

Kernel Launcher: C++ library for optimal-performance portable CUDA applications

netherlands
eScience center

iWAPT @ IPDPS, 19 May 2023
Stijn Heldens, Ben van Werkhoven

High-Performance Computing = GPUs



Lumi

2021-now

309 PFLOPS

10,240x AMD MI250x

Summit

2019-now

200 PFLOPS

27,648x NVIDIA V100

Frontier

2022-now

1,102 PFLOPS

37,888x AMD MI250x

Snellius

2021-now

14 PFLOPS

144x NVIDIA A100

GPU Programming

Writing GPU kernels requires extensive tuning:

- Optimal threads per block?
- Optimal items per thread?
- Optimal registers per thread?
- Optimal unroll factor?
- Use shared memory or not?
- ...



GPU Programming

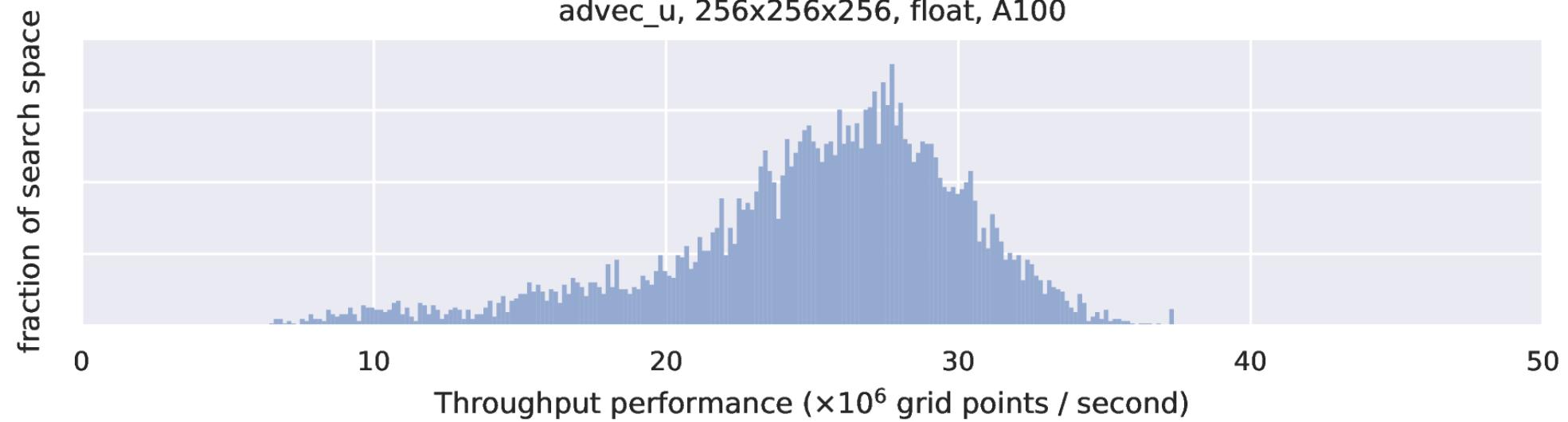
Writing GPU kernels requires extensive tuning:

- Optimal threads per block?
- Optimal items per thread?
- Optimal registers per thread?
- Optimal unroll factor?
- Use shared memory or not?
- ...

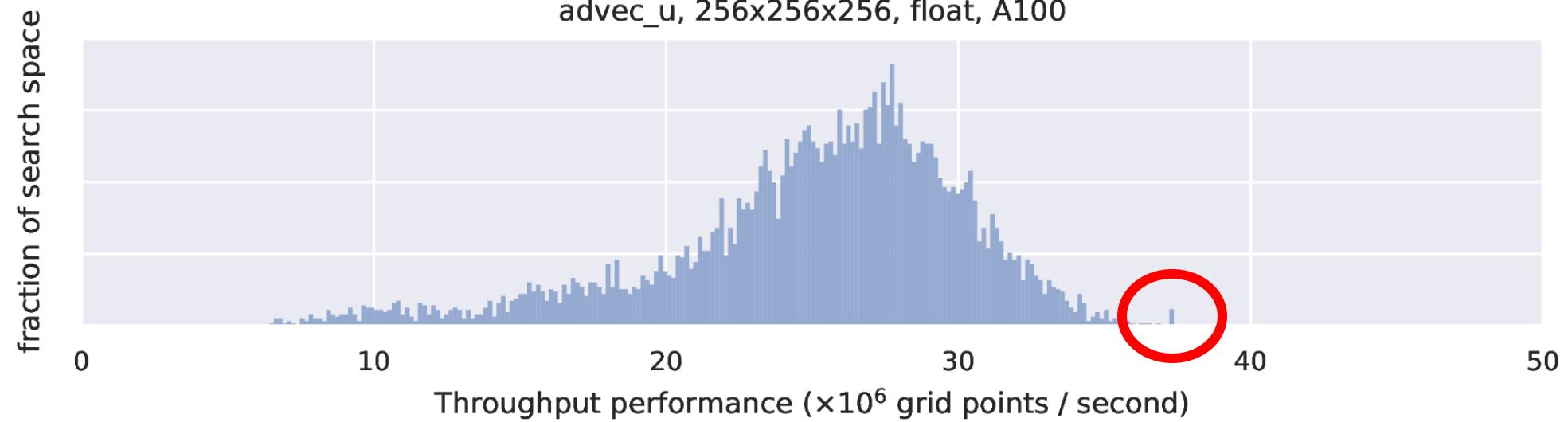


⇒ Large search space!

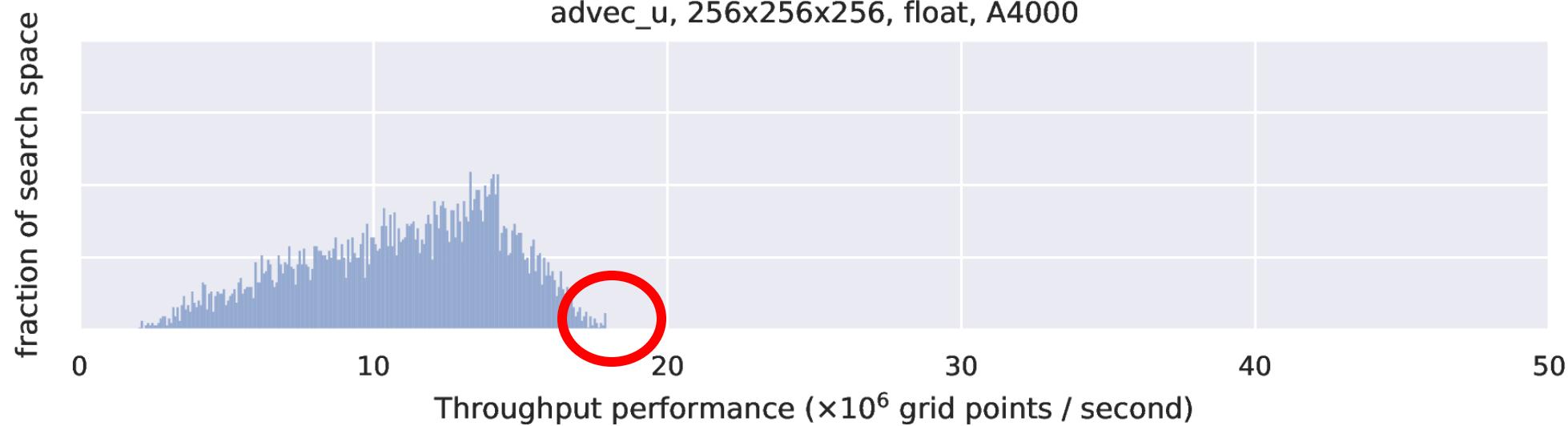
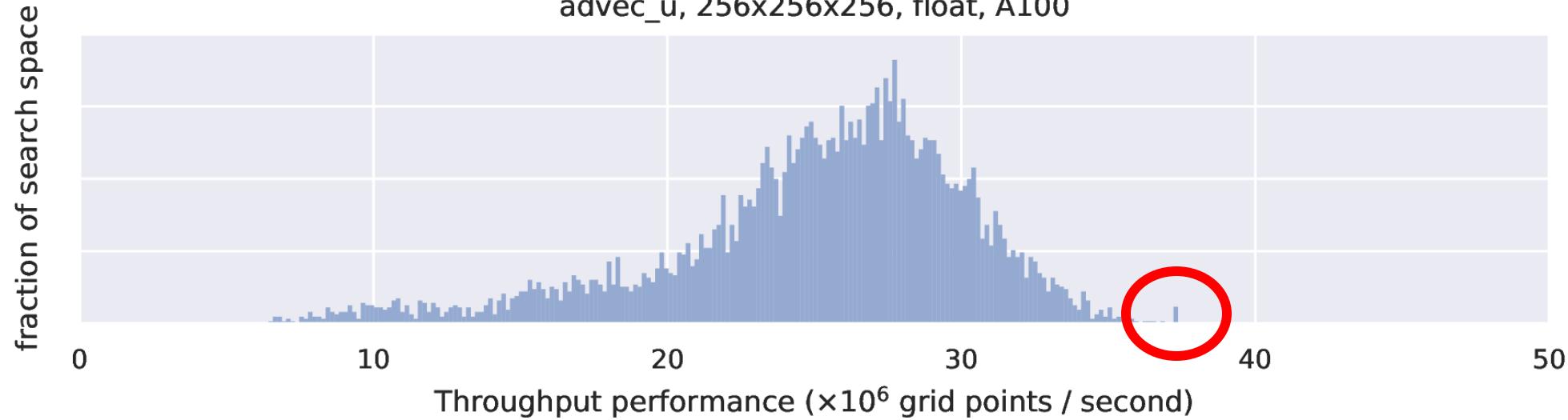
Performance example



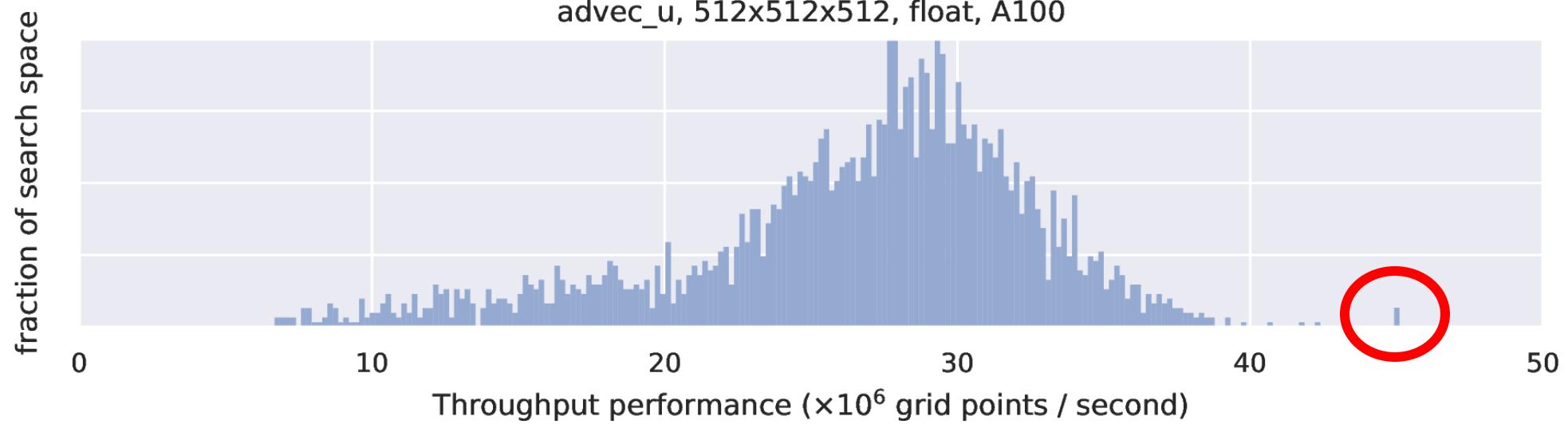
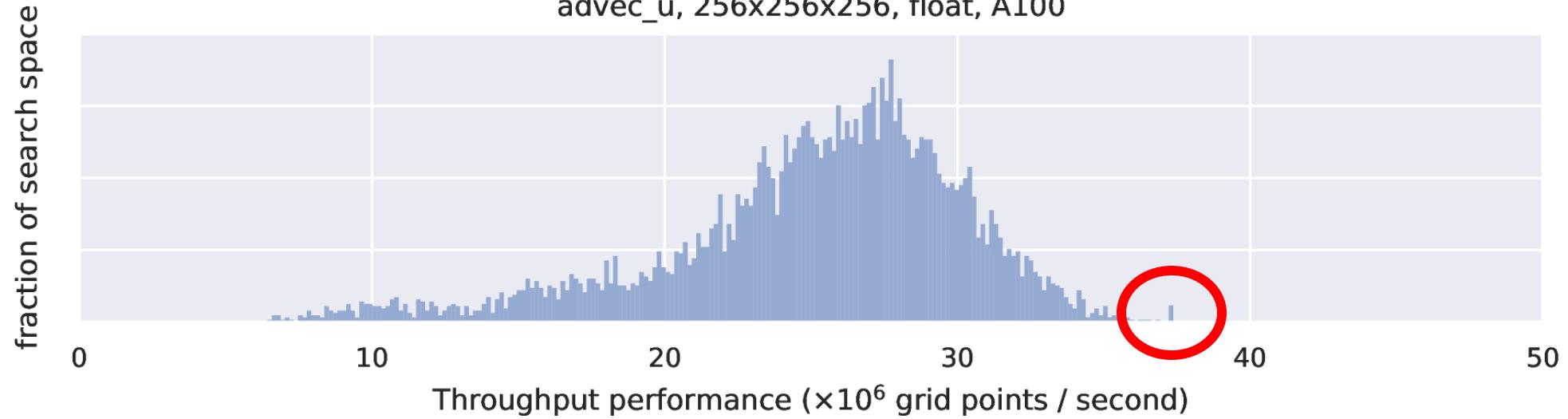
Performance example



Performance example



Performance example



Kernel Tuner

- Auto-tuning tool in Python
- Supports many GPU languages:
 - **CUDA**, OpenCL, OpenACC, HIP, C, ...
- Supports many search strategies
 - 15+ optimization algorithms
- In development for 7+ years
- Used by many projects at Netherlands eScience Center

KernelTuner / kernel_tuner Public

Code Issues 12 Pull requests 4 Discussions Actions Projects Wiki Security Insights Settings

master 21 branches 20 tags Go to file Add file <> Code

benvanherkoven Merge pull request #198 from KernelTuner/powerse... 4e0a80e 2 weeks ago 1,539 commits

.github/workflows Update docs-on-release.yml 7 months ago

doc Merge branch 'master' into refactor_interface 2 months ago

examples update example 3 weeks ago

kernel_tuner update to interface used by PowerSensor3 2 weeks ago

test Formatted with black. 2 months ago

.gitattributes fixing reported language on github 6 years ago

.gitignore updated packaging last year

.zenodo.json add .zenodo.json hoping to fix zenodo doi generation 3 years ago

CHANGELOG.md bump version 2 months ago

CITATION.cff Update CITATION.cff cffversion to 1.2.0 2 years ago

CONTRIBUTING.rst [fix] invalid links in contribution guide 5 months ago

INSTALL.rst fix AMD SDK link 7 months ago

LICENSE license file that github understands 6 years ago

MANIFEST.in finally using the same readme for github and pypi 7 years ago

README.rst update optimization strategies intro in README 3 months ago

roadmap.md Update roadmap.md 7 months ago

setup.cfg finally using the same readme for github and pypi 7 years ago

setup.py Formatted with black. 2 months ago

About Kernel Tuner kerneltuner.github.io/kernel_tuner/ python c testing machine-learning cplusplus gpu optimization opencl cuda autotuning software-development opencl-kernels kernel-tuner cuda-kernels gpu-computing auto-tuning

Readme Apache-2.0 license Cite this repository 162 stars 10 watching 38 forks Report repository

Releases 20 Version 0.4.4 (Latest) on Mar 9 + 19 releases

Packages No packages published Publish your first package

Contributors 21 + 10 contributors

Environments 2 github-pages Active dev_environment Active

Languages Python 100.0%

Kernel Tuner simplifies the software development of optimized and auto-tuned GPU programs, by enabling Python-based unit testing of GPU code and making it easy to develop scripts for auto-tuning GPU kernels. This also means no extensive changes and no new dependencies are required in the kernel code. The kernels can still be compiled and used as normal from any host programming language.

Kernel Tuner provides a comprehensive solution for auto-tuning GPU programs, supporting auto-tuning of user-defined parameters in both host and device code, supporting output verification of all benchmarked kernels during tuning, as well as many optimization strategies to speed up the tuning process.

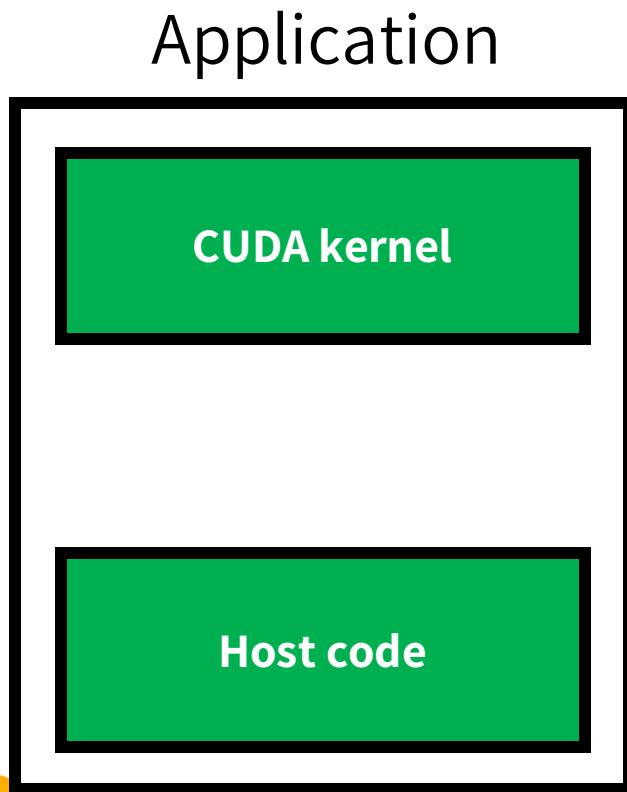
Documentation

The full documentation is available [here](#).

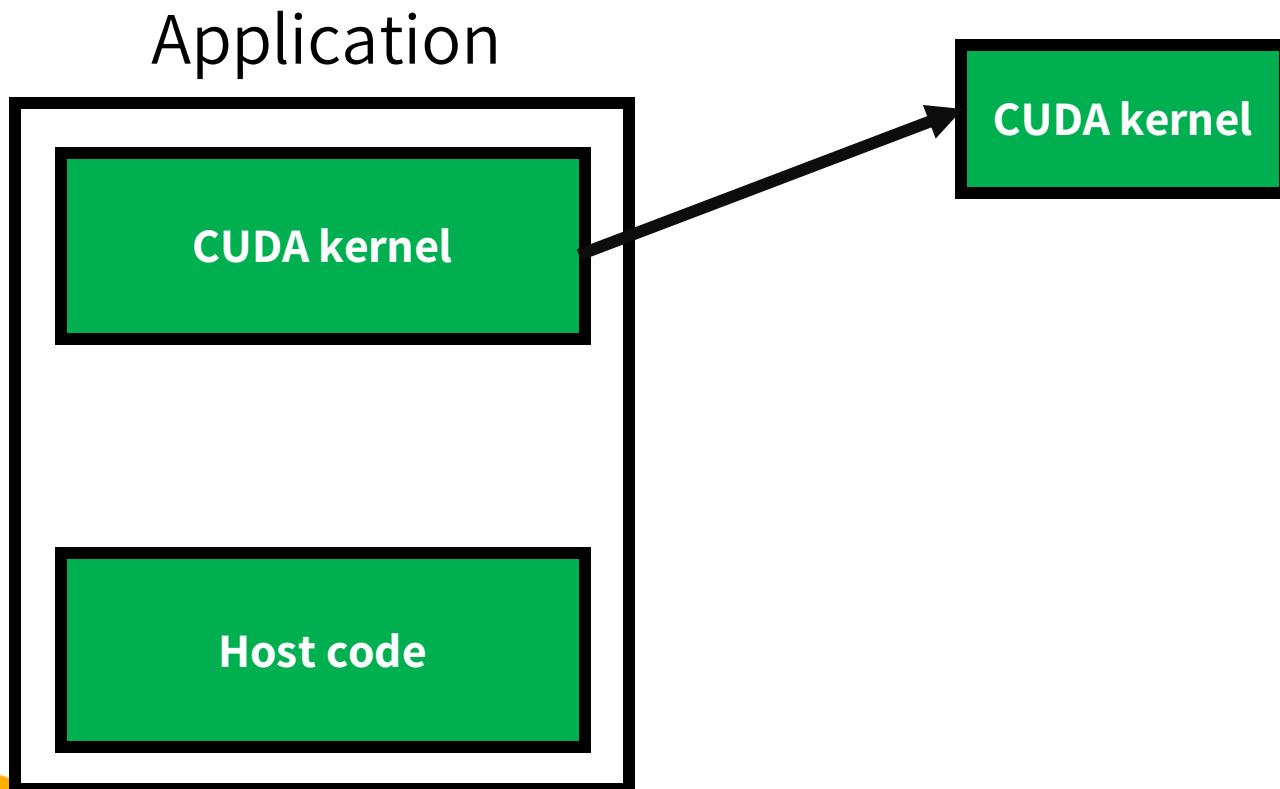
Installation

<https://kerneltuner.github.io/>

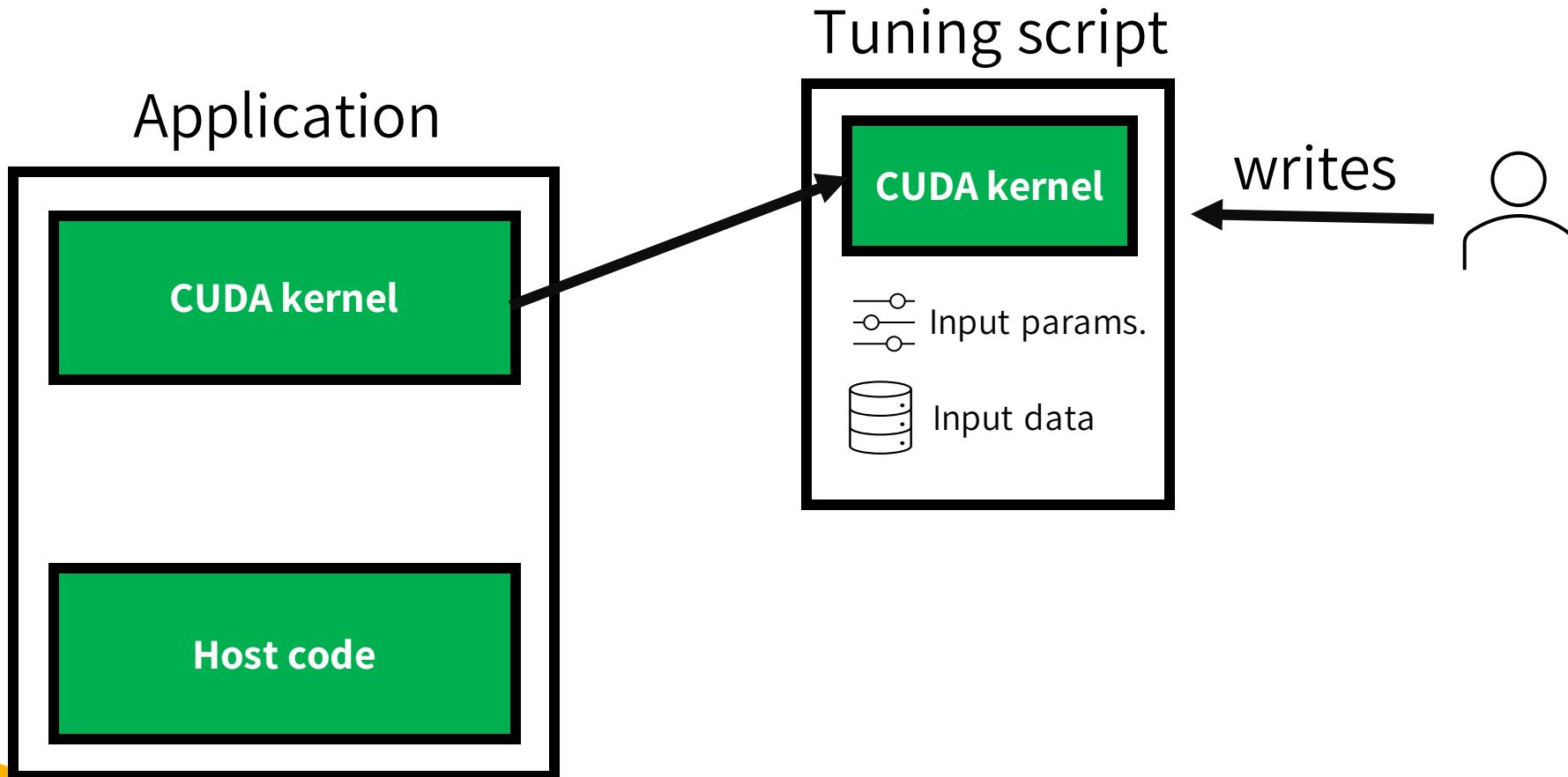
Previous workflow



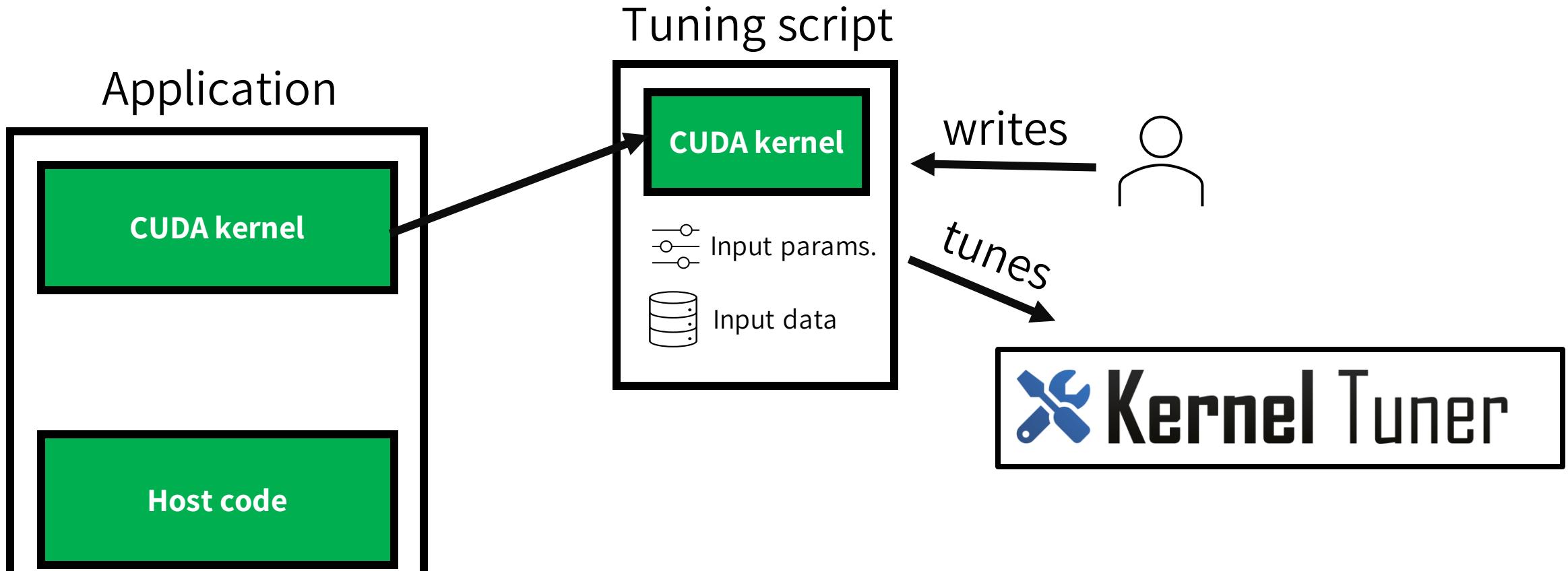
Previous workflow



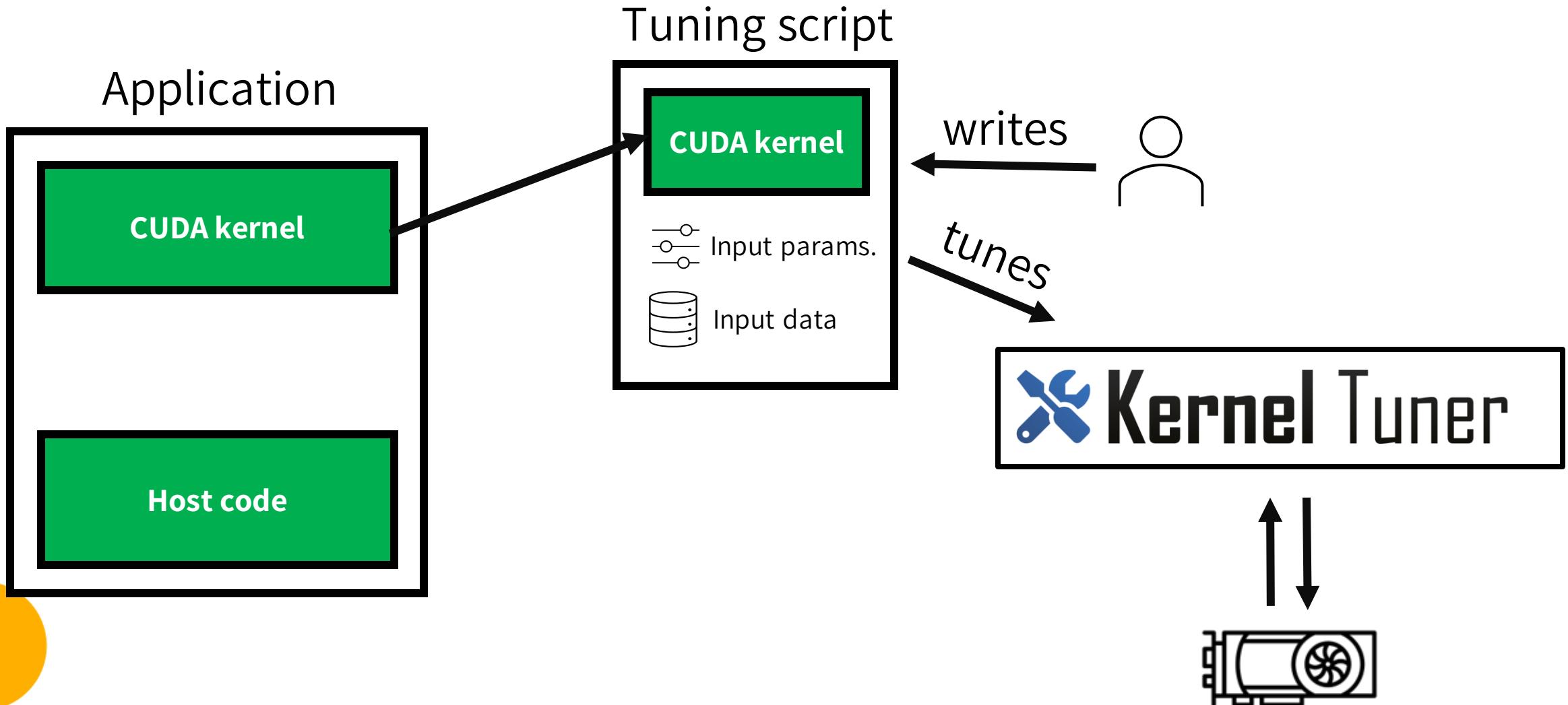
Previous workflow



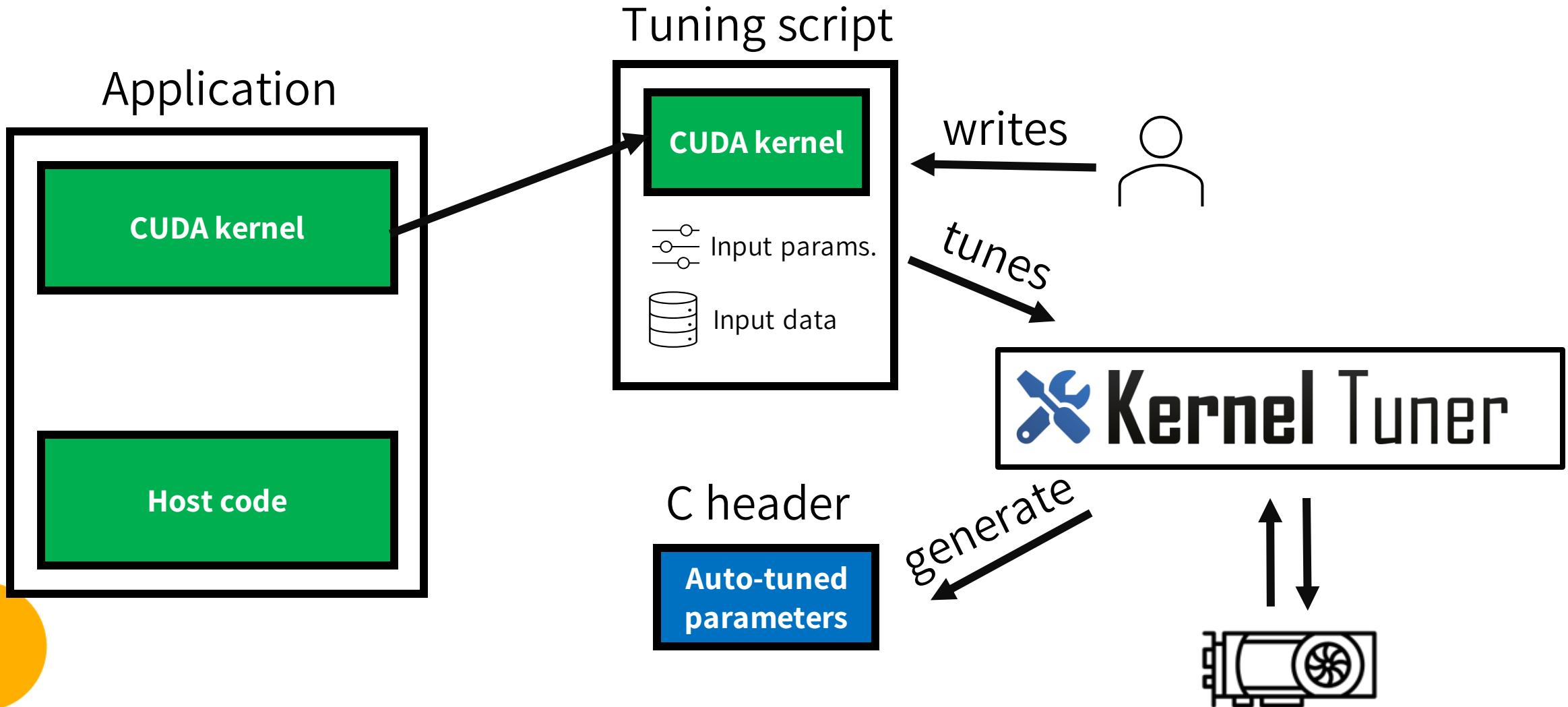
Previous workflow



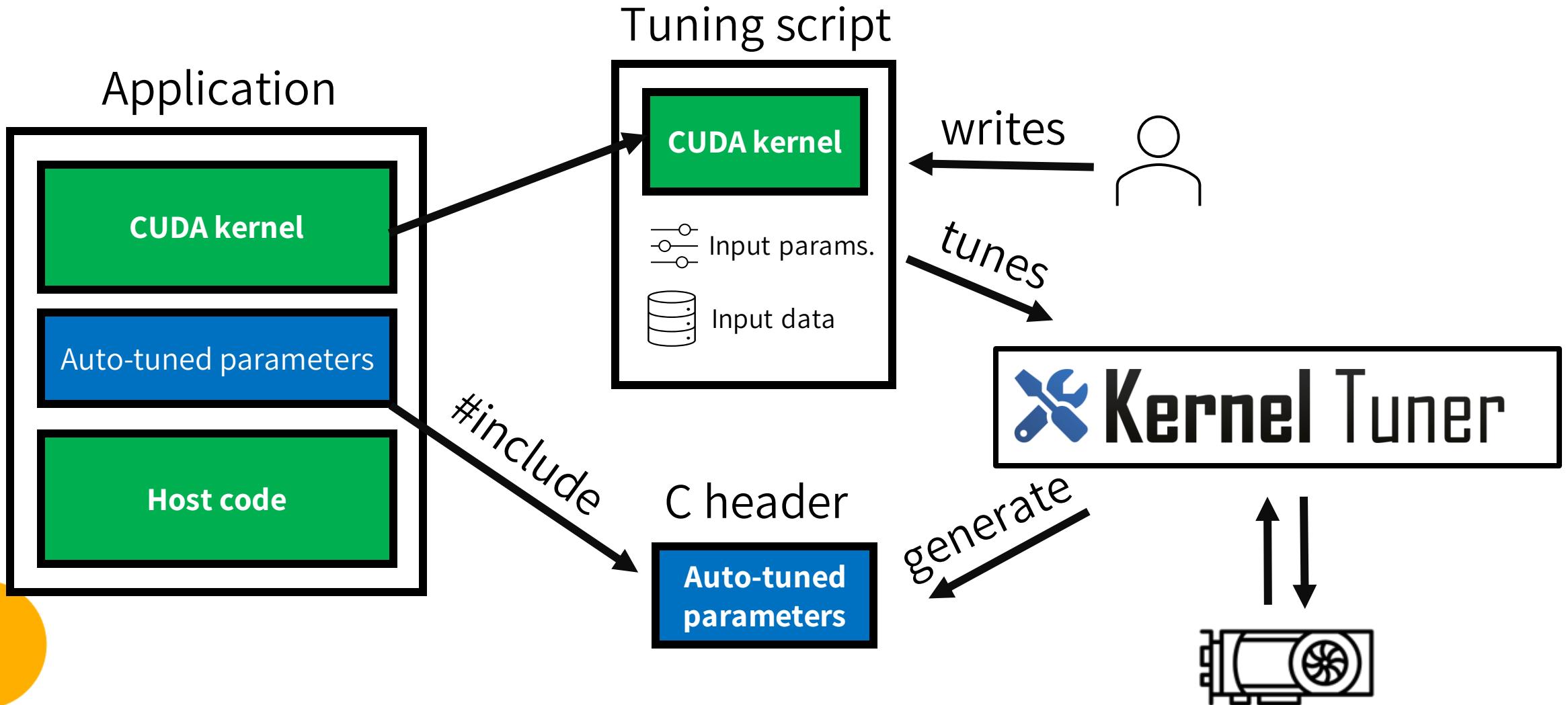
Previous workflow



Previous workflow



Previous workflow



Problem: Scaling the workflow

Challenges when scaling to large applications (100+ GPU kernels)

#1 Kernel code duplicated

- Can get out of sync

#2 Data generation

- Random data may not be representative

#3 Integrate tuning results

- Currently, fixed GPU during compilation

#4 Kernel selection

- Currently, fixed problem dimensions

Solution: Kernel Launcher

- Lightweight C++ library
- Helps integrate auto-tuning into CUDA applications
 - Export kernels to be tuned by Kernel Tuner
 - Import tuning results, compile optimal kernel



» Kernel Launcher View page source

Kernel Launcher



Kernel Launcher

Kernel Launcher is a C++ library that makes it easy to dynamically compile CUDA kernels at run time (using NVRT) and call them in easy type-safe way using C++ magic. There are two reasons for using run-time compilation:

- Kernels that have tunable parameters (block size, elements per thread, loop unroll factors, etc.) where the optimal configuration can only be determined at runtime since it depends dynamic factors such as the type of GPU and the problem size.
- Improve performance by injecting runtime values as compile-time constant values into kernel code (dimensions, array strides, weights, etc.).

Basic Example

This section shows a basic code example. See [Tutorial](#) for a more advanced example.

Consider the following CUDA kernel for vector addition. This kernel has a template parameter `T` and a tunable parameter `ELEMENTS_PER_THREAD`.

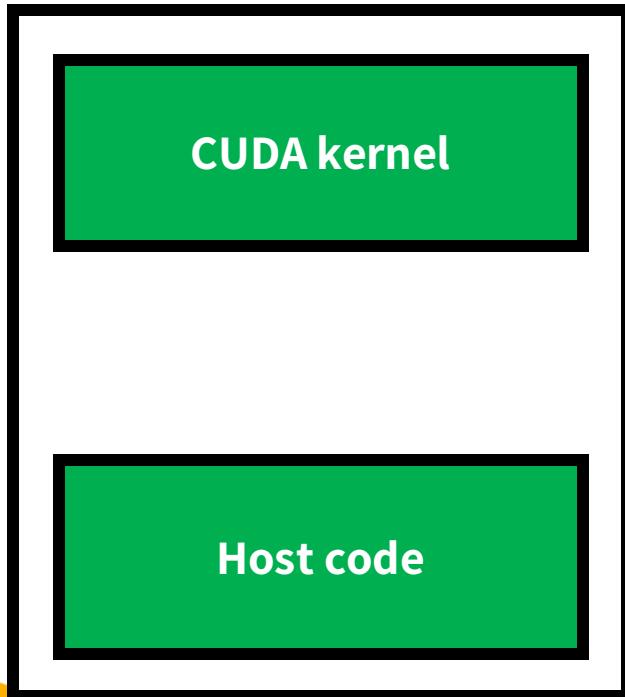
```
1 template <typename T>
2 __global__
3 void vector_add(int n, T* C, const T* A, const T* B) {
4     for (int k = 0; k < ELEMENTS_PER_THREAD; k++) {
5         int i = blockIdx.x * ELEMENTS_PER_THREAD * blockDim.x + k * blockDim.x + threadIdx;
6
7         if (i < n) {
8             C[i] = A[i] + B[i];
9         }
10    }
11 }
```

The following C++ snippet shows how to use *Kernel Launcher* in host code:

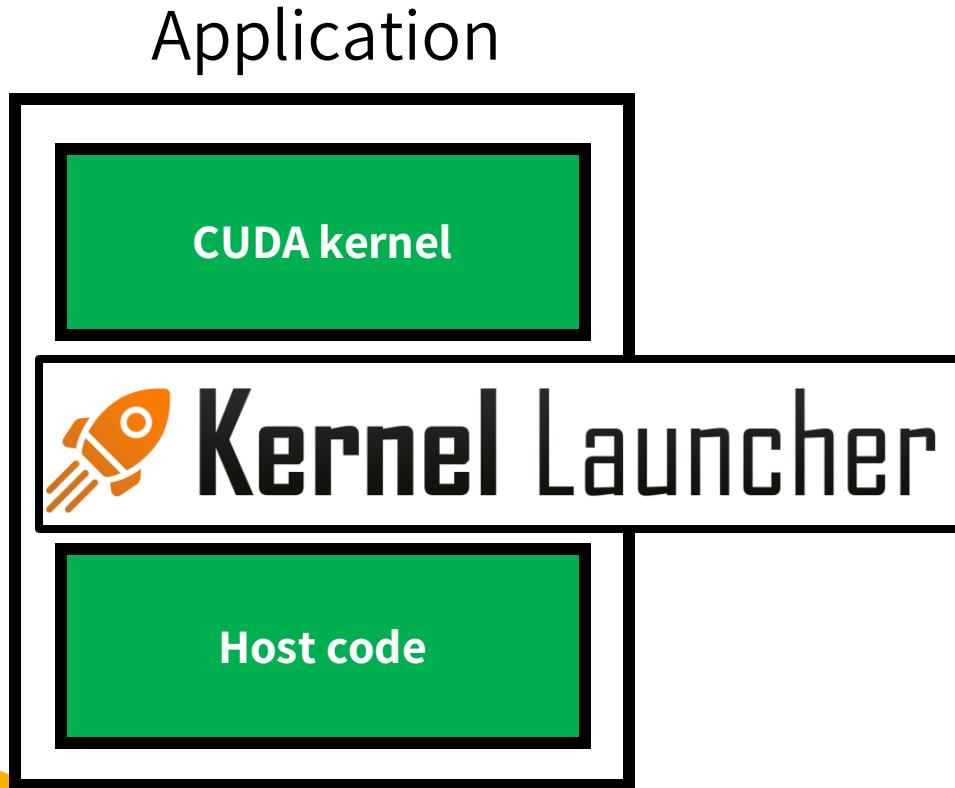
https://kerneltuner.github.io/kernel_launcher

New Kernel Launcher workflow

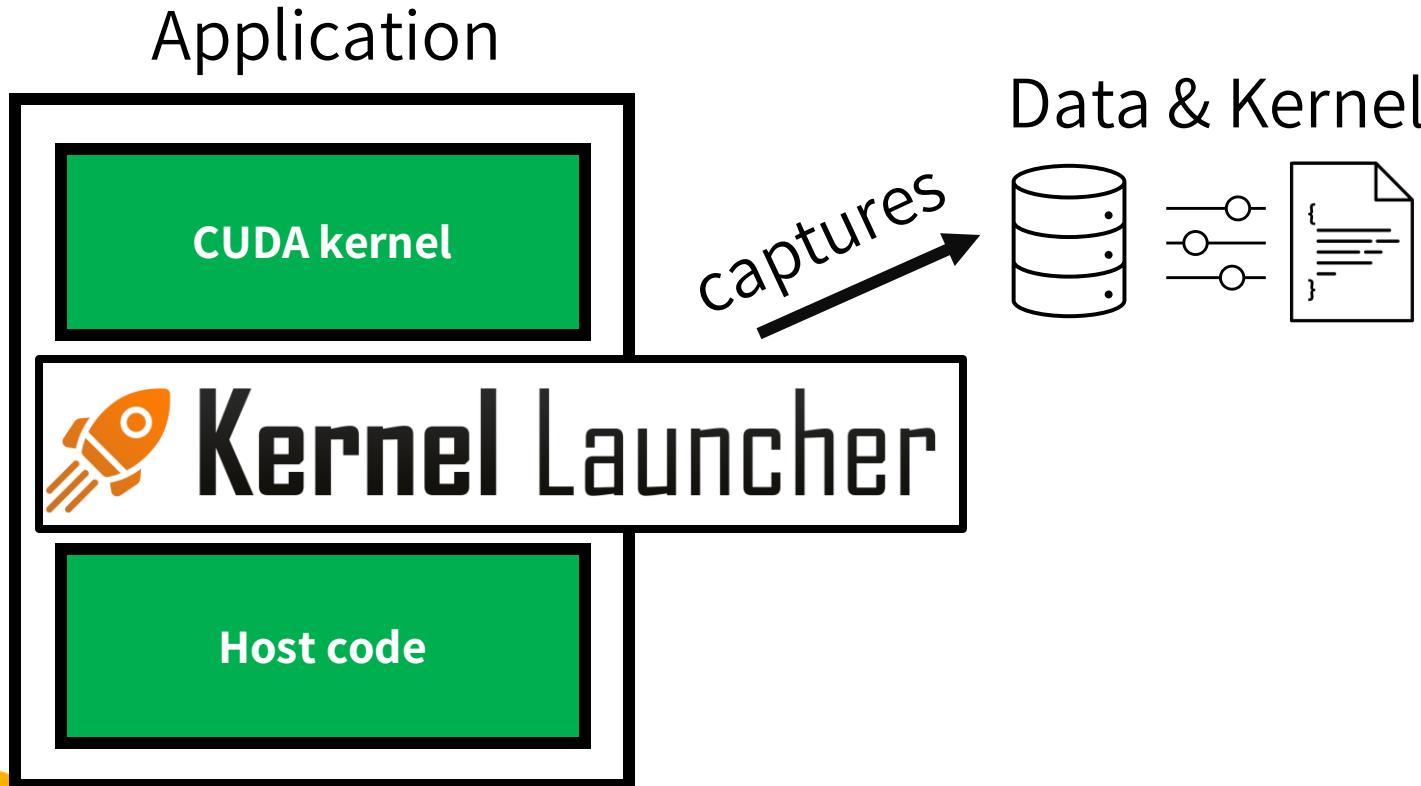
Application



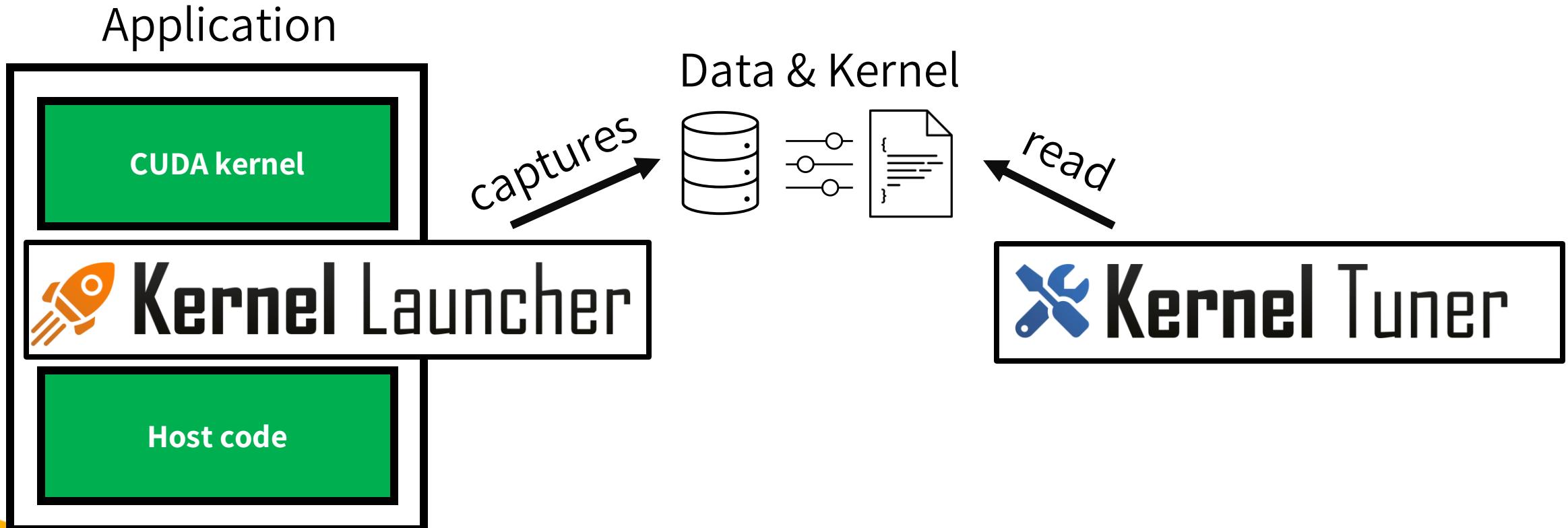
New Kernel Launcher workflow



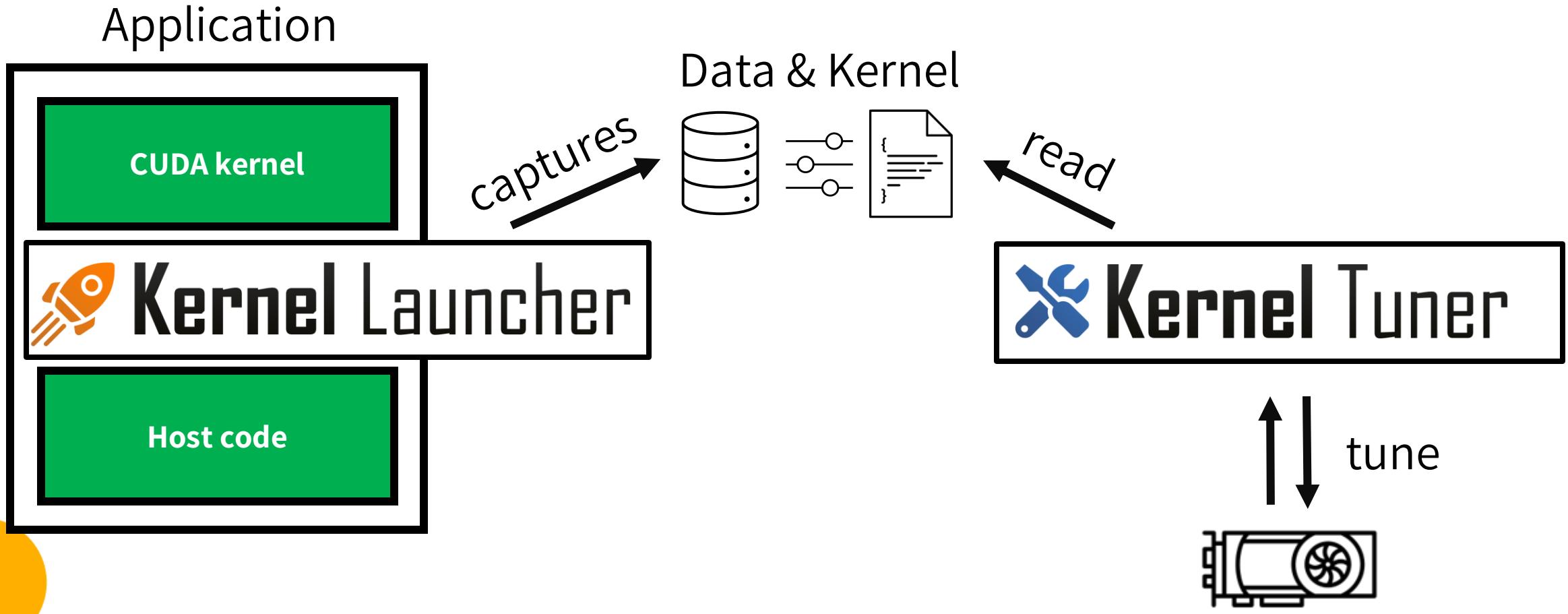
New Kernel Launcher workflow



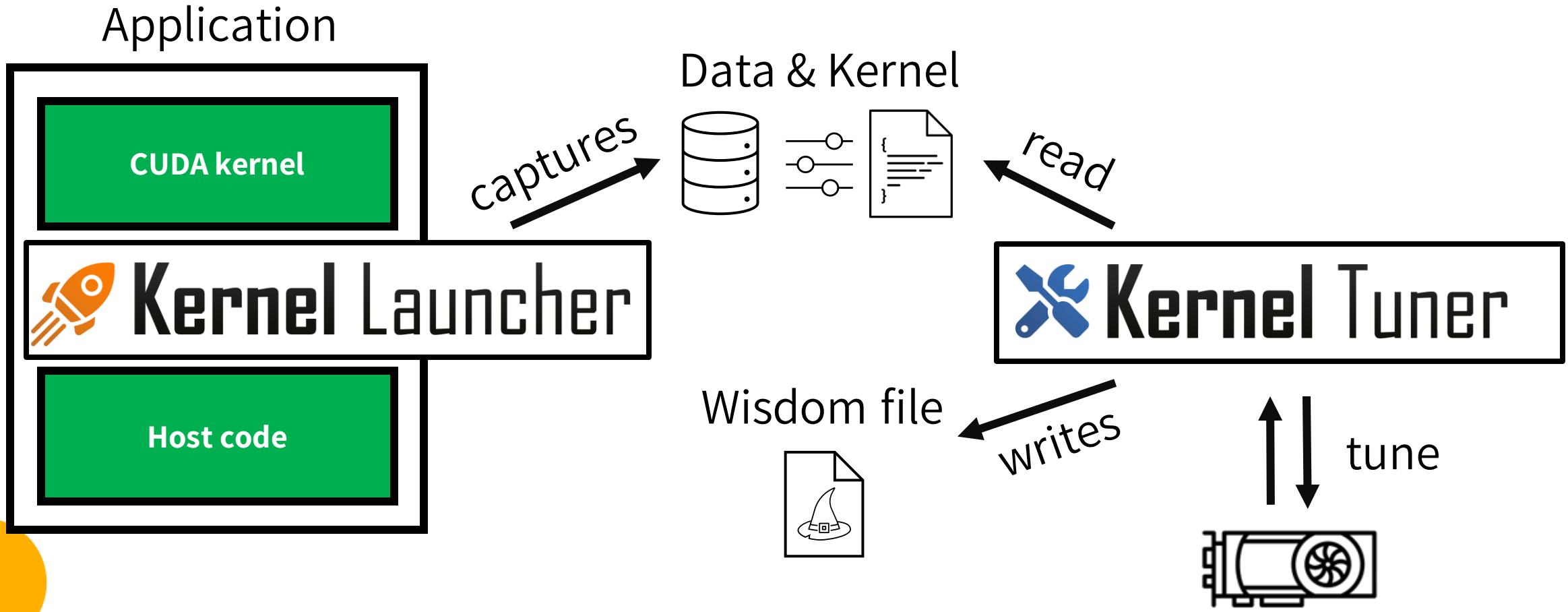
New Kernel Launcher workflow



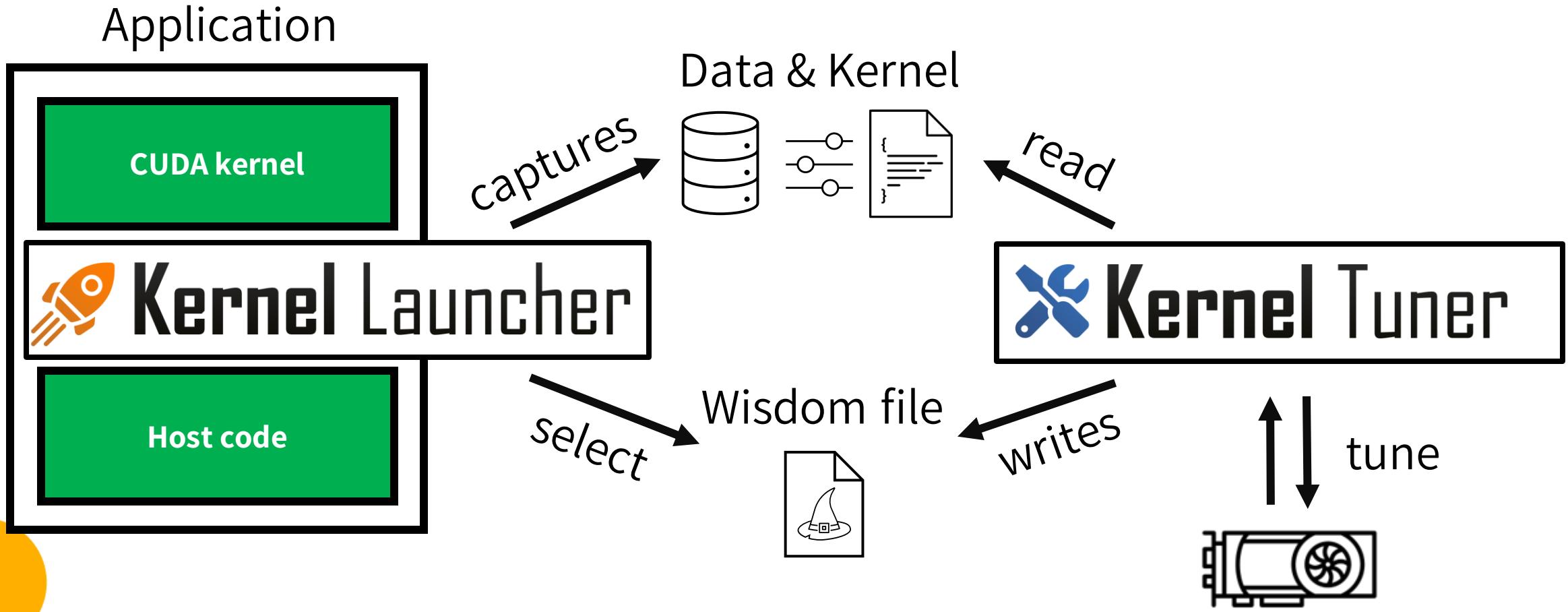
New Kernel Launcher workflow



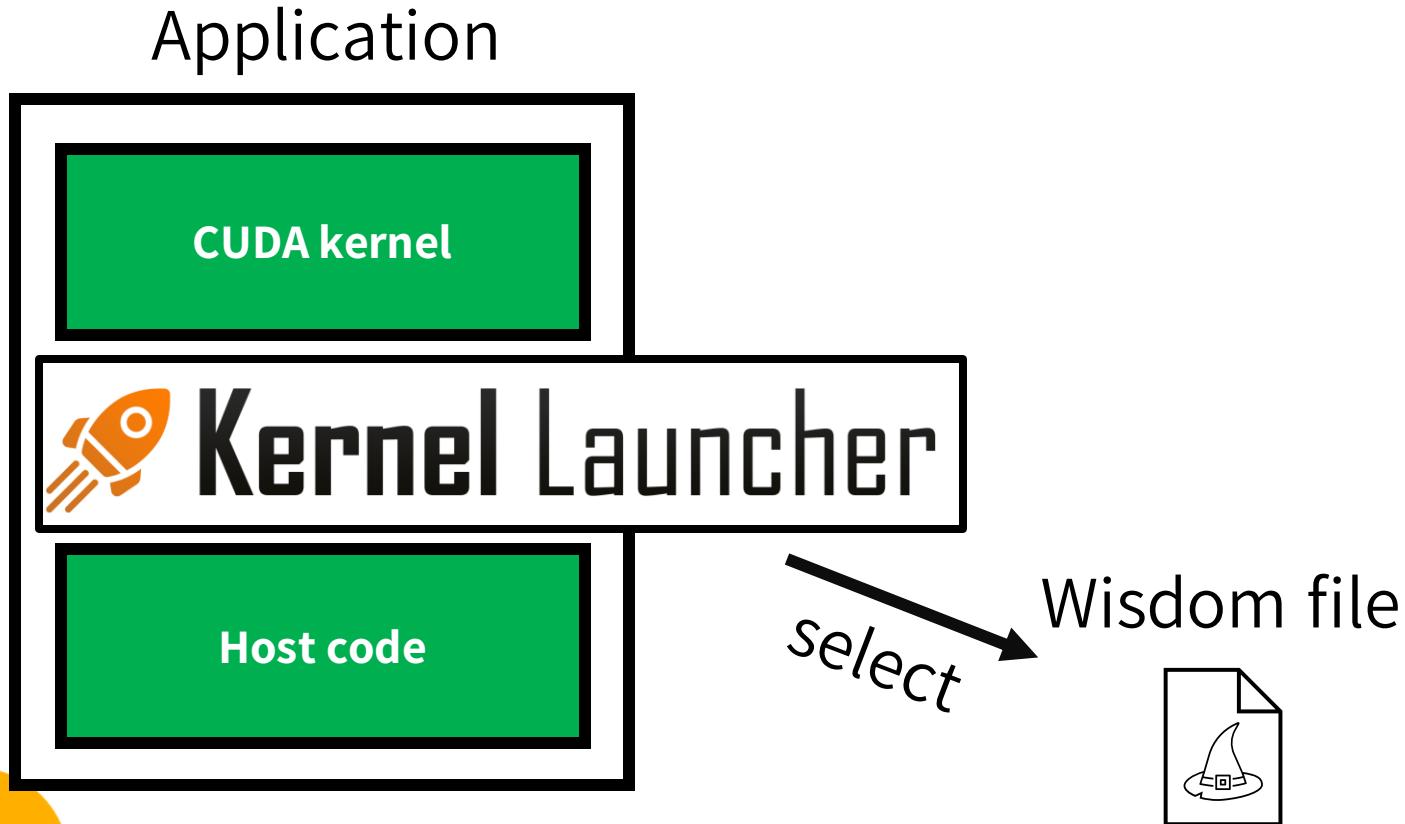
New Kernel Launcher workflow



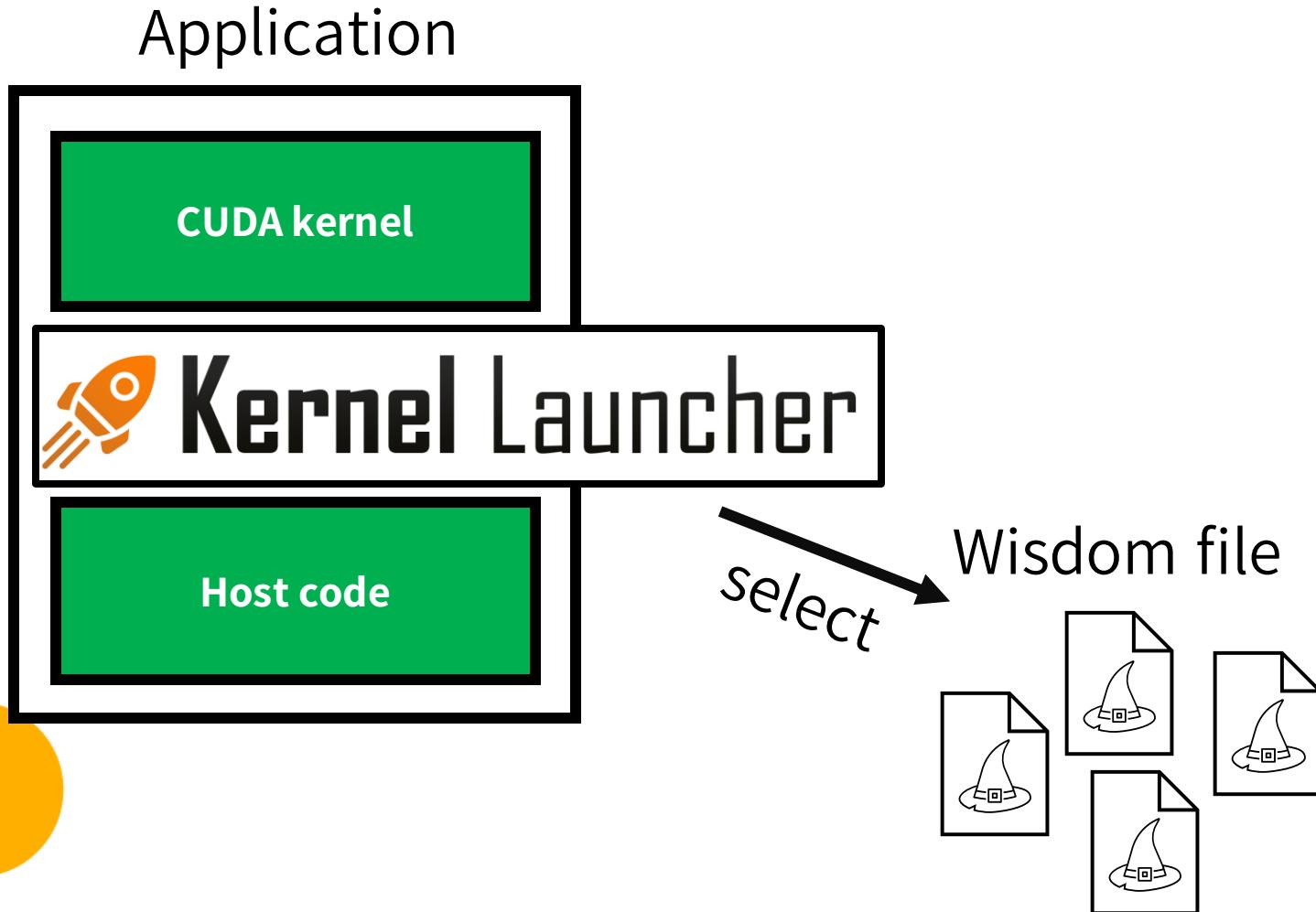
New Kernel Launcher workflow



New Kernel Launcher workflow



New Kernel Launcher workflow



Kernel Launcher solutions

#1 Kernel code duplicated

- Describe kernel in application host code
- Export kernel description to file

#2 Data generation

- "Capture" real kernel launch
- Export data to files on first kernel launch



Kernel Launcher solutions

#3 Integrate tuning results

- Write results to "wisdom" file
- Application reads wisdom at runtime

#4 Kernel selection

- Runtime kernel compilation (*NVRTC*)
- Can be device- and problem-dependent



Wisdom files

- One wisdom file per kernel
- Text file containing multiple results for...
 - Different problem sizes
 - Different GPUs
- Kernel launcher tries to find best match
 - Simple heuristic algorithm
- Runtime kernel compilation



Code example

Kernel-Launcher-based CUDA code

host.cpp

```
1
2
3 int main() {
4     // Initialize cuda
5     // cudaMalloc, cudaMemcpy, ...
6
7     // Set problem size
8     int n = 1_048_576;
9
10    // Set grid/block size
11    int block_size = 512;
12    int grid_size = n / block_size;
13
14    // Launch kernel
15    vector_add<<grid_size, block_size>>>(n, A, B, C);
16
17
18 }
```

kernel.cu

```
1
2
3
4
5 __global__ void vector_add(
6     int n, float* A, float* B, float* C
7 ) {
8     int i = threadIdx.x + blockIdx.x * blockDim.x;
9     C[i] = A[i] + B[i];
10 }
```



Code example

Kernel-Launcher-based CUDA code

host.cpp

```
1 #include "kernel_launcher.hpp"
2
3 int main() {
4     // Initialize cuda
5     // cudaMalloc, cudaMemcpy, ...
6
7     // Set problem size
8     int n = 1_048_576;
9
10    // Set grid/block size
11    //int block_size = 512;
12    //int grid_size = n / block_size;
13
14    // Launch kernel
15    kernel_launcher::launch(
16        kernel_launcher::PragmaKernel("vector_add", "kernel.cu"),
17        n, A, B, C);
18 }
```

kernel.cu

```
1 #pragma kernel tune(threads_per_block=32, 64, 128, 256)
2 #pragma kernel problem_size(n)
3 #pragma kernel block_size(threads_per_block)
4 #pragma kernel buffers(A[n], B[n], C[n])
5 __global__ void vector_add(
6     int n, float* A, float* B, float* C
7 ) {
8     int i = threadIdx.x + blockIdx.x * blockDim.x;
9     C[i] = A[i] + B[i];
10 }
```



Code example

Kernel-Launcher-based CUDA code

host.cpp

```
1 #include "kernel_launcher.hpp"
2
3 int main() {
4     // Initialize cuda
5     // cudaMalloc, cudaMemcpy, ...
6
7     // Set problem size
8     int n = 1_048_576;
9
10    // Set grid/block size
11    //int block_size = 512;
12    //int grid_size = n / block_size;
13
14    // Launch kernel
15    kernel_launcher::launch(
16        kernel_launcher::PragmaKernel("vector_add", "kernel.cu"),
17        n, A, B, C);
18 }
```

kernel.cu

```
1 #pragma kernel tune(threads_per_block=32, 64, 128, 256)
2 #pragma kernel problem_size(n)
3 #pragma kernel block_size(threads_per_block)
4 #pragma kernel buffers(A[n], B[n], C[n])
5 __global__ void vector_add(
6     int n, float* A, float* B, float* C
7 ) {
8     int i = threadIdx.x + blockIdx.x * blockDim.x;
9     C[i] = A[i] + B[i];
10 }
```

Use case: MicroHH

- Computational fluid dynamics for simulation of flows in atmosphere
- Supports MPI and GPUs
 - 210 CUDA kernels



For the simulation of turbulent flows in the atmosphere

About

MicroHH is a computational fluid dynamics code made for Direct Numerical Simulation (DNS) and Large-Eddy Simulation (LES) of turbulent flows in the atmosphere. The code is written in C++.

MicroHH is hosted on GitHub. There, the latest version of the source code can be found, as well as all releases, a bug tracker, and wiki pages to help you get started. In case of any questions, please contact Chiel van Heerwaarden (chielvanheerwaarden@gmail.com).

MicroHH is described in detail in van Heerwaarden et al. (2017). In case you decide to use MicroHH for your own research, the developers would appreciate to be notified and kindly request to cite their reference paper.

[View the code at GitHub »](#) [Bug tracker »](#) [Wiki pages »](#)

Animations

From time to time we post animations in our [Vimeo channel](#). The HD versions can be watched at the Vimeo website.

[View all animations on Vimeo »](#)



MicroHH activity map



Publications using MicroHH

Reference

→ G.C. van Heerwaarden, R.J.H. van Stratum, T. Huya, J. Gibbs, E. Fedorovich, J.R. Mallett (2017): MicroHH 1.0: a computational fluid dynamics code



Tuning MicroHH

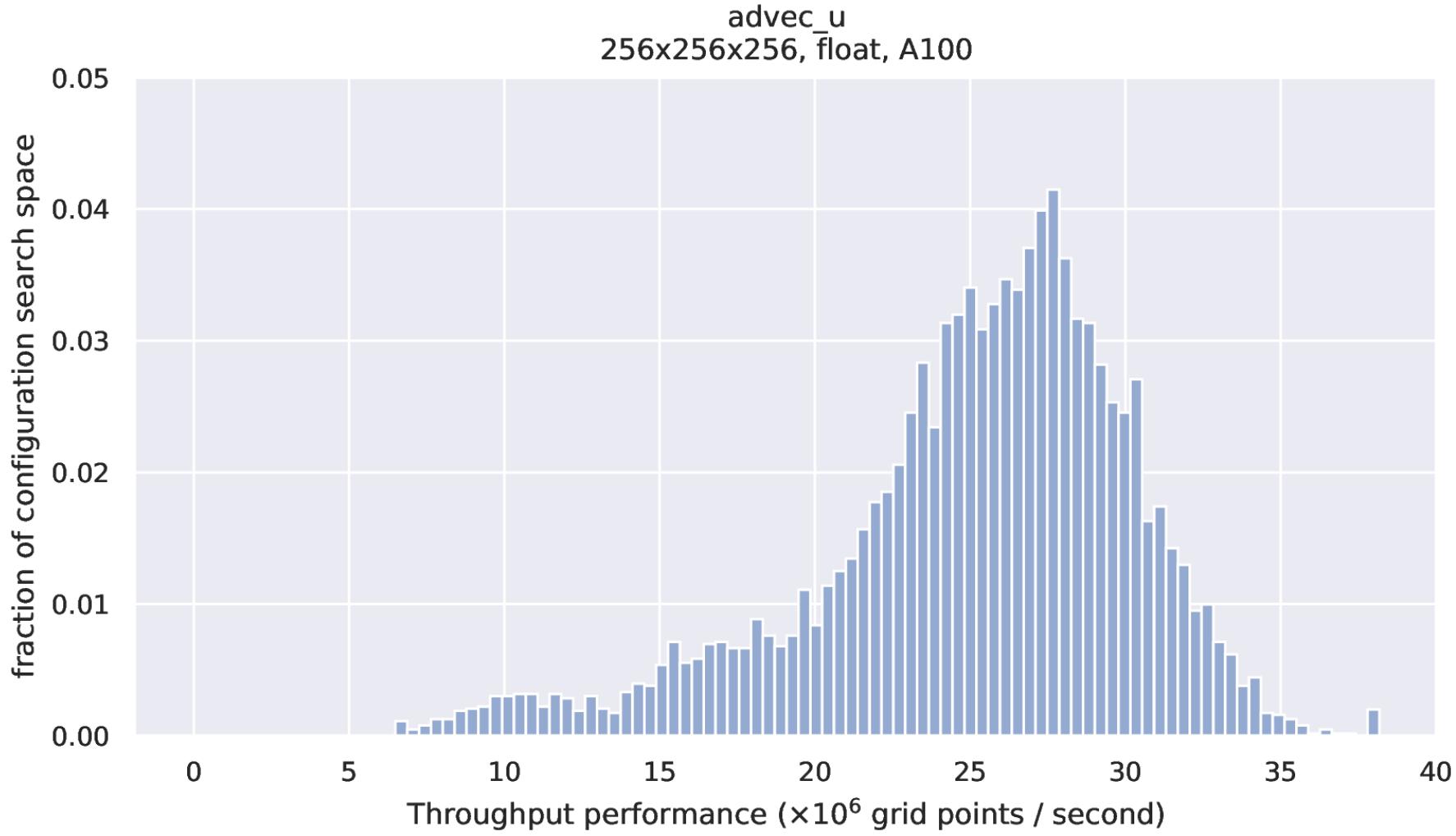
- Per kernel, introduced 14 tunable parameters
 - 7.7M configurations per kernel!
- Highly dynamic code
 - Variable grid size ($X \times Y \times Z$)
 - Variable precision (float, double)
 - GPU (Ampere, Volta...)
- Introduced **Kernel Launcher**

Table 2: List of tunable parameters and their default value.

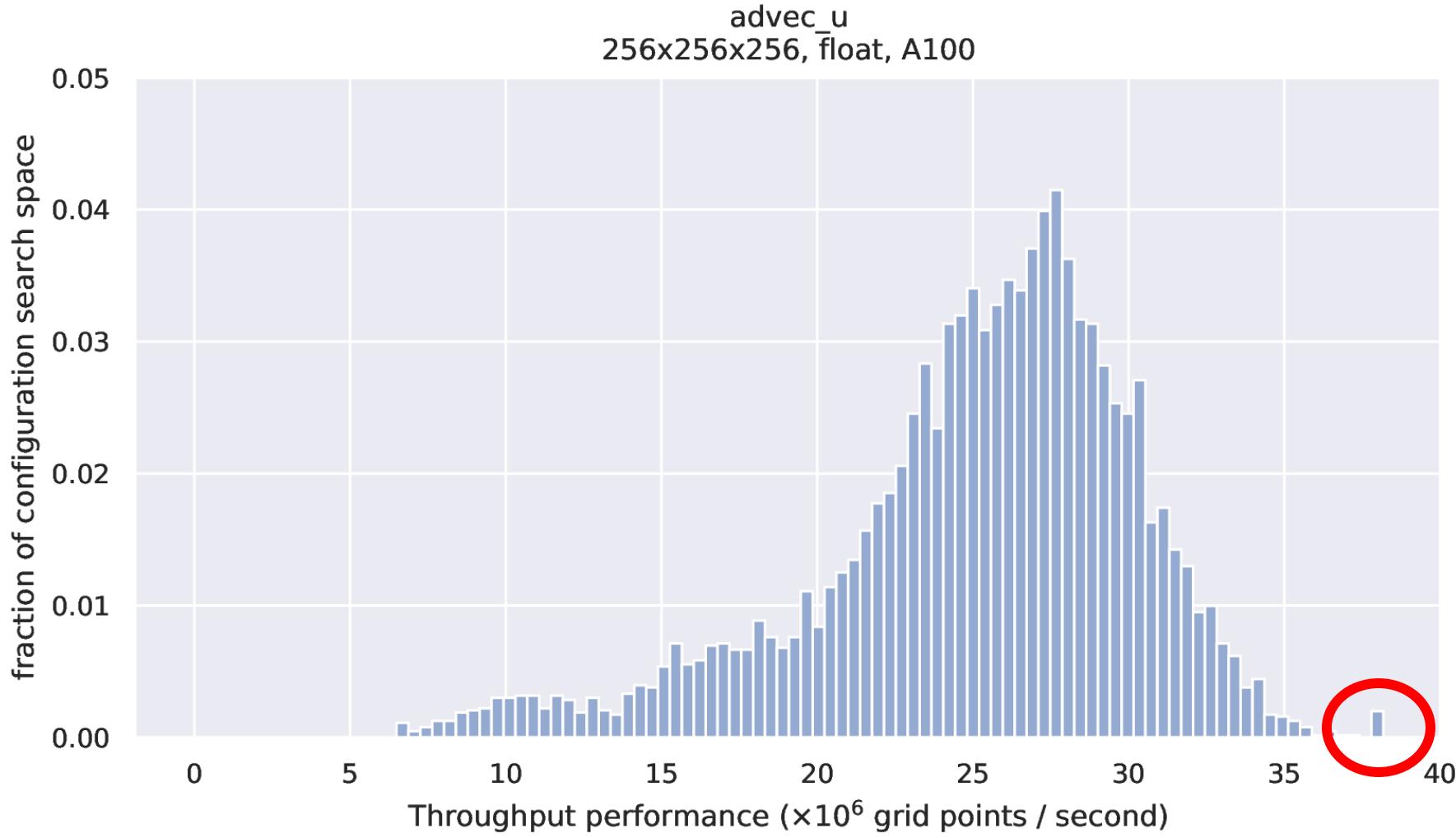
Name	Values	Default value
Block size X	16, 32, 64, 128, 256	256
Block size Y	1, 2, 4, 8, 16	1
Block size Z	1, 2, 4, 8, 16	1
Tile factor X	1, 2, 4	1
Tile factor Y	1, 2, 4	1
Tile factor Z	1, 2, 4	1
Unroll X	true, false	false
Unroll Y	true, false	false
Unroll Z	true, false	false
Tile contiguous X	true, false	false
Tile contiguous Y	true, false	false
Tile contiguous Z	true, false	false
Unravel permutation	XYZ, XZY, YXZ, YZX, ZXY, ZYX	XYZ
Min. blocks per SM	1, 2, 3, 4, 5, 6	1



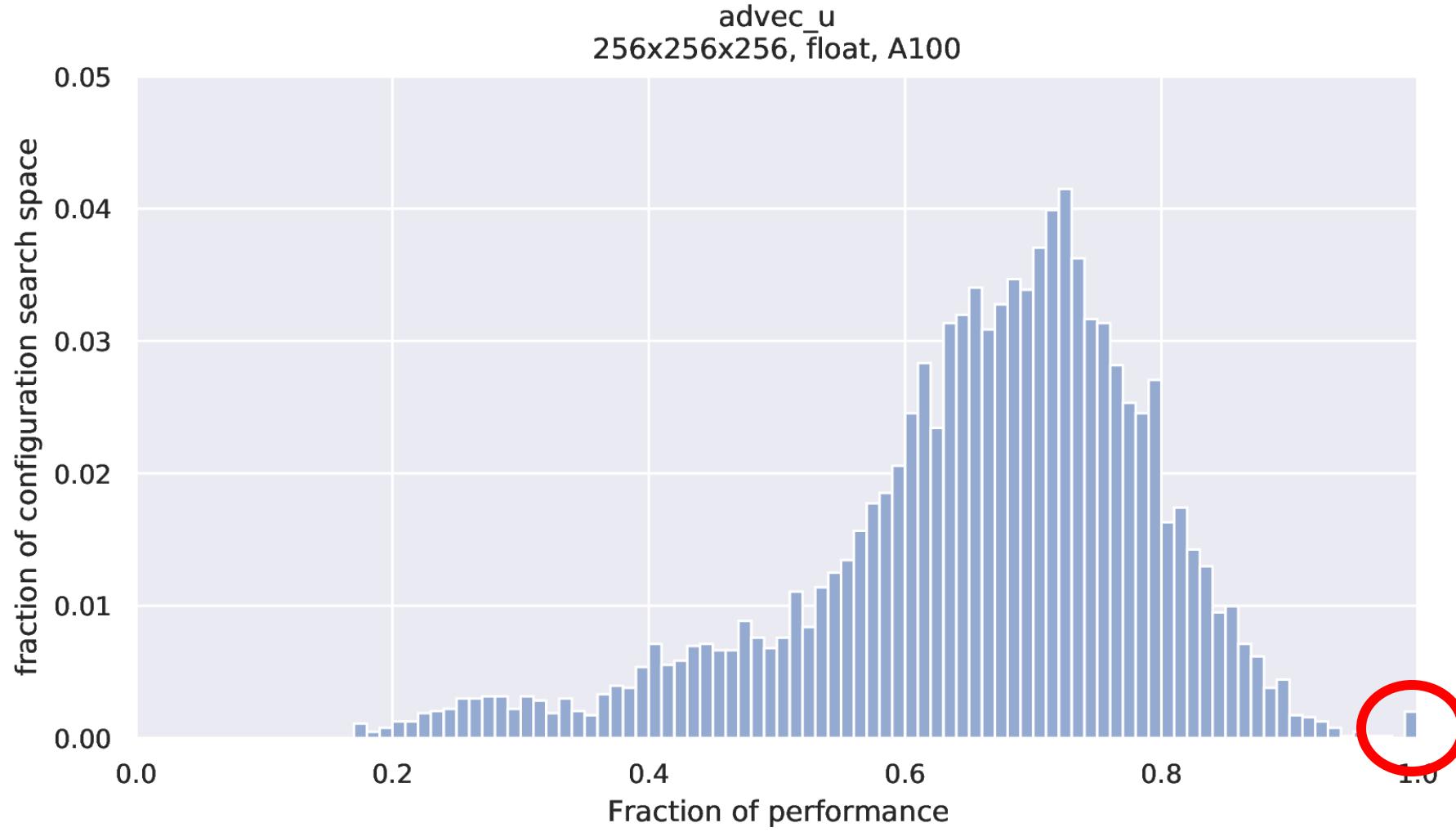
MicroHH



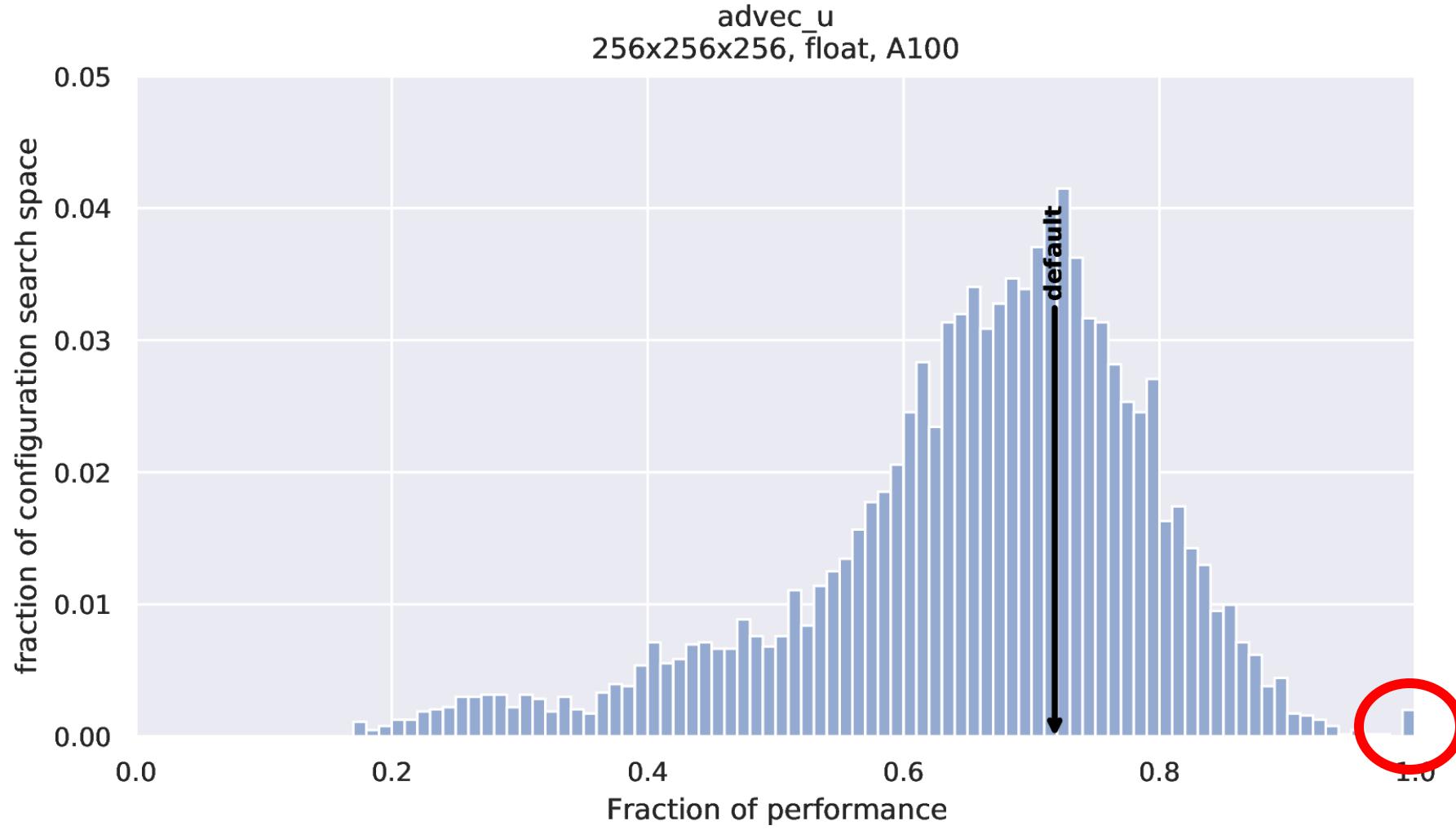
MicroHH

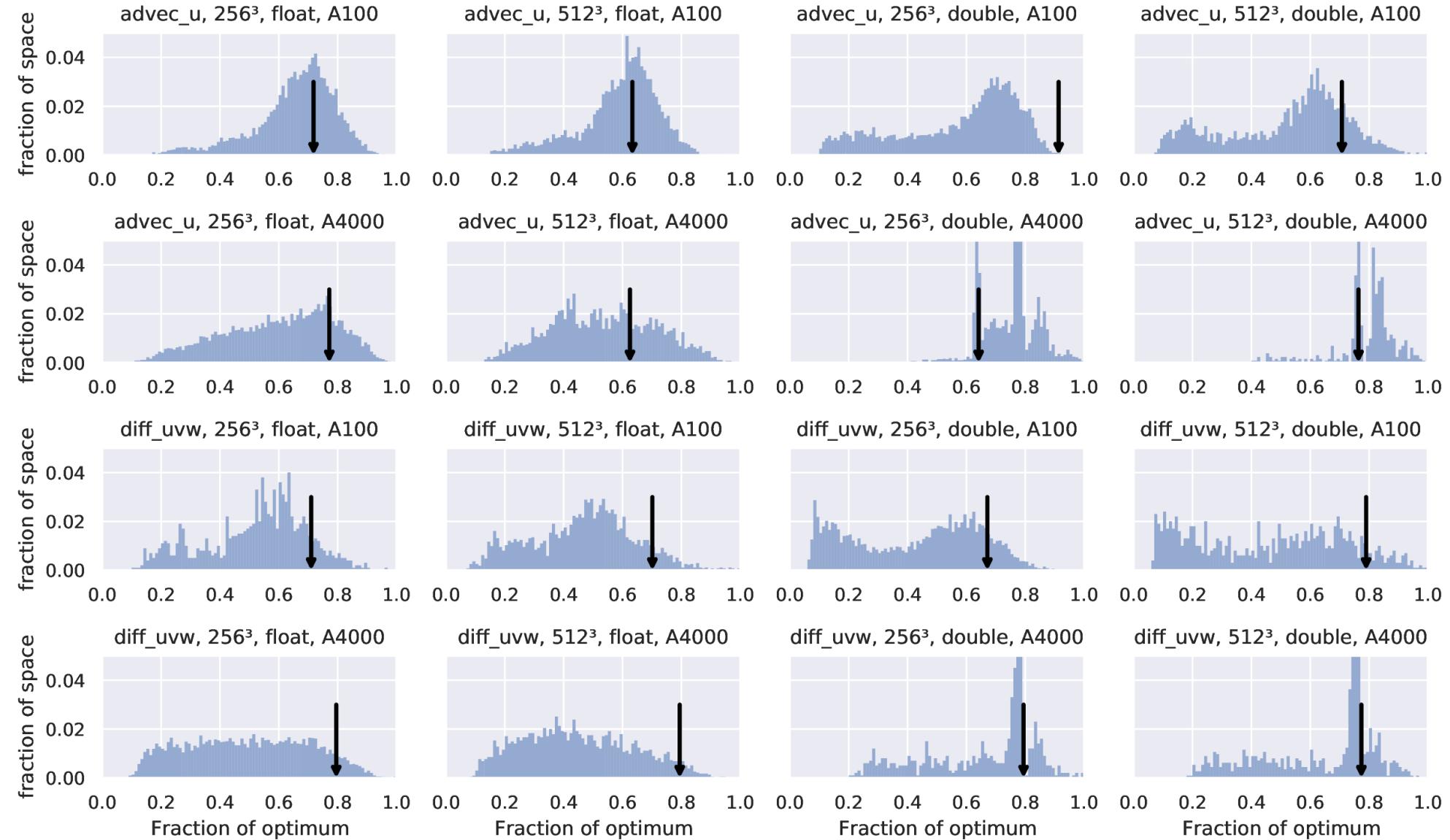


MicroHH

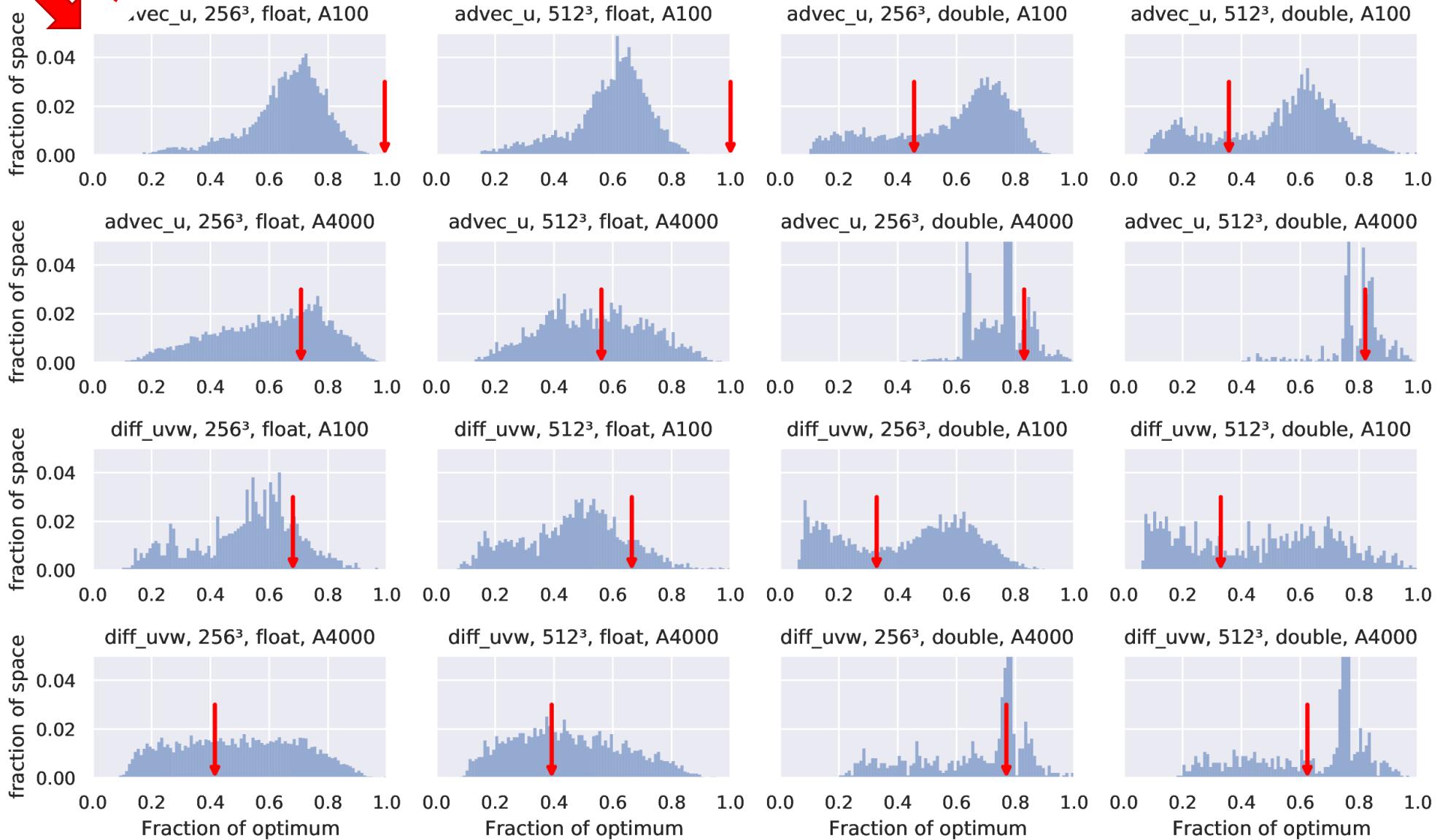


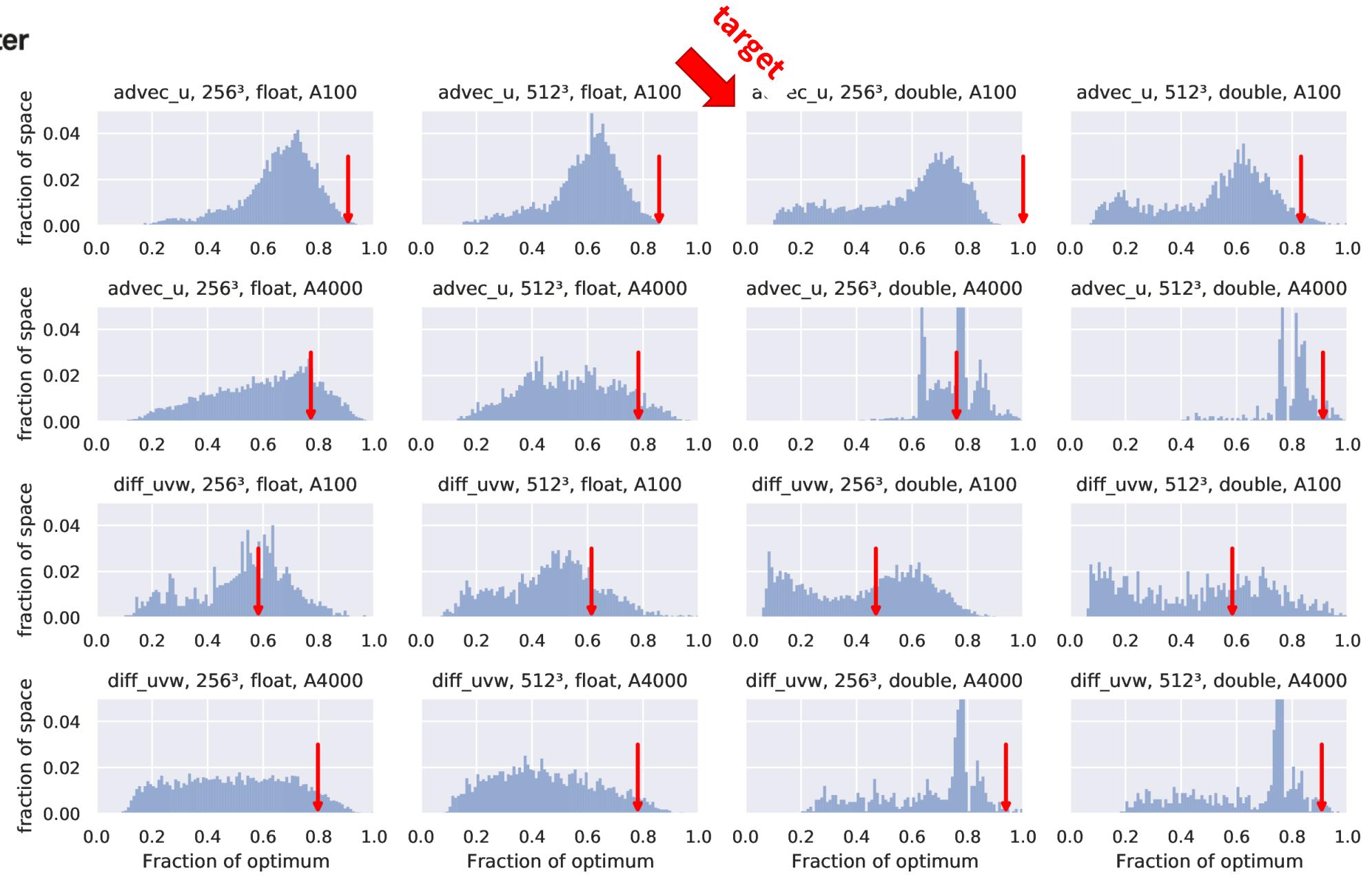
MicroHH

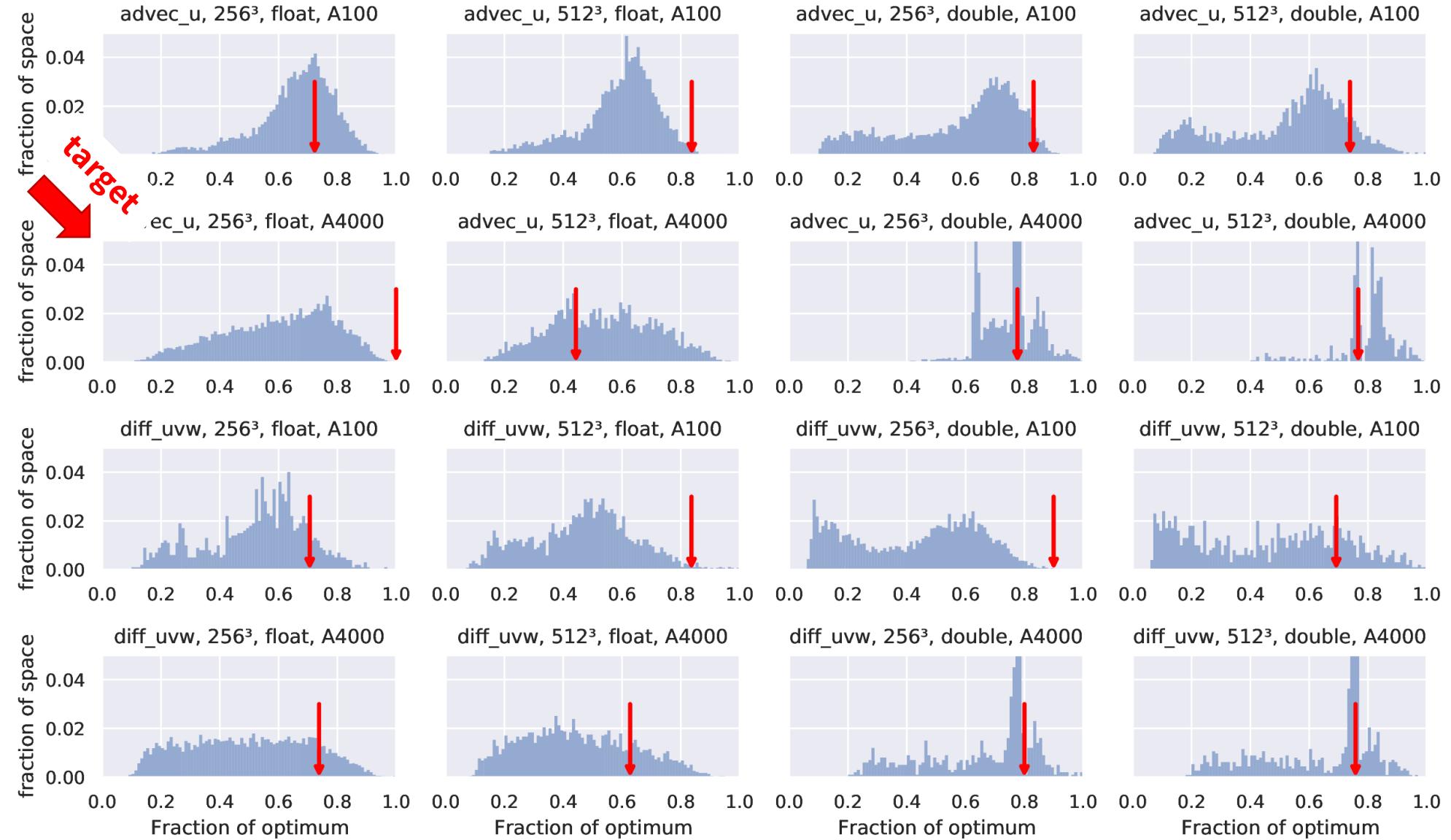


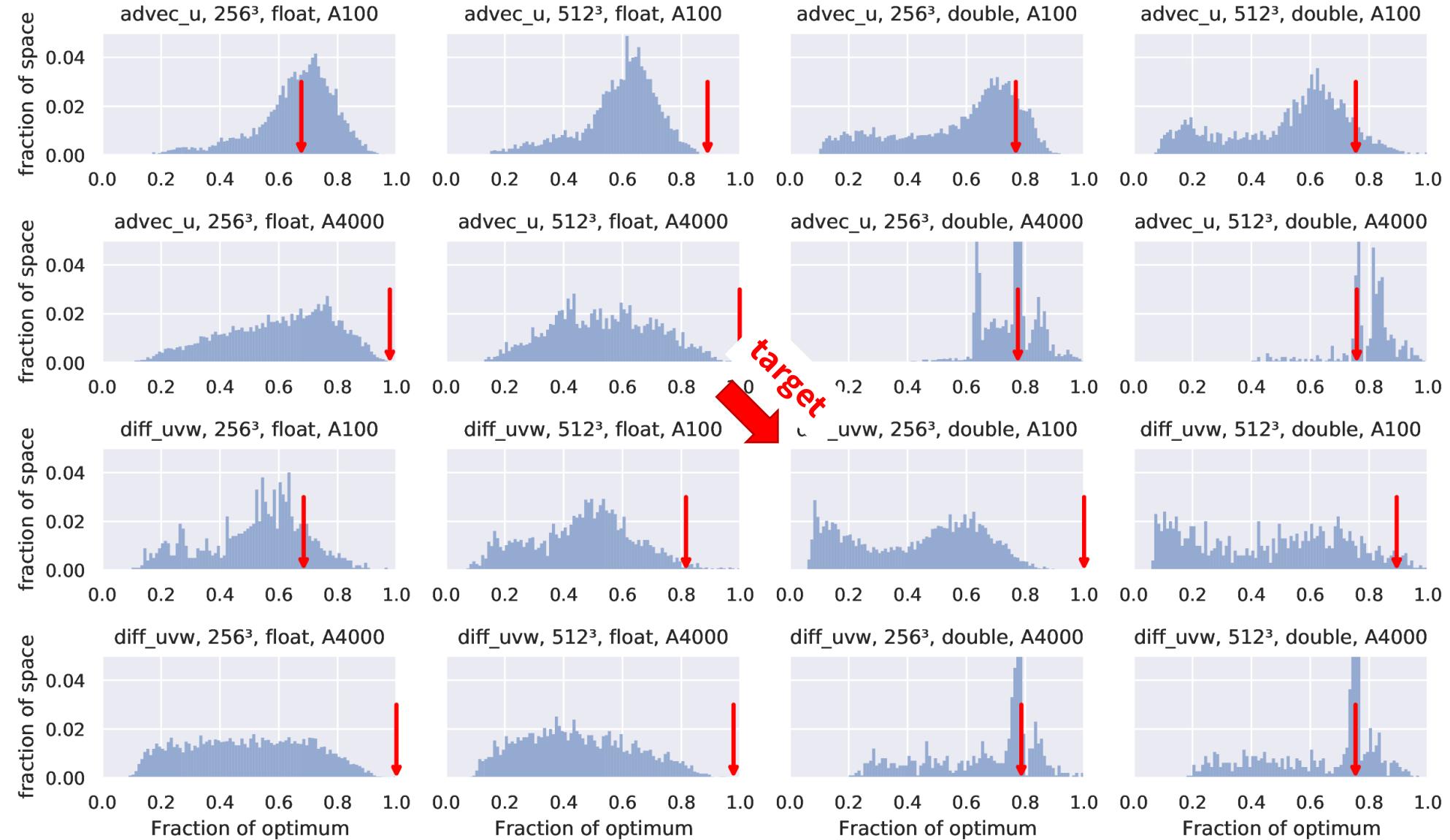


target



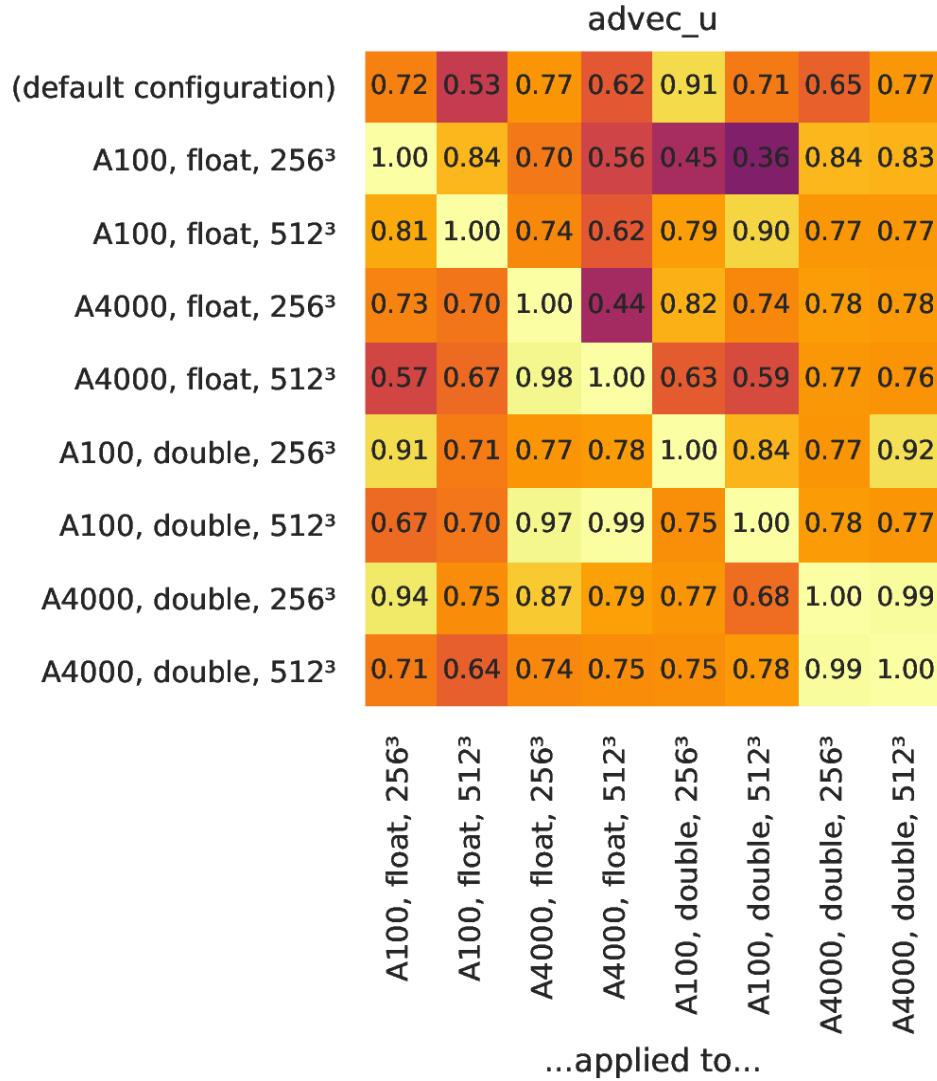




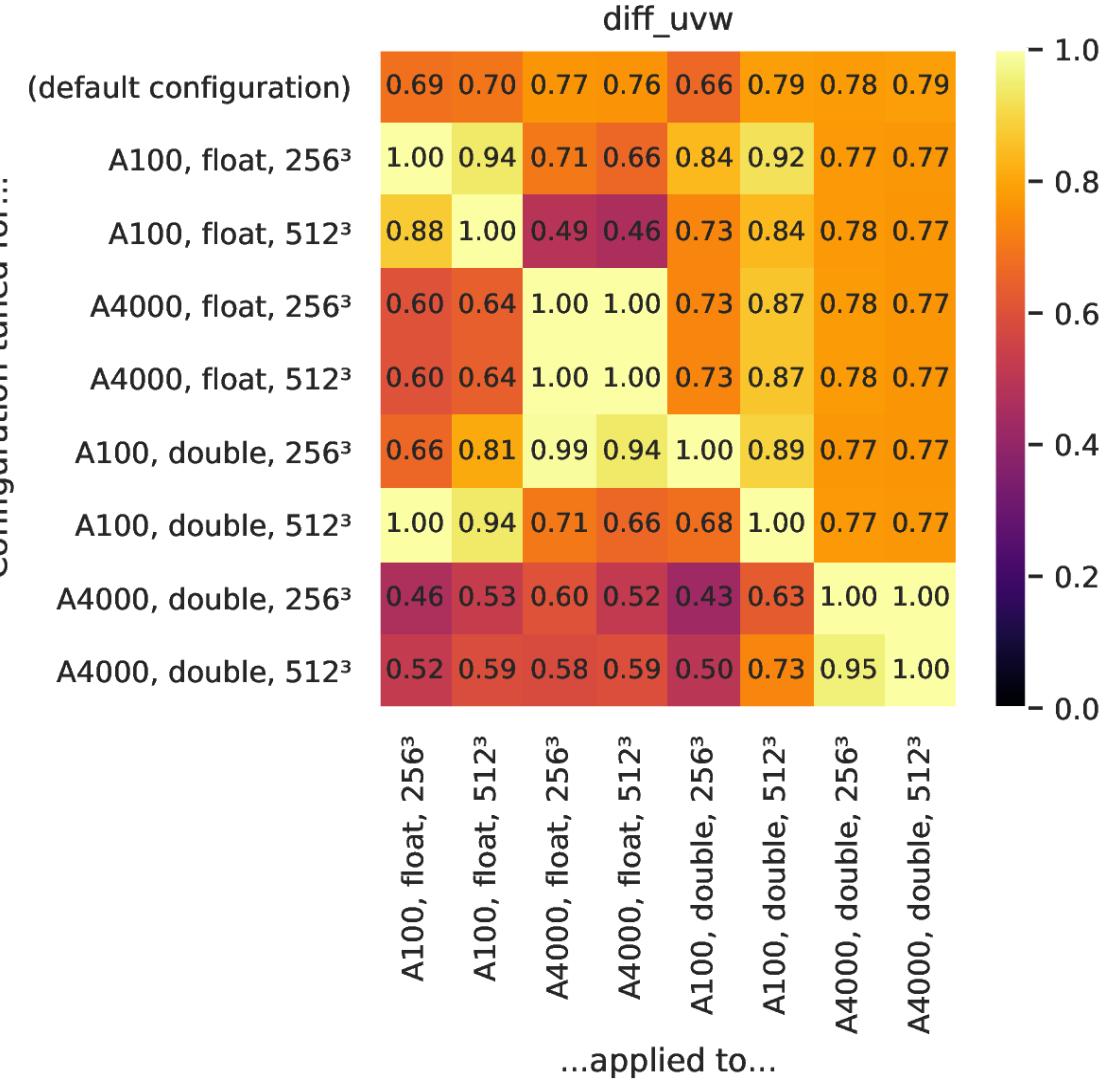


Portability

Configuration tuned for...

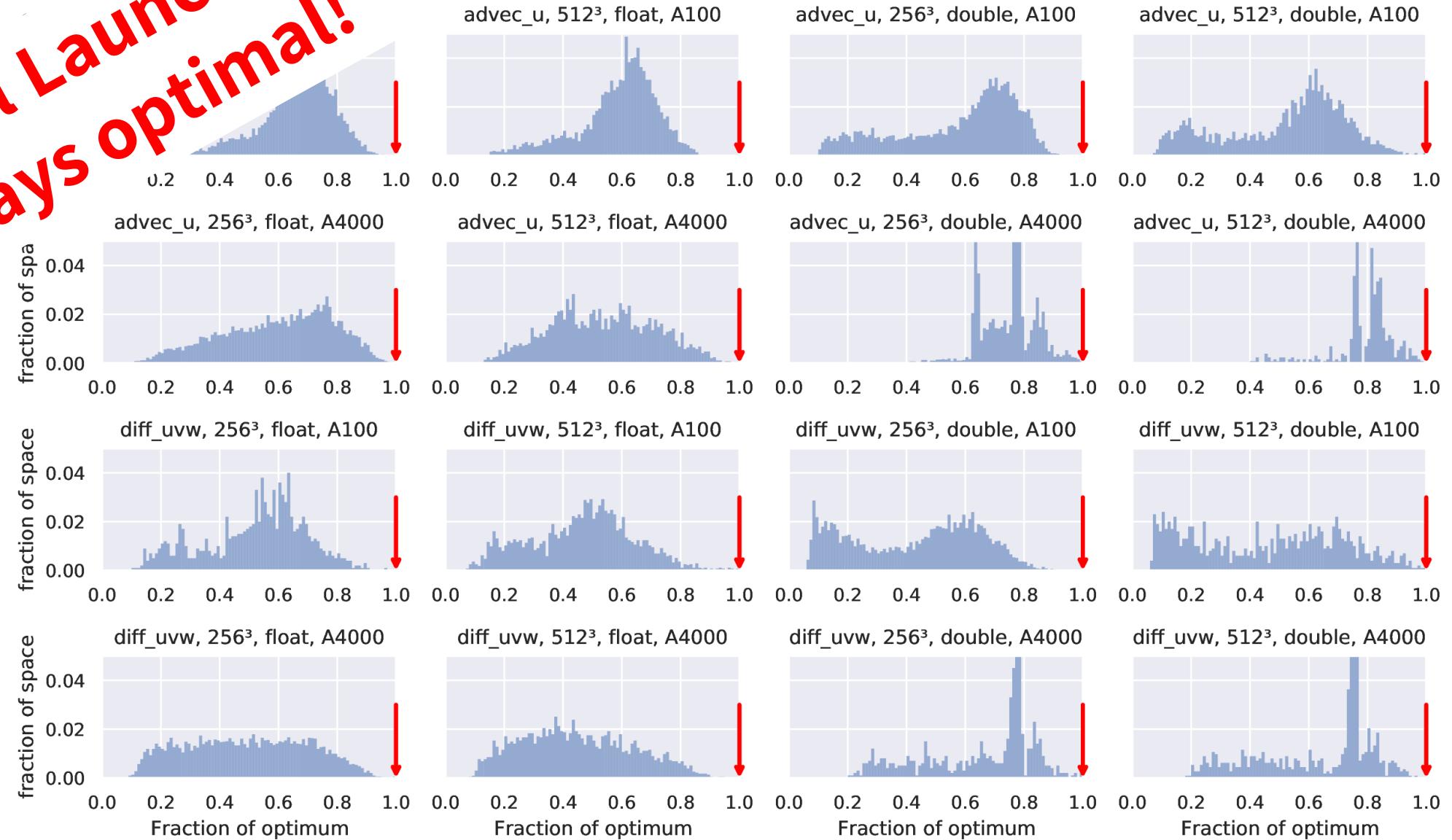


Configuration tuned for...





Kernel Launcher:
always optimal!



MicroHH

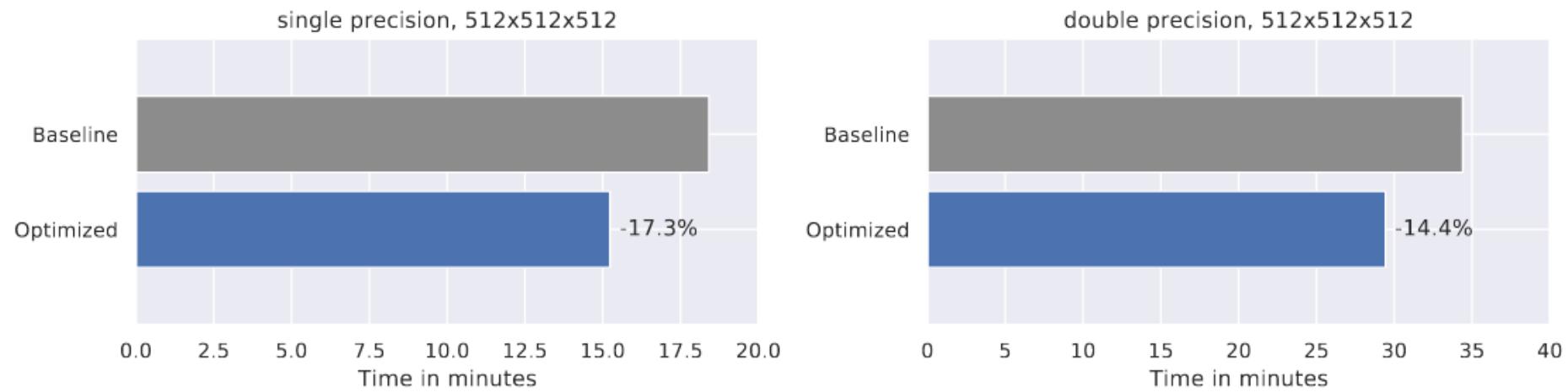


Figure 4: Run-time in minutes of full application for 5000 time iterations.

Conclusions

- Kernel Launcher: integrate auto tuning into CUDA applications
 - Exports kernel captures
 - Imports wisdom files
 - Run-time kernel selection means optimal-performance portability
- Contact us
 - s.heldens@esciencecenter.nl
 - kerneltuner.github.io
 - github.com/KernelTuner/



» Kernel Launcher

[View page source](#)

Kernel Launcher



Kernel Launcher

Kernel Launcher is a C++ library that makes it easy to dynamically compile CUDA kernels at run time (using [NVRTC](#)) and call them in easy type-safe way using C++ magic. There are two reasons for using run-time compilation:

- Kernels that have tunable parameters (block size, elements per thread, loop unroll factors, etc.) where the optimal configuration can only be determined at runtime since it depends dynamic factors such as the type of GPU and the problem size.
- Improve performance by injecting runtime values as compile-time constant values into kernel code (dimensions, array strides, weights, etc.).

Basic Example

This sections shows a basic code example. See [Tutorial](#) for a more advance example.

Consider the following CUDA kernel for vector addition. This kernel has a template parameter [T](#) and a tunable parameter [ELEMENTS_PER_THREAD](#).

```
1 template <typename T>
2 __global__
3 void vector_add(int n, T* C, const T* A, const T* B) {
4     for (int k = 0; k < ELEMENTS_PER_THREAD; k++) {
5         int i = blockIdx.x * ELEMENTS_PER_THREAD * blockDim.x + k * blockDim.x + threadIdx.x;
6
7         if (i < n) {
8             C[i] = A[i] + B[i];
9         }
10    }
11 }
```

The following C++ snippet shows how to use *Kernel Launcher* in host code:

https://kerneltuner.github.io/kernel_launcher

Let's stay
in touch



www.eScienceCenter.nl



info@esciencecenter.nl



+31 (0)20 460 4770

Funding

ESiWACE2 has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 823988.



ESiWACE3 is funded by EuroHPC JU and national co-funding bodies under grant agreement No 101093054.

The ConFu project is funded by the Netherlands eScience Center (file number 00020223-A).