracket\llbracket\text{unit}\rrbracket \triangleq \text{unit}}\ \text{SUBST}$$

$$\llbracket\Gamma\rrbracket \mid\!\vdash \llbracket\tau x := e\rrbracket : \llbracket\text{unit}\rrbracket, \llbracket\Gamma, x \mapsto \tau\rrbracket$$

APPENDIX B

CAIMAN APPENDIX

B.1 Caiman Examples

B.1.1 Typed Select Sum

```
#version 0.1.0

sptl space(s: BufferSpace) -> BufferSpace { returns s }

extern(cpu) pure sum(i64) -> i64

val select_sum(v1: i64, v2: i64, v3: i64) -> out: i64 {
    sum1 :- sum(v1)
    sum2 :- sum(v2)
    sum3 :- sum(v3)
    condition :- sum1 < 0
    result :- sum2 if condition else sum3
    returns result
}

tmln time(in: Event) -> out: Event {
    local, remote :- encode_event(in)
    sub :- submit_event(remote)
    sync :- sync_event(local, sub)
    returns sync
}

fn select_sum_impl(
    v1: i64 @ [node(val.v1)],
    v2: i64 @ [node(val.v2), input(tmln.in)],
    v3: i64 @ [node(val.v3), input(tmln.in)]
)
-> i64 @ [node(val.out),  node(tmln.in)] impls select_sum,time,space {
    let sum1 : i64 @ node(val.sum1) = sum(v1);
    let condition : bool @ node(val.condition) = sum1 < 0;
    if @ [node(val.result), input(tmln.in)] condition {
        let sum2 : i64 @ [node(val.sum2), input(tmln.in)] = sum(v2);
        sum2
    }
    else {
        let sum3 : i64 @ [node(val.sum3), input(tmln.in)] = sum(v3);
        sum3
    }
}

pipeline main { select_sum_impl }
```

B.1.2 Caiman IR Examples

First, we show an example of straightforward series of external operations:

```
version 0.0.2

// implements:
// fn foo() -> i64 {
//     x = 3;
//     y = 2;
//     n1 = x + y;
//     n2 = x + n1;
//     return n1 + n2;
// }

ffi i64;
event %event0;
buffer_space %buffspace;
native_value %i64 : i64;

function @add(%i64, %i64) -> %i64;
function @main() -> %i64;

external-cpu-pure[impl @add] %add(i64, i64) -> i64;

value[impl default @main] %foo() -> [%out: %i64] {
    %res_t = call @add(%n1, %n2); // 8 + 5 = 13
    %res = extract %res_t 0;
    %n2_t = call @add(%x, %n1); // 3 + 5 = 8
    %n2 = extract %n2_t 0;
    %n1_t = call @add(%x, %y); // 3 + 2 = 5
    %n1 = extract %n1_t 0;
    %x = constant %i64 3;
    %y = constant %i64 2;
    return %res;
}

timeline %time(%e : %event0) -> %event0 {
    return %e;
}

spatial %space(%bs : %buffspace) -> %buffspace {
    return %bs;
}

schedule[value val = %foo, timeline time = %time, spatial space = %space]
%foo_main<time-usable, time-usable>() ->
[%out : val.%out-usable %i64] {
    %x_loc = alloc-temporary local [] i64;
    %y_loc = alloc-temporary local [] i64;
    %n1_loc = alloc-temporary local [] i64;
    %n2_loc = alloc-temporary local [] i64;
    %res_loc = alloc-temporary local [] i64;

    local-do-builtin val.%x() -> %x_loc;
    local-do-builtin val.%y() -> %y_loc;
```

```
    %x = read-ref i64 %x_loc;
    %y = read-ref i64 %y_loc;
    local-do-external %add val.%n1_t(%x, %y) -> %n1_loc;
    %n1 = read-ref i64 %n1_loc;
    local-do-external %add val.%n2_t(%x, %n1) -> %n2_loc;
    %n2 = read-ref i64 %n2_loc;
    local-do-external %add val.%res_t(%n1, %n2) -> %res_loc;

    %res_val = read-ref i64 %res_loc;
    return %res_val;
}

pipeline "main" = %foo_main;
```

Second, we show an example involving a call to the GPU:

```
version 0.0.2

// Performs a single computation on the GPU,
// encoding, submitting, and waiting all in one funclet.

ffi i32;
native_value %i32 : i32;
ref %i32l : i32-local<flags=[map_read, map_write,
  copy_src, copy_dst, storage]>;
ref %i32g : i32-gpu<flags=[map_read, map_write,
  copy_src, copy_dst, storage]>;
event %event0;
buffer %buffer_gpu : gpu<flags = [map_read, map_write,
  copy_src, copy_dst, storage], alignment_bits = 0, byte_size = 1024>;
buffer_space %buff_space;

function @simple(%i32) -> %i32;
function @foo(%i32) -> %i32;

external-gpu[impl @simple] %simple(%x : i32) -> [%out : i32]
{
    path : "gpu_external.comp",
    entry : "main",
    dimensionality : 3,
    resource {
        group : 0,
        binding : 0,
        input : %x
    },
    resource {
        group : 0,
        binding : 1,
        output : %out
    }
}

value[impl @foo] %foo(%x : %i32) -> %i32 {
    %c = constant %i32 1;
    %y_t = call @simple(%c, %c, %c, %x);
    %y = extract %y_t 0;
    return %y;
```

```
}

timeline %foo_time(%e : %event0) -> [%out: %event0] {
    %enc = encoding-event %e [];
    %enc1 = extract %enc 0;
    %enc2 = extract %enc 1;
    %sub = submission-event %enc2;
    %snc = synchronization-event %enc1 %sub;
    return %snc;
}

spatial %foo_space(%bs : %buff_space) -> %buff_space {
    return %bs;
}

schedule[value val = %foo,
  timeline time = %foo_time, spatial space = %foo_space]
%foo_main<time.%e-usable, time.%out-usable>
(%x_loc : val.%x-usable %i32l)
-> [%out : val.%y-usable %i32] {
    %c_loc = alloc-temporary local [storage] i32;
    %x_gpu = alloc-temporary gpu [storage, copy_dst] i32;
    %y_gpu = alloc-temporary gpu [storage, map_read] i32;
    %y_loc = alloc-temporary local [map_write] i32;

    local-do-builtin val.%c() -> %c_loc;
    %enc = begin-encoding gpu time.%enc [%x_gpu, %y_gpu] [];
    encode-copy %enc %x_loc -> %x_gpu;
    %c = read-ref i32 %c_loc;
    encode-do %enc %simple val.%y_t(%c, %c, %c, %x_gpu) -> %y_gpu;

    %fnc = submit %enc time.%sub;
    sync-fence %fnc time.%snc;

    local-copy %y_gpu -> %y_loc;
    %result = read-ref i32 %y_loc;
    return %result;
}

pipeline "main" = %foo_main;
```

which includes associated WGSL code:

```
#version 450

layout(set = 0, binding = 0) readonly buffer Input_0 {
    int field_0;
} input_0;

layout(set = 0, binding = 1) buffer Output_0 {
    int field_0;
} output_0;

layout(local_size_x = 1, local_size_y = 1, local_size_z = 1) in;
void main()
{
    output_0.field_0 = input_0.field_0 + 1;
```

```
}
```

B.1.3 Caiman IR Explicated Implementation

Partially example in the current working version of Caiman. Note that this example is not "maximally" explicated, but will hopefully illustrate what can be written as–is.

```
version 0.0.2

ffi i64;
ffi array<i64, 4>;
ref %i64l : i64-local<flags=[]>;
event %event0;
buffer_space %buffspace;
native_value %array4 : array<i64, 4>;
native_value %i64 : i64;

function @sum(%array4) -> %i64;
function @is_negative(%i64) -> %i64;
function @select_sum(%array4) -> %i64;

external-cpu-pure[impl @sum] %sum(array<i64, 4>) -> i64;
external-cpu-pure[impl @is_negative] %is_negative(i64) -> i64;

value[impl default @select_sum] %main(
  %v1 : %array4, %v2 : %array4, %v3 : %array4) -> [%out : %i64] {
    %res = select %sel %left %right;
    return %res;

    %s_t = call @sum(%v1);
    %s = extract %s_t 0;
    %sel_t = call @is_negative(%s);
    %sel = extract %sel_t 0;

    %left_t = call @sum(%v2);
    %left = extract %left_t 0;
    %right_t = call @sum(%v3);
    %right = extract %right_t 0;
}

timeline %time(%e : %event0) -> %event0 {
    return %e;
}

spatial %space(%bs : %buffspace) -> %buffspace {
    return %bs;
}

schedule[value val = %main,
  timeline time = %time, spatial space = %space]
%select_sum_head<time-usable, time-usable>(
    %v1 : val.%v1-usable time-usable space-usable %array4,
```

```
    %v2 : val.%v2-usable time-usable space-usable %array4,
    %v3 : val.%v3-usable time-usable space-usable %array4
) ->
    val.%out-usable time-usable space-usable %i64
{
    %_ = alloc-temporary local [] i64;

    local-do-external %sum ? ? -> ?;
    %_ = read-ref i64 ?;

    local-do-external %is_negative ? ? -> ?;
    %sel = read-ref i64 ?;

    %djoin = default-join;
    %join = inline-join %select_sum_join [] %djoin;

    schedule-select %sel
        [%select_sum_left, %select_sum_right]
        [val.%res, time, space]
        (%v2, %v3)
        %join;
}

schedule[value val = %main,
  timeline time = %time, spatial space = %space]
%select_sum_left<time-usable, time-usable>(
    %v2 : phi-val.%v2-usable time-usable space-usable %array4,
    %v3 : phi-val.%v3-usable time-usable space-usable %array4
) ->
    val.%left-usable time-usable space-usable %i64
{
    %_ = alloc-temporary local [] i64;

    local-do-external %sum ? ? -> ?;
    %_ = read-ref ? ?;
    return ?;
}

schedule[value val = %main,
  timeline time = %time, spatial space = %space]
%select_sum_right<time-usable, time-usable>(
    %v2 : phi-val.%v2-usable time-usable space-usable %array4,
    %v3 : phi-val.%v3-usable time-usable space-usable %array4
) ->
    val.%right-usable time-usable space-usable %i64
{
    %_ = alloc-temporary local [] i64;

    local-do-external %sum val.%right_t ? -> ?;
    %_ = read-ref ? ?;
    return ?;
}

schedule[value val = %main,
  timeline time = %time, spatial space = %space]
%select_sum_join<time-usable, time-usable>(
    %res : val.%res-usable time-usable space-usable %i64
) ->
```

```
    val.%out-usable time-usable space-usable %i64
{
    return ?;
}

pipeline "main" = %select_sum_head;
```

B.1.4 Caiman Frontend Examples

The following examples all compile and run as expected in the current build of Caiman.

Nested conditional logic:

```
#version 0.1.0

tmln time(e: Event) -> Event { returns e }
sptl space(bs: BufferSpace) -> BufferSpace { returns bs }


val main() -> i64 {
    b :- true
    c :- false
    d :- false
    one :- 1
    two :- 2
    three :- 3
    four :- 4
    left :- one if b else two
    right :- three if c else four
    z :- left if d else right
    returns z
}

fn foo() -> i64
    impls main, time, space
{
    let d = false;
    var v;
    if d {
        let b = true;
        let two = 2;
        v = two;
        if b {
            let one = 1;
            v = one;
        }
    } else {
        let c = false;
        let four = 4;
        v = four;
        if c {
            let three = 3;
```

```
            v = three;
        }
    }
    v
}

pipeline main { foo }
```

Recursion:

```
#version 0.1.0

tmln time(e: Event) -> Event { returns e }
sptl space(s: BufferSpace) -> BufferSpace { returns s }

val gcd(a: i64, b: i64) -> i64 {
    returns a if b == 0
            else gcd(b, a % b)
}

fn gcd_impl(a: i64, b: i64) -> i64 impls gcd, time, space {
    if b == 0 {
        a
    } else {
        gcd_impl(b, a % b)
    }
}

pipeline main { gcd_impl }
```

WGSL Function Calls:

```
#version 0.1.0

extern(cpu) pure baz() -> i32
extern(cpu) pure bar() -> i32
extern(gpu) gpu_merge(x : i32, y: i32) -> out: i32
{
    path : "gpu_merge.comp",
    entry : "main",
    dimensions : 3,
    resource {
        group : 0,
        binding : 0,
        input : x
    },
    resource {
        group : 0,
        binding : 1,
        input : y
    },
    resource {
        group : 0,
        binding : 2,
        output : out
    }
```

```
}

val foo(c: bool) -> out: i32 {
    a :- baz()
    b :- bar()

    snd :- a if c else b

    r :- gpu_merge'<1, 1, 1>(a, snd)
    returns r
}

tmln foo_time(e: Event) -> out: Event {
    loc, rem :- encode_event(e)
    sub :- submit_event(rem)
    snc :- sync_event(loc, sub)
    returns snc
}

sptl foo_space(bs: BufferSpace) -> BufferSpace {
    returns bs
}

fn foo_impl(c: bool) -> i32 impls foo, foo_time, foo_space {
    @in {input: input(tmln.e), output: node(tmln.out) };
    let a = baz();
    let b = bar();
    let e = encode-begin @ node(tmln.(loc, rem))
      { a_gpu, b_gpu, y_gpu } gpu;
    encode e.copy[a_gpu <- a];
    if c {
        @in { input: node(tmln.loc), output: node(tmln.loc),
            a: node(tmln.loc), b: node(tmln.loc), e: node(tmln.rem),
            a_gpu: node(tmln.rem), b_gpu: node(tmln.rem),
              y_gpu: node(tmln.rem) };
        encode e.copy[b_gpu <- a];
    } else {
        @in { input: node(tmln.loc), output: node(tmln.loc),
            a: node(tmln.loc), b: node(tmln.loc), e: node(tmln.rem),
            a_gpu: node(tmln.rem), b_gpu: node(tmln.rem),
              y_gpu: node(tmln.rem) };
        encode e.copy[b_gpu <- b];
    }
    @in { input: node(tmln.loc), output: node(tmln.out),
        e: node(tmln.rem),
        a_gpu: node(tmln.rem), b_gpu: node(tmln.rem),
          y_gpu: node(tmln.rem) };
    encode e.call[y_gpu <- gpu_merge'<1, 1, 1>(a_gpu, b_gpu)];
    let f = submit @ node(tmln.sub) e;
    let r = await @ node(tmln.snc) f;
    @out { input: node(tmln.out), output: node(tmln.out) };
    r.y_gpu
}

pipeline main { foo_impl }
```

Fixed–size arrays for select sum:

```
#version 0.1.0

feq sum {
    extern(cpu) pure sum1([i64; 4]) -> i64
    extern(cpu) pure sum2([i64; 4]) -> i64
}

val select_sum(v1: [i64; 4], v2: [i64; 4], v3: [i64; 4]) -> out: i64 {
    sum1 :- sum(v1)
    sum2 :- sum(v2)
    sum3 :- sum(v3)
    condition :- sum1 < 0
    result :- sum2 if condition else sum3
    returns result
}

sptl space(s: BufferSpace) -> BufferSpace { returns s }

tmln time(e: Event) -> Event { returns e }

fn select_sum_impl(
    v1: [i64; 4],
    v2: [i64; 4],
    v3: [i64; 4]
)
-> i64 impls select_sum,time,space {
    if sum1(v1) < 0 {
        sum2(v2)
    }
    else {
        sum2(v3)
    }
}

pipeline main { select_sum_impl }
```

BIBLIOGRAPHY

[1] Khronos Vulkan registry. https://www.khronos.org/registry/vulkan/.

[2] WGPU, 2019. https://github.com/gfx-rs/wgpu.

[3] Bytecode Alliance. Cranelift, 2016. https://cranelift.dev/.

[4] Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. Cuda: Compiling and optimizing for a gpu platform. *Procedia Computer Science*, 9:1910–1919, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[5] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, 2020.

[6] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[7] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 703–718, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Khronos. Opencl, 2009. https://www.khronos.org/opencl/.

[9] Khronos. Sycl, 2014. https://www.khronos.org/sycl/.

[10] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[11] W. B. Langdon and M. Harman. Evolving a cuda kernel from an nvidia template. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[12] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[13] Luke Maurer, Zena Ariola, Paul Downen, and Simon Peyton Jones. Compiling without continuations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'17)*, pages 482–494. ACM, June 2017.

[14] Mozilla. WebGL, 2011. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

[15] NVIDIA. Cuda programming guide, 2009. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[16] Biagio Peccerillo, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture*, 129:102561, 2022.

[17] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, jun 2013.

[18] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 86–97, New York, NY, USA, 2010. Association for Computing Machinery.