

Compatibility, Maintainability, and Portability Improvements to the Taichi Graphics Programming Framework

Cheng Cao



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-112

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-112.html>

May 13, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Compatibility, Maintainability, and Portability
Improvements to the Taichi Graphics Programming
Framework

Cheng Cao

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

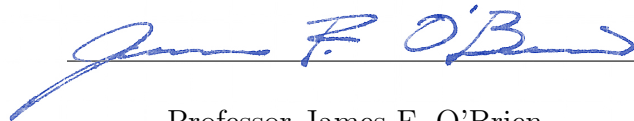
Approval for the Report and Comprehensive Examination:

Committee



Professor Ren Ng

Research Advisor



Professor James F. O'Brien

Second Reader

1 Abstract

Computer graphics is an important field that is becoming ever ubiquitous. Programming languages and tools are important factors that define the industry, but these tools have not scaled asymptotically as quickly as the visual fidelity requirements, resulting in longer development time and larger budgets.

This project builds on and extends the Taichi programming language, a Domain Specific Language (DSL) embedded in Python. Taichi was created with the goal of writing cross-platform compatible programs that achieves better performance compared to lower-level programming tools, while using an order of magnitude less code.

This project introduces a common computing device abstraction layer (referred to as the Unified Device API), with an adaptive back-end code generation system (referred to as the SPIR-V Codegen), that is able to massively improve maintainability of Taichi by eliminating 1000 or more lines of backend-dependent code, while being able to provide an even greater range of device compatibility. This part of work has been proposed and designed by myself, and I developed the majority of the implementation.

In addition, this project also introduces the design of a new ahead-of-time compilation system, that helps engineers run their Taichi programs efficiently outside of Python. I proposed the initial design for the new API and conducted the user-studies that guides our development. This API is developed by multiple contributors, including myself, Ye Kuang (Taichi Graphics Inc), Ailing Zhang (Taichi Graphics Inc), Haidong Lan (Taichi Graphics Inc), Dunfan Lu (Facebook Inc), Gabriel Huau (OPPO US Research), and various Taichi users from the industry.

By the completion of this report, the Unified Device API and the SPIR-V Codegen has been submitted and incorporated into the Taichi programming language, and the AOT pipeline, the Taichi CGraph, is currently still in development. Finally, none of this would be possible without the collaboration and support of all the developers and users from the Taichi Open Source Community.

Contents

1	Abstract	1
2	Introduction	3
2.1	Graphics Programming	3
2.2	Prior Programming Languages and Frameworks	3
2.3	The Taichi Programming Language	4
2.4	Core Contributions of this Report	6
3	Improving Taichi’s Compatibility and Maintainability	8
3.1	Designing the Unified Device API	8
3.1.1	Survey of current Graphics and Compute Backend APIs	8
3.1.2	Specification of the Unified Device API	10
3.1.3	Rationale	11
3.1.4	Implementation	13
3.2	SPIR-V Codegen Pipeline	14
3.2.1	SPIR-V and Cross-Compilation	15
3.2.2	Implementation	15
3.3	Evaluation	17
4	Running Taichi Programs without Python	19
4.1	Motivation for Ahead-of-Time (AOT) Compilation	19
4.2	Current AOT API	19
4.3	Target Users of the AOT API	21
4.4	User Study of the Current AOT API	22
4.4.1	Motivating Tasks	22
4.4.2	User Data	22
4.4.3	Conclusions from User Data	23
4.5	Designing the new AOT API	24
4.5.1	Design of Taichi Compute Graph	24
4.5.2	Formative Study: Implicit or Explicit API	26
4.6	Next Steps	27
5	Appendix	31
5.1	Survey of GPU APIs	31
5.2	Taichi program producing an animation of the 2D Julia Set	32
5.3	Conjugate-gradient Solver for Implicit FEM written in Taichi	33
5.4	Example Programs of the Compute Graph API	34
5.4.1	Example program using the implicit Compute Graph API	34
5.4.2	Material Point Method fluid simulator using an explicit Compute Graph API	34

2 Introduction

2.1 Graphics Programming

When people talk about graphics programming, they usually mean writing code that generates a visual output, processes visual data, manipulates 3D or higher dimension data, simulates real world interactions, etc. Due to the increasing ubiquity of graphics applications, graphics programming now targets an entire spectrum of devices. These include the traditional CPUs, GPUs, compute focused General Purpose GPUs (Nee-lima and Raghavendra, 2010), to recent inventions such as Ray-Tracing accelerators (Wyman et al., 2018), and Tensor Accelerators (Jouppi et al., 2018).

Many algorithms used in computer graphics are easily parallelizable, or designed with the possibility of parallelization in mind. These work in conjunction with the GPU, where these algorithms can achieve extremely high levels of performance. This re-search project will focus on the programming framework on General Purpose Graphics Processing Units (GPGPU).

The power of GPUs comes from the Single Instruction Multiple Thread (SIMT) and Single Instruction Multiple Data (SIMD) model. By running many threads with the same program, the hardware can share the instruction / control side hardware while duplicating the data-path hardware. Under this model, a single program gets dupli-cated across tens of thousands of threads. This model allows for massive throughput on a variety of tasks (Hennessy and Patterson, 2011).

However, there are a multitude of methods to program GPUs. People have developed many languages and frameworks to achieve their parallel graphics programming tasks. These range from traditional SIMD and Vector programming models, SIMT / SPMD / kernel programming languages, batched tensor programming frameworks, to more domain specific approaches such as node graph based systems. In the next section, we will detail different programming frameworks for GPUs.

2.2 Prior Programming Languages and Frameworks

GPU Programming languages and tools range from the lowest level assembly to the highest level abstract tensor programming tools. On the lowest level, the GPU is essentially a SIMD machine (Hennessy and Patterson, 2011), thus the assembly lan-guage and the IR uses a SIMD / Vector representation. People rarely directly use these to program GPUs, but rather usually used as intrinsics in higher level languages to achieve higher performance.

The two main types of high level programming are the SIMT / SPMD / Kernel style programming, and tensor style programming. Common SIMT style programming

framework includes CUDA for Nvidia GPUs, GLSL for OpenGL or Vulkan (Kessenich et al., 2019), and a C subset for OpenCL (Khronos OpenCL Working Group, 2022). CUDA is limited to only Nvidia’s GPU. GLSL is limited in terms of its capabilities, but it supports most of the common desktop platforms. On the other hand, OpenCL C has a much higher level of capability compared to GLSL, but its hardware support has been lacking. In addition to these hurdles, these frameworks usually require a large amount of boilerplate code.

On the other hand, the tensor-style GPU programming framework has improved dramatically with the massive machine learning and differentiable programming research. Common frameworks that we can find in this field include PyTorch, Tensorflow, JAX, etc. In contrast to the kernel style programming frameworks, these usually can be easily installed with a simple ‘pip’ command as they are shipped in a Python package (TensorFlow, 2022). Using them is quite simple as well, a very minimal amount of code is needed to set up the framework, and the majority of the code can be actually spent on implementing the core algorithm. However, many graphics and simulation programs are quite hard to express in a tensor-style programming framework, and sometimes using a kernel based programming language can bring much higher levels of performance (Macklin, 2022).

2.3 The Taichi Programming Language

Taichi is a kernel-style programming language developed originally by Yuanming Hu (Hu et al., 2019b) and later maintained and developed by the Taichi open-source community. Users can write Python functions and decorate it with Taichi decorator, which is a single line decoration, and when the function is invoked, the Taichi Just-In-Time (JIT) compiler will auto-parallelize the function and compile it to different supported CPU or GPU backend. See Appendix for an example Taichi program.

Ease-of-use Taichi is designed to be easy to use. For example, the user can produce an animated 2D Julia Set (or ”Fractals”) program with only 30 lines of code, including all the set-up and visualization code (program listing in Appendix 5.2). Some of the other examples include a 88 lines Material-Point-Method based fluid simulator, a 99 lines MPM simulator with fluid-solid coupling, etc. Taichi also supports differentiable programming (Hu et al., 2019a) and sparse data structure natively. Taichi Structural-Node (SNode) system decouples spatial data structure with algorithm, which enables the user to change the memory layout of their data structure without modifying their programs (Hu et al., 2019b).

Performance With Taichi, the user can write similar performance differentiable simulators with 4x less code compared to CUDA; or 188x faster compared to TensorFlow with similar lines of code, when JIT compiles to the CUDA backend. Bench-

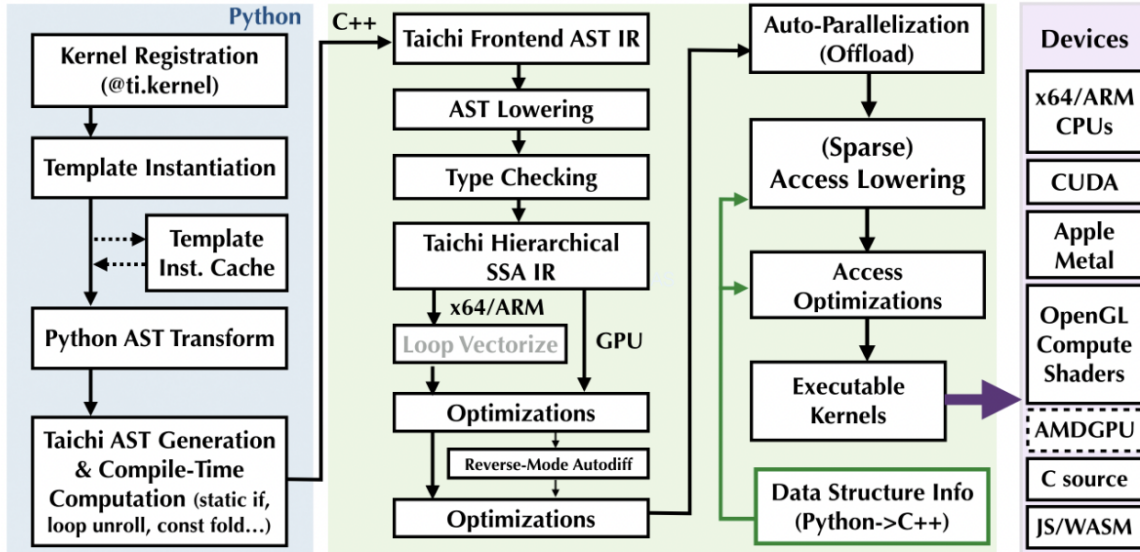


Figure 1: The Life of a Taichi Kernel (Hu, 2020)

marking across a wide range of algorithms also show that Taichi performs mostly on-par with CUDA with some cases even exceeding CUDA (Lan, 2022).

JIT and Cross-Compatibility The JIT compilation pipeline works by first transforming the Python Abstract Syntax Tree (AST) of a given Python function into the Taichi “Front-end AST”. Then the Front-end AST is auto-parallelized and transformed into an intermediate representation (IR) developed in-house by Taichi called “Chi-IR”. It is under Chi-IR where the Taichi compiler performs all the domain-specific optimizations, such as access lowering, atomic demotion, pointer analysis, etc. Afterwards, the optimized Chi-IR is sent to the “Taichi backend”, where it is compiled into executable GPU kernels. Taichi runs on multiple “Backend APIs”, such as CUDA, CPU, OpenGL, etc. Each Taichi backend compiles the optimized Chi-IR to a different format that can be used by the different Backend APIs.

This architecture of a simple-to-use Python language front-end, common middle level IR, API specific back-ends, and a JIT compilation pipeline, enables programmers and their team to write high-performance GPU software that is able to run on many devices without modification or recompilation, and the user can expect the program to behave the same across different devices. The JIT system also allows for rapid iteration, further reducing development time. The unique capability of such system is demonstrated in the case of ETH-Zurich’s physical simulation course, where students are able to develop their high-performance physics simulators in collaboration with people using drastically different hardware, while being able to utilize the full potential of their GPU as if they are programming with CUDA (Taichi Graphics LLC, 2022).

Ahead-of-Time Compilation Taichi’s ease-of-use has attracted industry users, who want to have a way to run their Taichi programs within their applications. However, many applications are limited by performance or platform regulations that they can not afford or use the JIT pipeline, or the Python run-time that Taichi is embedded in. These requirements motivated the development of an Ahead-of-Time compilation pipeline.

Engineering Challenges By mid 2021, Taichi already has a fully functioning CUDA / CPU backend that supports all the sparse data structures and data types. It also has a Apple Metal backend with sparse support, and in addition to that a limited OpenGL compute shaders backend. At this point of development, the Taichi backend code-base was growing rapidly: in order to meet the cross-compatibility goal of Taichi, we need to support a wide range of backend-APIs. This expansion makes the Taichi code-base harder to maintain, as each Taichi back-end uses an entirely different code-generation pipeline. With each back-end being independent, the gaps in performance and feature sets are also growing, missing our target of having program behave the same on all devices. In addition, in order to maintain and develop each Taichi-backend, the respective developer will need to understand both compilers and specific graphics back-end APIs, which means a lot of community members will not have the ability to contribute to the project. Such challenge is fundamentally caused by a lack of layering and code-sharing between different Taichi back-ends, presenting an opportunity for large improvements.

In addition, we have collected a lot of feed-backs on the previous ahead-of-time compilation API. Many users have reported difficulties or annoyances using the previous AOT API, these feed-backs and concerns will be detailed later in this paper. These two challenges motivated the development of this project.

2.4 Core Contributions of this Report

This project has two goals. First is to unify multiple graphics API based Taichi backends, in search of better compatibility while improving code maintainability. The second goal is to design an easy-to-use ahead-of-time compilation pipeline, to enable users to deploy their compiled Taichi programs outside of Python.

The first section of this report introduces a new GPU abstraction layer (later referred to as “Unified Device API”) that I designed and incorporated into Taichi, along with two conforming implementations based on two different backend APIs. This API interface is designed to be cross-platform and vendor-neutral, where it layers on top of existing graphics and compute APIs. After we introduce the Unified Device API, we will also detail the design and engineering of an accompanying adaptive code generation pipeline (later referred to as the “SPIR-V Codegen”). The combination

of the Unified Device API and the SPIR-V Codegen means that we can run Taichi programs on almost all GPUs on the market, many previously not available, such as mobile devices, AMD server GPUs, while addressing the maintainability and inconsistency problems of previous Taichi back-end implementations.

In addition to being able to support a wide range of GPUs, the second section of this report proposes a new ahead-of-time compilation (AOT) pipeline called “Taichi CGraph”. This new API would replace the original AOT pipeline, addressing the users’ concerns. We will talk about the design decisions and rationale behind the new API, and the user studies that guided this design.

3 Improving Taichi’s Compatibility and Maintainability

There are a few different backend APIs with compute capability available on consumer devices, including OpenGL, DirectX, Vulkan, Metal, etc. However, as there is currently no compute capable backend-API that is available on all mainstream consumer platforms (as we can see from Table 1), we will always need multiple Taichi backends to effectively support all the mainstream devices.

Recall that under the current Taichi backend design, each backend is API dependent with incompatible runtimes and code-generation pipelines, causing minimal code reuse. However, all of these backend APIs have a lot of similarities: many features are common between these backend APIs, and many others can be emulated to some degree. Many game engines and rendering middleware are using an abstraction layer, usually called Render Hardware Interface (RHI) so that the application can share code and run on different backend APIs. Although all the backend APIs use a different source or intermediate-representation (IR) format for the GPU kernels, shaders cross-compilation tools are available thanks to the cross-platform RHI work pioneered by game engines and rendering middlewares.

Since we are mainly targeting rendering and simulation workloads that utilizes the general purpose compute capability of GPUs, instead of traditional hardware rasterization workloads, we decided to create our own RHI called the Unified Device API, due to differences in resource management strategy, and drastically different programming styles between our workloads and typical game engines. Based on this RHI, we designed a common and adaptive Taichi backend code-generation pipeline called the SPIR-V Codegen, achieving much wider levels of compatibility while improving maintainability of Taichi backend code-base. In the next section, we will first discuss the design of the Unified Device API with relevant background information.

3.1 Designing the Unified Device API

3.1.1 Survey of current Graphics and Compute Backend APIs

In order to design a common abstraction that layers on top of the different backend APIs, we first need to find out the commonalities between them, and find out the differences so that we can adapt to all of them. We will be looking at the major graphics APIs available on all kinds of consumer devices: OpenGL / GLES, Vulkan, DirectX, Metal, and CUDA.

Usually the design of an abstraction layer will be focused on the lowest common denominator, but that might hurt performance and optimization potential. Therefore, we will also look at the unique features from different APIs, and try to keep these

	OpenGL 4.3	Vulkan	Metal	DirectX 11/12	Cuda
Android	Supported	Supported			
iOS		Translated	Supported		
macOS		Translated	Supported		
Windows	Supported	Supported		Supported	Supported
Linux	Supported	Supported		Translated	Supported

Table 1: Back-end API Compatibility Matrix

unique features optionally available in the Unified Device API. Table 1 lists the compatibility between different back-end APIs and platforms. Keep in mind that if an API is supported, it is not guranteed to be available: for example, Windows supports CUDA, but Windows devices without a Nvidia GPU will not support CUDA.

There are a few main distinguishing differences between these APIs, usually an API falls into a combination of categories. These differences will guide our API design strategies. The detailed per-API survey can be found in Appendix 5.1.

Work-submission Model There are two main types of work-submission models: immediate, and batched. Under an immediate mode API, all API calls are considered to be serial and dependency between API calls are managed automatically by the API. Under these APIs, you can assume each API call is dispatched immediately on the GPU, thus making them "immediate mode" APIs. The actual execution might be batched for optimization reasons, but these details are hidden under the API. The other type is batched, where usually the API provides "streams" and "command list". Different APIs use different terminologies, but the concept are the same. Each stream represents a queue on the GPU, where batches of commands are sent to the queue in bundles called "command list". This model provides more transparency and less overhead, and can usually be found in newer APIs. Keep in mind that some APIs support both modes of job submission.

Resource Model There are two main types of resource model: either managed and tracked by the API, or explicitly controlled by the user. Under an API that tracks resources, an API call to resource allocation and deallocation might be lazy. This means that the API will make sure when the GPU command executes, all resources used by the command will be available during the execution. Under an API that exposes explicit management, resouce managment API calls will be reflected directly on the GPU, and it is possible to accidentally destroy resources that is still in-use by the GPU. The explicit model allows advanced resource management techniques, while putting more management burden, previously on drivers provided by vendors, onto the programmers.

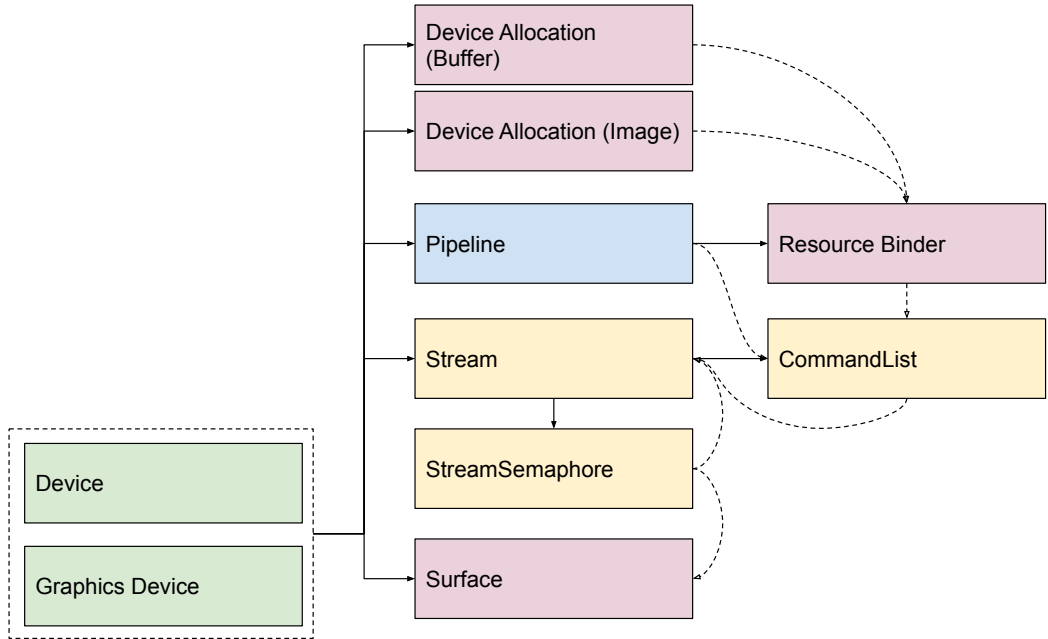


Figure 2: Unified Device API Objects

3.1.2 Specification of the Unified Device API

In this section, we will first outline the final specification of the Unified Device API, we will then detail the rationale behind this design in the next section.

The Unified Device API is presented in an object oriented form, where all the states are contained within objects, so that we can avoid any sort of global states. The most important object is the “Device” object, which represents a physical or logical device provided by a backend API, and all resources are allocated from Device.

The objects and the relationship between the objects we expose from the Unified Device API are shown in Figure 2. Resources can be created from the Device using `allocate_memory` and `create_image` calls. These calls result in a device resource handle called `DeviceAllocation`. A resource handle can be considered as a reference to a piece of GPU memory that contains either a raw buffer or a structured image.

In order to submit work to the GPU, the user will need to acquire a `Stream` object

from the Device object, which represents a single queue on the GPU. Recall that we mentioned in the survey of current APIs, multiple queues can execute concurrently, and commands inside each stream are guaranteed to start execution in-order but may finish out-of-order. Thus in-queue dependency needs to be controlled through barrier commands, while inter-queue dependency must be controlled by a `StreamSemaphore` object.

We mentioned that our abstraction uses batched execution, thus we can create `CommandList` from `Queue`, which represents a batch of commands. The user can also create `Pipeline` from `Device`, where a `Pipeline` represents a program that is run on the GPU. In graphics APIs, this is sometimes called shaders, in compute APIs, this is usually called a kernel. A GPU program can be bound to a `CommandList` by the `bind_pipeline` call, and the user can use the `dispatch` call to launch the bound program with a certain amount of threads. Resources can be passed into the GPU program through the use of `ResourceBinder`, which consumes `DeviceAllocation` handles.

3.1.3 Rationale

The goal of the Unified Device API is to be able to share code across all the backends, while having the ability to fully utilize optionally hardware features and optimizations, maintaining high efficiency if the backend API allows for it.

Under the previous Taichi architecture, each backend uses backend-specific code-paths, as shown in Figure 3.

Under the new Taichi architecture with Unified Device API shown in Figure 4, we now have a unified abstraction, providing isolation between code-gen and run-time, and thus enabling code sharing.

Given that batched execution and graph based execution can provide more optimization potential when supported by the hardware, thus we decided to use the concept of Streams and CommandLists for batched execution and multi-threaded job submission. On implementations using some backend APIs that are fully serial, or one that does not support threading on the CPU, these commands will be serialized onto a single command stream, thus maintaining compatibility, as a stronger semantics (fully serial) is always a subset of a weaker semantics (relaxed ordering).

Because APIs like CUDA, Vulkan, and DirectX 12 support multiple queues and concurrent execution of commands, the dependency between commands submitted to different queues need to be explicitly managed. Therefore, we expose the concept of binary Semaphore to the user to synchronize between command submission on different queues and different threads. When a batch of commands is submitted to a queue, it can signal a semaphore, and it can optionally wait on a series of semaphores

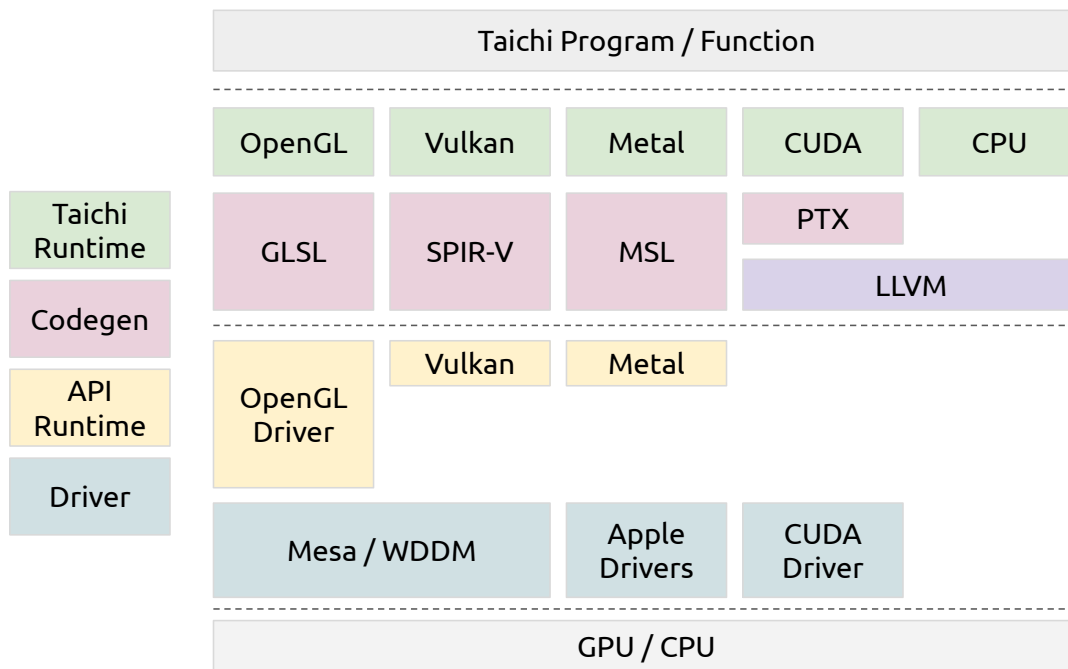


Figure 3: Previous Taichi Back-end Architecture

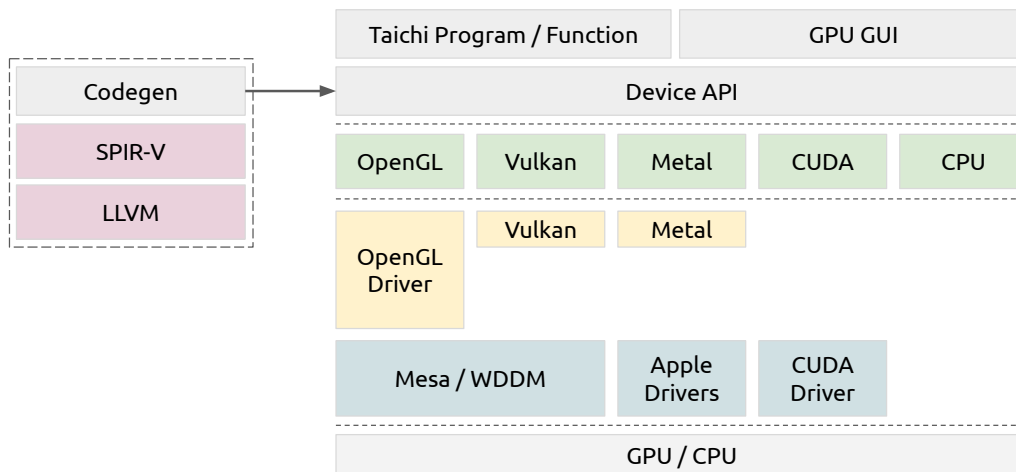


Figure 4: Taichi Back-end Architecture with Unified Device API

to begin GPU execution. On an API that does not support semaphores, their commands are fully serial anyways, we just need to make sure the multiple threads on the host are submitting jobs in-order.

We want to expose explicit memory management to the user as its behavior is predictable. Users have reported frustrations when using OpenGL where the actual resource management is hidden by the driver, so the user has no idea whether an opaque object is actually using GPU resources or not. However, unlike Vulkan or DirectX 12, we will want to provide resource tracking so that the users don't need complicated synchronization. Recall that in Vulkan and DirectX 12, when you deallocate a GPU resource, the API will do so immediately without waiting until the GPU finishes working on such resource. If we track the resource internally within the Unified Device API, the user will not need to worry about such a problem. The actual mechanism of resource tracking will be detailed in the implementation section.

Most importantly, we still want to fully utilize features available from the hardware, while being able to share code for the common feature set, thus the Unified Device API needs to support the concept of optional features and variable-capabilities. When the Unified Device API is initialized for a specific backend, all the supported features and limitation parameters will be queried, and common optional features between different backends will be exposed uniformly. If the same feature is optionally supported under multiple backend APIs, the Unified Device API will provide a uniform abstraction for such optional features.

3.1.4 Implementation

Two different implementations of the Unified Device API are created initially, one layered on top of the Vulkan API and the other on the OpenGL API, so that we can cover basically all platforms, while proving that the Unified Device API abstraction is actually cross-platform compatible. The community has later contributed a DirectX 11 implementation. We will mainly talk about the Vulkan implementation of the Unified Device API in this section, as it is the most complex in implementation, and Vulkan is the most feature-rich backend API.

Since Vulkan exposes physical devices directly to the user, we map each available Vulkan device on the system to a Unified Device API Device instance. Vulkan devices support “extensions” that describe optional capabilities of the hardware. This maps quite straight forward to the DeviceCapabilities concept from the Unified Device API, which is an attribute of a particular Device.

For resource allocation, because our API uses opaque object handles to map to different backends and resources are always allocated from a Device, we will maintain an internal map within the Device instance between the Unified Device API resource

handles and the respective backend Vulkan device objects. We mentioned earlier that under Vulkan and DirectX 12, it is possible to cause a fault by deallocating resources that are still in-use by the GPU, thus our implementation needs to track usage of all resources. These dependencies can not only exist between resources, but also exist temporally: resources referenced by an in-flight GPU command will need to remain available until the respective command finishes. Using strong references would not work as the dependency is complex, where multiple objects may depend on the same Vulkan resource, thus we decided to employ a reference counting scheme on all of our device objects and resources.

As the GPU runs concurrently with the host, we do not know whether a GPU command has finished execution, thus making it hard to track the life-time of dependent GPU resources. Since under the Unified Device API, all GPU commands are submitted in batches (in the form of `CommandList`), we can deduce that all the resources referenced by all the commands within the `CommandList` will be completed after the entire batch has completed. Thus, it makes sense to hold a temporary reference to all the objects used in `CommandList` within the `CommandList` object itself. After the `CommandList` is submitted to the `Queue`, all the references are then transferred to the `Queue`, as the user may want to release the `CommandList` that has already been submitted to the `Queue`. Tracking the completion of each individual command could be costly, thus we decided to use queue synchronization events to track command completion. This means that we can simply remove all the references to the used objects once there is a queue synchronization event. In theory this approach might cause over-allocation of GPU resources, as some commands in the queue may finish earlier. However in practice, we have yet to meet a workload that consumes a lot of memory, that does not have a semi regular queue synchronization event. In the future we can potentially introduce some heuristics that track individual `CommandList` using a large amount of resources, but the current implementation seems to be sufficient.

3.2 SPIR-V Codegen Pipeline

The Unified Device API abstracts away the API that is used to control the GPUs, but it does not provide a common format for GPU kernels, as each back-end API uses different formats and languages. Thankfully the game industry has found a solution to this: shader cross-compilation, where a target IR is cross-compiled to forms that can be accepted by all the different backend APIs. In this section, we will first introduce SPIR-V and the reason why we picked SPIR-V as the target GPU kernel intermediate representation (IR) for the Unified Device API.

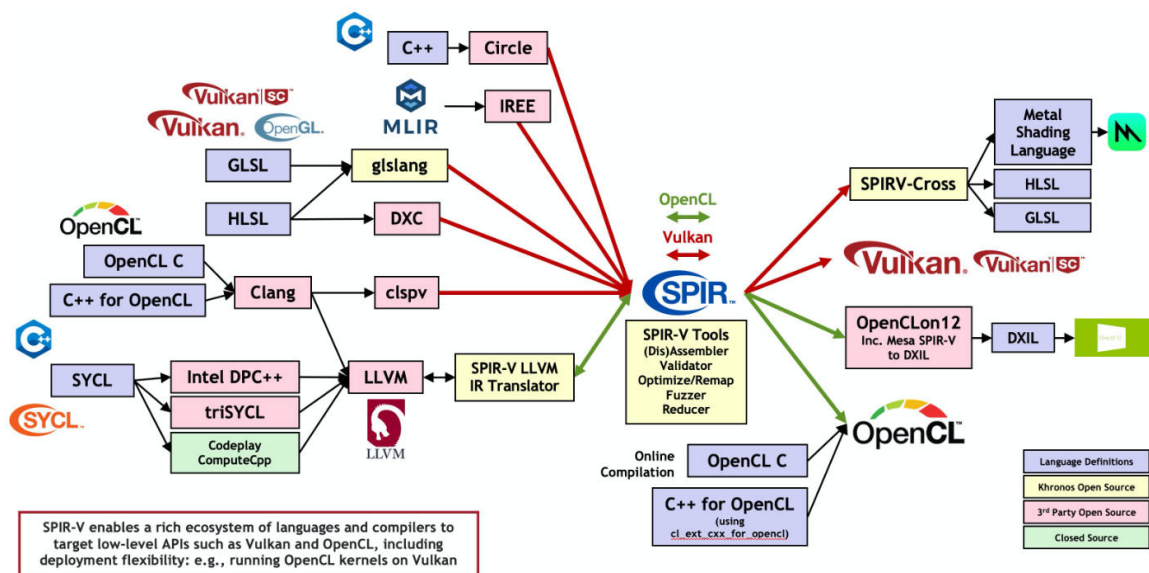


Figure 5: SPIR-V Ecosystem Map (Kessenich et al., 2022a)

3.2.1 SPIR-V and Cross-Compilation

There are multiple cross-compilation paths available in the industry. However if we are to achieve the widest range of cross-compilation targets, SPIR-V is currently the best solution from the industry. SPIR-V is a standard GPU intermediate representation (Kessenich et al., 2022b) developed by Khronos (the same organization that maintains OpenGL and Vulkan).

There are two benefits for choosing SPIR-V for our backend IR code-generation target: portability and efficiency. Figure 5 shows that if we use SPIR-V as our code generation target, there is at least a pathway to all the major graphics APIs including DirectX, Vulkan, OpenGL, etc. The ecosystem also provides existing tooling for SPIR-V such as SPIR-V validation and optimization. In addition, since SPIR-V is a Single Static Assignment form IR, we should be able to transform Chi-IR to SPIR-V relatively efficiently, and the backend compiler doesn't need to do as much work compared to parsing a text based source code, and we are not limited by the expressibility of GLSL or HLSL.

3.2.2 Implementation

The SPIR-V backend code-generation is implemented as an IR transform, which walks the original Chi-IR tree from Taichi, and transforms all the Chi-IR statements into SPIR-V statements. This transform process itself is quite straightforward, however translating Taichi data structures and memory accesses is tricky. Another tricky part is to support a wide spectrum of devices with very different capabilities, quirks, and

sometimes even bugs.

We will first discuss the code-generation of memory access statements. In Taichi, there are a few different types of memory: global memory in the form of SNode, global memory in the form of NDAarray, “block-shared” memory, and thread-local memory. SNode describes the data structure used in Taichi. It forms a tree using different types of container nodes, and the containers can then be mapped to multiple axes of an array / tensor. Currently the Vulkan backend only supports dense structures, which is simply translated by converting an n-dimensional index to a linear one. Block-shared memory and thread-local memory is an advanced concept and currently only used for optimization, thus the SPIR-V codegen has not supported that feature. We will focus on translating SNode and NDAarray memory access into SPIR-V.

In SPIR-V (and all the related graphics API, such as DirectX, Vulkan, and OpenGL), big chunks of GPU memory is bound to the GPU kernel in the form of “buffers” (The Khronos® Vulkan Working Group, 2022), in OpenGL it is called “SSBO” (Segal and Akeley, 2022). In the GLSL shading language and the SPIR-V specification, multiple sized arrays of different types are supported within a buffer, just like the “struct” layout in C / C++. However, when translating Taichi SNodes to SPIR-V struct, we met a graphics driver bug on the OpenGL backend: the Nvidia GLSL compiler will get stuck if a struct contains a large sized array. There’s an alternative that overcomes this driver bug, that is to use an unsized array. However, using an unsized array prevents us from using different types for an entire buffer, and we can not afford to map each SNode to a buffer, as each GPU kernel has a limit on the number of buffers used simultaneously. This limit is 8 in OpenGL, an especially small number. In Taichi there might be hundreds of SNode, for example when describing a field of matrices, and we know for sure that all these matrices can be packed into the same buffer. Thus we need to figure out a way to use a single buffer for all SNodes, while supporting any data types that can appear in the buffer.

If you are familiar with C or C++ programming, the concept of pointers specifically, you might be confused why we need to do this. The root cause is that SPIR-V does not support generic or numeric pointers in the compute shaders profile. This means that all pointers are opaque, and must be acquired through linear indexing of a buffer of specific type, and there is no pointer casting support. However we can still overcome this limitation through aliasing. Aliasing happens when the same memory (same buffer) binds to different binding points of a kernel. Think of this as passing the same memory pointer to a function multiple times, just that each time with a different pointer type. This is not prohibited in OpenGL, although it does not provide sufficient specification on the behavior, but this is a defined behavior under Vulkan Memory Model, which means in practice this is supported on all the GPUs that we can find, we just need to be careful of visibility between memory operations of different types. This means a C / C++ style pointer system can be simulated,

as long as all the pointers can be accessed within the same contiguous chunk of memory. We mentioned that our code generation pipeline is adaptive, and thankfully the backend API designer has realized the importance of supporting raw pointers in GPU kernels, thus on Vulkan backends where the driver exposes the “Buffer Device Address” capability, we can translate pointer operations directly through pointer-to-numerical casts and its reverse.

Another particularly tricky part of the code-generation is the translation of atomic operation. Atomics are extremely important in particle or position based simulation algorithms, for example Position Based Dynamics, or Material Point Method, thus in Taichi atomics operations are guaranteed to be supported. However, many GPUs do not support atomic operation on floating-point values. When the code-generation pipeline detects that the GPU supports native floating point atomics, we can directly translate these atomics. When otherwise, we will simulate this through atomic compare-and-swap operations. Since integer atomics are always supported, we use the pointer aliasing trick mentioned before that effectively casts a floating point pointer to an integer pointer with the same width. Then we can use the compare-and-swap loop to perform the atomics operation.

3.3 Evaluation

The original motive of implementing a unified GPU abstraction is to reuse as much code as possible while reducing code maintenance difficulty as we expand to more and more backends. Currently as of writing of this report, we have a cross-platform Vulkan backend that is powered by the Unified Device API, and we have a prototype branch replacing the legacy OpenGL backend with Unified Device API, which eliminates the GLSL code-generation and OpenGL runtime code, replacing them with the adaptive SPIR-V codegen and unified runtime. This process eliminated around 1800 lines of OpenGL specific code, while adding about 600 lines for the new abstraction layer, resulting in a net decrease of 1200 lines. More excitingly, we verified that the API is indeed portable and maintainable as a community member contributed about 900 lines of code, consisting entirely of an DirectX 11 implementation of Unified Device API, and we now have complete DirectX 11 support in Taichi. Before this work, the contributor would not only need to know how to use the specific backend API, but also know all the details on the compilation pipeline and the program transform. Now, people knowledgeable with compilers can work efficiently without getting bogged down by API specifics, and people with a rendering background can contribute to Taichi without the need to understand compilers.

In addition to the JIT runtime, with the work on ahead-of-time compilation, we are able to use the generated SPIR-V code on many devices, further confirming that our code-generation pipeline is also highly portable. We have successfully got our MPM-MLS based fluid simulation code running on all sorts of device and API combinations:

iPhone with Metal, Android phones with OpenGL and Vulkan running on many GPU vendors (Arm Mali, Imagination PowerVR, Qualcommn Adreno), Linux with OpenGL and Vulkan, Windows with OpenGL / Vulkan / DirectX 11, and finally macOS with Metal and Vulkan through an API translation layer. We will detail our ahead-of-time compilation pipeline in the next section.

4 Running Taichi Programs without Python

Users from the industry wanted to deploy simulation and special effects written in Taichi into production-level applications. For example, short-video platform Kuaishou shipped a material-point-method based fluid simulator for adding 2D water effects into their videos. Other industry users are also interested in doing so, but due to confidentiality reasons we can not disclose their identity and their intended applications of Taichi programs. Taichi had an ahead-of-time compilation system to accompany the need of these industry users, but the previous AOT API had to generate backend-specific modules as each back-end is independent. We hope that based on our unified GPU abstraction and our cross-platform code generation can improve on the previous AOT system.

4.1 Motivation for Ahead-of-Time (AOT) Compilation

It is great that Taichi is embedded into Python where users can write extremely simple code that runs efficiently on the GPU and CPU. However this is also a limitation of such design. The user has to use Python in order to use Taichi, and there will always be an initial Just-in-Time (JIT) compilation overhead, which can be costly for less powerful mobile platforms. Keep in mind that Taichi is a DSL that only compiles a subset of Python syntax, thus the user still needs to write control logic that invokes these high-performance kernels. Unfortunately, these control logic are limited by the Python interpreter and its runtime overhead. If the program is not written carefully, there can be significant Python overhead.

In addition to efficiency concerns, many users that want to use Taichi may not be able to deploy a Python program at all in their production environment. For example, the iOS AppStore guideline used to not allow the use of JIT at all (Apple Inc, 2014), while current guidelines does not allow "downloaded" code (Apple Inc, 2022). The production environment is ultimately dominated by the constraints of the project, and that usually results in the users wanting to be able to invoke their Taichi kernel from C or C++, which ideally are ahead-of-time compiled instead of JIT compiled.

Due to these considerations, Taichi was exploring ahead-of-time compilation options when the legacy OpenGL and Metal backend was added. In the next section we will introduce the current AOT API, and then we will introduce the user study on the current AOT API.

4.2 Current AOT API

Recall the JIT pipeline of Taichi, since we don't perform any interpretation before JIT, we should be able to simply take the JIT compiled result and serialize them as binaries, with accompanying descriptions that informs the user how these GPU

kernels should be invoked. This is intuitively what the current AOT API does.

Under the current API, the user will create an AOT Module, and Taichi kernels can be explicitly added into the module. When the user finalizes the module, all of the kernels in this module will be JIT compiled. Here is an example usage of the current API that exports a set of kernels for a Material-Point-Method based fluid simulation program:

```
1 def export():
2     # Initializes an Ahead-Of-Time Compilation module for the
   Vulkan backend API
3     m = ti.aot.Module('vulkan')
4     # Add the data fields / states of the simulator program
5     m.add_field('x', x)
6     m.add_field('v', v)
7     m.add_field('C', C)
8     m.add_field('J', J)
9     m.add_field('grid_v', grid_v)
10    m.add_field('grid_m', grid_m)
11    # Add the Taichi kernels used by the simulator program
12    m.add_kernel(init)
13    m.add_kernel(substep)
14    # Compile and export the results
15    filename = 'mpm88'
16    m.save('./data/aot/mpm88/', filename)
```

Using this API, the user can export all the Taichi kernels used in their Taichi program within Python, and the AOT compiler will dump out the compiled programs into their respective format of their backend. Notice here the AOT system is backend-dependent: that is to say if the user wants to use OpenGL as the compute API in their program, they need to initialize the 'opengl' AOT module. If the user uses another backend, the AOT compilation result will be different, and they need to be used differently.

After the modules are exported and loaded by the user in their application, they can now launch these Taichi kernels. Here is a snippet of doing so for the aforementioned fluid simulator program (more specifically, the code for calling the "init" Taichi Kernel):

```
1 private void init() {
2     glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, root_buf);
3     glBufferData(GL_SHADER_STORAGE_BUFFER, root_buf_size, null
   , GL_DYNAMIC_COPY);
4     glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1,
   global_tmp_buf);
```

```

5     glBufferData(GL_SHADER_STORAGE_BUFFER, 80, null,
GL_STATIC_COPY);
6     glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, arg_buf);
7     glBufferData(GL_SHADER_STORAGE_BUFFER, 64*5, args,
GL_STATIC_READ);
8     Ndarray[] init_ndarrays = programs[0].getNdarrays();
9     Kernel[] init_kernel = programs[0].getKernels();
10    for (int i = 0; i < init_ndarrays.length; i++) {
11        glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
init_ndarrays[i].getBind_idx(), init_ndarrays[i].getSsbo())
12    ;
13    }
14    for (int i = 0; i < init_kernel.length; i++) {
15        glUseProgram(init_kernel[i].getShader_program());
16        glMemoryBarrierByRegion(GL_SHADER_STORAGE_BARRIER_BIT)
17    ;
18        glDispatchCompute(init_kernel[i].getNum_groups(), 1,
19        1);
20    }
21    for (int i = 0; i < init_ndarrays.length; i++) {
22        glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
init_ndarrays[i].getBind_idx(), 0);
23    }
24 }

```

Keep in mind that in Python, calling the "init" Taichi kernel is basically the same syntax as calling a normal Python function called "init". However as we can see, in the AOT version things are not so simple, and we will return to this topic later in this paper.

4.3 Target Users of the AOT API

Before we do any user research or collecting our feedback, we need to clearly define our users. Different target users lead to very different design constraints. We identified two types of users that this API would target. First one are "production engineers", whose job is to take algorithms prototyped in Python with Taichi or PyTorch and try to deploy these programs into an application. The other type of users we identified are researchers and engineers who want their Python Taichi application to run more efficiently by reducing the Python overhead and kernel launching overhead, which is proven to be significant (Macklin 2022).

For the first kind of users, they usually know their target deployment environment very well: if they are shipping a game, they would be more familiar with the back-end graphics APIs. This group of users would care about efficiency, as almost every

production application has clearly defined performance goals, and otherwise they would directly use Python. However, this group of people may not know all the details about the algorithms they are porting, as usually these are separate jobs for different people (algorithm engineers and production engineers).

For the second kind of users, Macklin’s talk on Warp (Macklin, 2022) has some quite convincing arguments: with smaller simulation programs, constant overhead dominates, which in Taichi’s case mostly consists of Python side overhead. This group of users knows all the details of their algorithms, but might not have as much experience in optimizing for lower overhead and latency. They would want to improve the efficiency of their code, whether to iterate faster by shrinking down the runtime, or trying to approximate the performance of their algorithms when deployed without being overshadowed by constant overhead.

4.4 User Study of the Current AOT API

Since Taichi is an active project with many industry users using it, we closely followed a few industry use cases and collected a lot of observational data and direct user feedback. We want to know the usability of the current API, and we wish to find out where are the major usability bottlenecks.

4.4.1 Motivating Tasks

We have two groups of users from the industry wanting to use Taichi as a programming language to develop special effects for their mobile software. One group want to deploy a fluid simulation program into their existing mobile communication application, the other group want to develop a new application that deploys a various suite of Taichi programs for animated interactive desktop.

Both tasks are similar in a sense that both requires a way to export a Taichi program into a non-Python environment, while being able to have the program run efficiently as it is a mobile platform with limited compute capability.

There are two slight differences: for one user, as it is deploying in an existing application, there is a much stricter limitation, such as the backend API selection, performance requirements, and software dependency limitations. The other group is developing a new application, thus there is less limitations as they don’t have an existing established code-base to worry about.

4.4.2 User Data

We closely tracked the development progress with these two groups of users so that we know how much time is taken for the user to complete different parts of this task. In

addition to observation data, we collected direct feed-backs from the users, including comments on their experience, the types of programs they are trying to create and deploy in Taichi, and the features they wanted the most.

First observation is that the user took majority of their time trying to debug their exported kernels. Keep in mind that these programs are already working within Python, which means that our user is having trouble using the generated modules. Granted some of debugging is related to our compiler code-generation quality that failed on certain edge-case devices, however the process of debugging these AOT output is hard by itself, as the debugging must be done on the target device with an Ahead-of-time compilation output, which lacks the ease-of-debugging attribute of a JIT compiled program. Since the user does not have access to the Python runtime during AOT, the user all had to re-implement a run-time system for our GPU kernels within their target application. This also took precious development time, while causing bugs to arise that we observed.

The second observation is that the user might confuse our DSL’s semantics with Python’s semantics. As we are embedded within Python, the user might assume features such as global variables and states. As they are not possible to capture in AOT, the user gets confused when they find out their programs, written under Python’s semantics, do not work.

The final observation is that a lot of the time a Taichi program may still have large amount of critical Python code, either used simply to invoke a series of Taichi kernels, or in some advanced case act as a control logic. One example would be the conjugate gradient (CG) solver in an implicit Finite Element Method soft body simulator, listed in Appendix 5.3

In this case, each function call with the CG solver body is a Taichi kernel invocation, and the function will keep iterating until the value “alpha” drops below a threshold. The current AOT pipeline only generates all the Taichi kernel used, but neither does it compile the CG solver function body, as it is a Python function not a Taichi kernel, nor does it serialize this termination logic. When the user wants to integrate this CG based soft-body solver into a game, the user needs to rewrite the CG solver body in their target language, such as C++ or Java.

4.4.3 Conclusions from User Data

It is clear that an ideal AOT compilation solution not only needs to export the GPU kernels, but also serialize the Python side control logic, that can be loaded and invoked later in a non-Python environment. In addition, the user ideally should be able to use the same Taichi runtime in their target language of choice, removing the inconsistency between Taichi’s implementation and the user’s own implementation.

In addition to these critical features, we also wish that the new AOT API will be able to help the users understand the features and limitations of Taichi, as we observed many times that user confused Taichi’s DSL with generic Python, where they are confusing Python’s semantics with Taichi’s semantics when writing a program. We also want to have our API to be able to guide users to write efficient code, since in one group of users we observed that some API usage caused a lot of performance inefficiency, while the potential pitfalls of such usage patterns are not obvious.

4.5 Designing the new AOT API

To achieve our design goal of an ideal AOT compilation solution, we will introduce a compute-graph API called Taichi CGraph, that is similar to CUDA Graph, TorchScript (Paszke et al., 2019), and the data flow graph powering TensorFlow (Abadi et al., 2017), where a static graph of operations are assembled and can be invoked later. In addition to that, we will add a limited degree of dynamic control flow, inspired from the idea introduced by TensorFlow’s team (Yu et al., 2018), to support use cases such as the stop condition of an iterative solver. Different from CUDA Graph, this graph can be serialized and then loaded by the user through a runtime library.

4.5.1 Design of Taichi Compute Graph

Since Taichi AOT is a production level tool, the Taichi Compute Graph (CGraph) must be realistic and attainable. Thus, before we design the language front-end, we decided to design the back-end framework first. This way we can make sure that we indeed meet the goals we have set out to achieve.

Comptue Graph and Graph Nodes The compute graph is designed to have extremely simple yet efficient dynamic control flows, as supporting complex dynamic control flow is proven to be a complex task (Yu et al., 2018). The pipeline starts with an “AOT Module”, and each module can contain multiple graphs, each graph can be invoked independently. Each graph is entirely pure, as all states are passed into the kernel as parameters. You can think of this as passing a reference to a state into a function, and this saves us from needing to capture global states for each graph, where these global states might not exist when the AOT module is compiled and loaded from another application.

A graph is composed from three types of nodes: dispatch node, sequential node, and conditional node. A “dispatch” node specifies a kernel invocation operation, which means when the node is evaluated, it will invoke a Taichi kernel with a specific set of parameters. These parameters can either be compile-time constants, or they can be references to dynamic states that are passed in when the user invokes the entire graph. A “sequential” node contains a series of children, and these child nodes are evaluated

in a sequential order. With dispatch nodes and sequential nodes, the Taichi CGraph is very similar to the command list concept in Vulkan / Metal or the stream capture concept in CUDA, where each graph is essentially a series of kernel invocations.

Next we will discuss the conditional node. This is currently the only dynamic control flow node that we support. This node takes a reference to a state as a predicate. When the predicate evaluates to true, the true child-node will be evaluated, otherwise the false child-node will be evaluated. Conditional nodes can be mapped relatively easily to the GPU, as there are ways to achieve this feature entirely within the GPU. In Taichi's Python JIT mode, when the user writes an if statement that depends on a state variable on the GPU, or a value yielded from a previous GPU kernel, the Taichi runtime needs to synchronize the host and the GPU, so that it can observe the predicate. After the predicate is observed, the Python interpreter will execute the if statement and enqueue more work onto the GPU depending on the code path taken. However this process involves a large overhead: not only does it take time to synchronize the CPU and the GPU, it also exposes the ramp-up and tail latency of a GPU parallel workload, as the GPU waits for all previous tasks to fully finish and new tasks would need to ramp up. Fortunately, we should be able to statically expand both sides of the conditional node as recursion is not allowed. With backends that support conditional command execution, we can simply add a predicate for the kernel dispatch command. With backends that do not support this feature, we can move the predicate into the kernel, where the kernel will still be dispatched by the GPU but will immediately end if the predicate is met. This means that we should be able to avoid host-GPU synchronization entirely if we only support the conditional node. We have tested that with a workload like an iterative solver, the synchronization overhead is larger than the overhead of conservatively dispatching GPU kernels even if a lot of them are skipped during execution.

Front-end Graph Construction Language In terms of front-end language design, we expect that an explicit API will be significantly harder for the user to use. However, explicit API might be better at communicating the system's capability, as it does not assume the syntax of Python, thus preventing the user to confuse its semantics with Python's. Therefore, we created two prototypes of the AOT API, one implicit and one explicit. Example programs can be found in Appendix 5.4

We implemented functional prototypes for both implicit and explicit API design. Although these functional prototypes can not verify the performance of these APIs, they are enough for a usability study. A formative study is run to decide which API we should settle on for the Taichi Compute Graph.

4.5.2 Formative Study: Implicit or Explicit API

Because we need to evaluate which style of API would be more ideal for the Taichi Compute Graph API, we would first want to know whether the user can accept the limitations of the explicit Compute Graph API. Second, we want to find out the best way to communicate the limitations of our AOT system to the user so that they won't be surprised if something does not work.

We asked a few users, including people from both groups of target users we have identified, to try to use these new APIs to construct two classic simulation algorithms: one is simple and linear (Material-Point-Method based fluid simulator), the other one requires handling of a dynamic branch (Finite-Element-Method based soft-body simulator), which we suspected will be the pain point of an explicit API. Since we are targeting a niche field, we can not gather enough participants, thus our data is be empirical and might be biased to a certain group of people.

User Data For the implicit API, it does seem that it is very easy to learn: The user writes a Python function normally, and that function gets automatically compiled to different platforms. However, the user quickly stumbled when trying to convert the FEM program to Compute Graph API. Because the limitations in our dynamic flow control, normal for loops and while loops must be static, while the user tried to use a loop statement with a break statement dependent on the output of a kernel for the iterative solver, which makes it a case of dynamic flow control that we do not support, and the compiler thrown an error.

We observed that the users took some significant amount of time on trying to get a version of the code that adheres to our dynamic flow control rules. In addition to that, the user also met a few errors on using Python global variables as mutable state variables, however these are relatively easily identified and fixed.

On the other hand, the test of the explicit Compute Graph API also went quickly for the simple linear MPM program. It seems that for users to start working with the explicit API, they would need some amount of time looking at sample programs and API references to fully understand the API. The explicit graph construction API requires the user to have a prior understanding of representing a program as a graph of operations on data. However it does seem that the users we are targeting are all able to quickly learn the concept of the API if given sufficient documentation. The user also reported that they are fine with the API as long as it is clearly documented. In addition, one interesting thought we received from the user for the explicit compute graph API is that they want to see a visualization of the graph they have constructed, however this is not brought up for the implicit approach, perhaps as the compute graph for the implicit DSL is actually hidden and generated by the compiler.

When we move to the finite element method program, the user initially did not figure out a way to make it work with the limited graph nodes we provide in the DSL. We needed to show an example of converting a loop with a stop condition to a unrolled loop with predication (conditional execution). However after showing them an example, they are able to understand this concept quickly and write their own version. In addition, they expressed that they understand why these limitations are in place once we explained the back-end performance and efficiency implications, and the users are willing to work with these limitations as long as they know the performance or efficiency is guaranteed.

Findings The most intriguing findings from this study is that it seems users prefer the language and API itself to communicate its capability and limitations directly, rather than compiler messages. Users get visibly annoyed or confused when dealing with compiler messages, even if the message are clear, the users still need to go over their code and rewrite many sections of them. This seems not efficient, and a good DSL design should discourage or disallow illegal or inefficient use through its semantics. Our users have referenced their experience using Rust and C++, where after writing a lot of code that seemingly should work, the compiler is throwing out unexpected error messages. Perhaps these messages are entirely user error and the user did not check the documentations clearly, but in terms of user experience the user much prefer their code can run directly the first time, even if it means it takes them longer to write the code in the first place. Based on this critical finding from the study, we decided to settle on the explicit compute graph API. It also highlighted the importance of proper documentation and accessible examples, and even visualizations to help the programmers understand their tools.

4.6 Next Steps

We are currently in the process of implementing our explicit compute graph API in Taichi, we hope that after we have a fully functioning version that actually has all the run-time capability to use in a C++ or Java project, we would like to create a sample project and use that process to perform another user testing to evaluate whether the end product still works as well as we envisioned it to be.

In addition, we think that the philosophy of using semantics and syntax to enforce limitations instead of relying on compiler error messages is interesting, and we think this philosophy should be further tested in more DSLs or even general purpose programming languages. Because our target users are very specific and niche, it is possible that our findings only represent a very small group of people. However it also makes sense for a specialized programming tool to adopt this philosophy, as these specialized tools usually have a lot more restrictions than a general purpose language. Thus we think further work can be done to test this philosophy widely and quantitatively.

References

- Martín Abadi, Michael Isard, and Derek G. Murray. 2017. A Computational Model for TensorFlow: An Introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Barcelona, Spain) (*MAPL 2017*). Association for Computing Machinery, New York, NY, USA, 1–7. <https://doi.org/10.1145/3088525.3088527>
- Apple Inc. 2014. App Store Review Guidelines. <https://web.archive.org/web/20141214145424/https://developer.apple.com/app-store/review/guidelines/>.
- Apple Inc. 2022. App Store Review Guidelines. <https://developer.apple.com/app-store/review/guidelines/>.
- Apple Inc. 2022. MTLCommandQueue — Apple Developer Documentation. <https://developer.apple.com/documentation/metal/mtlcommandqueue>.
- John L Hennessy and David A Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Elsevier.
- Yuanming Hu. 2020. The Taichi Programming Language. In *ACM SIGGRAPH 2020 Courses* (Virtual Event, USA) (*SIGGRAPH '20*). Association for Computing Machinery, New York, NY, USA, Article 21, 50 pages. <https://doi.org/10.1145/3388769.3407493>
- Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2019a. DiffTaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935* (2019).
- Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019b. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* 38, 6, Article 201 (nov 2019), 16 pages. <https://doi.org/10.1145/3355089.3356506>
- Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. Motivation for and evaluation of the first tensor processing unit. *ieee Micro* 38, 3 (2018), 10–19.
- John Kessenich, Dave Baldwin, and Randi Rost. 2019. The OpenGL Shading Language. <https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf>.
- John Kessenich, Boazouriel, and Raun Krisch. 2022a. SPIR-V Overview. <https://www.khronos.org/spir/>.
- John Kessenich, Boazouriel, and Raun Krisch. 2022b. SPIR-V Specification. <https://www.khronos.org/registry/SPIR-V/specs/unified1/SPIRV.pdf>.

- Khronos OpenCL Working Group. 2022. The OpenCL™ C Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf.
- Haidong Lan. 2022. Is Taichi Lang comparable to or even faster than CUDA? <https://docs.taichi-lang.org/blog/is-taichi-lang-comparable-to-or-even-faster-than-cuda>.
- Miles Macklin. 2022. Warp: A High-performance Python Framework for GPU Simulation and Graphics. <https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41599/?playlistId=playlist-2f49ed16-dbc2-43c7-9c22-eeeecede6c139>.
- B Neelima and Prakash S Raghavendra. 2010. Recent trends in software and hardware for GPGPU computing: a comprehensive survey. In *2010 5th International Conference on Industrial and Information Systems*. IEEE, 319–324.
- Nvidia Corporation. 2022. CUDA Runtime API. https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.
- Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
- Mark Segal and Kurt Akeley. 2022. The OpenGL® Graphics System: A Specification. <https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf>.
- Taichi Graphics LLC. 2022. Taichi, a perfect programming framework for computer graphics courses. <https://www.taichi-lang.org/user-stories/>.
- TensorFlow. 2022. Install TensorFlow with pip. <https://www.tensorflow.org/install/pip>.
- The Khronos® Vulkan Working Group. 2022. Vulkan® 1.3.213 - A Specification. <https://www.khronos.org/registry/vulkan/specs/1.3/pdf/vkspec.pdf>.
- Chris Wyman, Shawn Hargreaves, Peter Shirley, and Colin Barré-Brisebois. 2018. Introduction to DirectX Raytracing. In *ACM SIGGRAPH 2018 Courses* (Vancouver, British Columbia, Canada) (*SIGGRAPH '18*). Association for Computing Machinery, New York, NY, USA, Article 9, 1 pages. <https://doi.org/10.1145/3214834.3231814>
- Chenle Yu, Sara Royuela, and Eduardo Quiñones. 2020. OpenMP to CUDA Graphs: A Compiler-Based Transformation to Enhance the Programmability of NVIDIA Devices. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems* (St. Goar, Germany) (*SCOPES '20*). Association for Computing Machinery, New York, NY, USA, 42–47. <https://doi.org/10.1145/3378678.3391881>

Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. 2018. Dynamic Control Flow in Large-Scale Machine Learning. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) (*EuroSys '18*). Association for Computing Machinery, New York, NY, USA, Article 18, 15 pages. <https://doi.org/10.1145/3190508.3190551>

5 Appendix

5.1 Survey of GPU APIs

OpenGL OpenGL is an graphics API maintained by Khronos Segal and Akeley (2022), and since we are focusing on compute, we will need the compute shaders capability which is introduced into OpenGL through OpenGL 4.3. The OpenGL standard also has a mobile subset called OpenGL ES (GLES) that is used in mobile devices. However compute capabilities are introduced through GLES 3.1, which means it is not available to Apple devices. OpenGL is also an immediate mode API, which means that all commands are issued immediately with each OpenGL API call. In addition, OpenGL is single threaded only, and all commands and APIs are implicitly synchronized, which means the user does not need to manage any concurrency, keeping in mind that a GPU actually executes commands concurrently with the CPU. This approach means that it can be less efficient as the programmer has no control over synchronization and the performance will entirely depend on the graphics drivers’ best effort optimization. In terms of algorithmic support, basic integer atomics operations are supported, but native atomic floating point operations are only supported on Nvidia cards with its proprietary extensions. This is a trend that we will see down the line with other APIs. As many simulation algorithms require floating point atomics, we will need to simulate it through atomic compare-and-swap operations.

CUDA Another immediate mode API in our list is CUDA, however it recently introduced a graph based API as well called CUDA Graph, where commands are composed into a graph, and the entire command graph is submitted to the GPU queue as a batch. The driver can utilize this information to optimize the command stream, thus achieving higher performance Yu et al. (2020). In addition, using a batched command submission reduces the time spent on communications overhead between the CPU and the GPU. Batched command submissions is a trend that we will see in other more recent APIs. CUDA also supports multi-threaded command execution, and the user can run parallel commands on the GPU through the use of multiple command streams Nvidia Corporation (2022). CUDA also has the capability to enqueue the GPU kernel directly from the GPU instead of needing the CPU to enqueue all the operations. CUDA is the only API listed here that supports memory pointers to both global memory and device local fast scratchpad memory, function pointers, and floating point atomics natively.

DirectX 12 and Vulkan DirectX 12 and Vulkan are largely similar. They are both new APIs designed to reduce CPU overhead by giving the programmers more direct synchronization and memory allocation control, with batched GPU submission in the form of “queue” and “command buffers” The Khronos® Vulkan Working

Group (2022). The commands submitted in one queue are guaranteed to start in order but may complete out-of-order without explicitly using barriers, and there is no ordering guarantee between different queues, thus requiring the use of semaphores. The Khronos® Vulkan Working Group (2022). Another feature of these APIs are that memory and resource allocation are explicitly managed by the host, where the host will need to make sure the memory GPU is using is always resident in the GPU memory. This complicates resource management and we will want to abstract low level resource management away from the user. The abstraction and resource strategy will be detailed in the implementation section. In addition to these API features, Vulkan also supports extensions and a variety of configurations, where the hardware supported feature set might greatly differ between devices, and there is not a widely adopted “baseline” feature set that we can target. This means that our code generation pipeline must be adaptive to the hardware: utilizing advanced features if we can, while keeping it compatible with the bare minimal device. These optional advanced features include different width arithmetics (int8, int16, float16, etc.), advanced atomics and barriers, even buffer memory pointer support.

DirectX 11 and Metal Metal and DirectX 11 are similar as they provide some lower-level access, but the API provides resource tracking ability so that the programmer does not need to worry about it Apple Inc. (2022). They also support batched execution, called `Encoder` in Metal and `CommandList` in DirectX 11. The implementation of RHI layered on top of these backends are fairly straight forward.

5.2 Taichi program producing an animation of the 2D Julia Set

```

1  import taichi as ti
2
3  ti.init(arch=ti.gpu)
4
5  n = 320
6  pixels = ti.field(dtype=float, shape=(n * 2, n))
7
8  @ti.func
9  def complex_sqr(z):
10     return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])
11
12 @ti.kernel
13 def paint(t: float):
14     for i, j in pixels: # Parallelized over all pixels
15         c = ti.Vector([-0.8, ti.cos(t) * 0.2])
16         z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
17         iterations = 0

```

```

18     while z.norm() < 20 and iterations < 50:
19         z = complex_sqr(z) + c
20         iterations += 1
21         pixels[i, j] = 1 - iterations * 0.02
22
23 gui = ti.GUI("Julia Set", res=(n * 2, n))
24
25 i = 0
26 while gui.running:
27     paint(i * 0.03)
28     gui.set_image(pixels)
29     gui.show()
30     i = i + 1

```

5.3 Conjugate-gradient Solver for Implicit FEM written in Taichi

Python side control logic shown here. All function calls within are Taichi kernel invocations.

```

1 def cg(it):
2     get_force(x, f, vertices, gravity[0], gravity[1], gravity
3     [2])
4     get_b(v, b, f)
5     matmul_edge(mul_ans, v, edges)
6     matmul_edge(mul_ans, v, edges)
7     add(r0, b, -1, mul_ans)
8
9     ndarray_to_ndarray(p0, r0)
10    dot2scalar(r0, r0)
11    init_r_2()
12    CG_ITERS = 10
13    for _ in range(CG_ITERS):
14        matmul_edge(mul_ans, p0, edges)
15        dot2scalar(p0, mul_ans)
16        update_alpha(alpha_scalar)
17        add_scalar_ndarray(v, v, 1, alpha_scalar, p0)
18        add_scalar_ndarray(r0, r0, -1, alpha_scalar, mul_ans)
19        if alpha_scalar < eps:
20            break
21        dot2scalar(r0, r0)
22        update_beta_r_2(beta_scalar)
23        add_scalar_ndarray(p0, r0, 1, beta_scalar, p0)
24    fill_ndarray(f, 0)
25    add(x, x, dt, v)

```

5.4 Example Programs of the Compute Graph API

5.4.1 Example program using the implicit Compute Graph API

```
1 from cgraph import cgraph, cond, set_cgraph_dry_run
2
3 ti.init(arch=ti.vulkan)
4
5 @ti.kernel
6 def kernel1() -> ti.int32:
7     ...
8
9 @ti.kernel
10 def kernel2():
11     ...
12
13 @ti.kernel
14 def iter_solver():
15     ...
16
17 @cgraph
18 def run_sim(template_cond):
19     if template_cond:
20         kernel1()
21         cond_a = kernel2()
22         for i in range(50):
23             cond_a = cond(cond_a, iter_solver, None)
24
25 # Note here the arguments are passed in from the beginning
26 run_sim_compiled = run_sim(True)
27 # Invoking the compiled arguments uses no arguments, making it
28 # clearer that all
29 # function arguments are "template arguments"
run_sim_compiled.invoke()
```

5.4.2 Material Point Method fluid simulator using an explicit Compute Graph API

```
1
2 # Compile Time
3 import taichi as ti
4 import explicit_cgraph as cgraph
5 from explicit_cgraph import Arg
6
7 n_particles = 8192
8 n_grid = 128
```

```

9 dx = 1 / n_grid
10 dt = 2e-4
11
12 p_rho = 1
13 p_vol = (dx * 0.5)**2
14 p_mass = p_vol * p_rho
15 gravity = 9.8
16 bound = 3
17 E = 400
18
19 @ti.kernel
20 def substep_reset_grid(grid_v : ti.any_arr(element_dim=1),
    grid_m : ti.any_arr()):
21     for i, j in grid_m:
22         grid_v[i, j] = [0, 0]
23         grid_m[i, j] = 0
24
25 @ti.kernel
26 def substep_p2g(x : ti.any_arr(element_dim=1),
27     v : ti.any_arr(element_dim=1),
28     C : ti.any_arr(element_dim=2),
29     J : ti.any_arr(),
30     grid_v : ti.any_arr(element_dim=1),
31     grid_m : ti.any_arr()):
32     for p in x:
33         Xp = x[p] / dx
34         base = int(Xp - 0.5)
35         fx = Xp - base
36         w = [0.5 * (1.5 - fx)**2, 0.75 - (fx - 1)**2, 0.5 * (fx
37             - 0.5)**2]
38         stress = -dt * 4 * E * p_vol * (J[p] - 1) / dx**2
39         affine = ti.Matrix([[stress, 0], [0, stress]]) + p_mass
40         * C[p]
41         for i, j in ti.static(ti.ndrange(3, 3)):
42             offset = ti.Vector([i, j])
43             dpos = (offset - fx) * dx
44             weight = w[i].x * w[j].y
45             grid_v[base + offset] += weight * (p_mass * v[p] +
46                 affine @ dpos)
47             grid_m[base + offset] += weight * p_mass
48
49 @ti.kernel
50 def substep_update_grid_v(grid_v : ti.any_arr(element_dim=1),
51     grid_m : ti.any_arr()):
52     for i, j in grid_m:

```

```

49     if grid_m[i, j] > 0:
50         grid_v[i, j] /= grid_m[i, j]
51     grid_v[i, j].y -= dt * gravity
52     if i < bound and grid_v[i, j].x < 0:
53         grid_v[i, j].x = 0
54     if i > n_grid - bound and grid_v[i, j].x > 0:
55         grid_v[i, j].x = 0
56     if j < bound and grid_v[i, j].y < 0:
57         grid_v[i, j].y = 0
58     if j > n_grid - bound and grid_v[i, j].y > 0:
59         grid_v[i, j].y = 0
60
61 @ti.kernel
62 def substep_g2p(x : ti.any_arr(element_dim=1),
63               v : ti.any_arr(element_dim=1),
64               C : ti.any_arr(element_dim=2),
65               J : ti.any_arr(),
66               grid_v : ti.any_arr(element_dim=1)):
67     for p in x:
68         Xp = x[p] / dx
69         base = int(Xp - 0.5)
70         fx = Xp - base
71         w = [0.5 * (1.5 - fx)**2, 0.75 - (fx - 1)**2, 0.5 * (fx
72 - 0.5)**2]
73         new_v = ti.Vector.zero(float, 2)
74         new_C = ti.Matrix.zero(float, 2, 2)
75         for i, j in ti.static(ti.ndrange(3, 3)):
76             offset = ti.Vector([i, j])
77             dpos = (offset - fx) * dx
78             weight = w[i].x * w[j].y
79             g_v = grid_v[base + offset]
80             new_v += weight * g_v
81             new_C += 4 * weight * g_v.outer_product(dpos) / dx
82         **2
83         v[p] = new_v
84         x[p] += dt * v[p]
85         J[p] *= 1 + dt * new_C.trace()
86         C[p] = new_C
87
88 @ti.kernel
89 def generate_vbo(x : ti.any_arr(element_dim=1), vertices : ti.
90               any_arr(element_dim=1)):
91     for p in x:
92         vertices[p] = x[p]

```

```

91 @ti.kernel
92 def init_particles(x : ti.any_arr(element_dim=1),
93                  v : ti.any_arr(element_dim=1),
94                  J : ti.any_arr(),
95                  x_init : ti.any_arr(element_dim=1),
96                  v_init : ti.any_arr(element_dim=1)):
97     for i in range(n_particles):
98         x[i] = x_init[i]
99         v[i] = v_init[i]
100        J[i] = 1
101
102 g = cgraph.Module()
103
104 g_init = g.new_graph('init')
105 g_init.emplace(init_particles, Arg('x'), Arg('v'), Arg('J'),
106               Arg('x_init'), Arg('v_init'))
107
108 substep = cgraph.Sequential()
109 substep.emplace(substep_reset_grid, Arg('grid_v'), Arg('grid_m
110                '))
111 substep.emplace(substep_p2g, Arg('x'), Arg('v'), Arg('C'), Arg
112                ('J'), Arg('grid_v'), Arg('grid_m'))
113 substep.emplace(substep_update_grid_v, Arg('grid_v'), Arg('
114                grid_m'))
115 substep.emplace(substep_g2p, Arg('x'), Arg('v'), Arg('C'), Arg
116                ('J'), Arg('grid_v'))
117
118 # Static loop??
119 g_update = g.new_graph('update')
120 for i in range(50):
121     g_update.append(substep)
122
123 g_update.emplace(generate_vbo, Arg('x'), Arg('vertices'))
124
125 g.compile()
126 g.save('mpm.mod')
127
128 # Runtime. We will expose a similar C++ API.
129 import numpy as np
130
131 ti.init(arch=ti.vulkan)
132
133 vertices = (np.random.random((n_particles, 2)) * 0.4 + 0.2).
134             astype(np.single)
135 init_v = np.zeros((n_particles, 2)).astype(np.single)

```



```

130
131 x = ti.Vector.ndarray(2, ti.f32, shape=(n_particles))
132 v = ti.Vector.ndarray(2, ti.f32, shape=(n_particles))
133 C = ti.Matrix.ndarray(2, 2, ti.f32, shape=(n_particles))
134 J = ti.ndarray(ti.f32, shape=(n_particles))
135 grid_v = ti.Vector.ndarray(2, ti.f32, shape=(n_grid, n_grid))
136 grid_m = ti.ndarray(ti.f32, shape=(n_grid, n_grid))
137
138 g = cgraph.load_module('mpm.mod')
139 compiled_g_init = g.get_graph('init')
140 compiled_g_init.bind(('x', x), ('v', v), ('J', J))
141 compiled_g_init.run(('x_init', vertices), ('v_init', init_v))
142
143 compiled_g_update = g.get_graph('update')
144 compiled_g_update.bind(('x', x), ('v', v), ('C', C), ('J', J),
145                        ('grid_v', grid_v), ('grid_m', grid_m))
146
147 gui = ti.GUI('MPM')
148 while gui.running:
149     compiled_g_update.run(('vertices', vertices))
150     gui.clear(0x112F41)
151     gui.circles(vertices, radius=1.5, color=0x068587)
152     gui.show()

```