AsyncTaichi: Whole-Program Optimizations for Megakernel Sparse Computation and Differentiable Programming

YUANMING HU*, MIT CSAIL
MINGKUAN XU*, Tsinghua University
YE KUANG, Tsinghua University
FRÉDO DURAND, MIT CSAIL

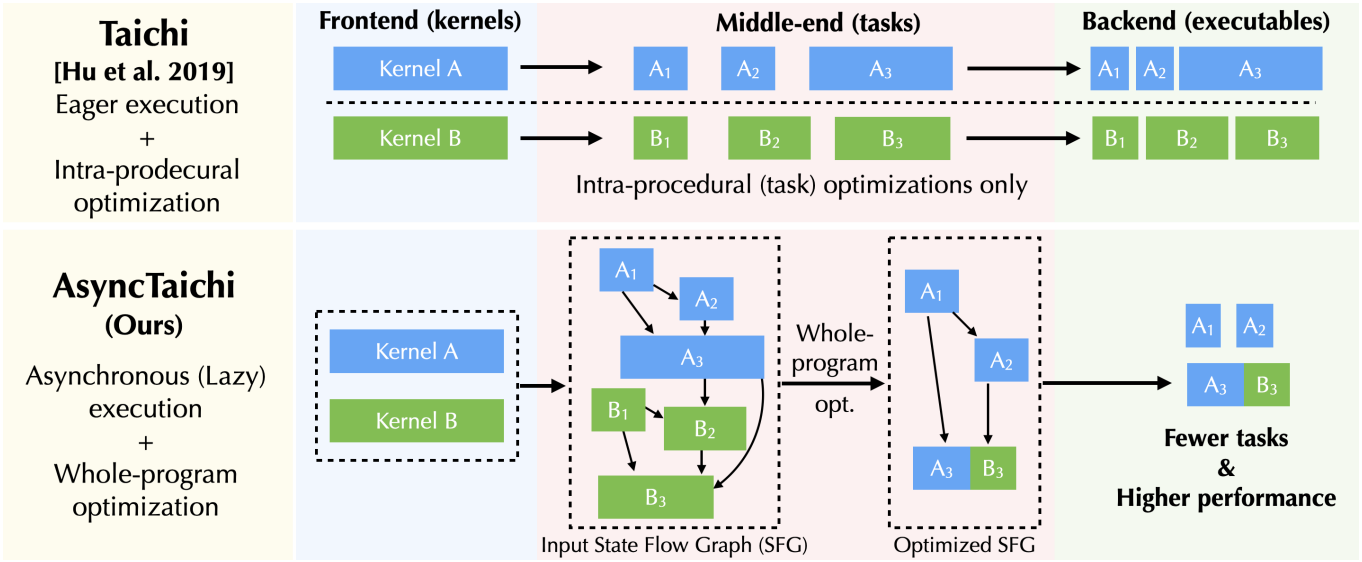


Fig. 1. **Top:** In the original Taichi system [Hu et al. 2019], computational kernels are eagerly compiled and launched. Therefore there is no chance for the optimizer to optimize beyond a single kernel. **Bottom:** In our work, we accumulate kernels in a execution buffer, only flushing the execution 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, so that sparse computation kernels can be optimized at a whole-program level. After a suite of domain-specific optimization passes including list generation removal, sparse data structure activation elimination, the tasks are much better optimized compared to those in the original Taichi system. As a result, the whole-program optimized Taichi programs run much faster on parallel devices.

We present a whole-program optimization framework for the Taichi programming language. As an *imperative* language tailored for *sparse* and *differentiable* computation, Taichi's unique computational patterns lead to attractive optimization opportunities that do not present in other compiler or runtime systems. For example, to support iteration over sparse voxel grids, excessive list generation tasks are often inserted. By analyzing sparse computation programs at a higher level, our optimizer is able to remove the majority of unnecessary list generation tasks. To provide maximum programming flexibility, our optimization system conducts *on-the-fly* optimization of the whole computational graph consisting of Taichi kernels. The optimized Taichi kernels are then just-in-time compiled in parallel, and dispatched to parallel devices such as multithreaded CPU and massively parallel GPUs. *Without any code modification* on Taichi programs, our new system leads to 3.07 – 3.90× fewer kernel launches and 1.73 – 2.76× speed up on our benchmarks including sparse-grid physical simulation and differentiable programming.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; • **Computing methodologies** → **Parallel programming languages**; **Physical simulation**.

Additional Key Words and Phrases: Sparse Data Structures, GPU Computing.

ACM Reference Format:
Yuanming Hu, Mingkuan Xu, Ye Kuang, and Frédo Durand. 2020. AsyncTaichi: Whole-Program Optimizations for Megakernel Sparse Computation and Differentiable Programming. 1, 1 (December 2020), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Both authors contributed equally to this work.

Authors' addresses: Yuanming Hu, MIT CSAIL, yuanming@mit.edu; Mingkuan Xu, Tsinghua University, xmk17@mails.tsinghua.edu.cn; Ye Kuang, Tsinghua University, ykuang.me@gmail.com; Frédo Durand, MIT CSAIL, fredo@mit.edu.

XXXX-XXXX/2020/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Taichi is a new programming language for spatially sparse [Hu et al. 2019] and differentiable programming [Hu et al. 2020]. It has shown its productivity and performance, in forward physical simulation, and deep learning tasks with differentiable physical modules. However, existing Taichi compilers optimizations are restricted within kernels, without high-level views of the whole program. This motivates a new system that can conduct whole-program automatic performance optimizations for Taichi programs.

Whole-program optimization, also known as "link-time optimization" or "interprocedural optimization" in some literature, is not a new idea. For example, in traditional ahead-of-time compilers such as `gcc`, whole program optimizations can be easily achieved using the `-flto` flag. In more dynamic array programming languages such as JAX/HLO [Bradbury et al. 2018], array-level fusion [Frostig et al. 2018] has been an effective optimization. However, three major challenges do exist when applying the idea of whole-program optimization to Taichi.

Firstly, Taichi eagerly launches computational tasks. The existing Taichi system just-in-time (JIT) compiles and then launches every kernel immediately after it is called. This eager execution scheme destroys opportunities to analyze and optimize beyond a single kernel.

Secondly, Taichi is imperative, has mutable data buffers and needs to support spatially sparse programming. Unlike systems that rely on dense, immutable, and holistic arrays ("tensors" in systems like TensorFlow), partial updates and mutable, sparse buffers can lead to difficulties when analyzing a Taichi program.

Thirdly, the spatially sparse programming feature of Taichi can lead to further complexities to the analysis and optimization process. While tools for analyzing dense array programs are well established, their counterparts in sparse array computations are largely underexploited.

In this work, we propose a *state-flow formulation* of Taichi programs for analyzing Taichi programs, a new *asynchronous execution engine* for Taichi that opens up spaces for performance optimization, and most importantly, a whole-program optimizer for imperative spatially sparse and differentiable computation.

In contrast to previous Taichi system that does no whole program optimization, in this work we build a dynamic graph consisting of pending kernels, then compile and run them lazily. This enables us to perform cross-kernel optimizations ranging from common optimizations such as *kernel fusion* to sparse-computation-specific *activation/list generation removal*. We can also perform parallel compilation, which significantly reduces compilation time.

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

- **Transparent to users.** No code modification is needed for users to leverage the new execution and optimization system.
- **Compile Just-in-time.** The just-in-time compilation system in Taichi has proven to bring users great flexibility and performance, since JIT *delays* the need of value of "compile-time constant" values. For example, Δt in physical simulators is often run-time variable in ahead-of-time compilation, but with JIT Δt would be a compile-time constant, which allows the compiler to do more optimizations such as constant folding.
- **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 $100,000,000$ particles. For small-scale tasks, *compilation* time may be the bottleneck; for large scale tasks, *computation* time is more important.

The rest of this paper covers our detailed design decisions motivated by the design guidelines. We summarize our contributions as follows:

(1) A state-flow formulation of spatially sparse computation. The resulted *state-flow graph (SFG)* serves as a high-level intermediate representation (IR) of a Taichi program that captures essential information to analyze sparse computation.
(2) An asynchronous task execution scheme for the Taichi programming language that enables parallel compilation and exposes whole-program optimization opportunities;
(3) Most importantly, *a whole-program optimizer* for asynchronous spatially sparse and differentiable computation, leveraging the state-flow graph for performance optimizations;
(4) A systematic study of the resulted asynchronous Taichi system. Based on the benchmarks, we show $1.73-2.76\times$ (geometric mean) execution time improvements over the traditional synchronous Taichi system without whole-program optimizations [Hu et al. 2019].

2 RELATED WORK

Sparse data structures. Taichi directly draws inspirations from popular sparse data structures in computer graphics, including VDB [Hoetzlein 2016; Museth et al. 2013; Wu et al. 2018] and SPGrid [Gao et al. 2018; Setaluri et al. 2014]. 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 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 that would need a whole-program analysis to optimize.

Another thread of work on sparse computation is sparse linear algebra languages, such as TACO [Chou et al. 2018; Kjolstad et al. 2017], which can effectively generate kernels for Einstein summations on sparse matrices and tensors. Instead of explicitly building the sparse matrices, Taichi encourages *matrix-free* linear algebra computations, which are often the more effective way for high-performance linear algebra solves for physical simulation (see, for example [Liu et al. 2018]).

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 cannot directly represent array states, where partial updates may happen. Array SSA forms have been proposed and successfully adopted in parallelization [Knobe and Sarkar 1998] and array privatization [Maydan et al. 1993]. 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 Interprocedural 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. While WPO is extensively explored in classical compiling systems, it is still underexploited for spatially sparse computation. The unique computational pattern in Taichi brings higher complexity and the need for a unified high-level intermediate representation for analysis and optimization.

Compute 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 compute graph: *immutable, dense feature maps* directly map to compute graph *edges*, and *operators* (such as convolutions, max pooling, and element-wise add) maps to compute graph *nodes*. High-level optimizations on the compute graph have been a popular feature in deep learning frameworks. The HLO IR of XLA and PyTorch GLOW [Rotem et al. 2018] are representative examples. Based on the compute graph, traditional computer optimizations such as operator fusion, dead code elimination (DCE), common subexpression elimination (CSE) can be applied. We refer the readers to [Li et al. 2020] for a good survey. Our system is similar to these systems in that a high-level graph-based IR is used, yet the high-level IR for Taichi must consider its sparse, imperative, and megakernel nature.

3 TAICHI BACKGROUND: IMPERATIVE, MEGAKERNEL, SPARSE, AND DIFFERENTIABLE PROGRAMMING

Taichi [Hu et al. 2019] 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-intense kernels (Megakernels, similar to a `__global__` GPU kernel in CUDA) into parallel executables. Users can flexibly launch the kernels using Python. Key features of Taichi are described below.

Data-oriented programming. `field` is the key concept in Taichi that represents data. A `field` can not only represent one- to eight-dimensional dense tensors, but also their sparse variants. Taichi also allows programmers to change data layouts using a data layout description language, which can easily describe common data layouts such as array-of-structures (AOS) or structures-of-arrays (SOA). Taichi *decouples algorithms from data structures*. This allows the users to exploit the vast design space of data structures and layouts, so that maximum memory performance (e.g., cache hit-rate and cacheline utilization) can be achieved via rapid trial-and-error.

Sparse programming. is a unique feature [Hu et al. 2019] of Taichi. Most 3D graphics data (especially those stored on voxel grids) are spatially sparse. Taichi has first-class support for sparse data structures. In order to make sparse data structures as easy to use as dense data structures, various designs are made on the syntax, compiler, and runtime levels:

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

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

List generation is the key mechanism to achieve efficient sparse for-loops (Fig. 2).

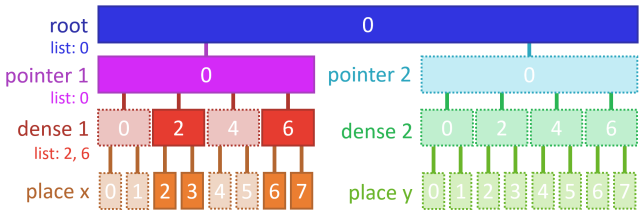


Fig. 2. The structure of `ti.root.pointer(ti.i, 4).dense(ti.i, 2).place(x)` in Taichi, a two-level 1D sparse data structure with the upper level being a pointer array and the lower level being dense blocks. Highlighted cells are active. Lists of each layer are defined to be a collection of active node indices.

- *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 corresponding voxels of `y` does not have to be explicit activated before this for loop. Taichi will automatically activate `y[i // 2, j // 2, k // 2]` and zero-fill the initial data. See Figure 3 for an example.

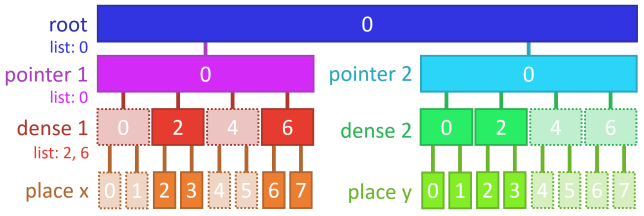


Fig. 3. Execution result of simple program `for i in x: y[i // 2] += 1`. `y[1]` and `y[3]` are activated on write. Because of the constraints of the dense node, `y[0]` and `y[2]` are also activated.

- *Automatic memory management* frees the user from worrying about memory allocation and deallocation. Taichi's high-performance memory allocator will automatically manage sparse data structure nodes.
- *Programmable megakernels* allows users to easily write parallel programs with very high flexibility and rich expressiveness. Taichi kernels allow complex control flows - in fact, users can easily write a recursive ray tracer (See, e.g., [Hu et al. 2019]).
- *Automatic parallelization* Taichi *kernels* are decomposed into *tasks* that are serial or parallel.

Differentiable Programming in Taichi. Unlike some other differentiable programming systems for deep learning such as TensorFlow and PyTorch that operate on functional ("immutable") buffers, Taichi is imperative (providing "if" statements, serial and parallel "for" loops). The imperative nature makes it easier to port most physical simulation code written in popular languages like C++, to the Taichi programming than to other modern functional programming languages such as TensorFlow. Taichi uses a novel *two-scale reverse-mode automatic differentiation* [Hu et al. 2020]: a light-weight gradient tape stores kernel launches for end-to-end simulation differentiation, and source-code transform is used for for differentiating within kernels. This design ensures that the gradient versions of the megakernels are still megakernels, and naturally uses global fields as checkpoints for gradient computation.

Observations. Many potential performance optimizations in Taichi programs need a *whole-program* view, which motivates the remaining of this work. For example, sparse struct-for loops may easily incur redundant list generation in the data structure tree. Eliminating these unnecessary list generations needs a whole-program understanding of the program.

4 A STATE-FLOW FORMULATION OF SPARSE COMPUTATION

In imperative programming, *Program = State + Compute*. Specifically in Taichi, because of the existence of sparse programming support, "state" means way more than values of `fields`, and "compute" means more than GPU kernels that operates on data. This creates more challenges on modelling and analyzing Taichi programs, compared to traditional imperative programming languages (such as C++) and array-based systems such as TensorFlow.

In TensorFlow, every operation creates a new, immutable buffer ("tensor"). In simulation, we have to go imperative, not only because graphics programmers have been accustomed to using imperative programming for decades, but also because in-place operations in imperative programming offer significant performance advantages in graphics applications, such as physical simulation.

We reformulate the imperative computation scheme of Taichi into a collection of *states* and *tasks*. The ultimate goal of our IR design is to strengthen and simplify optimization rules for spatially sparse computation. To systematically optimize spatially sparse computation, we formulate an AsyncTaichi program as a state-flow graph (SFG), which is a 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, and SFGs can be considered as an extended version of data-flow analysis to handle spatially sparse computation.

Background: Structural Nodes (SNodes). In Taichi, 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). Taichi provides Structural Nodes (SNodes) to compose the hierarchy and particular properties. Commonly used SNodes include `dense`, `bitmasked`, `pointer`, `dynamic`.

4.1 States

States split the holistic description of a Taichi program into a suitable granularity for analysis and optimization. `dense` is the only SNode that has no sparsity information. Other SNodes can be spatially sparse, so we must decompose the holistic descriptions of their data into the following states:

- *A value state* simply represents the collection of numerical value stored in 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 and have the same meaning as those in data flow optimization. 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 value 0.
- *A mask state* cannot be treated as plain data flow since we need to understand it and do domain-specific optimization. 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. Essentially mask is not 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 a 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. See [Hu et al. 2019] for more details.
- *An allocator state* represents the state of Taichi's memory allocator. For computation that allocates/deallocates sparse data structure nodes, the allocator states are changed.

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

Taichi tasks. Each Taichi task has input edges (input states), output edges (modified states). It also maintains its own metadata, such as loop ranges (range/struct-fors). These edges and metadata will be used for whole-program optimization.

For each state, we use a *latest state tracker* to track which task holds the latest version of this state.

4.2 State-flow chains

Now let's focus on a single state. For simplicity, let's use value state S (Fig. 5), which is operated on by kernels 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"[1] is created. Clearly, only the latest writer holds the latest version of a state, while readers only fetches a copy without creating a new copy. If we only consider the writers, we basically get a chain structure for each state, with a few branches for readers, see the example in Fig. 5 as an concrete example.

Therefore, for a single state we can easily build a chain (a directed acyclic graph, DAG), which we call "state-flow chain" (SFC).

[1]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.

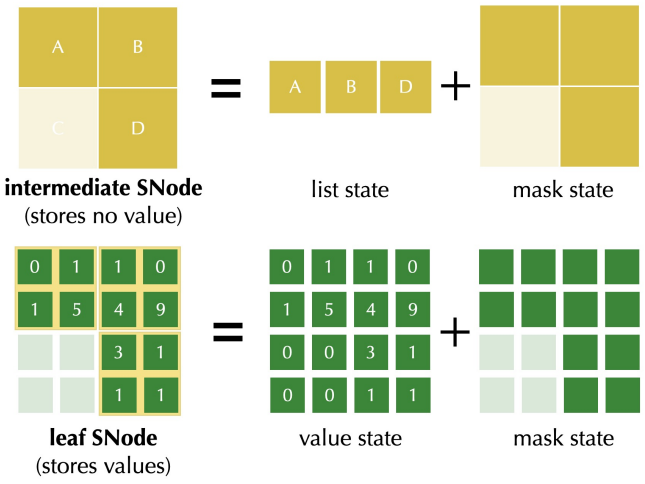


Fig. 4. State decomposition of a two-level sparse array, containing a sparse intermediate layer and a dense leaf layer. Note that the value state covers all pixels, even if the pixel is inactive. In other words, whenever an access reads a pixel from the sparse array, the mask state will first be queried. 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.

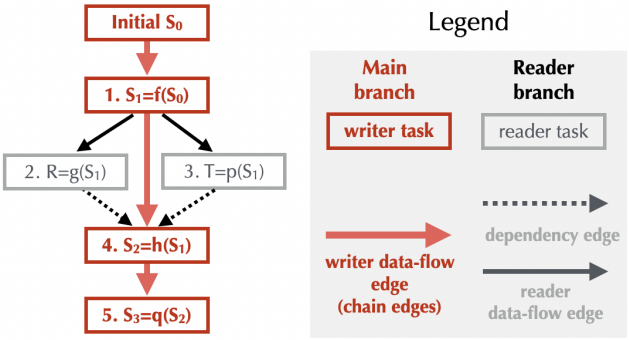


Fig. 5. A state-flow chain of value state S. The edges in the state-flow chain depicts the kernel dependency relationships. Note that each state-flow chain always has a main branch (write-after-write) and a reader branch (read-after-write & write-after-read). In the main branch, each node (task) creates a new version of the state.

4.3 State-flow graphs

A Taichi program can easily have hundreds of states. Here we introduce state-flow graphs (SFG), which are essentially state-flow chains sticking to together (Fig. 6). SFGs completely describes the relationship between tasks in Taichi. Since unions of DAGs are still DAGs, SFGs are DAGs too.

The SFG serves as the IR for whole-program sparse computation optimization. The allows us to use well-established graph theory languages for optimization (e.g., fusion as shown later).

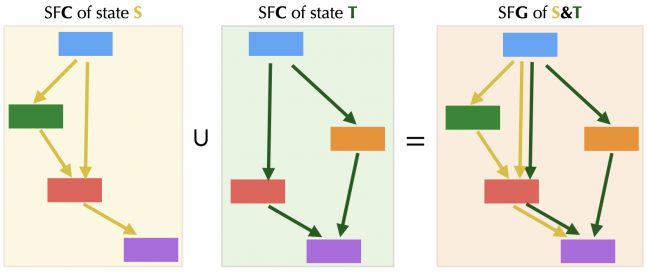


Fig. 6. A state-flow graph, by definition, is a union of state-flow chains of all the states used in a program.

Whenever a task is inserted into the execution queue, we dynamically create a SFG node and create corresponding dependency edges. SFG has 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 correctly handle edges that are connected to the removed node.

5 MAKING TAICHI ASYNCHRONOUS

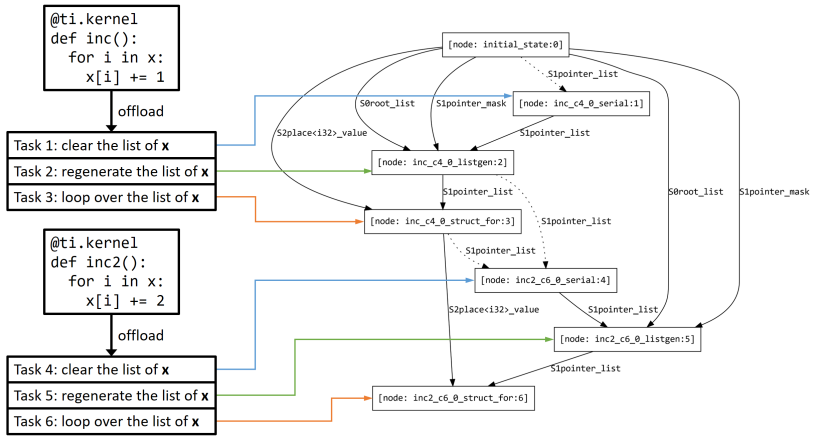
The existing Taichi system (JIT) compiles and then launches every kernel *eagerly*. This simple strategy actually prevents cross-kernel optimization from happening, since the system only sees one kernel at a time. Therefore, in order to make the SFG practically useful, we need to hold the SFG nodes from executing before optimizations are done.

This motivates us to develop an *asynchronous* execution engine for Taichi. By making Taichi *asynchronous*, we can obtain a list of kernels to compile and run *lazily*, and we can perform parallel compilation, which reduces compilation time. More importantly, we can perform kernel fusion, which reduces memory bandwidth consumption and has a direct contribution to performance.

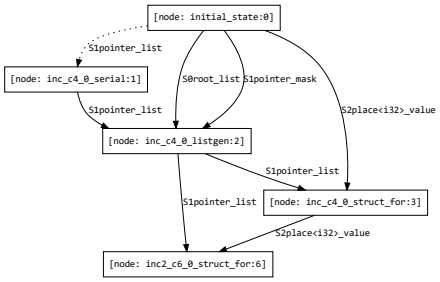
By-product: parallel compilation. As Taichi becomes more widely adopted, the compiler needs to deal with programs with increasing instructions and optimization passes. It it not uncommon that compilation can sometimes take more than 70% of program end-to-end run time. In the previous eager execution 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 wasted on compilation.

6 OPTIMIZE BEYOND KERNELS

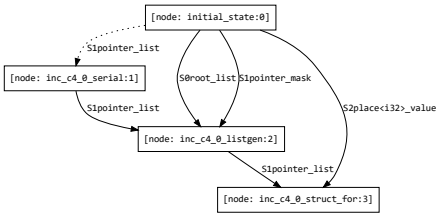
With the state-flow graph IR that describes the whole programs, and the asynchronous execution engine that saves the tasks from being



(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.

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

executed too early, we can finally conduct analyses and optimizations on the state-flow graph. In this section we discuss potential whole-program 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.

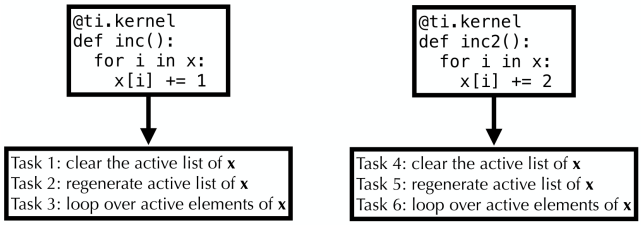


Fig. 8. The generated tasks of 2 kernels without kernel fusion. **x** is a sparse data structure.

As shown in Figure 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 8, and Taichi performs asynchronous computing, we can fuse the 2 kernels into 1 kernel, perform some analysis to know that the sparsity (i.e., which elements are active) is not changed, and finally get Figure 9 after optimizations. In this case,

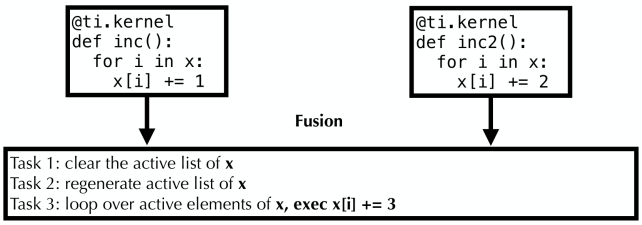


Fig. 9. The generated tasks of 2 kernels with kernel fusion.

we reduce the number of generated tasks from 6 to 3. *Kernel fusion is not new, but fusing kernels that operates on sparse data strictures 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 like 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.

Taichi's sparse computation model motivates us to apply the following domain-specific compiler optimizations:

- List generation removal
- Activation demotion
- Task 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 outputs 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 optimizations. For example, if two struct-for tasks are using the same list after list generation removal, a *task fusion* may be able to 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 activation was checked by not re-activated. This extra activation checking not only creates diverging instruction flow on CPU/GPUs that harms 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 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. 10):

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

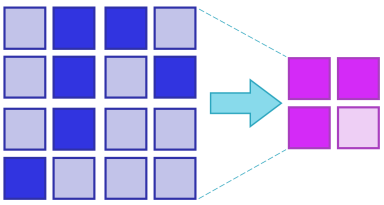


Fig. 10. The activation pattern of `for i, j in x: y[i // 2, j // 2] += x[i, j] * 0.25`. `x` is the grid on the left, and `y` is the grid on the right.

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 avoid false-positives mask state modifications, and can further bring down the list generation kernel tasks by 6.7× in the MGPCG example.

6.4 Task fusion

Clearly, we need to know the data dependency before we fuse (G2P2G, stencil etc, multigrid, multi-channel advection (improved cacheline utilization)) 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 intraprocedural optimizations, but it might be time-consuming when there are too many tasks.

6.5 Dead store elimination

We can also perform some cross-kernel analysis with asynchronous computing. For example, `ti.clear_all_gradients()` may excessively zero-fill unrelated tensors, 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 cases like this that a field is completely overwritten, our optimizer can eliminate the previous dead stores.

7 IMPLEMENTATION DETAILS

The whole-program 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 find to have direct impacts to performance.

7.1 Asynchronous Execution Engine

We implement an asynchronous execution engine that performs parallel compilation and kernel fusion.

We store all tasks into a queue until synchronization, which may happen when there is anything we need to output or the Python program embedding Taichi comes to an end. When synchronization happens, we check each task that which tensors' sparsity are changed, and remove list generation tasks when there is a previous one for the same tensor and there is no task between them changing the sparsity of the tensor.

After removing redundant list generation tasks, we check each adjacent tasks to see if they can be fused. For now, we only fuse tasks that loop over the active elements of the same tensor, or tasks that are simple parallel loops with the same range.

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 improve 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 meta of the IR.

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 Intermediate Representation (IR) is relatively hierarchical, we build a data-flow graph for data-flow analysis, to perform optimizations across the whole kernel including store-to-load forwarding, dead store elimination, and identical store/load elimination. For example, in Figure 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

Metrics. On each test case, we evaluate the performance with four metrics: execution time on the backend, number of tasks launched, number of instructions emitted, and number of tasks compiled. Each case is executed multiple times on CPU (x64) and GPU (CUDA) with a synchronization after each run in asynchronous mode, and the average running time is recorded.

Benchmark cases. We constructed 10 simply yet indicative microbenchmarks (tens of lines of code each) to unit test specific whole-program optimizations. Three more complex test cases (hundreds of lines of code each) tests the behavior of our optimizer on real-world programs.

8.1 Microbenchmarks

We constructed 10 microbenchmark cases to unit-test the system. The results are promising: Without code modification, the new system leads 3.9× fewer kernel launches on GPUs and 2.5× speed up on our benchmarks, as shown in Fig. 11. More details on the microbenchmarks are discussed in the appendix.

8.2 MacCormack advection

In this benchmark case we use the MacCormack advection scheme [Selle et al. 2008] with RK3 path integration, to advect three scalar physical fields. We follow recent trends to use collocated grids (see, e.g. [Gagniere et al. 2020; Nielsen and Bridson 2016]) to improve cacheline utilization. We find that on CUDA our optimizer leads to 1.53× performance boost, and 3.75× fewer tasks launched on both backends. The improved performance and reduced tasks launched in this benchmark is because of the task fusion optimization, and list generation removal.

8.3 Multigrid preconditioned conjugate gradients (MGPCG)

In this benchmark we use a sparsely populated region in a 512×512 domain. We follow the MGPCG solver design in [Hu et al. 2019]. Four multigrid levels are used, and for each level we use a two-level sparse grid. Notably, our optimizer is able to bring down the amount of tasks launched from 880, 820 to 177, 614, which is 5.0× fewer. This is because the restriction, smoothing, and prolongation operations leads to 351, 440 redundant list generation tasks, which are reduced to 343 (1025× fewer) with our list generation removal and activation demotion. Note that the CUDA speed up (3.34×) is much higher than the x64 speed up (1.07×), 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. The task fusion pass is also able to fuse the Jacobi smoothing and reduction kernels, leading to improved memory performance.

8.4 AutoDiff: nodal forces from energy gradients

We implemented MLS-MPM [Hu et al. 2018] with Lagrangian forces [Jiang et al. 2015]. In the simulation, the structural is modeled using triangular meshes and a NeoHookean hyperelastic model. 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 [Hu et al. 2020]. 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 cases, the result of the total energy L is not used, so by looking at the whole program our optimizer can automatically eliminate the forward computation, only doing the backward gradient evaluation. Whole-program dead store elimination plays the most important role in this benchmark case.

An interesting observation is that our system gets significantly higher speed up 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 whole-program 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 smaller fraction on x64 (since P2G remains the bottleneck), hence a smaller speed up.

Table 1. Benchmarks against the original Taichi system [Hu et al. 2019]. In [Hu et al. 2019] the benchmarks are done against state-of-the-art manually engineered CPU and GPU implementations. 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 of execution time boost is 1.73×, the reduction of task launched is 3.07×.

Cases	Backend	Execution time (s)		Tasks launched		Instructions emitted		Tasks compiled	
		Reference	Ours	Reference	Ours	Reference	Ours	Reference	Ours
MacCormack	x64	8.973	8.899	9001	2401	8308	8210	96	30
	CUDA	0.477	0.313	9001	2401	8308	8210	96	30
MGPCG	x64	16.222	15.188	880820	177614	2808	3166	189	96
	CUDA	6.084	1.823	880820	177614	3234	3299	189	96
AutoDiff energy	x64	17.171	15.799	88204	56604	1353	2145	23	32
	CUDA	2.688	0.588	88204	56604	1353	2375	23	33

9 CONCLUSION

We have presented a whole program optimization framework with an asynchronous execution engine in Taichi, tailored for spatially sparse programming and differentiable programming. In our test cases, we get $1.73-2.76\times$ performance improvement without requiring users to change any code. Taichi's spatially sparse programming patterns open up new opportunities for whole-program optimizations. For example, we successfully removed redundant list generations of sparse data structures, and detect sparse array access patterns that must be already activated according to the context.

We believe our optimizer can greatly improve the productivity and performance of Taichi programs, since programmers can more flexibly code in Taichi without worrying about the redundant underlying tasks. We also hope our whole-program optimization framework can help optimize other programming systems in the near future.

REFERENCES

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. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* (2018).

Steven W Gagniere, David AB Hyde, Alan Marquez-Razon, Chenfanfu Jiang, Ziheng Ge, Xuchen Han, Qi Guo, and Joseph Teran. 2020. A Hybrid Lagrangian/Eulerian Collocated Advection and Projection Method for Fluid Simulation. *arXiv preprint arXiv:2003.12227* (2020).

Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana-Tampubolon, Eftychios Sifakis, Yuksel Cem, and Chenfanfu Jiang. 2018. GPU Optimization of Material Point Methods. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 32, 4 (2018), 102.

Rama Karl Hoetzlein. 2016. GVDB: Raytracing sparse voxel database structures on the GPU. In *Proceedings of High Performance Graphics*. Eurographics Association, 109–117.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *ICLR* (2020).

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. 2018. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 4 (2018), 150.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 201.

Chenfanfu Jiang, Craig Schroeder, Andrew Selle, Joseph Teran, and Alexey Stomakhin. 2015. The affine particle-in-cell method. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–10.

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

Kathleen Knobe and Vivek Sarkar. 1998. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 107–120.

Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2020. The Deep Learning Compiler: A Comprehensive Survey. *arXiv preprint arXiv:2002.03794* (2020).

Haixiang Liu, Yuanming Hu, Bo Zhu, Wojciech Matusik, and Eftychios Sifakis. 2018. Narrow-band Topology Optimization on a Sparsely Populated Grid. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 251:1–251:14.

Dror E Maydan, Saman P Amarasinghe, and Monica S Lam. 1993. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2–15.

Ken Museth, Jeff Lait, John Johanson, Jeff Budsberg, Ron Henderson, Mihai Alden, Peter Cucka, David Hill, and Andrew Pearce. 2013. OpenVDB: an open-source data structure and toolkit for high-resolution volumes. In *Acm siggraph 2013 courses*. 1–1.

Michael B Nielsen and Robert Bridson. 2016. Spatially adaptive FLIP fluid simulations in bifrost. In *ACM SIGGRAPH 2016 Talks*. ACM, 41.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. 2008. An unconditionally stable MacCormack method. *Journal of Scientific Computing* 35, 2-3 (2008), 350–371.

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 33, 6 (2014), 205.

Kui Wu, Nghia Truong, Cem Yuksel, and Rama Hoetzlein. 2018. Fast fluid simulations with sparse volumes on the GPU. In *Computer Graphics Forum (Proc. Eurographics)*, Vol. 37. Wiley Online Library, 157–167.

APPENDIX

Microbenchmark cases

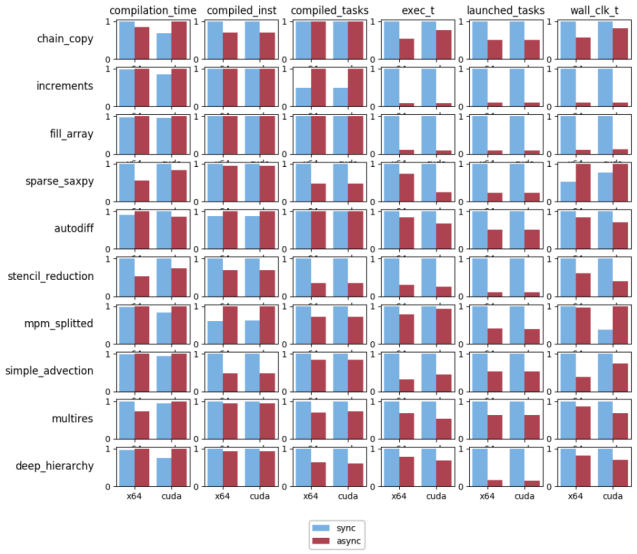


Fig. 11. Microbenchmarks in synchronous/asynchronous mode. On average 2.76× performance boost is achieved on the execution time (`exec_t`) running time metric.

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 8. They are fused in asynchronous mode.

increments contains 10 inc() kernels in Figure 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 [Hu et al. 2018].

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 12 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.

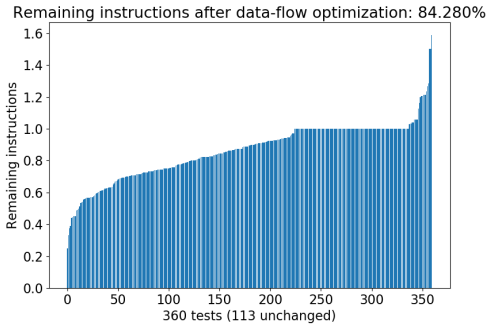


Fig. 12. Ratio of emitted instructions with/without data-flow optimization among unit tests. On most cases fewer instructions are emitted. The cases with more instructions do happen, because data-flow optimization indirectly triggers some other optimization passes that lead to high-performance but also more (and cheaper) instructions.