The Julia interface to the Halide library*

Jiuning Chen <johnnychen94@hotmail.com>

April 2021

Abstract

Both Halide and Julia provides its own solution to high-performance computation, this project provides a native Julia interface to Halide library. By calling Halide routines inside Julia, Julia users could further improve the performance of their algorithms.

Contents

*This is for the Google Summer of Code (GSoC) 2021 project in the Halide community.

1 Introduction

Halide[1, 2, 3] is a programming language designed to make it easier to write high-performance image and array processing code on modern machines. Halide is embedded in C++, which means users are actually writing C++ codes to build the Halide data types and computation graphs. Halide separates the code implementation into algorithm and schedule, and exposes schedule strategies with an easy-to-use API, which allows users to implement and optimize the codes efficiently.

Halide has several frontends. The static typed language C++ is the native host language of Halide. Halide also has a frontend in Python, one of the most popular dynamic typed language. The Python frontend allows users to write Halide code in a read–eval–print loop (REPL) experience, so that algorithm developers can efficiently implement, debug and optimize the code. Halide also provides an code generation pipeline to the C++ interface of Pytorch, this allows developers to implement the Pytorch neural network code in Halide including both the forward and backward computation.

Although many scientific computation researchers uses Python as their main coding language, Python itself as an interpreted language has very bad performance; all the high-performance libraries are backed by its C/C++ part implementation and optimization. There are also compiled variants of Python, e.g., Cython [4] and Numba [5]. However, even though these JIT variants can hit the optimization limit on small examples, for any complex algorithms, it is still a challenge to optimize the codes.

Julia[6] is another programming language that focuses on scientific computation and high performance. As a compiled language, Julia is capable of reaching the state of the art performance in many tasks [7]. Unlike C++, Julia fully takes advantages of the LLVM[8] Just-in-time (JIT) feature, which allows the compilation process happens during the code run time. Thus, Julia also provides a REPL experience, while still allows very high-performance code execution. Unlike Cython and Numpy, Julia's provides a cleaner and extensible way to develop high-performance codes, and it breaks the old convention that *prototyping in one high-level language and optimizing in another low-level language*.

This project aims to provide a Julia interface for Halide library so that Julia users could take advantages of both the Halide library and other well developed Julia packages to boost their scientific researches.

2 Targets, Principles, Goals and Non-goals

The final goal of this project is to bring a native Julia experience of using Halide. To achieve this, not all Halide functionalities will be binded and not all Halide API will be preserved. In this section, I'll explain the targets, principles, goals and non-goals of this project. More implementation details and decisions deduced from these will be covered in section 3.

2.1 Targets and Principles

The main target of this project is Julia users. Although C++ is a good choice for performance, it is usually not a good choice for fast prototyping. On the contrary, Julia provides a much better research-develop workflow for scientific researches while still promises high-performance within the same language; Julia nicely solves the two-language problem that one has to prototype in one language and implement in the other language. High-level languages such as Python, Matlab, R, and Julia are currently the favorite language choices among scientific researchers. For this reason, when designing the API of the Julia frontend, it would compromise more on the Julia users (the researchers) and less on the Halide internals (the library developers). It will be no surprise if the Halide C++ users find the Julia frontend a foreign library.

The core principle of this project is to make the Julia frontend of Halide an open library. Many high-performance libraries in scientific computation (e.g., Numpy, Tensorflow, Pytorch) make themselves self-contained so as to reach the performance limit. The trade-off is that they also reject the possibility to closely interact with other packages because none of them knows each other. The Halide C++ library also builds its own wall. However, this is not the same story for Julia; Julia's multiple dispatch perfectly balances the abstraction for easy usage and the specification for performance [6]. This makes Julia a big ecosystem and most Julia packages admit a much more close interaction between each other. For example, the `CUDA.jl`[9] provides the CUDA array type `CuArray` which is just one of many custom array types in Julia, and `Zygote.jl`[10] provides a source-to-source automatic differential (AD) library. The Zygote design is for generic array so it applies to any array types, yet it still allows custom optimization for `CuArray` via method specification. For this reason, the Julia frontend of Halide could and should be designed to embrace the open ecosystem of Julia and thus take advantages of many well optimized Julia packages, e.g., the AD package `Zygote.jl`[10], the differential equation library `DifferentialEquations.jl`[11], the image processing library `Images.jl`[12], and the optimization library `JuMP.jl`[13].

Julia is not a part of the Halide ecosystem. The Julia frontend of Halide

is a part of Julia's ecosystem. Julia users can take advantages of Halide to implement an efficient Julia function and use it in a larger Julia project. Halide users, in general, can not use Julia to write part of the Halide code and then use it in a larger Halide C++ project.

Unless there are performance issues, what can be implemented in native Julia, implement it in Julia. There are two reasons for this: 1) type information, and 2) generic programming. The great amount of type information that Julia passes to lower-level LLVM is one of the secret behind the scene for Julia's success on high-performance in numerical science. C++ runtime library, even though highly-optimized, is still a black box to Julia and thus calling it would loss a lot of information that may be useful to Julia. Compared to the single dispatch in C++, Julia's multiple dispatch allows a more flexible and powerful generic programming. Re-implementing part of Halide functions could bring better Halide support for Julia types, and eventually make other Julia packages to take advantages of Halide.

2.2 Goals

There are several high-level goals of this project:

Availability Julia users can easily install the Halide package `Halide.jl` with Julia's own package manager `Pkg` via `pkg> add Halide`. This add operation includes both the Julia source codes and the prebuilt runtime library of Halide.

Documentation The Julia frontend has its own documentation and tutorial setup for both Julia users and Halide users.

Native array support Julia functions that apply to generic array types can also apply to Halide array. Halide functions that apply to Halide array can also apply to Julia generic array.

Performance The Julia frontend gives as performant result as the C++ frontend.

Based on these high level goals, there are several Halide functionalities that will be covered by this project.

JIT Julia itself is fully JIT enabled, supporting the Halide JIT compilation gives the most seamless coding experiences for Julia users. This is the basic requirement of this project.

Native buffer array User-defined custom array can be as performant as Julia's built-in array type `Array`. Making Halide Buffer a Julia array provides the possibility for Halide to use many of the well-optimized functions.

Native graph (Stretch goal) Build Halide computational graph using Julia's own type. This breaks the Halide type wall and allows better interaction with other Julia packages.

2.3 Non-goals

This is an experimental project, there are some non-goals that this project won't cover, mostly they are advanced Halide functionalities that won't benefit common Julia users very much given the current status of the Julia ecosystem (Julia is still very young):

AOT/Generator Ahead-of-time compilation of Halide pipeline is a good strategy to reduce compilation time for application. However, as a deployment optimization strategy this will not be covered in the first version of `Halide.jl`.

Cross-compilation Julia has no support for this as far as I known. Julia users still prefer a REPL coding experience and a script-like usage: `julia script.jl`.

There are some Halide-supported platforms that Julia does not yet support[14]. For this reason, this project will only focus on platforms that Julia declares Tier-1 support. They are: x86-64 (64-bit) macOS, x86-64 (64-bit) Windows[1], x86-64 (64-bit) Linux, ARMv8 (64-bit) Linux, i686 (32-bit) Linux and x86-64 (64-bit) FreeBSD.

Halide also supports GPU computing, auto-scheduler, and has its associated debug tools to investigate the generated codes. These are stretch goals of this project, and will be covered when time allows.

3 Halide Types and the Julia Ecosystems

To make the Julia frontend a native Julia package[2], we need to briefly review the Halide types and the Julia Ecosystem. This helps justify our decisions from a technically point of view.

3.1 Halide

The core concept of Halide is to separate the algorithm from its schedule; Halide promises that as long as the algorithm is unchanged, one can iteratively try out different valid schedules and use the most efficient one. In Halide, the *algorithm* defines the dependency of the data. For example, `f = x + y` defines a simple algorithm that `f` depends on two pieces of data `x` and `y`. However, whether `x` or `y` get calculated first is not defined. The role of Halide *schedule* is to define this "undefined" part of the execution order. A typical Halide codes consists of three parts: algorithm, schedule and its realization.

A Halide *algorithm* defines a directed acyclic graph (DAG), which corresponds to the `Halide::Func`. A DAG consists of sub-graphs `Halide::Stage`,

[1] As of the time of writing, because Halide requires MSVC 2019 (Microsoft Visual C++ 2019) and Julia's Binary Builder only support MinGW, it is yet not clear if this is feasible.

[2] By saying "native", it means that this package can be seamlessly connected to other Julia packages.

where each sub-graph consists of an operation node and data nodes. Operation node is just a normal function, A data node can be either another `Halide::Stage`, an expression `Halide::Expr`, or an iteration index `Halide::Var`.

A Halide *schedule* defines a transformation from one DAG to another DAG in a way that does not change the data dependency. Halide provides plenty of schedule primitives, e.g., `Halide::Func::split` to split one for loop into two for loops, `Halide::Func::fuse` to combine two for loops into one for loop. By applying primitives appropriately, one composes a schedule.

The *realization* of an algorithm is the process that concrete calculation gets executed in the CPU/GPU memory. To conceptualize this process, Halide defines its own plain data types (`int`, `uint`, `float` and `handle`) as well as its own array type, i.e., `Halide::Buffer`.

To let Julia users call Halide routines inside Julia, the bindings must provide:

- a way to create Halide algorithm,
- a way to create and apply schedule to given algorithm, and
- a way to read from and write to Halide buffer.

For this reason, most of the public Halide types and methods to create algorithm and schedule will be ported to Julia. The `Halide::Buffer` type will also be ported to Julia as an array.

However, not all `Halide::Buffer` methods will be ported because Julia provides a lot of functions for generic arrays; ideally, we only need to tell Julia the necessary information of the Buffer (e.g., dimension, size, stride, element type) and then let the native Julia functions do the work.

There are also some types that will not be ported to Julia, most of which are runtime and internal types. Some helper types have their direct alternative in Julia. For example, `Halide::Tuple` can be replaced by the Julia `Tuple`. It is not checked yet whether this kind of types can be avoided during the binding. To make the project maintainable, the basic idea is to provide a minimal binding. Hence if there was no need to port `Halide::Tuple` to Julia, it will be hided as the binding internals and not exposed to Julia users, and all corresponding API with `Halide::Tuple` will be replaced by Julia's own `Tuple` type.

3.2 Julia

3.2.1 CxxWrap

`CxxWrap.jl` will be used to for this project. To set up `CxxWrap.jl`, two parts are needed: the C++ part `libhalide-julia` and the Julia part `Halide.jl`. `libhalide-julia` translates Halide types, names, and functions to a way that Julia's C call could understand. `Halide.jl` then provides a thin wrapper and abstraction of the exposed Halide symbols so that Julia users don't need to directly communicate with Halide internal.

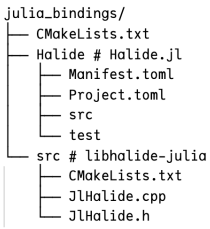

Figure 1: Julia Binding folder structure

The `libhalide-julia` and `Halide.jl` folder will be organized in a way that similar to the existing python binding, as shown in fig. 1. The binding test will be put into `julia_bindings/Halide/test` folder as Julia codes.

3.2.2 Multiple dispatch

Multiple dispatch is the key to Julia's success for its high performance. With multiple dispatch, Julia finds a balance between the abstraction, which is friendly to programmers, and the specification, which is friendly to hardware.

Unlike the single dispatch method in C++, Julia's multiple dispatch does not declare any types as "first-class citizens". For this reason, Julia has plenty of array types and plain data types. For example, JuliaImages, the image processing toolbox in Julia, interprets any array of element type `Colorant` as an image. A typical code snippet that one could find inside JuliaImages is:

```
# defines how psnr of gray image are calculated
function assess_psnr(x::AbstractArray{Gray},
                     ref::AbstractArray{Gray})
    20log10(1) - 10log10(mse(x, ref))
end
# recursively call the gray method
function assess_psnr(x::AbstractArray{RGB},
                     ref::AbstractArray{RGB})
    assess_psnr(channelview(x), channelview(ref))
end
# a fallback implementation
function assess_psnr(x::AbstractArray{<:Colorant},
                     ref::AbstractArray{<:Colorant})
    ...
end
```

Creating a new array type in Julia is super easy and yet still performant. For this reason, we can provide a very thin wrapper on `Halide::Buffer`, for example:

```
struct Buffer{T,N,AT} <: AbstractArray{T,N}
    data::AT
```

```
end

# This eventually calls the Halide routine to get the buffer
    information
Base.axes(A::Buffer) = ...
Base.size(A::Buffer) = ...
Base.getindex(A::Buffer, i::Int...) = ...
```

where the actual buffer data `data` can be either a CPU buffer or a GPU buffer, which totally depends on how we interpret the Halide buffer.

A lot of Julia functions are written for generic arrays, ideally, they would just work on our `Buffer` array type without any extra modifications.

3.2.3 Documentation

The documentation consists of two major parts: the references and the tutorials.

Almost all Julia packages use `Documenter.jl` to set up their documentation. It describes how Julia functions are documented in either Julia source codes or markdown. Essentially, `Documenter.jl` is a static HTML generator, this makes it incompatible with Halide's existed documentation repository.

The author himself provides a `Documenter.jl` extension `DemoCards.jl` to generate demos in a literate programming manner. In very simple words, it converts a well-written and runnable Julia source codes into Markdown and Jupyter notebook, and then pipe the generated Markdown files into `Documenter.jl`. A live example can be found in the documentation page of JuliaImages.

For this project, `Documenter.jl` will be used to generate the references, and `DemoCards.jl` will be used to write the Julia version of Halide tutorials.

3.2.4 BinaryBuilder

The Julia interface of Halide will not require Julia users to compile it from scratch, instead, `BinaryBuilder` will be used to generate the precompiled libraries/artifacts for each supported platforms. When Julia users install `Halide.jl` via the Pkg interface `pkg> add Halide`, the Julia package servers around the world will ships both the Julia source codes of `Halide.jl` and the associated precompiled artifacts of the target platform.

Julia `BinaryBuilder` mimics a sandbox environment with predefined LLVM, GCC, and other tools for reproducible consideration; including Windows and macOS, all precompiled artifacts are cross-compiled in the Linux host environment.

Unfortunately, `BinaryBuilder` only provides `minGW` for Windows target, while `Halide` only supports `Visual Studio C++`. For this reason, it is not guaranteed that precompiled artifacts for Windows will be available.

3.2.5 API and behavior changes

The goal of this project is to provide native Julia interface to better incorporate with Julia ecosystem. Hence, there will be some API and behavior changes to make it more friendly to Julia users. I'll give two typical examples of API change and behavior change:

API change There are many Halide methods that support `std::vec`, for example, `Func::realize` use `std::vec` for sizes. However, Julia itself uses `Tuple`(not `Halide::Tuple`) for sizes. For this reason, all `std::vec` methods will be replaced by a corresponding `Tuple` version.

Behavior change Julia uses 1-based indexing assumption, while Halide/C++ uses 0-based indexing assumption. An offset will be added when Halide realization is called from inside Julia.

As the project proceeds, there might be more similar changes. All these note-worthy changes will be discussed in details in a minimal PR form.

3.3 Proof of Concept

To make sure that this project has at least some outputs, some simple and fast prototyping work is done. As of the time of writing, the simplest binding JIT demo[3] is already available for x86-64 (64-bit) Linux platform with Halide-required LLVM:

```
# Create Func and Var
f = Func("Blur")
x, y = Var("x"), Var("y")
# Build the stage
f[x, y][] = x + y
# Do the real calculation and return the Halide Buffer
# as a normal Julia array
result = realize(f, (4, 4))
```

which outputs

```
4×4 Matrix{Int32}:
 0  1  2  3
 1  2  3  4
 2  3  4  5
 3  4  5  6
```

[3]The C++ version can be found in https://github.com/halide/Halide/wiki/Minimal-JIT-compiled-example.

4 Delivery, Schedule and Timeline

4.1 Delivery

From bottom to top, the outcome of this project are:

- The Julia binding library `libhalide-julia`, this can be placed either in the main repository of Halide library, or in the new `Halide.jl` repository.
- The pre-built runtime library for `libhalide` and `libhalide-julia` as Julia artifacts `Halide_jll.jl`.
- The Julia package `Halide.jl` and associated documentation and tutorials, this provides a native Julia interface, and internally calls Halide library via `libhalide-julia`.

At most three separate repositories in Halide organization are needed in order to provide separate version control for `Halide.jl` and `Halide_jll.jl` . The latter can be alternatively placed in JuliaBinaryWrapper organization, where the Julia community places their prebuilt artifacts to. Because they uses different system, the documentation for `Halide.jl` will be hosted in a different repository than Halide's current one; the `Halide.jl` documentation can be hosted in the same repository of `Halide.jl`, but can also be hosted in a new repository reserved for documentation.

The final goal of this project is to let Julia users use the Julia package manager `Pkg` to install Halide via `pkg> add Halide`. By doing this, Julia ships both the pre-built artifacts `Halide_jll.jl` and the source codes `Halide.jl` via its own servers and third-party mirrors around the world. Unlike the C++/Python users of Halide, *Julia users do not need to build Halide library from scratch with CMake commands.*

4.2 Schedule

This project will be shipped by four (mostly) sequential stages:

Prototype provides a working binding library on x86-64 (64-bit) Linux system with Halide-recommended LLVM setup and without BinaryBuilder.

Buildbot automates the build scripts with Julia BinaryBuilder so that we can use CI to track future changes and detect potential regression and bugs.

Julia API designs the `Halide.jl` API to better cooperate with the larger Julia ecosystem.

Release add more platform supports, more documentation and tutorials. Finally register both the frontend source codes `Halide.jl` and the prebuilt artifacts `Halide_jll.jl`.

This binding project would finally adds thousands of lines of codes to the Halide organization, so it would be quite challenge for potential reviewers to track the differences. To solve this issue, the prototype version should be a pure binding version that faithfully stick to Halide's design and behaviors. Then immediately sets up the build script with Julia's BinaryBuilder so that we can add CI support for our prototype version. This two stages will likely be delivered in form of a hard-to-review giant PR with plenty of test cases. Since there will be as few changes as possible to the Halide behavior, making sure test behaviors are correct should be sufficient enough to justify the work.

Once the prototype and CI stages are done, it comes to the Julia API stage. In this stage most codes are pure Julia codes, and there will be some Julia-specific changes to the Halide behavior to better suits the Julia's usage. One typical example of such change will be the index shift from 0 to 1; Halide and C++ uses 0-based indexing, while Julia uses 1-based indexing. All these changes will be tracked with small and easy-to-review PRs.

After the Julia API stage, the core part of the API will become stable, and users can get a developed version of it from the Halide repository. More platforms support, more documentation and tutorials[4] are needed to increase the coverage, and finally two packages will be registered in Julia's default package registry `General`[5]. This makes Halide world-wide available to normal Julia users with a single `pkg> add Halide`.

4.3 GSoC Timeline

For the purpose of GSoC phase evaluation, I'll list the expected timeline of this project:

Present - June 7 The *prototype* and *buildbot* stages, i.e., setting up the prototype and adding CI for it.

June 7 - August 9 The *Julia API* stage, i.e., add more Julia codes to the Julia frontend for a native Julia experience.

August 9 - August 16 The *release* stage, i.e., add more platform supports.

August 16 - future Release, bug fixes and maintenance.

GSoC 2021 has two evaluation phases, this is a reference criterion that I have in mind; whether it is adopted depends on mentors' own judgement:

Phase 1 student and mentor should submit the phase 1 evaluation before July 12. As long as there is a prototype version of `Halide.jl` for future enhancement development purpose, it should be considered sufficient to pass.

[4] Most of the documentation and tutorials will be added together with the defined Julia API sets.

[5] https://github.com/JuliaRegistries/General

Final Phase student and mentor should submit the final phase evaluation before August 23. It should be considered a pass once 1) there are specific documentation, tutorials and 2) users can get `Halide.jl` via `pkg> add Halide` on several widely-used platforms. Whether the proposed API during the GSoC period becomes stable should not be considered a factor; it takes time to apply `Halide.jl` to other downstream packages and applications.

5 Challenges

This section summaries some challenges that I foresee before implementing all the details. It is definitely not a complete list, but hopefully it catches the most important ones.

5.1 Type conversions

Julia has a much broader type system than what Halide has. For example, `FixedPointNumbers.jl` defines `N0f8` to represent values in $[0, 1]$ range with 8-bits number, `ColorTypes.jl` defines `Gray` and `RGB` to represent the gray and RGB pixel type. A widely applicable conversion rules would make Halide useful to a broader Julia ecosystem.

5.2 Generic array support

Julia also has many custom array types. For example, `OffsetArrays.jl` defines `OffsetArray`, which is an array with n-based index. `MappedArrays.jl` defines a lazy map array. Although the schedule primitives for Halide C++ runtime only supports the very typical array with actual contiguous memory layout, it would be very useful to let them support generic arrays.

This could be done by providing an custom array type that reinterpret the Halide's internal `Buffer` type. However, efforts should be paid to the lifetime and ownership management of the memory to make buffer operations from the Julia side safe.

5.3 Extended syntax

Julia does not allow override operator to assignment `==` and thus a workaround to it is to override the `setindex!` function, which adds an extra pair of bracket for stage generation.

```
1 f = Func("Blur");
2 x, y = Var("x"), Var("y");
3 f[x, y][] = x + y; # an extra [] here
```

Julia has a powerful meta-programming support with its macros, so it is totally possible to define a Halide-specific macro for common usage. For example, `ParameterizedFunctions.jl` provides a convenient way to define function:

$$\frac{dx(t)}{dt} = \frac{v\left(y\left(t\right)\right)^{n}}{k^{n} + \left(y\left(t\right)\right)^{n}} - x\left(t\right)$$
$$\frac{dy(t)}{dt} = \frac{x\left(t\right)}{k_{2} + x\left(t\right)} - y\left(t\right)$$

```
f = @ode_def positiveFeedback begin
    dx = v*y^n/(k^n+y^n) - x
    dy = x/(k_2+x) - y
end v n k k_2
```

The Julia frontend of Halide could possibly define its own macro syntax. However, whether this improves user experiences is not yet validated.

5.4 LLVM

Julia itself ships a shared LLVM runtime library, thus using a different LLVM to build Halide library might cause some symbol conflicts. If we can build Halide using Julia's shared LLVM, then it will avoid many potential symbolic bugs and conflicts.

Another issue related to LLVM is that the shared LLVM version is bundled with Julia, which means that `LLVM 12` will not be available to `Julia v1.6`, which will very likely to be the next long-term-support (LTS) version.

In the worst case scenario, I will have to use a custom LLVM to build both Julia and Halide. This will be bad for end users because Julia ships with prebuilt binaries for all platforms.

5.5 Prebuilt Artifacts

Julia's BinaryBuilder is designed to make artifact building process reproducible in Linux environment. The only way that BinaryBuilder support to cross-compile to Windows targets is using MinGW. However, Halide has dropped the MinGW support. This means that some custom build utilities are needed if we are going to ship prebuilt artifacts for Windows.

6 About the Author

Jiuning Chen[6], the author of this project, is currently a second-year PhD student in the school of mathematical sciences, East China Normal University, Shanghai. His major research field is image processing. In the past two years, he spent a lot of time on developing Julia image processing packages and on introducing Julia to Chinese users. In github, he is a member of the Julialang organization,

[6]The unofficial name "Johnny Chen" is used more frequently. His GitHub profile can be found at `https://github.com/johnnychen94`.

a core maintainer of JuliaImages and many relevant packages. He was a GSoC 2019 student in the Julia community[7].

Julia, Matlab and Python are languages that he knows a lot of. He has a lot of Linux development experiences. As a core maintainer of JuliaImages for over two years, he has good feelings about maintenance and reliability. He has nearly no C++ knowledge before but is a quick learner; the runnable binding demo presented in this proposal is written within 3 weeks of learning and experimenting of C++, CMake and CxxWrap.

There might be one or two weeks unavailable during the GSoC coding period, but he will spend basically all his available time on this project since April.

References

[1] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)*, 31(4):1–12, 2012.

[2] 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. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[3] Jonathan Millard Ragan-Kelley. *Decoupling algorithms from the organization of computation for high performance image processing*. PhD thesis, Massachusetts Institute of Technology, 2014.

[4] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2010.

[5] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[7] Julia micro-benchmarks. https://julialang.org/benchmarks. Accessed: 2021-04-03.

[8] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[7]The project information can be found at https://summerofcode.withgoogle.com/archive/2019/projects/4506751443927040/

[9] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[10] Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018.

[11] Christopher Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017. Exported from https://app.dimensions.ai on 2019/05/05.

[12] Juliaimages. https://github.com/JuliaImages/Images.jl. Accessed: 2021-04-04.

[13] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[14] Julia currently supported platforms. https://julialang.org/downloads/. Accessed: 2021-04-03.