The Taichi High-Performance and Differentiable Programming Language for Sparse and Quantized Visual Computing

by

Yuanming Hu

B.Eng., Tsinghua University (2017)
S.M., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2021

Author . . . . . .
Department of Electrical Engineering and Computer Science
March 30, 2021

Certified by . . . . . .
Frédo Durand
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . . . .
William T. Freeman
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

The Taichi High-Performance and Differentiable Programming Language for Sparse and Quantized Visual Computing

by

Yuanming Hu

Submitted to the Department of Electrical Engineering and Computer Science
on March 30, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Using traditional programming languages such as C++ and CUDA, writing high-performance visual computing code is often laborious and requires deep expertise in performance engineering. This implies an undesirable trade-off between performance and productivity. Emerging visual computing workloads such as sparse data structure operations, differentiable programming, and quantized computation, lead to further development difficulties with existing programming systems. To address these issues, we propose *Taichi*, an imperative and parallel programming language, tailored for developing high-performance visual computing systems. Taichi leverages domain-specific features of visual computing tasks, providing first-class abstraction and support for spatially sparse computation, differentiable programming, and quantization. With Taichi's optimizing compiler that has a high-level understanding of these domain-specific language constructs and automatically optimizes Taichi programs, we achieve performance and productivity simultaneously in various visual computing tasks, especially physical simulation. For example, with Taichi we can easily achieve $4.55\times$ higher performance using $1/10$ lines of code on sparse computations, effortlessly develop 10 differentiable physical simulators, and simulate unprecedented 235 million material point method (MPM) particles on a single GPU.

Thesis Supervisor: Frédo Durand
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: William T. Freeman
Title: Professor of Electrical Engineering and Computer Science

Thesis Reader: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Thesis Reader: Jonathan Ragan-Kelley
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

First of all, I want to deeply thank my advisors, Frédo Durand and Bill Freeman, who have guided and supported me, with great enthusiasm and thoughtfulness, to work on research projects that I truly enjoy. Without their technical and mental support, the Taichi programming language would not have been made possible. There are countless moments during my graduate study that I feel inspired by my advisors. Here, I would only point out two important tips, which I got at the very beginning of my Ph.D. study from my advisors. The two tips reshaped the way I do research: Frédo highlights the importance of "writing, communication, and taste", which I believe I should keep improving even after graduate school; Bill showed me "the paper impact curve", which encourages me to always aim for high-quality papers with a large impact, instead of many mediocre papers.

My thesis committee members Saman Amarasinghe and Jonathan Ragan-Kelley provided useful comments to this dissertation. In addition to that, Saman also provided valuable feedback to the initial Taichi system during my research qualification exam, and Jonathan coauthored early papers on Taichi.

Apart from my thesis committee, many advisors helped me along the way. I was fortunate to work with Toshiya Hachisuka and Seiichi Koshizuka while visiting the University of Tokyo. Under their guidance, I sharpened my skills in realistic rendering and physical simulation. Later, I visited Stanford university and worked with Ron Fedkiw on reconstructing trees. My first research paper was done at Microsoft Research Asia when I was an undergrad research intern, mentored by Steve Lin. Steve was extremely patient and thoughtful. As a beginner in research, I learned so much from Steve. Our research turned out to be an oral paper at CVPR 2017, and a paper on ACM Transactions on Graphics, later presented at SIGGRAPH 2018. In 2016, Eftychios Sifakis explained to me the importance of memory hierarchy while we were having breakfast together in Wisconsin, Madison. At that time, I did not realize our conversation on high-performance computing, on that sunny graduate school visit day, would later become a central topic

of my Ph.D. research. Right before my graduate study at MIT, I visited Chenfanfu Jiang at the University of Pennsylvania. The visit was fruitful and resulted in my first SIGGRAPH paper. Fanfu taught me so much on how to deliver a high-quality SIGGRAPH paper, which laid a solid foundation for my graduate study. During my master's study at the computational fabrication group, I had fun working with Wojciech Matusik on computational design, 3D printing, robotics, and machine learning. Qi Sun was an extremely kind and helpful host during my summer internship at Adobe in 2019. Vinod Grover mentored my summer internship at NVIDIA in 2020 and taught me some deep knowledge of how GPU compilers work.

I would also like to thank my collaborators duing my Ph.D. study. I am fortunate to collaborate with Zhiao Huang, Tao Du, Jiafeng Liu, Mingkuan Xu, Xuanda Yang, Prof. Weiwei Xu, Dr. Qiang Dai, Prof. Hao Su, Prof. Joshua B. Tenenbaum, Prof. Daniela Rus, Dr. Chuang Gan, Luke Anderson, Dr. Tzu-Mao Li, Dr. Qi Sun, Dr. Nathan Carr, Dr. Xinxin Zhang, Dr. Ming Gao, who have been both great colleagues and highly supportive friends of mine.

Many researchers in related fields, with whom I did not get a chance to collaborate, did provide useful inputs to my research. Sylvain Paris provided insightful advice to many of my research projects. Charles Leiserson attended my research qualification exam, and provided helpful feedback to the Taichi system. Yunming Zhang, Andrew Adams, Shoaib Kamil shared with me helpful compiler insights. Comments from anonymous SIGGRAPH reviewers are always extremely detailed and insightful, and I learned so much from those generous and brilliant reviewers.

My research has been partly supported by an Edwin Webster fellowship, a Snap Research fellowship, an Adobe Research fellowship, a Facebook Research fellowship. Apart from fellowships that helped me better focus on research, Toyota Research Institute, the NSF/Intel Partnership on Computer Assisted Programming for Heterogeneous Architectures (CCF-1723445) have also supported my research projects. I appreciate the financial support from all these funding agencies.

Taichi is now contributed by over 60 open-source developers across the world.

Many of them, especially Ye Kuang, Mingkuan Xu, Jiafeng Liu, Xuanda Yang, have become close friends to me.

Last but not least, I would thank my parents, my girlfriend, and my best graduate-school buddies Jie Xu and Liang Shi. They have been unconditionally supporting me, especially during the COVID-19 pandemic in the US.

This dissertation is dedicated to everyone above, and many others who have helped me along this fruitful graduate study journey.

Contents

Chapter 1

Introduction

From image processing and realistic rendering to physical simulation and VR/AR, *visual computing* tasks are nowadays everywhere, bridging the physical and digital world. However, visual computing often demands extremely high performance to process massive visual data such as images, videos, particles, and volumes. Researchers and developers have devoted tremendous efforts to designing and optimizing the *hardware* (multicore CPUs, massively parallel GPUs, and DSPs), *software* (e.g., renderers and physical simulators), and *algorithms* (such as fast Fourier transform [24] and multigrid methods [118]) for visual computing problems. This dissertation is focused on the *software* development part.

1 Why a new programming language for visual computing?

Well-optimized visual computing software systems are often implemented in languages such as C++ and CUDA, which are close to computing hardware and offer possibility to achieve high efficiency. Unfortunately, simply *using (without further optimizations)* a close-to-hardware language is often not enough for performance. The need for high resolution and real-time performance of visual computing tasks usually implies tedious and challenging performance engineering, to

make a working system run more efficiently. A heavily performance-engineered visual computing program can sometimes run one order of magnitude faster than a reasonable C++/CUDA implementation. For example, a particle-to-grid splatting kernel of the moving least squares material point method ([48]), when properly engineered, can run 7× faster than a normal implementation using the same C++ programming language and computing hardware. The performance improvement comes from 4-wide vectorization using SSE and implementing a software-defined local scratchpad that caches frequent grid node data from the global sparse grid in the L1 data cache.

At a high level, these traditional programming languages tend to suffer from a **tension between productivity and performance**. C++ and CUDA themselves are not the easiest languages to learn. To enable a direct mapping to hardware and zero-cost abstractions [114], C++ offers advanced features such as template metaprogramming, yet the need to master these features further poses an even higher barrier for programmers. Moreover, to develop a high-performance visual computing system, developers need a deep understanding of how a C++ program is compiled and how processors work. In one word, a basic C++ program typically does not lead to good enough visual computing performance on its own, while performance engineering techniques, such as vectorization, loop unrolling, accelerating data structures, transforming data layouts, and data field compression, can easily lead to code that is hard to read, maintain, and debug.

At the same time, traditional high-performance languages usually **lack portability** and tend to tie implementation to specific accelerators (e.g., multi-thread CPUs and massively parallel GPUs). For example, C++ ties implementations to CPUs, while CUDA ties to NVIDIA GPUs. This is undesirable, since visual computing can happen everywhere, from workstations with high-end GPUs to mobile devices without programmable graphics hardware where software has to fall back to CPUs. We need a portable system that can deploy the same piece of code on various platforms.

Beyond the issues with productivity, performance, and portability that have ex-

isted for decades, there are **emerging types of workloads that further complicate programs written in these traditional languages**. For example,

- 3D visual data are often *spatially sparse*, and sparse data structures are key to achieving high performance in those cases. However, programming spatially sparse computation is hard with traditional languages, even with the help of a mature data structure library.
- The rise of deep learning motivated researchers to make visual computing components *differentiable*, while neither C++ nor CUDA natively supports differentiable programming.
- Saving memory bandwidth and space using *low-precision (quantized)* data types becomes an effective way to improve resolution and performance, as **a)** the gap between processor throughput and memory bandwidth enlarges, and **b)** general-purpose GPUs with hard limits on memory space become predominant. Programming visual computing systems with low-precision and quantized data types while achieving high performance needs special language and compiler support that does not exist in traditional languages.

In a nutshell, *substantial redesign* of languages and compilers is needed to alleviate the tension between productivity and performance, make visual computing systems more portable, and more importantly, serve emerging computation patterns in visual computing.

2 The Taichi programming language

To address the aforementioned issues, we propose the *Taichi* programming language. Taichi aims to be both productive and performant on traditional and emerging visual computing workloads, by providing domain-specific language abstractions and compiler optimizations. Its portability is achieved via multiple compilation backends including x64, ARM, CUDA, Metal, OpenGL compute shaders, and even Javascript.

Scope Although Taichi aims to cover visual computing tasks as widely as possible, some types of computation are **outside** the scope of Taichi: 1) tasks with domain-specific hardware support, and 2) coarse-granularity tasks, where function call overheads and data transfer time are negligible, with well-optimized libraries. For example,

- Classical rendering tasks, with rasterization and ray tracing hardware support. Real-time graphics APIs such as OpenGL, DirectX, and Vulkan are often good enough.
- Video encoding/decoding, often done by hardware video codec.
- Deep neural networks with standard layers such as convolution and batch normalization [55], as well-solved by deep learning frameworks such as TensorFlow [2].

While Taichi is not designed for these workloads, we do need to consider the communication between Taichi and other parts of the visual computing system. Well-designed zero-copy APIs are often good options. For example, it would be helpful to let OpenGL vertex shaders (for rendering) directly read from the buffers with data from compute shaders compiled by Taichi (for simulation). Another good example is Taichi's PyTorch interface allows (differentiable) Taichi programs to interact with PyTorch for a close integration of deep neural networks and numerical computation kernels.

Goals The Taichi project has two high-level goals:

1. Simplify the process of high-performance visual computing system development and deployment;
2. Explore novel language abstractions and compilation approaches for visual computing.

In the next section only, we will briefly cover our work towards the first goal, as an overview of the engineering aspects of the Taichi system. The majority of this dissertation will be focused on the second goal, namely the novel language abstractions and compilation approaches, which are the research contributions.

3 Pragmatic design decisions

Before diving into the research contributions, we summarize the software engineering aspect of the Taichi system. The goal of these pragmatic design decisions is to *make Taichi easier to use by end-users*. We devoted a huge amount of engineering efforts on the Python frontend and compatibility across platforms[1].

The compilation workflow is depicted in Figure. 1-1. Key design decisions of Taichi are discussed below.

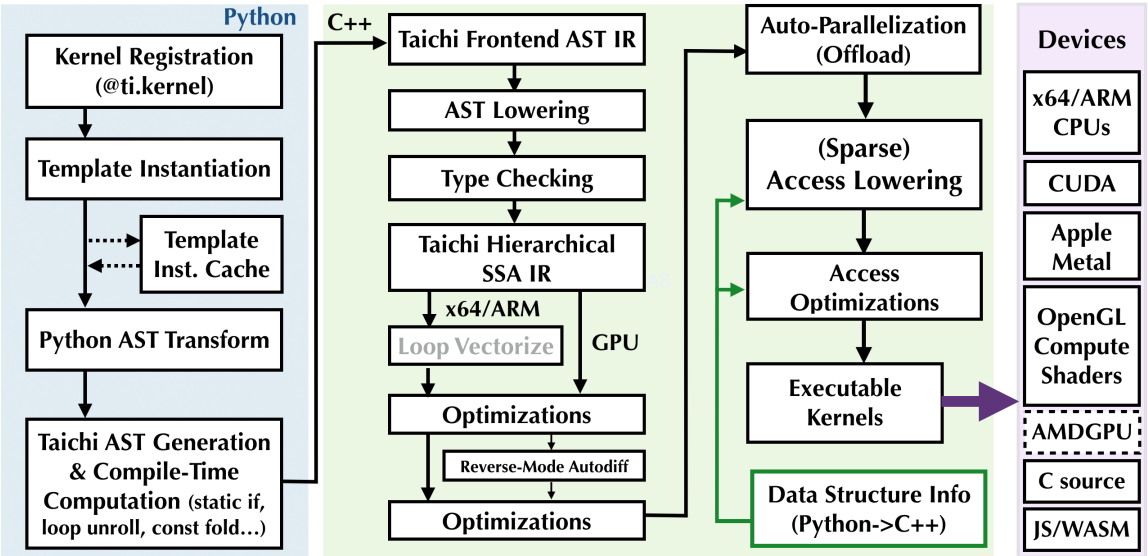


Figure 1-1: Life of a Taichi kernel. Python ASTs are progressively lowered into high-performance executable kernels that run on CPUs and GPUs. Domain-specific transformation and optimization passes ensure run-time performance.

[1] Alghough this dissertation will **not** cover these real-world engineering details beyond this section, readers are welcome to check out the Taichi project on GitHub for more details on how the system is engineered.

Imperative. Graphics programs, especially physical simulation, are usually computation hungry and need imperative programming, which is closer to hardware compared to functional programming, to extract the maximum possible performance out of parallel processors. Another motivation to stick to imperative programming, is to preserve maximum compatibility with existing graphics algorithms: easy control flow (`if`, `for`, `while`, `continue`, `break`) are supported in Taichi, and all data buffers ("fields") are mutable and can be in-place modified.

Programmable megakernels. Taichi uses a "megakernel" programming approach, allowing the programmer to naturally (and sometimes aggressively) fuse multiple stages of computation into a single kernel. Compared to composing the computational graph using element-wise linear algebra operators ("Op"s, as in TensorFlow [2] and PyTorch [96]), Taichi kernels have a higher arithmetic intensity (i.e., number of floating-point operations per byte fetched) and are therefore more efficient for visual computing tasks.

Embed in Python. Python is easy to learn and widely adopted, the frontend syntax of Taichi is a subset of Python. This allows every Python programmer to easily learn Taichi. Embedding Taichi in python also has the following advantages:

- Easy to run. Compared to embedding in a compiled language (such as C++), no ahead-of-time compilation of the container language is needed to run a Taichi kernel, since Python is interpreted.

- Easily reuse and interact with existing Python infrastructure, including IDEs, package manager (pip), and existing Python packages such as matplotlib and numpy.

We leverage the flexible AST inspection and manipulation features of Python to convert a Python function into a Taichi frontend AST, which is later compiled to high-performance kernels.

Compile just-in-time (JIT). JIT not only provides great programming flexibility, but also *delays* the need of "compile-time constant" values. For example, Δt in physical simulators is often a runtime variable in ahead-of-time compilation, but with JIT, Δt would be a compile-time constant. This allows the compiler to do more optimizations such as constant folding. Meanwhile, Taichi supports template metaprogramming with on-demand JIT compilation, saving the compiler from doing unnecessary compilation. We also provide ahead of time compilation, especially for mobile computing environments.

Data-orientated design. As visual computing applications are often limited by memory bandwidth, we adopt the *data-oriented* design philosophy (see, for example, [4, 73]) instead of traditional *object-oriented* design. This allows us to improve cacheline utilization and cache hit rate. Also note that Taichi *decouples* data layout from computation, as we show in chapter 2.

All these design decisions combined, Taichi is becoming a piece of computing infrastructure welcomed by many visual computing researchers and developers, who wish to write high-performance (GPU) code in the Python environment. As of March 2021, it has been downloaded $500,000$ times and has $12.9K$ stars on GitHub. Taichi is now developed by over 60 developers across the world. In production environments, Taichi powers the fluid simualtion engine in the Kuaishou app, a popular video sharing platform with over 500,000,000 daily active users (Fig. 1-2).

A simple Taichi program is shown in Figure. 1-3. For more details on the syntax of the Taichi programming language, please check out our Taichi course on SIGGRAPH 2020 [46].

Figure 1-2: A real-time GPU physics-based AR effect in the Kuaishou mobile app. A single piece of Taichi fluid simulation code compiles to Metal and OpenGL compute shader code simultaneously, powering both iOS and Android platforms. A user can freely rotate a smartphone to control the fluid motion via gravity.

4 Domain-specific language abstractions and tailored optimizing compilers

Exploring *novel language abstractions and compilation approaches* for visual computing is the focus of this dissertation. The *fundamental reason* why Taichi can achieve both high performance and high productivity, is because Taichi provides domain-specific language abstractions as first-class citizens, and the Taichi compiler can then conduct domain-specific optimizations *automatically*. Note that in general-purpose languages such as C++ and CUDA, programmers have to conduct these optimizations manually. This is because the domain-specific language constructs will be lowered into the general-purpose intermediate representation (IR) used by these compilers, which may not be suitable for the analysis and optimization of domain-specific operations that are often used in visual computing.

The power of domain-specific IR and compilers We use spatially sparse programming (chapter 2) as a motivating example to demonstrate the advantage of a domain-specific compiler over a general-purpose compiler. Similar logic partly

```python
import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default

n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```

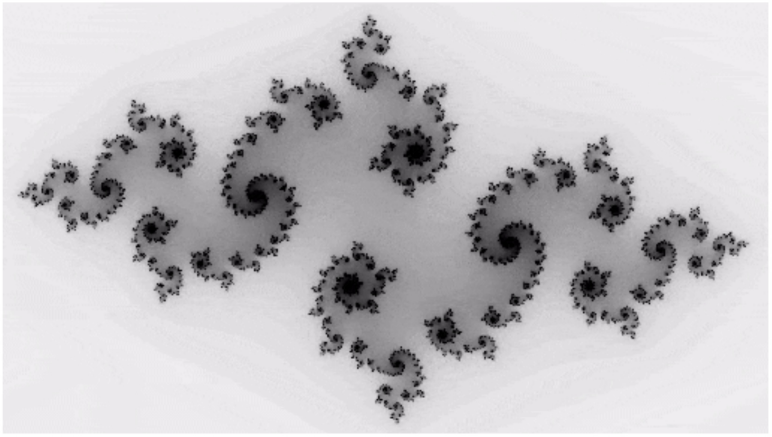


Figure 1-3: A simple Taichi program (left) generating a fractal animation (right). Taichi lives in the Python scripting language and is very easy to learn, especially for those who are familiar with Python. Our compiler and runtime system efficiently executes intense computation (such as the "paint" kernel) on parallel devices such as GPUs.

applies to the quantization system (chapter 5).

When computing on a multi-level sparse data structure such as VDB [89], data accesses to the voxels happen layer-by-layer, from the root node to the leaf node of the data structure tree. It is often the case that in a single loop iteration, neighboring voxels are accessed, for example, when applying a stencil operation to a sparse grid. A performance-aware programmer will usually try to merge common paths on the tree structure manually (Fig. 1-4).

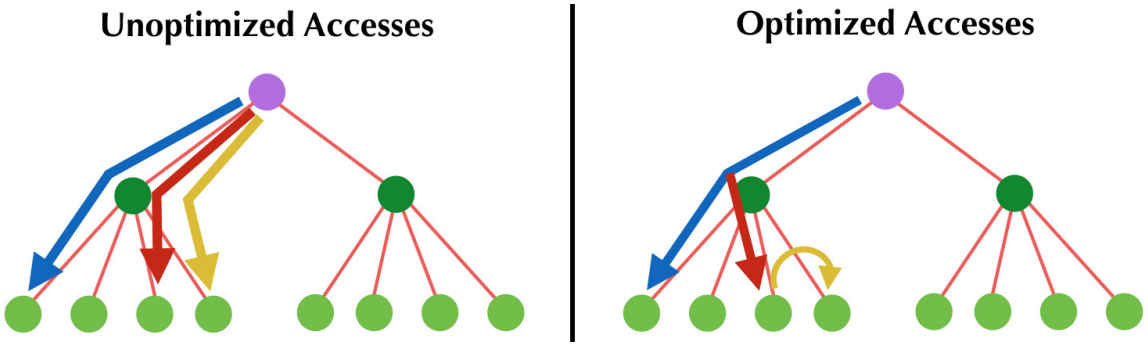


Figure 1-4: An experienced programmer may sacrifice code readability and decompose data structure accesses on the tree, to merge redundant memory addressing. Note that on sparse data structures memory addressing may involve looking up an expensive hash table, so optimizing it can lead to a significant performance gain.

There are many ways for a compiler to internally represent these data struc-

ture accesses (Fig. 1-5), but only some of them are suitable for such optimization. General-purpose compilers usually use LLVM [72] IR (or some other IR at a similar granularity), which is too fragmented for the compiler to have a concise understanding of the underlying semantics. In Taichi we have an IR stage that is tailored for these domain-specific access optimizations, allowing the compiler to do automatically what used to be done by experienced performance engineers (Fig. 1-6). Turning off these domain-specific optimizers and leaving all the optimization job to a general-purpose compilation backend (such as gcc, clang or LLVM), leads to programs that run $3.01\times$ slower on average (geometric mean, source: Table 4.1).

Figure 1-5: The same operation represented in different IRs. Finer granularity often means more instructions.

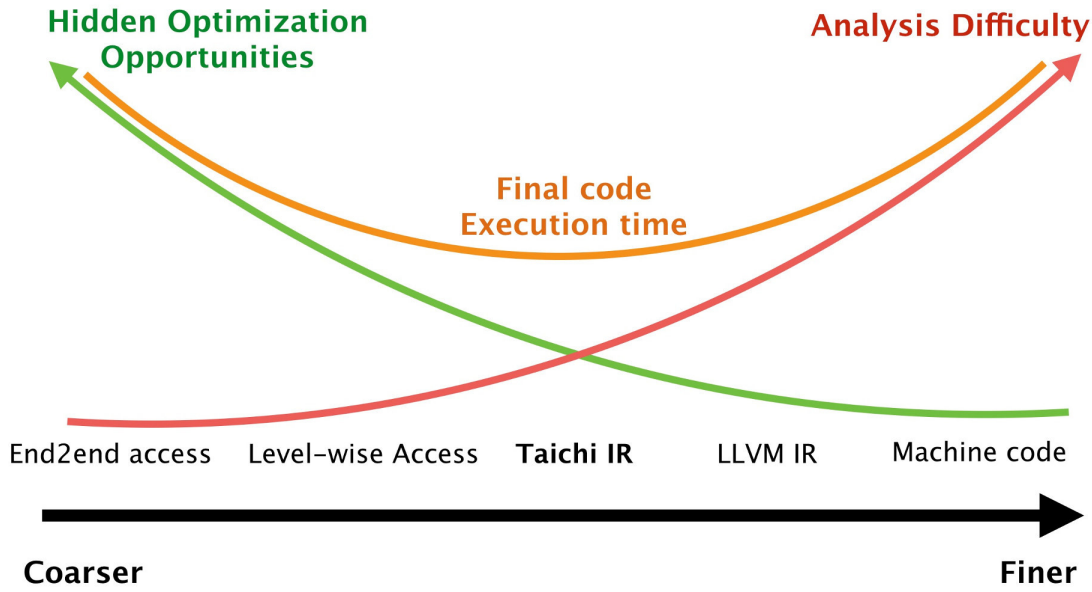


Figure 1-6: IR with too coarse granularity hides optimization opportunities, yet one that is too fragmented is hard to analyze. Only a granularity that lies at a sweet spot in this trade-off, opens up the most space for domain-specific optimizations.

5 Dissertation overview and research contributions

This dissertation is organized as follows, where each chapter presents a research contribution.

In chapter 2, we present the initial version of the Taichi programming language, focused on spatially sparse computation. Since 3D visual computing often involves complex sparse data structures, we propose a new data-oriented programming language based on Taichi, for efficiently authoring, accessing, and maintaining 3D sparse data structures. The language offers a high-level, data structure-agnostic interface for writing computation code. The user independently specifies the data structure. This *decoupling* of data structures from computation makes it easy to experiment with different data structures without changing computation code, and allows users to write computation as if they are working with a dense array. Our compiler conducts domain-specific optimizations on sparse computation code. With $\frac{1}{10}$th as many lines of code, we achieve $4.55\times$ higher performance on average, compared to hand-optimized reference implementations. *This work is published at SIGGRAPH Asia 2019 [49].*

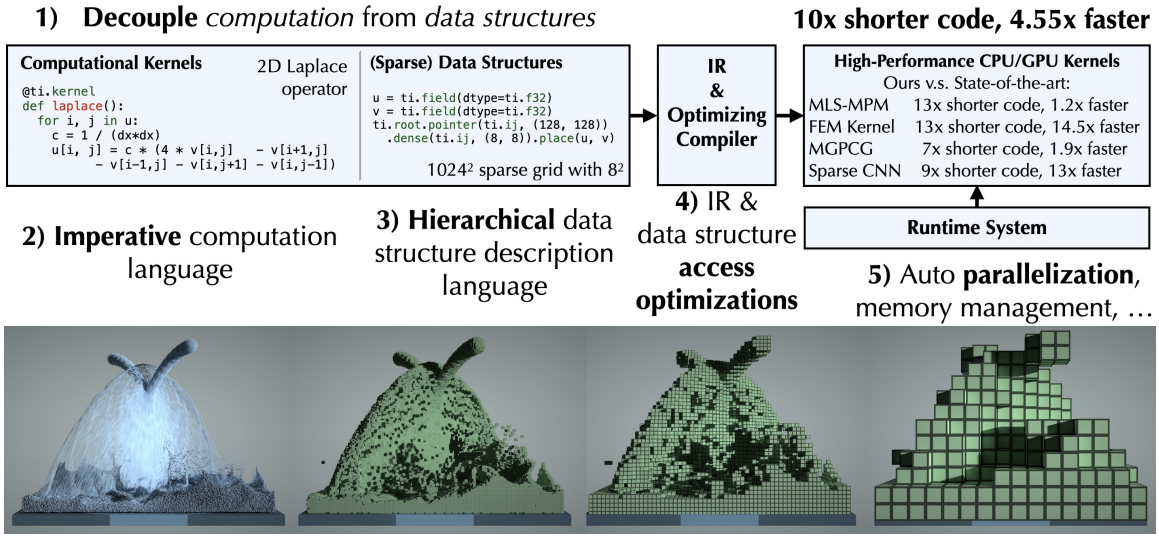


Figure 1-7: (Top) We expose a high-level interface for developing and processing spatially sparse multi-level data structures, and an optimizing compiler that automatically reduces data structure overhead. Programmers write code as if they are accessing dense voxels, while specifying the data arrangement *independently*. Our compiler automatically generates optimized, high-performance code tailored to the data structure. This results in concise code and better performance than highly-optimized reference implementations for various tasks. (Bottom) A fluid simulation using the material point method. We used a three-level sparse voxel grid with sizes $1^3, 4^3, 16^3$. Involved voxels are visualized in green. Both simulation and rendering are done using programs written in Taichi.

Chapter 3 details a novel two-scale differentiable programming system to the Taichi programming language (DiffTaichi). Inside Taichi kernels, DiffTaichi generates gradient kernels using source code transformations, which preserves arithmetic intensity (i.e., number of floating-point operations per byte fetched) and parallelism. Outside kernels, a light-weight tape is used to record the kernel launching information and replay the corresponding gradient kernels in a reversed order, for end-to-end backpropagation. A differentiable elastic object simulator written in DiffTaichi is $4.2\times$ shorter than the hand-engineered CUDA version yet runs as fast, and is $188\times$ faster than the TensorFlow implementation. Using our differentiable programs, neural network controllers are typically optimized within only tens of iterations (Fig. 1-8). *DiffTaichi is published at ICLR 2020 [47].*

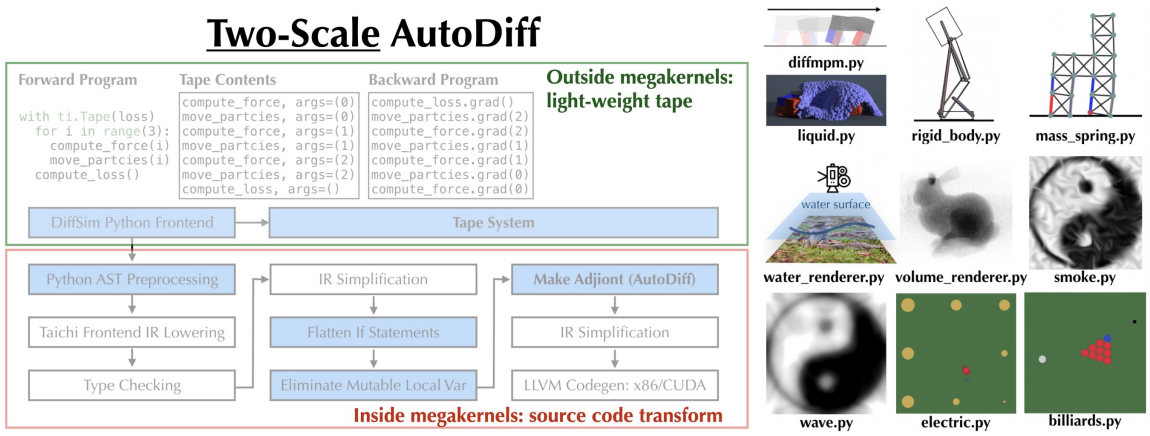


Figure 1-8: **Left:** The DiffTaichi system. **Right:** 10 differentiable simulators written in DiffTaichi.

To better optimize Taichi programs, we developed an inter-kernel optimization system, with an asynchronous execution engine (chapter 4). Inter-kernel optimization is especially relevant for tasks beyond traditional dense arrays computation, as exhibited by modern computer graphics and machine learning workloads, such as physical simulation with *spatial sparsity* and automatic gradient evaluation via *differentiable programming*. We show that these emerging computational patterns lead to new and exciting automatic optimization opportunities. *Without any computational code modification*, our new system leads to $4.02\times$ fewer kernel launches and $1.87\times$ speed up on our GPU benchmarks, including sparse-grid physical simulation and differentiable programming (Fig. 1-9).

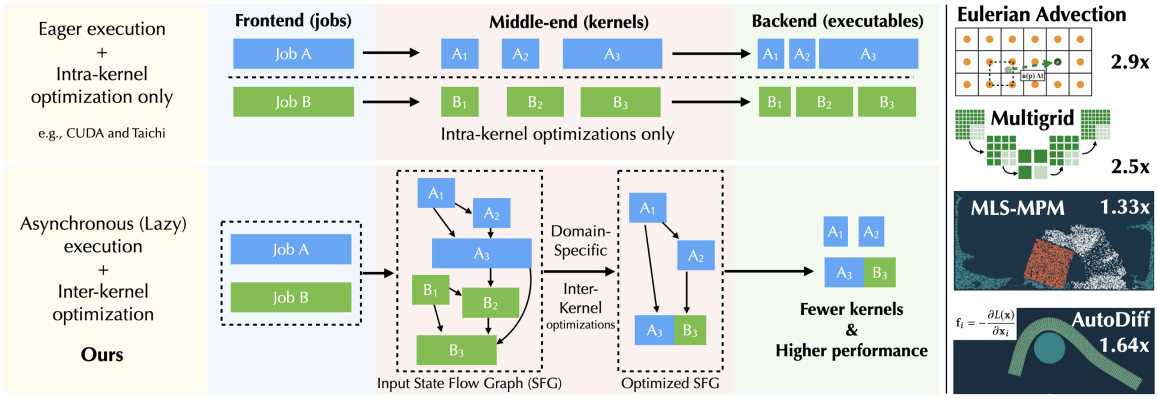


Figure 1-9: **Top left:** In existing parallel imperative programming systems (such as CUDA and the synchronous version of Taichi [49]), imperative computational kernels are eagerly launched, leaving a tiny room for the optimizer to optimize *beyond a single kernel*. **Bottom left:** In our new system (chapter 4), we accumulate kernels in an execution buffer, only flushing the execution queue when necessary. This allows the optimizer to gain more context and conduct optimization beyond a single kernel. We dynamically build a dependency graph ("state-flow graph") of kernels for easy analysis, so that computation kernels can be optimized at an inter-kernel level just in time. **Right:** After a suite of domain-specific optimization passes including list generation removal, sparse data structure activation elimination, and kernel fusion, kernels are much better optimized. As a result, the inter-kernel optimized programs run $1.87\times$ faster on GPUs, *without the user modifying any of the computation code*.

Finally, the quantization system of Taichi is introduced in chapter 5. We present a set of language abstractions and compiler optimizations that can achieve both high performance and significantly reduced memory costs, by enabling flexible and aggressive *quantization*. Low-precision ("quantized") numerical data types are used and packed to represent simulation states, leading to reduced memory space and bandwidth consumption. Our programming language and compiler, based on *Taichi*, allow developers to effortlessly switch between different full-precision and quantized simulators, to explore the full design space of quantization schemes, and ultimately to achieve a good balance between space and precision. For example, on a *single* GPU, we can simulate a Game of Life with 20 billion cells, an Eulerian fluid system with 421 million active voxels, and a hybrid Eulerian-Lagrangian elastic object simulation with 235 million particles (Fig. 1-10). *This work is published at SIGGRAPH 2021 [50].*

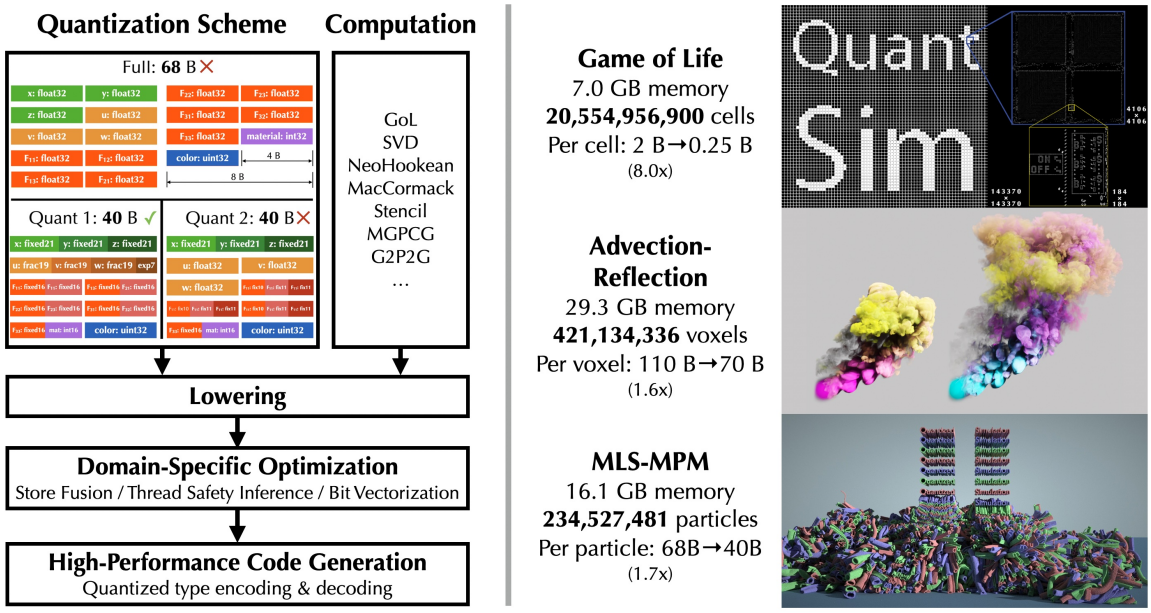


Figure 1-10: The Taichi quantization system (chapter 5) enables programmers to easily use low-precision and quantized data types for high-resolution simulations with reduced memory consumption.

Chapter 2

Programming with Spatial Sparsity

3D visual computing data are often spatially sparse. To exploit such sparsity, people have developed hierarchical sparse data structures, such as multi-level sparse voxel grids, particles, and 3D hash tables. However, developing and using these high-performance sparse data structures is challenging, due to their intrinsic complexity and overhead. We propose *Taichi*, a new data-oriented programming language for efficiently authoring, accessing, and maintaining such data structures. The language offers a high-level, data structure-agnostic interface for writing computation code. The user independently specifies the data structure. We provide several elementary components with different sparsity properties that can be arbitrarily composed to create a wide range of multi-level sparse data structures. This *decoupling* of data structures from computation makes it easy to experiment with different data structures without changing computation code, and allows users to write computation as if they are working with a dense array. Our compiler then uses the semantics of the data structure and index analysis to automatically optimize for locality, remove redundant operations for coherent accesses, maintain sparsity and memory allocations, and generate efficient parallel and vectorized instructions for CPUs and GPUs.

Our approach yields competitive performance on common computational kernels such as stencil applications, neighbor lookups, and particle scattering. We demonstrate our language by implementing simulation, rendering, and vision tasks

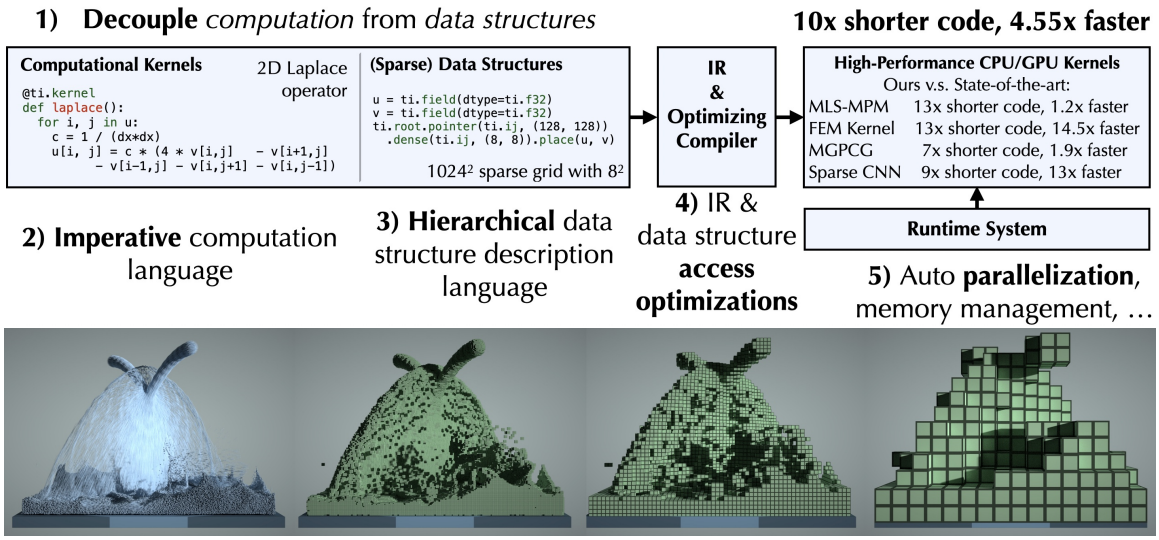


Figure 2-1: (Top) We propose the Taichi programming language, which exposes a high-level interface for developing and processing spatially sparse multi-level data structures, and an optimizing compiler that automatically reduces data structure overhead. Programmers write code as if they are accessing dense voxels, while specifying the data arrangement *independently*. Our compiler automatically generates optimized, high-performance code tailored to the data structure. This results in concise code and better performance than highly-optimized reference implementations for various tasks. (Bottom) A fluid simulation using the material point method, where two liquid jets collide with each other, forming a thin sheet structure. We used a three-level sparse voxel grid with sizes $1^3, 4^3, 16^3$. Involved voxels are visualized in green. Both simulation and rendering are done using programs written in Taichi.

including a material point method simulation, finite element analysis, a multi-grid Poisson solver for pressure projection, volumetric path tracing, and 3D convolution on sparse grids. Our computation-data structure decoupling allows us to quickly experiment with different data arrangements, and to develop high-performance data structures tailored for specific computational tasks. With $\frac{1}{10}$th as many lines of code, we achieve $4.55\times$ higher performance on average, compared to hand-optimized reference implementations.

1 Introduction

Large-scale 3D simulation, rendering, and vision tasks often involve volumetric data that are spatially sparse. Hierarchical and sparse data structures have been studied extensively to effectively exploit such sparsity. For example, in fluid simulation (Fig. 5-1), a multi-level grid is often used to represent the fluid field, where the fluid's spatial sparsity can be represented by nesting hash tables, bitmasks, or pointer arrays at different levels of the grid.

Writing high-performance code for these data structures is a daunting task due to their irregularity. Accessing their active elements in parallel imposes several engineering challenges (Fig. 2-2). First, naively traversing the hierarchy can take one or two orders of magnitude more clock cycles than the essential computation. This is especially troublesome for spatially coherent accesses commonly seen in, for example, stencil operations, since common access paths in the hierarchical data

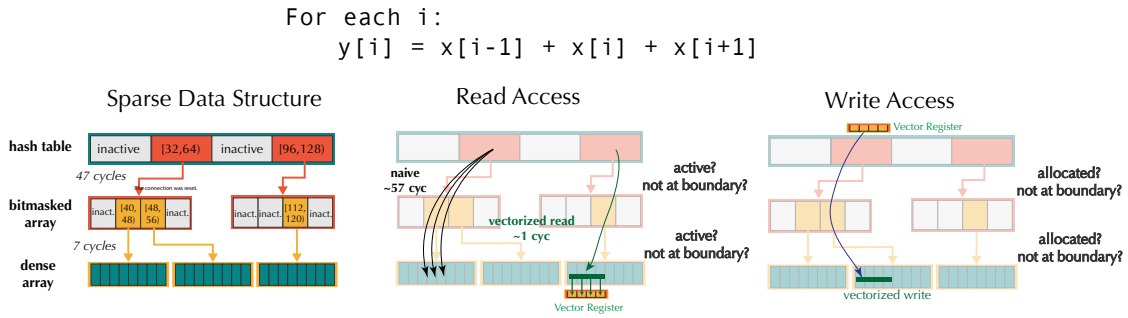


Figure 2-2: Accessing a multi-level sparse array is significantly more involved than accessing a dense array. The figure illustrates an example three-level sparse array and the read and write access of a stencil `y[i]=x[i-1]+x[i]+x[i+1]`. Naive code is usually inefficient, since the hierarchy makes traversal costly, and it is especially problematic for spatially coherent accesses, where the top of the traversal is often redundant. Optimized and vectorized code needs to leverage access locality to amortize the access cost, check for sparsity, handle boundary cases, and allocate memory when necessary. Writing code for these accesses is tedious and error-prone, and it often leads to code that is highly-coupled with the data structure. Our language decouples the data structure implementation and the access, while our compiler automatically generates optimized code given the access pattern and the specific data structure. As a result, users write code as if they are accessing dense arrays, while having the freedom to change the data layout and sparsity representation without affecting the computation code.

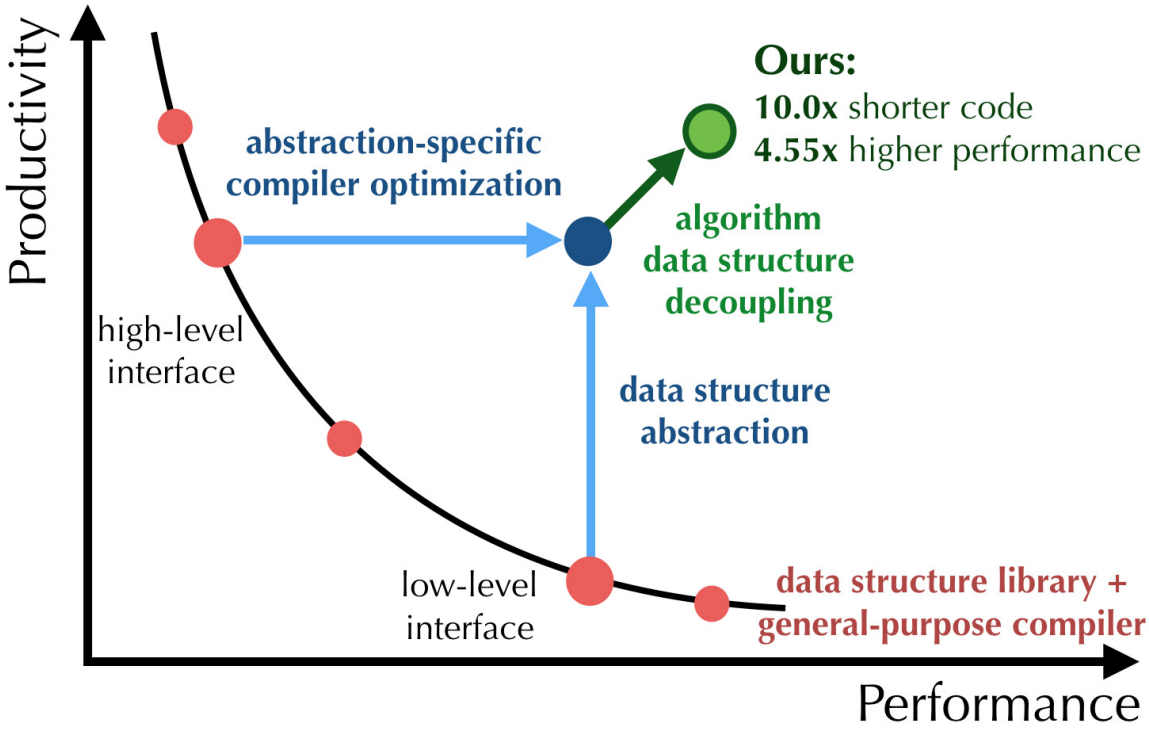


Figure 2-3: Traditionally, a user who writes data structure access code faces a dilemma between easy programming and high performance. The goal of our language is to achieve both the productivity of a high-level library and the high performance of manually optimized code. Furthermore, since our language makes it easy to experiment with different data structures, users can often achieve even higher performance by exploring the data structure design space and adopting the most efficient design for a given task.

structure are traversed redundantly. Second, we need to ensure load-balancing for efficient parallelization. Third, we need to allocate memory and maintain sparsity when accessing inactive elements.

Data structure libraries do not guarantee high-performance code, since performance is not easily composable. Multiple calls to the library interface will result in redundant and costly traversals of the hierarchy. Unfortunately, because of the code complexity of these data structures, and potential race conditions and pointer aliasing, current general-purpose compilers often fail to optimize between library function calls. To achieve high performance, libraries usually have to expose low-level interfaces to users, leading to a leaky abstraction, making computation code highly coupled with data structures. We propose a new programming model that

decouples data structures from computation, to achieve both high performance and easy programming (Fig. 5-2). Users write computation code using a high-level and data-structure-agnostic interface, as if they are operating on a dense multi-dimensional array. The internal data arrangement and the associated sparsity are specified independently from the computation code by composing elementary components such as dense arrays and hash tables to form a hierarchy.

Our compiler tailors optimizations for the specified data structure components, and generates efficient sparsity and memory maintenance code. We develop several domain-specific strategies for optimizing spatially-coherent accesses, using index analysis derived from high-level information about the data layout and the access patterns. Our compiler analyzes accesses to efficiently compute memory addresses, uses a caching strategy for better locality, and parallelizes/vectorizes loops from high-level instructions from the programmer. This is enabled by our compact intermediate representation, specially designed for optimizing hierarchical sparse data structures. Our compiler generates C++ code or CUDA code from the intermediate representation, making switching backends effortless.

On many common computations such as stencils, neighbor lookups, and particle splatting, our compiler generates code faster than highly-optimized reference implementations. We implement several popular simulation, graphics and vision algorithms in our language's embedded C++ frontend, including the material point method [112, 35, 48], finite element kernel [79], multigrid Poisson solver [86], sparse 3D convolution [37], and volumetric path tracing. Compared with highly-optimized reference implementations, our code requires on average only $\frac{1}{10}$th the number of lines of code, while being $4.55\times$ faster (geometric mean).

We can quickly explore different choices of data structures, while our compiler generates high-performance code. For example, we derived more efficient data structure designs for the material point method that not only lead to performance improvements of up to $1.2\times$ over previous, highly-optimized, state-of-the-art implementation [35], but also simplify the whole algorithm (Sec 6 .1).

Our model can express a wide variety of data structures used in physical sim-

ulation and rendering. In particular, it can describe different multi-level sparse grids (e.g. SPGrid [110], OpenVDB [89], and other novel data structures), particles, and dense and sparse matrices. We assume the hierarchy is known at compile-time to facilitate compiler optimization, therefore we do not directly model structures with variable depth such as k-d trees.

Our language and compiler are open-source[1]. All performance numbers from our system in this paper can be reproduced with the provided commands. All visual results are simulated and rendered using programs written in our language.

We summarize our contributions as follows:

- A programming language that decouples data structures from computation (Sec. 3 .1). We provide a unified abstraction to map multi-dimensional indices to memory addresses. Such an abstraction allows programmers to define computation independently of the internal arrangements of the involved data structures.

- A data structure description mini-language, which provides several elementary data structure components that can be composed to form a wide range of sparse arrays with static hierarchies (Sec. 3 .2).

- An optimizing compiler that uses index analysis and information from the data structures to automatically optimize for locality, minimize redundant operations for coherent accesses, manage sparsity, and to generate parallelized and vectorized backend code for x86_64 and CUDA (Sec. 5 and Sec. 6).

- A thorough evaluation of our system, and state-of-the-art implementations of several graphics and vision algorithms as by-products.

[1] https://github.com/taichi-dev/taichi

2 Goals and Design Decisions

Most sparsity patterns in 3D computing tasks exhibit spatial coherency (Figure 2-4). The sparsity may come from fluid simulation, clouds in volume rendering, or point clouds of surfaces from LiDAR and Kinect scans. To obtain high performance, we want to model the spatial sparsity effectively so we can utilize the spatial coherency while not wasting computational resources on empty space.

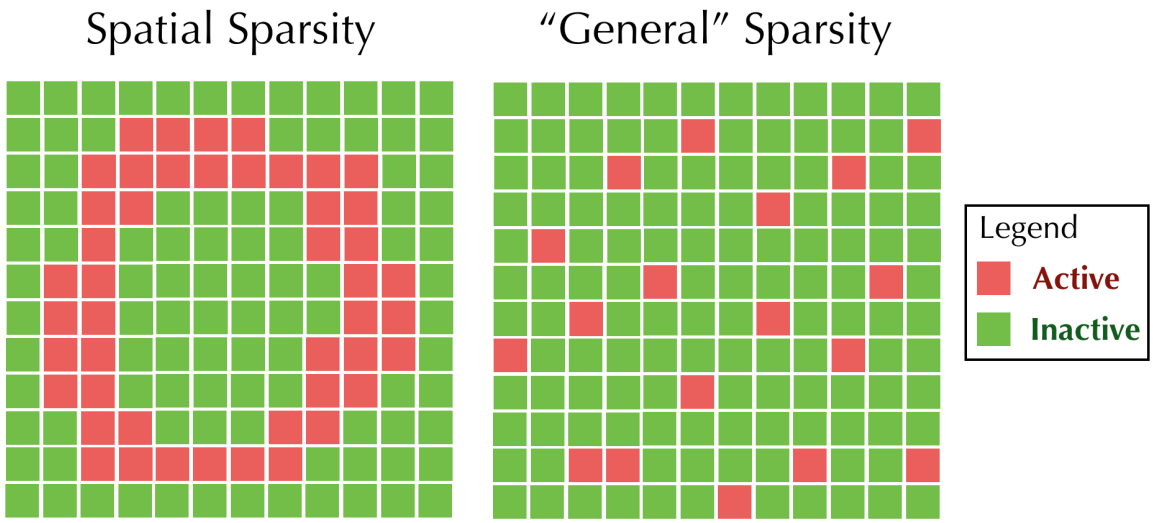


Figure 2-4: **Left:** We focus on *spatial sparsity*, where the data is globally sparse yet locally dense; **Right:** *General sparse problems with random patterns* are less suited to our language.

We aim to develop a high-performance programming language to exploit spatial sparsity using dedicated data structures. The four high-level goals are as follows:

Expressiveness Our target applications often feature complex computational kernels, such as stencils of different sizes, particle splatting, and ray-voxel intersection. Therefore, the language should be expressive enough to cover these numerical computation patterns. Taichi allows users to read/write to arbitrary elements in the sparse data structures, and provides constructs for branching and looping. This distinguishes our languages from more domain-specific ones, such as taco [67] (linear algebra).

Performance On modern computer architectures, achieving high performance means good exploitation of locality and parallelism. However, the desired memory-friendly and parallel data structure is usually task- and hardware-dependent, so existing data structure libraries that only provide a single data structure design do not completely solve the performance issue.

Productivity Traditionally, programming on sparse data structures requires manually handling memory allocation, parallelization, exceptions and boundary conditions. Even with libraries, low-level programming is still necessary to achieve high performance. Our language allows programming on sparse data structures *as if they are dense*, while the compiler automatically generates optimized code. To our knowledge, Taichi is the first system that makes it possible to write large-scale physical simulations on complex data structures within only a few hundred lines of code.

Portability The language should automatically generate optimized code for different hardware environments. We do **not** offer programmer access to low-level control over hardware when it would sacrifice performance portability, like the prefetch intrinsics on x86 CPUs or warp-level intrinsics on NVIDIA GPUs.

2 .1 Design Decisions

Our design decisions are made based on the aforementioned goals and non-goals.

- **Decouple data structures from computation.** The user should write high-level code for computation as if they are processing a dense array, while also being able to explore different sparse data structures without affecting the computation code. We achieve this by abstracting data structure access with Cartesian indexing, while the actual data structures define the mapping from the index to the actual memory address (Sec. 3 .1).

- **Regular grids as building blocks** The basic data structure entities of our system are regular grids, which can be easily flattened into 1D arrays that map closely to modern computer architecture with linear memory addressing. We do not directly model more irregular structures such as meshes or graphs[2]. Multiresolution representations such as adaptive grids [82] need to be composed manually in our language (see Section 6 .3, or Setaluri et al.'s multigrid preconditioner [110]).

- **Describe data structures through hierarchical composition.** To model spatial sparsity, and to express a wide variety of data structures, we develop a data structure mini-language to compose data structure hierarchies (Sec. 3 .2). The mini-language is made up of several elementary components, such as dense arrays and hash tables, that are arbitrarily composable.

- **Fixed data structure hierarchy.** We facilitate compiler optimizations and simplify memory allocation by assuming the hierarchy to be fixed at compile time. We do not support octrees or bounding volume hierarchies with dynamic depth. Many state-of-the-art physical simulation systems use data structures with a fixed hierarchy such as SPGrid [110] and VDB [89, 41].

- **Single-Program-Multiple-Data (SPMD) with sparse iterators.** We adopt an imperative SPMD model to harness the power of modern hardware such as vectorized instructions on CPUs and massively parallel GPUs. To exploit sparsity, we design computation kernels to be parallel for loops with sparse iterators on active elements only. This provides programmers a simple yet expressive interface to sparse computation.

- **Generate optimized backend code automatically.** Our compiler should generate high-performance backend code automatically, while optimizing for locality (Sec. 4 .1), minimizing redundant accesses using access coherency (Sec. 4 .2), automatically parallelizing (Sec. 4 .3) and allocating memory (Sec. 5

[2]It is possible to use 1D arrays for storing vertices and edges in meshes/graphs.

.2). The user should only need to provide the backend target architecture and optionally some scheduling hints for the compiler to generate better optimized code.

3 The Taichi Programming Language

We demonstrate our language using a 2D Laplace operator $u = \nabla^2 v$, which is frequently used in physical simulation and image processing. After finite difference discretization, the operation is defined as:

$$u_{i,j} = \frac{1}{\Delta x^2}(4v_{i,j} - v_{i+1,j} - v_{i-1,j} - v_{i,j+1} - v_{i,j-1}).$$

3 .1 Defining Computation

To decouple data structures from computation, we abstract data structures as *mappings from multi-dimensional indices to the actual value*. For example, access to the 2D scalar field u is always done through indexing, i.e. `u[i, j]`, no matter what the internal data structure is. This is similar to high-level interfaces of some data structure libraries, yet our compiler analyzes these accesses and produces code that minimizes redundancy across multiple accesses.

Our language's frontend is embedded in C++. Computations in our language are usually defined as kernels looping over *active* data structure elements (e.g. non-zero pixels or voxels), to efficiently exploit data sparsity. The kernel contains imperative code that operates on the data structures.

We define the aforementioned Laplace operator as a kernel, using a for loop over variable `u`, which iterates over all pairs `(i, j)` where `u[i, j]` is an *active* element:

```
Kernel(laplace).def([&]() {
  For(u, [&](Expr i, Expr j){
    auto c = 1.0f / (dx * dx);
```

```
    u[i, j] = c * (4 * v[i, j] - v[i+1, j]
              - v[i-1, j] - v[i, j+1] - v[i, j-1]);
  });
});
```

For loops over active elements are key to sparse computation in Taichi. The compiler automatically maintains sparsity. When reading from an inactive element of `v`, the compiler returns an *ambient value* (e.g., 0). When writing to an inactive element of `u`, the compiler automatically changes the internal data structure, allocates memory, and marks the element as active (In this specific kernel, no activation will occur, since we are only writing to active elements of `u`, and interaction with `v` is read-only).

We adopt the Single-Program-Multiple-Data paradigm. Our language is similar to other SPMD languages such as ispc and CUDA, with three additional components: 1) parallel sparse `For` loops, 2) multi-dimensional sparse array accessors, and 3) compiler hints for optimizing program scheduling.

The `For` loop is automatically parallelized and vectorized. Our language supports typical control flow statements, such as `If-Then-Else` and `While` loops. We allow users to define mutable local variables (`Var`). Our language can be used to write a full volumetric path tracer with complex control flow (Sec. 6 .5). The language constructs supported inside computation kernels are listed below.

```
// Parallel loop over the sparse tensor "var"
For(Expr var, std::function)
// Loop over [begin, end)
For(Expr begin, Expr end, std::function)
// Access one element in "var" with index (i, ...)
operator[](Expr var, Expr i, ...)

While(Expr cond, std::function)
If(Expr cond)
If::Then(std::function)
If::Else(std::function)
Var(Expr) // Declare a mutable local variable
```

```
Atomic(A) += B // Atomic add to global element A
```

Our language also offers compiler hints for scheduling:

```
// For CPU
Parallelize(int num_threads) // Multi-threading
Vectorize(int width)         // Loop vectorization
// For GPU
BlockDim(int blockDim) // Specify GPU block size
// For scratchpad optimization
AssumeInRange(Expr base, int lower, int upper)
Cache(Expr)
// Cache data into GPU L1 cache
CacheL1(Expr)
```

More discussions on hints for scratchpad optimization (`AssumeInRange` and `Cache`) and `CacheL1` are in Section 4 .1.

3 .2 Describing Internal Structures Hierarchically

After writing the computation code, the user needs to specify the internal data structure hierarchy. Specifying a data structure includes choices at both the macro level, dictating how the data structure components nest with each other and the way they represent sparsity, and the micro level, dictating how data are grouped together (e.g. structure of arrays vs. array of structures).

Structural nodes and their decorators Our language provides *structural nodes* to compose the hierarchy, and *decorators* to provide structural nodes with particular properties. These constructs and their semantics are listed below:

`dense`: A fixed-length contiguous array. `hash`: Use a hash table to maintain the mapping from active coordinates to data addresses in memory. Suitable for high sparsity. `dynamic`: Variable-length array, with a predefined maximum length. It serves the role of `std::vector`, and can be used to maintain objects (e.g. particles) contained in a block.

(a) structural nodes

`morton`: Reorder the data in memory using a Z-order curve (Morton coding), for potentially higher spatial locality. For `dense` only. `bitmasked`: Use a mask to maintain sparsity information, one bit per child. For `dense` only. `pointer`: Store pointers instead of the whole structure to save memory and maintain sparsity. For `dense` and `dynamic`.

(b) node decorators

These data structure components provide trade-offs regarding access cost and space consumption. For example, a hash table has relatively long access time (e.g. 50 CPU cycles), but it is very economical in terms of memory space, especially in extremely sparse cases (e.g. 0.1%). Therefore it is often suitable for the top layer, when only a few hundred children are active out of, say, $128 \times 128 \times 128$. On the other hand, a dense array with a bitmask can be activated and accessed quickly, but the bitmask will occupy space inefficiently in highly sparse cases.

Defining the hierarchy Users can compose the data structure components arbitrarily to form desired hierarchies and to explore different trade-offs. The compiler will then synthesize how computational kernels are executed on the specific sparse data structure (Fig. 2-5).

For example, the following code specifies two fixed-size 2D dense arrays over `u` and `v`.

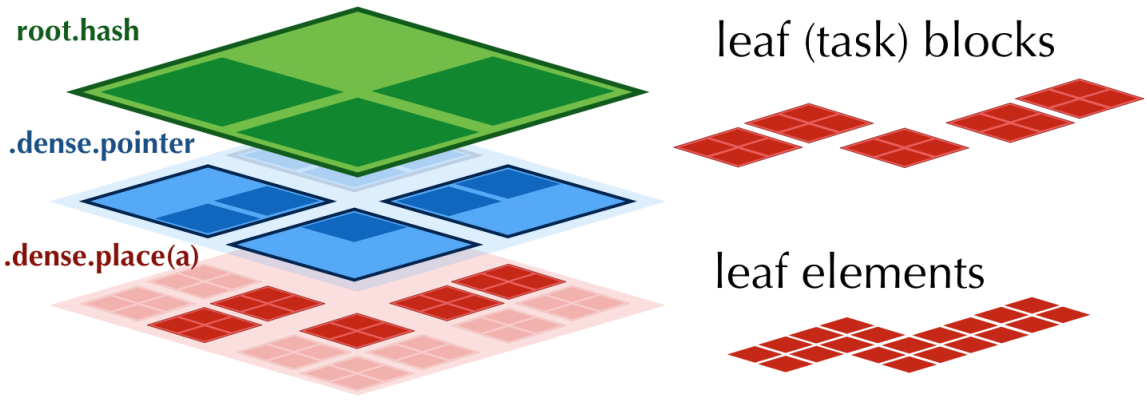


Figure 2-5: In our language, programmers define data structures by nesting elementary components such as hash tables and dense arrays. Kernels are defined as iterations over **leaf elements** (i.e., voxels or pixels), independent of the internal data organization. **Leaf blocks**, immediate blocks of leaf elements, are the smallest quantum of storage and computation tasks.

```
Global(u, f32); Global(v, f32);
layout([&]() {
  auto ij = Indices(0, 1);
  // Allocate a structure-of-arrays dense grid.
  // Equivalent to:
  // float u[256][256]; float v[256][256];
  root.dense(ij, {256, 256}).place(u);
  root.dense(ij, {256, 256}).place(v);
});
```

`Global(u, type)` declares an N-dimensional (sparse) tensor of name `u` and type `type`. These tensors are accessible by all kernels, so we call them *global variables*.

`layout` takes a C++ lambda function that describes the data structure hierarchy. `Indices` are used to specify sizes of structural nodes.

`root` denotes the root of the hierarchy. `dense`, a *structural node* of the tree, creates a child node of the root. Calling `dense` on `root` twice creates two children. Each structural node function call has two arguments, the first specifies the dimensions of its children, the second specifies the number of elements in the corresponding dimension. Here, `dense(ij, {256, 256})` means the 2D dense array has 256 cells

along index `i` (x-axis) and 256 cells along `j` (y-axis).

`place(u)` and `place(v)` assign the global variables `u, v` to the corresponding data structure hierarchies. The equivalent C-style data structure definition is provided in the comments.

The code above specifies a structure-of-arrays (SOA) layout. We can easily switch to an array-of-structures (AOS) layout using the following code:

```
// struct node {float u, v;};
// node data[256][256];
auto &node = root.dense(ij, {256, 256});
node.place(u); node.place(v);
// or equivalently
root.dense(ij, {256, 256}).place(u, v);
```

In this case, a single `dense` node contains both `u` and `v`, since we called `place` twice on the same `dense` node. As syntactic sugar, `place` can also take more than one parameter. When materialized in memory, in this AOS layout $u_{i,j}$ and $v_{i,j}$ are next to each other, while in the previous SOA layout $u_{i,j}$ is next to $u_{i,j+1}$ and is far away from $v_{i,j}$. These two layouts have very different memory behaviors (e.g. cacheline utilization) in different applications.

We can *nest* the structural nodes to specify the hierarchical tree structure in a top-down order. For example, the following code defines a three-level sparse grid, with the top-level being a hash table, the second-level being a dense array of pointers, and the third-level being a fixed-size dense array (Fig. 2-5):

```
root.hash(ij, {4, 4})
    .dense(ij, {4, 4}).pointer()
    .dense(ij, {16, 16}).place(u, v);
```

Apart from multiple global variables, structural nodes can also have multiple structural nodes as children. For example, the following code defines a bitmasked sparse array, where each of its elements is composed of a dense array and a dynamic array (similar to `std::vector`):

```
Global(u, f32); Global(v, f32); Global(p, f32);
```

```
root.hash(ijk, 32).dense(ijk, 16).pointer()
    .dense(ijk, 8).place(u, v, w);
```

(a) 3D VDB-style [89] structure with configuration [5, 4, 3]. The root-level hash table allows negative coordinates to be accessed, providing the user with an unbounded domain.

```
root.dense(ijk, 512).morton().bitmasked()
    .dense(ijk, {8, 4, 4}).place(flags, u, v, w);
```

(b) 3D SPGrids [110] occupying voxels in the bounding box $[0, 4096) \times [0, 2048) \times [0, 2048)$. The data structure is relatively shallow (only two levels), so root-to-leaf accesses have relatively low cost.

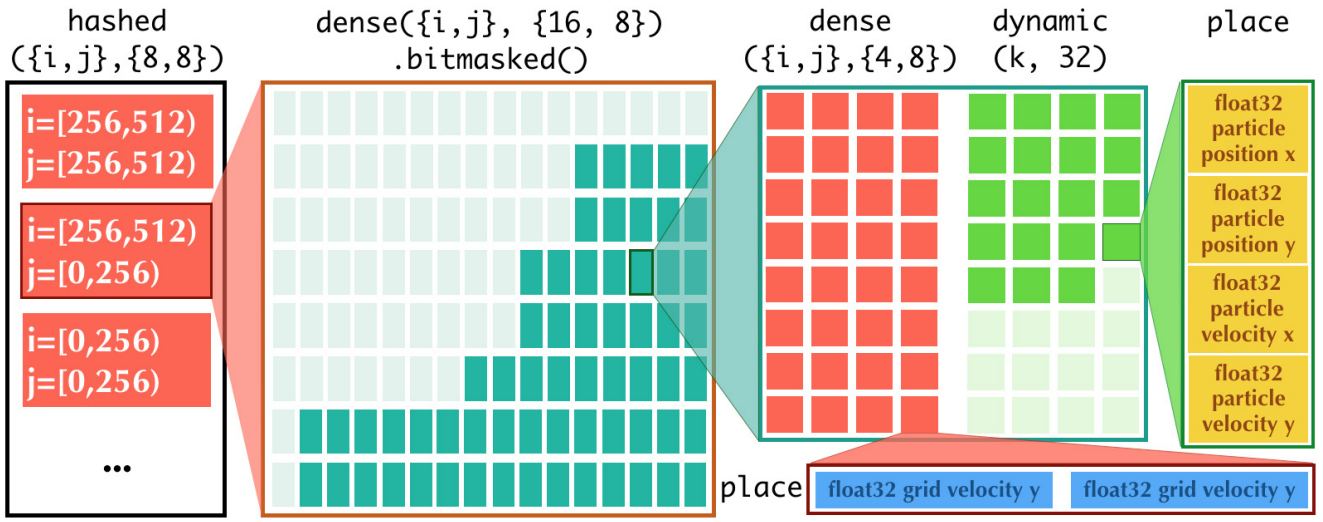
```
// "Hierarchical Particle Buckets": each leaf block contains all indices of particles within its range
root.dynamic(l, 2048).place(particle_x, particle_y, particle_z, particle_mass);
root.hash(ijk, 512).dense(ijk, 32).pointer().dense(ijk, 8).pointer().dynamic(l, 2048).place(particle_index
    );

// "SPVDB": Unbounded shallow data structures with bitmasks and Morton coding. (VDB and SPGrid combined.)
root.hash(ijk, 512).dense(ijk, 512).morton().bitmasked().dense(ijk, {8, 4, 4}).place(flags, u, v, w);
```

```
// "Hybrid Eulerian-
    Lagrangian Grid"
auto &block = root
 .hash(ij, 8)
 .dense(ij, {16, 8})
 .bitmasked();

// Child 1: grid nodes
block.dense(ij, {4,
    8})
      .place(grid_vx)
      .place(grid_vy);

// Child 2: particles
block.dynamic(i, 32)
      .place(part_x)
      .place(part_y)
      .place(part_vx)
      .place(part_vy);
```



(c) **"HPB", "SPVDB", "HLEG":** We can easily design new data structures with customized features.

Figure 2-6: The layout language allows users to define data structures using our building blocks. We can reproduce two popular multi-level sparse grid used in simulation (a) (b). Furthermore, we can use our language to design new data structures (c) by chaining and forking elementary components. Hybrid Eulerian-Lagrangian simulations (e.g. FLIP [134] and MPM [112]) often need to maintain both particles and grids, and the required data structures are usually complicated. Using these building blocks, we easily found a data structure with a hierarchical pointer list of particles, which we call *Hierarchical Particle Buckets*, that is especially useful for the material point method simulation (Sec.6 .1).

```
auto k = Index(2);
auto &block = root.dense(ij, {16, 16});
// Child 1: dense array
block.dense(ij, {16, 16}).place(u, v);
// Child 2: dynamic array
block.dynamic(k, 256).place(p);
```

The equivalent C++ code is:

```
struct Child1Node {
  float u;
  float v;
};
struct Block {
  Child1Node child1[16][16];
  std::vector<float> child2; // p
  // Note: in Taichi the dynamic array has a
  // pre-defined maximum size, unlike std::vector that grows
     arbitrarily.
};
struct Root {
  Block blocks[16][16];
};
```

The structural node types are concise, but they are capable of expressing a large variety of data structures. Figure 2-6 illustrates a few complex data structures represented with our language. A new data structure can be designed with a few lines of code. Rapidly experimenting with these data structures allows us to find the optimal one for a specific task and hardware architecture.

4 Domain-Specific Optimizations

Hierarchical data structures provide an efficient representation for sparse fields but have high access costs due to their complexity, especially when parallelism is desired. Our compiler reduces access overhead from three typical sources:

Out-of-cache access. In modern architectures, loading data from main memory is around one hundred times slower than an in-cache access. Ensuring data locality is thus crucial for performance. This is particularly important on GPU, and it means that we need to efficiently utilize shared memory to cache data.

Data structure hierarchy traversal. Traversing hierarchical data structures is expensive. For example, hash table queries may take tens or hundreds of clock cycles. Fortunately, we can often amortize the cost by leveraging spatial locality (Fig. 2-2).

Instance activation. For write access, we need to activate previously inactive nodes. This usually involves atomic operations or spinlocks, which are not only intrinsically slow, but also serialized.

We present three types of optimizations that lead to higher performance, through better cache locality, reduction of redundant accesses, and automatic parallelization and vectorization.

4 .1 Scratchpad Optimization through Boundary Inference

The "scratchpad" pattern is a common optimization to reduce load-to-use latency and memory bandwidth consumption, when potential data reuse exists. Scratchpads are small software-managed local data arenas, typically stored in L1 cache (CPU) or shared memory (GPU), and are intended for fast local computation. But programming with scratchpads is error-prone, and the size of a scratchpad is coupled with the leaf block size.

We provide a construct `Cache(v)` to enable the scratchpad optimization, which can be specified in a kernel. For example, take the discrete Laplace operator from Sec 3 .1. Let us also assume that the inputs are stored in dense arrays at the leaf level (`dense(ij, {4, 4}).place(v)`). Our bound inference engine will infer that each output `u[i, j]` requires values from the 3×3 neighborhood of input `v`, and then allocate a local scratchpad array with the necessary size for this leaf block

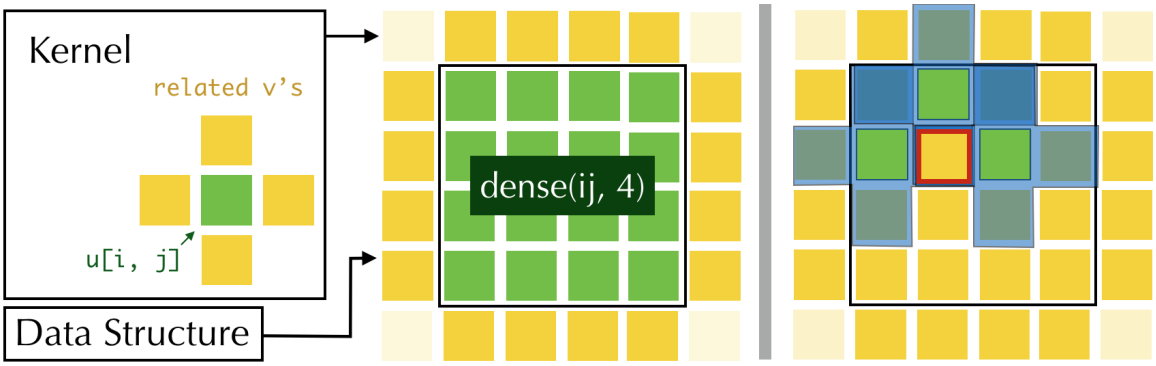


Figure 2-7: **Left and middle:** Combining kernel and data structure information, the compiler will infer the elements required by this compute block. In this specific case, a 6×6 scratchpad will be generated, covering region $[-1, 5) \times [-1, 5)$. **Right:** The v (yellow square with red border) element is loaded into shared memory, and then reused by u five times (three shown in green blocks with their stencil shown in semitransparent blue). By doing this we reduce data load-to-use latency and main memory bandwidth consumption significantly.

(Fig. 2-7, left and middle). We use interval analysis for bounds inference as in Halide [102] to determine a rectangular bound.

Our bounds inference requires the access offsets to be known at compile time. However, in many cases this is too restrictive. Fortunately, oftentimes it is possible to determine bounds using domain knowledge from the data. Therefore we provide an `AssumeInRange` construct for specifying the bounds of individual variables. The compiler then propagates these bounds to generate a scratchpad. For example, in the semi-Lagrangian advection kernel below, the backtrace distance is bounded by the Courant-Friedrichs-Lewy number and supplied to the compiler:

```
Kernel(advect).def([&]() {
  For(m, [&](Expr i, Expr j){
    auto u = velocity(0)[i, j];
    auto v = velocity(1)[i, j];

    auto backtrace_i = Var(i - cast<int32>(u * dt/dx));
    auto backtrace_j = Var(j - cast<int32>(v * dt/dx));

    backtrace_i = AssumeInRange(i, {-2,3});
    // i.e., i - 2 <= backtrace_i < i + 3;
```

```
    backtrace_j = AssumeInRange(j, {-2,3});
    // i.e., j - 2 <= backtrace_j < j + 3;

    m = m_input[backtrace_i, backtrace_j];
}});
```

In our current implementation, the scratchpad optimization is only applied for the GPU backend. The latency and bandwidth difference between software managed shared memory and hardware managed L2 cache makes such an optimization especially profitable on GPU. We anticipate this optimization would also help for the CPU backend, but since the CPU L1 cache already plays a similar role, the improvement might be less significant.

Apart from the `Cache` construct that provides shared memory usage hints, the `CacheL1(V)` construct for the GPU backend instructs the compiler to issue `__ldg` intrinsics to force data loads from the global variable `V` into GPU L1 cache. Unlike x86 CPUs, NVIDIA GPUs cache data in L2 cache by default. Since L1 caches are maintained by GPU hardware on the fly, no compile-time bound inference is needed.

4 .2 Removing Redundant Accesses

As illustrated in Fig. 2-2, the cost of an expensive hierarchical data structure's access can often be amortized. By considering multiple accesses simultaneously, the compiler can leverage common traversal paths. This is a form of constant propagation and common subexpression elimination. We develop a minimal intermediate representation to represent data structure operations. The intermediate representation is specially designed for vectorized accesses and contains explicit information about accesses and data structure boundaries. This allows us to perform optimizations that a typical compiler cannot conduct automatically. We detail the intermediate representation and the expression simplification algorithms in Section 5 and Appendix A .

As an example, consider again the Laplace operator from Sec. 3 .1. This time we

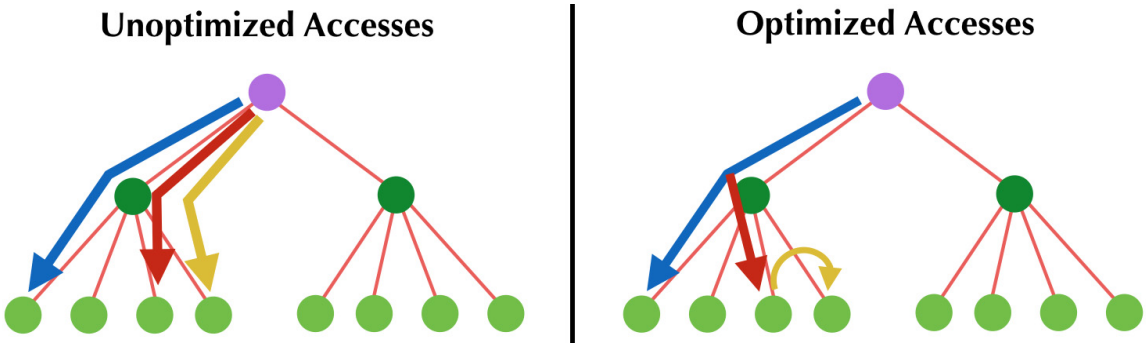


Figure 2-8: Access optimization assuming the three accesses occur from left to right. The common paths of the accesses are eliminated. The yellow access is simplified to a compile-time known offset relative to the red access.

assume we are accessing a three-level data structure like the one in Fig. 2-2. If index `j` is 4-wide loop vectorized, we know that `j` must be a multiple of four and `x[i, j+1]` must share the same ancestor with `x[i, j]`. Therefore, it will be possible to traverse the data structure just once for both `i, j` and `i, j+1`. Our compiler detects this and handles boundary cases using the specific offset information stored in the IR (Fig. 2-8), while traditional compilers' heuristics usually fail to optimize due to code complexity and potential race conditions and pointer aliasing.

A similar optimization can also be applied for write operations. If two write accesses happen in the same memory address in the same kernel, the second write does not need to perform the expensive sparsity check and allocation.

4 .3 Automatic Parallelization and Task Management

Parallelization and Load Balancing Evenly distributing work onto processor cores is challenging on sparse data structures. Naively splitting an irregular tree into pieces can lead to partitions with drastically different numbers of leaf blocks (Fig.2-9).

Our strategy is to generate a task list of leaf blocks, which flattens the data structure into a 1D array, circumventing the irregularity of incomplete trees. Importantly, we generate a task per block instead of a task per element (Fig. 2-5), to amortize the generation cost.

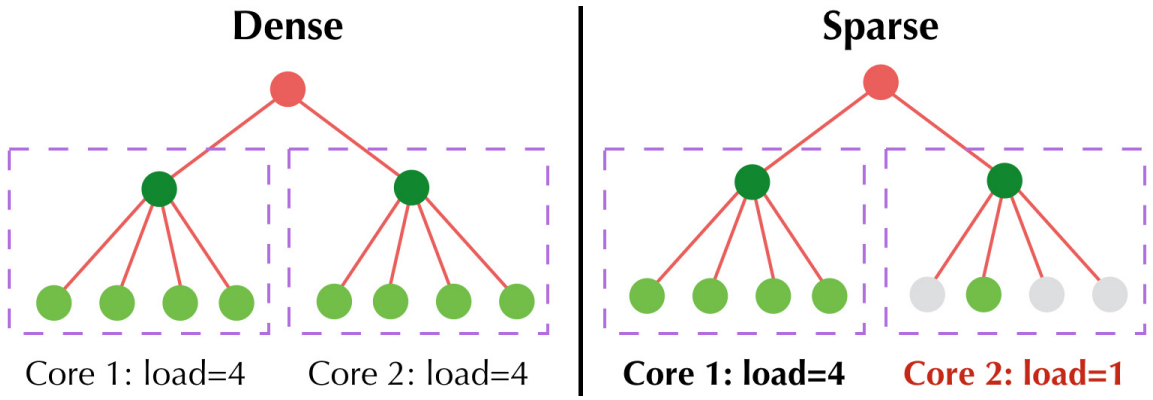


Figure 2-9: Unlike the dense case (left), in sparse data structures, partitioning leaf nodes at a certain level may lead to an unsatisfactory load imbalance and therefore inefficient parallelism (right).

On CPU, generating the task list can be done via a light-weight traversal of the tree in serial. The task list is then queued into a thread pool. We then process the task queue in parallel via OpenMP.

On GPU, generating the task list in serial is infeasible. Instead, we maintain multiple task lists, one for each structural node on the root-to-leaf path. The lists are generated in a layer-by-layer manner: starting from the root node, the queue of active parent nodes is used to generate the queue of active child nodes. A global atomic counter is used to keep track of the current queue head.

Kernel launch management on GPU Synchronizing GPU kernels with the CPU host can be quite costly. In our system, CPU-GPU synchronization (i.e., `cudaDeviceSynchronize()`) will only happen when the user explicitly calls the synchronization function or tries to read/write data from/to the data structure on GPU memory. This design makes asynchronized execution on GPUs transparent to the user.

5 Compiler and Runtime Implementation

The Taichi programming language is embedded in C++14, providing easy interoperability with the host language. (We later released a Python 3 embedding to further lower the language learning barrier and development cost, but let us focus

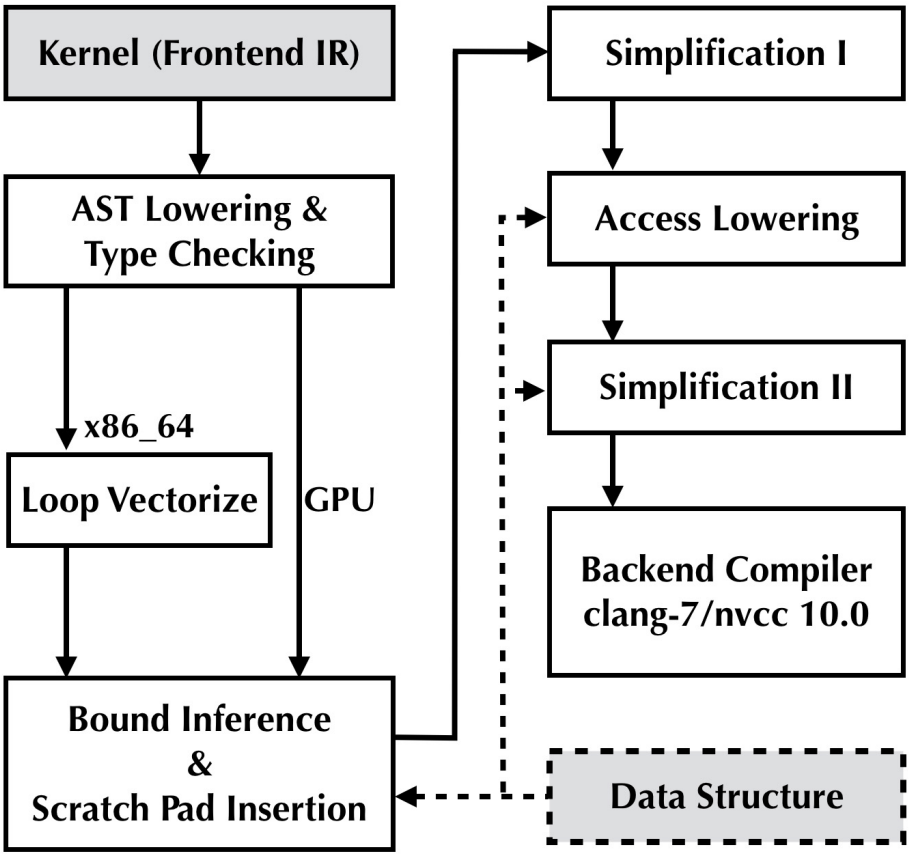


Figure 2-10: The compilation pipeline. The solid lines represent our computation IR pipeline, while dotted lines indicate the use of data structure information.

on the C++ frontend now.) The compiler is implemented in C++17, borrowing infrastructure from the Taichi library [45]. The frontend kernel code is lowered to an intermediate representation before being compiled into standard C++ or CUDA code. Key components of our compiler and runtime are a two-phase simplifier for reducing instructions and removing redundant accesses, an access lowering transform, a customized memory management system for memory allocation and garbage collection, and a CPU loop vectorizer. The compilation workflow is summarized in Fig. 2-10.

Our intermediate representation follows the static single assignment design and is similar to LLVM [72]. Our intermediate representation is more high-level, containing explicit information about data structure accesses, such as the access index bounds and the size of the data structure element. This, combined with data

structure composition information, makes it possible for our compiler to perform automatic access optimizations. The full list of intermediate representation nodes is described in Appendix A . We also include a snippet of compiled code in the supplementary material.

5 .1 Simplification

Apart from the dedicated optimization for the data structure access, our simplification phase applies most common general-purpose compiler optimizations, such as common subexpression elimination, local variable store forwarding, dead instruction elimination, and lowering "if"-statements into conditional moves.

We split the simplification into two phases. The first phase greatly reduces and simplifies the number of instructions and makes it easier for the second simplification phase. In practice we have observed cases where disabling the first phase increases compilation time from a few seconds to tens of minutes. Removing "if"-statements yields bigger straightline code regions, enabling more potentially helpful optimizations.

Central to data structure access simplification are what we call *micro access* instructions: `OffsetAndExtractBit`, `SNodeLookup`, `GetCh`, and `IntegerOffset`. They are produced during the *access lowering* phase, where a root-to-leaf access (e.g. `x[i]`) is broken down into several stages for each level in the hierarchy. Since many different accesses share a similar path from root to leaf, similar micro access operations can be merged. As shown in Table 2.4, disabling the access lowering phase has a significant impact on performance.

The stages of moving down a single hierarchy in the data structure are as follows. First, offsets at each dimension are computed, along with the starting and ending position of each index represented as bit masks (`OffsetAndExtractBit`). This instruction is data-structure-aware. For example, if the kernel is 4-wide loop vectorized over index `j` and the child of the current block has a size larger than 4, we are guaranteed that `OffsetAndExtractBit` will return the same value for `j`

`, j + 1, j + 2, j + 3`. Inference like this allows us to aggressively simplify accesses. Next, the extracted multi-dimensional indices are flattened into a linear offset (`Linearize`). Then a pointer to the item in the data structure is fetched from the current level of the data structure using the linear offset, along with a check of whether the node is active or not (`SNodeLookUp`). We need to pay special attention to `SNodeLookUp` when the node is not active: for read accesses `SNodeLookUp` returns an "ambient node" with all fields being ambient values such as 0; for write access `SNodeLookUp` first allocates the node and then return the new node. Finally the corresponding field in the item is fetched (`GetCh`).

In cases where two micro access instructions of the same type lead to a compile-time-known non-zero offset, we replace the second micro access instruction with an `IntegerOffset` instruction, representing the relationship between the two accesses in bytes, avoiding data structure traversals.

5 .2 Memory Management

Our system relies heavily on the allocation-on-demand mechanism and supports data structures with dynamic topology. Therefore, efficient management of memory is a key to performance, especially on massively parallel GPUs.

Memory allocators for variable size requests usually need complex data structures to maintain available segments, leading to an unacceptable runtime cost. Therefore, we designed a memory management system that needs only very simple data structures, specialized for our abstraction.

The memory manager has a memory allocator tailored for each node that requires on-demand allocation, e.g. the pointer and hash nodes. The benefit of having multiple allocators is that each allocator only needs to allocate memory segments of a fixed size, which greatly simplifies and accelerates the process.

To minimize the internal data structure used by each memory allocator, we conservatively reserve a memory pool from our virtual address space, whose size is equal to the amount of physical memory. Only the actual used space will become

Table 2.1: Benchmarks. Commands to reproduce our performance numbers are provided in detailed tables in each subsection. Geometric means of the four benchmarks where access has strong coherence are calculated for the summary. "-Opt" means with our domain-specific optimizations off, leaving the code generation and optimization to the backend general-purpose compiler; "+Opt" means with our optimizations on. If we include comparisons of our GPU backend with reference CPU implementations, we are on average $4.55\times$ faster, otherwise $2.82\times$ faster on the same hardware. Machine specifications for each benchmark are detailed in Appendix B. `clang-format-6.0` was used to reformat the code into the same style to get fair lines of code (LoC) numbers, with a right margin at 80 characters and all empty lines removed.

Benchmark	Reference Timing	CPU-Opt	CPU+Opt	GPU-Opt	GPU+Opt	Ref. LoC	Ours LoC
MLS-MPM	3.85ms (GPU, Pascal)	-	-	7.24ms	3.15ms	3091	237
FEM Linear Elasticity	30.71ms (CPU, AVX2)	182.19ms	17.16ms	11.78ms	2.11ms	267	21
MGPCG Solver	2.20s (CPU)	5.68s	2.98s	1.78s	1.13s	~ 2000	~ 300
Sparse CNN	37.44ms (GPU, Turing)	-	-	5.56 ms	3.02 ms	183	20
Summary (coherent cases) :	**Ours:Ref=**$2.82\times$	**Opt On:Off=**$3.02\times$		**GPU:CPU=**$4.63\times$		$\sim 10.0\times$ **shorter code**	
Volumetric Path Tracing	554.14s (CPU)	243.69s	232.52s	2.34s	2.35s	-	-

a resident page in physical memory. This design allows us to implement memory allocation with a single integer atomic operation.

We make heavy use of the virtual memory system in modern operating systems, inspired by the SPGrid virtual memory design [110]. The runtime system will first reserve a virtual address space of size 2^{40}B $= 1$TB. The memory pages will not be allocated immediately, but in an on-demand manner, with pages zero-initialized by the hardware. We use the unified memory access feature on NVIDIA GPUs, thus this address space is shared by the CPU and GPU.

We additionally maintain a list of metadata for each block, including its memory location and coordinates.

5 .3 Loop Vectorization on CPUs

We designed a loop vectorizer to utilize vector instruction sets such as SSE and AVX2/512 on modern CPUs. The design is similar to ISPC [99] where masking is used to avoid side effects of diverging control flow. We ensure that access to data structures is done through vectorized loads and writes whenever possible.

Vectorized memory access on CPUs To achieve good memory behavior, it is necessary to issue vectorized memory operations instead of scalar loads.[3] We emit SIMD loads and then blend instructions to make maximum usage of the vector units, based on the compile-time-known offset information after the access simplifications (Fig. 2-11).

5 .4 Interaction with the Host Language

Our language can interact with the C++ host language easily. C++ can be used to initialize the data, invoke the compiled kernels, and possibly store the outputs. After a `Kernel`, `laplace`, has been defined, it can be used as follows:

[3] On GPUs this optimization is done via the memory coalescing hardware on the fly, relieving the compiler of the burden of this optimization.

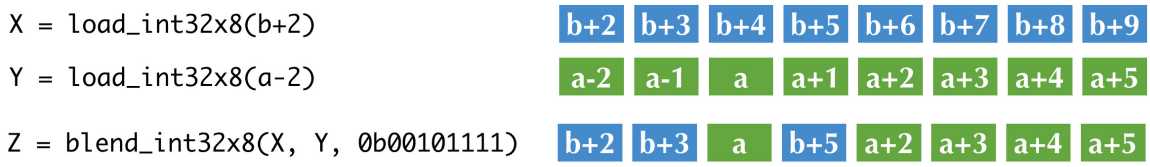


Figure 2-11: Loading an 8-wide vector with elements $[x_{b+2}, x_{b+3}, x_a, x_{b+5}, x_{a+2}, x_{a+3}, x_{a+4}, x_{a+5}]$. The Taichi compiler, to utilize AVX instructions on x86 for high performance, only issues two vectorized loads that fetch contiguous data from memory, and then a SIMD blend to generate the desired vector, with binary mask "00101111". Note that a naive data loading code generator would issue one scalar load, one scalar to vector promotion, one vector shuffle, and finally one vector blend instruction, for *each* element in the vector.

```
// Initialize
for (int i = 0; i < n; i++)
  for (int j = 0; j < 32; j++)
    x.val<float32>(i, i + j) = sin(j);

// Run the kernel on the active region
laplace();

// Output
printf("%f\n", y.val<float32>(n/2, n/2));
```

6 Evaluation and Applications

In this section, we evaluate our language on end-to-end applications for large-scale visual computing tasks covering physical simulation, rendering, and 3D deep learning. The results are summarized in Table 4.1. In computation with coherent accesses, our domain-specific optimizations boost performance by a geometric mean of $3.02\times$ on the same device. Our implementations require $\frac{1}{10}$ as many lines of code and run $2.82\times$ faster than the reference implementations. The code for our implementations can be found in the supplementary material.

6 .1 Moving Least Squares Material Point Method

The Material Point Method [115, 112] is a hybrid Eulerian-Lagrangian method, and is one of the state-of-the-art approaches for elastoplastic continuum simulation. The method is challenging to implement efficiently due to the interaction between particles and grids. Gao et al. [35] implemented a high-performance Moving Least Squares Material Point Method [48] solver on GPU with intensive manual optimization[4], including

1. A tailored SPGrid variant on GPUs;

2. Staggered particle-block ownership (Fig. 2-15, left and middle) for parallel scattering, with shared memory utilization;

3. Warp-level reductions to reduce atomic operations during scattering;

4. Dedicated sorting and delayed reordering to reduce memory bandwidth consumption.

It took us a few attempts, but thanks to the easy data structure exploration supported by our language, we eventually surpassed their performance by 18%. We initially followed the structure of arrays (SOA) particle layout in their reference implementation. Although we are easily able to implement optimization (1) and (2) within ten lines of code (instead of hundreds in the reference implementation), the warp-level optimization (3) is below our level of abstraction[5], and we did not implement the complex sorting and reordering scheme (4) for simplicity. When particles are perfectly sorted, we were able to achieve comparable performance with the reference implementation. However, when the simulation progresses and the spatial distribution of particles changes, our performance drops drastically (Table 2.2, row "SOA"), especially when simulating liquids.

[4]We obtained their open-source CUDA solver and did further performance optimizations which made this reference implementation $1.98\times$ faster, and carefully confirmed that we have achieved the best-human-effort performance following their design decisions.

[5]For portability, we do not provide warp-level intrinsics such as `__ballot`.

Table 2.2: [35] used an SOA particle layout that makes sequential access efficient, yet complex sorting and reordering schemes are needed. When particles' attributes are randomly shuffled in memory, the simulation runs $6.03\times$ slower due to insufficient GPU cacheline utilization under random memory access. Our AOS particle layout is easy to implement and, more importantly, less sensitive to particle order, because even under random particle access order, different attributes of the particle stay in the same or nearby cacheline. Unlike CPUs, NVIDIA GPUs have no prefetching, so cacheline usage is key to performance, and access predictability is of less importance. This makes sorting unnecessary, leading to a much simpler and more efficient algorithm.

Particle Layout	Ordered	Randomly Shuffled
SOA	3.52ms	21.23 ms
AOS	3.15ms	4.28 ms

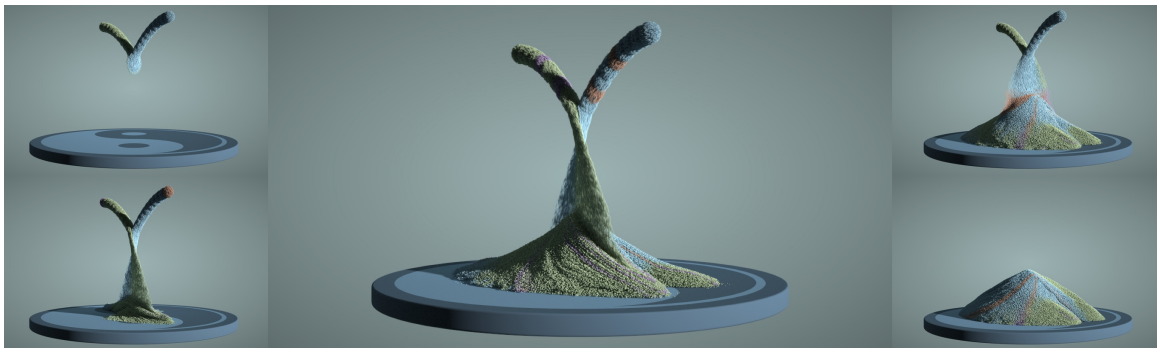


Figure 2-12: A sand jet animation using MLS-MPM with up to 3 million particles, simulated using on average 2.2 sec/frame (100 substeps/frame).

Fortunately, using our language we were able to quickly explore different particle/grid layout schemes and found that switching particle layouts from structure of arrays to array of structures resolved this issue (Table 2.2, row "AOS"). In contrast, in the reference implementation the data layout is tightly coupled with the computational kernels, making it difficult to experiment with different data structures.

The data structure code for the high-performance data structure we found for MPM is illustrated in Figure 2-13. For particles we use array of structures, for grids we use structure of arrays, and each block maintains a list of indices of its contained particles. This greatly simplifies the data structure and algorithms used by Gao et

Table 2.3: Using scratchpad memory (SPM, a.k.a. “shared memory" on NVIDIA GPUs) makes the P2G kernel $2.54\times$ faster and G2P kernel $2.73\times$ faster. In our language this optimization can be easily achieved with the “Cache” hint.

	GPU-SPM	GPU+SPM
P2G	5.102ms	2.011ms
G2P	1.975ms	0.722ms

al., for example we avoid the complex radix sort of the particles. The original grid hierarchy used by [35] is sparse yet bounded. This leads to simplicity and lower access cost, yet often leads to unnatural behavior when the simulation bound cannot be predetermined. In our language, adding a `hash` or `dense().pointer()` node at the top level of the grid conveniently makes the simulation domain virtually unbounded, which will suit corresponding boundary conditions (Figure 2-14).

Our implementation has only four kernels: sort particle indices to their containing blocks, particle to grid (P2G), grid normalization, and grid to particle (G2P). In contrast, the reference implementation has over 20 kernels, with the majority of them dealing with data structure maintenance. Our compiler automatically generates code to maintain the topology of the data structure. For example, it automatically activates a block and its parents when a particle touches it.

In the P2G and G2P kernels, we use the `AssumeInRange` construct to hint to the compiler the spatial relationship between blocks and their containing particles. We also apply Gao et al.'s stagger particle-grid ownership optimization by offsetting the particle position by Δx (Fig. 2-15), leading to a tighter access bound at the parent level. The compiler will automatically allocate scratchpads for each particle's $3 \times 3 \times 3$ span on each $4 \times 4 \times 4$ block, which is a $6 \times 6 \times 6$ scratchpad in shared memory. We did an ablation study on the scratchpad optimization, and it indeed leads to a significant speedup (Table 2.3).

Examples of MLS-MPM sand and liquid animation simulated and rendered with Taichi programs are shown in Fig. 2-12 and Fig. 2-16.

```
auto i = Index(0), j = Index(1), k = Index(2);
auto p = Index(3);
auto &fork = root.dynamic(p, max_n_particles);
// Particle array of structures
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    fork.place(particle_F(i, j)); // 3x3 force matrix
// ... do the same for other particle attributes
// Grid structures of arrays
auto &block = root.dense({i, j, k}, n / grid_block_size).pointer();
block.dense({i,j,k}, grid_block_size).place(grid_v(0));
block.dense({i,j,k}, grid_block_size).place(grid_v(1));
block.dense({i,j,k}, grid_block_size).place(grid_v(2));
block.dense({i,j,k}, grid_block_size).place(grid_m);
// Each block stores a list of particle indices
block.dynamic(p, pow(grid_block_size, 3) * 64).place(l);
```

Figure 2-13: The data structure code for our material point method simulation. The interaction between particle and grid in this hybrid-Eulerian-Lagrangian approach leads to a huge space of potential data structure designs. We use array of structures for the particles and structure of arrays for the grids. We also store a dynamic list of particles in each block for speeding up particle lookup (the "Hierarchical Particle Buckets" in Fig. 2-6). We can easily modify the code to change the layout, or switch to a hash table for the top level of the grid to achieve an unbounded domain (Fig. 2-14).

6 .2 Linear Elasticity Finite Element Kernel

A large scale sparse grid-based finite element solver was presented by Liu et al. [79] for high-resolution topology optimization. They proposed a matrix-free elasticity operator for the conjugate gradient iterations on x86_64 with vectorization. Their hand-optimized kernel is tailored for SPGrid [110], with carefully implemented vectorized load instructions (e.g. via the _mm256_loadu_ps intrinsic). This is a highly compute-bound task. For each voxel, over one thousand multiply and add instructions are issued, while fetching material parameters from only $2 \times 2 \times 2$ cells and 3D displacements from $3 \times 3 \times 3$ nodes. The whole algorithm is gather-only so it parallelizes naturally. We consider Liu et al.'s code a highly-optimized reference implementation for evaluating our language and compiler in a compute-bound situation.

We reproduced their algorithm in our language. Our compiler is especially

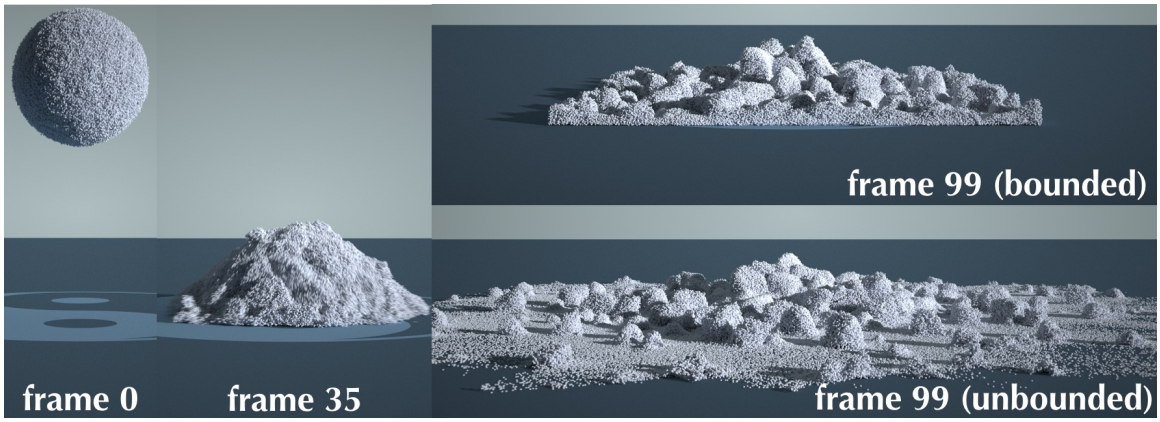


Figure 2-14: Smashing a snow ball onto the ground: bounded vs. unbounded simulation. By changing data structures (and boundary conditions), we can easily switch to a virtually unbounded domain.

good at compute-bound tasks, as our access optimization and auto-vectorization significantly reduce the number of instructions (Table 2.4). With all optimizations on, our implementation is $1.77\times$ faster on an x86 CPU. Without modifying the code, our program runs on a GPU $8.2\times$ faster than the generated CPU code, and $14.6\times$ faster than the reference CPU implementation. We conduct a comprehensive ablation study of our compiler optimizations in Table 2.4, and found our compiler optimizations lead to $10.6\times$ and $5.58\times$ higher performance on CPU and GPU.

6 .3 Multigrid Poisson Solver

Large-scale Poisson equation solving has extensive use in graphics, including fluid simulation [82], image processing [5] and mesh reconstruction [65]. We implement a Multigrid-Preconditioned Conjugate Gradients (MGPCG) solver [86], which has become popular for pressure projection in physically based animation.

We implemented a simplified version of the reference implementation, with the following differences:

- Smoothers: the reference implementation uses Gauss-Seidel for boundary smoothing and damped Jacobi for interior smoothing, while we used red-black Gauss-Seidel smoothing for both boundary and interior regions.

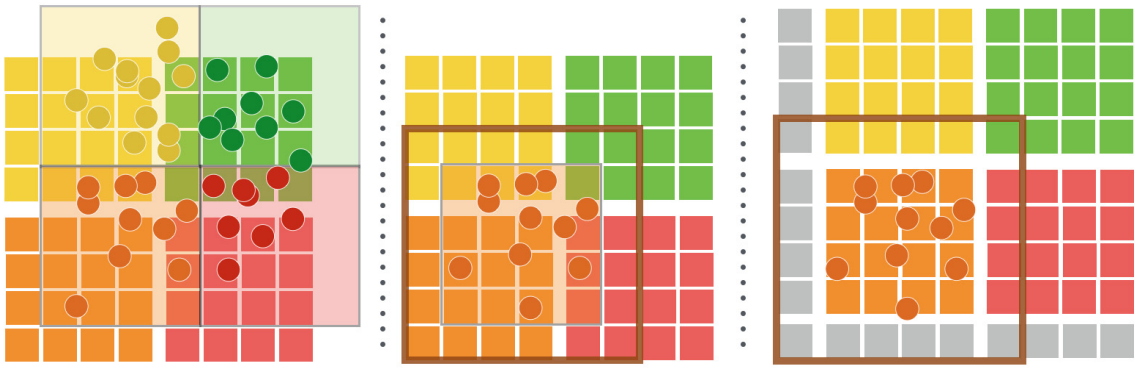


Figure 2-15: Optimizing particle sorting. **Left:** We first sort the indices of particles to their respective grid blocks. P2G and G2P can then be done in a block-wise manner with high locality. **Middle:** We sort particles to a block staggered by Δx. Gao et al. [35] describe a similar optimization: by doing so, particles sorted to each block will touch $2 \times 2 \times 2$ blocks only, instead of $3 \times 3 \times 3$ blocks in the case without staggering. **Right**: Note the extra grey cells without staggering. Since our compiler can automatically apply bounds inference, we quickly experimented with this approach and observed a $1.29\times$ speed up.

- Restriction and prolongation: instead of using the $4 \times 4 \times 4$ trilinear interpolation operator, we use $2 \times 2 \times 2$ averaging.
- Boundary conditions: we support zero Dirichlet boundary conditions only, while the reference implementation also supports Neumann boundaries and their coarsening.
- Operator fusing: the reference implementation aggressively fuses operations, such as smoothing and dot products, to save memory bandwidth. We use temporary buffers to store some of these results to simplify the code.

Fully implementing McAdams et al.'s algorithm is possible in our language. For this specific benchmark we only need a simplified version.

A user experienced with both our language and physical simulation was able to implement our multigrid preconditioner within only 80 minutes and 300 lines of code.

We run both our implementation and reference on an x86 CPU until convergence. Our performance is $1.35\times$ lower than the reference, likely because our implementation has a slightly inferior convergence rate and uses temporary buffers

Table 2.4: An ablation performance study on the linear elasticity FEM kernel. Disabling the simplification I pass before lowering access does no harm to run-time performance, yet it increases compilation time from several seconds to 40 minutes when generating CPU code. Disabling the simplification II pass after lowering access leads to a binary size of 8.1MB instead of 377KB, since clang failed to remove redundant accesses. Using a bad data layout makes performance drop by nearly an order of magnitude.

Ablation	CPU Time	GPU Time
No multithreading	73.43ms	-
No vectorization	83.54ms	-
No vectorized load instructions	22.69ms	-
No simplification I	17.01ms	2.13 ms
No access lowering	182.19ms	6.046 ms
No simplification II	85.51ms	11.784 ms
AOS instead of SOA	136.03ms	20.992 ms
All optimizations on	17.16ms	2.11 ms

to simplify the code. On the other hand, changing our backend to GPU requires no effort, and it runs $2.64\times$ faster than our CPU version and $1.9\times$ faster than the reference implementation. An ablation study on our compiler optimizations and parallelization is shown in Table 4.5.

Our solver automatically generalizes to an irregular and sparse case, while the reference implementation deals with only dense grids. To implement this multi-resolution approach, we generated a structural tree of 36 nodes and 31 kernels for

Table 2.5: An ablation performance study on the MGPCG Poisson solver.

Ablation	CPU Time	GPU Time
No multithreading	7.30s	-
No vectorization	4.01s	-
No access lowering	5.68s	1.78 s
All optimizations on	2.98s	1.13 s

a five-level multigrid hierarchy.

The performance data are obtained from a 256^3 dense grid, with zero Dirichlet surrounding voxels. The initial right-hand side is an analytical field $\sin(2\pi x)\cos(2\pi y)\sin(2\pi z)$, with $x, y, z \in [0, 1)$ being the spatial coordinates. Initial guesses for the conjugate gradient are set to zero. We stop iterating when the l_2 norm of residual is reduced by a factor of 10^6.

The same solver can potentially be used for other graphics applications such as panorama image stitching [5] or mesh reconstruction [65].

6 .4 3D Convolutional Neural Networks

3D deep learning requires convolutional neural networks to operate on voxels instead of images. Unlike images, voxels require an efficient sparse representation for high-resolution 3D objects. Several sparse voxel approaches have been proposed for 3D deep learning [106, 37, 122, 123]. We implemented a 3D convolution layer, operating on multi-channel sparse voxels. The kernel is as simple as the mathematical definition of convolution, while the compiler automatically generates the code for efficiently accessing the sparse voxels. We compare to the Sparse Convolutional Network [37] implemented in CUDA, which uses a hash table with pointers to a dense matrix to store sparse 3D feature maps. We take the Stanford bunny, voxelize it into a $256 \times 256 \times 256$ grid, and copy over 16 channels. We then apply a $3 \times 3 \times 3 \times 16 \times 16$ convolution layer. By using a two-level hierarchy with pointer arrays, under 1% sparsity, we are roughly 12 times faster than the reference code. Under 10% sparsity, we are 23 times faster. We use the `CacheL1` schedule to cache the convolution weights in GPU L1 cache. This schedule hint boosts performance by $1.8\times$.

6 .5 Volumetric Path Tracing

We implemented a volumetric path tracer inspired by Mitsuba [57]'s implementation, with Woodcock tracking [129] and an isotropic phase function. We compare

against the Tungsten renderer[6], which uses VDB [89] to represent volumes.

The benchmark scene includes a $584 \times 576 \times 440$ density field containing bunny-shaped smoke and a single point light source. We rendered 512×512 images with 128 samples per pixel and a path length limit of 128. On CPU, our implementation is 2.38$\times$ faster than the reference implementation. Our GPU version is 98.86$\times$ faster than our CPU version and 235.6$\times$ faster than the reference implementation. Our domain-specific optimizations only lead to a 5% performance boost on CPU and no performance improvement on GPU, since the access pattern is largely incoherent in volume rendering. Still, we obtain a fast GPU renderer with no additional implementation, and are able to explore different sparse data structures.

We made our best effort to match our implementation to Tungsten's. With slight modifications to both renderers[7], we get qualitatively similar results (see the supplemental material).

7 Limitations

Although we in general get satisfactory results on the five benchmark cases, there are limitations and potential for future work:

Low arithmetic intensity tasks In the material point method (Sec. 6 .1) and finite element method (Sec. 6 .2) cases, when the performance is compute-bound, our access optimizer can greatly improve performance by reducing access instructions. However, in the multigrid Poisson solver case (Sec. 6 .3), although the optimizer improves performance by a factor of $1.9\times$, we are soon bounded by memory bandwidth. In these cases reducing instructions no longer helps. As a result, the unvectorized reference implementation is still faster than our vectorized implementation by $1.3\times$. This is because the reference is more bandwidth-efficient, due to operator fusing optimizations. This suggests investigating approaches that can automati-

[6]https://github.com/tunabrain/tungsten

[7]We implemented Woodcock tracking in Tungsten, and used a two-level grid in our implementation to approximate the OpenVDB hierarchical DDA traversal [90] in Tungsten.

cally fuse operators, which might require further decoupling of computation and scheduling [102].

Less coherent accesses. For the volume rendering example (Sec. 6 .5), while the rays exhibit some coherent behavior, our compiler is not able to infer this at compile time. Approaches that extract locality information at run-time such as ray reordering [98] could potentially be used to boost performance.

8 Related Work

8 .1 Array Compilers

Many programming models for efficiently compiling array operations have been proposed.

Halide [102, 104] decouples image processing operations and lower-level scheduling such as loop transformations and vectorization. Several polyhedral compilers adopt a similar idea [7, 88, 120, 8]. All these compilers focus on dense data structures and do not model sparsity. Our language decouples algorithms from the internal organization of sparse data structures, allowing programmers to quickly switch between data organizations to achieve high performance.

Several sparse tensor compilers target linear algebra operations (e.g. taco [67, 22], ParSy [20, 21]). They focus on constructing efficient iteration spaces between different sparse matrices under linear algebra operations. Several compilers target graph operations such as breath-first-search or shortest path (e.g. [125, 133]). In contrast, we focus on generating high-performance traversal code for spatially coherent access to hierarchical and sparse data structures.

To efficiently vectorize access to data structures, we adopt the Single-Program-Multiple-Data model [26] in our computational kernels, which is the foundation of modern parallel languages such as CUDA, OpenCL [113], ispc [99], and IVL [74].

Physical Simulation Languages Several domain-specific languages exist for physical simulation. They usually abstract the domain as a graph structure for representing meshes. Liszt [30] focuses on solving partial differential equations on meshes. Simit [69] models the domain as sparse matrices while Ebb [13] employs a relational data model. We provide a different abstraction for lower-level optimizations, focusing on hierarchical sparse data structures.

8 .2 Data-Oriented Design

Inspired by the increasing relative expense of memory operations, the video game and visual effects industries have recently started to adopt the data-oriented design philosophy [4, 73]. It is a software engineering approach focused on data access, as opposed to the more traditional object-oriented design where the storage is fragmented. Adopting a similar philosophy, ispc [99] and IVL [74] both provide constructs for transformations between array of structures and structure of arrays. Our language facilitates data-oriented design and shares the same philosophy through decoupling of data structures and computation.

8 .3 Hierarchical Sparse Grids in Graphics

Computer graphics, especially in the field of physical simulation, has a long history of using multi-level sparse regular grids for finite element methods, level set methods [95], or Eulerian fluid simulation. Sparse grids are used for representing large-scale simulation data. Bridson [15] uses a two-level grid, Houston et al. [43] use run-length-encoding to compress data. DTGrid [93] employs compressed-row-storage. VDB [89] uses a static B+tree-like structure to represent an unbounded domain. SPGrid [110] uses a shallow hierarchy while utilizing the modern virtual memory system. GPU variants of VDB [131, 41] and SPGrid [35] have recently been designed. Nielsen and Bridson [92] propose a wide-branching tile tree of voxels for fluid simulation. Bailey et al. [9] sort particles to corresponding voxel blocks, similar to our "Hierarchical Particle Buckets" described in Section 6 .1. Outside of

simulation, Kazhdan et al. [65] and Agarwala [5] used octrees for solving Poisson's equation for image stitching and mesh reconstruction, respectively. Chen et al. [19] use a hierarchical grid for storing signed distance fields.

9 Conclusion

We have presented a new programming language and its optimizing compiler for developing high-performance implementations of sparse visual computing tasks. Our novel design allows the language to provide both productivity and performance.

The computation-data structure decoupling allows the programmer to quickly explore different data structure hierarchies. As an example, we used this successfully to find a new efficient layout for the material point method, demonstrating the potential of the language for developing novel, high-performance data structures.

Our compiler's automatic parallelization and access optimizations are especially useful in reducing the number of instructions for compute-bound tasks, while the scratchpad optimization improves memory locality. Our compiler enables programmers, for the first time, to implement optimized large-scale simulations within a few hundred lines of code.

Appendix for chapter 1

A Intermediate Representation Instructions

Our intermediate representation follows the typical static single assignment form. It contains the following control flow nodes: `StructFor` (for looping data structures), `RangeFor` (for looping data over ranges), `If`, `While`, `WhileControl` (while combined with break).

The expression tree contains the following nodes: `Const`, `Alloca` (for local mutable variables), `UnaryOp`, `BinaryOp`, `TrinaryOp`, `Rand`, `ElementShuffle` (for loop vectorization), `RangeAssumption` (for bound inference).

When a data structure is accessed, `GlobalLoad` and `GlobalStore` are issued with addresses pointed to by `GlobalPtr`. `SNodeOp` is used to activate grids and check for sparsity. When a local mutable variable is accessed, `LocalStore` and `LocalLoad` are issued. Atomic instructions are represented by `AtomicOp`. `ClearAll` cleans up the data structures.

As mentioned in Sec. 5 .1, `GlobalPtr` is lowered to the following micro access nodes, to facilitate expression simplification:

- `OffsetAndExtract`
- `Linearize`
- `SNodeLookup`
- `GetCh`
- `IntegerOffset`

B Benchmark Machine Specifications

Here we list the machine specifications for our benchmarks for reproducing the performance numbers.

The MLS-MPM, FEM and MGPCG benchmarks were done on an Intel Core i7-7700K CPU with four cores at 4.2GHz, 32 GB main memory, and an NVIDIA GTX 1080Ti graphics card.

The CNN benchmark was done on an Intel Core-i7 9800X CPU with eight cores at 3.8GHz, 32 GB main memory, and an NVIDIA RTX 2080 GPU.

The rendering benchmark was done on an Intel Xeon E5-2690 v4 with 28 cores at 2.60GHz, 64 GB main memory, and an NVIDIA Tesla V100 GPU.

`clang-7` and `nvcc 10.0` were used as backend compilers.

Although finding a machine with these exact specifications may be difficult, the relative performance numbers are roughly machine-independent. We encourage the reader to run the example programs with the provided commands to reproduce our results, and to explore different combinations of data structures and compiler optimizations.

C Intermediate Representation Sample

Below is a simple kernel and its IR, in multiple compilation stages.

```
// Kernel
Kernel(inc).def([&]() {
  For(a, [&](Expr i) {
    a[i] = b[i] + 1;
  });
});
// AST
for tmp2 where a_global active {
  #a_global[tmp2] = (load #b_global[tmp2] + 1)
}
// IR
<i32x1> $0 = alloca
for $0 where a active, step 1 {
  $2 = local load [ [$0[0]]]
  <i32x1> $3 = ptr [S1], index [$2] activate=false
  $4 = global load $3
```

```
  <i32x1> $5 = const [1]
  $6 = add $4 $5
  $7 = local load [ [$0[0]]]
  <i32x1> $8 = ptr [S0], index [$7] activate=true
  $9 : global store [$8 <- $6]
}
// Access Lowered
<i32x1> $0 = alloca
for $0 where S0 active, step 1 {
  <i32x1> $2 = local load [ [$0[0]]]
  <i32x1> $3 = ptr [S1], index [$2] activate=true
  <i32x1> $4 = shuffle $2[0]
  $5 = linearized(ind {}, stride {})
  $6 = [S4][root]::lookup(root, $5) coord = {$4} activate = false
  $7 = get child [S4->S3] $6
  $8 = bit_extract($4 + 0, 7~14)
  $9 = linearized(ind {$8}, stride {128})
  $10 = [S3][dense]::lookup($7, $9) coord = {$4} activate = false
  $11 = get child [S3->S2] $10
  $12 = bit_extract($4 + 0, 0~7)
  $13 = linearized(ind {$12}, stride {128})
  $14 = [S2][dense]::lookup($11, $13) coord = {$4} activate = false
  $15 = get child [S2->S1] $14
  <i32x1> $16 = shuffle $15[0]
  <i32x1> $17 = global load $16
  <i32x1> $18 = const [1]
  <i32x1> $19 = add $17 $18
  <i32x1> $20 = ptr [S0], index [$2] activate=true
  <i32x1> $21 = shuffle $2[0]
  $22 = linearized(ind {}, stride {})
  $23 = [S4][root]::lookup(root, $22) coord = {$21} activate = false
  $24 = get child [S4->S3] $23
  $25 = bit_extract($21 + 0, 7~14)
  $26 = linearized(ind {$25}, stride {128})
  $27 = [S3][dense]::lookup($24, $26) coord = {$21} activate = false
  $28 = get child [S3->S2] $27
```

```
  $29 = bit_extract($21 + 0, 0~7)
  $30 = linearized(ind {$29}, stride {128})
  $31 = [S2][dense]::lookup($28, $30) coord = {$21} activate = false
  $32 = get child [S2->S0] $31
  <i32x1> $33 = shuffle $32[0]
  <i32x1> $34 : global store [$33 <- $19]
}
// Final Optimized
<i32x1> $0 = alloca
for $0 where S0 active, step 1 {
  <i32x1> $2 = local load [ [$0[0]]]
  $3 = linearized(ind {}, stride {})
  $4 = [S4][root]::lookup(root, $3) coord = {$2} activate = false
  $5 = get child [S4->S3] $4
  $6 = bit_extract($2 + 0, 7~14)
  $7 = linearized(ind {$6}, stride {128})
  $8 = [S3][dense]::lookup($5, $7) coord = {$2} activate = false
  $9 = get child [S3->S2] $8
  $10 = bit_extract($2 + 0, 0~7)
  $11 = linearized(ind {$10}, stride {128})
  $12 = [S2][dense]::lookup($9, $11) coord = {$2} activate = false
  $13 = get child [S2->S1] $12
  <i32x1> $14 = global load $13
  <i32x1> $15 = const [1]
  <i32x1> $16 = add $14 $15
  $17 = get child [S2->S0] $12
  <i32x1> $18 : global store [$17 <- $16]
}
```

D MGPCG code comparison

Below is the 7-point stencil code

(Relaxation_And_Dot_Product_Interior_Helper.h) from [86]. Note although a dense 2-level grid is used (instead of a more complex sparse one), the code is

already convoluted. The redundant code is necessary, however, for performance since they enforce complex strides to be evaluated at compile time.

```
enum WORKAROUND {
  x_block_size = 4,
  y_block_size = 4,
  z_block_size = 4,
  padded_y_size = y_size + 2,
  padded_z_size = z_size + 2,
  coarse_y_size = y_size / 2,
  coarse_z_size = z_size / 2,
  coarse_padded_y_size = coarse_y_size + 2,
  coarse_padded_z_size = coarse_z_size + 2,
  x_shift = padded_y_size * padded_z_size,
  y_shift = padded_z_size,
  z_shift = 1,
  coarse_x_shift = coarse_padded_y_size * coarse_padded_z_size,
  coarse_y_shift = coarse_padded_z_size,
  coarse_z_shift = 1,
  x_plus_one_shift = x_shift,
  x_minus_one_shift = -x_shift,
  y_plus_one_shift = y_shift,
  y_minus_one_shift = -y_shift,
  z_plus_one_shift = z_shift,
  z_minus_one_shift = -z_shift
};
...
template <class T, int y_size, int z_size> void
Relaxation_And_Dot_Product_Interior_Size_Specific_Helper
<T, y_size, z_size>::
Compute_Delta_X_Range(const int xmin,
                      const int xmax,
                      const int partition_number) {
  const T one_sixth = 1. / 6.;
for(int block_i=xmin;block_i<=xmax;
    block_i+=x_block_size)
for(int block_j=1;block_j<=y_size;block_j+=y_block_size)
```

```
for(int block_k=1;block_k<=z_size;block_k+=z_block_size)
for(int i=block_i;i<block_i+x_block_size;i++)
for(int j=block_j;j<block_j+y_block_size;j++)
for(int k=block_k;k<block_k+z_block_size;k++) {
  int index = i * x_shift + j * y_shift + k * z_shift;
  delta[index] = one_sixth * (
    u[index + x_plus_one_shift] +
    u[index + x_minus_one_shift] +
    u[index + y_plus_one_shift] +
    u[index + y_minus_one_shift] +
    u[index + z_plus_one_shift] +
    u[index + z_minus_one_shift]) -
    u[index];
}}
```

Code in our language that defines the data structure and the same stencil:

```
int block_size = 4;
layout([&] {
    root.dense(ijk, n / block_size)
        .dense(ijk, block_size).place(u);
    root.dense(ijk, n / block_size)
        .dense(ijk, block_size).place(delta);
});

Kernel(relaxation).def([&]{
  Parallelize(8);
  For(y, [&](Expr i, Expr j, Expr k){
    delta[i, j, k] = 1.0f/6.0f *
      (u[i, j, k - 1] + u[i, j, k + 1] + u[i, j - 1, k]
     + u[i, j + 1, k] + u[i - 1, j, k] + u[i + 1, j, k])
       - u[i, j, k];
  });
});
```

E Volumetric Path Tracing Renderings

A visual comparison of our rendering result compared to Tungsten's is in Fig. 2-17.

F Simplification Pass Details

The simplification pass of the Taichi compiler does the following optimizations:

- Dead instruction elimination.
- Common subexpression elimination.
- Store forwarding.
- Useless local store elimination, i.e. delete stores whose results that are overwritten by future stores.
- Simplify if statements into conditional stores.

Two simplifications passes are applied to the program, before and after the access lowering pass, respectively. The main goal of the first pass is to reduce the number of instructions, while the second pass is mainly to optimize access. Note that the access simplification is essentially a common subexpression elimination pass over the micro-access instructions.

Other possible optimizations such as constant folding are left for the backend compiler.

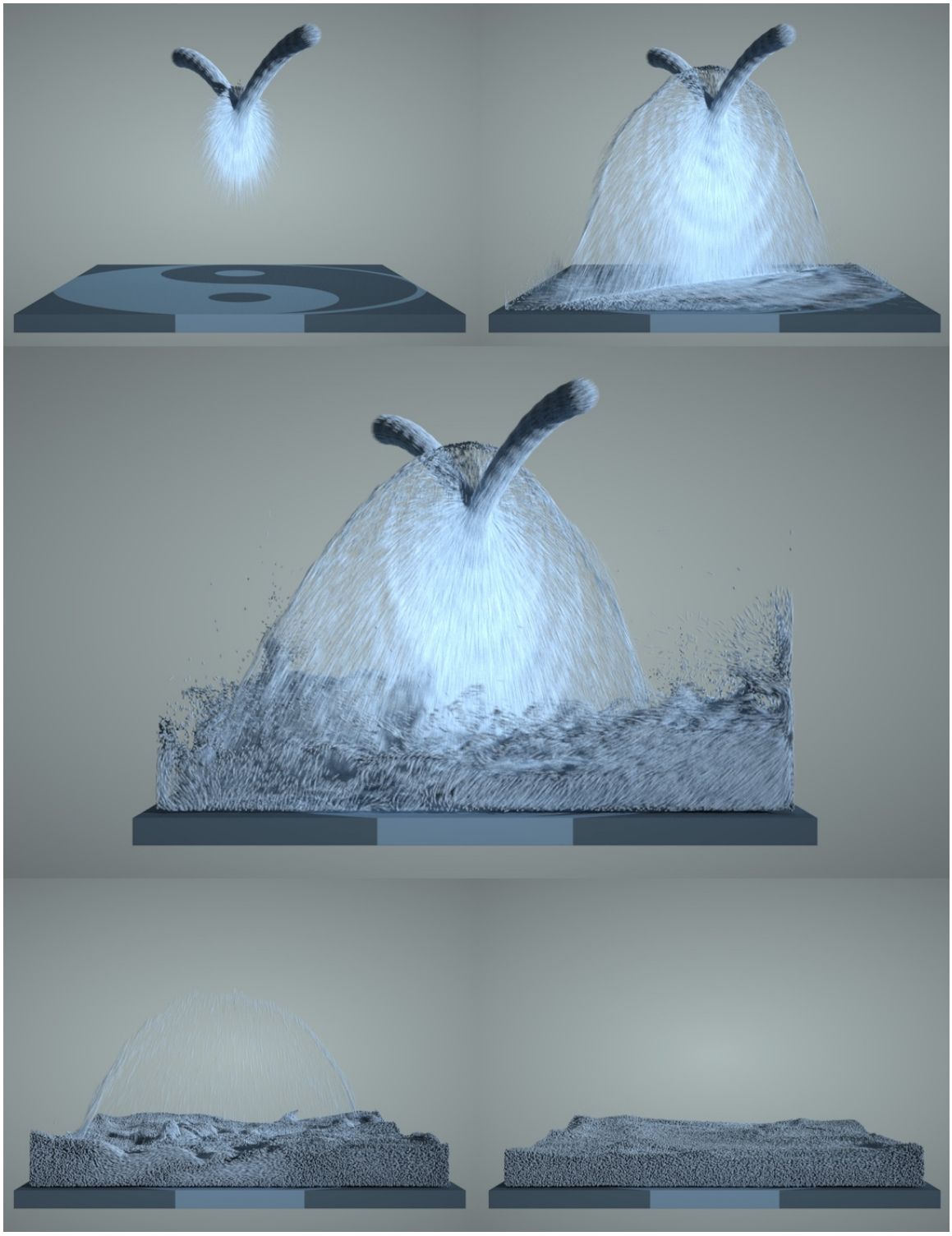


Figure 2-16: A fluid animation using MLS-MPM with up to 3 million particles.

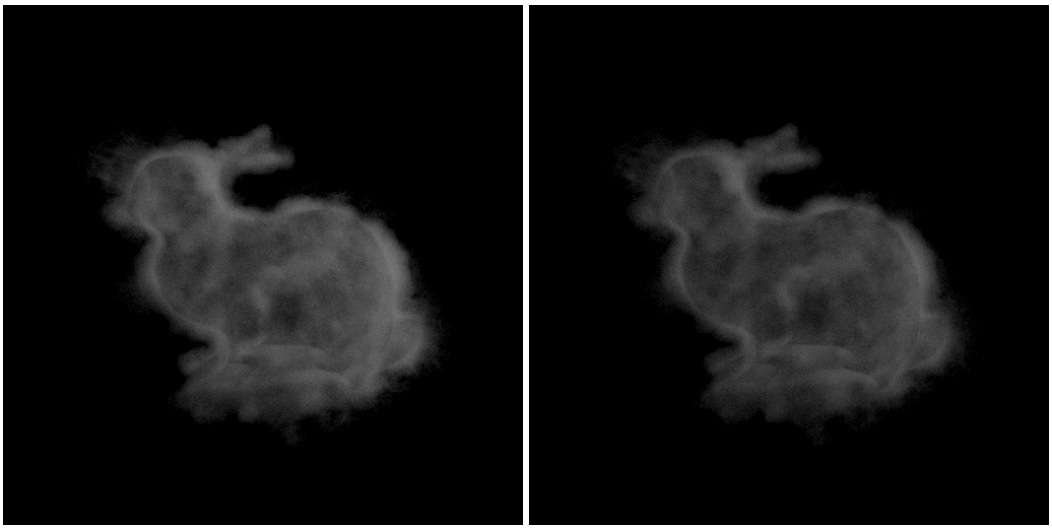


Figure 2-17: Smoke in the shape of a bunny rendered with our volumetric path tracer (left) and Tungsten's (right). Both rendered with 128 samples per pixel and a path length limit of 128.

G Code Samples

G .1 MLS-MPM with Comments

```
#include <taichi/lang.h>
#include <taichi/util.h>
#include <taichi/visual/gui.h>
#include <taichi/common/bit.h>
#include <taichi/system/profiler.h>
#include "svd.h"
TC_NAMESPACE_BEGIN
using namespace Tlang;
auto mpm_benchmark = []() {
  Program prog(Arch::gpu);
  // No accessing lowering in this specific application is needed,
     since
  // the scratchpad optimizations already significantly reduce data
     structure access
  prog.config.lower_access = false;
  // simulation constants
  constexpr int dim = 3, n = 256, grid_block_size = 4, n_particles =
     775196;
  const real dt = 1e-5_f * 256 / n, dx = 1.0_f / n, inv_dx = 1.0_f / dx
     ;
  auto particle_mass = 1.0_f, vol = 1.0_f, E = 1e4_f, nu = 0.3f;
  real mu = E / (2 * (1 + nu)), lambda = E * nu / ((1 + nu) * (1 - 2 *
     nu));
  // simulation precision
  auto f32 = DataType::f32;
  // global variables
  Vector particle_x(f32, dim), particle_v(f32, dim), grid_v(f32, dim);
  Matrix particle_F(f32, dim, dim), particle_C(f32, dim, dim);
  Global(grid_m, f32);
  Global(l, i32);
  Global(gravity_x, f32);
  // load input
```

```
int max_n_particles = 1024 * 1024;
std::vector<Vector3> p_x;
p_x.resize(n_particles);
std::vector<float> benchmark_particles;
auto f = fopen("dragon_particles.bin", "rb");
TC_ASSERT_INFO(f, "./dragon_particles.bin not found");
benchmark_particles.resize(n_particles * 3);
std::fread(benchmark_particles.data(), sizeof(float), n_particles *
    3, f);
std::fclose(f);
for (int i = 0; i < n_particles; i++) {
  for (int j = 0; j < dim; j++)
    p_x[i][j] = benchmark_particles[i * dim + j];
}
bool particle_SOA = false;
// layout function call, materialize the data structure
layout([&]() {
  auto i = Index(0), j = Index(1), k = Index(2), p = Index(3);
  SNode *fork;
  if (!particle_SOA)
    fork = &root.dynamic(p, max_n_particles);
  auto place = [&](Expr &expr) {
    if (particle_SOA) {
      root.dynamic(p, max_n_particles).place(expr);
    } else {
      fork->place(expr);
    }
  };
  for (int i = 0; i < dim; i++)
    for (int j = 0; j < dim; j++)
      place(particle_F(i, j));
  for (int i = 0; i < dim; i++)
    for (int j = 0; j < dim; j++)
      place(particle_C(i, j));
  for (int i = 0; i < dim; i++)
    place(particle_x(i));
```

```
  for (int i = 0; i < dim; i++)
    place(particle_v(i));
  TC_ASSERT(n % grid_block_size == 0);
  auto &block = root.dense({i, j, k}, n / grid_block_size).pointer();
  constexpr bool block_soa = true;
  if (block_soa) {
    block.dense({i, j, k}, grid_block_size).place(grid_v(0));
    block.dense({i, j, k}, grid_block_size).place(grid_v(1));
    block.dense({i, j, k}, grid_block_size).place(grid_v(2));
    block.dense({i, j, k}, grid_block_size).place(grid_m);
  } else {
    block.dense({i, j, k}, grid_block_size)
        .place(grid_v(0), grid_v(1), grid_v(2), grid_m);
  }
  block.dynamic(p, pow<dim>(grid_block_size) * 128).place(l);
  root.place(gravity_x);
});
// sort particle indices into their belonging block
Kernel(sort).def([&] {
  BlockDim(1024);
  For(particle_x(0), [&](Expr p) {
    // compute the block coordinates
    auto node_coord = floor(particle_x[p] * inv_dx - 0.5_f);
    // insert the particle index
    Append(l.parent(),
           (cast<int32>(node_coord(0)), cast<int32>(node_coord(1)),
            cast<int32>(node_coord(2))),
           p);
  });
});
// Particle to grid transfer
Kernel(p2g_sorted).def([&] {
  // GPU block dim
  BlockDim(128);
  // allocate scratch pads for the velocity and mass channels
  Cache(0, grid_v(0));
```

```
Cache(0, grid_v(1));
Cache(0, grid_v(2));
Cache(0, grid_m);
For(l, [&](Expr i, Expr j, Expr k, Expr p_ptr) {
  // for each particle, compute its momentum contribution and
     scatter to surrounding grid nodes
  auto p = Var(l[i, j, k, p_ptr]);
  auto x = Var(particle_x[p]), v = Var(particle_v[p]),
       C = Var(particle_C[p]);
  auto base_coord = floor(inv_dx * x - 0.5_f), fx = x * inv_dx -
     base_coord;
  Matrix F = Var(Matrix::identity(dim) + dt * C) * particle_F[p];
  particle_F[p] = F;
  Vector w[] = {Var(0.5_f * sqr(1.5_f - fx)), Var(0.75_f - sqr(fx -
      1.0_f)),
                Var(0.5_f * sqr(fx - 0.5_f))};
  auto svd = sifakis_svd(F);
  auto R = Var(std::get<0>(svd) * transposed(std::get<2>(svd)));
  auto sig = Var(std::get<1>(svd));
  auto J = Var(sig(0) * sig(1) * sig(2));
  auto cauchy = Var(2.0_f * mu * (F - R) * transposed(F) +
                    (Matrix::identity(3) * lambda) * (J - 1.0f) * J
                       );
  auto affine =
      Var(particle_mass * C - (4 * inv_dx * inv_dx * dt * vol) *
         cauchy);
  int low = 0, high = 1;
  // The AssumeInRange intrinsics tells the compiler how big the
     scratchpad should be
  auto base_coord_i =
      AssumeInRange(cast<int32>(base_coord(0)), i, low, high);
  auto base_coord_j =
      AssumeInRange(cast<int32>(base_coord(1)), j, low, high);
  auto base_coord_k =
      AssumeInRange(cast<int32>(base_coord(2)), k, low, high);
  for (int a = 0; a < 3; a++)
```

```
      for (int b = 0; b < 3; b++)
        for (int c = 0; c < 3; c++) {
          auto dpos = dx * (Vector({a, b, c}).cast_elements<float32
              >() - fx);
          auto weight = w[a](0) * w[b](1) * w[c](2);
          auto node = (base_coord_i + a, base_coord_j + b,
              base_coord_k + c);
          // Atomic adds for safe parallelism
          Atomic(grid_v[node]) +=
              weight * (particle_mass * v + affine * dpos);
          Atomic(grid_m[node]) += weight * particle_mass;
        }
  });
});
// grid operations
Kernel(grid_op).def([&]() {
  For(grid_m, [&](Expr i, Expr j, Expr k) {
    auto v = Var(grid_v[i, j, k]);
    auto m = Var(grid_m[i, j, k]);
    int bound = 8;
    // normalize momentum into velocity
    If(m > 0.0f, [&]() {
      auto inv_m = Var(1.0f / m);
      v *= inv_m;
      // apply gravity
      auto f = gravity_x[Expr(0)];
      v(1) += dt * (-1000_f + abs(f));
      v(0) += dt * f;
    });
    // boundary conditions
    v(0) = select(n - bound < i, min(v(0), Expr(0.0_f)), v(0));
    v(1) = select(n - bound < j, min(v(1), Expr(0.0_f)), v(1));
    v(2) = select(n - bound < k, min(v(2), Expr(0.0_f)), v(2));
    v(0) = select(i < bound, max(v(0), Expr(0.0_f)), v(0));
    v(2) = select(k < bound, max(v(2), Expr(0.0_f)), v(2));
    If(j < bound, [&] { v(1) = max(v(1), Expr(0.0_f)); });
```

```
    grid_v[i, j, k] = v;
  });
});
// grid to particle transfer
Kernel(g2p).def([&]() {
  // GPU block dim
  BlockDim(128);
  // allocate scratchpads for the velocity channels
  Cache(0, grid_v(0));
  Cache(0, grid_v(1));
  Cache(0, grid_v(2));
  For(l, [&](Expr i, Expr j, Expr k, Expr p_ptr) {
    auto p = Var(l[i, j, k, p_ptr]);
    auto x = Var(particle_x[p]), v = Var(Vector(dim)),
         C = Var(Matrix(dim, dim));
    for (int i = 0; i < dim; i++) {
      v(i) = Expr(0.0_f);
      for (int j = 0; j < dim; j++) {
        C(i, j) = Expr(0.0_f);
      }
    }
    auto base_coord = floor(inv_dx * x - 0.5_f);
    auto fx = x * inv_dx - base_coord;
    Vector w[] = {Var(0.5_f * sqr(1.5_f - fx)), Var(0.75_f - sqr(fx -
        1.0_f)),
                  Var(0.5_f * sqr(fx - 0.5_f))};
    int low = 0, high = 1;
    auto base_coord_i =
        AssumeInRange(cast<int32>(base_coord(0)), i, low, high);
    auto base_coord_j =
        AssumeInRange(cast<int32>(base_coord(1)), j, low, high);
    auto base_coord_k =
        AssumeInRange(cast<int32>(base_coord(2)), k, low, high);
    for (int p = 0; p < 3; p++)
      for (int q = 0; q < 3; q++)
        for (int r = 0; r < 3; r++) {
```

```
            auto dpos = Vector({p, q, r}).cast_elements<float32>() - fx
                ;
            auto weight = w[p](0) * w[q](1) * w[r](2);
            auto wv =
                weight *
                grid_v[base_coord_i + p, base_coord_j + q, base_coord_k
                    + r];
            v += wv;
            C += outer_product(wv, dpos);
          }
      particle_C[p] = (4 * inv_dx) * C;
      particle_v[p] = v;
      particle_x[p] = x + dt * v;
    });
});
// initial particle reordering
auto block_id = [&](Vector3 x) {
  auto xi = (x * inv_dx - Vector3(0.5f)).floor().template cast<int>()
       /
            Vector3i(grid_block_size);
  return xi.x * pow<2>(n / grid_block_size) + xi.y * n /
      grid_block_size +
          xi.z;
};
std::sort(p_x.begin(), p_x.end(),
          [&](Vector3 a, Vector3 b) { return block_id(a) < block_id(b
              ); });
for (int i = 0; i < n_particles; i++) {
  for (int d = 0; d < dim; d++) {
    particle_x(d).val<float32>(i) = p_x[i][d];
  }
  particle_v(0).val<float32>(i) = 0._f;
  particle_v(1).val<float32>(i) = -3.0_f;
  particle_v(2).val<float32>(i) = 0._f;
  for (int p = 0; p < dim; p++)
    for (int q = 0; q < dim; q++)
```

```
        particle_F(p, q).val<float32>(i) = (p == q);
  }
  // main simulation loop
  auto simulate_frame = [&]() {
    grid_m.parent().parent().snode()->clear_data_and_deactivate();
    auto t = Time::get_time();
    for (int f = 0; f < 200; f++) {
      grid_m.parent().parent().snode()->clear_data();
      sort();
      p2g_sorted();
      grid_op();
      g2p();
    }
    prog.profiler_print();
    auto ms_per_substep = (Time::get_time() - t) / 200 * 1000;
    TC_P(ms_per_substep);
  };
  // Visualization omitted...
};
TC_REGISTER_TASK(mpm_benchmark);
TC_NAMESPACE_END
```

See Hu et al.'s article [48] for the derivation of the algorithm.

G .2 FEM Linear Elasticity Kernel

This kernel is the implementation of Equation (1) of Liu et al.'s work [79].

```
Kernel(compute_Ap).def([&] {
  For(Ap(0), [&](Expr i, Expr j, Expr k) {
    auto cell_coord = Var(Vector({i, j, k}));
    auto Ku_tmp = Var(Vector(dim));
    Ku_tmp = Vector({0.0f, 0.0f, 0.0f});
    // Unrolled for loop
    for (int cell = 0; cell < pow<dim>(2); cell++) {
      auto cell_offset =
          Var(Vector({-(cell / 4), -(cell / 2 % 2), -(cell % 2)}));
```

```
      auto cell_lambda = lambda[cell_coord + cell_offset];
      auto cell_mu = mu[cell_coord + cell_offset];
      // Unrolled for loop
      for (int node = 0; node < pow<dim>(2); node++) {
        auto node_offset = Var(Vector({node / 4, node / 2 % 2, node %
            2}));
        // Unrolled for loop
        for (int u = 0; u < dim; u++)
          // Unrolled for loop
          for (int v = 0; v < dim; v++)
            Ku_tmp(u) += (cell_lambda * K_la[cell][node][u][v] +
                          cell_mu * K_mu[cell][node][u][v]) *
                         p[cell_coord + cell_offset + node_offset](v);
      }
    }
  });
});
```

G .3 MGPCG Program

Please check out `examples/cpp/mgpcg.cpp`.

G .4 CNN Kernel

```
  Kernel(forward).def([&] {
    if (opt && gpu) {
      if (cache_l1)
        CacheL1(weights);
    }
    if (!gpu) {
      Parallelize(8);
      Vectorize(block_size);
    } else {
      BlockDim(256);
    }
```

```
  For(layer2, [&](Expr i, Expr j, Expr k, Expr c_out) {
    auto sum = Var(0.0f);
    for (int c_in = 0; c_in < num_ch1; c_in++) {
      for (int dx = -1; dx < 2; dx++) {
        for (int dy = -1; dy < 2; dy++) {
          for (int dz = -1; dz < 2; dz++) {
            auto weight = weights[Expr(dx + 1), Expr(dy + 1), Expr(dz
                + 1),
                                  c_in * num_ch2 + c_out];
            sum += weight * layer1[i + dx, j + dy, k + dz, c_in];
          }
        }
      }
    }
    layer2[i, j, k, c_out] = sum;
  });
});
```

G .5 Volume Renderer

The code is at `examples/cpp/volume_renderer.cpp`.

Chapter 3

Making Taichi Differentiable

In this chapter we present DiffTaichi, a differentiable programming extension to Taichi, tailored for building high-performance differentiable physical simulators. Based on the imperative programming language Taichi, DiffTaichi generates gradients of simulation steps using source code transformations that preserve arithmetic intensity (i.e., number of floating-point operations per byte fetched) and parallelism. A light-weight tape is used to record the whole simulation program structure and replay the gradient kernels in a reversed order, for end-to-end back-propagation. We demonstrate the performance and productivity of DiffTaichi in gradient-based learning and optimization tasks on 10 different physical simulators. For example, a differentiable elastic object simulator written in DiffTaichi is $4.2\times$ shorter than the hand-engineered CUDA version yet runs as fast, and is $188\times$ faster than the TensorFlow implementation. Using our differentiable programs, neural network controllers are typically optimized within only tens of iterations.

We also replace the origin C++ frontend with a Python frontend for higher productivity.

1 Introduction

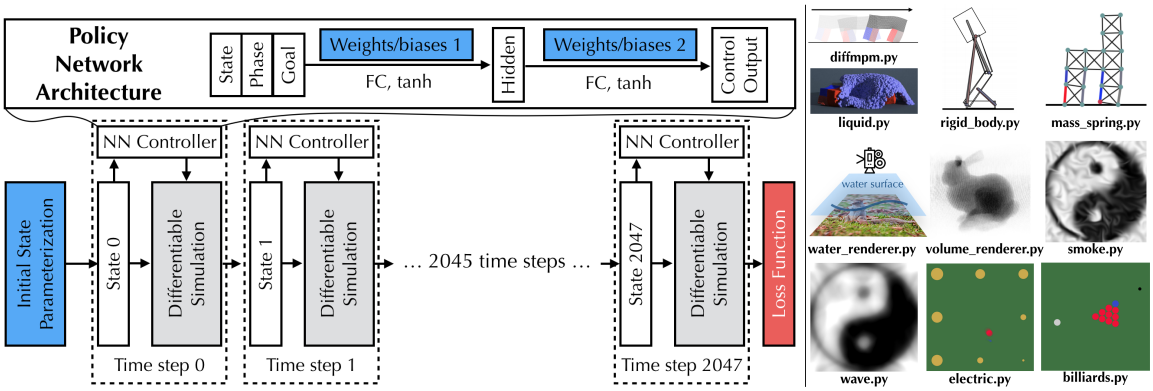


Figure 3-1: **Left:** DiffTaichi allows us to seamlessly integrate a neural network (NN) controller and a physical simulation module, and update the weights of the controller or the initial state parameterization (blue). Our simulations typically have $512 \sim 2048$ time steps, and each time step has up to one thousand parallel operations. **Right:** 10 differentiable simulators built with DiffTaichi.

Differentiable physical simulators are effective components in machine learning systems. For example, [27] and [51] have shown that controller optimization with differentiable simulators converges one to four orders of magnitude faster than model-free reinforcement learning algorithms. The presence of differentiable physical simulators in the inner loop of these applications makes their performance vitally important. Unfortunately, using existing tools it is difficult to implement these simulators with high performance.

We present *DiffTaichi*, a new differentiable programming language for high performance physical simulations on both CPU and GPU. It is based on the Taichi programming language [49]. The DiffTaichi automatic differentiation system is designed to suit key language features required by physical simulation, yet often missing in existing differentiable programming tools, as detailed below:

Megakernels DiffTaichi uses a "megakernel" approach, allowing the programmer to naturally fuse multiple stages of computation into a single kernel, which is later differentiated using source code transformations and just-in-time compilation. Compared to the linear algebra operators in TensorFlow [2] and PyTorch [96],

DiffTaichi kernels have higher arithmetic intensity and are therefore more efficient for physical simulation tasks.

Imperative Parallel Programming In contrast to functional array programming languages that are popular in modern deep learning [11, 2, 77], most traditional physical simulation programs are written in imperative languages such as Fortran and C++. DiffTaichi likewise adopts an imperative approach. The language provides parallel loops and control flows (such as "if" statements), which are widely used constructs in physical simulations: they simplify common tasks such as handling collisions, evaluating boundary conditions, and building iterative solvers. Using an imperative style makes it easier to port existing physical simulation code to DiffTaichi.

Flexible Indexing Existing parallel differentiable programming systems provide element-wise operations on arrays of the same shape, e.g. `c[i, j] = a[i, j] + b[i, j]`. However, many physical simulation operations, such as numerical stencils and particle-grid interactions are not element-wise. Common simulation patterns such as `y[p[i] * 2, j] = x[q[i + j]]` can only be expressed with unintuitive `scatter`/`gather` operations in these existing systems, which are not only inefficient but also hard to develop and maintain. On the other hand, in DiffTaichi, the programmer directly manipulates array elements via arbitrary indexing, thus allowing partial updates of global arrays and making these common simulation patterns naturally expressible. The explicit indexing syntax also makes it easy for the compiler to perform access optimizations [49].

The three requirements motivated us to design a tailored two-scale automatic differentiation system, which makes DiffTaichi especially suitable for developing complex and high-performance differentiable physical simulators, possibly with neural network controllers (Fig. 3-1, left). Using DiffTaichi, we are able to quickly implement and automatically differentiate 10 physical simulators[1], covering rigid

[1]Our language, compiler, and simulator code is open-source. All the results in this work can be reproduced by a single Python script. Visual results in this work are presented in the supplemental

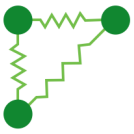
bodies, deformable objects, and fluids (Fig. 3-1, right). A comprehensive comparison between DiffTaichiand other differentiable programming tools is in Appendix A .

2 Recap: The Taichi Programming Language

DiffTaichi is based on the Taichi programming language [49]. Taichi is an imperative programming language embedded in C++14. It delivers both high performance and high productivity on modern hardware. The key design that distinguishes Taichi from other imperative programming languages such as C++/CUDA is the decoupling of computation from data structures. This allows programmers to easily switch between different data layouts and access data structures with indices (i.e. `x[i, j, k]`), as if they are normal dense arrays, regardless of the underlying layout. The Taichi compiler then takes both the data structure and algorithm information to apply performance optimizations. Taichi provides “parallel-for" loops as a first-class construct. These designs make Taichi especially suitable for writing high-performance physical simulators. For more details, readers are referred to [49].

The DiffTaichi language frontend is embedded in Python, and a Python AST transformer compiles DiffTaichi code to Taichi intermediate representation (IR). Unlike Python, the DiffTaichi language is compiled, statically-typed, parallel, and differentiable. We extend the Taichi compiler to further compile and automatically differentiate the generated Taichi IR into forward and backward executables.

We demonstrate the language using a mass-spring simulator, with three springs and three mass points, as shown right. In this section we introduce the forward simulator using the DiffTaichi frontend of Taichi, which is an easier-to-use wrapper of the Taichi C++14 frontend.



video.

Allocating Global Variables Firstly we allocate a set of global tensors to store the simulation state. These tensors include a scalar `loss` of type `float32`, 2D tensors `x, v, force` of size `steps×n_springs` and type `float32x2`, and 1D arrays of size `n_spring` for spring properties: `spring_anchor_a (int32), spring_anchor_b (int32), spring_length (float32).`

Defining Kernels A mass-spring system is modeled by Hooke's law $\mathbf{F} = k(\|\mathbf{x}_a - \mathbf{x}_b\|_2 - l_0)\frac{\mathbf{x}_a - \mathbf{x}_b}{\|\mathbf{x}_a - \mathbf{x}_b\|_2}$ where k is the spring stiffness, $\mathbf{F}$ is spring force, $\mathbf{x}_a$ and $\mathbf{x}_b$ are the positions of two mass points, and l_0 is the rest length. The following kernel loops over all the springs and scatters forces to mass points:

```
@ti.kernel
def apply_spring_force(t: ti.i32):
  # Kernels can have parameters. Here t is a parameter with type int32.
  for i in range(n_springs): # A parallel for, preferably on GPU
    a, b = spring_anchor_a[i], spring_anchor_b[i]
    x_a, x_b = x[t - 1, a], x[t - 1, b]
    dist = x_a - x_b
    length = dist.norm() + 1e-4
    F = (length - spring_length[i]) * spring_stiffness * dist / length
    # Apply spring impulses to mass points.
    force[t, a] += -F # "+=" is atomic by default
    force[t, b] +=  F
```

For each particle i, we use semi-implicit Euler time integration with damping: $\mathbf{v}_{t,i} = e^{-\Delta t\alpha}\mathbf{v}_{t-1,i} + \frac{\Delta t}{m_i}\mathbf{F}_{t,i}, \mathbf{x}_{t,i} = \mathbf{x}_{t-1,i} + \Delta t\mathbf{v}_{t,i}$, where $\mathbf{v}_{t,i}, \mathbf{x}_{t,i}, m_i$ are the velocity, position and mass of particle i at time step t, respectively. α is a damping factor. The kernel is as follows:

```
@ti.kernel
def time_integrate(t: ti.i32):
  for i in range(n_objects):
    s = math.exp(-dt * damping) # Compile-time evaluation since dt and damping are
        constants
    v[t, i] = s * v[t - 1, i] + dt * force[t, i] / mass # mass = 1 in this example
    x[t, i] = x[t - 1, i] + dt * v[t, i]
```

Assembling the Forward Simulator With these components, we define the forward time integration:

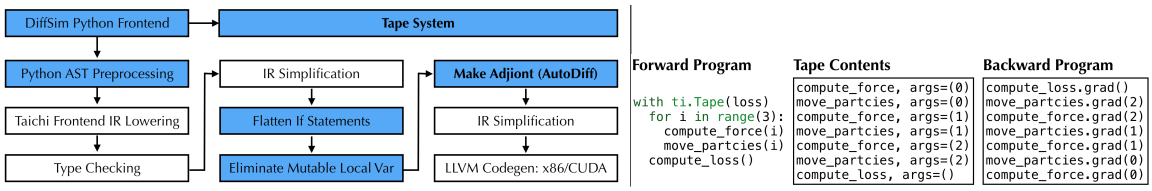


Figure 3-2: **Left:** The DiffTaichi system. We reuse some infrastructure (white boxes) from Taichi, while the blue boxes are our extensions for differentiable programming. **Right:** The tape records kernel launches and replays the gradient kernels in reverse order during backpropagation.

```
def forward():
  for t in range(1, steps):
    apply_spring_force(t)
    time_integrate(t)
```

3 Automatically Differentiating Physical Simulators in Taichi

The main goal of DiffTaichi's automatic differentiation (AD) system is to generate gradient simulators automatically with **minimal code changes** to the traditional forward simulators.

Design Decision Source Code Transformation (SCT) [38] and Tracing [128] are common choices when designing AD systems. In our setting, using SCT to differentiate a whole simulator with thousands of time steps, results in high performance yet poor flexibility and long compilation time. On the other hand, naively adopting tracing provides flexibility yet poor performance, since the "megakernel" structure is not preserved during backpropagation. To get both performance and flexibility, we developed a **two-scale** automatic differentiation system (Figure 3-2): we use SCT for differentiating within kernels, and use a light-weight tape that only stores function pointers and arguments for end-to-end simulation differentiation. The global tensors are natural checkpoints for gradient evaluation.

Assumption Unlike functional programming languages where immutable output buffers are generated, imperative programming allows programmers to freely modify global tensors. To make automatic differentiation well-defined under this setting, we make the following assumption on imperative kernels:

> **Global Data Access Rules:**
> 1) If a global tensor element is written more than once, then starting from the second write, the write must come in the form of an atomic add ("accumulation").
> 2) No read accesses happen to a global tensor element, until its accumulation is done.

In forward simulators, programmers may make subtle changes to satisfy the rules. For instance, in the mass-spring simulation example, we record the whole history of `x` and `v`, instead of keeping only the latest values. The memory consumption issues caused by this can be alleviated via checkpointing, as discussed later in Appendix D .

With these assumptions, kernels will not overwrite the outputs of each other, and the goal of AD is clear: given a primal kernel f that takes as input $X_1, X_2, \ldots, X_n$ and outputs (or accumulates to) $Y_1, Y_2, \ldots, Y_m$, the generated gradient (adjoint) kernel f^* should take as input $X_1, X_2, \ldots, X_n$ and $Y_1^*, Y_2^*, \ldots, Y_m^*$ and accumulate gradient contributions to $X_1^*, X_2^*, \ldots, X_m^*$, where each X_i^* is an **adjoint** of X_i, i.e. $\partial(\text{loss})/\partial X_i$.

Storage Control of Adjoint Tensors Users can specify the storage of adjoint tensors using the Taichi data structure description language [49], as if they are primal tensors. We also provide `ti.root.lazy_grad()` to automatically place the adjoint tensors following the layout of their primals.

3 .1 Local AD: Differentiating Taichi Kernels using Source Code Transforms

A typical Taichi kernel consists of multiple levels of for loops and a body block. To make later AD easier, we introduce two basic code transforms to simplify the loop body, as detailed below.

```
int a = 0;                  // flatten branching        // eliminate mutable var
if (b > 0) { a = b;}        int a = 0;
  else     { a = 2b;}       a = select(b > 0, b, 2b);   ssa1 = select(b>0, b, 2b);
a = a + 1;                  a = a + 1                   ssa2 = ssa1 + 1
return a;                   return a;                   return ssa2;
```

Figure 3-3: Simple IR preprocessing before running the AD source code transform (left to right). Demonstrated in `C++`. The actual Taichi IR is often more complex. Containing loops are ignored.

Flatten Branching In physical simulation branches are common, e.g., when implementing boundary conditions and collisions. To simplify the reverse-mode AD pass, we first replace "if" statements with ternary operators `select(cond, value_if_true, value_if_false)`, whose gradients are clearly defined (Fig. 3-3, middle). This is a common transformation in program vectorization (e.g. [63, 99]).

Eliminate Mutable Local Variables After removing branching, we end up with straight-line loop bodies. To further simplify the IR and make the procedure truly single-assignment, we apply a series of local variable store forwarding transforms, until the mutable local variables can be fully eliminated (Fig. 3-3, right).

After these two custom IR simplification transforms, DiffTaichi only has to differentiate the straight-line code without mutable variables, which it achieves with reverse-mode AD, using a standard source code transformation [38]. More details on this transform are in Appendix B .

Loops Most loops in physical simulation are parallel loops, and during AD we preserve the parallel loop structures. For loops that are not explicitly marked as

parallel, we reverse the loop order during AD transforms. We do not support loops that carry a mutating *local* variable since that would require a complex and costly run-time stack to maintain the history of local variables. Instead, users are instructed to employ *global* variables that satisfy the global data access rules.

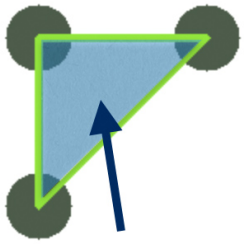
Parallelism and Thread Safety For forward simulation, we inherit the "parallel-for" construct from Taichi to map each loop iteration onto CPU/GPU threads. Programmers use atomic operations for thread safety. Our system can automatically differentiate these atomic operations. Gradient contributions in backward kernels are accumulated to the adjoint tensors via atomic adds.

3 .2 Global AD: End-to-end Backpropagation using A Light-Weight Tape

We construct a tape (Fig. 3-2, right) of the kernel execution so that gradient kernels can be replayed in a reversed order. The tape is very light-weight: since the intermediate results are stored in global tensors, during forward simulation the tape only records kernel names and the (scalar) input parameters, unlike other differentiable functional array systems where all the intermediate buffers have to be recorded by the tape. Whenever a DiffTaichi kernel is launched, we append the kernel function pointer and parameters to the tape. When evaluating gradients, we traverse the reversed tape, and invoke the gradient kernels with the recorded parameters. Note that DiffTaichi AD is evaluating gradients with respect to input global tensors instead of the input parameters.

Learning/Optimization with Gradients Now we revisit the mass-spring example and make it differentiable for optimization. Suppose the goal is to optimize the rest lengths of the springs so that the triangle area formed by the three springs becomes 0.2 at the end of the simulation. We first define the loss function:

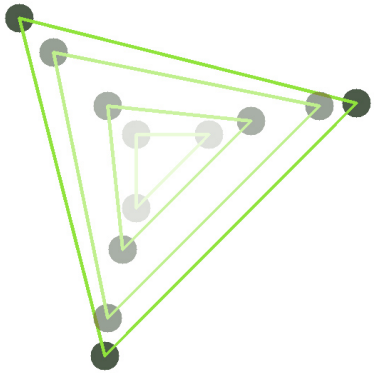
```
@ti.kernel
def compute_loss(t: ti.i32):
  x01 = x[t, 0] - x[t, 1]
  x02 = x[t, 0] - x[t, 2]
  # Triangle area from cross product
  area = ti.abs(0.5 * (x01[0]*x02[1] - x01[1]*x02[0]))
  target_area = 0.2
  loss[None] = ti.sqr(area - target_area)
  # Everything in Taichi is a tensor.
  # "loss" is a scalar (0-D tensor), thereby indexed with [None].
```

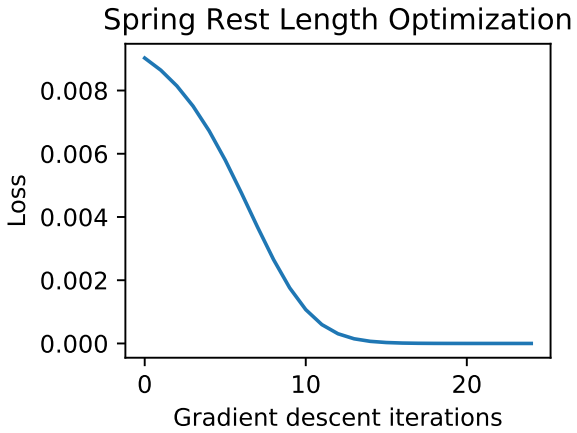


Goal:
Adjust the spring rest lengths, so that this area=0.2 after 1024 time steps (initial area=0.005)

The programmer uses `ti.Tape` to memorize forward kernel launches. It automatically replays the gradients of these kernels in reverse for backpropagation. Initially the springs have lengths $[0.1, 0.1, 0.14]$, and after optimization the rest lengths are $[0.600, 0.600, 0.529]$. This means the springs will expand the triangle according to Hooke's law and form a larger triangle:

```
def main():
  for iter in range(200):
    with ti.Tape(loss):
      forward()
      compute_loss(steps - 1)
    print('Iter=', iter)
    print('Loss=',loss[None])
    # Gradient descent
    for i in range(n_springs):
      spring_length[i] -=
        lr * spring_length.grad[i]
```





Complex Kernels Sometimes the user may want to override the gradients provided by the compiler. For example, when differentiating a 3D singular value decomposition done with an iterative solver, it is better to use a manually engineered SVD derivative subroutine for better stability. We provide two more decorators `ti.complex_kernel` and `ti.complex_kernel_grad` to overwrite the default automatic differentiation, as detailed in Appendix C . Apart from custom gradients, complex kernels can also be used to implement checkpointing, as detailed in Appendix D .

4 Evaluation

We evaluate DiffTaichi on 10 different physical simulators covering large-scale continuum and small-scale rigid body simulations. All results can be reproduced with the provided script. The dynamic/optimization processes are visualized in the

supplemental video. In this section we focus our discussions on three simulators. More details on the simulators are in Appendix E .

4 .1 Differentiable Continuum Mechanics for Elastic Objects [diffmpm]

First, we build a differentiable continuum simulation for soft robotics applications. The physical system is governed by momentum and mass conservation, i.e. $\rho\frac{D\mathbf{v}}{Dt} = \nabla\cdot\sigma + \rho\mathbf{g}$, $\frac{D\rho}{Dt} + \rho\nabla\cdot\mathbf{v} = 0$. We follow ChainQueen's implementation [51] and use the moving least squares material point method [48] to simulate the system. We were able to easily translate the original CUDA simulator into DiffTaichi syntax. Using this simulator and an open-loop controller, we can easily train a soft robot to move forward (Fig. 3-1, `diffmpm`).

Performance and Productivity Compared with manual gradient implementations in [51], getting gradients in DiffTaichi is effortless. As a result, the DiffTaichi implementation is $4.2\times$ shorter in terms of lines of code, and runs almost as fast; compared with TensorFlow, DiffTaichi code is $1.7\times$ shorter and $188\times$ faster (Table 3.1). The Tensorflow implementation is verbose due to the heavy use of `tf.gather_nd/scatter_nd` and array transposing and broadcasting.

Table 3.1: `diffmpm` performance comparison on an NVIDIA GTX 1080 Ti GPU. We benchmark in 2D using 6.4K particles. For the lines of code, we only include the essential implementation, excluding boilerplate code.

Approach	Forward Time	Backward Time	Total Time	# Lines of Code
TensorFlow	13.20 ms	35.70 ms	48.90 ms (188.×)	190
CUDA	0.10 ms	0.14 ms	0.24 ms (0.92×)	460
DiffTaichi	0.11 ms	0.15 ms	0.26 ms (1.00×)	110

Table 3.2: `smoke` benchmark against Autograd, PyTorch, and JAX. We used a 110×110 grid and 100 time steps, each with 6 Jacobi pressure projections. . Note that the Autograd program uses float64 precision, which approximately doubles the run time.

Approach	Forward Time	Backward Time	Total Time	# Essential LoC
PyTorch (CPU, f32)	405 ms	328 ms	733 ms (13.8×)	74
PyTorch (GPU, f32)	254 ms	457 ms	711 ms (13.4×)	74
Autograd (CPU, f64)	307 ms	1197 ms	1504 ms (28.4×)	51
JAX (GPU, f32)	24 ms	75 ms	99 ms (1.9×)	90
DiffTaichi (CPU, f32)	66 ms	132 ms	198 ms (3.7×)	75
DiffTaichi (GPU, f32)	24 ms	29 ms	53 ms (1.0×)	75

4 .2 Differentiable Incompressible Fluid Simulator `[smoke]`

We implemented a smoke simulator (Fig. 3-1, `smoke`) with semi-Lagrangian advection [111] and implicit pressure projection, following the example in Autograd [83]. Using gradient descent optimization on the initial velocity field, we are able to find a velocity field that changes the pattern of the fluid to a target image (Fig. 3-7a in Appendix). We compare the performance of our system against PyTorch, Autograd, and JAX in Table 3.2. Note that as an example from the Autograd library, this grid-based simulator is intentionally simplified to suit traditional array-based programs. For example, a periodic boundary condition is used so that Autograd can represent it using `numpy.roll`, without any branching. Still, Taichi delivers higher performance than these array-based systems. The whole program takes 10 seconds to run in DiffTaichi on a GPU, and 2 seconds are spent on JIT. JAX JIT compilation takes 2 minutes.

4 .3 Differentiable rigid body simulators `[rigid_body]`

We built an impulse-based [17] differentiable rigid body simulator (Fig. 3-1, `rigid_body`) for optimizing robot controllers. This simulator supports rigid body collision and friction, spring forces, joints, and actuation. The simulation is end-to-end differentiable except for a countable number of discontinuities. Interestingly, although the forward simulator works well, naively differentiating it with DiffTaichi leads to completely misleading gradients, due to the rigid body collisions. We discuss

the cause and solution of this issue below.

Improving collision gradients Consider the rigid ball example in Fig. 3-4 (left), where a rigid ball collides with a friction-less ground. Gravity is ignored, and due to conservation of kinetic energy the ball keeps a constant speed even after this elastic collision.

In the forward simulation, using a small Δt often leads to a reasonable result, as done in many physics simulators. Lowering the initial ball height will increase the final ball height, since there is less distance to travel before the ball hits the ground and more after (see the loss curves in Fig.3-4, middle right). However, using a naive time integrator, no matter how small Δt is, the evaluated gradient of final height w.r.t. initial height will be 1 instead of -1. This counter-intuitive behavior is due to the fact that time discretization itself is not differentiated by the compiler. Fig. 3-4 explains this effect in greater detail.

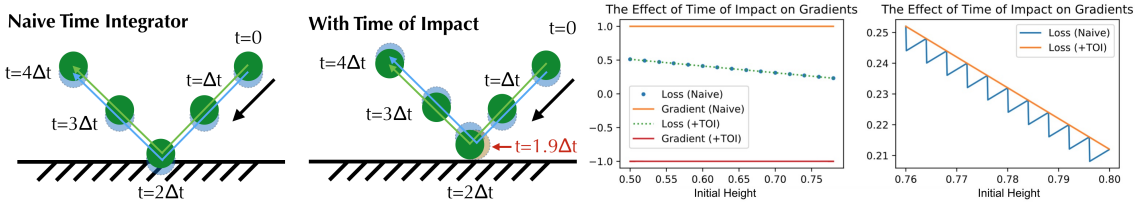


Figure 3-4: How gradients can go wrong with naive time integrators. For clarity we use a large Δt here. **Left:** Since collision detection only happens at multiples of Δt ($2\Delta t$ in this case), lowering the initial position of the ball (light blue) leads to a lowered final position. **Middle Left:** By improving the time integrator to support continuous time of impact (TOI), collisions can be detected at any time, e.g. $1.9\Delta t$ (light red). Now the blue ball ends up higher than the green one. **Middle Right:** Although the two time integration techniques lead to almost identical forward results (in practice Δt is small), the naive time integrator gives an incorrect gradient of 1, but adding TOI yields the correct gradient. Please see our supplemental video for a better demonstration. **Right:** When zooming in, the loss of the naive integrator is decreasing, and the saw-tooth pattern explains the positive gradients.

We propose a simple solution of adding continuous collision resolution (see, for example, [105]), which considers precise time of impact (TOI), to the forward program (Fig. 3-4, middle left). Although it barely improves the forward simulation (Fig. 3-4, middle right), the gradient will be corrected effectively (Fig. 3-4, right).

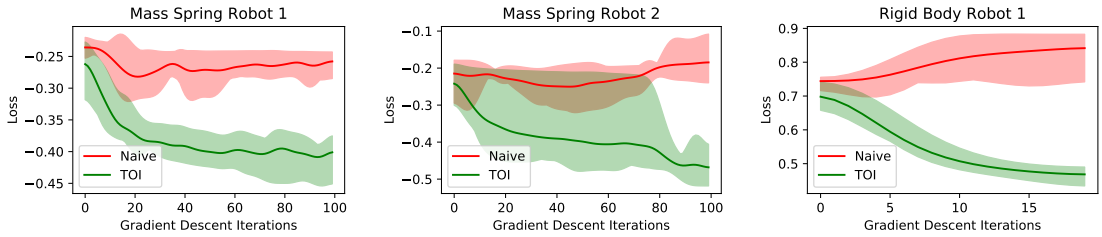


Figure 3-5: Adding TOI greatly improves gradient and optimization quality. Each experiment is repeated five times.

The details of continuous collision detection are in Appendix F . In real-world simulators, we find the TOI technique leads to significant improvement in gradient quality in controller optimization tasks (Fig. 3-5). Having TOI or not barely affects forward simulation: in the supplemental video, we show that a robot controller optimized in a simulator with TOI, actually works well in a simulator without TOI.

The takeaway is, *differentiating physical simulators does not always yield useful gradients of the physical system being simulated, even if the simulator does forward simulation well*. In Appendix G , we discuss some additional gradient issues we have encountered.

5 Related Work

Differentiable programming The recent rise of deep learning has motivated the development of differentiable programming libraries for deep NNs, most notably auto-differentiation frameworks such as Theano [11], TensorFlow [2] and PyTorch [96]. However, physical simulation requires complex and customizable operations due to the intrinsic computational irregularity. Using the aforementioned frameworks, programmers have to compose these coarse-grained basic operations into desired complex operations. Doing so often leads to unsatisfactory performance.

Earlier work on automatic differentiation focuses on transforming existing scalar code to obtain derivatives (e.g. [119], [40], [97]). A recent trend has emerged for modern programming languages to support differentiable function transforma-

tions through annotation (e.g. [54], [127]). These frameworks enable differentiating general programming languages, yet they provide limited parallelism.

Differentiable array programming languages such as Halide [104, 77], Autograd [83], JAX [14], and Enoki [58] operate on arrays instead of scalars to utilize parallelism. Instead of operating on arrays that are immutable, DiffTaichi uses an imperative style with flexible indexing to make porting existing physical simulation algorithms easier.

Differentiable Physical Simulators Building differentiable simulators for robotics and machine learning has recently increased in popularity. Without differentiable programming, [10], [18] and [87] used NNs to approximate the physical process and used the NN gradients as the approximate simulation gradients. [29] and [28] used Theano and PyTorch respectively to build differentiable rigid body simulators. [108] differentiates position-based fluid using custom CUDA kernels. [100] used a differentiable rigid body simulator for manipulating physically based animations. The ChainQueen differentiable elastic object simulator [51] implements forward and gradient versions of continuum mechanics in hand-written CUDA kernels, leading to performance that is two orders of magnitude higher than a pure TensorFlow implementation. [78] built a differentiable cloth simulator for material estimation and motion control. The deep learning community also often incorporates differentiable rendering operations (OpenDR [81], N3MR [64], redner [76], Mitsuba 2 [94]) to learn from 3D scenes.

6 Conclusion

We have presented DiffTaichi, a differentiable programming language designed specifically for building high-performance differentiable physical simulators. Motivated by the need for supporting megakernels, imperative programming, and flexible indexing, we developed a tailored two-scale automatic differentiation system. We used DiffTaichi to build 10 simulators and integrated them into deep neu-

ral networks, which proved the performance and productivity of DiffTaichi over existing systems. We hope our programming language can greatly lower the barrier of future research on differentiable physical simulation in the machine learning and robotics communities.

Appendix for chapter 2

A Comparison with Existing AutoDiff Systems

Workload differences between deep learning and differentiable physical simulation Existing differentiable programming tools for deep learning are typically centered around large data blobs. For example, in AlexNet, the second convolution layer has size $27 \times 27 \times 128 \times 128$. These tools usually provide users with both low-level operations such as tensor add and mul, and high-level operations such as convolution. The bottleneck of typical deep-learning-based computer vision tasks are convolutions, so the provided high-level operations, with very high arithmetic intensity[2], can fully exploit hardware capability. However, the provided operations are "atoms" of these differentiable programming tools, and cannot be further customized. Users often have to use low-level operations to compose their desired high-level operations. This introduces a lot of temporary buffers, and potentially excessive GPU kernel launches. As shown in [51], a pure TensorFlow implementation of a complex physical simulator is $132\times$ slower than a CUDA implementation, due to excessive GPU kernel launches and the lack of producer-consumer locality[3].

The table below compares DiffTaichi with existing tools for build differentiable physical simulators.

[2]FLOPs per byte loaded from/stored to main memory.

[3]The CUDA kernels in [51] have much higher arithmetic intensity compared to the TensorFlow computational graph system. In other words, when implementing in CUDA immediate results are cached in registers, while in TensorFlow they are "cached" in main memory.

Table 3.3: Comparisons between DiffTaichi and other differentiable programming tools. **Note that this table only discusses features related to differentiable physical simulation**, and the other tools may not have been designed for this purpose. For example, PyTorch and TensorFlow are designed for classical deep learning tasks and have proven successful in their target domains. Also note that the XLA backend of TensorFlow and JIT feature of PyTorch allow them to fuse operators to some extent, but for simulation we want complete operator fusion within megakernels. "Swift" AD [127] is partially implemented as of November 2019. "Julia" refers to [54].

Feature	DiffTaichi	PyTorch	TensorFlow	Enoki	JAX	Halide	Julia	Swift
GPU Megakernels	✓			✓	✓	✓		
Imperative Scheme	✓			✓			✓	✓
Parallelism	✓	✓	✓	✓	✓	✓		
Flexible Indexing	✓					✓	✓	✓

B Differentating Straight-line Taichi Kernels using Source Code Transform

Primal and adjoint kernels Recall that in DiffTaichi, (primal) kernels are operators that take as input multiple tensors (e.g., X, Y) and output another set of tensors. Mathematically, kernel f has the form

$$f(X_0, X_1, .., X_n) = Y_0, Y_1, \ldots, Y_m.$$

Kernels usually execute uniform operations on these tensors. When it comes to differentiable programming, a loss function is defined on the final output tensors. The gradients of the loss function "L" with respect to each tensor are stored in *adjoint tensors* and computed via *adjoint kernels*.

The adjoint tensor of (primal) tensor X_{ijk} is denoted as X^*_{ijk}. Its entries are defined by $X^*_{ijk} = \partial L / \partial X_{ijk}$. At a high level, our automatic differentiation (AD)

system transforms a *primal* kernel into its *adjoint* form. Mathematically,

$$\textbf{(primal)}\ f(X_0, X_1, .., X_n) = Y_0, Y_1, \ldots, Y_m$$

$$\Big\downarrow \text{Reverse-Mode Automatic Differentiation}$$

$$\textbf{(adjoint)}\ f^*(X_0, X_1, .., X_n, Y_0^*, Y_1^*, \ldots, Y_m^*) = X_0^*, X_1^*, .., X_n^*.$$

Differentiating within kernels: The "make_adjoint" pass (reverse-mode AD) After the preprocessing passes, which flatten branching and eliminate mutable local variables, the "make_adjoint" pass transforms a forward evaluation (primal) kernel into its gradient accumulation ("adjoint") kernel. It takes straight-line code directly and operates on the hierarchical intermediate representation (IR) of Taichi[4] . Multiple outer for loops are allowed for the primal kernel. The Taichi compiler will distribute these parallel iterations onto CPU/GPU threads.

During the "make_adjoint" pass, for each SSA instruction, a local adjoint variable will be allocated for gradient contribution accumulation. The compiler will traverse the statements in reverse order, and accumulate the gradients to the corresponding adjoint local variable.

For example, a 1D array operation $y_i = \sin x_i^2$ has its IR representation as follows:

```
for i in range(0, 16):
  %1 = load x[i]
  %2 = mul %1, %1
  %3 = sin(%2)
  y[i] = %3
```

The above primal kernel will be transformed into the following adjoint kernel:

```
for i in range(0, 16):
  // adjoint variables
  %1adj = alloca 0.0
  %2adj = alloca 0.0
```

[4]Taichi uses a hierarchical static single assignment (SSA) intermediate representation (IR) as its internal program representation. The Taichi compiler applies multiple transform passes to lower and simplify the SSA IR in order to get high-performance binary code.

```
  %3adj = alloca 0.0
  // original forward computation
  %1 = load x[i]
  %2 = mul %1 %1
  %3 = sin(%2)
  // reverse accumulation
  %4 = load y_adj[i]
  %3adj += %4
  %5 = cos(%2)
  %2adj += %3adj * %5
  %1adj += 2 * %1 * %2adj
  atomic add x_adj[i], %1adj
```

Note that for clarity the transformed code is not strictly SSA here. The actual IR has more instructions. A following simplification pass will simplify redundant instructions generated by the AD pass.

C Complex Kernels

Here we demonstrated how to use complex kernels to override the automatic differentiation system. We use singular value decomposition (SVD) of 3×3 matrices ($\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$) as an example. Fast SVD solvers used in physical simulation are often iterative, yet directly evaluate the gradient of this iterative process is likely numerically unstable. Suppose we use [85] as the forward SVD solver, and use the method in [59] (Section 2.1.1.2) to evalute the gradients, the complex kernels are used as follows:

```python
# Do Singular Value Decomposition (SVD) on n matrices
@ti.kernel
def iterative_svd(num_iterations: ti.f32):
  for i in range(n):
    input = matrix_M[i]
    for iter in range(num_iterations):
      ... iteratively solve SVD using McAdams et al. 2011 ...
    matrix_U[i] = ...
    matrix_Sigma[i] = ...
    matrix_V[i] = ...

# A custom complex kernel that wraps the iterative SVD kernel
@ti.complex_kernel
```

```
def svd_forward(num_iterations):
  iterative_svd(num_iterations)

@ti.kernel
def svd_gradient():
  for i in range(n):
    ... Implement, for example, section 2.1.1.2 of Jiang (2015) ...

# A complex kernel that is registered as the svd_forward complex kernel
@ti.complex_kernel_grad(svd_forward)
def svd_backward(num_iterations):
  # differentiave SVD
  svd_gradient()
```

D Checkpointing

In this section we demonstrate how to use checkpointing via complex kernels. The goal of checkpointing is to use recomputation to save memory space. We demonstrate this using the `diffmpm` example, whose simulation cycle consists of particle to grid transform (`p2g`), grid boundary conditions (`grid_op`), and grid to particle transform (`g2p`). We assume the simulation has $O(n)$ time steps.

D .1 Recomputation within Time steps

A naive implementation without checkpointing allocates $O(n)$ copied of the simulation grid, which can cost a lot of memory space. Actually, if we recompute the grid states during the backward simulation time step by redoing `p2g` and `grid_op`, we can reused the grid states and allocate only one copy. This checkpointing optimization is demonstrated in the code below:

```
@ti.complex_kernel
def advance(s):
  clear_grid()
  compute_actuation(s)
  p2g(s)
  grid_op()
  g2p(s)
```

```
@ti.complex_kernel_grad(advance)
def advance_grad(s):
  clear_grid()
  p2g(s)
  grid_op() # recompute the grid

  g2p.grad(s)
  grid_op.grad()
  p2g.grad(s)
  compute_actuation.grad(s)
```

D .2 Segment-Wise Recomputation

Given a simulation with $O(n)$ time steps, if all simulation steps are recorded, the space consumption is $O(n)$. This linear space consumption is sometimes too large for high-resolution simulations with long time horizon. Fortunately, we can reduce the space consumption using a segment-wise checkpointing trick: We split the simulation into segments of S steps, and in forward simulation store only the first simulation state in each segment. During backpropagation when we need the remaining simulation states in a segment, we recompute them based on the first state in that segment.

Note that if the segment size is $O(S)$, then we only need to store $O(n/S)$ simulation steps for checkpoints and $O(S)$ reusable simulation steps for backpropagation within segments. The total space consumption is $O(S + n/S)$. Setting $S = O(\sqrt{n})$ reduces memory consumption from $O(n)$ to $O(\sqrt{n})$. The time complexity remains $O(n)$.

E Details on 10 Differentiable Simulators

E .1 Differentiable Continuum Mechanics for Elastic Objects [diffmpm]

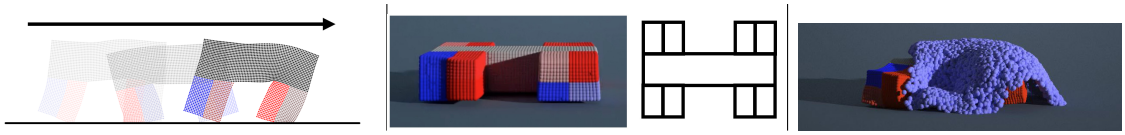
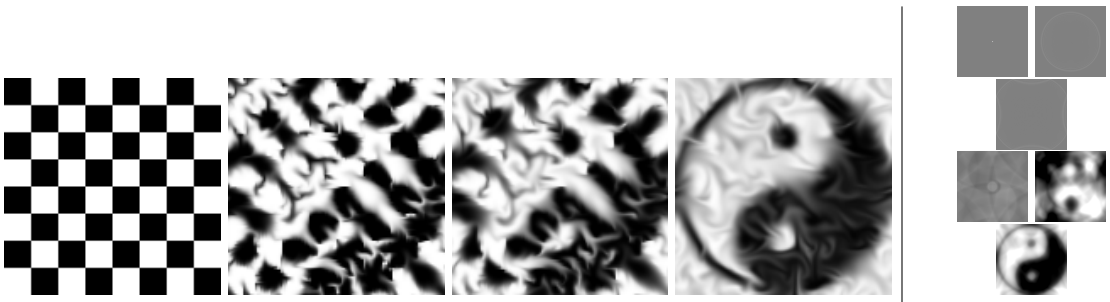


Figure 3-6: Controller optimization with our differentiable continuum simulators. **Left:** the 2D robot with four muscles. **Middle:** A 3D robot with 16 muscles and 30K particles crawling on the ground. **Right:** We couple the robot (30K particles) and the liquid simulator (13K particles), and optimize its open-loop controller in this difficult situation.

E .2 Differentiable liquid simulator [liquid]

We follow the weakly compressible fluid model in [116] and implemented a 3D differentiable liquid simulator within the [diffmpm3d] framework. Our liquid simulation can be two-way coupled with elastic object simulation (Figure 3-6, right).

E .3 Differentiable Incompressible Fluid Simulator [smoke]



(a) smoke (b) wave

Figure 3-7: a: (Left to right) with an optimized initial smoke velocity field, the fluid changes its pattern to a “Taichi" symbol. b: Unoptimized (top three) and optimized (bottom three) waves at time step 3, 189, and 255.

Backpropagating Through Pressure Projection We followed the baseline implementation in Autograd, and used 10 Jacobi iterations for pressure projection.

Technically, 10 Jacobi iterations are not sufficient to make the velocity field fully divergence-free. However, in this example, it does a decent job, and we are able to successfully backpropagate through the unrolled 10 Jacobi iterations.

In larger-scale simulations, 10 Jacobi iterations are likely not sufficient. Assuming the Poisson solve is done by an iterative solver (e.g. multigrid preconditioned conjugate gradients, MGPCG) with 5 multigrid levels and 50 conjugate gradient iterations, then automatic differentiation will likely not be able to provide gradients with sufficient numerical accuracy across this long iterative process. The accuracy is likely worse when conjugate gradients present, as they are known to numerically drift as the number of iterations increases. In this case, the user can still use DiffTaichi to implement the forward MGPCG solver, while implementing the backward part of the Poisson solve manually, likely using adjoint methods [32]. DiffTaichi provides "complex kernels" to override the built-in AD system, as shown in Appendix C .

E .4 Differentiable Height Field Shallow Water Simulator `[wave]`

We adopt the wave equation in [121] to model shallow water height field evolution:

$$\ddot{u} = c^2\nabla^2 u + c\alpha\nabla^2\dot{u}, \tag{3.1}$$

where u is the height of shallow water, c is the "speed of sound" and α is a damping coefficient. We use the $\dot{u}$ and $\ddot{u}$ notations for the first and second order partial derivatives of u w.r.t time t respectively.

[121] used the finite different time-domain (FDTD) method [71] to discretize Eqn. 3.1, yielding an update scheme:

$$u_{t,i,j} = 2u_{t-1,i,j} + (c^2\Delta t^2 + c\alpha\Delta t)(\nabla^2 u)_{t-1,i,j} - p_{t-2,i,j} - c\alpha\Delta t(\nabla^2 u)_{t-2,i,j},$$

where

$$(\nabla^2 u)_{t,i,j} = \frac{-4u_{t,i,j} + u_{t,i,j+1} + u_{t,i,j-1} + u_{t,i+1,j} + u_{t,i-1,j}}{\Delta x^2}.$$

We implemented this wave simulator in DiffTaichi to simulate shallow water. We used a grid of resolution 128×128 and 256 time steps. The loss function is defined as

$$L = \sum_{i,j} \Delta x^2 (u_{T,i,j} - \hat{u}_{i,j})^2$$

where T is the final time step, and $\hat{u}$ is the target height field. 200 gradient descent iterations are then used to optimize the initial height field. We set $\hat{u}$ to be the pattern “Taichi", and Fig. 3-7b shows the unoptimized and optimized wave evolution.

We set the “Taichi" symbol as the target pattern. Fig. 3-7b shows the unoptimized and optimized final wave patterns. More details on discretization is in Appendix E .

E .5 Differentiable Mass-Spring system `[mass_spring]`

We extend the mass-spring system in the main text with ground collision and a NN controller. The time-of-impact fix is implemented for improved gradients. The optimization goal is to maximize the distance moved forward with 2048 time steps. We designed three mass-spring robots as shown in Fig. 3-8 (left).

E .6 Differentiable Billiard Simulator `[billiards]`

A differentiable rigid body simulator is built for optimizing a billiards strategy (Fig. 3-8, middle). We used forward Euler for the billiard ball motion and conservation of momentum and kinetic energy for collision resolution.

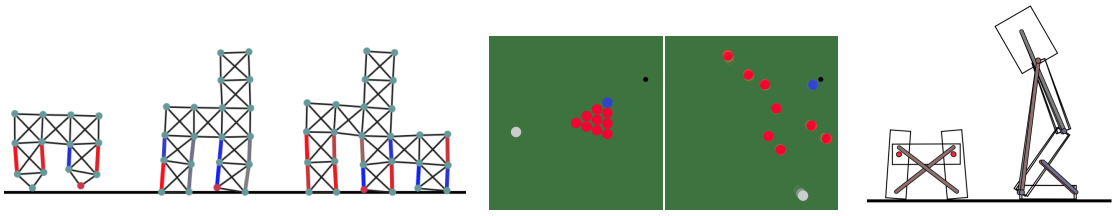


Figure 3-8: **Left:** Three mass-spring robots. The red and blue springs are actuated. A two layer NN is used as controller. . **Middle:** Optimizing billiards. The optimizer adjusts the initial position and velocity of the white ball, so that the blue ball will reach the target destination (black dot). **Right:** Optimizing a robot walking. The rigid robot is controlled with a NN controller and learned to walk in 20 gradient descent iterations.

E .7 Differentiable Rigid Body Simulator `[rigid_body]`

Are rigid body collisions differentiable? It is worth noting that discontinuities can happen in rigid body collisions, and at a countable number of discontinuities the objective function is non-differentiable. However, apart from these discontinuities, the process is still differentiable almost everywhere. The situation of rigid body collision is somewhat similar to the "ReLU" activation function in neural networks: at point $x = 0$, ReLU is not differentiable (although continuous), yet it is still widely adopted. The rigid body simulation cases are more complex than ReLU, as we have not only non-differentiable points, but also discontinuous points. Based on our experiments, in these impulse-based rigid body simulators, we still find the gradients useful for optimization despite the discontinuities, especially with our time-of-impact fix.

E .8 Differentiable Water Renderer `[water_renderer]`

We implemented differentiable renderers to visualize the refracting water surfaces from `wave`. We use finite differences to reconstruct the water surface models based on the input height field and refract camera rays to sample the images, using bilinear interpolation for meaningful gradients. To show our system works well with other differentiable programming systems, we use an adversarial optimiza-

tion goal: fool VGG-16 into thinking that the refracted squirrel image is a goldfish (Fig. 3-9).

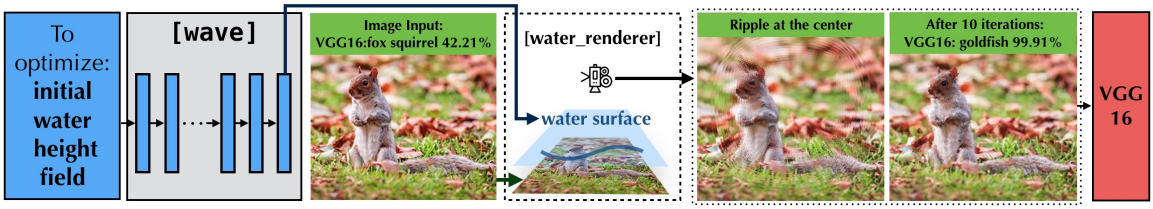


Figure 3-9: This three-stage program (simulation, rendering, recognition) is end-to-end differentiable. Our optimized initial water height field evolves to form a refraction pattern that perturbs the image into one that fools VGG16 (99.91% goldfish).

E .9 Differentiable Volume Renderer `[volume_renderer]`

We implemented a basic volume renderer that simply uses ray marching (we ignore light, scattering, etc.) to integrate a density field over each camera ray. In this task, we render a number of target images from different viewpoints, with the camera rotated around the given volume. The goal is then to optimize for the density field of the volume that would produce these target images: we render candidate images from the same viewpoints and compute an L2 loss between them and the target images, before performing gradient descent on the density field (Fig. 3-10). Essentially, this demonstrates how to use gradients to reconstruct 3D objects out of X-ray photos in a brute-force manner. Other approaches to this task include algebraic reconstruction techniques (ART) [36].

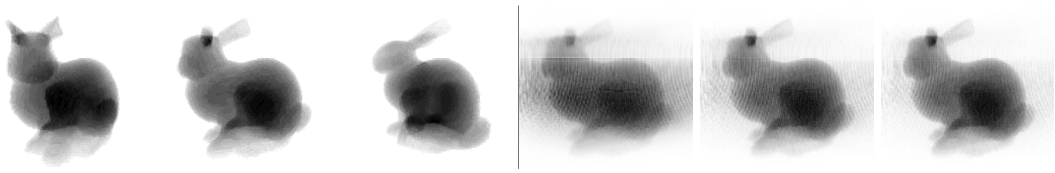
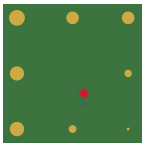


Figure 3-10: Volume rendering of bunny shaped density field. **Left:** 3 (of the 7) target images. **Right:** optimized images of the middle bunny after iteration 2, 50, 100.

E .10 Differentiable Electric Field Simulator [electric]

Recall Coulomb's law: $\mathbf{F} = k\frac{q_1 q_2}{r^2}\hat{\mathbf{r}}$. In the right figure, there are eight electrodes carrying static charge. The red ball also carries static charge. The controller, which is a two-layer neural network, tries to manipulate the electrodes so that the red ball follows the path of the blue ball. The bigger the electrode, the more positive charge it carries.



F Fixing Gradients with Time of Impact and Continuous Collision Detection

Here is a naive time integrator in the mass-spring system example:

```
@ti.kernel
def advance(t: ti.i32):
  for i in range(n_objects):
    s = math.exp(-dt * damping)
    new_v = s * v[t - 1, i] + dt * gravity * ti.Vector([0.0, 1.0])
    old_x = x[t - 1, i]
    depth = old_x[1] - ground_height
    if depth < 0 and new_v[1] < 0:
      # assuming a sticky ground (infinite coefficient of friction)
      new_v[0] = 0
      new_v[1] = 0

    # Without considering time of impact, we assume the whole dt uses new_v
    new_x = old_x + dt * new_v

    v[t, i] = new_v
    x[t, i] = new_x
```

Implementing TOI in this system is relative straightforward:

```
@ti.kernel
def advance_toi(t: ti.i32):
  for i in range(n_objects):
    s = math.exp(-dt * damping)
    old_v = s * v[t - 1, i] + dt * gravity * ti.Vector([0.0, 1.0])
    old_x = x[t - 1, i]
    new_x = old_x + dt * old_v
    toi = 0.0
```

```
    new_v = old_v
    if new_x[1] < ground_height and old_v[1] < -1e-4:
      # The 1e-4 safe guard is important for numerical stability
      toi = -(old_x[1] - ground_height) / old_v[1] # Compute the time of impact
      new_v = ti.Vector([0.0, 0.0])

    # Note that with time of impact, dt is divided into two parts,
    #   the first part using old_v, and second part using new_v
    new_x = old_x + toi * old_v + (dt - toi) * new_v

    v[t, i] = new_v
    x[t, i] = new_x
```

In rigid body simulation, the implementation follows the same idea yet is slightly more complex. Please refer to `rigid_body.py` for more details.

G Additional Tips on Gradient Behaviors

Initialization matters: flat lands and local minima in physical processes A trivial example of objective flat land is in `billiards`. Without proper initialization, gradient descent will make no progress since gradients are zero (Fig. 3-11). Also note the local minimum near $(-5, 0.03)$.

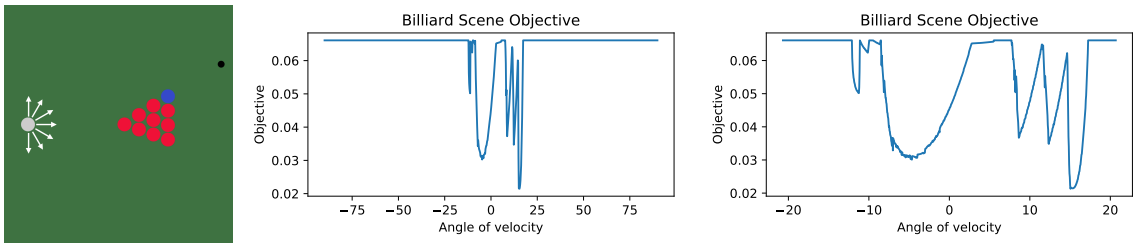


Figure 3-11: **Left:** Scanning initial velocity in the billiard example. **Middle:** Most initial angles yield a flat objective (final distance between the blue ball and black destination) of 0.065, since the white ball does not collide with any other balls and imposes no effect on the pink ball via the chain reaction. **Right:** A zoomed-in view of the middle figure. The complex collisions lead to a lot of local minimums.

In `mass_spring` and `rigid_body`, once the robot falls down, gradient descent will quickly become trapped. A robot on the ground will make no further progress, no

matter how it changes its controller. This leads to a more non-trivial local minimum and zero gradient case.

Ideal physical models are only "ideal": discontinuities and singularities Real-world macroscopic physical processes are usually continuous. However, building upon ideal physical models, even in the forward physical simulation results can contain discontinuities. For example, in a rigid body model with friction, changing the initial rotation of the box can lead to different corners hitting the ground first, and result in a discontinuity (Fig. 3-12). In `electric` and `mass_spring`, due to the $\frac{1}{r^2}$ and $\frac{1}{r}$ terms, when $r \to 0$, gradients can be very inaccurate due to numerical precision issues. Note that $d(1/r)/dr = -1/r^2$, and the gradient is more numerically problematic than the primal for a small r. Safeguarding r is critically important for gradient stability.

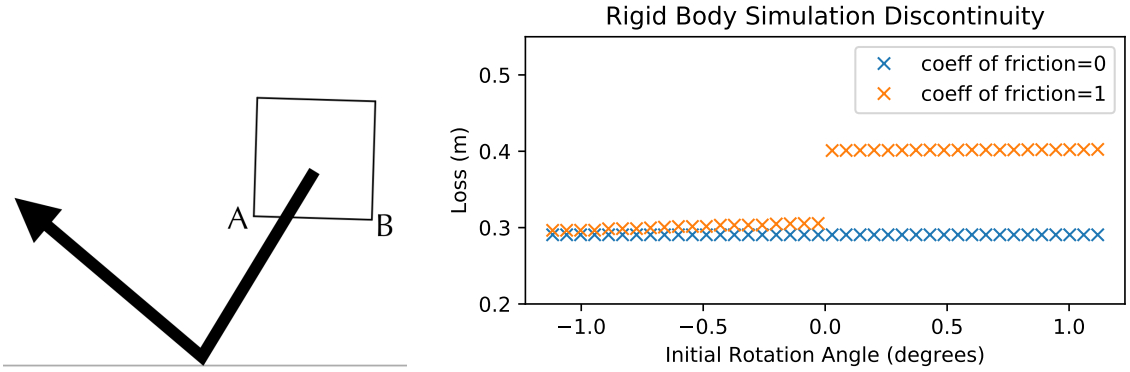


Figure 3-12: Friction in rigid body with collision is a common source of discontinuity. In this scene a rigid body hits the ground. Slightly rotating the rigid body changes which corner (A/B) hits the ground first, and different normal/friction impulses will be applied to the rigid body. This leads to a discontinuity in its final position (loss=final y coordinate). Please see our supplemental video for more details.

Chapter 4

Asynchronous execution and inter-kernel optimizations

Writing high-performance graphics code that fully utilizes parallel processors such as GPUs takes skilled and time-consuming performance engineering. For example, when writing a high-performance GPU physical simulator, issues such as kernel launching overhead and poor memory locality often lead to low processor utilization. We seek to increase programmer productivity and improve performance by introducing an automatic inter-kernel optimization approach for parallel computation, in an *imperative* context where data structures are mutable. Existing imperative programming systems such as CUDA and Taichi launch computational kernels in an eager fashion, severely restricting the ability to perform inter-kernel optimizations such as inter-kernel dead code elimination and kernel fusion. Our approach preserves the benefits of eager approaches while performs on-the-fly analysis of possible inter-kernel optimizations before actually launching kernels.

Inter-kernel optimization is especially relevant for tasks beyond traditional dense arrays computation, as exhibited by modern computer graphics and machine learning workloads, such as physical simulation with *spatial sparsity* and automatic gradient evaluation via *differentiable programming*. We show that these emerging computational patterns lead to new and exciting automatic optimization opportunities. For example, by analyzing computation programs across kernel boundaries, our

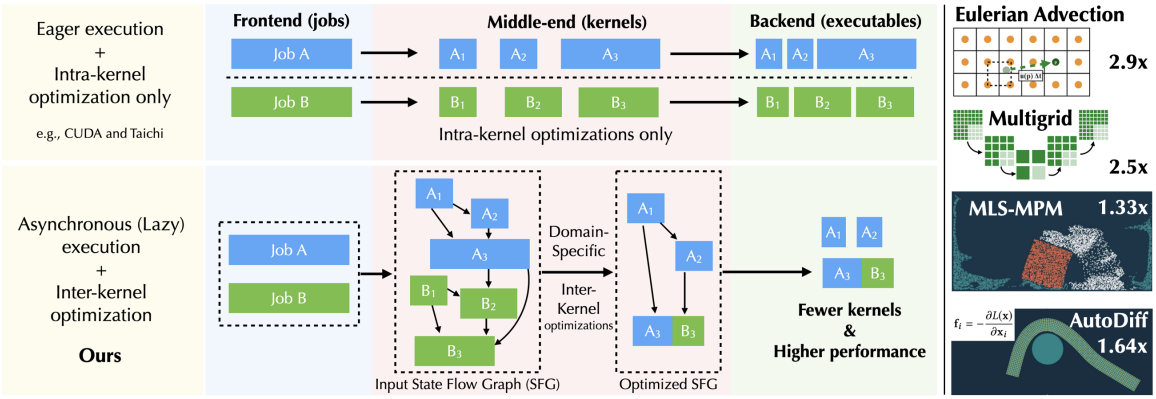


Figure 4-1: **Top left:** In existing parallel imperative programming systems (such as CUDA and Taichi [49]), imperative computational kernels are eagerly launched, leaving a tiny room for the optimizer to optimize *beyond a single kernel*. **Bottom left:** In our work, we accumulate kernels in an execution buffer, only flushing the execution queue when necessary. This allows the optimizer to gain more context and conduct optimization beyond a single kernel. We dynamically build a dependency graph ("state-flow graph") of kernels for easy analysis, so that computation kernels can be optimized at a inter-kernel level just in time. **Right:** After a suite of domain-specific optimization passes including list generation removal, sparse data structure activation elimination, and kernel fusion, kernels are much better optimized. As a result, the inter-kernel optimized programs run $1.87\times$ faster on GPUs, *without the user modifying any of the computation code*.

optimizer can eliminate unnecessary voxel list generation tasks for parallel iterations on sparse data structures, accelerate sparse data structure access, and remove unused gradient evaluations. To provide the maximum programming flexibility, our optimization system conducts *on-the-fly* optimization on a large window of computational graph consisting of parallel kernels. To analyze imperative programs with mutable data structures, we propose a *state-flow graph* formulation of *imperative* programs, which describes kernel relationships and enables easy analysis and optimization. The optimized parallel kernels are then just-in-time compiled in parallel, and dispatched to parallel devices such as multi-threaded CPU and massively parallel GPUs. *Without any computational code modification*, our new system leads to $4.02\times$ fewer kernel launches and $1.87\times$ speed up on our GPU benchmarks, including sparse-grid physical simulation and differentiable programming.

1 Introduction

The tension between the ease of programming and performance is challenging in many computational applications including physical simulation. This is especially true on modern hardware architectures such as GPUs. The implementation of a piece of clean, modular code often leads to severe inefficiencies due to overheads such as kernel launches for logically unrelated tasks on GPUs.

Automatic compiler optimizations can alleviate this performance-productivity tension. Among them, inter-kernel optimizations, which analyze and improve performance across kernel boundaries, are effective ways to speed up GPU code, yet if not done automatically, implementing these optimizations can be laborious. For example, an experienced GPU programmer may manually merge *logically unrelated* kernels to reduce kernel launching overhead and improve data locality, at the cost of devoting more low-level engineering efforts and sacrificing code readability.

Imperative GPU programming Domain-Specific Languages (DSLs), such as Taichi [49] for physical simulation applications, partially alleviate the performance-productivity trade-off. However, their existence further underscores the importance of automatic inter-kernel optimizations. This is because (1) DSL compilers usually do a great job regarding intra-kernel optimizations, leaving little room for further improvement within the kernel, (2) these compilers, as layers of abstraction, tend to introduce auxiliary computation kernels that are invisible to programmers and are impossible to optimize manually, and (3) the exotic patterns in DSLs are often different enough and they have not yet been addressed in the classical general-purpose compiler community.

We present an automatic inter-kernel optimization system for *parallel* and *imperative* programming, especially on GPUs. Our system not only serves as a flexible on-the-fly optimization infrastructure for general-purpose, imperative GPU programming systems, but also supports optimizing domain-specific computations such as spatially sparse simulation [89, 110, 49] and differentiable programming [47, 58, 76].

Optimizing across function and kernel boundaries is an existing technique in traditional ahead-of-time compilers (see, for example, gcc WHOPR [16]), and functional array-based parallel JIT systems, especially deep learning systems such as TensorFlow [3] and JAX [14]. However, three major challenges exist when applying the idea of inter-kernel optimization to modern graphics programming systems and applications, especially physical simulation:

1. Graphics programs are mostly imperative, has mutable data buffers, and allows *partial* updates of these buffers. Unlike systems that rely on dense, immutable, and holistic arrays ("tensors" in systems like TensorFlow [3]), partial updates and mutable, sparse buffers can lead to difficulties when analyzing an imperative graphics program.

2. Domain-specific features of graphics programming systems, such as spatial sparsity, can lead to further complexities to the analysis and optimization process. For example, while tools for analyzing dense array programs are well established [70], their counterparts in *spatially sparse* array computations are largely underexploited.

3. Imperative GPU programming systems often eagerly execute computational tasks. This eager execution scheme leaves little room to analyze and optimize beyond a single kernel.

In this work, we propose a *state-flow formulation* of imperative GPU programs, to analyze those programs with partial and in-place updates. "States" in our formulation refer not only to memory bytes, but also to domain-specific and abstract descriptions of data, such as the topology of the sparse data structures and auxiliary lists of active voxels. We build a *state-flow graph (SFG)* on-the-fly consisting of pending kernels, to depict kernel relationships. The rich expressiveness of SFGs allows us to conduct domain-specific inter-kernel optimizations, such as fusing kernels on dynamic sparse data structures, eliminate unnecessary voxel list generation tasks for parallel iterations on sparse data structures, accelerate sparse data structure access, and remove unused gradient evaluations.

To make the optimizer see more than a single kernel at a time, we built an *asynchronous execution engine* that maintains a window of kernels, leaving room for performance optimizations before launching the kernels. The execution engine also enables *parallel compilation*, which significantly reduces the JIT compilation time.

With productivity and performance in mind, our system is practically designed according to the following guidelines:

- **Imperative programming.** Graphics programs, especially physical simulation, are usually computation hungry and need imperative programming, which is closer to hardware compared to functional programming, to extract the maximum possible performance out of parallel processors. Our system sticks to the imperative programming setting for maximum compatibility with existing graphics algorithms.

- **Spatially sparse data structures.** Recent work in graphics [89, 110, 49] demonstrates the efficacy of sparse data strictures in simulation and rendering. Our systems need to support these widely adopted data storage infrastructures for performance and scalability. Meanwhile, we expect the versatile and mutable sparse data structures serve as a strong test case of the expressiveness of the state-flow graph for inter-kernel optimizations.

- **Transparent to users.** We wish users can get the benefits of inter-kernel optimizations for *free*. No code modification is needed in the computational kernels for users to leverage our optimizations.

- **Optimize on-the-fly for flexibility.** Graphics and numerical computation applications, especially iterative solvers, tend to have data-dependent control flows. For example, whether a conjugated gradients solver should stop iterating, depends on the residual of the last iteration, which is computed using a kernel. To keep control-flow flexibility at the kernel level, we dynamically build a state-flow graph and optimize in a kernel window of reasonable size.

- **Compile just-in-time (JIT).** JIT *delays* the need of value of "compile-time constant" values. For example, Δt in physical simulators is often a runtime variable in ahead-of-time compilation, but with JIT, Δt would be a compile-time constant. This allows the compiler to do more optimizations such as constant folding. More importantly, the optimized kernels need a JIT system to compile into hardware executables.

- **Problems of all scales matter.** Graphics applications cover a wide range of problem sizes. For example, a particle simulation may cover from $2,000$ to $100,000,000$ [124] particles. For small-scale tasks, *compilation* time may be the bottleneck; for large scale tasks, *computation* time is more important. Our asynchronous execution engine enables parallel compilation to reduce the JIT compilation delay.

- **GPU first.** General-purpose GPU programming is becoming increasingly popular, and the need for high-quality of graphics computation is often only achievable on massively GPUs. While our system can indeed improve multicore CPU performance, we put more priority on GPU computation when making design decisions.

We summarize our contributions as follows:

1. A state-flow formulation of imperative parallel computation. The resulted *state-flow graphs (SFGs)* serve as a high-level intermediate representation (IR) of GPU programs. It can model not only general-purpose GPU programs, but also domain-specific ones, such as those with spatial sparsity.

2. An asynchronous task execution engine that exposes inter-kernel optimization opportunities and enables parallel compilation;

3. Most importantly, *an inter-kernel optimizer* for asynchronous spatially sparse and differentiable computation. The optimizer can conduct general-purpose inter-kernel optimizations such as dead store elimination and kernel fusion,

as well as domain-specific optimizations such as list generation removal and sparse data structure activation demotion;

4. A systematic study of the resulted system. Based on the benchmarks, we show our inter-kernel optimizer delivers $1.87\times$ (geometric mean) wall-clock time improvements and $4.02\times$ fewer GPU kernel launches compared to the reference system [49]. All these can be achieved *without the programmer modifying a single line of kernel code*.

We pick Taichi [49] as the starting point to build our system, partially reusing its compiler infrastructure.

2 Related Work

Spatially sparse computation The idea of leveraging spatial sparsity in graphics originates from popular sparse data structures including VDB [91, 41, 131] and SPGrid [110, 35]. While these data structures have demonstrated effective computation and storage benefits over dense arrays, writing programs that leverage them is not an easy task. Taichi [49] provides a language abstraction that allows using these data structures as if they are dense, and runtime systems that automatically handle parallel voxel iteration and memory management. These designs benefit the end users, but may end up with more computation. These redundant jobs would need an inter-kernel analysis to optimize.

Another thread of work on sparse computation is sparse linear algebra languages, such as TACO [68, 23], which can effectively generate kernels for Einstein summations on sparse matrices and tensors. Instead of explicitly building the sparse matrices, some simulators use linear algebra computations, which are often the more effective ways for high-performance linear algebra solves in physical simulations (see, for example [79]).

Array data-flow analysis The static-single assignment (SSA) form has been a very popular IR structure. SSA forms are designed for scalar variables, and it can-

not directly represent array states, where partial updates may happen. Array SSA forms have been proposed and successfully adopted in parallelization [70] and array privatization [84]. However, related work in this topic is mostly focused on dense arrays. Our high-level IR system represents not only array partial updates, but also the topology changes in sparse arrays.

Whole-Program Optimization (WPO) WPO is also known as Inter-procedural optimization (IPO). For ahead-of-time compilation, IPO typically happens at link time, so sometimes it is also called link-time optimization (LTO). Many existing compilers, such as gcc, MSVC, and clang, already support LTO and WHO (see, e.g., gcc WHOPR [16]). While WPO is extensively explored in classical compiling systems, it is still underexploited for spatially sparse computation. The unique computational pattern in sparse computation brings higher complexity and the need for a unified high-level intermediate representation for analysis and optimization.

Computational graph optimization in deep learning frameworks A feed-forward deep neural (DNN) network can be naturally represented as directed acyclic graphs (DAG). This leads to a straightforward mapping between DNNs and the computational graph: *immutable, dense feature maps* directly map to the graph *edges*, and *operators* (such as convolutions, max pooling, and element-wise add) maps to the graph *nodes*. Consequently, modern deep learning frameworks (TensorFlow [3], PyTorch [**?**], ONNX [**?**], Theano [117]) have widely adopted the computational graph to represent the DNN models. High-level optimizations on the computational graph have been a popular feature in deep learning frameworks. The HLO IR of XLA and PyTorch GLOW [107] are representative examples. Based on the computational graph, traditional computer optimizations such as operator fusion, dead code elimination (DCE), common subexpression elimination (CSE) can be applied. We refer the readers to [75] for a good survey.

In deep learning frameworks such as TensorFlow [3], every operation creates

a new, immutable buffer ("tensor"), and DNNs are essentially data flow of feature maps. In graphics applications, however, we have to adopt an imperative programming paradigm and support in-place updates. This is not only because graphics programmers have been accustomed to using imperative programming (e.g., C++, CUDA, and GLSL) for decades, but also because in-place operations in imperative programming offer significant performance advantages in graphics applications, such as physical simulation, since in-place operations are closer to hardware compared to functional programming.

Our system is similar to these systems in that a high-level graph-based IR is used, yet the high-level IR must consider its partial updates, sparsity, and "megakernel" (i.e., many-in-many-out, hundreds of instructions per kernel) natures of sparse computation code.

GPU code optimization Extensive research has been done on code optimization for GPUs. For example, Hong et al. [42] optimize SASS via emulation and identifying bottlenecks. Filipovič et al. [33] optimize CUDA kernels via kernel fusion on operations in the forms of map, reduce, and demonstrated speed up on dense BLAS operations. Bo et al. [101] proposed an automatic fusion framework for image processing DSLs. However, an on-the-fly inter-kernel optimization system on GPU imperative programming models that provides maximum flexibility for general-purpose computation is still missing.

Differentiable array programming in graphics and AI Many differentiable programming tools operate on dense arrays, instead of scalars, to exploit parallelism [77, 83, 14, 58]. Admittedly, array-based programming interfaces, usually with limitations on partial updates, are easier to analyze. However, expressing many graphics computational patterns (such as scattering particles onto a regular grid) using arrays may not be a performance option.

Physical Simulation DSLs Developing high-performance physics solvers is challenging, and a lot of low-level engineering is needed to exploit the capabilities of modern parallel processors. High-level DSLs models simulations as meshes (Liszt [30]), sparse linear algebra [69], and relational data models [13].

A lower-level system that is more closely related to our work is the Taichi programming language [49]. Taichi is a DSL with first-class support for sparse data structures, a critical component of modern high-performance physical simulators. Taichi also supports differentiable programming [47], allowing developers to evaluate gradients of physical simulators for machine learning and optimization purposes. In the next section, we briefly cover core Taichi features related to this work.

3 Taichi background: Imperative, megakernel, sparse, and differentiable programming

Taichi [49] is a new programming language for spatially sparse and differentiable visual computing. As a domain-specific language embedded in Python, Taichi's just-in-time compiler transforms compute-intensive kernels ("Megakernels", similar to a `__global__` GPU kernel in CUDA) into parallel executables. Users can flexibly launch the kernels using Python. In many simulation workloads, Taichi programs can achieve comparable performance to handwritten CUDA code, using only $1/10$ lines of code (see, for example, benchmarks in [49]).

Taichi is almost as expressive as any other general-purpose GPU programming language such as CUDA. Meanwhile, as an open-source system, it provides opportunities for researchers to experiment with new compiler optimizations. Moreover, Taichi has first-class supports for two popular topics in the graphics programming community: (a) using sparse data structures to accelerate physical simulation, and (b) make graphics systems differentiable. All these features make it a suitable piece of infrastructure for our study.

We recap key Taichi features related to this chapter below.

3 .1 Data-oriented programming

`field` is a key concept in Taichi that represents data. A `field` is essentially a one- to eight-dimensional tensor. Each element of the tensor can be a scalar (e.g., density), a small vector (e.g., velocity), or a small matrix (e.g., stress tensor). Externally, a `field` in Taichi is *flat*: field elements are always accessed via an `x[i, j, k]`-style syntax, regardless of its data layout. Internally, however, data are organized in *hierarchical* tree structures, described via structural nodes (SNodes). See Fig. 4-2 as an example.

Commonly used SNodes in Taichi are `dense`, `bitmasked`, `pointer`, and `dynamic` [49]. They can easily compose into complex data structures that are dense or sparse.

Note that the field shapes are known at compile-time, allowing the compiler to easily conduct aliasing analysis.

3 .2 Sparse programming

A `field` in Taichi can be either dense (similar to a CUDA array) or spatially sparse (such as a VDB [89] or SPGrid [110]). The support for spatial sparsity [49] is a unique feature of Taichi. Most 3D graphics data (especially those stored on voxel grids) are spatially sparse, and Taichi has first-class support for sparse data structures to leverage this property for acceleration.

To make sparse data structures as intuitive to use as dense data structures, various designs are made on the syntax, compiler, and runtime levels:

1. *Sparse struct-for loops* allow users to iterate over *active voxels* of sparse data structures easily. For example, the following code loops over a 3D sparse field:

```
for i, j, k in x:
    x[i, j, k] += 1
```

While iterating over only active parts of a sparse data structure improves performance, implementing such iteration in parallel is highly non-trivial, since the sparse data structure trees are often highly unbalanced. Therefore, instead of recursively looping on the tree, we generate lists of active tree nodes, layer by layer, from top to bottom. Such *list generation* (Fig. 4-2, left branch) process is the key mechanism to achieve load-balanced parallel sparse for-loops.

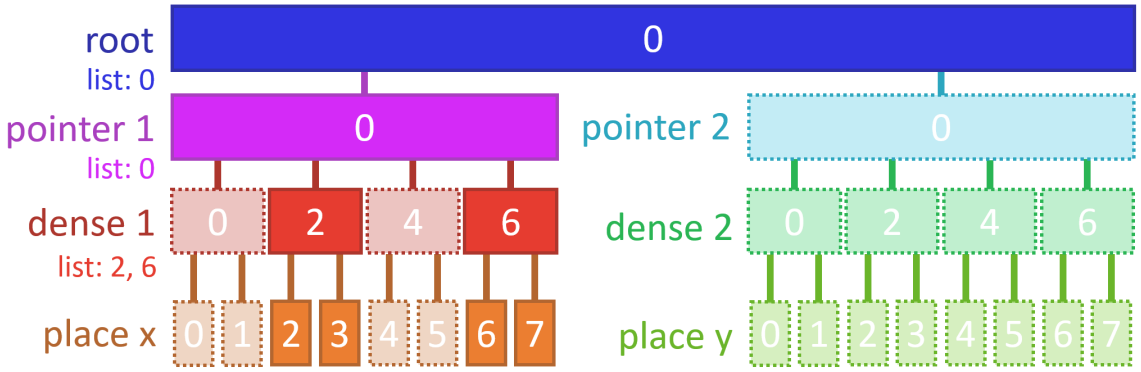


Figure 4-2: **Left branch:** The structure of `ti.root.pointer(ti.i, 4).dense(ti.i, 2).place(x)` in Taichi, a two-level 1D sparse data structure. The first level, `pointer(ti.i, 4)`, is a four-cell pointer array. Each cell of the pointer array can be a null pointer if it is inactive. The second level, `dense(ti.i, 2)`, is dense blocks with two cells each. Highlighted cells are active. Lists of each layer are defined to be collections of active node indices. **Right branch:** The same structure for `field y`, which is completely inactive for now.

Note that lists themselves are generated via GPU kernels. In certain cases, the time it takes to generate the lists is comparable to that of the essential parallel iteration.

2. *Activation on write* ensures sparse data structure nodes are implicitly activated on writing. For example, the following code generates a $2 \times 2 \times 2$ downsampled sparse field `y` from a higher-resolution sparse field `x`:

```
for i, j, k in x:
    y[i // 2, j // 2, k // 2] += x[i, j, k]
```

Note that the corresponding voxels of `y` may not be active before this for loop.

Taichi will automatically activate `y[i // 2, j // 2, k // 2]` and zero-fill the voxels. See Figure 4-3 for an example.

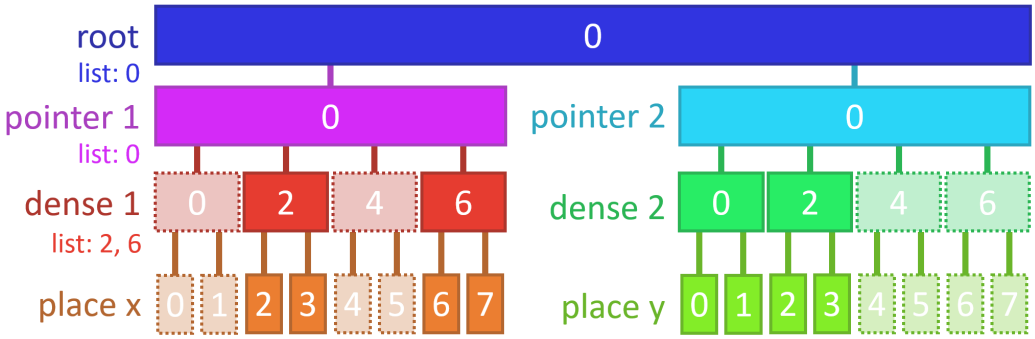


Figure 4-3: Execution result of simple program `for i in x: y[i // 2] += 1`. `y[1]` and `y[3]` are activated on write. Because dense nodes cannot be only partially active, `y[0]` and `y[2]` are also activated.

3. *Automatic memory management* frees users from worrying about memory allocation and deallocation. Taichi's high-performance memory allocator will automatically manage sparse data structure nodes.

4. *Programmable megakernels* allow users to easily write parallel programs with very high flexibility and rich expressiveness. Taichi kernels allow complex control flows - in fact, Taichi programmers can easily write a recursive ray tracer (See, e.g., [49]).

5. *Automatic parallelization*. Taichi *kernels* are decomposed into *tasks* that are serial or parallel. For example, in the following code, the for loops are automatically parallelized:

```
@ti.kernel
def reduce():
    s = 0 # Serial
    for i in range(128):
        s += x[i] # Parallel range-for
    for i in y:
        s += y[i] # Parallel struct-for
    print(s) # Serial
```

4 A State-flow formulation of Sparse and Differentiable Computation

In imperative programming, *Program* = *State* + *Compute*. In graphics applications, "state" usually means more than data per se. This is because complex acceleration data structures are often used in high-performance graphics code, such as bounding volume hierarchy (BVH) for ray tracing and sparse grids in simulation. Graphics computation is often coupled with these data structures. Specifically in spatially sparse and differentiable programming, "state" means way more than values of `fields`, and "compute" means more than GPU kernels that operate on data. For example, iterating over sparse data structures in parallel, implicitly leads to auxiliary computation on list generation and node activation, and differentiable programming automatically creates a gradient version of the forward computation. These patterns create more challenges in modeling and analyzing graphics programs.

For now, let us assume we know the whole execution history of a Taichi program. We reformulate the imperative computation scheme of Taichi into a collection of *states* and *tasks*. To systematically optimize imperative GPU programs, especially those with spatial sparsity support, we formulate a Taichi program as a **state-flow graph (SFG)**, which is a directed acyclic graph (DAG) with nodes being *tasks* and edges being *states*. This results in a state flow formulation and a high-level intermediate representation (IR). Scalar data-flow analysis is well studied in optimizing compilers (see, for example, [66]), and SFGs can be considered an extended version of data-flow analysis to handle auxiliary states such as lists in spatially sparse computation.

States States split the holistic description of a Taichi program into a suitable granularity for analysis and optimization. For SNodes that are spatially sparse, we must decompose the holistic descriptions of their *data, topology, and auxiliary structures* into the following states:

- *A value state* simply represents the collection of numerical values stored in the field. Note that in data structure trees of Taichi, only the leaf nodes (i.e., `place` SNodes) store numerical values. Value states are the most basic states, have the same meaning as those in data flow optimization, and are useful in almost all GPU programming systems. It is worth noting that in sparse data structures, every voxel has a numerical value, even if the voxel is sparse - in that case, the inactive voxel has an ambient value 0.

- *A mask state* of a SNode records the activation information of all its cells (Fig. 4-4, right). Mask states must be handled as first-class primitives in our SFG system for domain-specific optimizations. Masks are scattered in various forms in the data structure. They are either indicated by a bit in a bitmask SNode, or a non-null pointer for the pointer SNode. Mask is a unified concept for different data structures. Therefore we need to generate a unified element list for struct-fors on different structures.

- *A list state* of an SNode represents the data structure nodes maintained by the runtime system. Recall that Taichi needs to generate/consume data structure node lists for load-balancing parallel iterations over sparse data structure nodes. List generation tasks take the list and mask states of the parent SNode to generate the list of the current SNode. Lists are consumed by (parallel) struct-fors. See [49] for more details on load balancing and parallel fors on unbalanced trees.

- *An allocator state* represents the state of Taichi's memory allocator. For computation that allocates/deallocates sparse data structure nodes, the allocator states are marked as modified.

The relationship between value, mask, and list state is depicted in Fig. 4-4.

Tasks A Taichi kernel may be decomposed into multiple parallel tasks (GPU kernels). Without loss of generality, we assume that a Taichi kernel corresponds to a

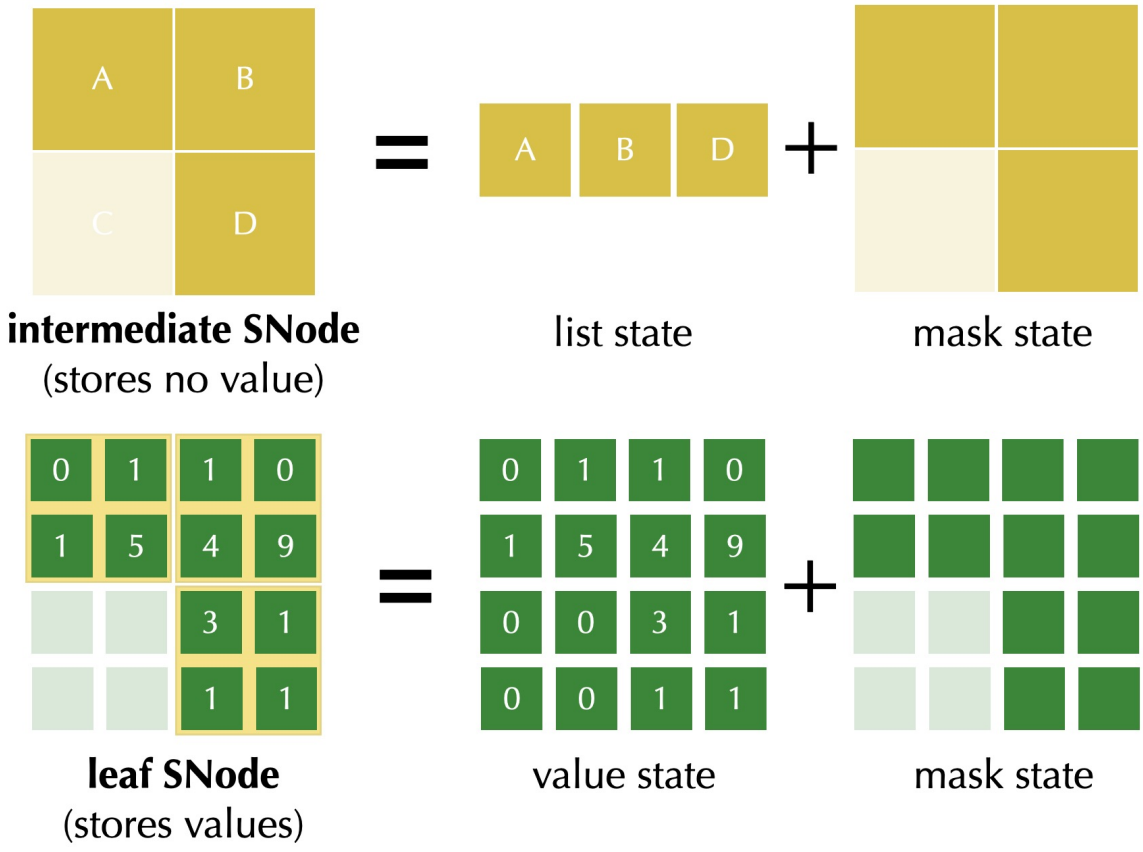


Figure 4-4: State decomposition of a two-level sparse array, containing a sparse intermediate layer and a dense leaf layer. Note that the value state covers all pixels, even if the pixel is inactive. In other words, whenever an access reads a pixel from the sparse array, the mask state will first be queried. If the mask state says the pixel is inactive, 0 will be returned. Otherwise, the system queries the value state and returns the corresponding value. Here we omit allocator states for simplicity.

single task and generates a single GPU kernel[1]. Each Taichi task has input edges (input states), output edges (modified states). It also maintains its metadata, such as loop ranges (range/struct-fors). These edges and metadata will be used for inter-kernel optimization.

4 .1 State-flow chains

Now let us focus on a single state. For example, we use value state S (Fig. 4-5), which is manipulated by kernels (tasks) f, g, p, h, q. Note that f, h and q read and write the value state S, yet g and p only reads the value of S. Every time we modify a state, a new "copy"[2] is created. Clearly, only the latest writer holds the most up to date version of a state, while readers only fetch the latest version without making a new copy. If we only consider the writers, we get a chain structure for each state, with a few branches for readers. Fig. 4-5 provides a concrete example.

It is worth pointing out that the ϕ node in the SSA form is not needed in SFC. In the execution model of Taichi, the kernel-level control flow is directly evaluated inside the host language (Python), rather than being part of the DAG.

For a single state, we can easily build a chain (which is also a DAG). We call the chain structure a "state-flow chain" (SFC).

4 .2 State-flow graphs

A Taichi program can easily have hundreds of states. Here we introduce state-flow graphs (SFGs), which are essentially state-flow chains sticking together, or unioning their nodes and edges (Fig. 4-6). SFGs completely describe the relationship between tasks in Taichi. Since unions of DAGs following the same topological order are still DAGs, SFGs are DAGs too.

The SFG serves as the IR for inter-kernel optimizations. The SFG formulation

[1]We use the term "task" and "kernel" interchangeably for a serial/parallel execution job on GPUs.

[2]Note that in imperative programming, the modifications are actually applied in place, yet for optimization purposes we assume that we always create a new virtual copy.

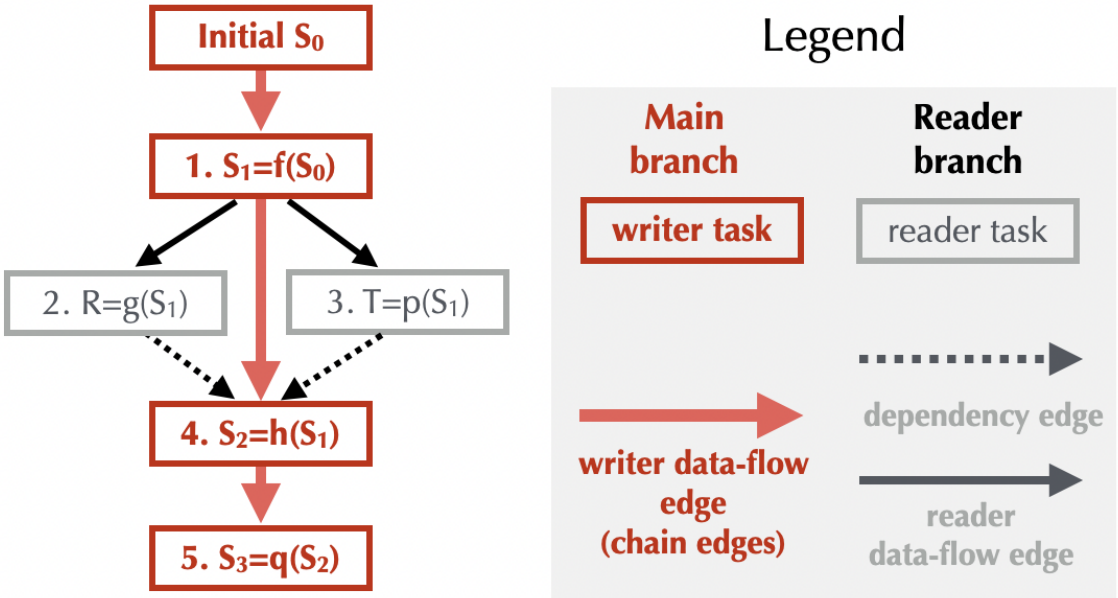


Figure 4-5: A state-flow chain of value state S. The edges in the state-flow chain depict the task dependency relationships. Note that each state-flow chain always has a main branch (write-after-write, red in the figure) and a few reader branches (read-after-write & write-after-read). On the main branch, each node (task) creates a new version of the state. We classify write-after-write and read-after-write as data-flow edges, since there are data produced and consumed. Write-after-read edges are classified as dependency edges.

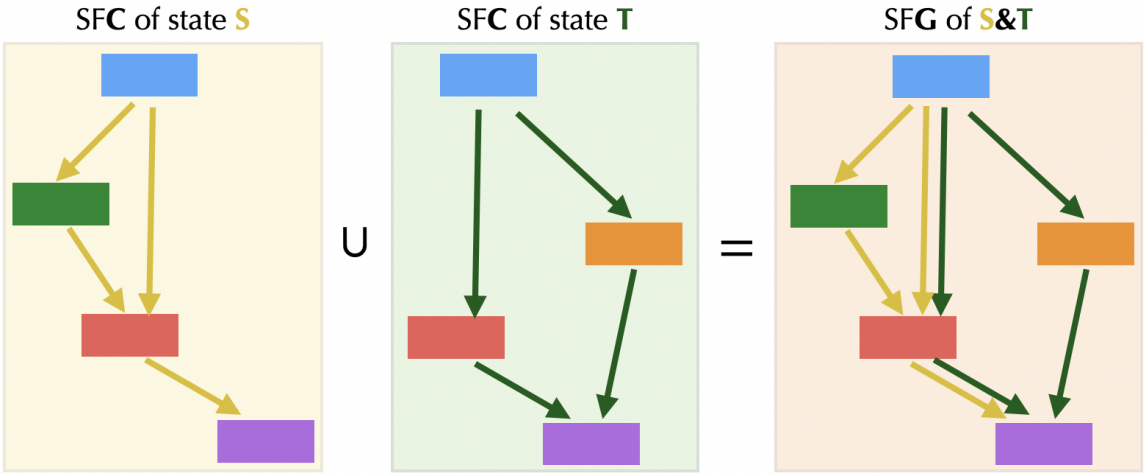


Figure 4-6: A state-flow graph, by definition, is a union of state-flow chains of all the states used in a program. Note that each edge represents a state and a node represents a task. Two tasks may be connected by more than two edges, each edge representing a state.

allows us to use well-established graph theory languages for compiler optimization. For example, our task fusion optimization uses reachability analysis in graphs (section 6 .4).

Whenever a task is inserted into the execution queue, we dynamically create an SFG node and create the corresponding dependency edges. SFGs have two useful properties:

1. **Order independency**. Any topologically ordered task sequence leads to the same program behavior.

2. **Reconstruction invariance**, corollary of "order independency". Any topologically ordered task sequence of G constructs the same graph G.

"Reconstruction invariance" is particularly useful when manipulating the graph nodes. For example, to remove a node from SFG, simply topologically sort the SFG nodes, remove the node from the sorted list, and rebuild the SFG. This frees us from worrying about how to handle edges that are connected to the removed node, or to update the latest set of owners of the affected states in the system.

5 Lazy and asynchronous GPU kernel launches

In existing parallel programming languages such as CUDA, kernels are ahead of time (AOT) compiled and launched immediately once called on the host[3]. However, we need two more execution mechanisms to make inter-kernel optimizations work: just-in-time (JIT) compilation and (kernel-level) lazy evaluation.

JIT compilation The issue with AOT compilation is that, at launch time, optimizers only have access to low-level assembly code (e.g., PTX or SASS), which is too fine-grained and fragmented for further optimizations. Fortunately, Taichi not only provides a JIT system, but also allows to lower the IR halfway to a level

[3]Existing GPU programming systems such as CUDA and OpenGL already provide some asynchrony between the CPU host and GPU devices, but we need more asynchrony for inter-kernel optimizations, as described later in this section.

that is very suitable for inter-kernel optimizations (see Fig 5-1, bottom left, where inter-kernel optimizations happen at the middle-end IR).

Asynchronous launching In CUDA, GPU kernel execution is asynchronous, but GPU kernel launches are still eager. The eager launching mechanism prevents cross-kernel optimization from happening, since the system only *sees one kernel at a time*. Therefore, to make the SFG practically useful, we need to hold the SFG nodes from executing before inter-kernel optimizations.

We developed an *asynchronous* execution engine for GPU programs. The existing Taichi system eagerly launches the kernels, but we can modify the system, making it *asynchronous*, and maintain a list of kernels to compile and run *lazily*. This opens up opportunities for inter-kernel optimizations detailed in the following section.

By-product: parallel compilation A drawback of JIT compilation is its compilation time. Note that ahead of time compilation does not have this issue. In fact, as Taichi becomes more widely adopted, the compiler needs to deal with programs with increasing instructions and optimization passes, in extreme cases compilation can take up to 70% of program end-to-end run time. In the previous eager execution scheme, a serial thread is used to compile and launch these kernels. In contrast, since the asynchronous execution engine sees multiple kernels at a time, parallel compilation can be done easily, which can significantly reduce wall-clock time spent on the compilation. The effectiveness of parallel compilation is evaluated in section 8 .6.

6 Optimize across kernel boundaries

With the state-flow graph IR that describes the task relationships, and the asynchronous execution engine that saves the tasks from being executed too early, we can finally conduct analysis and optimizations on the state-flow graph. In this sec-

tion, we discuss four effective inter-kernel optimizations on spatially sparse computation programs.

6 .1 A minimal example

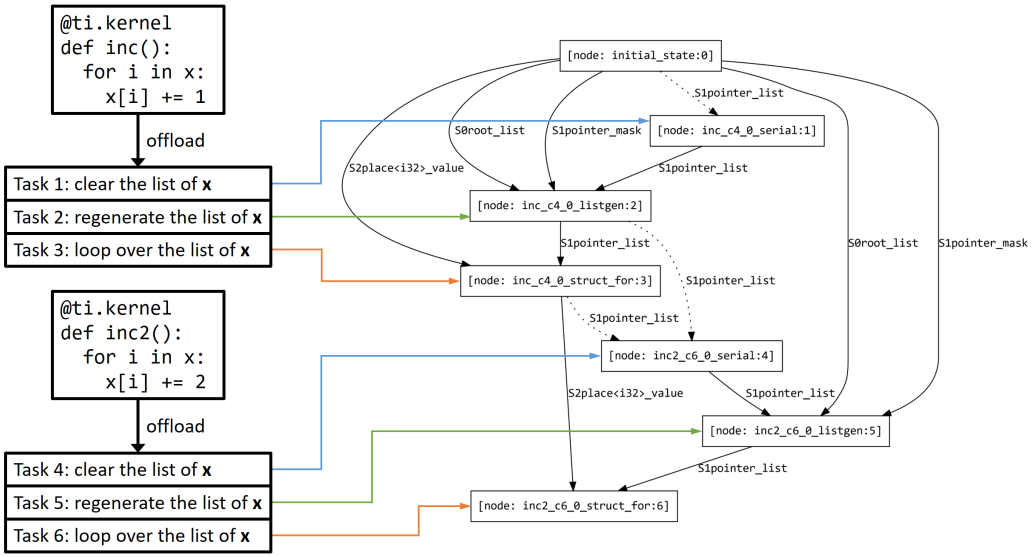
Here we show a trivial optimization example of two Taichi kernels. Note that in Taichi, the "struct-for" construct allows users to iterate over sparse tensors, which needs generating a list of elements before the real computation happens. The list generation itself has performance overhead, which can often be optimized.

As shown in Figure 4-8, since **x** is a sparse data structure, Taichi needs to generate an active list of **x** to know which elements of **x** need to be looped over. So there are 3 tasks per such a small kernel in synchronous mode. If the kernel on the right succeeds the kernel on the left of Figure 4-8, and Taichi performs asynchronous computing, we can perform some analysis to know that the mask state of **x** is not changed, fuse the two kernels into one, and finally get Figure 4-9 after optimizations. In this case, we reduce the number of generated tasks from 6 to 3. Kernel fusion is not new, but fusing kernels that operate on *sparse* data structures is a unique challenge in Taichi, since the iteration over active elements implicitly depends on the mask of the sparse data structures.

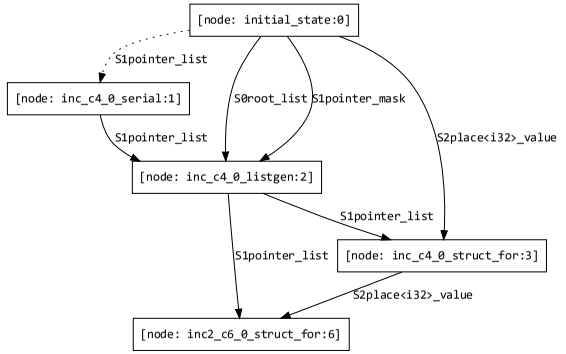
Even if the bodies of both kernels cannot be directly optimized as in this example, we can still remove some list generation tasks and reduce running time. This can be a significant improvement for small kernels where the list generation time is comparable to the real computation time.

Common GPGPU patterns and Taichi's sparse computation model motivates us to apply the following general-purpose and domain-specific compiler optimizations:

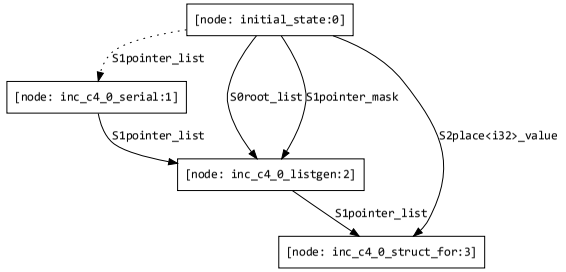
- List generation removal
- Activation demotion
- Task fusion



(a) Generated tasks from two kernels and the corresponding SFG. **x** is a sparse data structure.



(b) The SFG after eliminating list generation.



(c) The SFG after task fusion.

Figure 4-7: State-flow graph optimizations. (a) demonstrates the correspondence between Taichi kernels.

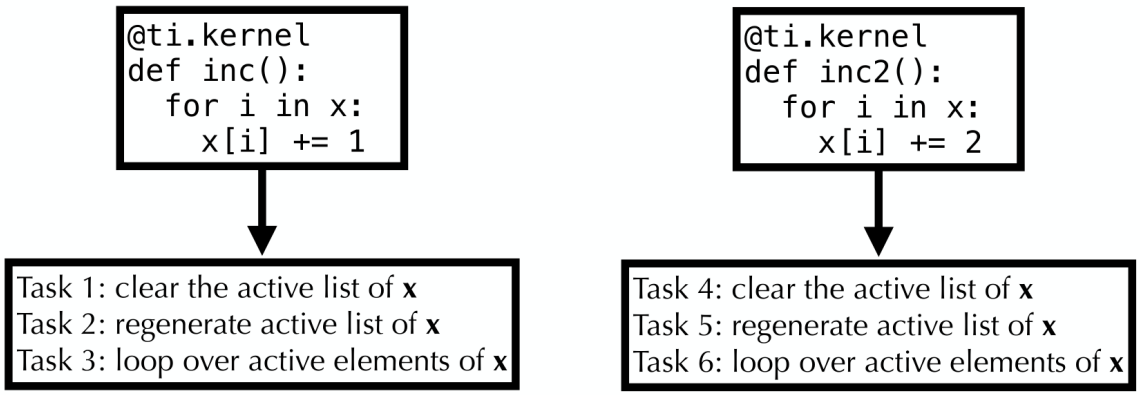


Figure 4-8: The generated tasks of two kernels without kernel fusion. **x** is a sparse data structure.

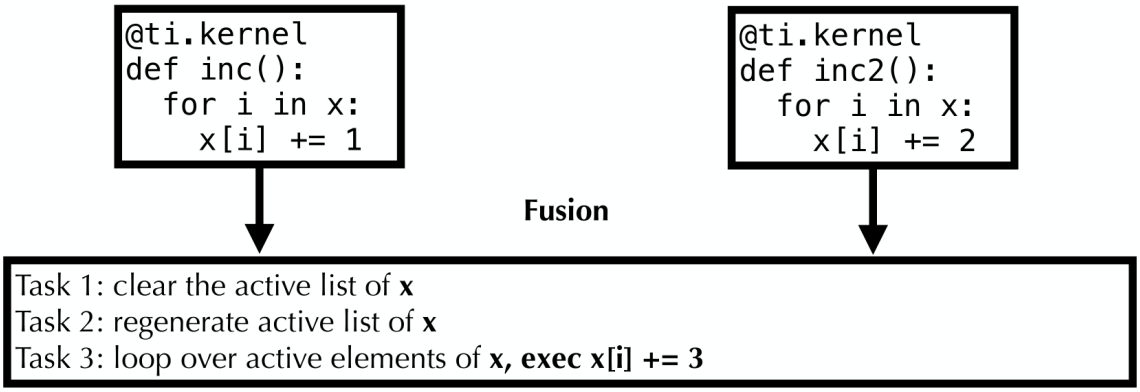


Figure 4-9: The generated tasks of two kernels with kernel fusion.

- Dead store elimination

The remainder of this section details these optimizations.

6 .2 List generation removal

This is the easiest whole program optimization, yet it leads to significantly higher performance for sparse computations in certain cases. A list generation task takes as input a mask and outputs a list. Two list generation tasks with the same parent list and the same mask as the input output the same list, and we can eliminate one of them.

List generation removal not only saves unnecessary execution time on generating the sparse element lists, but also opens up opportunities for other optimiza-

tions. For example, if two struct-for tasks are using the same list after list generation removal, a *task fusion* may fuse the tasks.

6 .3 Activation demotion

Recall that Taichi has an activation-on-write mechanism. However, it is often the case that the sparse element was already activated before the task execution, so the element activeness was checked to avoid unnecessary activation. This extra check not only creates diverging instruction flow on CPU/GPUs that harms the performance, but also creates a modification to the corresponding mask state, creating obstacles for list generation removal. Therefore, we should try to demote activating accesses to non-activating accesses.

Fortunately, many activations can be demoted, by analyzing the task contexts. If two struct-for tasks are identical, the loop lists are the same, and the activation statement in the second task depends only on the loop indices, then the activation in the second task can be removed.

This optimization is remarkably effective for repeated access patterns such as `[i // 2]`. For example, in the restriction (downsample) operator of multigrid solvers, it is common to have the following pattern (Fig. 4-10):

```
for i, j in x:
    y[i // 2, j // 2] += x[i, j] * 0.25
```

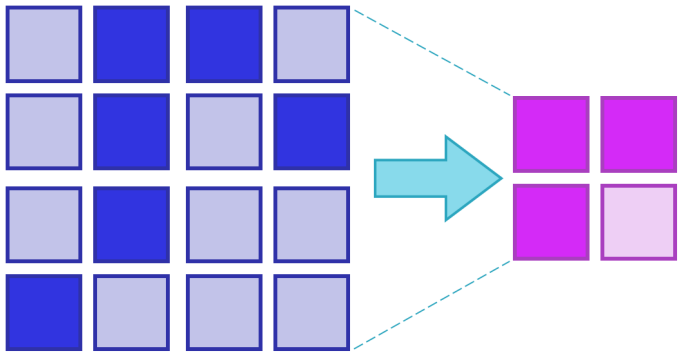


Figure 4-10: The activation pattern of `for i, j in x: y[i // 2, j // 2] += x[i, j] * 0.25`. `x` is the grid on the left, and `y` is the grid on the right.

Our activation elimination optimizer can successfully infer that if the mask of x has not been changed, then the mask of y will not change either. This avoids false-positive mask state modifications, and can further bring down the list generation kernel tasks by $6.7\times$ in the MGPCG example.

6 .4 Task fusion

Clearly, we need to know the data dependency before we fuse. If all tasks are serial: Tasks A and B can be fused if and only if there is no path of length ≥ 2 between A and B. For parallel tasks, fuse only when the loop ranges are the same. If there is an edge A $\rightarrow$ B in the SFG, we need every accesses on that SNode are at the same address, and that address is unique per iteration of the loop.

To find all fusible pairs of tasks, we compute the transitive closure of the SFG using bitsets. For pairs of tasks without edges, we group tasks by the tasks' type, loop range (if the type is range-for), or the SNode (if the type is struct-for). For each group, we use the transitive closure to find which pairs of tasks do not have any path to each other quickly. For each edge A $\rightarrow$ B in the SFG, we check if there is a task C such that A has a path to C and C has a path to B using the transitive closure, and apply the above check to find if A and B are fusible.

This is very effective because we have many intra-kernel optimizations, but it might be time-consuming when there are too many tasks.

6 .5 Dead store elimination

We can also perform some inter-kernel data-flow analysis with asynchronous computing. For example, `ti.clear_all_gradients()` may excessively zero-fill unrelated gradient fields, which can be eliminated with data-flow analysis.

For convenience, a user may frequently zero-fill fields in Taichi to ensure data are correctly re-initialized. This is a typical source of dead stores. For such cases where a field is completely overwritten, our optimizer can eliminate the previous dead store:

```python
@ti.kernel
def clear():
    for i in x:
        x[i] = 0
        y[i] = 0

@ti.kernel
def inc_x():
    for i in x:
        x[i] += 1

    for i in x:
        print(x[i])

clear()
# After DSE, y[i] = 0 in this kernel is elimianted

inc_x()
clear()
```

7 Implementation details

The inter-kernel optimizations are relatively simple to implement, but extra attention was paid to the infrastructure to support these optimizations. In this section, we briefly cover implementation details that we empirically found to directly impact performance.

7 .1 Asynchronous Execution Engine

We implement an asynchronous execution engine that performs SFG optimizations and parallel compilation.

All tasks invoked from the Python side are initially accumulated inside a queue, until either an implicit synchronization event (e.g., data transfer between the de-

vice and the host), or an explicit call to flush (explained in the next paragraph) happens. Upon such events, the tasks are popped off from the queue to construct an SFG instance. This SFG instance goes through the various kinds of optimizations mentioned in Section 6 . Once the tasks in the graph are finalized, they are sent to the JIT compilation workers running in parallel, then the backend device.

Flushing While the more tasks are deferred, the more information is retained for the SFG to optimize, it is usually undesirable to solely depend on the implicit synchronization events to flush the task queue, since this can easily cause starvation on either CPU or GPU side. To provide experienced users with more control over the asynchronous execution engine, we provide a simple API, `ti.async_flush()` , to flush the tasks to the SFG optimizer and then the GPU device. This API is non-blocking, which allows for overlapped execution between CPU and GPU. For most of the usage cases, our system is configured to periodically flush the tasks *automatically*. While this simple strategy could lead to sub-optimal executions, in practice, we have found this to yield sufficient performance. Note that setting the flushing period to 1 effectively turns off the asynchronous execution.

Partial SFG Garbage Collection To further mitigate the loss of information potentially caused by flushing or synchronization, each time an optimized SFG instance is sent for execution, Taichi does a partial garbage collection by preserving those nodes that are the latest owners of the states. As new tasks get launched, these nodes will become the roots in the new SFG instance. This enables the system to capture the information it needs for certain types of optimizations. For example, assuming one preserved SFG node is the latest owner of an SNode's mask state, and a new sparse struct-for loop task reading that SNode is launched. If the mask state has not been modified in between, the SFG optimizer can infer that it is safe to remove the list generation tasks preceding the struct-for task.

7 .2 IR handle and IR bank for caching compilation

Since a kernel can be launched many times with the same IR, we store all IRs into an IR bank to avoid repeated passes on the IR and to improve the asynchronous compilation performance. We use IR handles to access IRs in the bank. An IR handle consists of a pointer to the IR and the hash of the IR. We assign an IR handle to each task, and whenever we are going to do any modification to the IR, we check if we have already done it in the IR bank, where we cache the result of IR optimization passes such as fusion, activation elimination, and dead store elimination. If the result is not cached, we copy the IR on write to avoid corrupting the IR in the bank, do the modification, store the modified IR into the bank, and then cache the mapping from the IR handle before modification to the IR handle after modification into the bank. We also cache some data that do not need to modify the IR into the bank, such as the task's metadata.

7 .3 Intra-kernel data-flow optimizations

To achieve better performance after kernel fusion, we need an optimization pass on the task after fusion. As Taichi IRs are inherently hierarchical, we build a data-flow graph for data-flow analysis, to perform inter-kernel optimizations, including store-to-load forwarding, dead store elimination, and identical store/load elimination. For example, in Figure 4-9, on CPU we demote atomic addition operations into loads, adds and stores, and with store-to-load forwarding, we can replace the load of the second atomic addition (**x[i] += 2**) with the addition result of the first atomic addition (**x[i] += 1**), and get the final result as if the input was **x[i] += 3** with other optimizations.

More details on intra-kernel data-flow optimizations can be found in the Appendix.

8 Evaluation

In this section, we systematically evaluate our system on microbenchmarks and large-scale end-to-end test cases.

Metrics On each test case, we evaluate the performance with five metrics:

1. Wall-clock time (inter-kernel optimizer time included);
2. Backend (GPU or CPU thread pools) execution time;
3. Number of tasks launched;
4. Number of instructions emitted to the code generator;
5. Number of tasks compiled.

Each case is executed multiple times on GPU (CUDA) and CPU (x64), with a synchronization after each run in asynchronous mode, and the total metrics over all runs are recorded.

Benchmark cases We constructed 10 simple yet indicative microbenchmarks (tens of lines of code each) to unit-test specific inter-kernel optimizations. Four complex test cases (hundreds of lines of code each) test the behavior of our optimizer on real-world programs, including computational physics tasks on regular sparse grids (section 8 .2), hybrid Lagrangian-Eulerian schemes (particles and grids, section 8 .3), multi-resolution sparse grids (section 8 .4), and triangular meshes (section 8 .5).

8 .1 Microbenchmarks

We constructed 10 microbenchmark cases to unit-test the system. The results are promising: without code modification, the new system leads to $3.73\times$ fewer kernel launches on GPUs and $2.5\times$ speed up on our benchmarks, as shown in Fig. 4-16. More details on the microbenchmarks are discussed in the appendix.

Table 4.1: Benchmarks against the original Taichi system [49] without inter-kernel optimizations. The baseline system and applications are tuned against state-of-the-art manually engineered CPU and GPU implementations, as detailed in [49]. Benchmarks are done on a system with a quad-core Intel Core i7-6700K CPU with 32 GB of memory, and a GTX 1080 Ti GPU with 12 GB of GRAM. The geometric mean all benchmarks: the wall-clock speed up is $1.87\times$ (CUDA) / $1.33\times$ (x64) and the reduction of task launched is $4.02\times$. Commands to reproduce all the numbers are included in sections detailing each experiment.

Cases		**Backend**	**Wall-clock time (s)**		**Backend time (s)**		**Tasks launched**		**Instructions emitted**		**Tasks compiled**	
			Ref.	Ours	Ref.	Ours	Ref.	Ours	Ref.	Ours	Ref.	Ours
MacCormack		CUDA	8.497	2.907	8.479	2.874	4726	319	16880	6277	96	15
		x64	206.115	73.520	206.065	73.468	4726	319	16880	6277	96	15
MGPCG	2D	CUDA	9.185	3.690	7.799	2.101	1057560	224568	3387	3816	204	105
		x64	23.640	20.468	20.970	19.841	1057560	224570	2961	3331	204	106
	3D	CUDA	9.352	6.500	8.960	6.244	304392	63869	3652	4450	146	82
		x64	172.599	161.927	171.135	161.607	304392	63869	2728	3200	145	77
MLS-MPM		CUDA	14.059	10.633	13.987	10.584	18806	9201	8900	20008	122	91
		x64	292.615	280.415	292.307	280.376	18806	9201	9132	20492	122	91
AutoDiff		CUDA	2.256	1.377	2.124	1.181	65674	42454	1346	2184	22	32
		x64	60.947	53.278	59.308	53.159	65674	42454	1353	2174	22	30

Table 4.2: Geometric mean results of the 10 microbenchmark cases. Numbers are ratios between the reference system [49] and ours with inter-kernel optimizations.

Metric	CUDA	x64
Wall-clock time	$2.30\times$	$2.14\times$
Backend time	$2.56\times$	$2.32\times$
Tasks launched	$3.73\times$	$3.69\times$
Instructions emitted	$0.97\times$	$0.97\times$
Tasks compiled	$1.15\times$	$1.15\times$

8 .2 MacCormack advection

In this benchmark case, we use the MacCormack advection scheme [109] with RK3 path integration. We follow the recent trends to use collocated grids (see, e.g., [92, 34]) to improve cache line utilization. Our benchmark was carried out on a 3D unit grid of 256^3 cells. We advect three scalar physical fields over a sparsely populated vector field defined over a tube domain with inner radius 0.32 and outer radius 0.45. The statistics of the first five frames are discarded to exclude the effects of the JIT compilation. Our optimizer is able to achieve approximately $2.92\times$ and $2.80\times$ performance boost on CUDA and CPU, respectively. This is very close to the theoretical $3\times$ acceleration. Note that the program is memory-bound, when the three physical fields are adjacent in memory and are advected together, which improves cache line utilization by $3\times$. The number of launched tasks is reduced by $14.8\times$ on both backends. Compared to manually fused advection on different channels, our system automatically detects fusible patterns. This provides more coding flexibility and reduces the mental burden on developers.

We present an ablation study of four inter-kernel optimization passes in Table 4.3. The improved performance and the reduced number of launched tasks in this benchmark mainly attribute to the task fusion and the list generation removal optimization.

Table 4.3: An ablation study on the MacCormack advection benchmark. The main optimization comes from task fusion. In this case, disabling list generation optimization transitively disables fusion, therefore degrades the performance.

Ablation	**Tasks**	**Wall-clock/GPU time (s)**
Reference [49]	4726	8.498/8.479
No list generation removal	3781	8.542/8.488
No activation demotion	319	2.899/2.867
No task fusion	950	7.990/7.959
No dead store elimination	319	2.906/2.869
All optimizations on	319	2.907/2.874

8 .3 Moving Least Squares Material Point Method (MLS-MPM)

To evaluate our system in more challenging cases where both particles and grids are used, we benchmarked against [124]. As shown in Fig. 4-11, we seed a uniform cube of $16,777,216$ (256^3, on GPU) or $2,097,152$ (128^3, on x64 CPUs) fixed corotated particles into a grid of size $1 \times 1 \times 1$, with the distance between each two adjacent particles $1/512$. Each particle is a cube of length $1/512$, with $\rho = 10^3, E = 2 \times 10^4, \nu = 0.4$. The gravity is $g = 9.8$. Each frame consists of 400 substeps, with time step size $\Delta t = 10^{-4}$s. We benchmark the time of the second frame (i.e., the 401st to the 800th substeps). The ground is set to be at $z = 1/32$, and to make the cube touch the ground in the second frame, we set the initial z-coordinate of the center of the bottom-most particles to be $49/1024$. The x and y coordinates of the center of the cube are set to be the center of the grid.

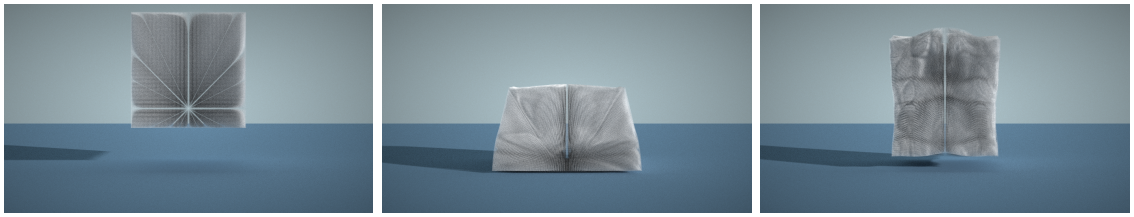


Figure 4-11: The falling cube in our MLS-MPM benchmark. The cube's initial position is higher than the one we used in the benchmark for better visualization.

To achieve the best performance, we use two grids and swap them before each

substep. We also hint to our optimizer that the values stored in the `pid` (particle ID) field is a permutation of all active indices in the `particle` SNode. Then our optimizer fuses G2P and P2G into a single G2P2G task, which is the main source of optimization. The dead store elimination further analyzes that the values stored to the `C` field (affine velocity around the particle, see [60]) in the G2P2G task are never used, leading to further improved performance. Our system achieves a $1.32\times$ performance boost on CUDA over the original Taichi system. The performance speedup due to fusion matches the originally reported number in [124][4].

An ablation study of four inter-kernel optimizations is listed in Table 4.4. In the microbenchmarks, there is a small-scale MPM test case (`mpm_splitted`) that simply fuses per-particle operations and grid boundary conditions, leading to $1.1\times$ speed up.

Table 4.4: An ablation study on the MLS-MPM benchmark. The main optimization comes from task fusion and dead store elimination. Note that without list generation removal, we cannot perform task fusion, and other optimizations will take much more time.

Ablation	**Tasks**	**Wall-clock/GPU time (s)**
Reference [49]	18806	14.059/13.987
No list generation removal	14006	15.059/13.908
No activation demotion	9201	10.976/10.927
No task fusion	11608	13.835/13.786
No dead store elimination	9201	12.589/12.541
All optimizations on	9201	10.633/10.584

8 .4 Multigrid preconditioned conjugate gradients (MGPCG)

Multigrid algorithms have frequent sparse data structure operations on grids of multiple resolutions. We test our system on a complex MGPCG Poisson solver. We

[4]Note that even in our implementation with G2P2G, Wang et al. [124] (wall-clock time 5.004 s on CUDA) is still $2.12\times$ faster than our system, because of their AOSOA acceleration data structure, which is outside the scope of this work. In their hand-engineered CUDA version, AOSOA+G2P2G is $2.1\times$ faster than G2P2G, which aligns well with the observations on our system.

use a sparsely populated region in a 1024×1024 (2D) and $256 \times 256 \times 256$ (3D) grid. We follow the MGPCG solver design in [49]. In the 2D case, for example, our optimizer is able to bring down the number of tasks launched from $1,057,560$ to $224,568$ (only 21% of the original number). This is because the restriction, smoothing, and prolongation operations lead to $420,912$ redundant list generation tasks, which are reduced to 8004 (2% of the original) by our list generation removal and activation demotion (Fig. 4-12). Note that the CUDA speed ups ($2.49\times$ in 2D and $1.44\times$ in 3D) are much higher than the x64 speed up ($1.12\times$ in 2D and $1.07\times$ in 3D), likely because parallel task launches on GPUs are relatively more expensive than that on CPUs, and the majority of the speed ups in this benchmark case is from eliminating small kernels such as list generation and clearing.

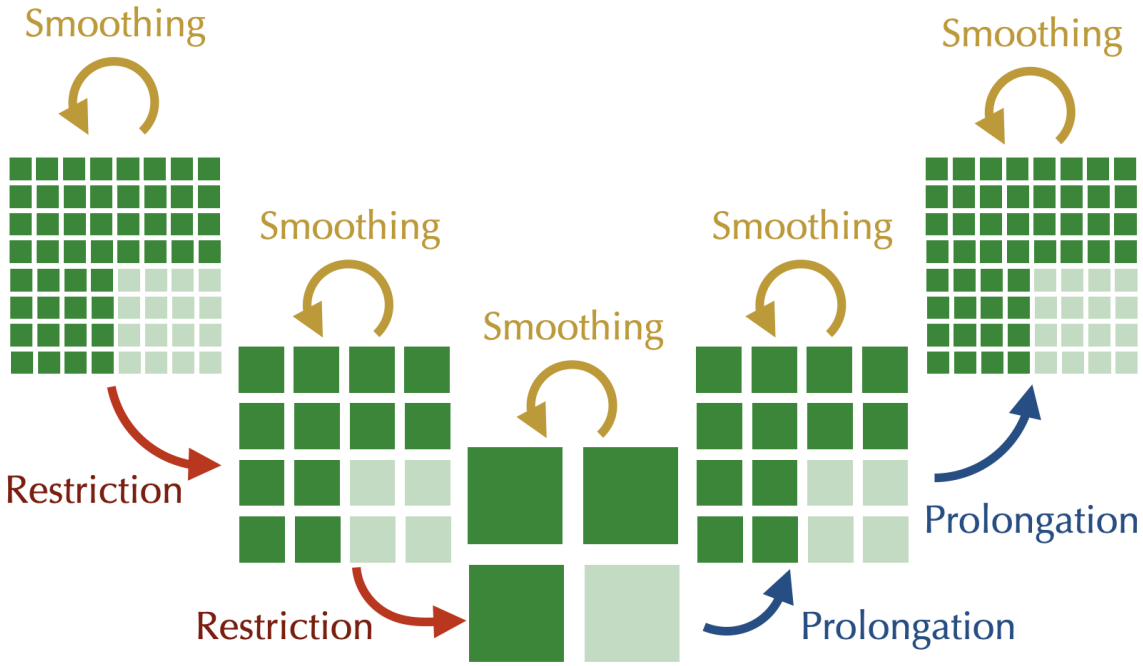


Figure 4-12: The key computational patterns in one iteration of the multigrid preconditioner. Note that in a single Poisson solve, once the sparse multigrid hierarchy is initialized, it will never change its topology. Our optimizer is able to infer the this property, since activation demotion will learn that the restriction kernels, starting from the second iteration, does not additionally activate any new voxels. The majority of list generations can also be removed.

To evaluate our system on problems of different scales, we scan the problem size and plot the wall-clock time and backend time in Figure 4-13. Most of the

optimizations come from eliminating fragmented kernels caused by list generation. As the problems scale up, these kernels occupy relatively less time, hence we observe the performance boost to be more significant at lower resolutions. An ablation study of four inter-kernel optimization passes is listed in Table 4.5.

Table 4.5: An ablation study on the 2D 1024×1024 MGPCG benchmark. The main optimization is list generation removal. Note that activation demotion helps further eliminate 20% of the list generation tasks

Ablation	**List / Total tasks**	**GPU time (s)**
Reference [49]	420912/1057560	7.776
No list generation removal	420912/827303	7.232
No activation demotion	10524/227104	2.213
No task fusion	8004/231695	2.156
No dead store elimination	8004/224585	2.098
All optimizations on	8004/224565	2.098

8 .5 AutoDiff: nodal forces from energy gradients

We implemented MLS-MPM [48] with Lagrangian forces [60]. In the simulation, the structure is modeled using a mesh of 160K triangles, and a NeoHookean hyperelastic model (Fig. 4-14). The force $\mathbf{f}_i$ on the particle i is by definition

$$\mathbf{f}_i = -\frac{\partial L(\mathbf{x})}{\partial \mathbf{x}_i}.$$

Since manually deriving the partial derivative on the right hand side is error-prone, we rely on Taichi's automatic differentiation system [47]. The key optimization opportunity is the following code:

```
with ti.Tape(total_energy):
    compute_total_energy()
```

The code above does the forward computation of total energy $L(\mathbf{x})$, and then automatically evaluates for `x.grad`, which is essentially $\frac{\partial L(\mathbf{x})}{\partial \mathbf{x}_i}$. In the majority of the

Table 4.6: An ablation study on the AutoDiff benchmark. The main optimization comes from dead store elimination. Fusion helps, to some extent, when evaluating and clearing the gradients. Since the data structures are all dense in this benchmark, list generation and activation demotion lead to no performance boost.

Ablation	**Tasks**	**Wall-clock/GPU time (s)**
Reference [49]	65674	2.256/2.214
No list generation removal	42454	1.386/1.191
No activation demotion	42454	1.383/1.188
No task fusion	54064	1.411/1.214
No dead store elimination	48424	2.305/2.082
All optimizations on	42454	1.377/1.181

cases, the result of the total energy L is not used, so by looking at a long window of kernels, our optimizer can automatically eliminate the forward computation, only doing the backward gradient evaluation. Inter-kernel dead store elimination plays the most important role in this benchmark case.

An interesting observation is that our system gets a significantly higher speedup on CUDA than x64. This is because the particle-to-grid (P2G) transfer step plays different roles in the total time consumption. Note that P2G requires atomic add, which is a relatively cheap operation on CUDA (native hardware support) yet expensive operation on x64 (needs software compare and swap). As a result, when our inter-kernel optimization is on, P2G takes 51% run time on x64, yet only 7% on CUDA. This means the forward total energy computation, which is optimized out, occupies a smaller fraction on x64 (since P2G remains the bottleneck), hence a smaller speedup.

8 .6 Discussions

Productivity An attractive feature of our system is users get a performance boost for *free*, simply by turning on an option to let the system conduct inter-kernel optimizations.

Parallel compilation Delay caused by compilation time may lead to potential performance issues in JIT systems. Fortunately, in our asynchronous execution engine, we have a whole buffer of kernels to compile, and parallel compilation significantly reduces this compilation delay. For example, in the 2D 1024×1024 MGPCG benchmark, we find the time of the first iteration, which is compilation delay as no kernels are compiled and cached, improves from $9.0s$ to $3.7s$ after switching to the asynchronous engine, a $2.43\times$ improvement. See Fig. 4-15 for a visualization of the multithreading behavior of the asynchronous optimization and execution engine.

Behavior on CPUs The motivation of our system is to accelerate GPU computation, but since Taichi also has CPU backends, we tested our system on CPU too. Interestingly, on CPU, the performance boost is less significant compared to GPUs. Initial investigations show three reasons:

1. When using the CPU backend, the optimizer and executor share the same processor, meaning the optimization process itself may slow down the execution. This issue does not exist on the GPU backend, since optimization overlaps with the GPU kernel execution time.

2. On CPU computation itself occupies a bigger fraction of the execution time. For example, in the AutoDiff example, we find 51% of the execution time was spent on scattering force contributions to nodes, which needs expensive software-emulated atomic add. On GPUs, atomic add is hardware-native and is much faster. When the task is fully memory-bound, e.g., the MacCormack advection benchmark, we do achieve a close to ideal $3\times$ speed up on CPUs, similar to the behavior on GPUs.

3. Some of our performance boost comes from eliminating kernels. Compared to the actual computation, kernel launching is expensive on GPU but relatively cheap on CPU.

Still, we believe our system to be a plus for programmers running parallel programs on CPUs.

Wall-clock time v.s. backend time In most cases, our wall-clock time is within 103% of the execution time, which indicates that our multi-threaded optimization and compilation system is able to keep GPU busy. However, in the 2D MGPCG example, we find wall-clock time to be $1.75\times$ higher than the execution time. This is an engineering limitation of our work: some optimization passes, such as kernel fusion, still takes a longer time than expected on relatively small-scale problems with many tasks. We believe a more carefully engineered optimization system can get rid of this issue.

9 Conclusion

We have presented an inter-kernel optimization system or parallel imperative programming, with domain-specific optimizations for spatially sparse programming and differentiable programming. In our test cases, we get $1.87\times$ wall-clock time performance improvement on GPUs, without users changing any computation code. We believe our system can alleviate the low-level performance burden on GPU programmers, and serve as an infrastructure for future work on developing high-performance systems for GPU programming in computer graphics.

Future work While our inter-kernel optimizer delivers satisfactory results on spatially sparse and differentiable computation, it does not automatically utilize kernel-level information on other computation patterns, such as sparse linear algebra [68] and graph computations [133]. Exploring the SFG model on other domain-specific computation is meaningful future work. Creating a "general SFG" that serves various DSLs is an exciting future direction. In spatially sparse computation itself, there are many unexplored compile-time metadata we can extract. For example, a colored Gauss-Seidel solver may only use the "white" cells in a checkerboard pattern. These features may enable further inter-kernel optimization opportunities in physical simulation and numerical linear algebra.

Appendix for chapter 4

Microbenchmark cases

Here we describe the cases in the microbenchmarks.

The case chain_copy contains 2 kernels `y[i] = x[i] + 1` and `z[i] = y[i] + 4`, like Figure 4-8. They are fused in asynchronous mode.

increments contains 10 inc() kernels in Figure 4-8.

fill_array contains 10 kernels, all filling a 1-D dense array with the same constant value. With task fusion, only 1 task is launched in these cases instead of 10 tasks. The running time is nearly 10x faster.

sparse_saxpy contains some kernels performing saxpy (Scalar Alpha X Plus Y) operations among sparse tensors. The performance boost of execution time comes from the elimination of list generation and task fusion. Sometimes the wall-clock time is slower than the synchronous mode because of the overhead of the asynchronous engine.

autodiff computes a loss function as reduction on an array and accumulates the gradients to another array 10 times. With dead store elimination, the forward tasks computing the loss function should be eliminated except for the last one, so the number of launched tasks reduces by roughly a half.

stencil_reduction performs stencil and reduce operations on a tensor. They are common operations in computer graphics.

mpm_splitted contains some substep() kernels in an MPM program [48].

simple_advection performs semi-Lagrangian advection 10 times. The performance boost comes from activation elimination and task fusion.

multires is a multi-resolution program downsampling in 4 levels.

deep_hierarchy contains 5 jitter() kernels `x[i] += x[i + 1]` when `i % 2 == 0`. The tasks are not fusible, but we can still get some performance boost by eliminating list generation tasks.

Intra-kernel data-flow optimizations

We apply the traditional control-flow analysis to optimize within kernels. We build a control-flow graph along with the hierarchical IR, and perform analysis on the graph. With the help of control-flow analysis, we perform optimizations including store-to-load forwarding, dead store elimination, and identical load/store elimination. These optimizations motivate task fusion as it greatly simplifies fused tasks.

We also utilize control-flow analysis to help compute task meta information. Since stores to a SNode may only partially modify a value state, the resulting value state (which contains the modified and unmodified part) may need a read from the previous version of the value state. We use control-flow analysis to detect which SNodes do not need a read from the previous version of the value state.

Figure 4-17 shows the effect of data-flow optimization on 360 Taichi test cases. Although these test cases are relatively simple, data-flow optimization still leads to 16% fewer instructions.

Key source code files

We partially reuse the existing Taichi system, with key modifications listed below:

- The SFG data structure and some optimization passes are implemented in `tahici/program/state_flow_graph.[h/cpp]`;
- The asynchronous execution engine is implemented in `tahici/program/async_engine.[h/cpp]`;
- The IR bank and some of the optimization passes are implemented in `tahici/program/ir_bank.[h/cpp]`.

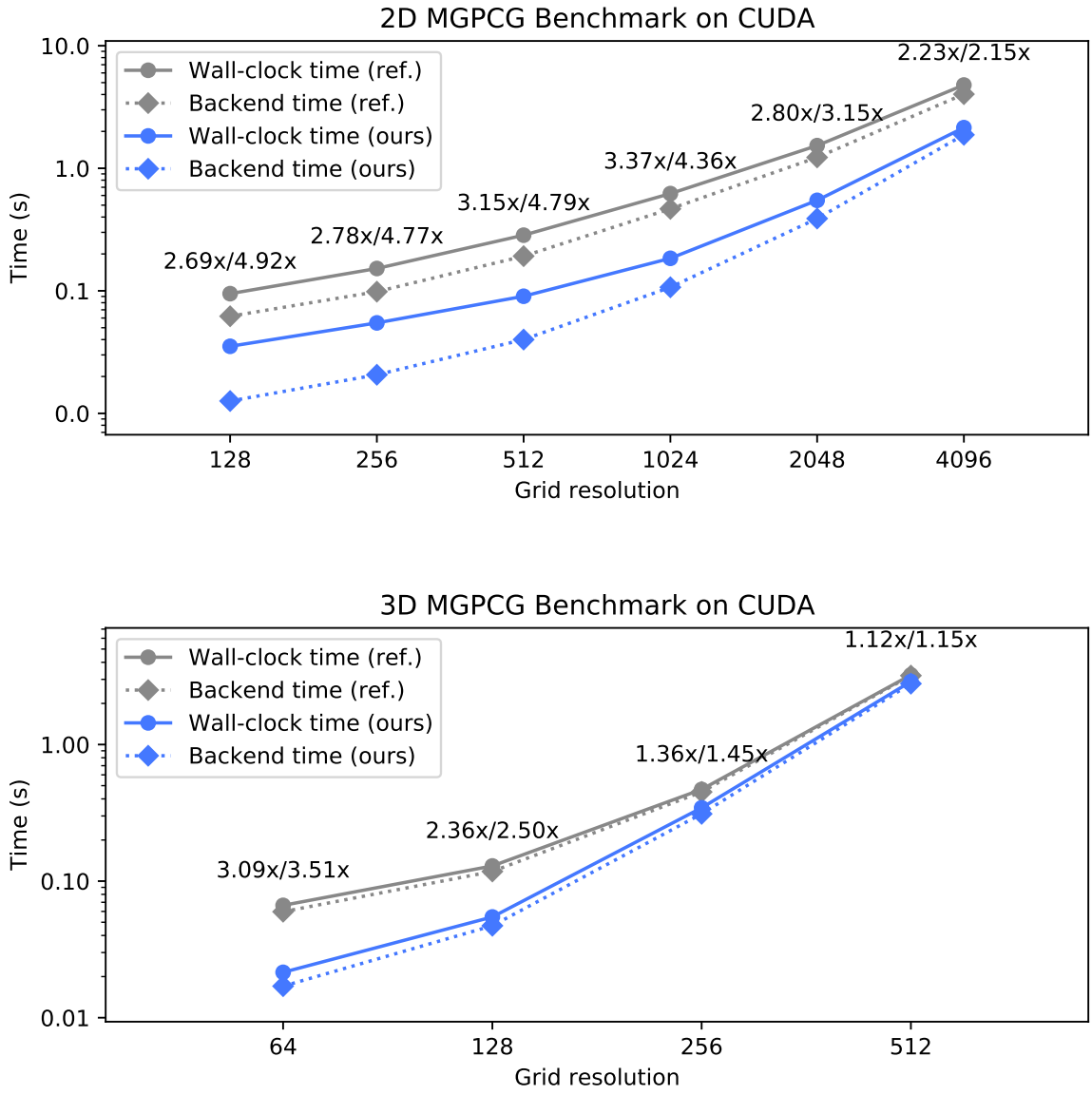


Figure 4-13: MGPCG benchmarks on CUDA. The annotation numbers on each data point indicate the acceleration on *wall-clock* time and *backend* time. Most of the performance boost in this example comes from list generation removal. Since list generation takes a smaller portion of total computation when resolution increases, the gap between ours and reference performance shrinks as resolution goes up.

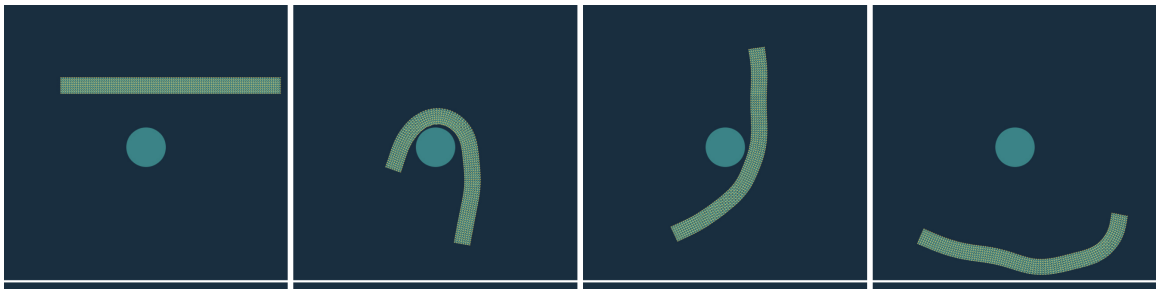


Figure 4-14: The AutoDiff test case: a hyper-elastic material falls down, hits an obstacle, and finally land on the ground. The nodal forces are computed using automatic differentiation. We used a low-resolution simulation here to better visualize the mesh structure.

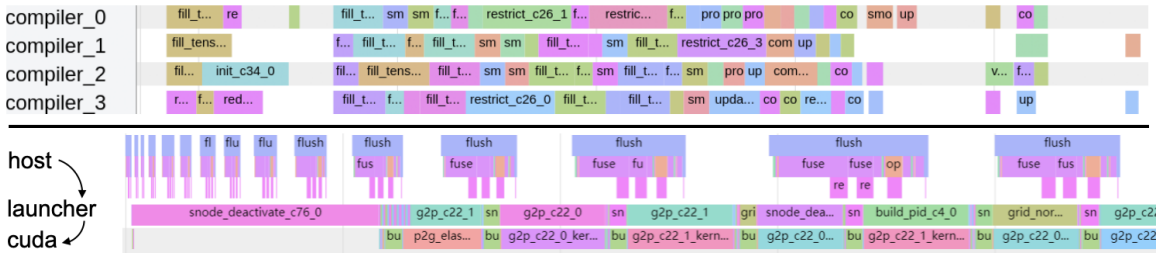


Figure 4-15: **Top:** In our asynchronous execution engine, multiple compilation threads simultaneously compile optimized IR to executable kernels, maximizing JIT benefits. Timelines are gathered from the MGPCG example, where our parallel compilation system leads to a $2.43\times$ shorted program start-up time. **Bottom:** At later stages of program execution, most possible inter-kernel optimized kernels are already compiled and cached, so the compilation threads are mostly idle. Still, our multithreading framework allows the inter-kernel optimizer to overlap with the launcher thread and GPUs, making sure no GPU starvation happens. Timelines are gathered from the MLS-MPM example.

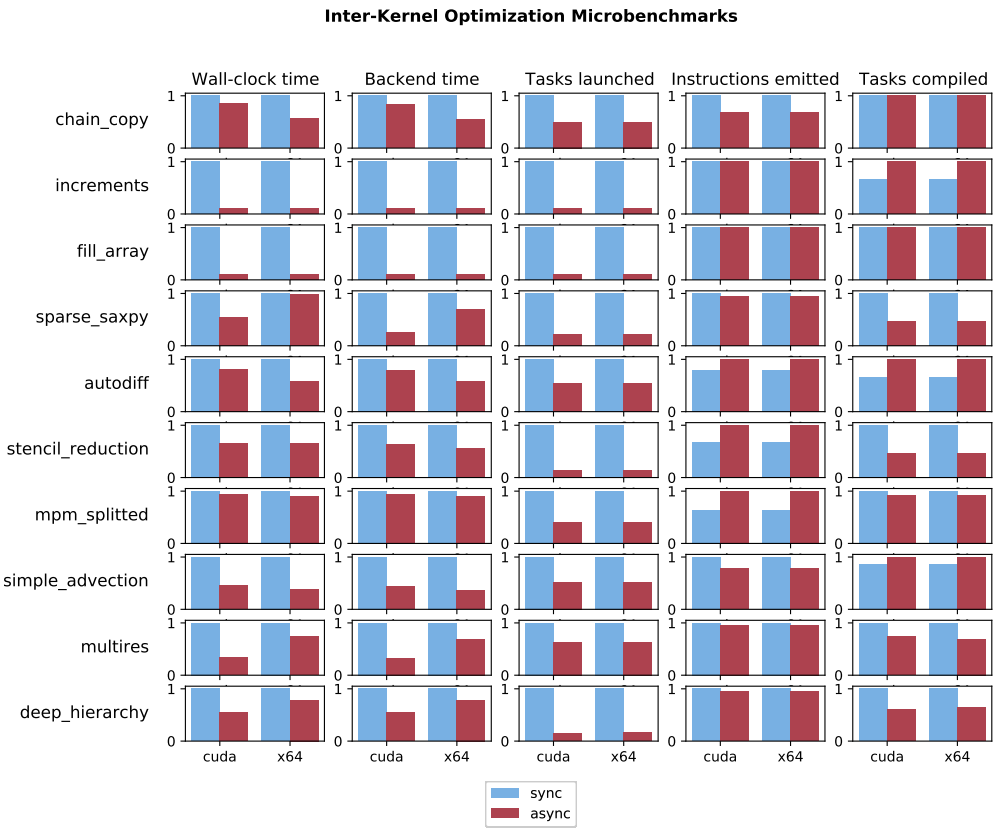


Figure 4-16: Microbenchmarks results. "Sync" refers to the reference system [49]. "Async" refers to our system with asynchronous execution and inter-kernel optimizations.

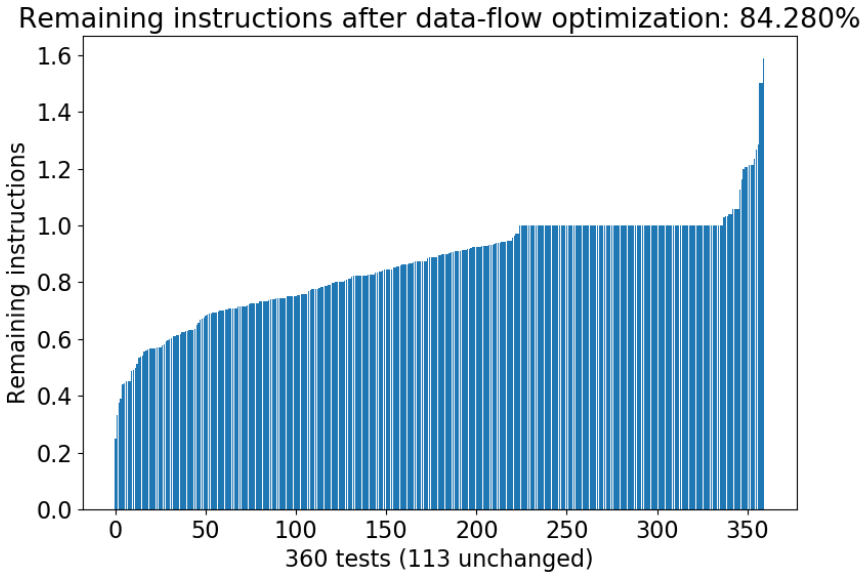


Figure 4-17: Ratio of emitted instructions with/without data-flow optimization among unit tests. On most cases fewer instructions are emitted. The cases with more instructions do happen, because data-flow optimization indirectly triggers some other optimization passes that lead to high-performance but also more (and cheaper) instructions.

Chapter 5

Quantized Computation in Taichi

High-resolution simulations can deliver great visual quality, but they are often limited by available memory, especially on GPUs. We present a compiler for physical simulation that can achieve both high performance and significantly reduced memory costs, by enabling flexible and aggressive *quantization*. Low-precision ("quantized") numerical data types are used and packed to represent simulation states, leading to reduced memory space and bandwidth consumption. Quantized simulation allows higher resolution simulation with less memory, which is especially attractive on GPUs. Implementing a quantized simulator that has high performance and packs the data tightly for aggressive storage reduction would be extremely labor-intensive and error-prone using a traditional programming language. To make the creation of quantized simulation practical, we have developed a new set of language abstractions and a compilation system. A suite of tailored domain-specific optimizations ensure quantized simulators often run as fast as the full-precision simulators, despite the overhead of encoding-decoding the packed quantized data types. Our programming language and compiler, based on *Taichi*, allow developers to effortlessly switch between different full-precision and quantized simulators, to explore the full design space of quantization schemes, and ultimately to achieve a good balance between space and precision. The creation of quantized simulation with our system has large benefits in terms of memory consumption and performance, on a variety of hardware, from mobile devices to

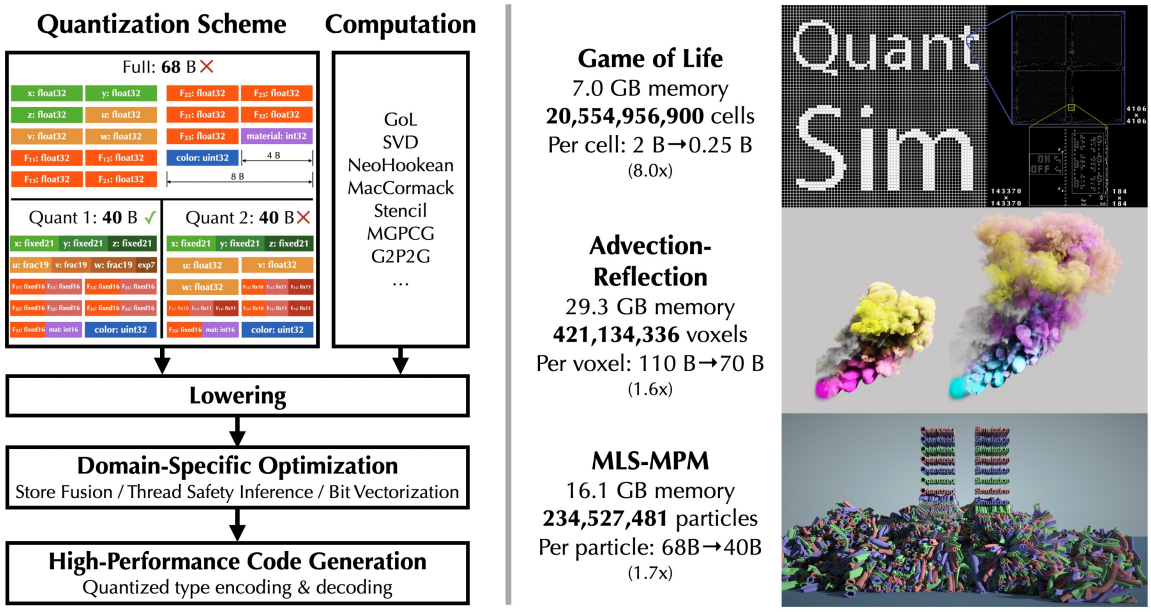


Figure 5-1: **Left:** Our system *decouples* quantization schemes from computation, allowing programmers to improve simulation resolutions using lower-precision ("quantized") numerical data types that consume less memory. Programmers can easily switch between different quantization plans to rapidly explore the design space of quantized simulators, to achieve a good balance between numerical precision and memory consumption. Our compiler is in charge of optimization and code generation. **Right:** High-resolution simulation demos built by our system. All demos here run on a single GPU with $\leq$ 32 GB memory, and each frame takes around one minute.

workstations with high-end GPUs. We can simulate with levels of resolution that were previously only achievable on systems with much more memory, such as multiple GPUs. For example, on a *single* GPU, we can simulate a Game of Life with 20 billion cells, a Eulerian fluid system with 421 million active voxels, and a hybrid Eulerian-Lagrangian elastic object simulation with 235 million particles. At the same time, quantized simulations create physically plausible results. We conducted a user study, showing that humans are not able to distinguish between full-precision and quantized versions of 3D fluid simulations. Our quantization techniques are *complementary* to existing acceleration approaches of physical simulation: they can be used in combination with these existing approaches, such as sparse data structures, for even higher scalability and performance.

1 Introduction

Computer graphics applications, such as physical simulation, require high resolution for visual quality. Unfortunately, as simulations scale up, they often run out of available memory to store the physical states, especially when running on GPUs with hard memory space limits. Existing techniques that scale up simulations are mostly focused on improving computation performance. The space for improving memory efficiency is largely underexploited.

Fortunately, many simulations do not need standard full-precision IEEE 754 data types, such as `float` and `double` in the C programming language. While these general-purpose floating-point formats are usually the only formats supported by processors for computation, we observe that, for storage, we can use more options, including low-bit integers, truncated fixed-point real numbers, and tuples of floating-point real numbers with shared exponents. This directly motivates us to leverage low-precision data types in simulation to save memory space and bandwidth.

While "using fewer bits in data types to save space" sounds like a straightforward idea, doing this robustly and efficiently is a significant challenge. Manually coding programs that operate on low-precision and quantized data types is extremely laborious and error-prone. For example, writing code to decode a low-precision float point number into IEEE 754 `float32` involves numerous low-level bit operations and can easily make simulator code unreadable. Quantized data type libraries may simplify development, but they often result in unsatisfactory performance, since general-purpose compilers may not easily optimize related memory operations that read or write only *part of* a hardware-native integer type such as 32- or 64-bit integers. See Fig. 5-2 for a high-level comparison between different approaches to implementing quantized simulators.

Moreover, determining the right quantization scheme often requires repeated *trial-and-error*. The most effective way to validate a quantization scheme is to implement the simulator and actually run it. Therefore, *flexibly switching between dif-*

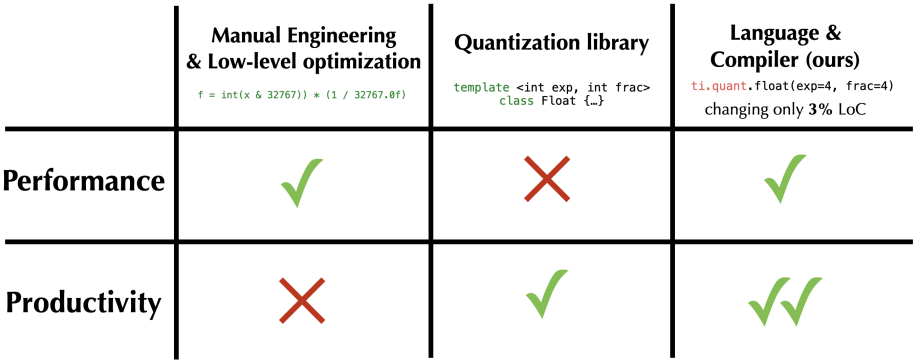


Figure 5-2: Features of different approaches. Our compiler approach aims for both productivity and performance.

ferent quantization schemes is vitally important for practically developing quantized simulators.

We introduce a language and compiler for *quantized simulation*, where low-precision ("quantized") numerical data types are used to represent simulation states, leading to reduced memory space and bandwidth consumption. In our system, developers write simulators as if they are using a traditional parallel imperative programming language, such as C++ and CUDA. When doing memory-space optimization, they do not modify *any* of the computation code. Instead, they use a simple language to specify numerical value quantization schemes and flexibly explore quantized versions of the simulator. Rapid experiments lead to properly compressed simulation states, improved memory space and bandwidth efficiency. Note that in memory-bound programs the overhead of encoding and decoding quantized data types would be negligible, and quantization may improve performance due to reduced memory bandwidth consumption.

Our tailored programming interface and compiler can greatly simplify the development of quantized simulators. Our system provides language-level abstraction and first-class compiler support for quantized computations and domain-specific optimizations. Programmers can easily specify customized and quantized data types for physical state storage. Since our system *decouples* quantization schemes from computation kernels, developers can easily experiment with different low-bit data formats, easily achieve a good balance between precision and

space via rapid experiments.

We summarize our contributions as follows:

1. A simple programming interface for *quantized simulation* that provides programmer bit-level control over numerical data types. The numerical data type interface is orthogonal to the actual computation, this allows the programmers to rapidly experiment with different quantization schemes.

2. A compilation system that automatically generates efficient code for encoding/decoding quantized data types. Our system supports x64, ARM64, CUDA, and Apple Metal backends.

3. A suite of domain-specific compiler optimizations further improves the memory performance of compiled quantized computation. These optimizations bring $4.10\times$ performance improvement on our microbenchmarks and up to $1.58\times$ on the large-scale GPU simulators.

4. Systematic evaluations of our system. We demonstrate that our system pushes the resolution of physical simulations to unprecedented resolutions. Under proper quantization, we achieve $8\times$ higher memory efficiency on each Game of Life cell, $1.57\times$ on each Eulerian fluid simulation voxel, and $1.7\times$ on each material point method [112] particle. To the best of our knowledge, this is the first time these high-resolution simulations can run on a single GPU. Our system achieves resolution, performance, accuracy, and visual quality simultaneously. A user study shows, for 3D simulations, our quantized simulators create almost visually indistinguishable results from full-precision simulators.

Our system is implemented as an extension to the Taichi programming language [49].

2 Related Work

2 .1 Bit-level compression

Compressed color formats in graphics and image processing Compressed data types have proven success in graphics. For example, the 16-bit color format "R5G6B5", where 5, 6, and 5 bits are used to store the red, green, and blue channels of a pixel, was widely adopted for legacy LCDs and graphics APIs (e.g., `GL_RGB565` in OpenGL). The 32-bit RGBE format (used in the RADIANCE rendering system [126]) used 8 bits for red, green and blue channels each, and a shared 8-bit exponent, providing larger dynamic ranges. In modern production rendering, RGBE formats are used for saving communication bandwidth. For example, Eisenacher et al. [31] used an RGB9e5 format for path tracing weights.

Quantized neural networks In deep learning, quantized neural networks (e.g., [25, 52, 56]) and specialized hardware (e.g., [62]) for them have been studied extensively, to use quantized data formats to improve computation throughput. Instead of using single-precision `float32` data type, recent work explores using low-bit data types such as fixed-point numbers, `int8` and even 1-bit integers for deep neural network training and inference. See [39] for a good survey.

Manipulating bits in programming laugnages In programming languages such as C/C++, bit-level compression is often implemented using efficient bitwise operators, such as and "`&`", or "`|`", and xor "`^`". Meanwhile, C++ "bit fields", whose behavior is not yet standardized, can sometimes be used for basic bit-level compression of integral types:

```
struct S {
    // 3 bits: value of x
    // 6 bits: value of y
    // 2 bits: value of z
    unsigned char x : 3, y : 6, z :2;
};
```

Although an extensive study has been conducted in quantized computation, a domain-specific system for productively developing high-performance simulators is missing. As a result, a developer who wants quantized computation has to write low-level code that is hard to maintain or resort to handcrafted libraries with performance issues.

2 .2 Floating-point formats

IEEE Standard for Floating-Point Arithmetic (IEEE 754)[53] serves as the guideline of floating-point format and computation. Notably, the IEEE 754 single- and double- precision floating-point formats (i.e., `float` and `double` in C), have been the prevalent floating-point formats used in computer graphics and scientific computing. They occupy 32 and 64 bits in memory, respectively. Floating-point bits include a sign bit, a few exponent bits, and significand bits. Since computing applications have different preferences between precision, compute throughput, memory bandwidth and space, many variants of float-point numbers do exist. For example, when higher precision is needed, C provides the non-standard `long double` format that usually comes with 80 bits, and the IEEE 754-2008 revision also defines “quadruple" with 128 bits, and “octuple" with 256 bits, both of which are rarely used. In fact, formats with lower precision are more frequently used, a typical example being the 16-bit `half`-precision format (5 bits for exponent and 10 bits for fraction). Recently the Brain Floating Point (`bfloat16`, 8 bits for exponent and 7 bits for fraction) format has been introduced in Google TPUs [61] for deep learning. `half` and `bfloat16` demonstrate interesting trade-offs between dynamic range (number of exponent bits) and accuracy (number of fraction bits, a.k.a. mantissa). See Fig. 5-3 for an depiction of these floating-point formats.

We provide a flexible programming interface for specifying custom floating- and fixed- point data types. Programmers can easily switch between different data types to achieve a good balance between space and precision.

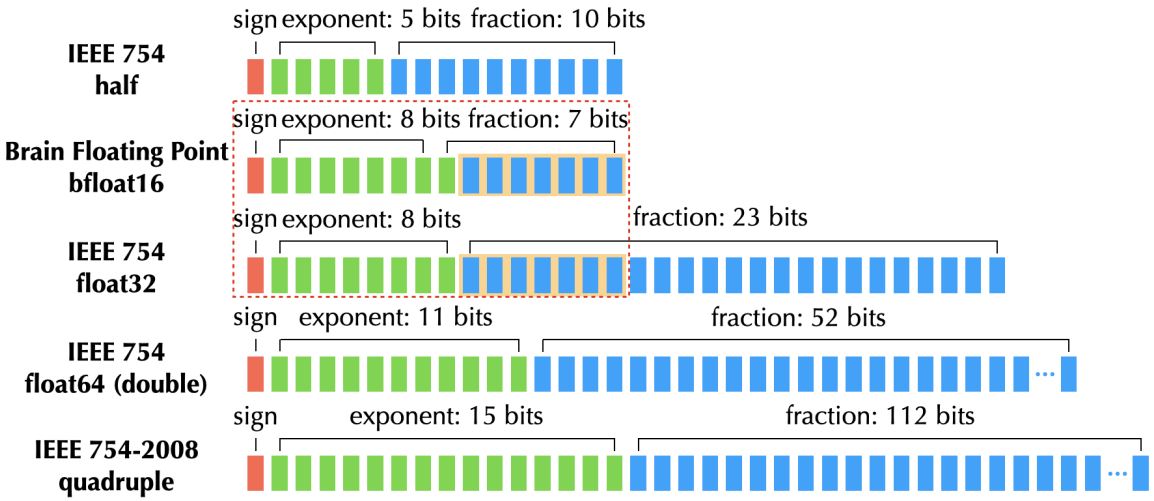


Figure 5-3: Various floating-point data types.

2 .3 High-resolution simulations

Work in graphics explores high-resolution simulation on multicore CPUs [86, 110, 80, 1, 79] and massively parallel GPUs [130, 35, 131, 124]. Corresponding sparse data structures are proposed to support the underlying structured grid, often with certain degree of bit-compression [44, 89, 41, 110]. Most of these attempts are based on manual low-level performance engineering using C++ and CUDA. In our work, we use a programming language approach for scaling up simulations, based on Taichi [49]. We focus on consumer-level computers with a single GPU for simplicity, yet our techniques can be applied to multi-GPU and multi-node settings as well.

Our system practically pushes the limit of simulation resolutions by alleviating the memory space constraints. For example, with the help of quantization, our single GPU MLS-MPM [48] simulation of 235 million particles, has a higher resolution than the existing highest resolution MPM simulation on 8 GPUs (134 million particles, see [124]).

2 .4 Programming systems for simulation

Engineering high-performance simulation systems can be a time-consuming task since a lot of low-level performance engineering is needed to fully exploit the capabilities of modern computer architecture. Domain-specific languages (DSL) play an important role in improving the productivity of simulation systems. One thread of work provides a high-level graph-based abstraction for meshes that discretize the domain. For example, Liszt [30] focuses on solving PDE on meshes. Several domain-specific languages exist for physical simulation. Simit [69] and Ebb [13] represent simulation problems using sparse linear algebra and relational data models. Insightful discussions on DSLs for simulation can be found in [12]. TACO [68, 23] is a sparse linear algebra compiler that can be potentially useful for solving linear systems in simulations.

More closely related to this work is the Taichi programming language [49]. Taichi is a DSL with first-class support for sparse data structures, a critical component of modern high-performance physical simulators. Taichi also supports differentiable programming [47], allowing developers to evaluate gradients of physical simulators for machine learning and optimization purposes. We will briefly cover key Taichi features related to this work.

3 Taichi background

We built our system on top of Taichi [49], our data-oriented programming language designed for simulation applications. We extend its type system, computation and data layout intermediate representation (IR), and code generator. Taichi supports spatial sparsity and differentiable programming [47], and our quantization system is orthogonal to these existing features of Taichi.

We partially reuse the LLVM code generation pipeline in Taichi, for x64 and CUDA on consumer-level desktop computers. Taichi also powers the physics engine on 500 million *mobile devices* in the Kuaishou app, allowing users around the

world to generate AR effects augmented with physics. Therefore, we have also implemented our system on the Apple Metal backend and evaluated its performance on an iPhone (section 7 .4).

The Taichi language has two parts: a computation language and a data layout language. This decoupling allows users to freely explore data layouts without modifying computational kernels. This work further allows programmers to freely switch data types of numerical values. Existing data types supported in Taichi are `ti.f32/f64` (32/64-bit IEEE 754 floating-point number), `ti.u8/16/32/64` (unsigned integers), and `ti.i8/16/32/64` (signed integers)[1]. In our system, users can flexibly define new data types for more compact storage.

Computation language Although the frontend (computation language) of Taichi is embedded in Python, Taichi will inspect the input Python abstract syntax tree and compile them into high-performance executable kernels on parallel devices. Taichi's frontend has a Python-style syntax, enhanced with automatic parallelization, and leads to executables with comparable performance to C++ or CUDA. Two simple Taichi kernels are shown below:

```
@ti.kernel
def saxpy(a: ti.f32):
    for i in x:
        # Parallel for loop over
        # active indices of x
        z[i] = a * x[i] + y[i]

@ti.kernel
def conditional_stencil():
    for i, j in y: # 2D parallel for loop
        if y[i, j] < 0:
            y[i, j] = x[i-1, j] - 2*x[i, j] + x[i+1, j]
```

[1]In this manuscript we will use the same format to refer to these standard data types, for example `f32` or `float32` for 32-bit IEEE 754 floating-point number, `i16` or `int16` for 16-bit signed integer.

The Taichi computation language is expressive: programmers can easily write a ray tracer with `if` branching and `while` loops in Taichi.

Data layout language More closely related to this work is the data layout language and IR. Taichi supports a flexible language to specify data layouts (see [49] for more details). Here we only introduce the concept of a Taichi `field`, which are essentially multidimensional dense or sparse tensors. Each element of a field can be a scalar (e.g., density), a small vector (e.g., velocity), or a small matrix (e.g., stress tensor). No matter how the internal data layout of a Taichi field is defined, in computational kernels (`@ti.kernel`), field elements are always accessed via an `x[i, j, k]`-style syntax, regardless of its data layout. The following code declares a field of `i32` elements, and then materializes along the `i` and `j` axes, each dimension with 32 and 64 elements.

```
x = ti.field(dtype=ti.i32)
ti.root.dense(ti.ij, (32, 64)).place(x)
# Equivalent to int x[32][64] in C
```

4 Quantized numerical data types for simulations

In this section, we introduce *quantized data types*, which are often customized to trade precision for memory efficiency. Our system provides both customized integral data types (lossless) and real data types (usually lossy).

4 .1 Customized integral types

Integral data types are relatively easy to specify since they are simply series of binary bits. For signed integral types, we pick the classical two's complement data format for negative numbers. We provide to the user the following APIs to create signed (default) and unsigned integral types:

```
i5 = ti.quant.int(bits=5)
u19 = ti.quant.int(bits=19, signed=False)
```

4 .2 Custom real types

We offer both fixed-point and floating-point data types. Fixed-point numbers have advantages when their range is strictly bounded in the simulation. For example, when representing x-coordinates of particles, if the boundary condition ensures their values are within $[-2, 2)$, then fixed-point real types can be safely used. They also provide higher precision compared to floating-point types of the same number of bits, since all the bits are used to represent the fraction part. However, the limited dynamic range of fixed-point types can be problematic when handling other physical properties, such as velocity. Therefore, we also offer *floating-point* types, for values with high dynamic ranges.

Fixed-point real numbers This is the easiest way to represent a real number using integral data types. In fact, we directly reuse a custom integral type plus a real scaling factor to represent custom fixed-point numbers. For example, if the range of the fixed-point number is $[-3.14, 3.14)$ and we have 17 bits, the value can be simply represented by $r = s \times i$, where i is a 17-bit signed integer and $s = 3.14/2^{16}$. Note that there is one sign bit in the underlying integer hence we use 2^{16} instead of 2^{17} as the denominator.

Fixed-point real numbers can be specified as follows:

```
fixed17 = ti.quant.fixed(frac=17, range=3.14)
# Range = [-3.14, 3.14)

ufixed5 = ti.quant.fixed(frac=5, signed=False, range=2)
# Range = [0, 2)
```

When the value is known to be always non-negative, the programmer can use `signed=False` to omit the sign bit and allow the fraction part to have one more bit for higher precision.

Floating-point real numbers For real numbers with improved dynamic ranges, we allow exponent bits in real number data types:

```
f18  = ti.quant.float(exp=4, frac=14)
uf22 = ti.quant.float(exp=6, frac=16, signed=False)
```

Same as `ti.quant.fixed`, when we know the stored values must be positive, the user can choose to save the sign bit for one more significand bit and get a higher precision. Note that, different from IEEE754 where the sign bit is the leading bit of the format, in our system we include the sign bit as part of the fraction bits. This is an intentional design to simplify the "shared exponent" case, as introduced below.

Shared exponents In simulations, real values often have physical meanings, and components of a physically meaningful vector typically do not need the same amount of precision, when their absolute values differ a lot. For example, in a 3D velocity vector $\vec{u} = (u, v, w)^T$, if we know the x-component u has much larger (absolute) value compared to y- and z-components, then we probably do not care about the exact value of v and w. This motivates us to use a "shared exponent" for all components, and leave more bits for components with larger absolute values.

We illustrate the internal organization of the real number types in Fig. 5-4.

4 .3 Compute types

Since most of the custom data types are not natively supported on hardware, we usually have to resort to decoding/encoding mechanisms to translate between representations that are storage-friendly and those are computation-friendly (Fig. 5-5, top). This means we have to specify a `compute` type for each custom data type:

```
i21 = ti.quant.int(bit=21, compute=ti.i64)
bfloat16 = ti.quant.float(exp=8, frac=8, compute=ti.f32)
```

By default, the system will use `i32` and `f32` as compute types for integral and real types.

For performance considerations, it may be occasionally profitable to *directly operate on the custom types* without encoding/decoding, especially when the hard-

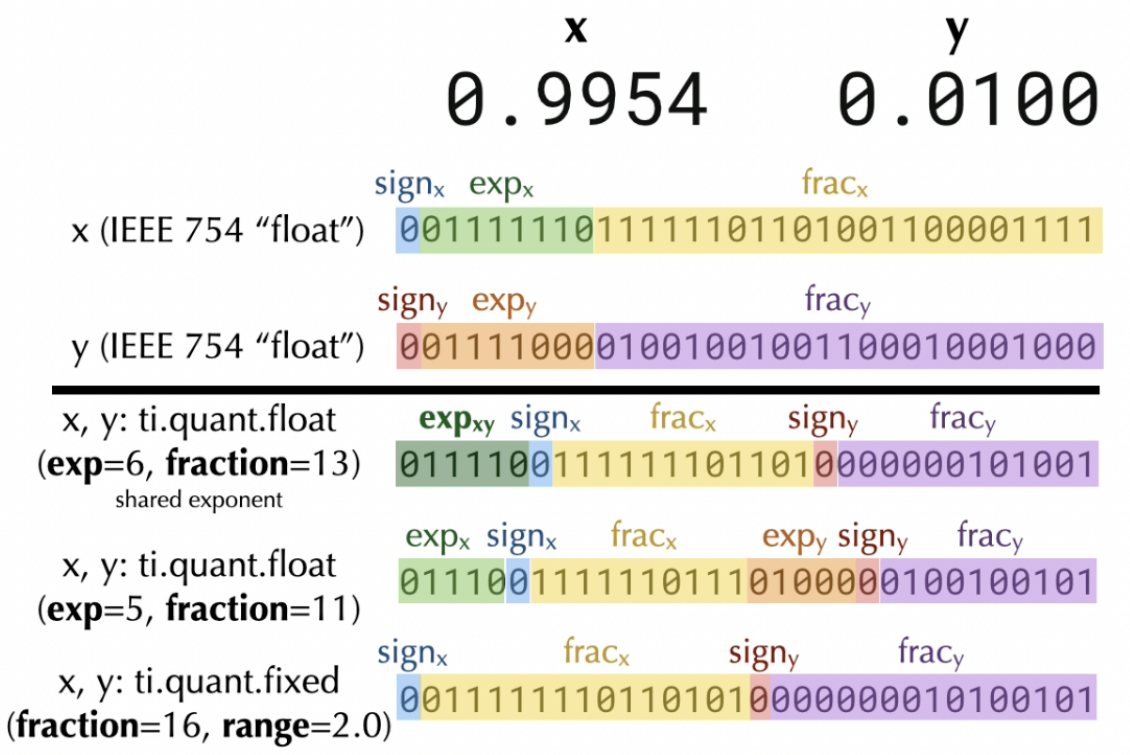


Figure 5-4: Real number types representing a 2D vector (x, y). .

ware supports related operations (Fig. 5-5, bottom). We show a few usages of this type of quantization in section 7 .2 and 7 .4.

4 .4 Bit adapters

Loading and storing data with custom types are typically not natively supported on hardware, so we need two types of *bit adapters* to pack custom data types into hardware supported data types with bit width 8, 16, 32, 64:

- **Bit structs** allow users to pack custom data types into hardware-native types. For example, a `u16` bit struct may contain `u5, u6, u5` as components.
- **Bit arrays** pack repeated data types. For example, users can use a 32-bit bit array to store 32 × `u1` types or 8 × `i4` types.

We extend the data layout language of Taichi [49]. Bit adapters are extensions to the existing Taichi Structural Node (SNode) system. We refer the readers to [49]

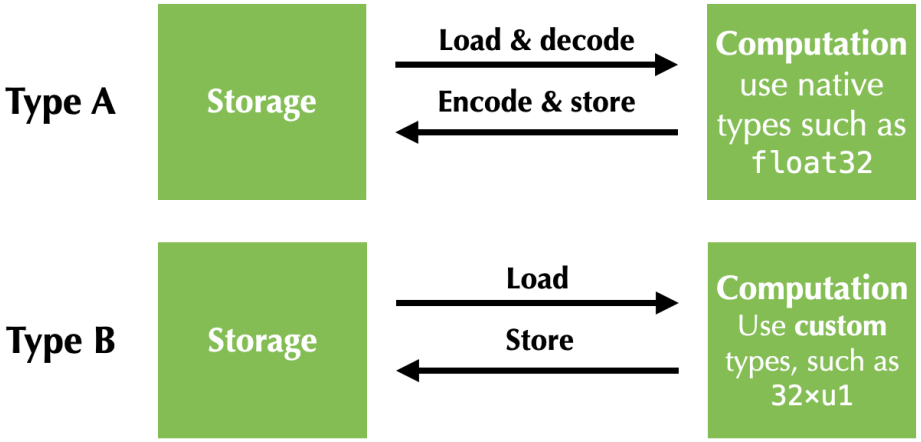


Figure 5-5: Two types of quantization investigated in our work. Note that "Type B" quantization may not be always possible and profitable compared to "Type A", since not all custom data types can be directly manipulated by hardware.

for more details on SNodes, which are *not* a prerequisite to the remaining of this manuscript.

A *bit struct* serves similarly to a `struct`, but has bit-level granularity. The following code declares two fields of quantized data types, and materialize them into two 2D 4×2 arrays:

```
u4 = ti.quant.int(num_bits=4, signed=False)
i12 = ti.quant.int(num_bits=12)

p = ti.field(dtype=u4)
q = ti.field(dtype=u4)

ti.root.dense(ti.ij, (4, 2))
       .bit_struct(num_bits=16)
       .place(p, q)
```

The `p` and `q` fields are laid in an array of structure (AOS) order in memory. Note the containing bit struct of a `(p[i, j], q[i, j])` is 16-bit wide.

Let's now look at a more practical example. In a 3D Eulerian fluid simulation, a voxel may need to store a 3D vector for velocity, and an integer value for "cell category" with three possible values: "source", "Dirichlet boundary", and "Neumann boundary". The developer can then use a single 32-bit `bit_struct` to store

all information on a voxel:

```
velocity_component_type =
   ti.quant.float(exp=6, frac=8, compute=ti.f32)
velocity = ti.Vector(3, dtype=velocity_component_type)

# Since there are only three cell categories,
# 2 bits are enough
cell_category_type =
   ti.quant.int(bits=2, signed=False, compute=ti.i32)
cell_category = ti.field(dtype=cell_category_type)

# The bit struct for 512x512x256 voxels
voxel = ti.root.dense(ti.ijk, (512, 512, 256))
          .bit_struct(num_bits=32)

# Place three components of velocity into the voxel,
# and let them share the components.
voxel.place(velocity, shared_exponent=True)
# Place the 2-bit cell category
voxel.place(cell_category)
```

The compression scheme above allows us to store 13 bytes ($4B \times 3 + 1B$) into just 4 bytes. Note that users can still use `velocity` and `cell_category` in the computation code, as if they are `float32` and `uint8`.

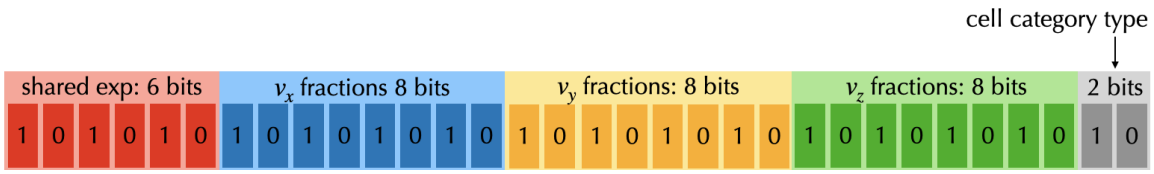


Figure 5-6: The `bit_struct` for a 3D smoke simulation voxel. Three components of `velocity` (v_x, v_y, v_z) share one common exponent which is placed in the highest 6 bits. The fractions of `velocity` occupy 24 following bits. The `cell_category` is placed in the lowest 2 bits.

Bit arrays are micro data structures that reinterpret hardware-native data types into arrays of low-bit types. For example, a programmer may want to store $8 \times$ `u4`

values in a single `u32` type, to represent bin values of a histogram (Fig. 5-7):

```
bin_value_type = ti.quant.int(num_bits=4, signed=False)

# The bit array for 512x512 bin values
array = ti.root.dense(ti.ij, (512, 64))
          .bit_array(ti.i, 8, num_bits=32)

# Place the unsigned 4-bit bin value into the array
array.place(bin_value_type)
```

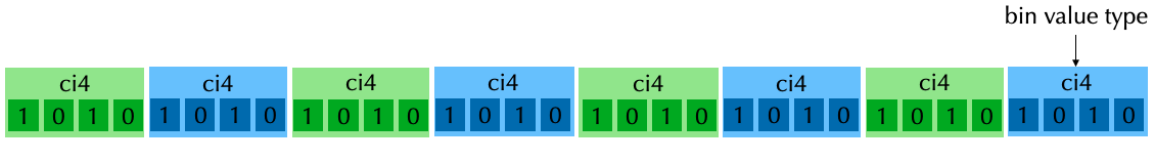


Figure 5-7: The bit array for bin values of a histogram. Eight 4-bit components are packed in a single `u32`.

4 .5 Decoupling numerical formats from computation

Similar to Halide [103] that decouples scheduling from algorithms, our system decouples numerical formats from computation.

This decoupling has crucial practical benefits in quantized simulations: it is challenging to predict how many bits are required for a numerical type in simulation, and the only way to confirm a quantization scheme does not cause too much truncation error, is to get the simulation running and observe the simulation result, quantitatively or qualitatively. This means the search for the optimal quantization scheme is, unfortunately, repeated trial-and-error. Real simulation code is much more complex, the only way to allows users to rapidly experiment with different quantization schemes is through a system design that separates numerical formats from the computation.

5 Code generation

In this section, we present a basic implementation of our compilation system, which mechanically translates the operations on custom data types to executable instructions on processors. We leave discussion on possible domain-specific optimizations to section 6 .

Our system is implemented based on Taichi, which has a hierarchical static-single assignment (SSA) intermediate representation (IR) system. Our system runs on x64, ARM64, CUDA, and Apple Metal devices. For x64, ARM64 and CUDA, code is just-in-time compiled using LLVM; for Apple Metal, Taichi emits Metal Shading Language source files and then leverages the Metal runtime system to launch GPU kernels.

5 .1 Type system

We *replaced* the original type system of Taichi, which only supports primitive data types such as `int32` and `float32`. We developed a new type system in the form of a hierarchical data structure composition system, internally implemented using a shallow tree of data types. See Fig. 5-8 for illustrations.

5 .2 Loading and storing custom integers

Loading a custom integer type from memory needs *addressing* and *decoding*. For *addressing*, we introduce *bit pointers* that precisely represent the starting bit of a custom integer type. *Decoding* is simply zero extension for unsigned integers and signed extension for signed ones. Similarly, custom integer stores need addressing and encoding (truncation).

Bit pointers Classical pointers only have byte granularity ("byte pointers", such as `char *` in C), but in our system we want a finer-resolution pointer at bit granularity, denoted as "bit pointer". A bit pointer has two components:

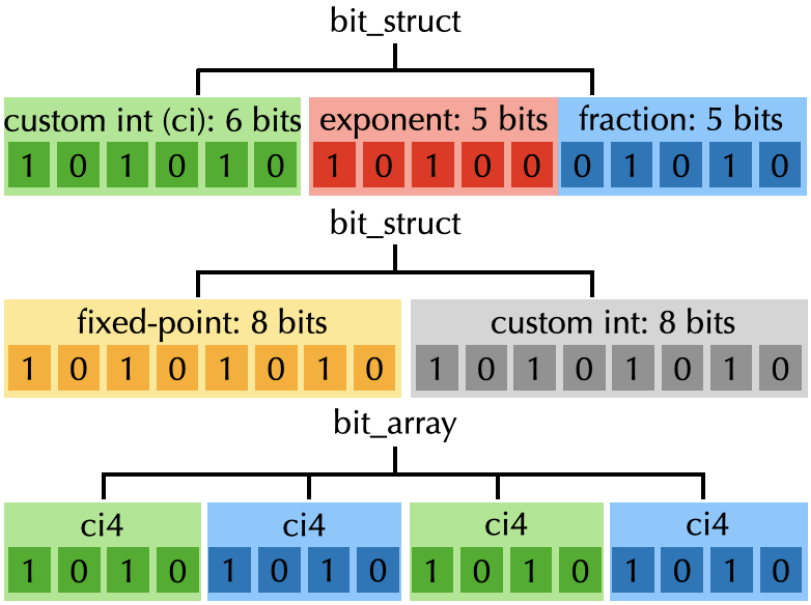


Figure 5-8: Illustration of our new type system. Here we show 3 examples of 16-bit `bit_structs/bit_arrays`. **Top:** One 6-bit custom integer ("ci" in short) and a floating-point number with 5-bit exponent and 5-bit fraction are placed in the `bit_struct`. **Middle:** There are one 8-bit fixed-point number and one 8-bit custom integer placed in the `bit_struct`. **Bottom:** A 16-bit `bit_array` is composed of four 4-bit custom integers.

1. a classical byte pointer that points to a byte or any other primitive types such as `i32, i64;`

2. an offset value that specifies a bit-level offset within the primitive type.

See Fig. 5-9 for an illustration.

Loading from and storing to bit pointers Hardware-native load/store instructions can only operate at `u8, u16, u32, u64` granularity. Therefore, to load or store data addressed by a bit pointer, we have to "simulate" partial bit loads and stores using hardware-supported memory operations and a series of bit operations (such as shifting) to extract or insert the bits we want.

Loading is relatively easy. We simply load the entire bit struct and use simple bit operations to extract the needed bits. For example, to extract the $[5, 8)$ bits from a 32-bit bit struct s, we can simply let the code generator emit `(s>>5)&7`.

Storing to a bit pointer means partially writing of these primitive types. This can be done via a load, a series of bit operations (Fig. 5-10), and finally a store. It is worth noting that sometimes multiple threads may write to the same bit struct, so we need the following read-modify-write operation to be atomic for thread safety. Note that the atomic read-modify-write (atomicRMW) has to be implemented via a while loop plus atomic compare-and-swap (atomicCAS) , and is relatively expensive. We implement corresponding compiler optimizations to avoid atomicRMW as much as possible when thread safety is not a concern (section 6).

5 .3 Efficiently decoding and encoding real numbers

Fixed-point numbers Since fixed-point numbers are simply integers multiplied by a compile-time constant scaling factor, after we load the underlying integer, the real number can be easily decoded by multiply the integer by the scaling factor. Encoding is the exact reverse process.

On fixed-point atomic adds, we override the decode-compute-encode cycle. We directly scale the real increment into an integer increment, and then use an atomic add on the integer type instead. This allows us to use atomic adds on integer to replace atomic adds on real numbers. In some cases, such as on OpenGL ES and Metal where only integral atomic adds are supported via the API and hardware, using integral instead of floating-point atomic add leads to a significant performance improvement (section 7 .4).

Rounding We adopt the round to *nearest* scheme when casting the scaled real number to an integer. Enforcing the current rounding scheme is critically important and has a direct impact on simulation results. See Fig. 5-11 for a comparison between different rounding schemes.

Floating-point numbers have exponent and fraction bits, which we handle independently. For the fraction part, we simply adopt an integer truncation with rounding to nearest. The exponent part, however, cannot be simply truncated. It is worth noting that the exponent format of IEEE 754 floating-point numbers does

not use two's complement for negative numbers[2], hence we need an integer to add operation before we truncate. Overflowing exponent bits are currently treated as undefined behavior, and in practice, we do not find this to be a problem as long as enough exponent bits are reserved.

Subnormal numbers Our system does not support subnormal floating-point numbers. These numbers are directly treated as zeros when encoding. To simplify and accelerate the decoding/encoding process, we turn on the "flush to zero" (FTZ) flags on CPU and GPUs `float32` data types.

Shared exponents Floating-point numbers with a shared exponent need special treatment. Suppose now we are encoding a set of `float32` numbers into binary bits. Denote the numbers as exponent-fraction pairs (e_i, f_i), where e_i and f_i can be extracted from the IEEE 754 floating-point compute type via cheap bit-wise operations. The encoding process works as follows:

1. Compute the maximum exponent of all floating-point numbers, $e = \max\{e_i\}$. Encode e to the exponent type format of the shared exponent type.

2. Each number now has an exponent offset $o_i = e - e_i$. For $e_i \neq 0$, we have to prepend the fraction part with e_i zeros. The padding zeros may lead to a precision degradation on values with small e_i, but having a small e_i itself implies the value is less important compared to the largest value sharing the same exponent.

3. Insert the shared exponent e and fractional bits into the bit struct.

When decoding we need to reconstruct the exponent offset o_i from each fraction bits. o_i can be reconstructed via a call into `__builtin_clz`, which computes the

[2]For example, the exponent of the `float32` type has range $[-126, 128)$ instead of $[-128, 128)$. The exponent of the represented floating-point number is 2^{e-127}, where e is the unsigned integer represented by the exponent bits.

leading zeros of the fraction part [3].

In practice, the decoding/encoding procedures for custom floating-point formats are tricky to get right, since there are a lot of variants such as signed v.s. unsigned, normal v.s. FTZ, shared v.s. non-shared exponents. Our implementations are included in the `taichi/codegen/codegen_llvm_quant.cpp` file.

6 Domain-Specific Optimizations

Theoretically, everything we have described so far can be implemented via a C++ library, with heavy operator overloading and templated data type classes. A "quantization library" in C++ may not be easy to use, but if zero-cost abstractions are properly used, it would have no performance difference from our domain-specific language. In this section, we show that, apart from usability, our compiler-based approach also has fundamental performance advantages over the library-based approach. This is because our tailored compilation system can conduct domain-specific optimizations that general-purpose compilers (such as `gcc` and `clang`) are not capable of.

Bit structs and bit arrays are first-class citizens of our compilation pipeline. The optimizer can easily analyze and optimize memory operations on their containing data types, leading to higher performance.

Before we detail our automatic optimizations, we stress that, if the data types are implemented via a "quantization library", a programmer can do all these optimizations manually through tedious low-level engineering. However, these optimizations are highly coupled with the underlying data layout and format, and manual optimization locks the code to a particular quantization scheme, leading to a "leaky abstraction". Since seeking the optimal quantization scheme needs repeated trial and error and even making problem-specific adjustments, manually doing the optimizations is not practical. Benchmarks that validate the effective-

[3] The most straightforward way to compute the exponent offset is to use a `std::log(float)` function call. However, this is too expensive in practice. Therefore we heavily use bit-wise operations that are much cheaper.

ness of these optimizations are detailed in section 7 .1 and 7 .2.

6 .1 Bit struct store fusion

In real-world applications, fields in a single bit struct are often accessed together, so it is highly possible that different components of a bit struct get stored by multiple statements in a single kernel. In this case, we can use a single atomicRMW for all the stores into that bit struct.

We introduce a new statement, namely `BitStructStoreStmt(addr, field1, field2, ...)`, in extension to the original general-purpose `GlobalStoreStmt(addr , field)` in Taichi IR, for domain-specific optimizations on bit struct stores.

We also add a few tailored optimization passes. The first pass converts `GlobalStoreStmt` into `BitStructStoreStmt` for easier analysis, and the second pass merges related `BitStructStoreStmt` into a single equivalent `BitStructStoreStmt`. See our supplemental document for a real-world example of this IR optimization.

Note that `BitStructStoreStmt` takes multiple field inputs. In the code generator, we additionally implemented a multi-field version of the partial bit storing procedural. This improves performance since the expensive atomicRMW is now amortized by all the fields to store into that bit struct.

We only optimize bit struct *stores*, because bit struct *loads* (load from an address and then extract the bits) are relatively easy to analyze and optimize by a general-purpose optimizer, such as the LLVM target-independent optimizer we are using. In contract, bit struct stores involve atomicRMW and general-purpose optimizers tend to be conservative regarding optimization. This is likely due to the difficulty of aliasing analysis, and the fact that in their IR a single atomicRMW statement would have been lower into quite a few non-trivial basic blocks with complex control flow connecting them.

6 .2 Thread safety inference

Take one step further, when there is certainly no data race on the bit struct stores, we can fully replace the atomicRMW with a much cheaper non-atomic version. Our compiler searches for two patterns for this optimization:

Element-wise accesses In parallel simulators, many operations happen in an "element-wise" manner: each independent thread processes one particle or voxel at a time. Memory loads/stores related to the particle or voxel are then completely free from data races. In this case, we can safely demote the atomicRMW by a non-atomic version. For example, in the following 2D grid boundary velocity projection code,

```
@ti.kernel
def project_velocity():
    for i, j in v:
        if j < 3 and v[i, j][0] <= 0:
            v[i, j][0] = 0
```

our optimizer will safely infer that the store to the first component of the velocity vector at `v[i, j]` (a bit struct) does not need any atomic operation since no other thread will access the same bit struct.

Storing the entire bit struct Recall that the reason why we need atomicRMW instead of the non-atomic version is to prevent overwriting other parts of the bit struct, which may be written simultaneously by another thread. However, it may be the case that the entire bit struct is stored in a single `BitStructStoreStmt`, then we do not need to worry about overwriting, since this thread is writing to the whole bit struct anyway. We find this pattern to be particularly frequent on particle and grid simulations.

6 .3 Bit array vectorization

Consider the following example, where data are copied from a 2D 128×128 `u1` ("boolean") array `x` to `y`:

```
x = ti.field(dtype=ti.quant.int(bits=1, signed=False)
y = ti.field(dtype=ti.quant.int(bits=1, signed=False)

cell = ti.root.dense(ti.ij, (128, 4))
cell.bit_array(ti.j, 32).place(x)
cell.bit_array(ti.j, 32).place(y)

@ti.kernel
def copy():
    for i, j in x:
        y[i, j] = x[i, j]
```

Although our system can easily improve storage efficiency, computationally this bit-wise for loop is inefficient for two reasons. Firstly, we have to use hardware-native 32-bit integer registers for our simulated 1-bit values, which uses only 1/32 of the operation bitwidth. Secondly, when store the results bit-by-bit, the code generator has to issue a large number of expensive atomicRMW operations for thread-safety, since multiple CPU/GPU threads may write to different bits within a single `u32`, leading to data races.

Bit-wise for loop vectorization The situation above inspires us to vectorize bit array load, store, and arithmetics, so that each iteration processes a whole `32xu1` bit array instead of a single `u1`. This not only utilizes the full bitwidth but also eliminates the need for atomicRMW. Unlike traditional vectorized operations supported by modern processors (such as SSE and AVX), bit-level vectorization has limited hardware instructions. Basically, the only "bit-vectorized" operations offered natively by hardware are bit-wise `and (&)`, `or (|)`, and `xor (^)`.

At this point, the compiler can efficiently handle simple element-wise load/store operations, plus some Boolean arithmetics. We further conduct two optimiza-

tions that improve code generation quality and capability.

Bit-vectorized loads with offsets In simulations it is often useful to load data from neighborhoods, such as `x[i+1, j]` and `x[i, j+1]`. Assuming bits are vectorized along the `j` axis, a vectorized load of `x[i+1, j]` is perfectly aligned with the bit arrays, but that of `x[i, j+1]` is not.

Actually, in most cases, data to fetch in a vectorized loop iteration do not perfectly align with the underlying bit arrays. Luckily, if we know the offset at compile-time, which is most likely true for stencils, we can still leverage most of the bit vectorization benefits by loading two adjacent bit array entries and use cheap bit operations to synthesis the load result, as shown in Fig. 5-12.

After this optimization, the following code snippet can be efficiently compiled:

```
@ti.kernel
def copy_with_offset():
    ti.bit_vectorize(32)
    for i, j in x:
        y[i, j] = x[i, j + 1]
```

Bit-vectorized integers and adders Even when operating on binary inputs and outputs, intermediate values may have large value ranges. To represent these intermediate values efficiently, for example, we store a n-bit 32-wide vectorized integer using n `u32` integer buffers, where the i-th buffer stores the i-th bit of all the integer values being vectorized. These "bit-vectorized" integers allow us to treat each bit of the n-bit integer in a vectorized manner, independent of other bits on the n-bit integer.

Adding two bit-vectorized integers can be implemented using cheap bit operations, just like implementing a "full adder" using logic gates. Besides adding, we also implemented comparison operations between bit-vectorized integers using bit-wise operators.

In the following example, since variable `count` as range $[0, 4)$, we use a 2-bit vectorized integer to store its value.

```
@ti.kernel
def count_neighbors():
    ti.bit_vectorize(32)
    for i, j in x:
        count = 0
        count += x[i, j + 1]
        count += x[i, j - 1]
        count += x[i + 1, j]
        y[i, j] = count == 3
```

7 Applications and Evaluations

In this section, we showcase the applications of our system and evaluate its performance and accuracy under memory space constraints. We set up three microbenchmarks (each round 50 lines of code) and three large-scale benchmarks (100 to 1500 lines of code). The large-scale simulation statistics are listed in Table 5.2.

7 .1 Microbenchmarks

We set up a suite of microbenchmarks to unit-test our domain-specific optimizations and study their impact on performance. Details and code of the benchmark cases are in the supplemental document.

The numbers obtained with "all optimizations off" mimics the performance of a library-based approach: via template metaprogramming and operator loading, the member functions of quantized data type classes directly emits operations to a general-purpose compiler, LLVM in our case. LLVM will not be able to merge the bit struct stores, since it lacks a high-level understanding of the bit struct stores.

The benchmark results (Table 5.1) validate that a compiler-based approach is substantially beneficial compared to a more traditional library-based approach. Computing the geometric mean of running time of all the cases on CPUs and GPUs, turning on store fusion leads to $1.43\times$ (x64 CPU)/$1.91\times$ (CUDA) speed up,

and further turning on atomic demotion leads to another $1.93\times$ (x64 CPU)/$2.15\times$ (CUDA) speed up.

7 .2 Game of Life

We first test our system on the classical Conway's Game of Life, a 2D grid-based simulation that is extremely simple to code but computational hungry at a high resolution. Each cell (i, j) can have two states, either *live* ($l_{i,j} = 1$) or *dead* ($l_{i,j} = 0$). The cell states follow a set of simple evolution rules, depending on its 3×3 neighborhood:

- **Birth**: each dead cell with *exactly* three neighbors becomes a live cell;
- **Survival**: each live cell with two or three live neighbors continues to be a live cell;
- **Overpopulation**: each live cell with four or more live neighbors dies;
- **Isolation**: each live cell with zero or one live neighbor dies.

Essentially, the next-step cell state $l'_{i,j}$ is defined as

$$l'_{i,j} = f\left(\sum_{x=i-1}^{i+1}\sum_{y=j-1}^{j+1} l_{x,y}\right),$$

where f maps the number of active neighbours into the cell state of the next time step.

Storage We created two copies of the grid, namely l' and l, and iterate back and forth. We use two hierarchical grids to store the current and next frames. Using bit arrays that pack 32 `u1` (1-bit unsigned integer) type into a single `u32`, each cell takes only 1 bit in each grid, leading to a 2 bit/cell total storage footprint. Note that in traditional languages such as C, programmers will have to use the `char` (`u8`) type for each cell, unless they manually pack/unpack the states. In our system, users

Table 5.1: An ablation study on three microbenchmark programs. “Demotion” means atomic demotion, and “fusion” means bit struct store fusion.

Cases	Backend	All optimizations off	Demotion only	Fusion only	All optimizations on	No quantization
Store	x64	688.570 ms	680.164 ms	571.710 ms	338.451 ms	312.925 ms
	CUDA	3.741 ms	3.729 ms	1.874 ms	0.623 ms	0.623 ms
Partial store	x64	882.831 ms	332.785 ms	536.609 ms	309.125 ms	345.032 ms
	CUDA	5.619 ms	2.759 ms	2.888 ms	2.762 ms	2.751 ms
Matmul	x64	272.056 ms	76.015 ms	185.773 ms	76.349 ms	-
	CUDA	9.704 ms	1.705 ms	5.410 ms	1.705 ms	-

Table 5.2: Statistics of our large-scale demos. The Game of Life demo adopts more steps per frame as the camera zooms out, and the numbers here are the maximum steps per frame after the camera fully zooms out. For the memory allocated, note that the memory allocator of Taichi maintains auxiliary data structure and enforces padding, leading to more memory allocated than actual used, especially on the Game of Life and MLS-MPM demos. See our video for more visual results.

Solver	**Demo**	**Frames**	**Steps/ frame**	**Seconds/ frame**	Δt	**Grid**	**Active cells**	**Particles**	**Byte/element**		**GPU**	**GB allocated**
									Full	Quant		
Game of Life	Fig. 5-1	720	1024	30.2	-	$131,072^2$	$10,487,808,100$	-	2	0.25	3080	4.0
	Fig. 5-13	720	1024	53.0	-	$262,144^2$	$20,554,956,900$	-	2	0.25	3080	7.0
Eulerian Fluids	Fig. 5-16 (top)	300	1	68.8	3×10^{-2}	$2,048^3$	$421,134,336$	-	110	70	V100	29.3
	Fig. 5-16 (bottom)	280	1	58.3	3×10^{-2}	$2,048^3$	$421,134,336$	-	110	70	V100	29.7
MLS-MPM	Fig. 5-19	240	129	76.2	7.8×10^{-5}	$4,096^3$	$72,392,832$	234,527,481	68	40	3090	16.1

can effortlessly improve storage efficiency by $8\times$, without any modification of the computation code.

Initialization To demonstrate the capability of our Game of Life simulator, we initialize the grid using tiled OTCA metapixels[4]. An OTCA metapixel consists of 2048×2048 cells, and it has an interesting behavior: when looking at a distance, the whole 2048×2048 metapixel behaves like a single 1×1 Game of Life cell (Fig. 5-13). We also set up a Game of Life pattern with words "Quant Sim" using 70×70 OTCA metapixels, reaching over 20 billion cells in a single simulation (Fig. 5-1). For a high-resolution visualization of this demo, please refer to our video and supplemental material.

The effectiveness of bit vectorization In each time step of Game of Life, a cell loads its 3×3 neighborhood states from the old state buffer and stores its new state to a new state buffer. Bit array vectorization improves performance by simultaneously handling 32 cells in a single thread. We compare the running time between three different implementations:

1. QuickLife from Golly[5], a fast algorithm leveraging sparsity in Game of Life. The implementation in Golly is highly optimized on CPUs.

2. Ours, without bit vectorization. Since this implementation only utilizes $1/32$ of the bitwidth and leads to excessive atomicRMW, we do not expect it to deliver satisfactory performance.

3. Ours, with bit vectorization. This is essentially (2) with a extra `ti.bit_vectorize(32)` pragma. Our compilation passes do the vectorization job automatically.

As expected, the performance of our non-vectorized implementation is much worse than QuickLife on CPU, while the same algorithm achieves comparable performance with the significant optimization of bit vectorization on CPU. The effec-

[4] https://www.conwaylife.com/wiki/OTCA_metapixel
[5] http://golly.sourceforge.net/

Table 5.3: Per step running time of three different implementations, on CPUs and GPUs. The benchmark is initialized using a random pattern with 41690^2 active cells with 50% live cells. We run CPU benchmarks on an Intel i7 processor (six cores each at 2.6 GHz) and 16GB of memory. We run GPU benchmarks on an NVIDIA RTX 3080 GPU with 10GB of GPU memory.

Implementation	Time per step (CPU)	Time per step (GPU)
QuickLife	318.5 ms	N/A
Non-vectorized brute-force	9313.4 ms	546.0 ms
Vectorized brute-force	417.2 ms	3.4 ms

tiveness of bit vectorization is further verified on GPU where the vectorized version is more than $150\times$ faster than its non-vectorized counterpart (Table 5.3).

Our implementations do not utilize spatial sparsity yet, and compared to QuickLife it does more work. This partially explains why our vectorized simulator is 31% slower than QuickLife on CPUs.

7 .3 Eulerian fluid simulation

We developed a sparse-grid-based advection-reflection [132] fluid solver to evaluate our system on grid-based physical simulators.

For advection, we use the MacCormack scheme [109], with RK3 path integration. For projection ("reflection"), we use multigrid preconditioned conjugate gradients (MGPCG) [86] to solve the Poisson problem. We follow the MGPCG solver design in [49]. We use two-level pointer grids (`ti.root.pointer(ti.ijk, ...).pointer(ti.ijk, ...).dense(ti.ijk, ...).place(...)`) for each level of the grid hierarchy.

The majority of memory consumption comes from the top level of the grid hierarchy. This is because the second level only has $1/8$ voxels due to multigrid coarsening. Also note that physical properties such as dye density (R, G, B) and velocity (u, v, w) only exist at the top level, and we need to store multiple copies of them for the MacCormack advection scheme.

Quantization scheme We focus our quantization on the advection solver, since it costs the majority of memory space (84 out of 110 B). By representing each component of velocity (u, v, w) using a 21-bit fixed-point number, we pack them in a single 64-bit bit struct. Also, we managed to pack three channels of dye density in a 32-bit bit struct by using floats with shared exponents: we use 9 bits for fractions and 5 bits for exponent, which adds up to 32 bits. Since we adopt a MacCormack advection solver, we use three bit structs to store v_t, v_{t+1} and an auxiliary $\bar{v}_{t+1}$ respectively. Similarly, the dye density needs three bit structs. Note that another bit struct is used as the velocity after the reflection operator being applied. In this way, each voxel occupies 44 bytes, while using float32 needs 84 bytes (Fig. 5-14).

The effectiveness of shared exponents To prove the effectiveness of shared exponents, we compare shared exponent, non-shared exponent and fixed-point using a 2D smoke simulation. All of these experiments use one 32-bit bit struct to store dye density (R, G, B) channels. In contrast, the uncompressed `float32` reference uses 96-bits per voxel.

For the simulation with shared exponent, we use 5 bits for exponent and 9 bits for fraction. The non-shared exponent simulation also uses 5 bits for exponent, but only 5 bits for fractions for each channel. The fixed-point simulation uses 10-bit fixed-point numbers per channel. We show the comparison result in Fig. 5-15.

Note that there is an exponential decay of density in the simulation, and ultimately all pixels will decay into the white background color. When compared to the `float32` version. In the fixed-point simulation, the smoke color stopped decaying after a few seconds and does not look clear. This is because the precision is too low to distinguish small changes. The non-shared one looks better but still suffers from the same problem. The shared exponent version, however, has sufficient fraction bits and looks much closer to the `float32` version compared with two other methods. To quantitatively study the preciseness of the decaying behavior using different data formats, we sum up all the cell density (Table 5.4) and find the shared exponent version has the closest total density compared to the `float32` reference

Table 5.4: Total density comparison. Note that since the decaying factor is a constant on all pixels and channels, the total density is a predictable value over the simulation, regardless of the turbulent nature of the fluid simulation. This experiment runs on a GTX 1080 Ti GPU with 11 GB memory.

Data Type	Total density
float32	15425.828
shared exponent: exp5 + frac9	17072.586
fixed-point: 10	24662.654
non-shared exponent: exp5 + frac5	41368.633

Table 5.5: Performance comparison of Eulerian fluid advection. This experiment also runs on a GTX 1080 Ti GPU with 11 GB memory.

Data Type		Time
Velocity	Dye density	
float32	float32	25.052 s
shared exponent: exp5 + frac9	fixed-point 10	31.776 s
fixed-point 10	fixed-point 10	24.760 s

simulation. This proves that shared-exponent floating-point formats achieve both good precision and dynamic range, compared to non-shared exponent floating-point formats and fixed-point formats.

Performance of quantized simulations Finally, we compare the performance of our quantized simulator against the `float32` reference implementation. We find the quantization scheme using shared exponent to be roughly 27% slower than the full-precision version, likely because encoding/decoding floats with shared exponent takes some additional computation. Interestingly, the all-fixed-point version is slightly faster than the reference version, despite needing more floating-point multiplications to encode/decode. Since the advection kernel is memory-bound, the quantized version consumes less memory bandwidth, hence running faster.

7 .4 Moving Least Squares Material Point Method

To test our system on hybrid Lagrangian-Eulerian methods where both particles and grids are used, we implemented the Moving Least Squares Material Point Method [48] with G2P2G transfer [124].

In MPM simulation, per-particle data is very memory- and bandwidth-consuming. Note that in MLS-MPM over 80% storage is for particles. A 4096^3 background sparse grid is used, leveraging the first-class spatially sparse data structure support in the original Taichi system. We use a 4^3 leaf block size.

When simulating elastic objects, we store position, velocity, and deformation gradients on each particle. After a few trial-and-error, we end up with a quantization scheme that compressed 68 B particle attributes to 40 B, a $1.7\times$ improvement. Note that in MPM we also need to store the grids and other acceleration structures such as the particle list, which is not quantized here. The overall memory efficiency improvement is $1.3\times$. See Fig. 5-17 for more details on the quantization scheme. A 235M-particle visual demo is shown in Fig. 5-19.

Performance In a GPU MLS-MPM simulator, the performance is limited by memory bandwidth and available FLOPs. Using quantization increases the amount of computation due to the need for encoding/decode, yet at the same time it reduces the memory bandwidth. We conduct a systematic performance study on our system, scanning all possible combinations of quantization, optimization, and particle-grid transfer scheme (separate P2G/G2P and fused G2P2G [124]). Results are listed in Table 5.6. Interestingly, we find our quantized simulator runs $1.03\times$ (G2P2G)/$1.14\times$(P2G+G2P) faster than the full-precision simulator, likely because the quantized simulator saves memory bandwidth. We get higher speed up on the P2G+G2P transfer scheme, because this version with two kernels needs more memory bandwidth. Our domain-specific optimizations leads to $1.09\times$ (G2P2G) and $1.80\times$(P2G+G2P) higher performance on the quantized simulator. Wang et al. [124], a CUDA MLS-MPM solver heavily engineered towards performance, our quantized version is $3\times$ slower, most likely due to the fact that we did not

Table 5.6: Domain-specific optimization ablation study on the MLS-MPM benchmark, with and without the G2P2G optimization [124]. Results are collected on a RTX 3090 GPU. We seed 16,777,216 particles in a 256^3 domain.

Optimizations	G2P2G Time (s)	P2G+G2P Time (s)
No optimization	16.52	28.43
Store fusion only	15.99	17.19
Atomic demotion only	16.36	16.69
All optimizations on	15.09	15.77
No quantization	15.57	17.96

implement the AOSOA data layout optimization. However, regarding memory consumption our system is $5.7\times$ more efficient (1.4GB ours v.s. 8GB [124]). Note that that memory efficiency gap is a combination of our quantization scheme and the fact that [124] is optimized towards performance instead of memory.

Quantization on mobile devices Since mobile devices have the relatively limited computing power and a strong need for real-time response, typically only small-scale simulations run there and storage is not really an issue. Surprisingly, we still find using quantized data types on the background grid to be beneficial: since mobile GPUs usually only has high-performance native atomicAdd support for `i32` but not for `f32`, using `ti.quant.fixed(fraction=32)` on the grids converts software-emulated `f32` atomicAdd to hardware-native `i32` atomicAdd significantly improves P2G performance in our MLS-MPM program on an iPhone XS (Fig. 5-18). See our supplemental video for more visual comparisons.

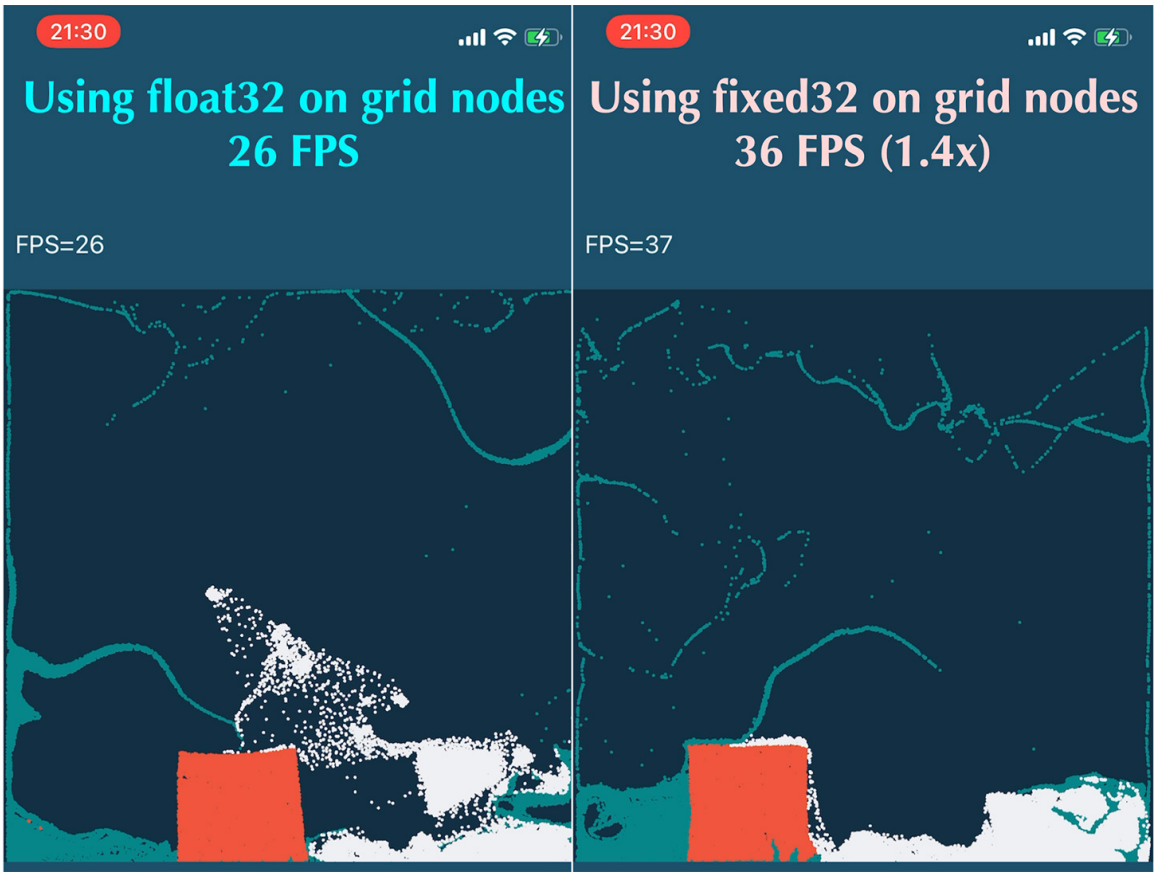


Figure 5-18: A benchmark of $36,000$ MLS-MPM particles with 256×256 grid nodes, on an Apple iPhone Xs. Using our `ti.quant.fixed(fraction=32)` types on the grid nodes improves FPS by $1.4\times$ compared to native `float32`.

7 .5 User studies

We have done a round of user study to investigate how much visual difference quantization injects into the simulator. We collected five groups of simulation videos, two of them are generated using full-precision `float32` simulators, and one of them is generated using quantized simulators. We present one of that `float32` videos and ask the user to select the one he thinks is closer to the video just played from the remaining two videos. The volunteers can also choose "cannot decide" if he or she thinks two videos are equally close to the first video. We asked 20 volunteers and collected 100 responses in total. User study cases and results are presented in Fig. 5-20. Our supplemental video includes all the user study videos.

2D v.s. 3D We find that 3D results lead to more "cannot decide". We hypothesis this is because in 2D users see everything in the simulation, so relatively more information is exposed. In 3D occlusions makes judgments harder.

Solid v.s. Fluids Surprisingly, most users can easily point out slight differences in solid simulations. This is not because quantized simulators are not realistic, but because quantization injects extra noise to the system and leads to a different final particle distribution. In fluid simulations, deciding becomes harder.

The 3D fluid results (case 3 and 5) are promising: users are mostly making random guesses on these cases.

7 .6 Discussions

Productivity Thanks to our decoupling of numerical data formats from numerical computation, the amount of code modified to transform a traditional full-precision physical simulator into a quantized simulator is no bigger than 3% of the total solver code. For example, the whole MLS-MPM solver has roughly 1000 lines of code, while specifying a single quantization scheme takes only 30 lines of code (3%). To quantize the 900-line fluid simulator, no more than 20 (2.2%) lines of code are added.

Failure cases It is not uncommon that using data types that have too low precision or dynamic range leads to simulation artifacts. Fortunately, our system which allows rapid trial-and-error, and allows programmers to simply use more bits in this case. For example, we found using 16-bit fixed-point numbers for fluid volume ratio $J = V_t/V_0$ [116] leads to a clear volume gain. This can be easily fixed by letting J have 23 bits instead of 16 (Fig. 5-21).

8 Conclusion

In order to make quantized simulation practically programmable, we have developed a tailored programming interface, compilation system, and domain-specific optimizations. Our system is orthogonal to many of the existing work to accelerating simulations, including sparse data structures [89, 110, 41, 49]. Using our system, programmers can effortlessly switch between different quantization schemes, leading to $1.57 \sim 8.00\times$ memory space efficiency improvement. A user study shows that 3D quantized simulation results are indistinguishable from full-precision ones. Our system is performant and easy to use: by modifying no more than 3% of the simulator code, a developer can quantize a MLS-MPM or Eulerian fluid simulator, running at comparable speed to the full-precision version.

Future work Currently, programmers still have to manually experiment with different quantization schemes. It would be helpful to have a system that automatically figures out suitable quantization schemes. Regarding engineering, adding a debugging system that detects overflowing of fixed- and floating- point numbers can help programmers more easily diagnose issues in a quantization scheme. Although our discussions are focused on a shared-memory environment, our quantization compiler can also help multi-GPU and distributed memory computation, since quantized physical states can significantly reduce communication overhead in those scenarios.

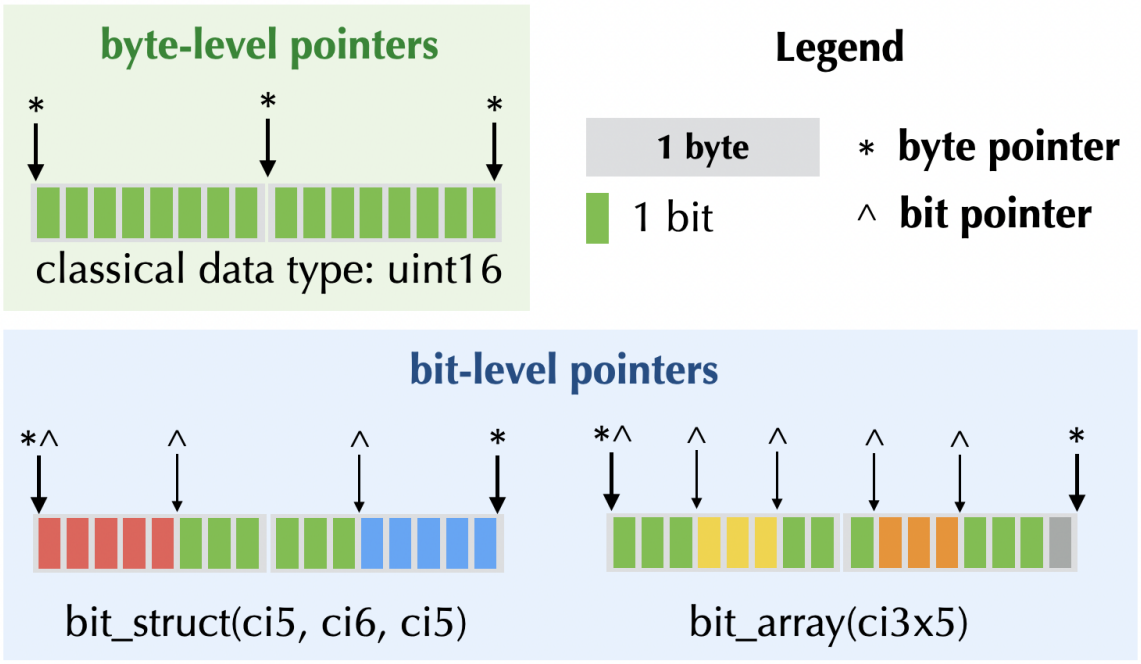


Figure 5-9: Byte pointers "*" and bit pointers "^". **Top left:** A traditional byte pointer only has a byte-level resolution. **Bottom:** Bit pointers have a bit-level resolution, and can easily point to components of bit structs and bit arrays.

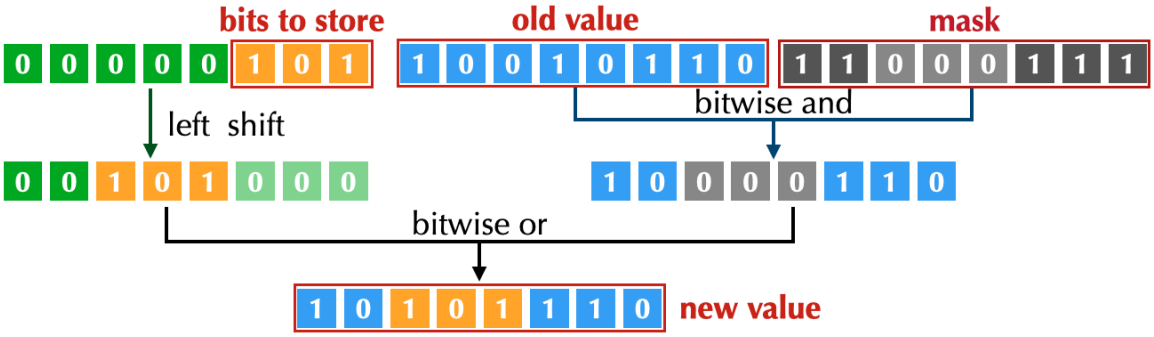


Figure 5-10: An illustration of storing bits partially in a bit struct. Here we show the process of inserting a 3-bit custom integer placed in an 8-bit bit struct, using a series of cheap bit-wise operations.

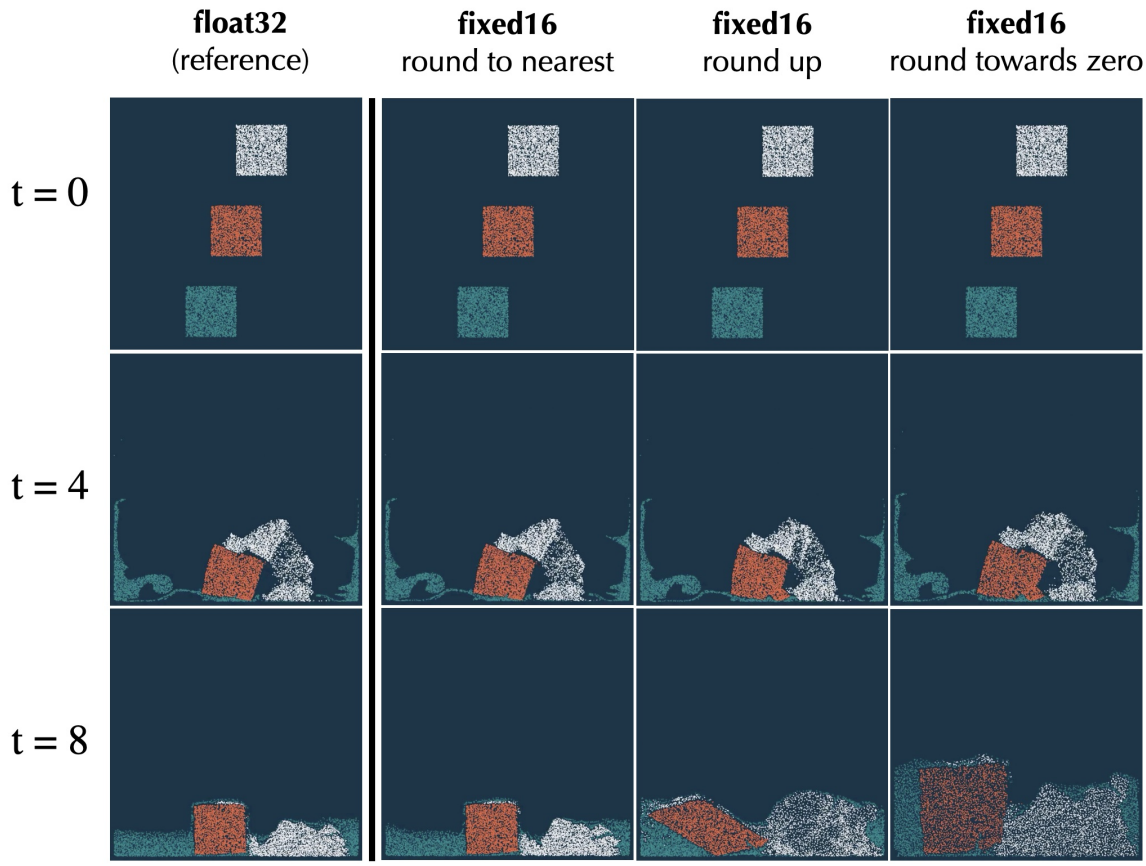


Figure 5-11: Rounding scheme matters. In this 2D MLS-MPM experiment [48] we use 16-bit fixed-point numbers for the deformation gradient variable on each particle. We stick to the "round to nearest" scheme, which ensures a close approximation to the `float32` reference. Note that the "round up" scheme leads to a shearing artifact to the elastic material (red), and that "round towards zero" results in an expanding volume.

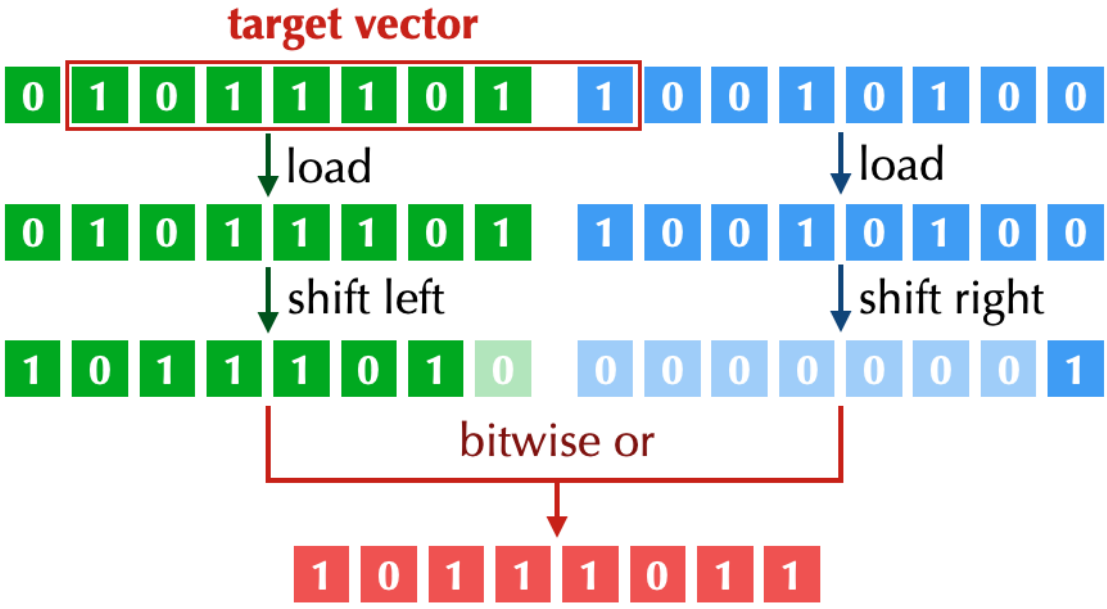


Figure 5-12: Bit-vectorized loads with offsets. In this simplified example, we store the bit array using `uint8` and each node represents an element in the bit array. To load `x[i, j+1]` (red box), we first need two vectorized loads (green and blue nodes). Then we use bit shifting operations to extract the target bits and move them to corresponding locations. Finally, we merge them using a bit "or" operation.

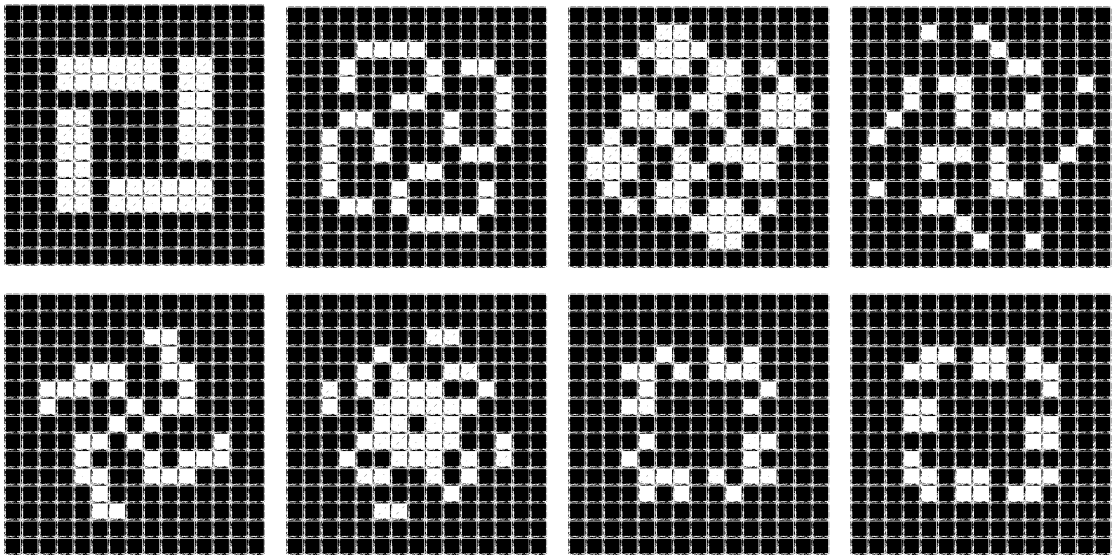


Figure 5-13: The evolution of the 15×15 galaxy pattern. Each cell is a 2048^2 OTCA metapixel, and a metapixel step is 32768 Game of Life time steps. Each metapixel evolves following the Game of Life rules when zoomed out.

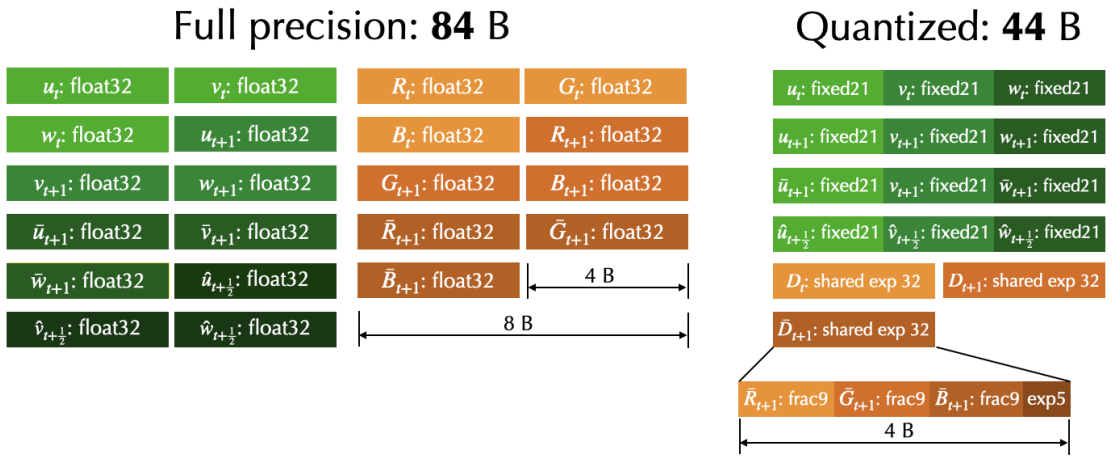


Figure 5-14: Eulerian smoke simulation quantization scheme. For each voxel, we pack velocity (u, v, w) in a 64-bit bit struct by three 21-bit fixed-point numbers. Dye density (R, G, B) is represented by 32-bit shared-exponent numbers with 5 bits for exponent and 9 bits for fraction. In this way, our smoke advection memory consumption is reduced by 48% from 84 bytes to 44 bytes. Considering 26 B from the MGPCG solver, the memory consumption per voxel for the entire fluid solver is reduced from 110B to 70B, a $1.57\times$ improvement.

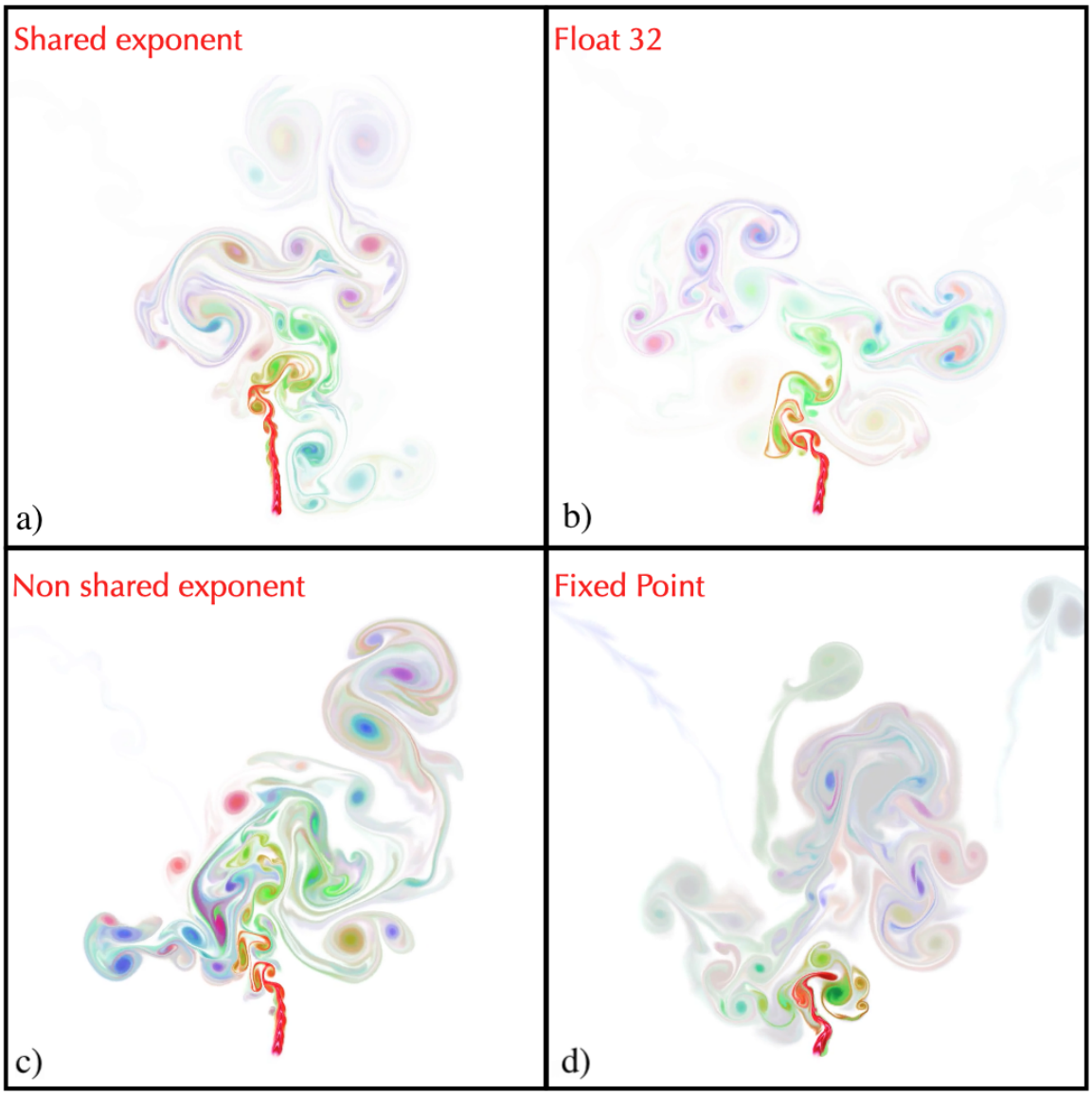


Figure 5-15: Shared exponent effectiveness comparison. We compare shared exponent, non-shared exponent, and fixed-point for a 2D smoke simulation. Notice shared exponent float looks closer to float32 version while non-shared exponent float and fixed-point suffer from low precision due to fewer fraction bits. Note that fluids are highly turbulent so the dye patterns are different from time to time, even when using `float32`.

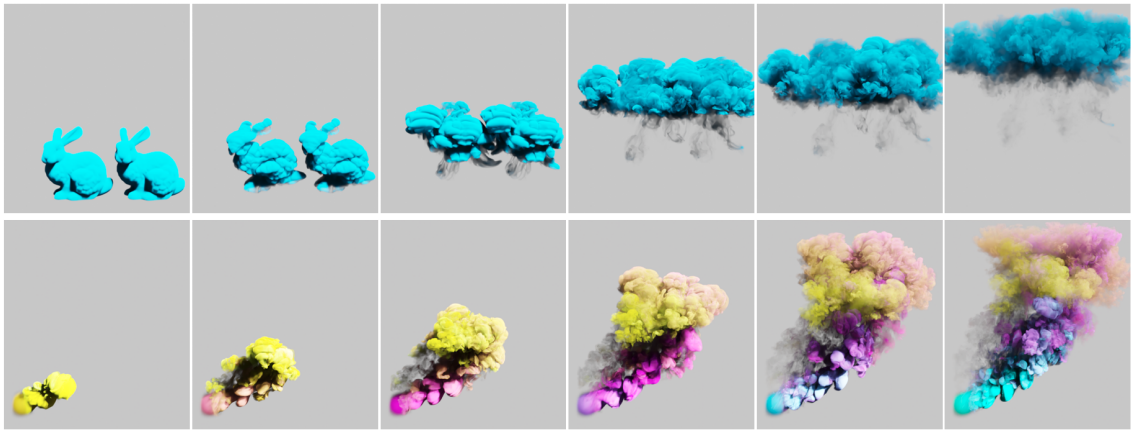


Figure 5-16: Two quantized smoke simulations. We use the quantization scheme described in section 7 .3. Both simulation run on a 2048^3 sparse grid with 421M active voxels. **Top**: Smoke initialized from two bunny meshes. **Bottom**: Smoke emitting from a spherical source.

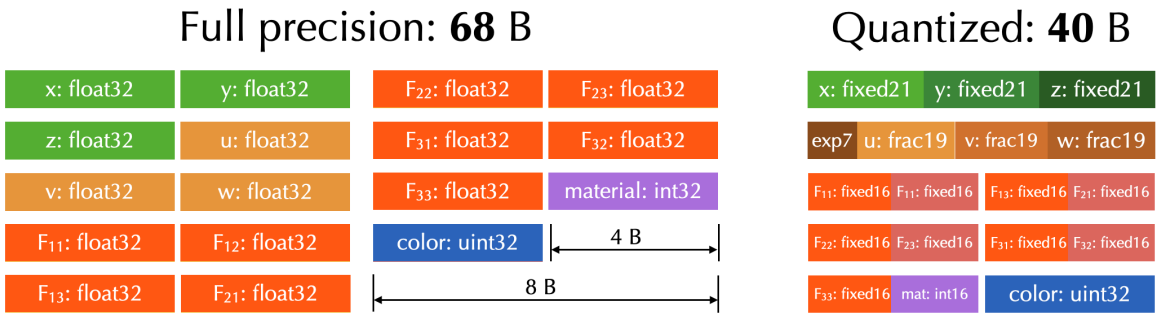


Figure 5-17: MLS-MPM particle attribute quantization scheme. For each particle, we store position (x, y, z) using 21-bit fixed-point numbers, velocity (u, v, w) using floating-point numbers with 17 fraction bits a shared 7-bit exponent. For deformation gradient $F_{3\times3}$, we use 16-bit fixed-point numbers. We also store material and color information. This brings down a particle storage footprint from 68 bytes to 40 bytes ($1.7\times$ fewer).

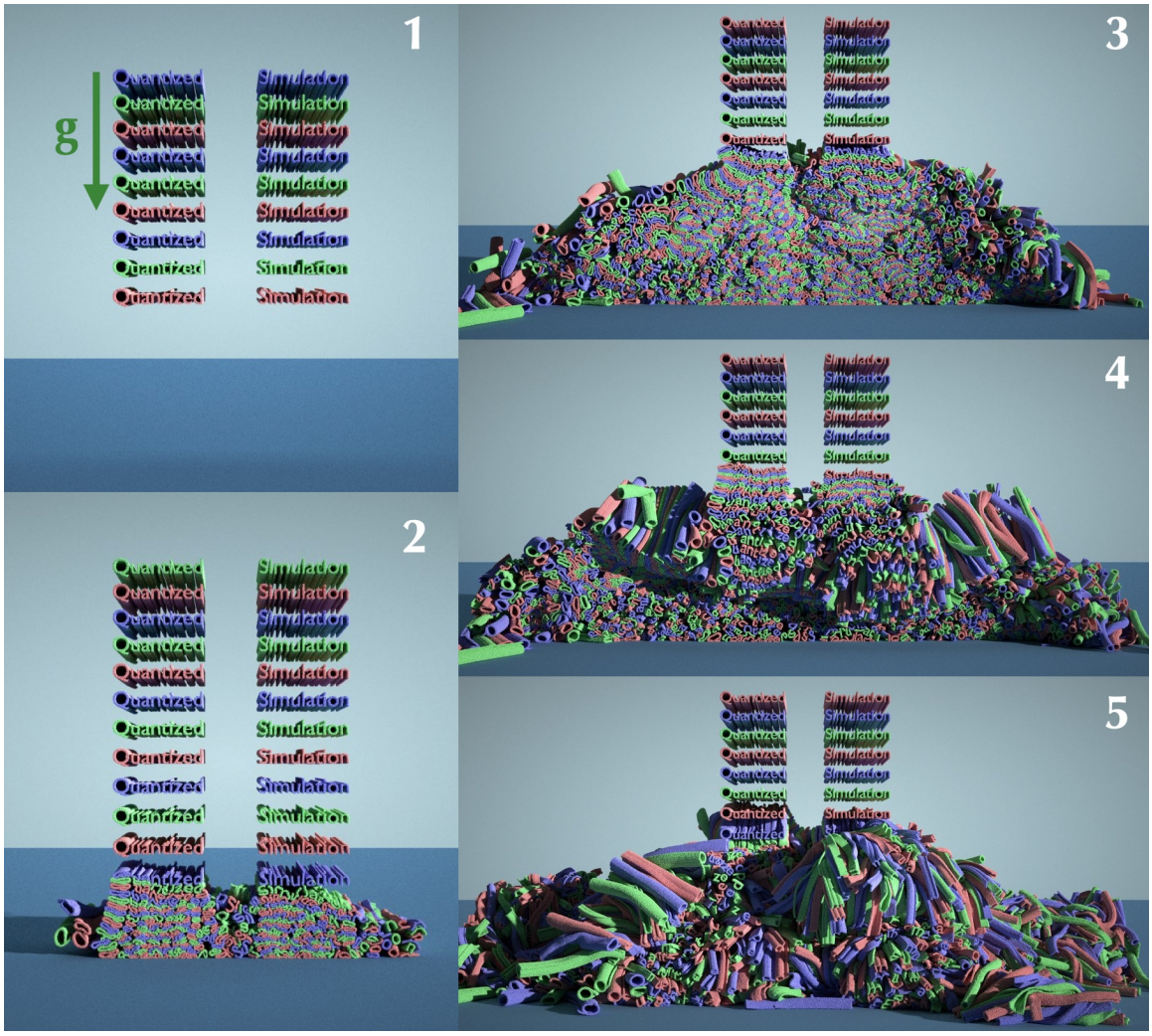


Figure 5-19: A 235M-particle MLS-MPM simulation. **(1)** $4,693$ elastic tubes fall down, **(2, 3)** form an interesting mountain structure, and **(4, 5)** ultimately collapse due to instability. Note the tubes are lengthy along the camera direction.

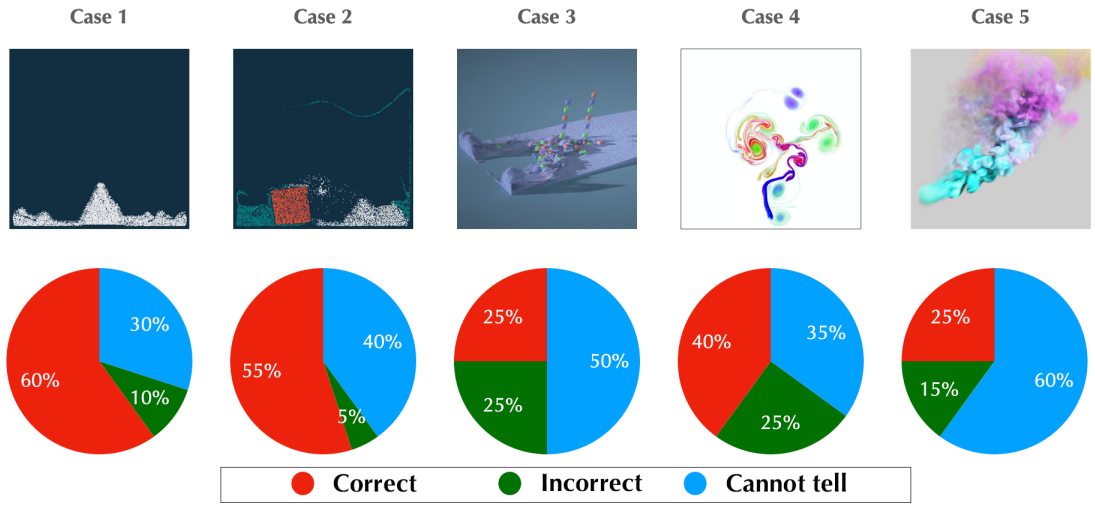


Figure 5-20: Five user study cases and results. In 2D cases with solids (case 1 and 2), users can point out the `float32` version reasonably well. In 3D and fluid cases (case 3, 4, 5), user feedback is close to random guess.

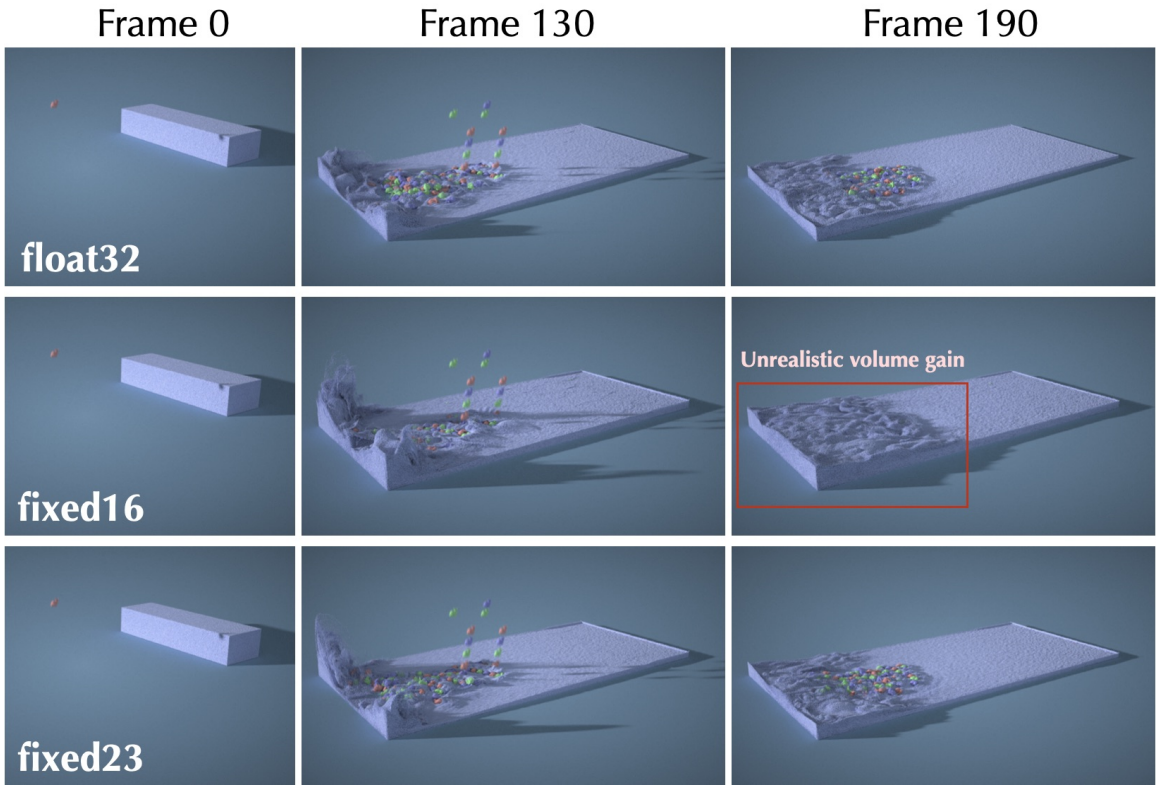


Figure 5-21: A 12M-particle MLS-MPM fluid simulation with volume ratio J on each particle quantized. Note that using `fixed16` leads to a unrealistic volume gain compared to the `float32` reference. The programmer can easily fix this by adding 7 more bits to the variable and use `fixed23` instead.

Chapter 6

Discussions

We have presented core research contributions of the Taichi programming language. This chapter discusses the relationships between Taichi and other programming systems, and potential future work.

1 Relationships to other programming systems

Research in domain-specific languages has effectively bridged the gap between emerging visual computing patterns and parallel hardware. In this section, we compare Taichi to related domain-specific programming systems, including TACO [67], deep learning frameworks (e.g., TensorFlow [3], PyTorch [96], and JAX [14]), Halide [102], Liszt [30], Simit [69] and Ebb [13]. More discussions on a specific Taichi feature can be found in the related work sections of previous corresponding chapter.

1 .1 Taichi v.s. TACO

Key difference: Spatially sparse computation v.s. sparse linear algebra.

The Tensor Algebra Compiler [67] is a compiler for sparse linear algebra. TACO takes *holistic* sparse tensors as inputs, offering a programming interface that evaluates various forms of Einstein sums, such as $A_{ij} = \sum_k B_{ijk} c_k$, with each tensor

being sparse or dense.

Taichi, on the contrary, offers a Single-Program Multiple Data (SMPD) programming interface to manipulate *individual* voxels in mutable sparse fields (tensors). This not only offers finer granularity, but also enables programmers to program imperatively. The benefit of such design is the flexibility to write complex programs such as physical simulators on the grid, with the majority of arithmetics being voxel-level in-place sparse tensor operations, often in the form of *random accesses*, and *parallel loops with complex control flows*. The cost of such flexibility, however, is that Taichi programs are less structured. For example, when expressing tensor linear algebra in Taichi, the compiler will not have a high-level understanding of the summation expression and will lack the capability to conduct TACO-style program transformations and optimizations.

Taichi and TACO have different design decisions given different application domains. Taichi is designed for spatially sparse computation such as sparse-grid physical simulation, while TACO is designed for sparse linear algebra. The majority of computation in Taichi, such as stencils, cannot be efficiently expressed in TACO. Similarly, TACO operations, e.g., sparse matrix multiplication, are not ideal use cases of Taichi.

1 .2 Taichi v.s. deep learning frameworks

Key difference: Explicit megakernel parallelism v.s. Array-based implicit parallelism.

Deep learning systems (such as TensorFlow [3], PyTorch [96], and JAX [14]) are designed around neural networks with classical layers such as convolution and batch normalization. Parallelism in these systems is implied by the fact that these systems typically operate on large arrays and array elements can usually be processed in parallel. Their design decisions are usually surrounding **immutable, dense tensors** (e.g., feature maps) with element-wise or pre-programmed opera-

tions.

Taichi has different design goals. Since Taichi programmers have finer programming granularity, Taichi is more suitable for relatively more irregular computational patterns. For example, the design of Taichi has advantages over deep learning frameworks on

- Computer graphics, including physical simulation and rendering;
- Irregular neural network layers (e.g., gathering/scattering) that are emerging;
- General differentiable programming cases.

Without Taichi, to compose computation that is less regular than classical neural network operations, developers have to manually write low-level CUDA kernels, or *abuse* element-wise operations in deep learning frameworks. For example, with the gather/scatter operations, it is possible to represent an explicit parallel for-loop in systems like TensorFlow or JAX. However, the developer will then have to resort to the compiler to fuse the operations and eliminate intermediate tensors. This may be less productive and efficient compared to programming in Taichi's megakernel (SPMD) interface.

Differentiability is a common feature of both Taichi and deep learning frameworks. However, Taichi has a two-scale system for differentiability while deep learning frameworks typically only need one scale. Taichi uses source-code transform inside kernels, and a lightweight tape outside the kernels, while deep learning systems such as JAX directly differentiates the computation on the single-layer computational graph. The reason why Taichi needs source-code transform is that computer graphics (especially physical simulation) application typically involves fine-grained operations with complex control flow, and source-code transform has minimal runtime cost for differentiation. In contrast, in deep learning workloads, each low-level operation tends to be relatively coarser-grained, so overheads of a runtime automatic differentiation system can usually be effectively amortized into each element or iteration of the operation and ultimately become negligible.

1 .3 Taichi v.s. Halide

Key difference: decoupling algorithms from *data structures* v.s. decoupling algorithms from *schedules*.

Halide [102] is a piece of seminal work that decouples *algorithms* (such as image processing operations) from *schedules*. *Schedules* in Halide include loop transformations and vectorization that change the way how computation maps to hardware while preserving the computation ouptuts. Some recent polyhedral compilers adopt similar ideas [7, 88, 120, 8]. Taichi chooses a different way to decompose programs: it decouples algorithms from the *internal organization of (sparse) data structures*, allowing programmers to quickly switch between data organizations to achieve high performance.

Computation-wise, Halide adopts array-based parallelism, just like deep learning frameworks. The benefit is clear: array-based parallelism preserves more high-level program structures and opens up the space to decouple schedules from algorithms. In Taichi, however, programmers code in monolithic megakernels with complex control flow and random accesses, the compiler then automatically applies a *subset* of scheduling optimizations that Halide does, including automatic vectorization and parallelization. This leads to more expressive but less structured code in Taichi than in Halide.

A megakernel in Taichi can be considered a fully fused pipeline in Halide, which means the programmer loses some flexibility in controlling recomputation. While recomputation plays an important role in image processing, practically we rarely find recomputation is needed in Taichi workloads (especially simulations) to improve performance.

1 .4 Taichi v.s. Liszt, Ebb, and Simit

Key difference: sparse grids v.s. meshes.

Physical simulation DSLs such as Liszt, Ebb and Simit usually abstract simula-

tion domains as a graph structure to represent unstructured meshes. For example, Liszt [30] focuses on solving partial differential equations on meshes. Simit [69] models the domain as sparse matrices while Ebb [13] employs a relational data model. On unstructured meshes, these languages offer a higher level of abstraction than Taichi, and has higher productivity. The strength of Taichi in simulation is mainly on hierarchical sparse data structures. We leave more discussions on potential mesh data structure support in Taichi to the next section.

2 Future work

We believe the following directions can be meaningful future work to extend Taichi:

Reusable IR for visual computing Currently Taichi only has a Python frontend, and it would be meaningful to reuse Taichi IR to build new programming languages. A set of reusable Taichi IR can easily deliver advanced features such as spatial sparsity, automatic differentiation, and quantization to other visual computing developers. Note that a new frontend based on Taichi IR does not have to be a programming language: visual programming, especially those based on node graphs (see, e.g., the node systems in digital content creation systems such as Autodesk Maya and SideFX Houdini), are expressive and intuitive ways to develop visual computing programs too. Creating a visual programming system based on Taichi IR will bring our infrastructure to a much wider range of users, especially artists who prefer node graphs to coding, when expressing their creative ideas.

Interfacing with sparse linear algebra The sparse computation system in Taichi is best suited for spatial sparsity. Matrix-free solvers (e.g., multigrid-preconditioned conjugated gradients on sparse and structured grids) can be easily implemented in the form of highly efficient stencils. However, matrix-free solvers may not always be the optimal solution for all users. For example, some users may want to simply create a sparse matrix (e.g., in CSR or COO format) and leverage mature sparse

linear solvers such as PARDISO [6]. It would he helpful to introduce sparse matrices to Taichi and extend the kernel language to build these matrices efficiently in parallel.

Auto-tuning Since Taichi decouples data structure (chapter 2) and (quantized) data format (chapter 5) from computation, it is possible to let an auto-tuner try different data layouts and formats, and pick one with the optimal runtime performance and memory efficiency. For example, when a user wants to figure out a tailored sparse data structure on a specific computer architecture, having an auto-tuning system to do that automatically can greatly reduce the burden on the programmer. It also makes sense to let the auto-tuner figure out a good quantization scheme with good performance and compression ratio. Other tunable parameters include GPU block dim and kernel fusion strategy.

First-class support for other data structures So far the only data structure that Taichi has first-class support for is spatially sparse multidimensional arrays. It would be meaningful to extend Taichi to include first-class support for various other data structures such as unstructured meshes.

- **Triangular and tetrahedral meshes** are conformal to simulation boundaries, offering higher accuracy near boundary conditions. Furthermore, meshes can be easily made adaptive. Meshes are frequently used together with discretization schemes such as finite volume and finite elements, and play an important role in computer-aided engineering (CAE) applications. Currently, users have to manually compose mesh data (e.g., face indices of tetrahedrons and normal velocity on faces) using 1D, dense data structures in Taichi. The Taichi compiler does not have first-class support and a high-level understanding of the represented mesh structures. Furthermore, meshes have various ways to store the internal data, and the data structure to store the indices of vertices, edges, faces, and elements have a critical impact on memory consumption and run-time performance.

- **Adaptive structured grids** provide opportunities to leverage both adaptivity and highly efficient memory accesses. A domain-specific compiler can help to generate highly efficiently parallel stencil kernels, specialized at each level of detail. Furthermore, cells at the boundary of a resolution level need special treatment and often result in less efficient kernels compared to those cells that do not involve resolution change. This opens up more space for specialization: a high-performance compiler can choose to generate simple, efficient kernels in uniform-resolution regions, and relatively less efficient kernels in regions where resolution changes. The runtime system can then dispatch ($8 \times 8 \times 8$ blocked) cells to different kernels. All these should be made transparent to programmers.

The idea to decouple data structure from computation can be reused in these future directions. Ideally, combining unstructured meshes and structured grids in a single program, and providing a unified programming interface on these data structures, would be extremely helpful to scientific computing developers.

Bibliography

[1] Mridul Aanjaneya, Ming Gao, Haixiang Liu, Christopher Batty, and Eftychios Sifakis. Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM Transactions on Graphics (TOG)*, 36(4):1–12, 2017.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[4] Mike Acton. Data-oriented design and C++, 2014.

[5] Aseem Agarwala. Efficient gradient-domain compositing using quadtrees. 26(3):94, 2007.

[6] Christie Alappat, Achim Basermann, Alan R Bishop, Holger Fehske, Georg Hager, Olaf Schenk, Jonas Thies, and Gerhard Wellein. A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication. *ACM Transactions on Parallel Computing (TOPC)*, 7(3):1–37, 2020.

[7] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, et al. PENCIL: A platform-neutral compute intermediate language for accelerator programming. In *Parallel Architecture and Compilation*, pages 138–149. IEEE, 2015.

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and

Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. pages 193–205, 2019.

[9] Dan Bailey, Ian Masters, Matt Warner, and Harry Biddle. Simulating fluids using a coupled voxel-particle data model. In *ACM SIGGRAPH 2013 Talks*, page 15. ACM, 2013.

[10] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. 2016.

[11] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, volume 1, pages 3–10, 2010.

[12] Gilbert Louis Bernstein and Fredrik Kjolstad. Perspectives: Why new programming languages for simulation? *ACM Transactions on Graphics (TOG)*, 35(2):1–3, 2016.

[13] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Transactions on Graphics*, 35(2):21:1–21:12, 2016.

[14] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[15] Robert Edward Bridson. *Computational Aspects of Dynamic Surfaces*. PhD thesis, Stanford University, Stanford, CA, USA, 2003.

[16] Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, and Ollie Wild. Whopr-fast and scalable whole program optimizations in gcc. *Initial Draft*, 12, 2007.

[17] Erin Catto. Modeling and solving constraints. In *Game Developers Conference*, page 16, 2009.

[18] Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. A compositional object-based approach to learning physical dynamics. *ICLR*, 2016.

[19] Jiawen Chen, Dennis Bautembach, and Shahram Izadi. Scalable real-time volumetric surface reconstruction. 32(4):113:1–113:16, 2013.

[20] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis. pages 13:1–13:13. ACM, 2017.

[21] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. Parsy: Inspection and transformation of sparse matrix computations for parallelism. In *International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 62, 2018.

[22] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):123, 2018.

[23] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[24] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[25] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[26] Frederica Darema, David A George, V Alan Norton, and Gregory F Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[27] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, pages 7178–7189, 2018.

[28] Filipe de Avila Belbute-Peres, Kevin A Smith, Kelsey Allen, Joshua B Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Neural Information Processing Systems*, 2018.

[29] Jonas Degrave, Michiel Hermans, Joni Dambre, et al. A differentiable physics engine for deep learning in robotics. *arXiv preprint arXiv:1611.01652*, 2016.

[30] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. Liszt: A domain specific language for building portable mesh-based pde solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9, 2011.

[31] Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. Sorted deferred shading for production path tracing. In *Computer Graphics Forum*, volume 32, pages 125–132. Wiley Online Library, 2013.

[32] Ronald M Errico. What is an adjoint model? *Bulletin of the American Meteorological Society*, 78(11):2577–2592, 1997.

[33] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing cuda code by kernel fusion: application on blas. *The Journal of Supercomputing*, 71(10):3934–3957, 2015.

[34] Steven W Gagniere, David AB Hyde, Alan Marquez-Razon, Chenfanfu Jiang, Ziheng Ge, Xuchen Han, Qi Guo, and Joseph Teran. A hybrid lagrangian/eulerian collocated advection and projection method for fluid simulation. *arXiv preprint arXiv:2003.12227*, 2020.

[35] Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana-Tampubolon, Eftychios Sifakis, Yuksel Cem, and Chenfanfu Jiang. GPU optimization of material point methods. 32(4):102, 2018.

[36] Richard Gordon, Robert Bender, and Gabor T Herman. Algebraic reconstruction techniques (art) for three-dimensional electron microscopy and x-ray photography. *Journal of theoretical Biology*, 29(3):471–481, 1970.

[37] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 3D semantic segmentation with submanifold sparse convolutional networks. pages 9224–9232, 2018.

[38] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

[39] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018.

[40] Laurent Hascoet and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. 39(3):20:1–20:43, 2013.

[41] Rama Karl Hoetzlein. Gvdb: Raytracing sparse voxel database structures on the GPU. In *Proceedings of High Performance Graphics*, pages 109–117. Eurographics Association, 2016.

[42] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P Sadayappan. Gpu code optimization using abstract kernel emulation and sensitivity analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 736–751, 2018.

[43] Ben Houston, Michael B. Nielsen, Christopher Batty, Ola Nilsson, and Ken Museth. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Trans. Graph.*, 25(1):151–175, 2006.

[44] Ben Houston, Michael B Nielsen, Christopher Batty, Ola Nilsson, and Ken Museth. Hierarchical rle level set: A compact and versatile deformable surface representation. *ACM Transactions on Graphics (TOG)*, 25(1):151–175, 2006.

[45] Yuanming Hu. Taichi: An open-source computer graphics library. *arXiv preprint arXiv:1804.09293*, 2018.

[46] Yuanming Hu. The taichi programming language. In *ACM SIGGRAPH 2020 Courses*, pages 1–50. 2020.

[47] Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. Difftaichi: Differentiable programming for physical simulation. *ICLR*, 2020.

[48] Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. 37(4):150, 2018.

[49] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6):201, 2019.

[50] Yuanming Hu, Jiafeng Liu, Xuanda Yang, Mingkuan Xu, Ye Kuang, Weiwei Xu, Qiang Dai, William T. Freeman, and Frédo Durand. Quantaichi: A compiler for quantized simulations. *ACM Transactions on Graphics (TOG)*, 40(4), 2021.

[51] Yuanming Hu, Jiancheng Liu, Andrew Spielberg, Joshua B Tenenbaum, William T Freeman, Jiajun Wu, Daniela Rus, and Wojciech Matusik. Chainqueen: A real-time differentiable physical simulator for soft robotics. *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[52] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[53] IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[54] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckus, Elliot Saba, Viral B Shah, and Will Tebbutt. Zygote: A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*, 2019.

[55] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[56] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[57] Wenzel Jakob. Mitsuba renderer, 2010. http://www.mitsuba-renderer.org.

[58] Wenzel Jakob. Enoki: structured vectorization and differentiation on modern processor architectures, 2019. https://github.com/mitsuba-renderer/enoki.

[59] Chenfanfu Jiang. *The material point method for the physics-based simulation of solids and fluids*. University of California, Los Angeles, 2015.

[60] Chenfanfu Jiang, Craig Schroeder, Andrew Selle, Joseph Teran, and Alexey Stomakhin. The affine particle-in-cell method. *ACM Transactions on Graphics (TOG)*, 34(4):1–10, 2015.

[61] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[62] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[63] Ralf Karrenberg and Sebastian Hack. Whole-function vectorization. In *Code Generation and Optimization*, pages 141–150, 2011.

[64] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3907–3916, 2018.

[65] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Eurographics Symposium on Geometry Processing*, volume 7, 2006.

[66] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice*. CRC Press, 2017.

[67] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[68] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[69] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, 2016.

[70] Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 107–120, 1998.

[71] Stig Larsson and Vidar Thomée. *Partial differential equations with numerical methods*, volume 45. Springer Science & Business Media, 2008.

[72] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. 2004.

[73] Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. Vectorized production path tracing. In *High Performance Graphics*, 2017.

[74] Roland Leißa, Sebastian Hack, and Ingo Wald. Extending a C-like language for portable SIMD programming. 47(8):65–74, 2012.

[75] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. The deep learning compiler: A comprehensive survey. *arXiv preprint arXiv:2002.03794*, 2020.

[76] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. In *SIGGRAPH Asia 2018 Technical Papers*, page 222. ACM, 2018.

[77] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (TOG)*, 37(4):139, 2018.

[78] Junbang Liang, Ming C Lin, and Vladlen Koltun. Differentiable cloth simulation for inverse problems. *Advances in Neural Information Processing Systems*, 2019.

[79] Haixiang Liu, Yuanming Hu, Bo Zhu, Wojciech Matusik, and Eftychios Sifakis. Narrow-band topology optimization on a sparsely populated grid. 37(6):251:1–251:14, 2018.

[80] Haixiang Liu, Nathan Mitchell, Mridul Aanjaneya, and Eftychios Sifakis. A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Transactions on Graphics (TOG)*, 35(6):1–12, 2016.

[81] Matthew M Loper and Michael J Black. Opendr: An approximate differentiable renderer. In *European Conference on Computer Vision*, pages 154–169. Springer, 2014.

[82] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. volume 23, pages 457–462. ACM, 2004.

[83] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, 2015.

[84] Dror E Maydan, Saman P Amarasinghe, and Monica S Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 2–15, 1993.

[85] Aleka McAdams, Andrew Selle, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. Computing the singular value decomposition of 3x3 matrices with minimal branching and elementary floating point operations. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2011.

[86] Aleka McAdams, Eftychios Sifakis, and Joseph Teran. A parallel multigrid poisson solver for fluids simulation on large grids. In *Symposium on Computer Animation*, pages 65–74. ACM/Eurographics Association, 2010.

[87] Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Li Fei-Fei, Joshua B Tenenbaum, and Daniel LK Yamins. Flexible neural representation for physics prediction. *1806.08047*, 2018.

[88] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, 2015.

[89] K. Museth. Vdb: High-resolution sparse volumes with dynamic topology. *ACM Trans Graph*, 32(3):27, 2013.

[90] Ken Museth. Hierarchical digital differential analyzer for efficient ray-marching in openvdb. 2014.

[91] Ken Museth, Jeff Lait, John Johanson, Jeff Budsberg, Ron Henderson, Mihai Alden, Peter Cucka, David Hill, and Andrew Pearce. Openvdb: an open-source data structure and toolkit for high-resolution volumes. In *Acm siggraph 2013 courses*, pages 1–1. 2013.

[92] Michael B Nielsen and Robert Bridson. Spatially adaptive flip fluid simulations in bifrost. In *ACM SIGGRAPH 2016 Talks*, page 41. ACM, 2016.

[93] Michael B. Nielsen and Ken Museth. Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets. *J. Sci. Comput.*, 26(3):261–299, 2006.

[94] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), November 2019.

[95] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on hamilton-jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988.

[96] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[97] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems*, 30(2):7:1–7:36, 2008.

[98] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. pages 101–108. ACM, 1997.

[99] Matt Pharr and William R Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing*, pages 1–13, 2012.

[100] Jovan Popović, Steven M Seitz, Michael Erdmann, Zoran Popović, and Andrew Witkin. Interactive manipulation of rigid body simulations. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 209–217. ACM Press/Addison-Wesley Publishing Co., 2000.

[101] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. Automatic kernel fusion for image processing dsls. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 76–85, 2018.

[102] 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. 31(4):32:1–32:12, 2012.

[103] 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.

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

[105] Stéphane Redon, Abderrahmane Kheddar, and Sabine Coquillart. Fast continuous collision detection between rigid bodies. In *Computer graphics forum*, volume 21, pages 279–287. Wiley Online Library, 2002.

[106] Gernot Riegler, Ali Osman Ulusoy, and Andreas Geiger. Octnet: Learning deep 3d representations at high resolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3577–3586, 2017.

[107] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[108] Connor Schenck and Dieter Fox. Spnets: Differentiable fluid dynamics for deep neural networks. *arXiv preprint arXiv:1806.06094*, 2018.

[109] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *Journal of Scientific Computing*, 35(2-3):350–371, 2008.

[110] R. Setaluri, M. Aanjaneya, S. Bauer, and E. Sifakis. Spgrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans Graph*, 33(6):205, 2014.

[111] Jos Stam. Stable fluids. In *Siggraph*, volume 99, pages 121–128, 1999.

[112] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. *ACM Transactions on Graphics (TOG)*, 32(4):102, 2013.

[113] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[114] Bjarne Stroustrup. Foundations of c++ (etaps 2012 keynote), 2012. https://www.stroustrup.com/ETAPS-corrected-draft.pdf.

[115] Deborah Sulsky, Shi-Jian Zhou, and Howard L Schreyer. Application of a particle-in-cell method to solid mechanics. *Computer physics communications*, 87(1-2):236–252, 1995.

[116] Andre Pradhana Tampubolon, Theodore Gast, Gergely Klár, Chuyuan Fu, Joseph Teran, Chenfanfu Jiang, and Ken Museth. Multi-species simulation of porous sand and water mixtures. *ACM Transactions on Graphics (TOG)*, 36(4):1–11, 2017.

[117] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[118] Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schuller. *Multigrid*. Elsevier, 2000.

[119] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. Openad/f: A modular open-source tool for automatic differentiation of fortran codes. 34(4):18, 2008.

[120] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv:1802.04730*, 2018.

[121] Jui-Hsien Wang, Ante Qu, Timothy R Langlois, and Doug L James. Toward wave-based sound synthesis for computer animation. *ACM Trans. Graph.*, 37(4):109–1, 2018.

[122] Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun, and Xin Tong. O-CNN: Octree-based convolutional neural networks for 3D shape analysis. *ACM Transactions on Graphics (SIGGRAPH)*, 36(4), 2017.

[123] Peng-Shuai Wang, Chun-Yu Sun, Yang Liu, and Xin Tong. Adaptive O-CNN: A patch-based deep representation of 3D shapes. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 37(6), 2018.

[124] Xinlei Wang, Yuxing Qiu, Stuart R Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, and Chenfanfu Jiang. A massively parallel and scalable multi-cpu material point method. *ACM Transactions on Graphics (TOG)*, 39(4):30–1, 2020.

[125] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. 51(8):11:1–11:12, 2016.

[126] Gregory J Ward. The radiance lighting simulation and rendering system. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 459–472, 1994.

[127] Richard Wei, Marc Rasi Dan Zheng, and Bart Chrzaszcz. Differentiable programming mega-proposal. https://github.com/apple/swift/blob/master/docs/DifferentiableProgramming.md, 2019. Accessed: 2019-09-25.

[128] R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, aug 1964.

[129] E Woodcock, T Murphy, P Hemmings, and S Longworth. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Applications of Computing Methods to Reactor Problems*, volume 557, 1965.

[130] Jun Wu, Christian Dick, and Rüdiger Westermann. A system for high-resolution topology optimization. *IEEE transactions on visualization and computer graphics*, 22(3):1195–1208, 2015.

[131] Kui Wu, Nghia Truong, Cem Yuksel, and Rama Hoetzlein. Fast fluid simulations with sparse volumes on the GPU. In *Computer Graphics Forum (Proc. Eurographics)*, volume 37, pages 157–167. Wiley Online Library, 2018.

[132] Jonas Zehnder, Rahul Narain, and Bernhard Thomaszewski. An advection-reflection solver for detail-preserving fluid simulation. *ACM Transactions on Graphics (TOG)*, 37(4):1–8, 2018.

[133] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. GraphIt: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):121, 2018.

[134] Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Transactions on Graphics (TOG)*, 24(3):965–972, 2005.