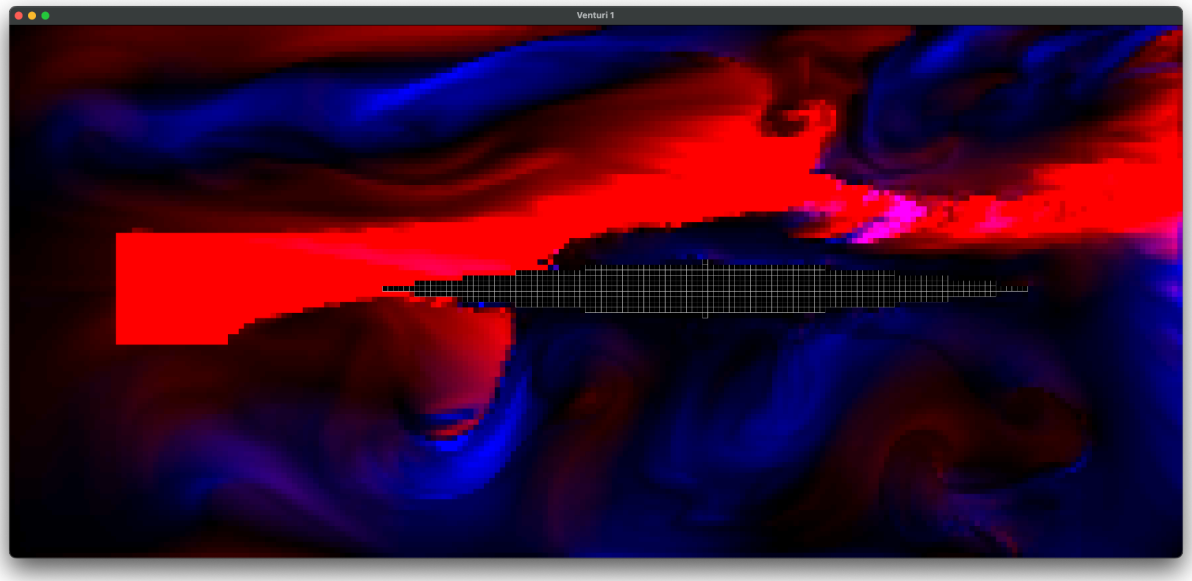

An Adventure in Programming Languages for Graphics

Spandan Goel

Abstract

In this paper, we discuss the design of domain-specific languages for graphics and scientific computing applications to boost accessibility to humans and LLMs alike, development time, and readability of highly-specialized configurations. In a case study on fluid engines, we successfully demonstrate significant simplification in configuration effort compared to traditional approaches, and evaluate the computational costs of such DSLs.

Appendix

This paper concludes with a discussion on two issues in Slang, a shading language that compiles to various shader targets such as GLSL, WGSL, Metal, and SPIR-V.

An Adventure in Programming Languages for Graphics

Preface

I was initially planning to work on `slang` to build out fixes for GitHub issues 6747 and 6714, and I spent an embarrassingly large amount of hours on various investigations into both issues, which I discuss later in this report. Unfortunately, I was not able to move towards a concrete fix at the time of writing, so am choosing to primarily present some PL-related work I did for a final project for two of my computing-based classes, over the last two weeks. I did want to showcase this at some point, but only came up with the idea after the proposal was submitted. It's not the *coolest* thing out there, but it took a substantial amount of effort, so I hope you enjoy it.

Flux

The source is located at https://github.com/Spandan14/venturi/tree/main under `src/fluxlang`.

Introduction

Most applications in the graphics or scientific computing domains use common file formats to specify configurations, such as JSON, YAML, or TOML. However, these formats are unable to express fundamentally mathematical constructs well, especially dynamic ones that may interact with the application in non-trivial ways.

For users looking to supply more detailed configurations, the options are usually limited to writing a program that interoperates with the application or an exposed API. This is challenging for those not versed in the specific languages and toolchains that an application may use.

To combat this, we introduce a prototypical language named *Flux*. Flux is a domain-specific language for designing scenes and simulation setups on the *Venturi* fluid engine. While Flux was designed as an addition to Venturi, the runtime can be extended with light effort to other engines to produce various configurations. More broadly, the ideas used in Flux can be used to design DSLs for other graphics-domain programs, such as renderers, general physics engines, mesh editors, and more.

Implementation

Features

Flux is supported by a parsing expression grammar. Programs can supply mathematical expressions using all the standard arithmetic operations and a variety of useful built-in utilities, such as trigonometric functions, exponentiation, absolute values, and clamps. These mathematical expressions can be composed to form scalars and vectors that are passed to various initializers to control forces, densities, flows, and solids at simulation-time.

Flux also lets users define dynamic cell-sets, which can be used as targets for the aforementioned initializers, providing fine-grained control without sacrificing concision or readability. These cell-sets can be defined mathematically, further increasing Flux's accessibility to users not familiar with the backend application's internals.

Finally, Flux exposes a set of variables that enable high interactivity with the simulation. These include the indices and locations of each cell and the current simulation time. While we do not expose more complex runtime information yet, such as curl, divergence, or pressure, these are relatively simple additions.

Transformation

Flux uses a PEG library to parse the initial AST from user-supplied programs. It then transforms this AST into Flux's internal AST representation consisting of a variety of expressions and statements. Each statement has a simulation property or cell-set as the rvalue, and a scalar or vector expression as the lvalue. We currently do not support holding expressions in rvalues.

The AST transformation process also acts as a "type-check" pass over the module, ensuring that built-in library functions are called with the right arguments, range expressions are bound to usable variables, usage of "all" as a target is "typed" (for now, just noted) distinctly from user-defined targets, etc. Flux does not support a proper type system yet: for example, all mathematical quantities are evaluated in floating-point operations. However, this has not been a hindrance in our usage thus far, as we remain able to differentiate between scalars and vectors.

Runtime

Once a transformed AST is built, it gets passed to the Flux runtime. This runtime evaluates all the statements and expressions to produce a simulation object that is populated with functions that determine cell types, initial density values, forces on every cell, sinks and sources, and inflow/outflow rates or ratios. Additionally, the runtime initializes base parameters like the simulation dimension, cell size, world grid sizes, and the renderer. At the time of writing, we do not expose any parts of the renderer as configurable in Flux, but plan to do so eventually to support some customizability in visualizations and rendering.

The evaluation of any expression in Flux returns a C++ lambda. These anonymous functions are managed using `std::function` and take the set of predefined simulation variables as inputs, as an expression can depend on any one of these. While these signatures are fixed in their arity at the moment (except for their dependency on simulation dimension), we expect the usage of variadic templates to be key in supporting more simulation-exposed variables in the future.

As previously mentioned, we support vectors, scalars, and cell-sets to be defined as and used in expressions. This means that the specific template instances of `std::function` are varied, which necessitates the usage of `std::variant` and thus runtime type reflection. This comes at a performance cost that is discussed more in later sections.

Evaluation of nested expressions produces a closure that captures the evaluations of the dependencies. This is easier said than done - it requires writing a global in-lambda applicator that allows for application of the inputs to a closure produced by a parent expression to the captured child expression lambdas. These child expressions may be of any “type”, therefore requiring RTTI once again.

Cell-sets are another tricky bit - they require the storage of lambdas that capture “membership” of a given cell in a set. Each cell-set definition in a Flux program corresponds to one “membership” function. Once a program is fully executed, the runtime builds a general lambda that provides information on which sets a cell belongs to (other than the universal `all` set). This general membership lambda is provided to the simulation, and is used to look-up the simulation properties specified for the cell-sets a given cell is a member of.

Evaluation

A bytecode-based runtime for Flux was a consideration, but was ultimately postponed due to a lack of development time. A simple stack-based virtual machine could be attached to the simulation, and used to execute “bytecode stubs” generated by the evaluation of Flux statements. If we then term the current approach to be “interpreted Flux”, there may be opportunities to introduce a sort of JIT compilation into Flux for more expensive simulation parameters, i.e., those that would suffer heavily from RTTI access overhead.

The cell-set membership check overhead is noticeable. The following profile was run over 15 seconds of simulation and shows that about 1.17s, or 7.8% of the execution time was spent in calls using `std::function` and `std::variant`. When the same simulation was defined internally within the Venturi binary, 0.3% of the execution time was spent in calls to `std::function`, which is quite a bit cheaper.

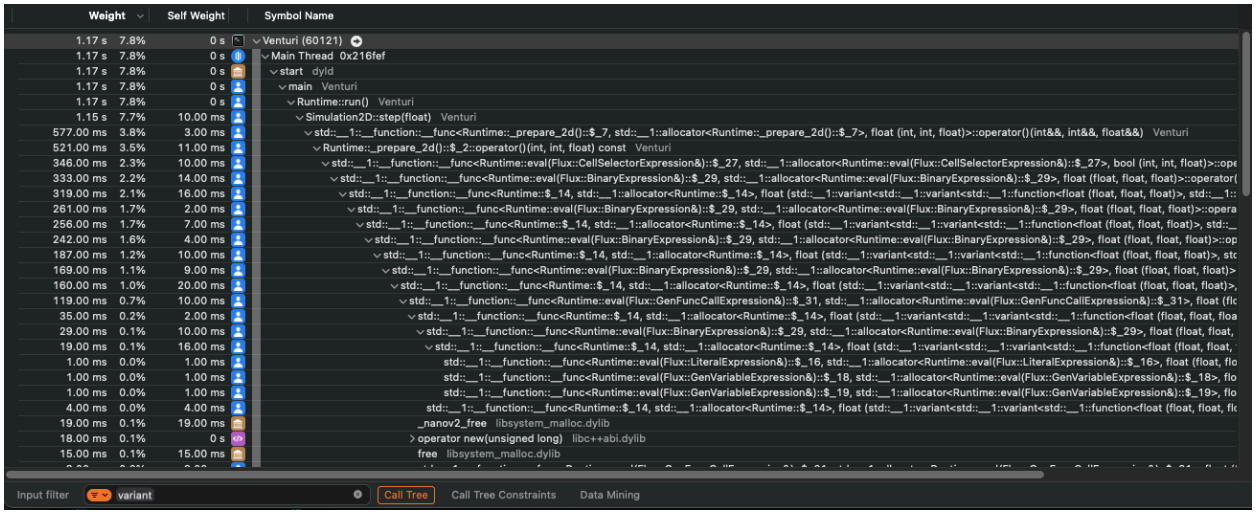

Figure 1: 15-second profile of Venturi with a simple Flux program spending roughly 8% of execution time inside calls to std::function and std::variant

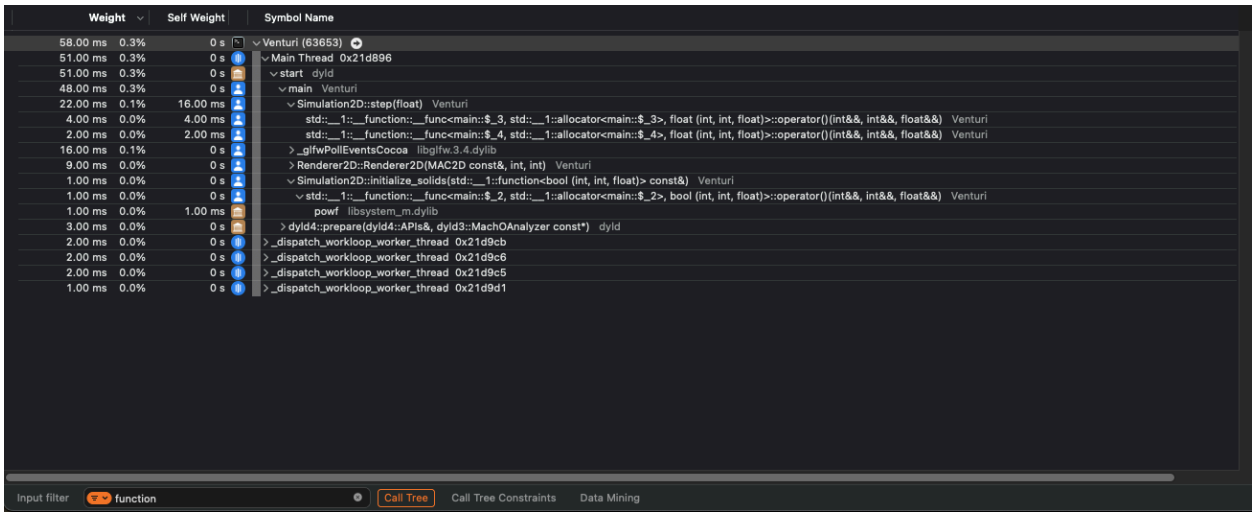

Figure 2: 15-second profile of Venturi with an internally-specified simulation, spending roughly 0.3% of execution time inside calls to std::function, with no calls to std::variant or RTTI needed

```
dim 2D

grid 220 x 100 with dx = 1
window 1760 x 800

set airfoil = cells where (abs(i - 130) / 60)^1.5 + (abs(j - 50) / 5)^1.2 <= 1

set init_density = cells where (20 <= i <= 40) and (40 <= j <= 60)
set sink = cells where i > 200

solid airfoil = true

density all = 0.15
density init_density = 1.0

flow init_density = min(3.0, 0.10 * t)
flow_ratio sink = max(1 - 0.01 * t, 0.5)

force all = (100, 0)
```

Figure 3: `airfoil.flx`, the Flux program used for the test simulation.

This Flux program is merely 11 lines long and is fairly readable for someone not familiar with Venturi's internals. The internal, C++-based definition of this simulation is on the order of 50 lines, and would also require recompilation for every change.

We were unable to write a conventional specification file-format parser, like one for JSON, but we imagine the overhead to be somewhere in the middle of these two options, with a simulation specification like this one being on the order of 80-100 lines.

Flux is so natural-language friendly that with one example, LLMs are able to generate scenes too. This is an example generated by Claude Sonnet 4.5, which works out of the box without any modifications. It is shown on the next page.

```
dim 2D
grid 300 x 150 with dx = 1
window 2400 x 1200

// Create two oscillating cylinders that generate von Kármán vortex streets
set cylinder1 = cells where (i - 80)^2 + (j - 50 - 15*sin(0.1*t))^2 <= 100
set cylinder2 = cells where (i - 80)^2 + (j - 100 + 15*sin(0.1*t))^2 <= 100

// Porous barrier that creates turbulence
set barrier = cells where (i >= 180) and (i <= 185) and ((j % 10) < 5)

// Density injection zones - multiple sources
set source_top = cells where (5 <= i <= 15) and (120 <= j <= 135)
set source_mid = cells where (5 <= i <= 15) and (70 <= j <= 80)
set source_bot = cells where (5 <= i <= 15) and (15 <= j <= 30)

// Exit zone
set outlet = cells where i > 290

// Solid obstacles
solid cylinder1 = true
solid cylinder2 = true
solid barrier = true

// Initialize low background density
density all = 0.08

// Inject different density fluids from multiple sources
density source_top = 1.2
density source_mid = 0.9
density source_bot = 1.5

// Create pulsing flow from sources
flow source_top = 2.5 + 0.8*sin(0.15*t)
flow source_mid = 2.8 + 0.5*sin(0.15*t + 1.0)
flow source_bot = 2.2 + 0.7*sin(0.15*t + 2.0)

// Gentle suction at outlet
flow_ratio outlet = 0.6

// Horizontal force with slight vertical oscillation
force all = (80 + 20*sin(0.05*t), 10*sin(0.08*t))
```

Figure 3: `claude.flx`, an LLM-generated Flux program that works out of

Contributing to `slang`

Working with Slang

I felt that getting started and ramped-up to `slang` was quite challenging for a variety of reasons. First, Slang does not support MacOS development very well, nor do they document that this is the case. This is perfectly understandable, but I did not expect the level of difficulty I ran into to be as severe as it was.

Setting up Nix was helpful, and I was able to compile the codebase and debug using `slang-unit-test-tool` and `slangc`, but was unable to generate debug symbols that LLDB could read. I spent around 7-8 hours trying to get debug symbols out, messing around with `dsymutil` and `.dSYM` files all over the place, trying my own CMake configs (which was somewhat successful), but in the end I had to switch over to a Linux system. Setup here took another couple of hours, but I was able to use GDB and walk through the code at least.

Commentary on Issues 6747 and 6714

Issue 6747

I made quite a lot of attempts to reproduce the issue the original poster was facing using a variety of runtimes. I attempted to use Dawn, which is maintained by Google, but Tint would validate and pass the shader. I attempted to use Naga and Naga's CLI, which both worked. A variety of online runtimes/playgrounds worked as well. As a last resort, I tried to render in Chrome with a simple HTML/JS setup, in C++ by using WebGPU headers, and in Rust by using the Rust `wgpu` crate.

All of these succeeded, and I was honestly lost at this point. I decided to downgrade `wgpu`, and lo and behold, the issue resurfaced. I also believe that the WGSL spec is a bit ambiguous on its definition of `const`, which was also not very helpful.

Issue 6714

This issue is certainly much more challenging than it looks on the surface, at least to my understanding. The primary issue arises from the failure to construct a witness that shows that a `typedef` for an `associatedtype` conforms to the requirement of a specific interface, when observed in relation to another `typedef` linked to an `associatedtype` in a parent type.

More specifically, Slang calls `checkAndConstructSubtypeWitness` when calling `isSubtype` during `TryCheckOverloadCandidateConstraints` which all is used to check whether certain types are compatible with types constraints on structs or interfaces. This function first checks the "facets" of each type and sees if any of the subtype's facets are equal to the supertype, and

then performs a variety of dynamic casts in the event that the subtype is of a special case (such as an aggregate, or an expanded type, or an existential type) to attempt to generate a witness.

These cases make it easy to check, in the given issue, `Layer` against `Associated`, `Layer.INPUT` against `Layer.INPUT`, `Chain<Layer, Layer>` against `Associated` but virtually impossible to compare two `.INPUT` types that are not the exact same.

I faced a lot of trouble with trying to add a case here, because I was unable to trace and find where typedef or alias information is stored for types, or what class I might downcast a general Type to. Facets do not store this information, and while I tried casting to an `AssocTypeDecl`, it did not give me any information I could use. When observing what happens to typedefs in GDB, I noticed that the `ASTBuilder` tries to store a modifier that signals that a given type is an alias, but this information doesn't seem to get propagated to a point where I could use it inside any conformance check.

My understanding at the moment is that typedefs are slightly broken for this reason - in fact, even a more trivial version of the example in the issue fails to compile. I think a proper aliasing setup will need to be added to types that are defined as typedefs to other associated types, but I'm pretty sure I spent close to 20 hours reading through code and scribbling notes down.

Future with Slang

I am committed to working on issue 6747 as it has been assigned to me by one of the contributors, so I will be sorting that one out in the next two weeks and reporting my findings. With regards to 6714, it is still an interesting challenge. It's unfortunate I was not able to get through it this time, but I do feel I can take another stab at it once I am done with finals.

I felt limited primarily by the scope of the codebase and the lack of documentation (I even searched in other issues for help, but other than code comments, I was unable to find anything of substance).