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

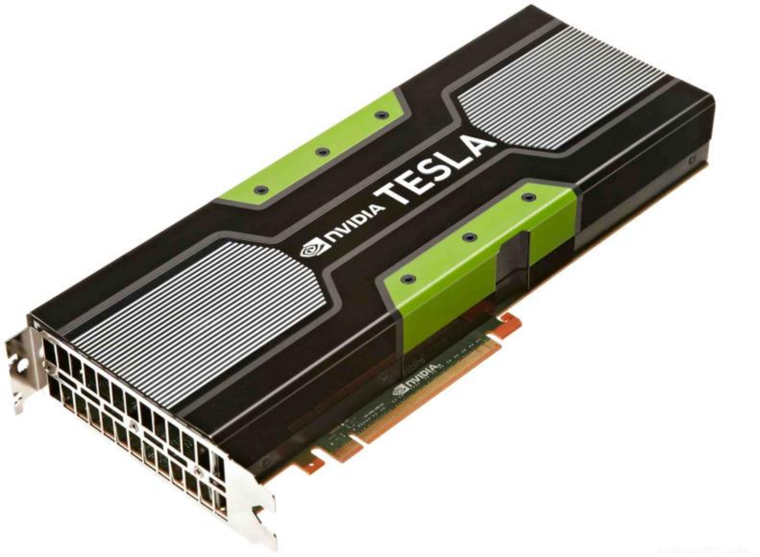
Snellius
2021-now
14 PFLOPS
144x NVIDIA A100

netherlands eScience center

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?
- …

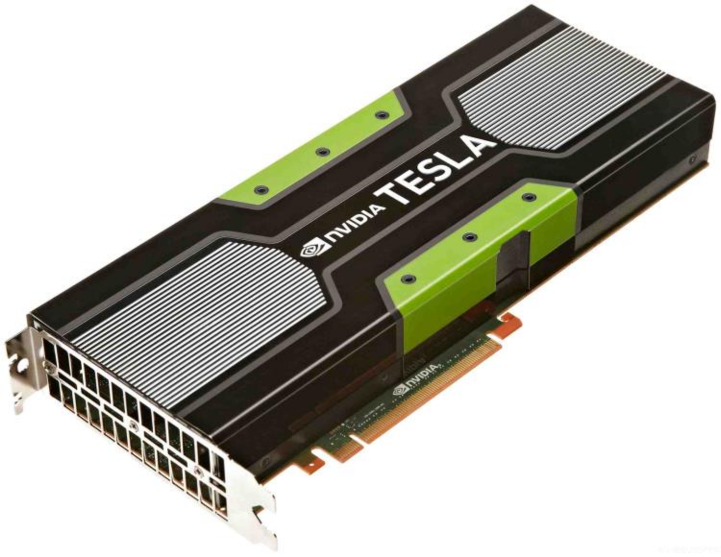


netherlands eScience center

GPU Programming

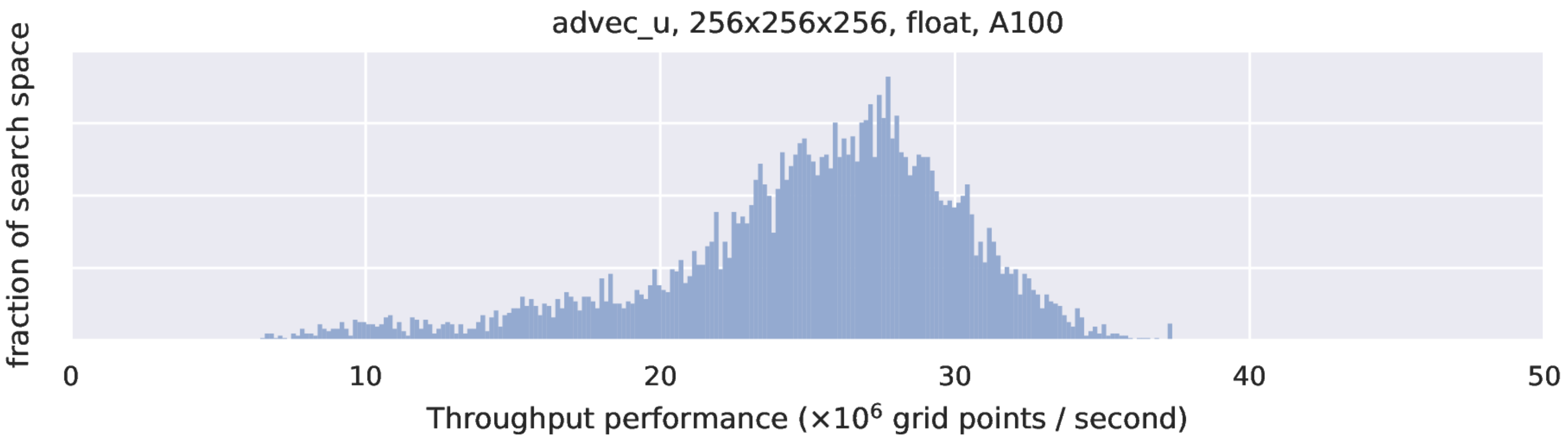
Writing GPU kernels requires extensive tuning:

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



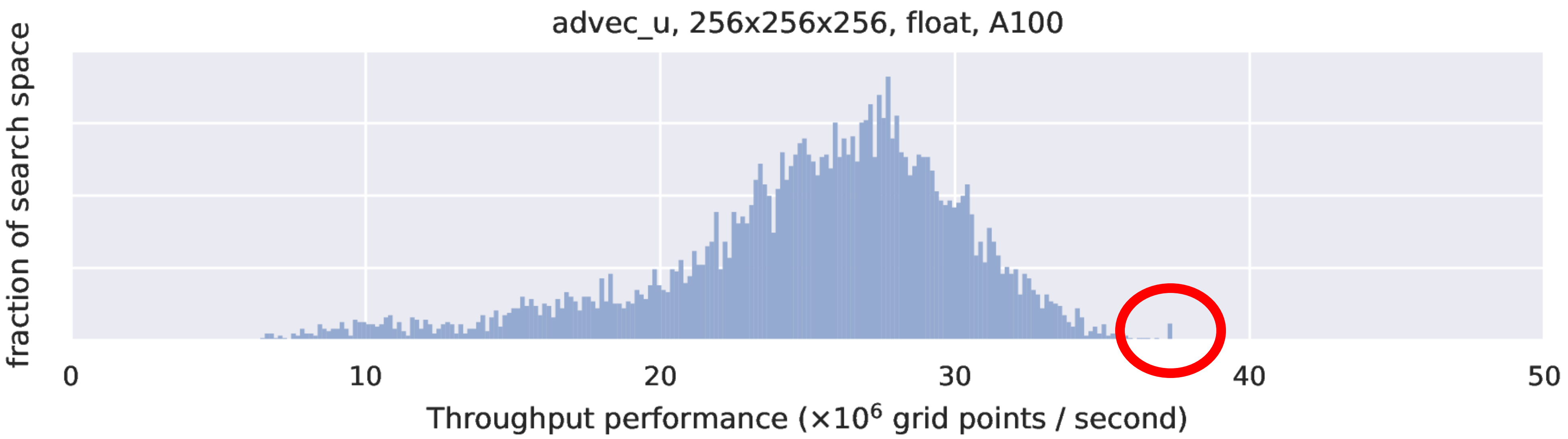
⇛ Large search space!

Performance example

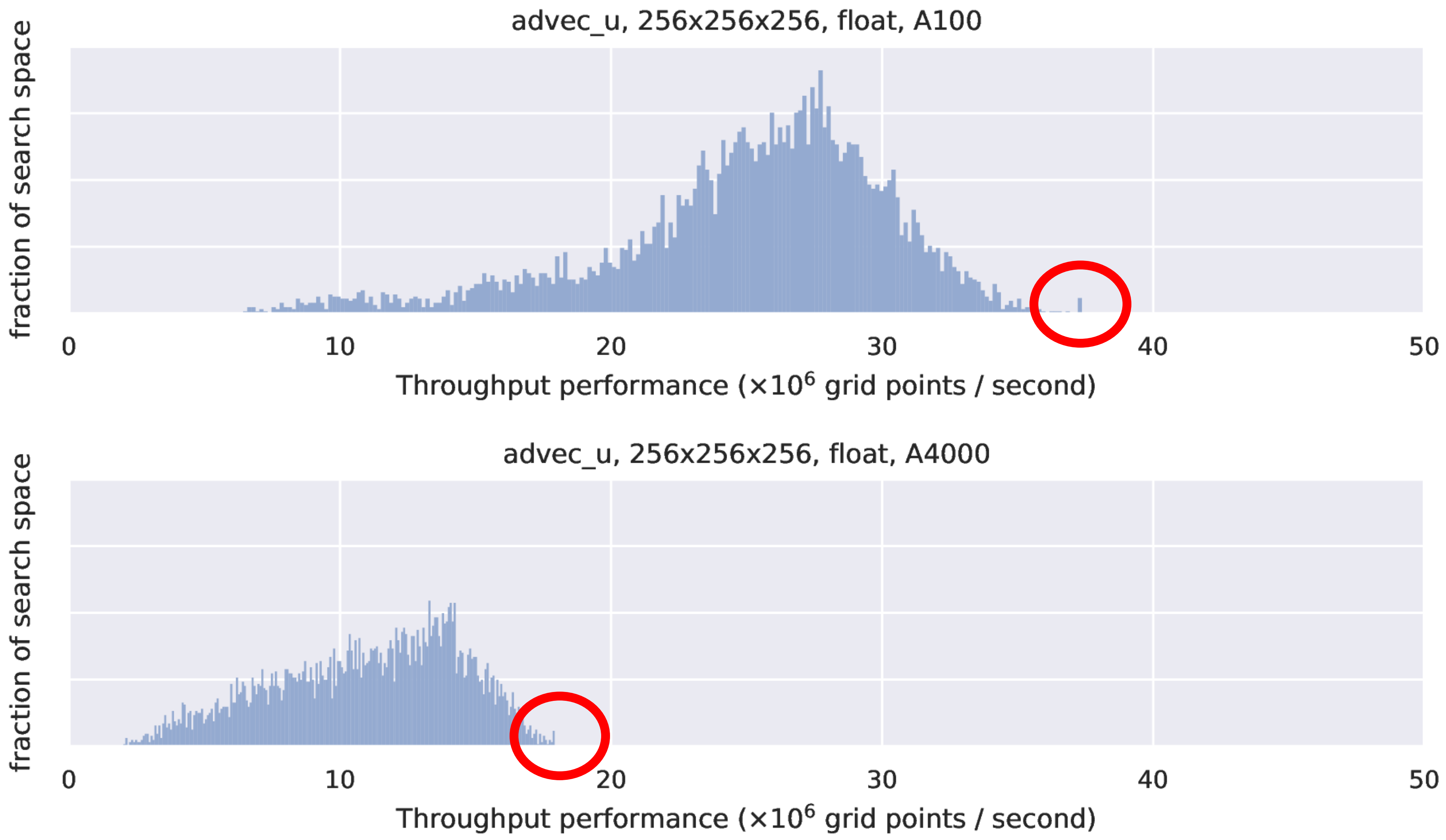


netherlands eScience center

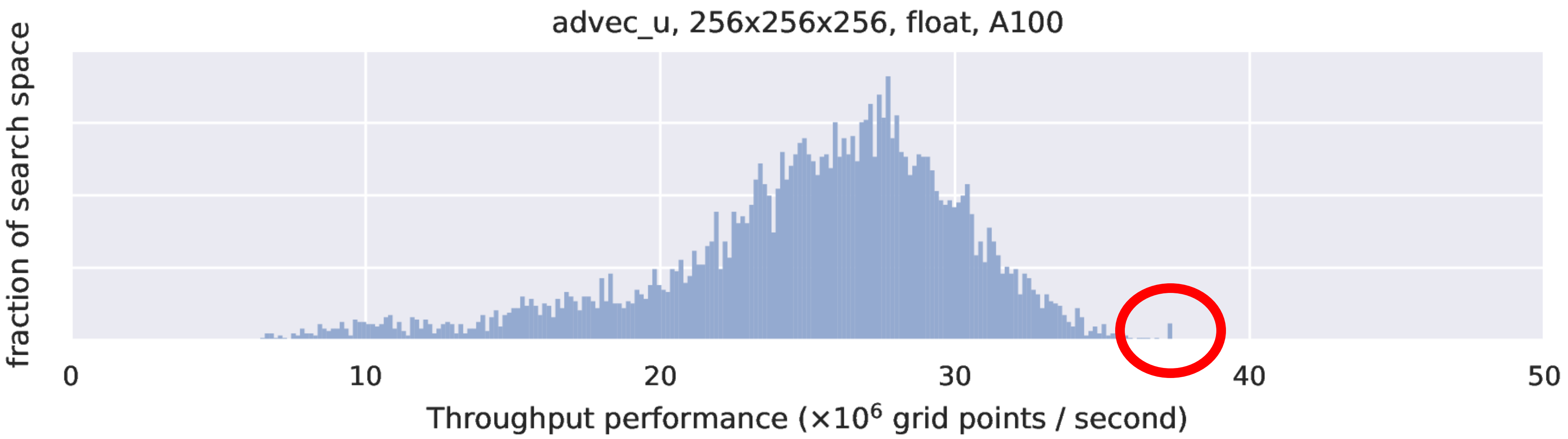
Performance example

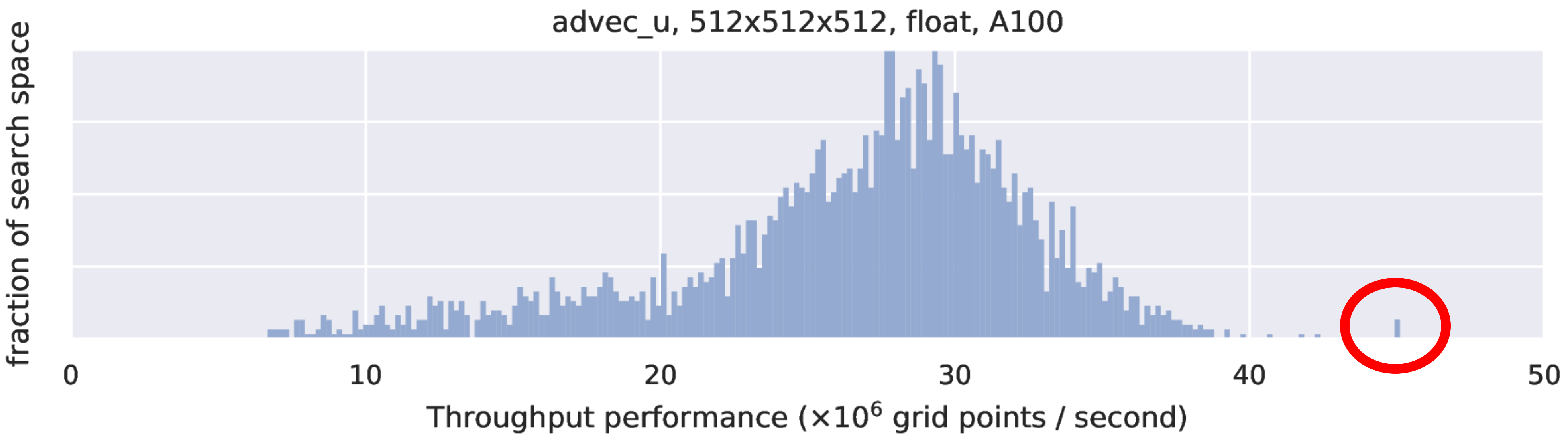


Performance example



Performance example

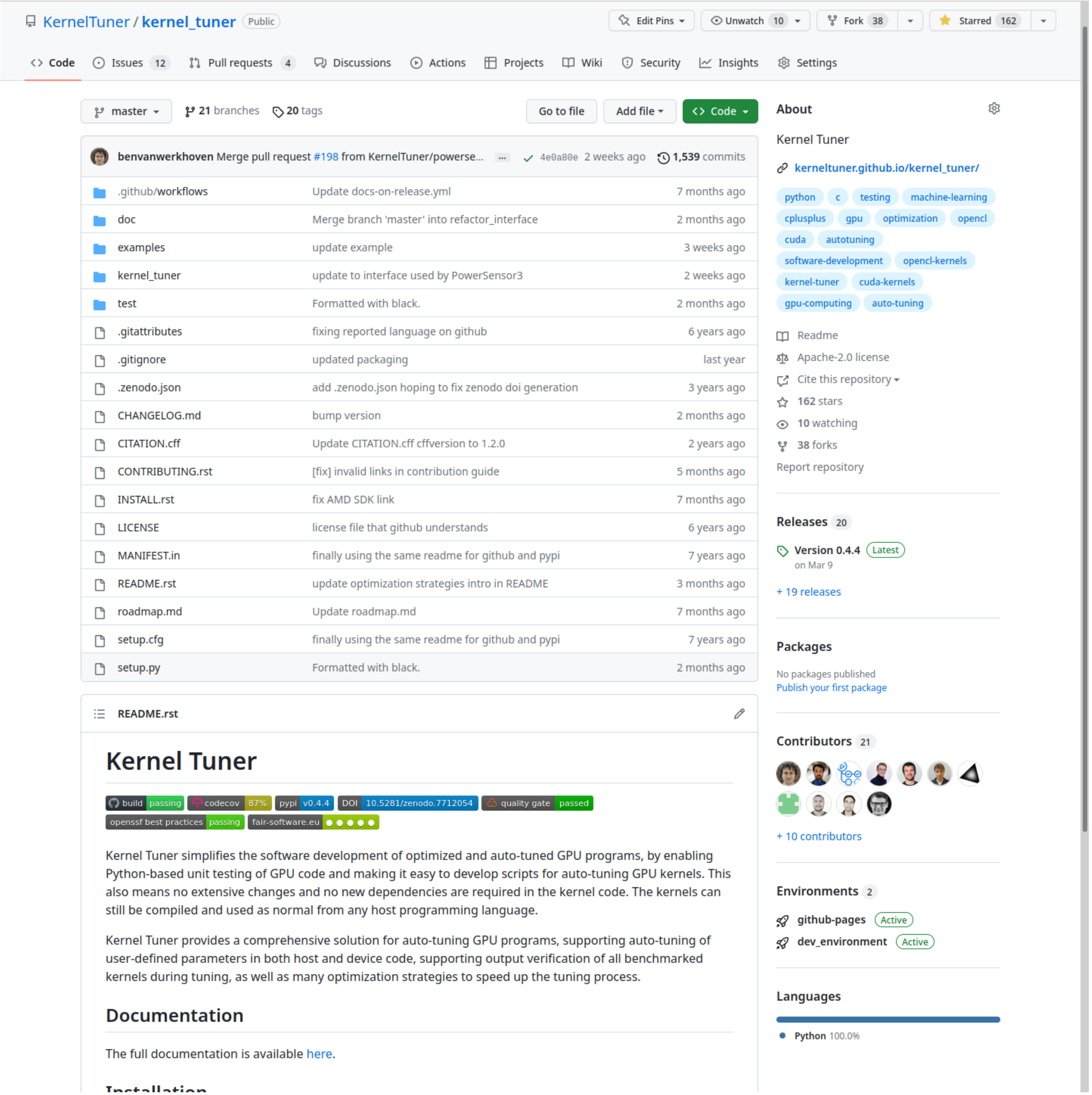




netherlands eScience center

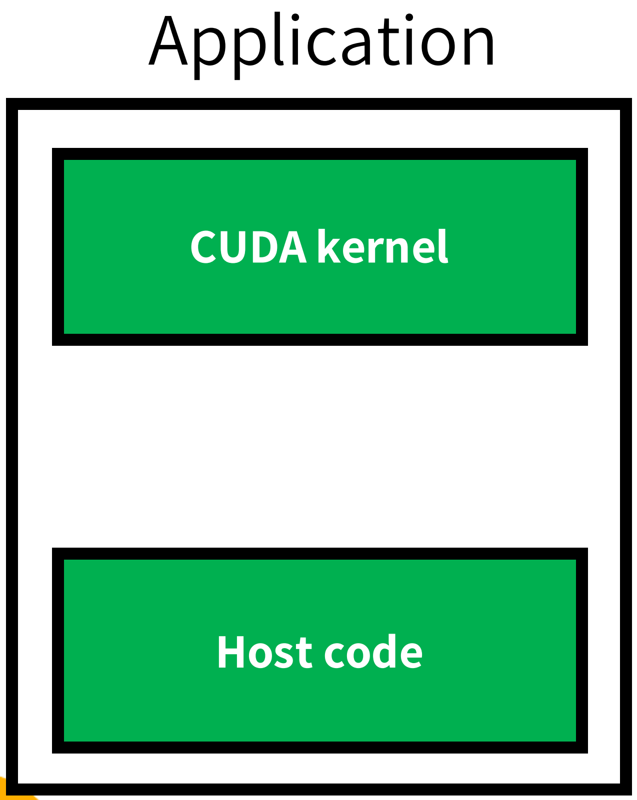
Kernel Tuner

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

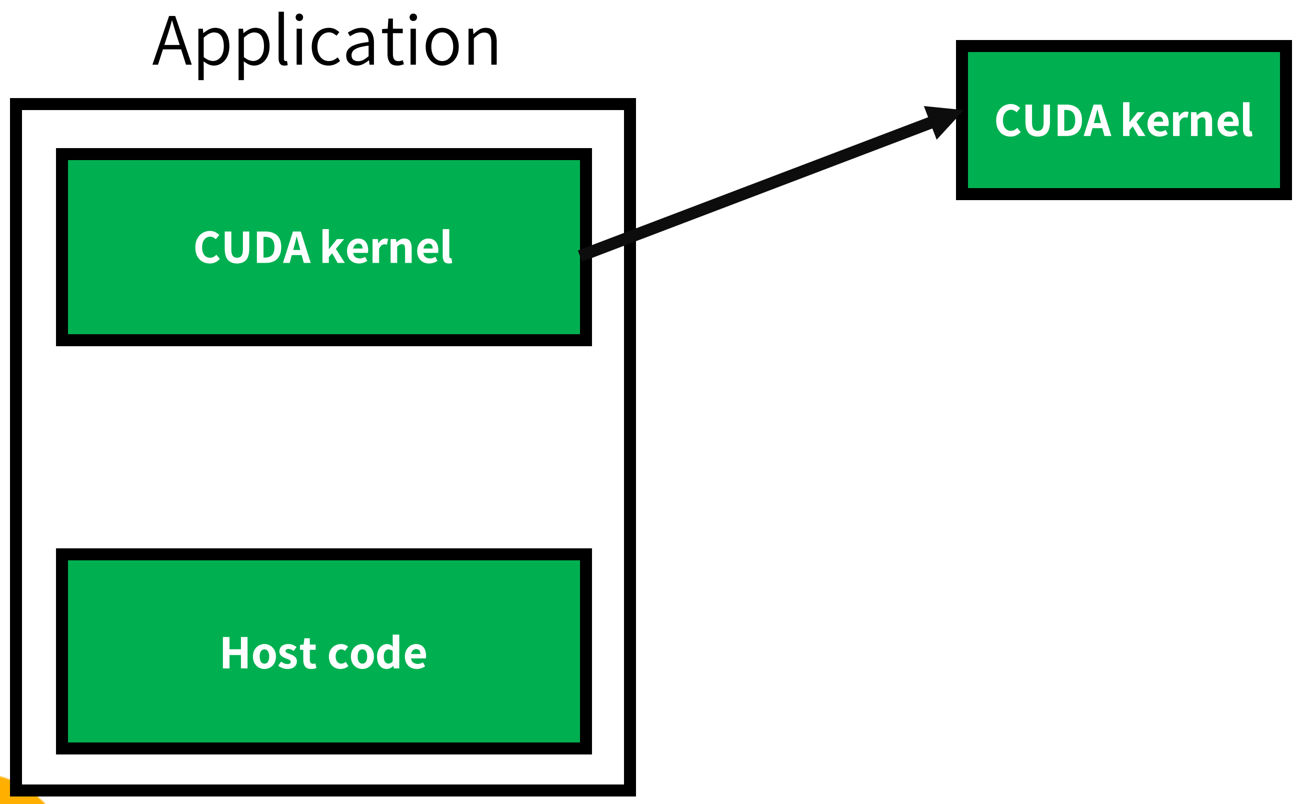


https://kerneltuner.github.io/

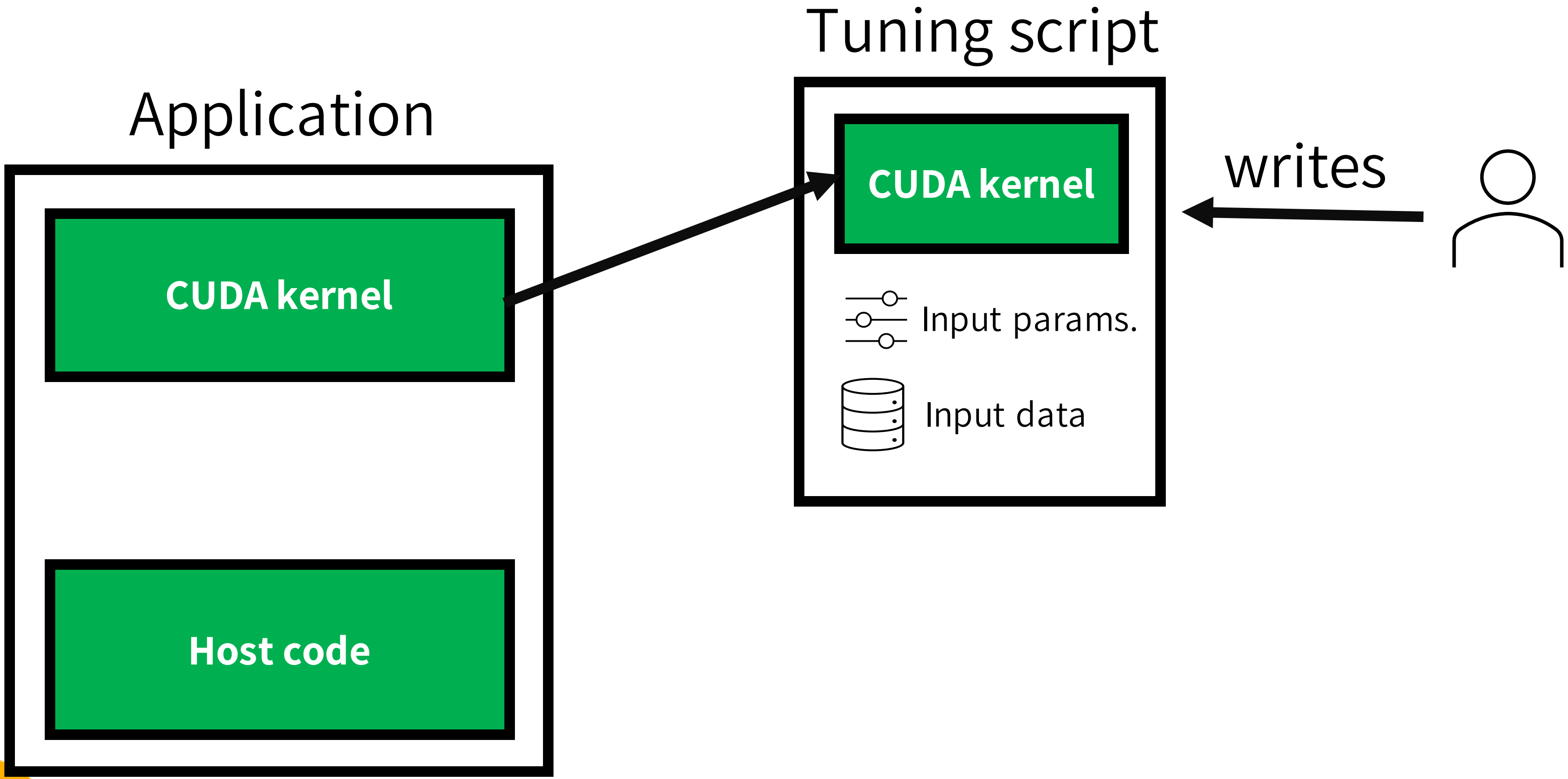
Previous workflow



Previous workflow

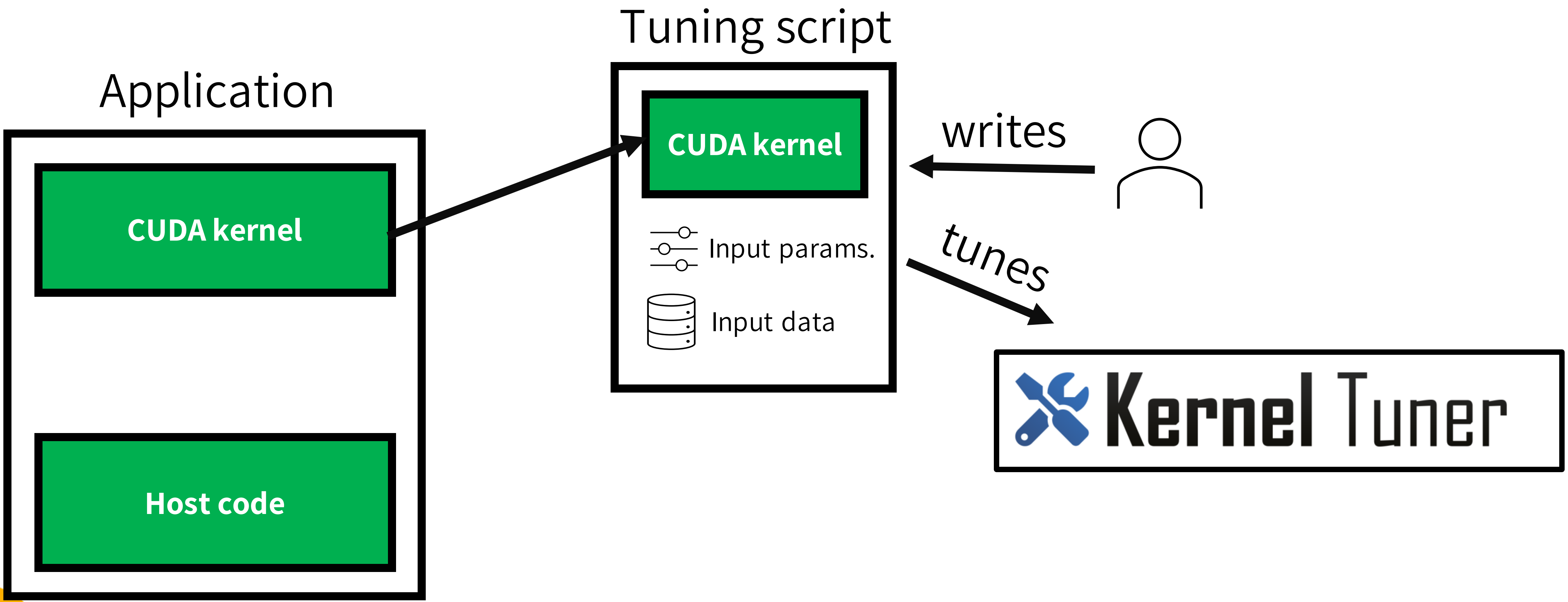


Previous workflow

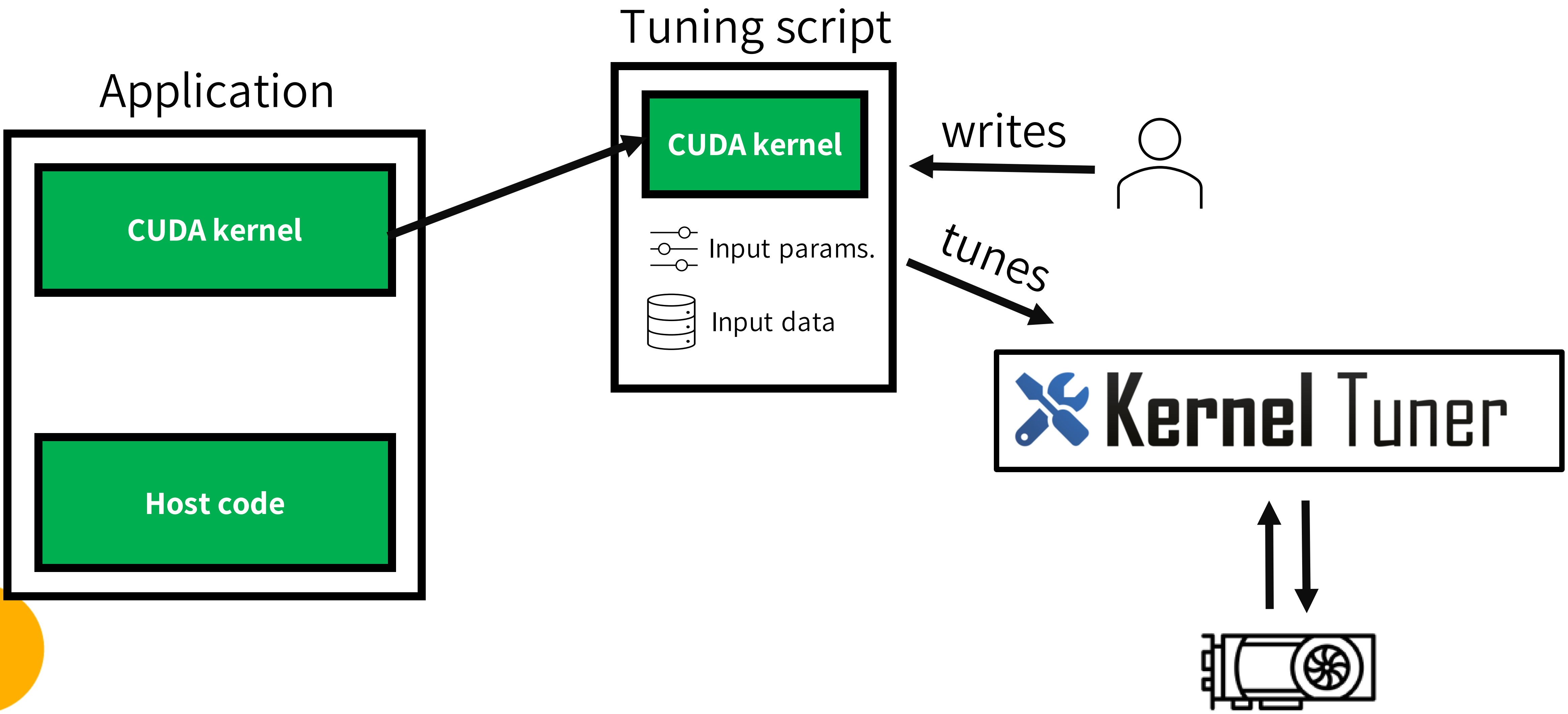


netherlands eScience center

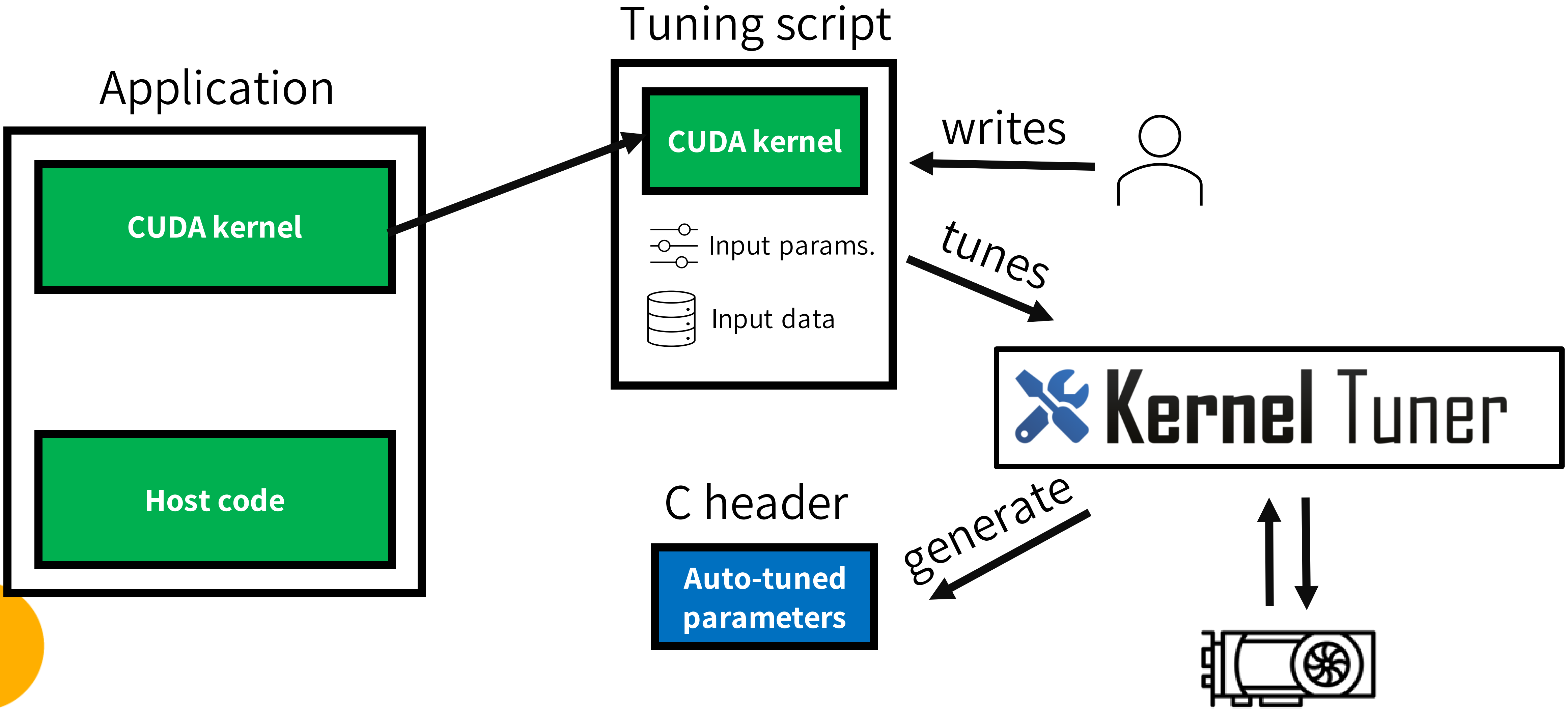
Previous workflow



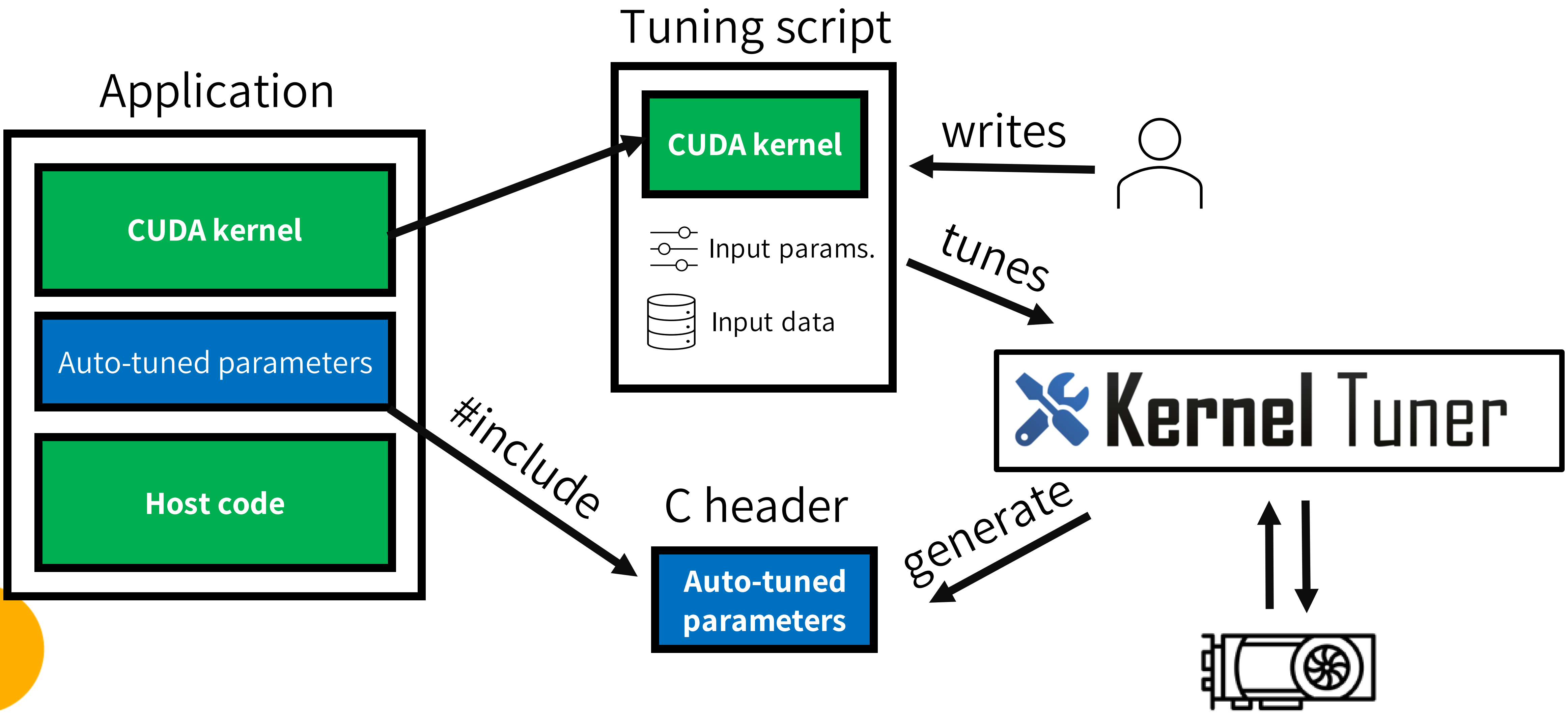
Previous workflow



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

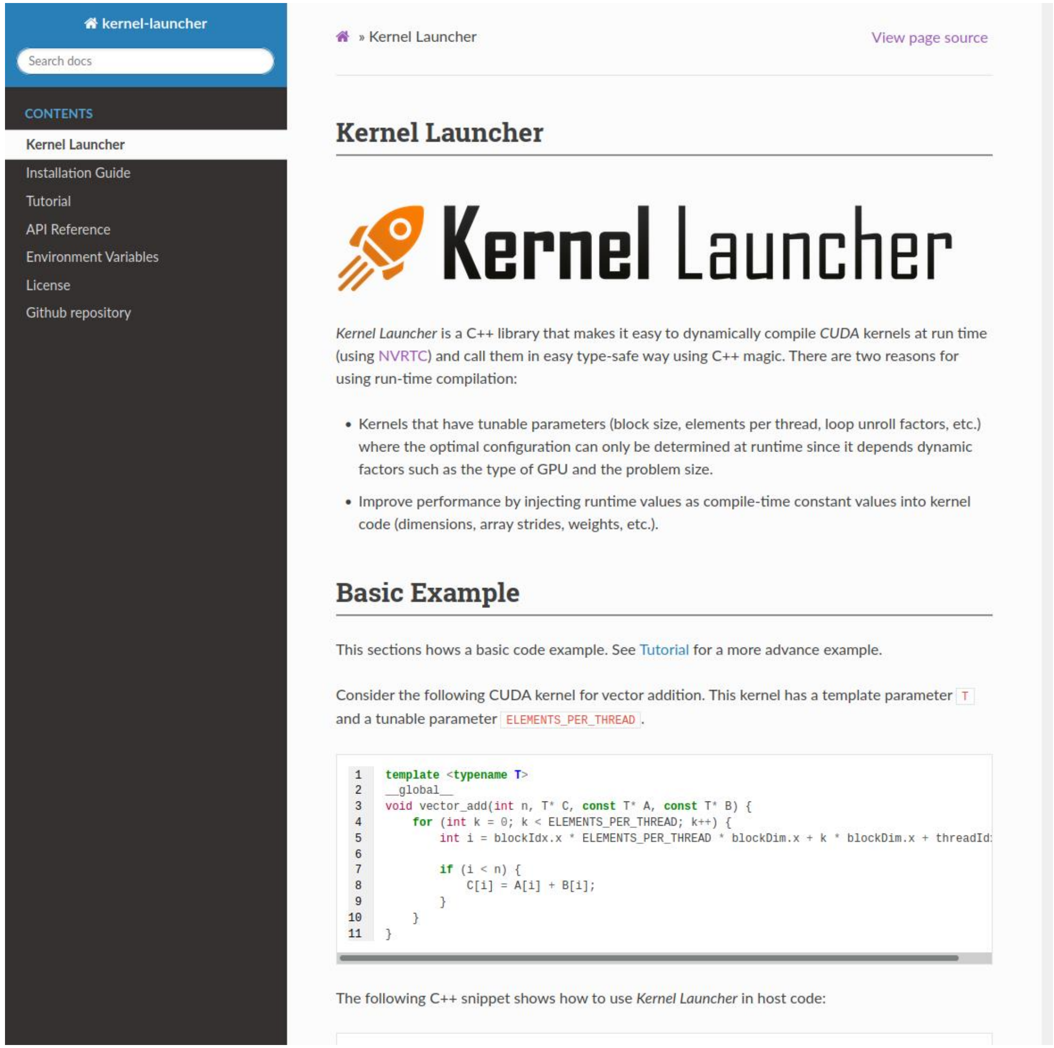
- Currently, fixed GPU during compilation

#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

Search docs

CONTENTS

- Kernel Launcher
- Installation Guide
- Tutorial
- API Reference
- Environment Variables
- License
- Github repository

» 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 hows a basic code example. See Tutorial for a more advance example.

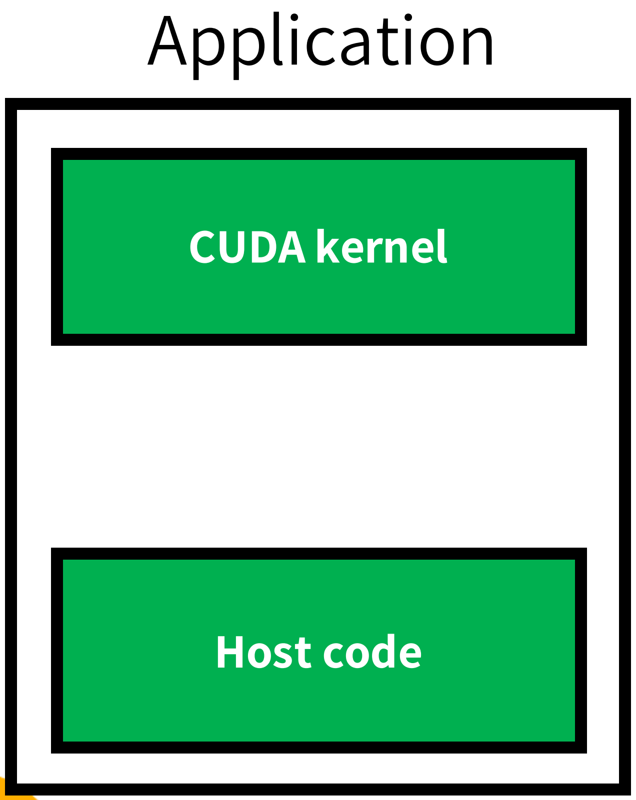
Consider the following CUDA kernel for vector addition. This kernel has a template parameter `T` and a tunable parameter `ELEMENTS_PER_THREAD`.

```
template <typename T>
__global__
void vector_add(int n, T* C, const T* A, const T* B) {
    for (int k = 0; k < ELEMENTS_PER_THREAD; k++) {
        int i = blockIdx.x * ELEMENTS_PER_THREAD * blockDim.x + k * blockDim.x + threadId
        
        if (i < n) {
            C[i] = A[i] + B[i];
        }
    }
}
```

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

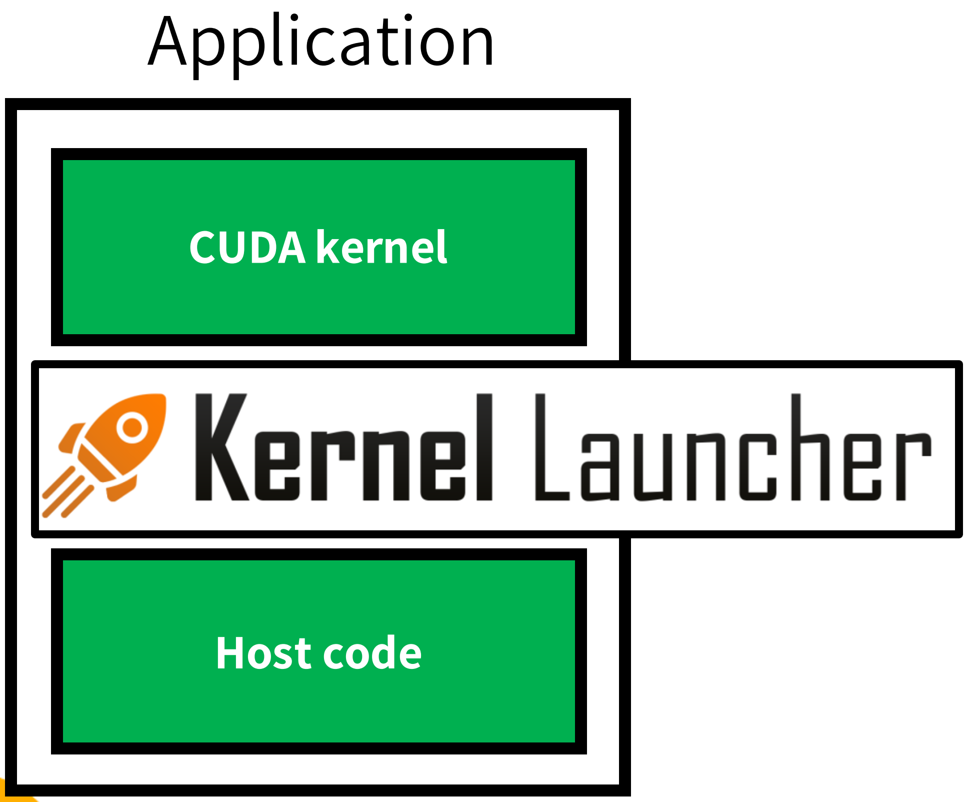
https://kerneltuner.github.io/kernel_launcher

New Kernel Launcher workflow

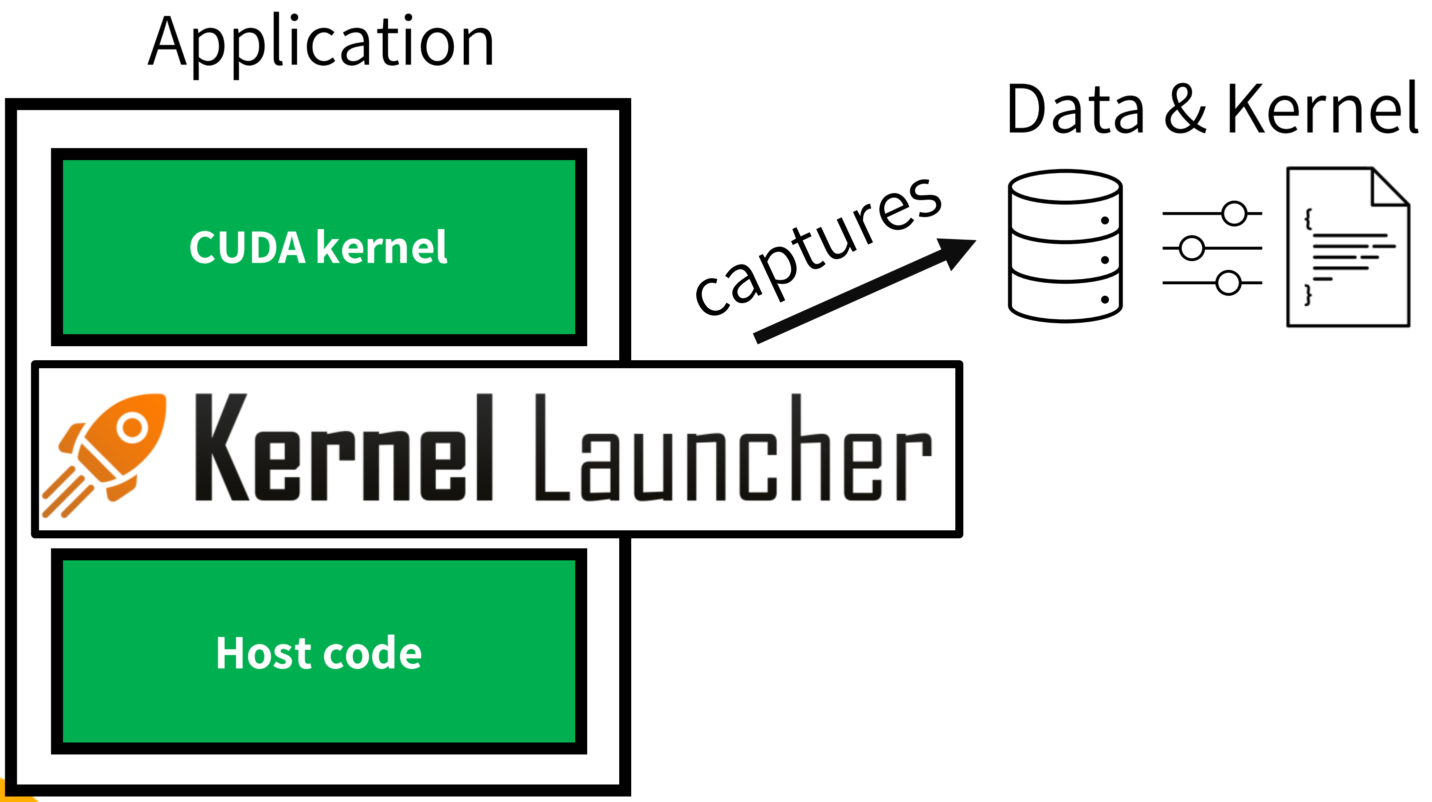


netherlands eScience center

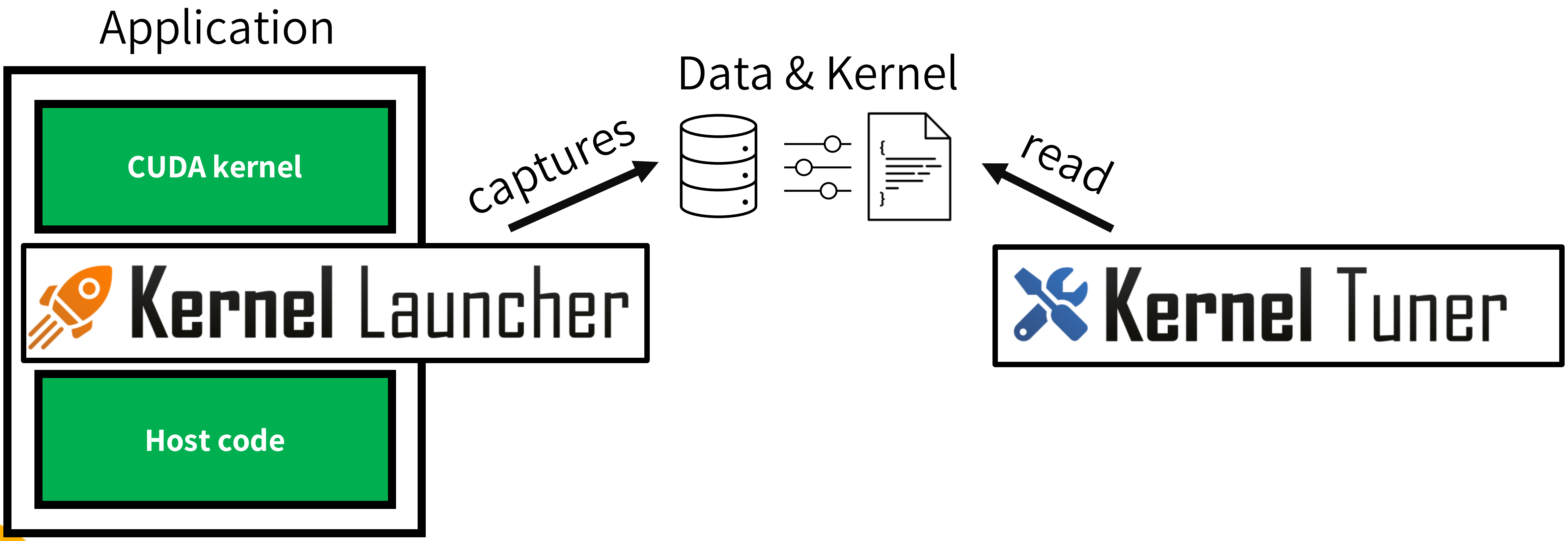
New Kernel Launcher workflow



New Kernel Launcher workflow

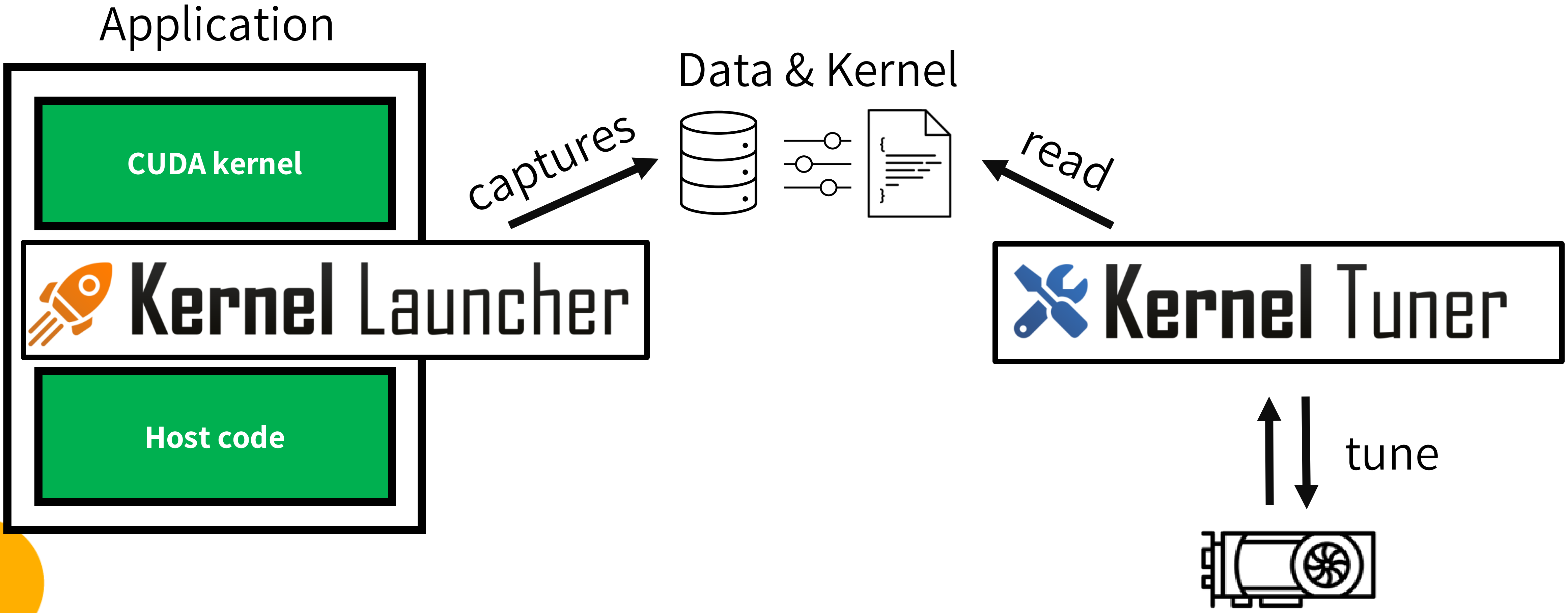


New Kernel Launcher workflow



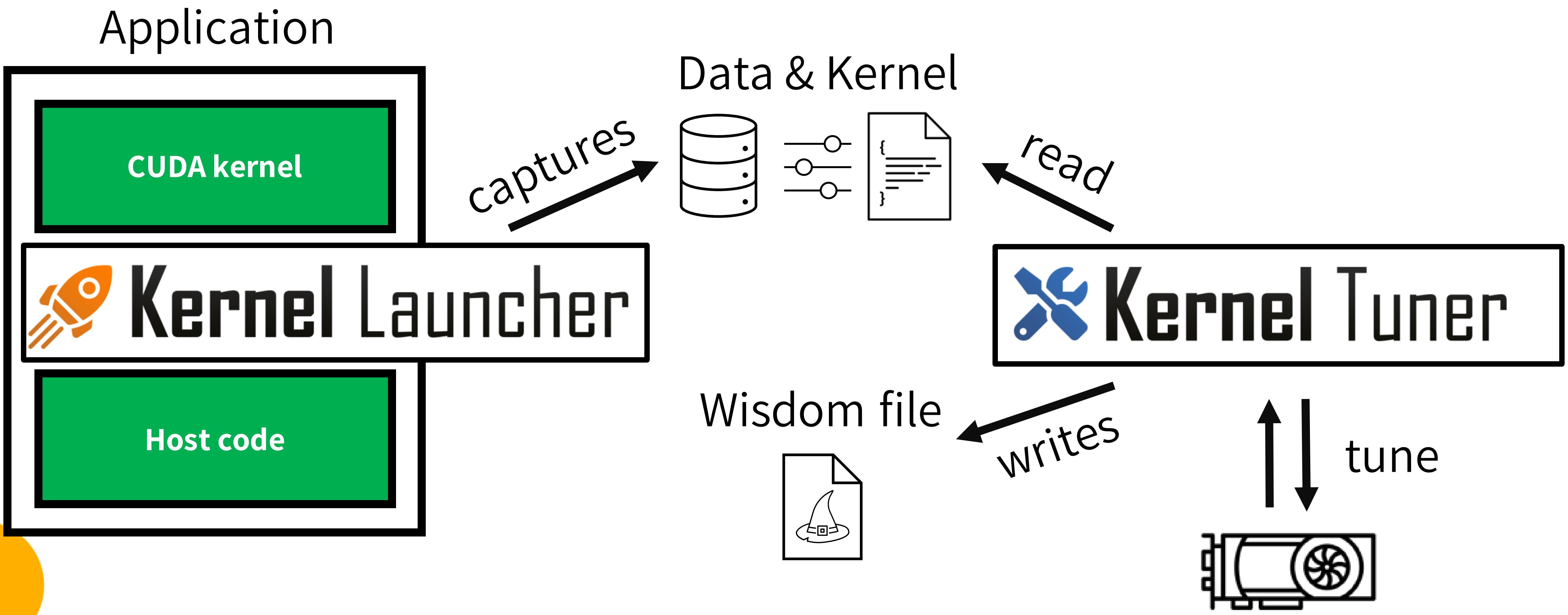
netherlands eScience center

New Kernel Launcher workflow

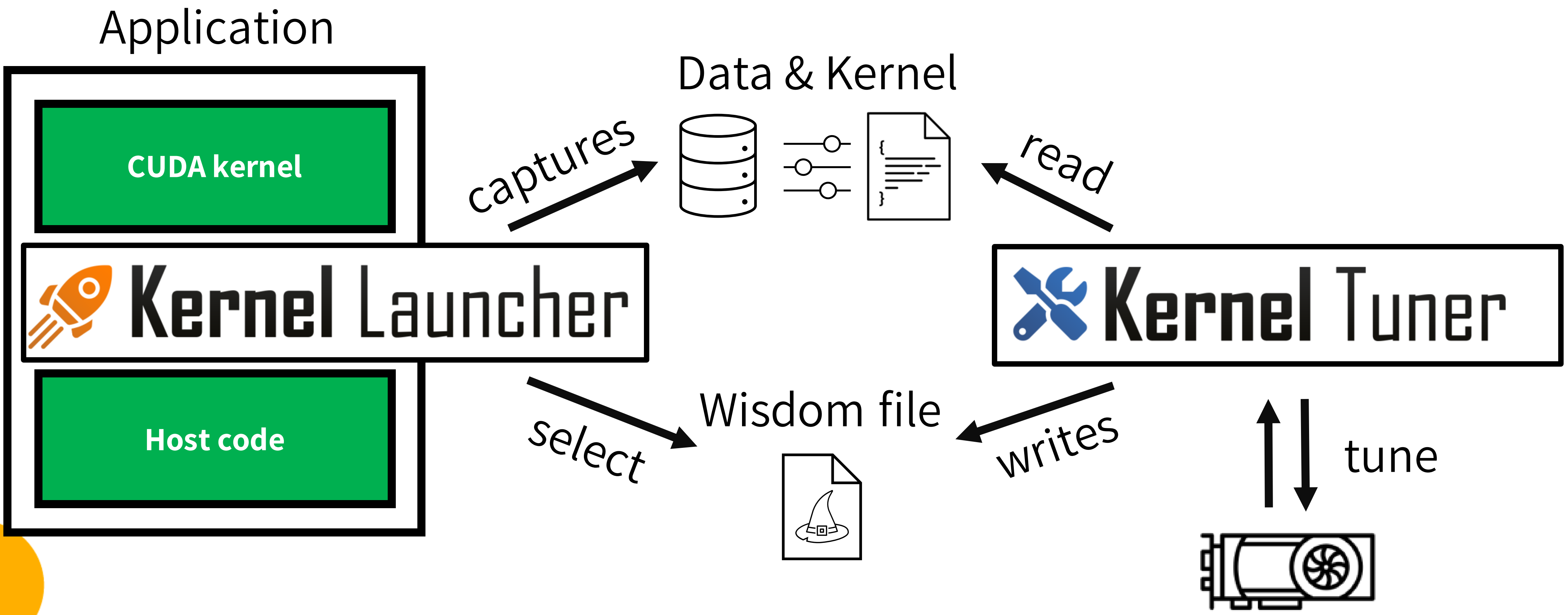


netherlands eScience center

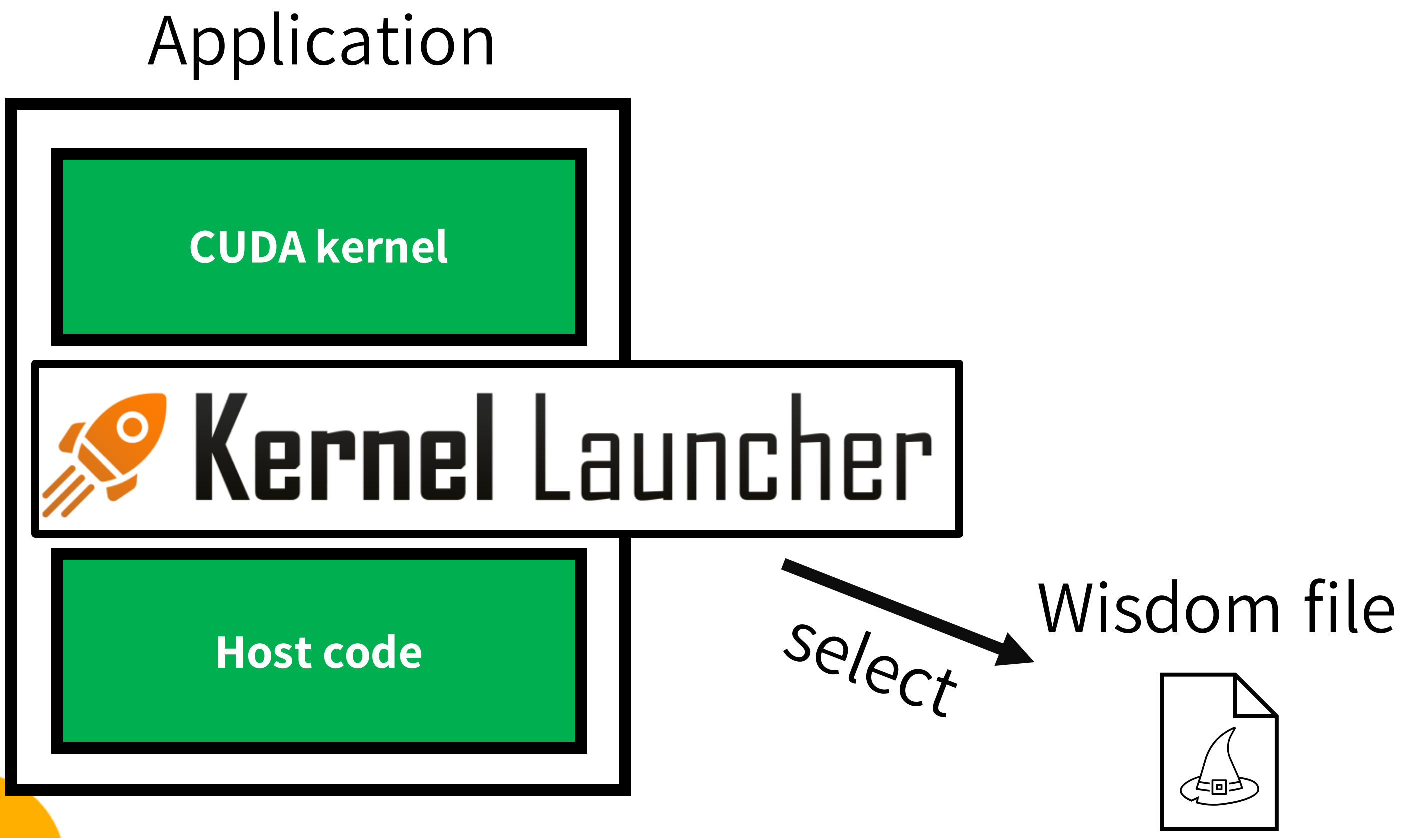
New Kernel Launcher workflow



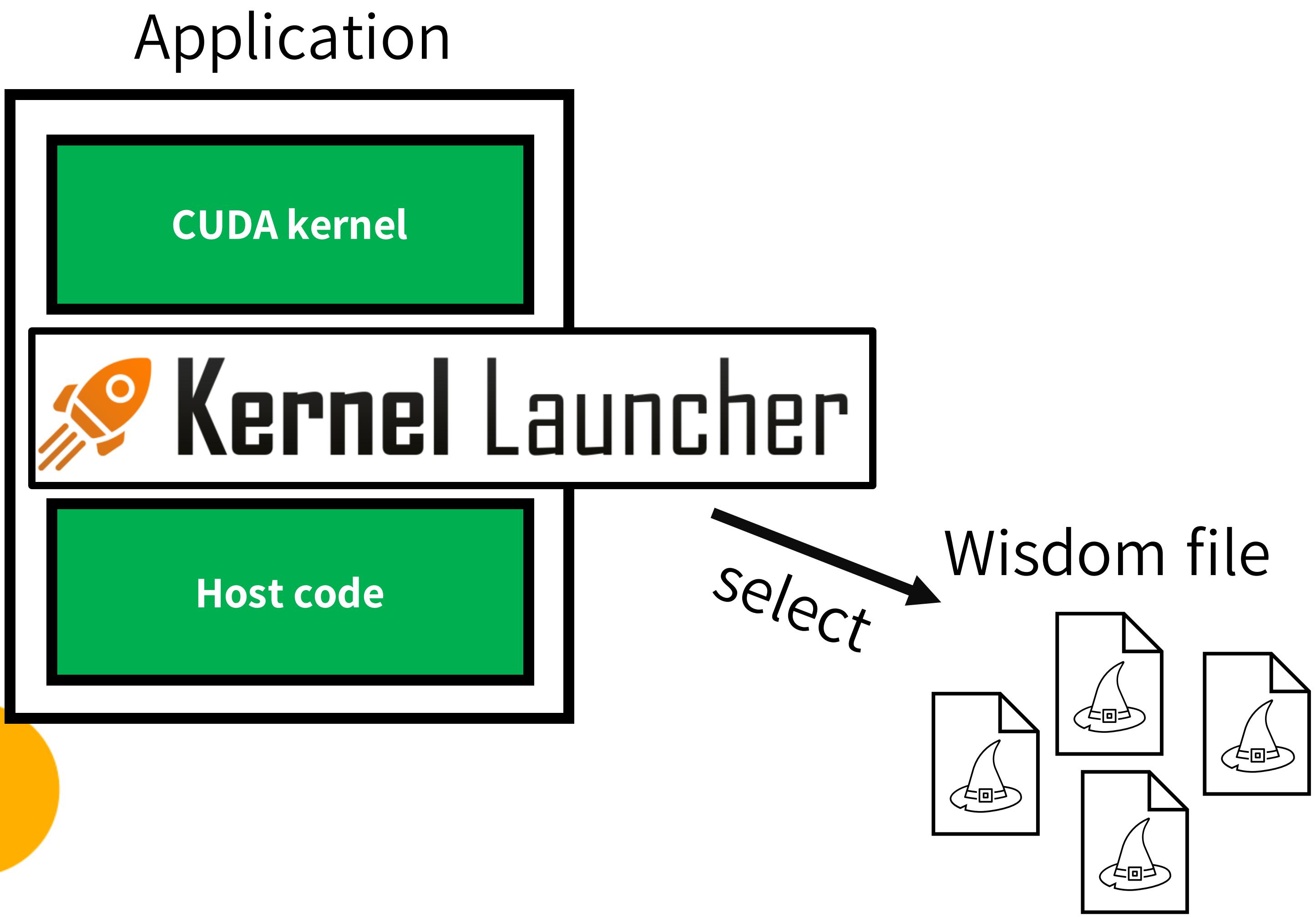
New Kernel Launcher workflow



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

netherlands eScience center

Wisdom files

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

Code example

Kernel-Launcher-based CUDA code

host.cpp

```cpp
int main() {
  // Initialize cuda
  // cudaMalloc, cudaMemcpy, ...

  // Set problem size
  int n = 1_048_576;

  // Set grid/block size
  int block_size = 512;
  int grid_size = n / block_size;

  // Launch kernel
  vector_add<<grid_size, block_size>>>(n, A, B, c);


}
```

kernel.cu

```cuda
__global__ void vector_add(
    int n, float* A, float* B, float* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  C[i] = A[i] + B[i];
}
```

Code example

Kernel-Launcher-based CUDA code

host.cpp

```cpp
#include "kernel_launcher.hpp"

int main() {
  // Initialize cuda
  // cudaMalloc, cudaMemcpy, ...

  // Set problem size
  int n = 1_048_576;

  // Set grid/block size
  //int block_size = 512;
  //int grid_size = n / block_size;

  // Launch kernel
  kernel_launcher::launch(
    kernel_launcher::PragmaKernel("vector_add", "kernel.cu"),
    n, A, B, C);
}
```

kernel.cu

```cuda
#pragma kernel tune(threads_per_block=32, 64, 128, 256)
#pragma kernel problem_size(n)
#pragma kernel block_size(threads_per_block)
#pragma kernel buffers(A[n], B[n], C[n])
__global__ void vector_add(
    int n, float* A, float* B, float* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  C[i] = A[i] + B[i];
}
```

Code example

Kernel-Launcher-based CUDA code

host.cpp

```cpp
#include "kernel_launcher.hpp"

int main() {
  // Initialize cuda
  // cudaMalloc, cudaMemcpy, ...

  // Set problem size
  int n = 1_048_576;

  // Set grid/block size
  //int block_size = 512;
  //int grid_size = n / block_size;

  // Launch kernel
  kernel_launcher::launch(
    kernel_launcher::PragmaKernel("vector_add", "kernel.cu"),
    n, A, B, C);
}
```

kernel.cu

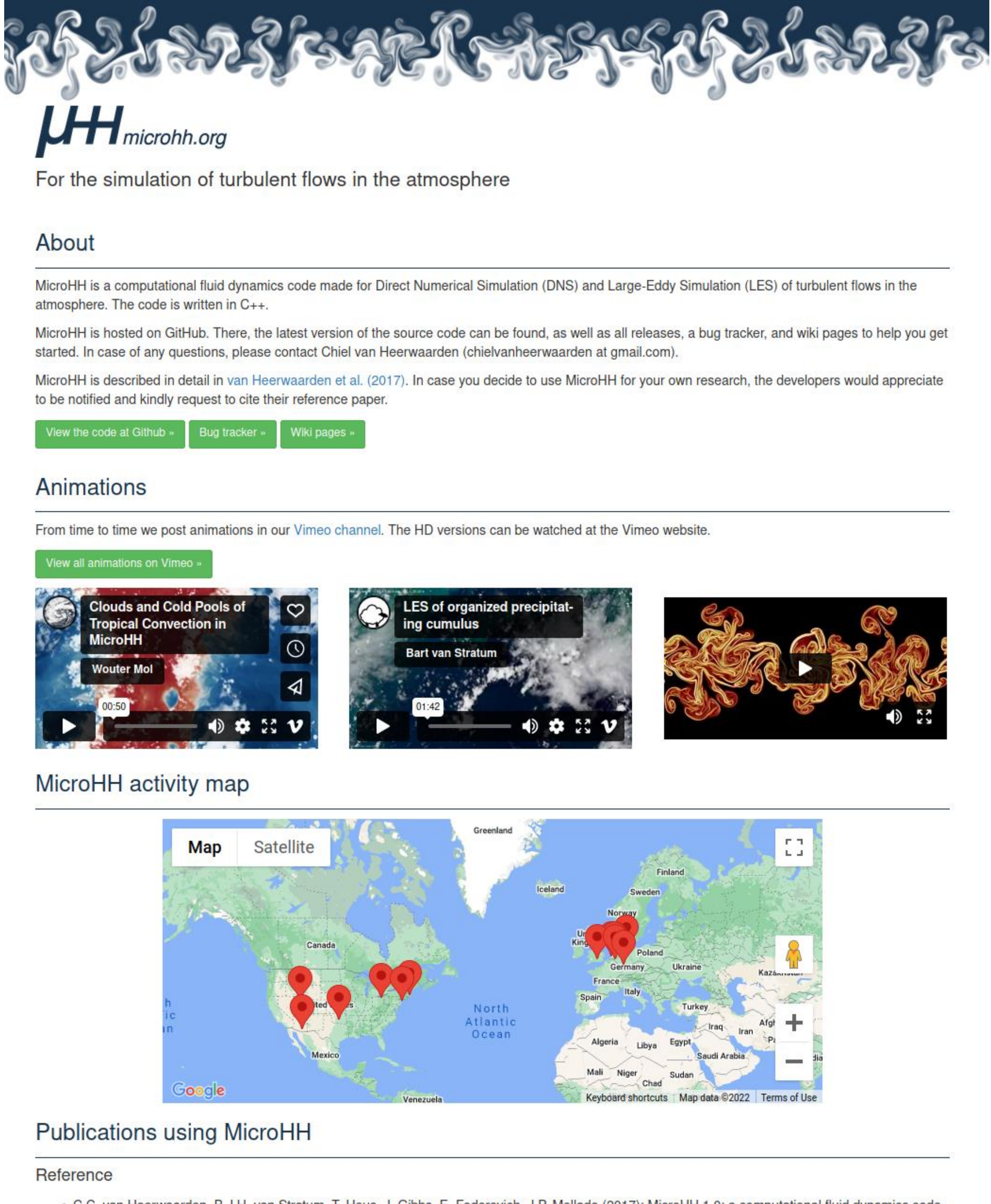
```cuda
#pragma kernel tune(threads_per_block=32, 64, 128, 256)
#pragma kernel problem_size(n)
#pragma kernel block_size(threads_per_block)
#pragma kernel buffers(A[n], B[n], C[n])
__global__ void vector_add(
    int n, float* A, float* B, float* C
) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  C[i] = A[i] + B[i];
}
```

netherlands eScience center

Use case: MicroHH

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

esiwace
CENTRE OF EXCELLENCE IN SIMULATION OF WEATHER AND CLIMATE IN EUROPE



Tuning MicroHH

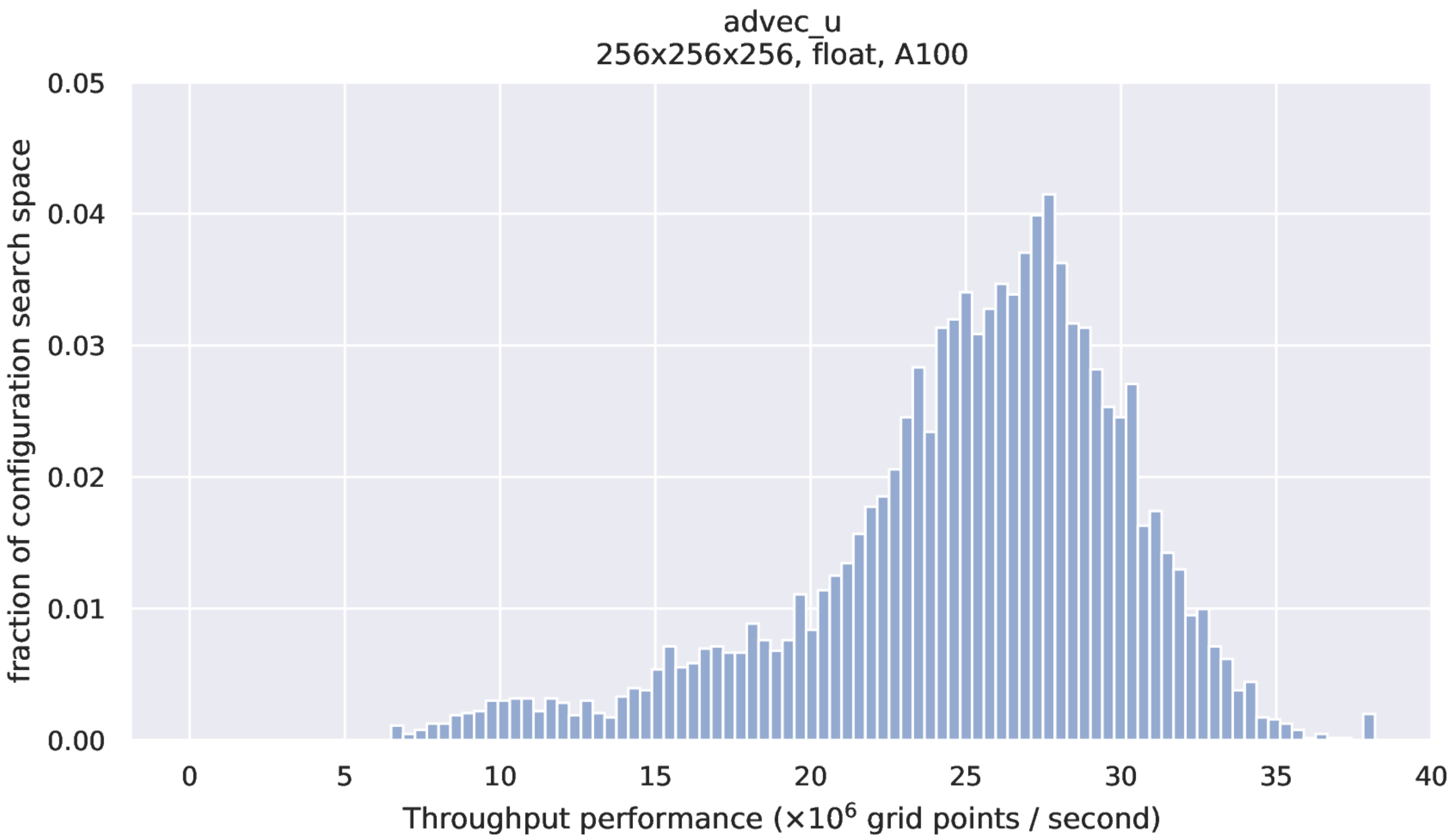
- Per kernel, introduced 14 tunable parameters
  - 7.7M configurations per kernel!
- Highly dynamic code
  - Variable grid size (X × Y × Z)
  - Variable precision (float, double)
  - GPU (Ampere, Volta…)
- Introduced **Kernel Launcher**

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

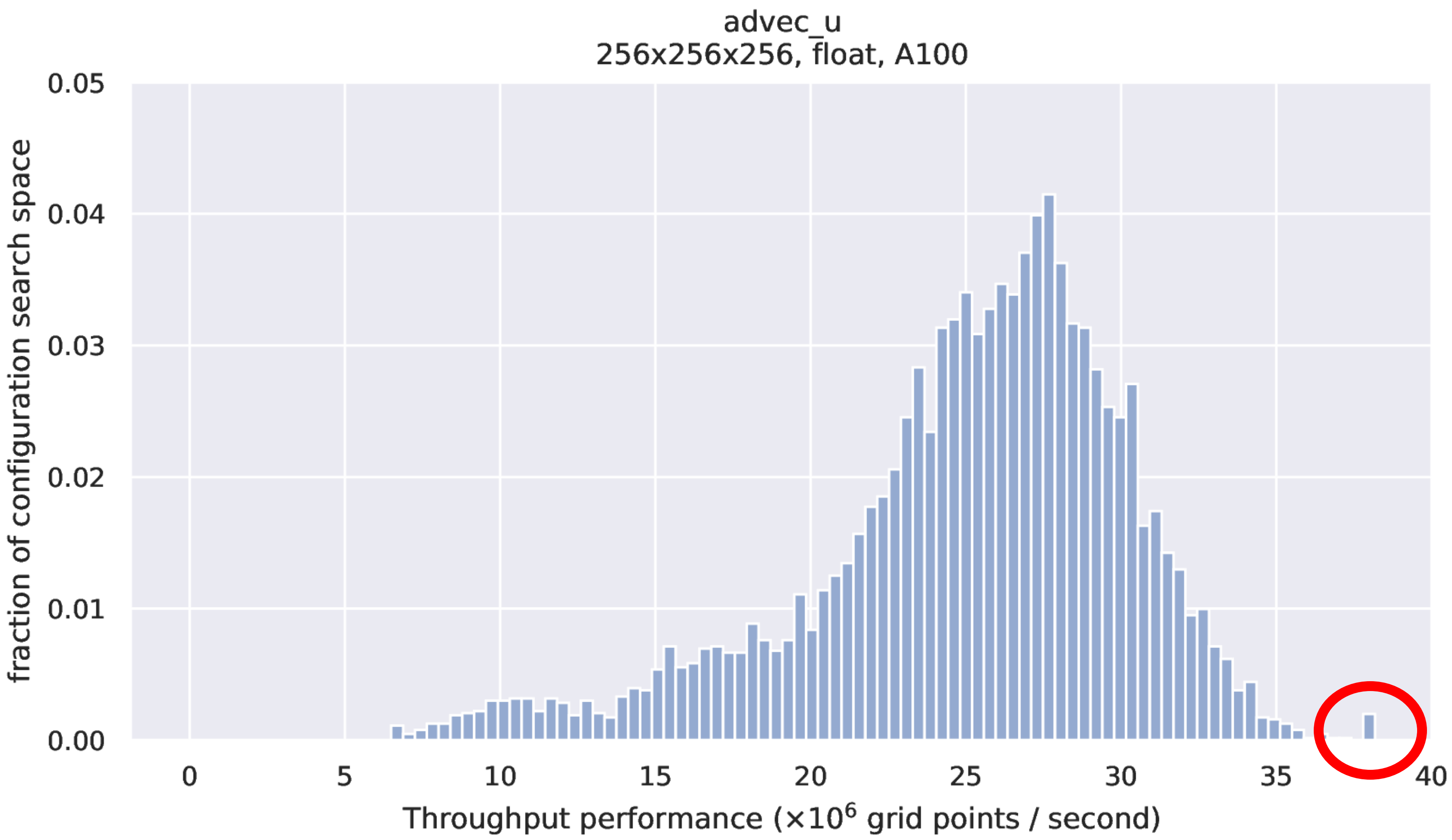
Name	Values	Default value
Block size X	16, 32, 64, 128, 256	256
Block size Y	1, 2, 4, 8, 16	1
Block size Z	1, 2, 4, 8, 16	1
Tile factor X	1, 2, 4	1
Tile factor Y	1, 2, 4	1
Tile factor Z	1, 2, 4	1
Unroll X	`true`, `false`	`false`
Unroll Y	`true`, `false`	`false`
Unroll Z	`true`, `false`	`false`
Tile contiguous 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

netherlands eScience center

MicroHH

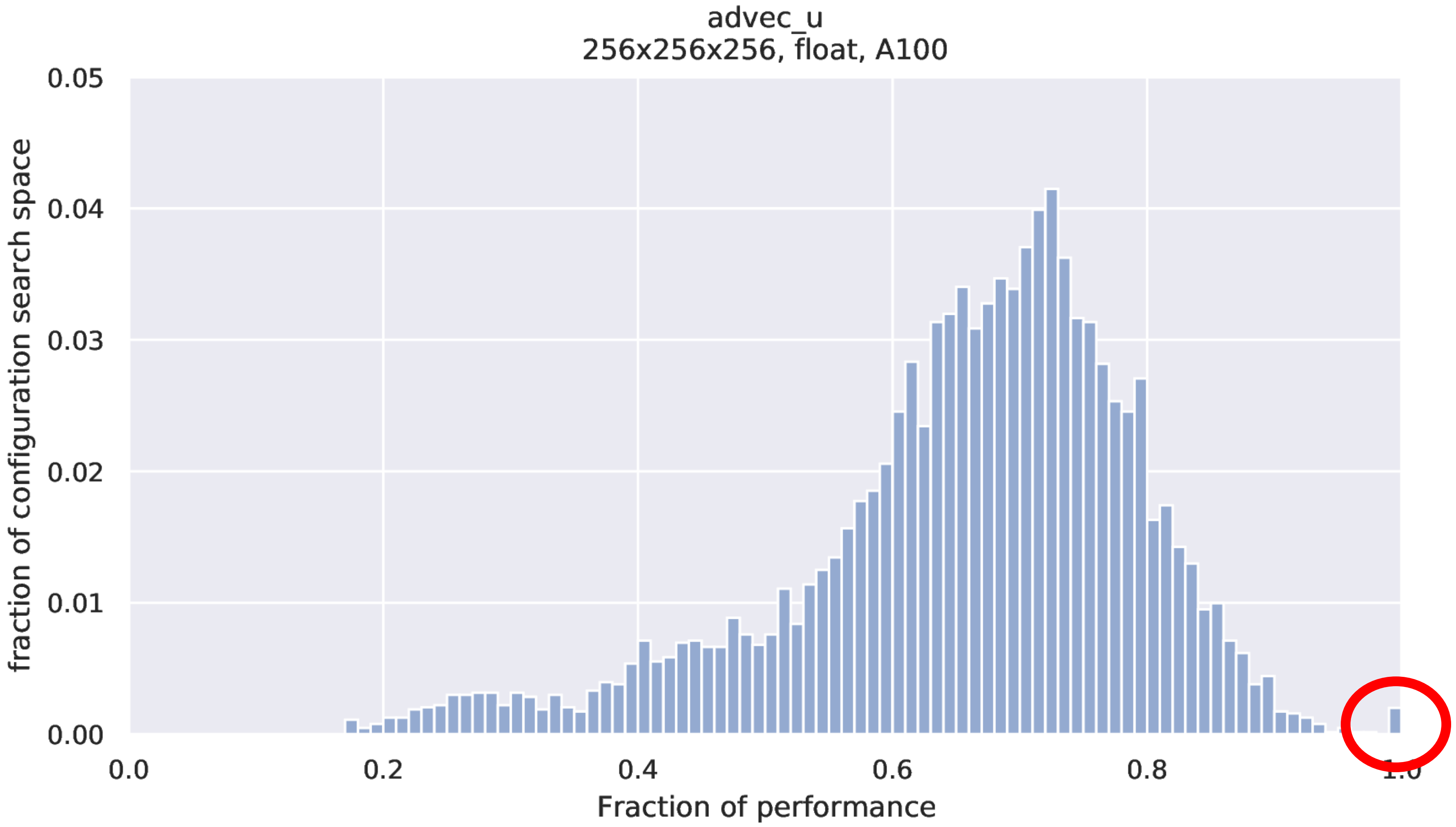


MicroHH



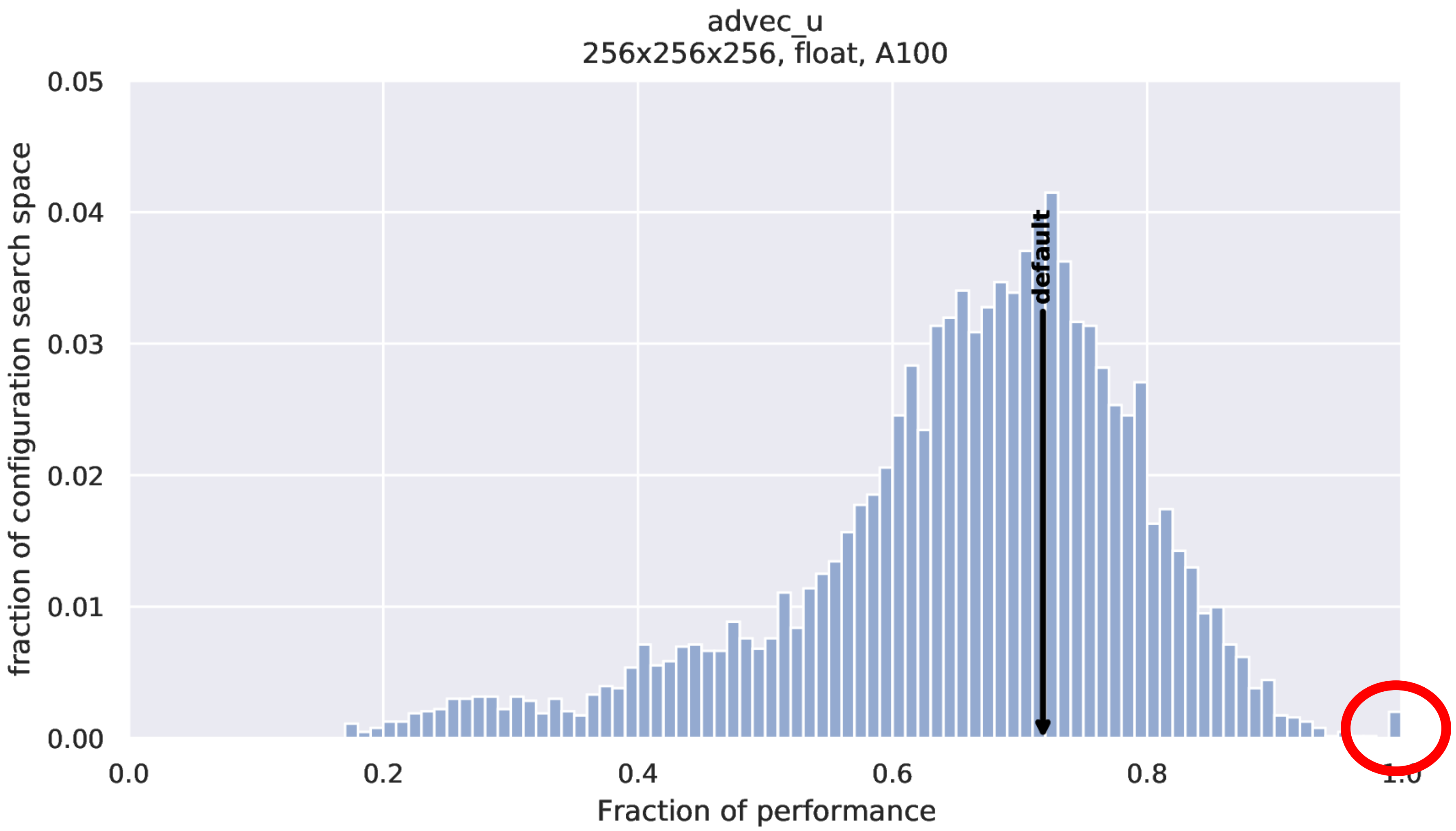
advec_u
256x256x256, float, A100
fraction of configuration search space
0.05
0.04
0.03
0.02
0.01
0.00
0
5
10
15
20
25
30
35
40
Throughput performance ($\times 10^6$ grid points / second)

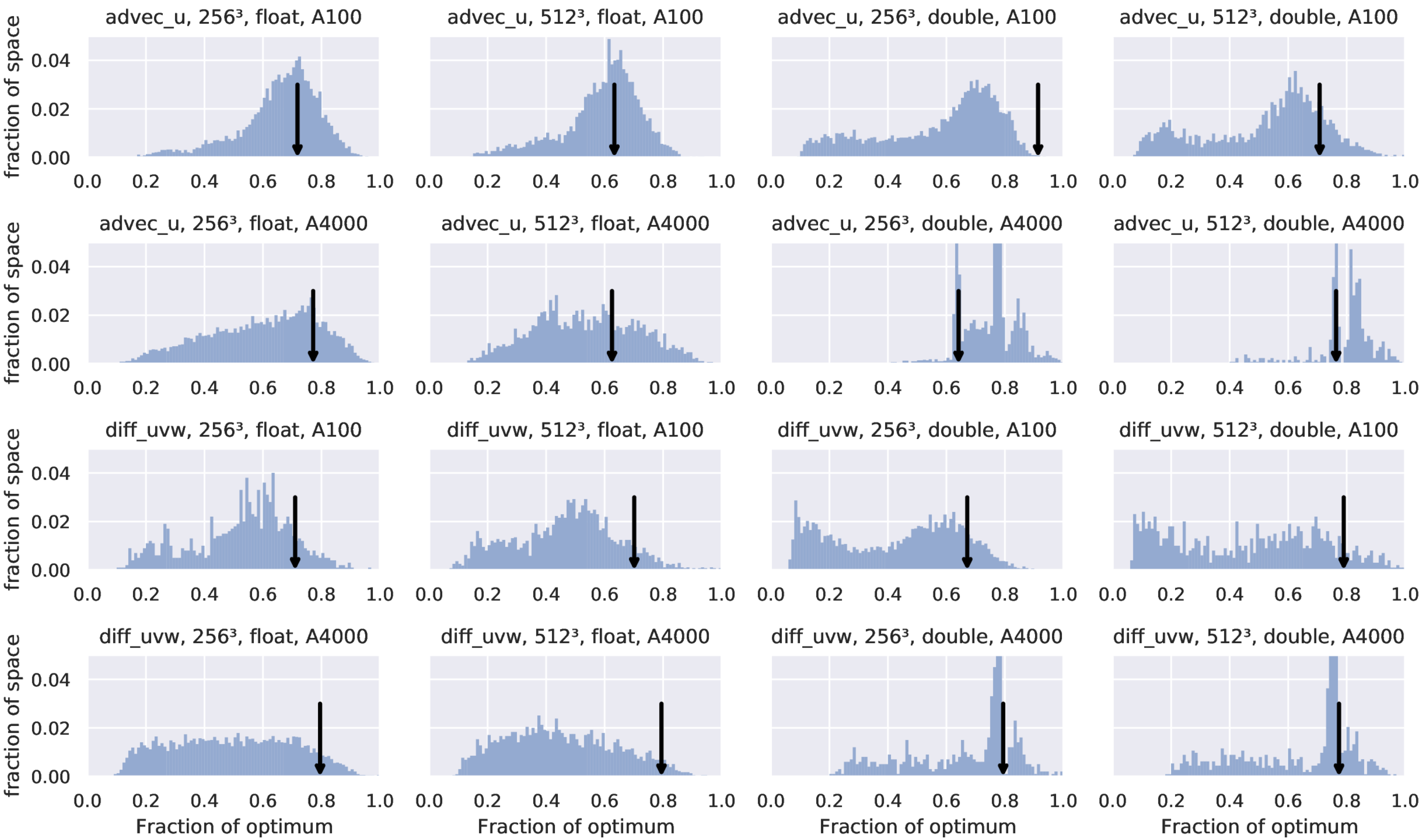
MicroHH

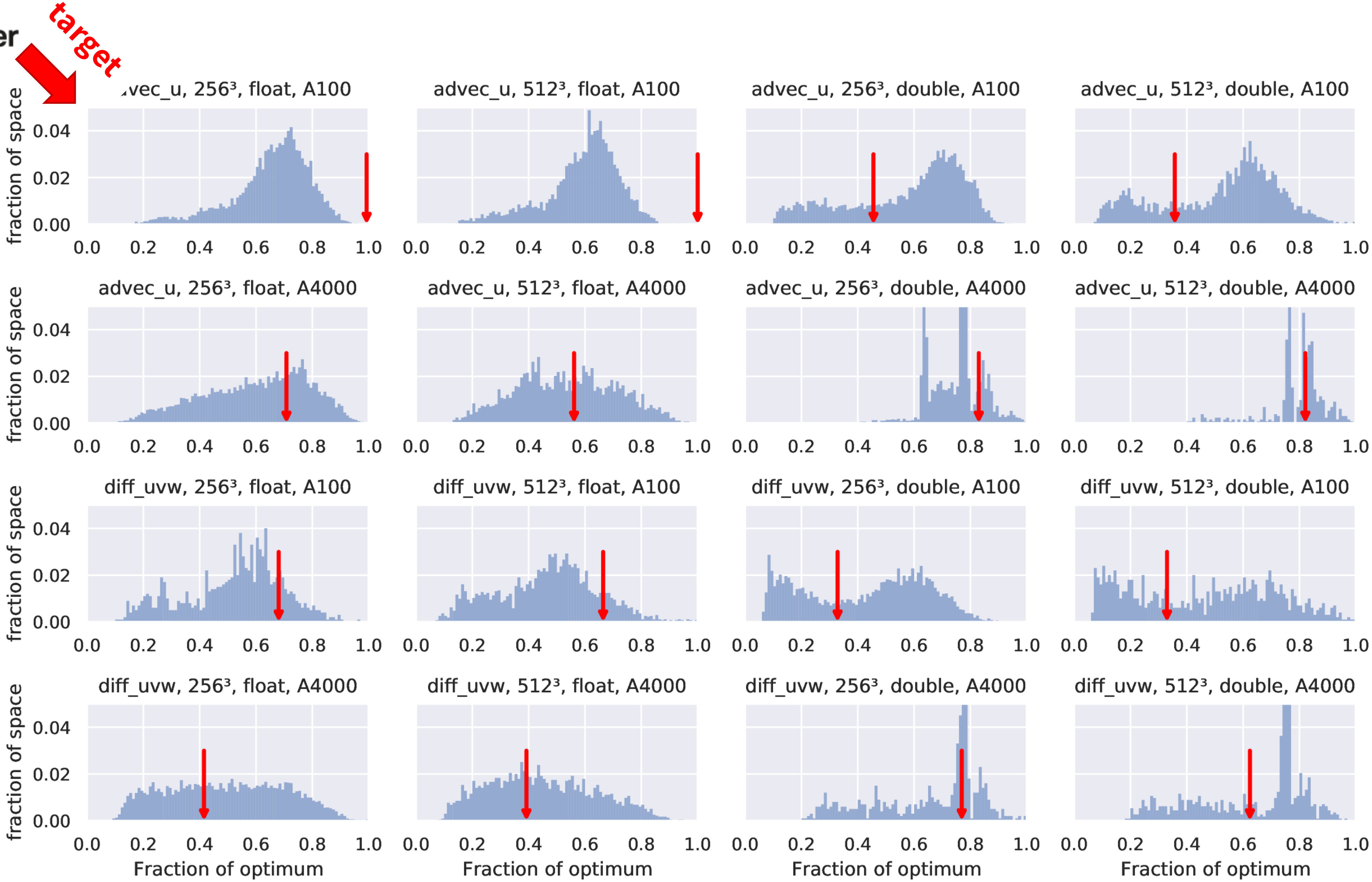

advec_u
256x256x256, float, A100
0.05
0.04
0.03
0.02
0.01
0.00
fraction of configuration search space
0.0
0.2
0.4
0.6
0.8
1.0
Fraction of performance

netherlands eScience center

MicroHH

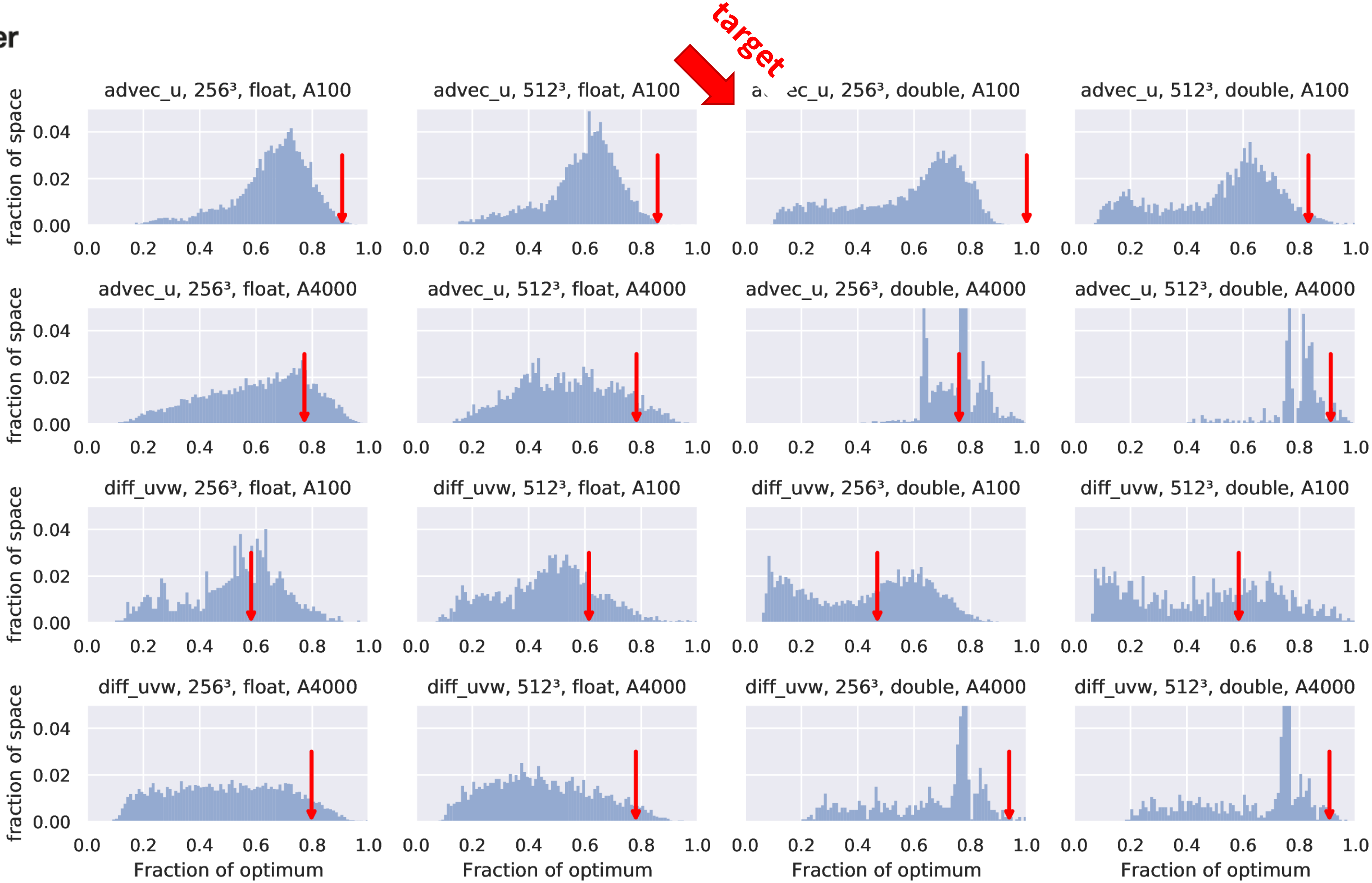


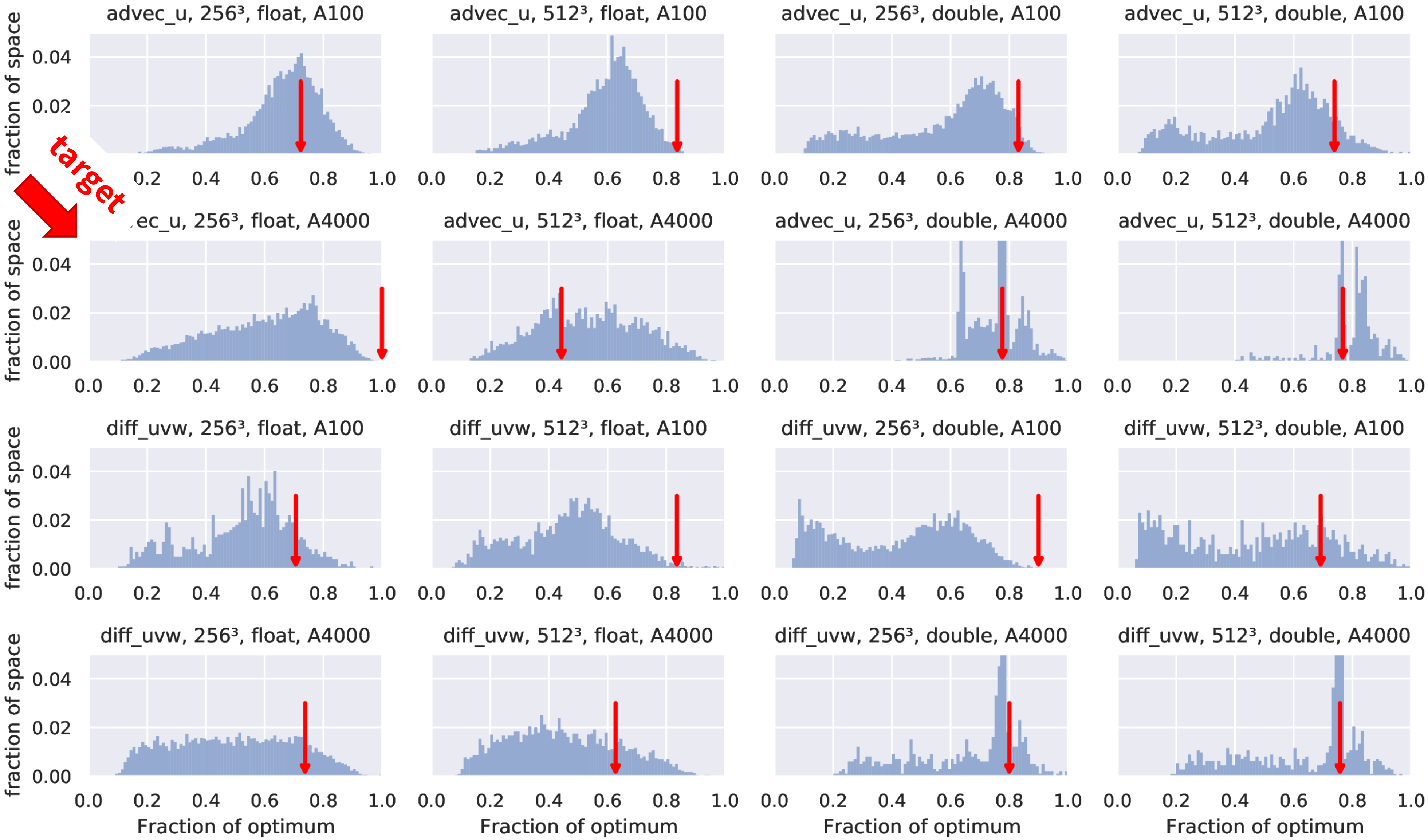

advec_u, 256³, float, A100
advec_u, 512³, float, A100
advec_u, 256³, double, A100
advec_u, 512³, double, A100
advec_u, 256³, float, A4000
advec_u, 512³, float, A4000
advec_u, 256³, double, A4000
advec_u, 512³, double, A4000
diff_uvw, 256³, float, A100
diff_uvw, 512³, float, A100
diff_uvw, 256³, double, A100
diff_uvw, 512³, double, A100
diff_uvw, 256³, float, A4000
diff_uvw, 512³, float, A4000
diff_uvw, 256³, double, A4000
diff_uvw, 512³, double, A4000
fraction of space
Fraction of optimum
0.00
0.02
0.04
0.0
0.2
0.4
0.6
0.8
1.0



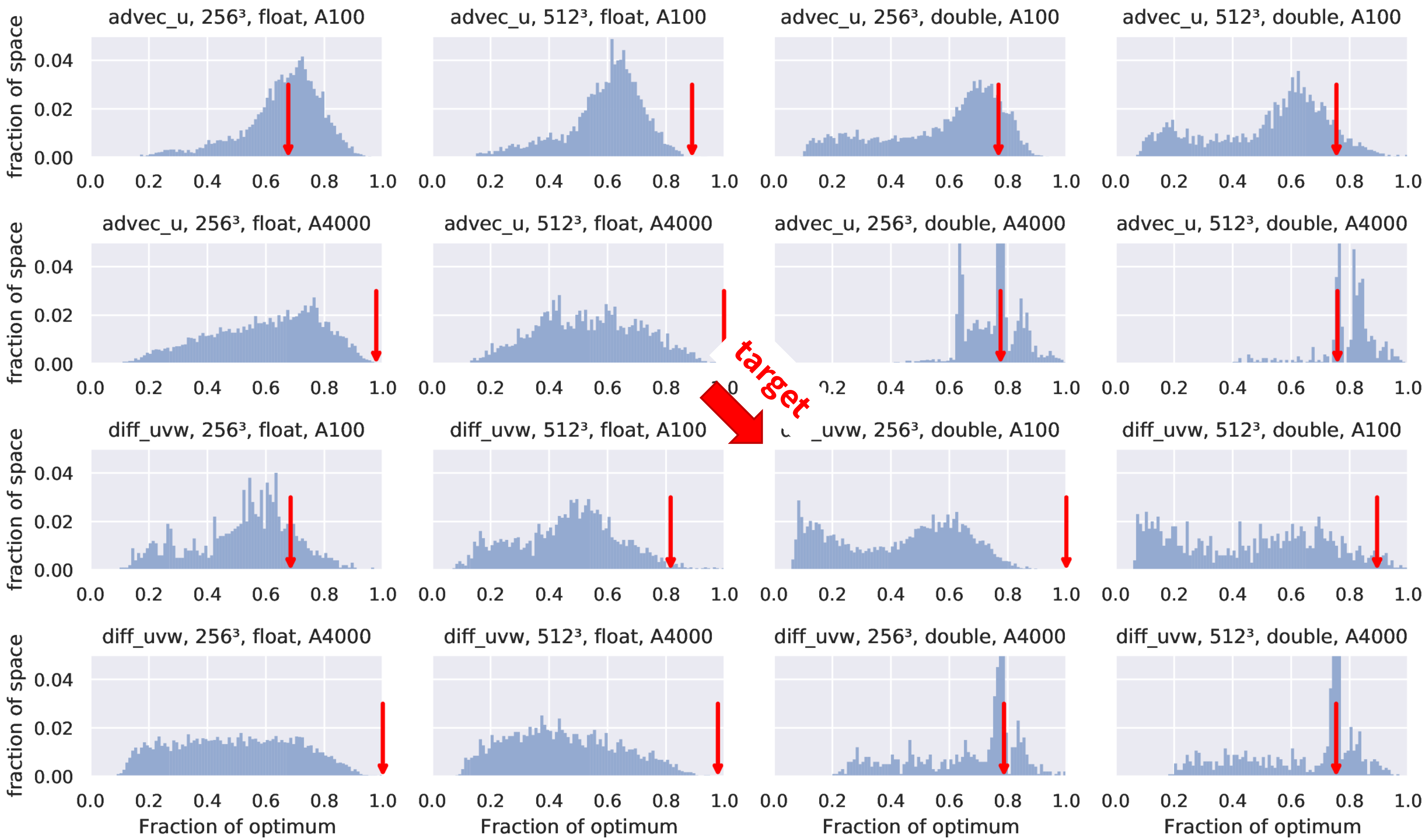
target
advec_u, 256³, float, A100
advec_u, 512³, float, A100
advec_u, 256³, double, A100
advec_u, 512³, double, A100
advec_u, 256³, float, A4000
advec_u, 512³, float, A4000
advec_u, 256³, double, A4000
advec_u, 512³, double, A4000
diff_uvw, 256³, float, A100
diff_uvw, 512³, float, A100
diff_uvw, 256³, double, A100
diff_uvw, 512³, double, A100
diff_uvw, 256³, float, A4000
diff_uvw, 512³, float, A4000
diff_uvw, 256³, double, A4000
diff_uvw, 512³, double, A4000
fraction of space
0.00
0.02
0.04
0.0
0.2
0.4
0.6
0.8
1.0
Fraction of optimum

netherlands eScience center

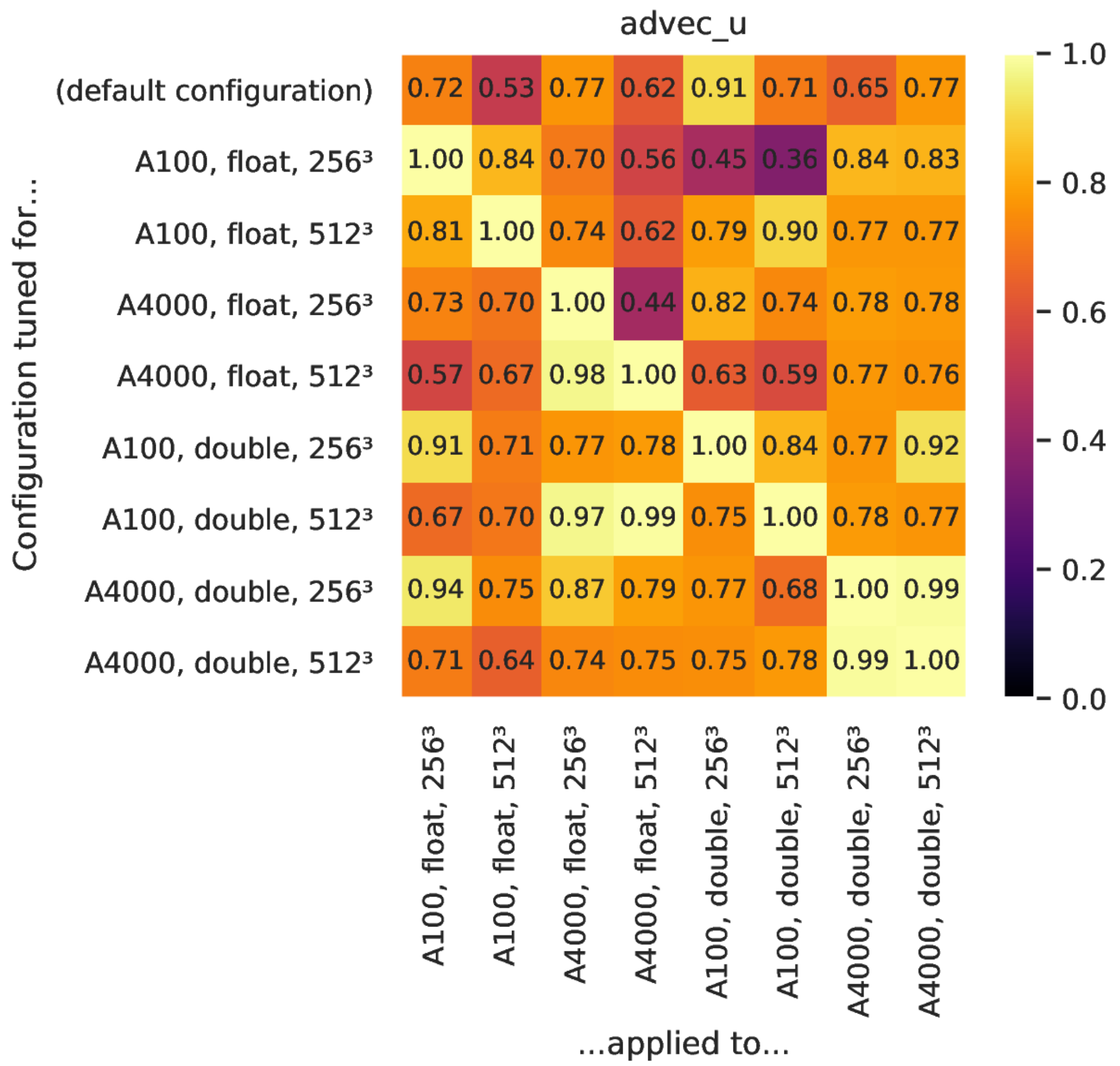


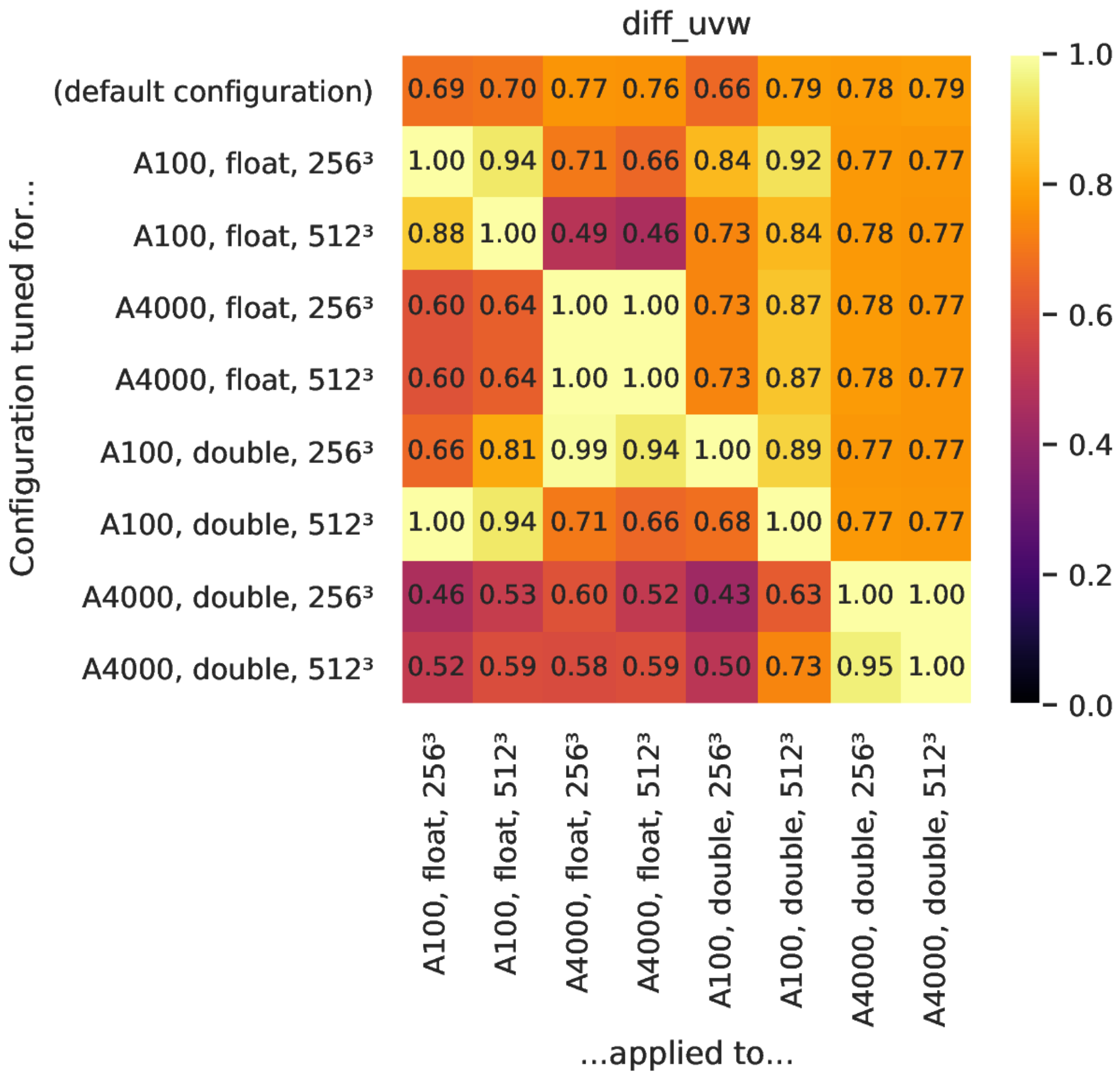


advec_u, 256³, float, A100
advec_u, 512³, float, A100
advec_u, 256³, double, A100
advec_u, 512³, double, A100
advec_u, 256³, float, A4000
advec_u, 512³, float, A4000
advec_u, 256³, double, A4000
advec_u, 512³, double, A4000
diff_uvw, 256³, float, A100
diff_uvw, 512³, float, A100
diff_uvw, 256³, double, A100
diff_uvw, 512³, double, A100
diff_uvw, 256³, float, A4000
diff_uvw, 512³, float, A4000
diff_uvw, 256³, double, A4000
diff_uvw, 512³, double, A4000
target
fraction of space
Fraction of optimum
0.00
0.02
0.04
0.0
0.2
0.4
0.6
0.8
1.0

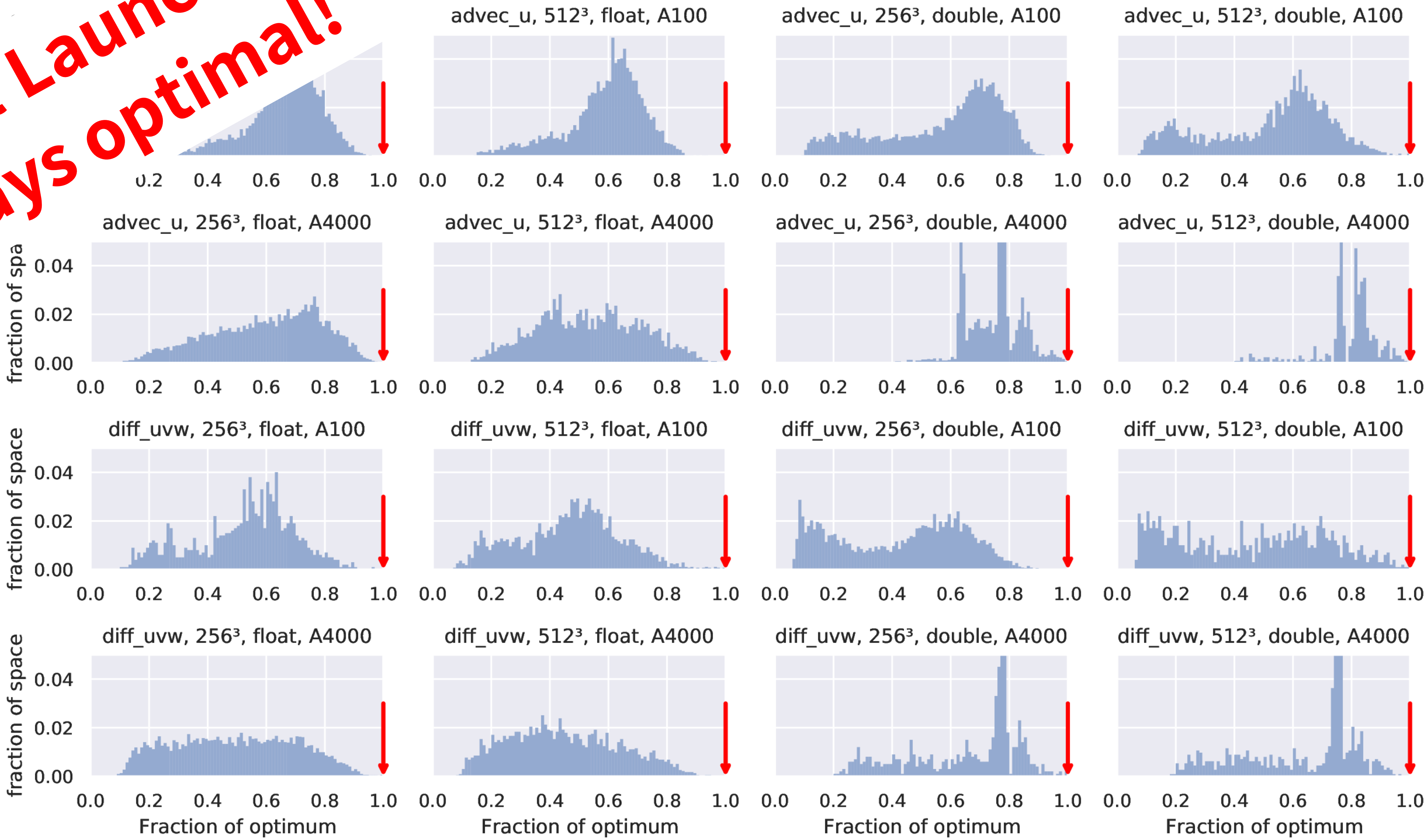

advec_u, 256³, float, A100
advec_u, 512³, float, A100
advec_u, 256³, double, A100
advec_u, 512³, double, A100
advec_u, 256³, float, A4000
advec_u, 512³, float, A4000
advec_u, 256³, double, A4000
advec_u, 512³, double, A4000
diff_uvw, 256³, float, A100
diff_uvw, 512³, float, A100
_uvw, 256³, double, A100
diff_uvw, 512³, double, A100
diff_uvw, 256³, float, A4000
diff_uvw, 512³, float, A4000
diff_uvw, 256³, double, A4000
diff_uvw, 512³, double, A4000
fraction of space
0.00
0.02
0.04
0.0
0.2
0.4
0.6
0.8
1.0
Fraction of optimum
target

Portability





Kernel Launcher: always optimal!



MicroHH

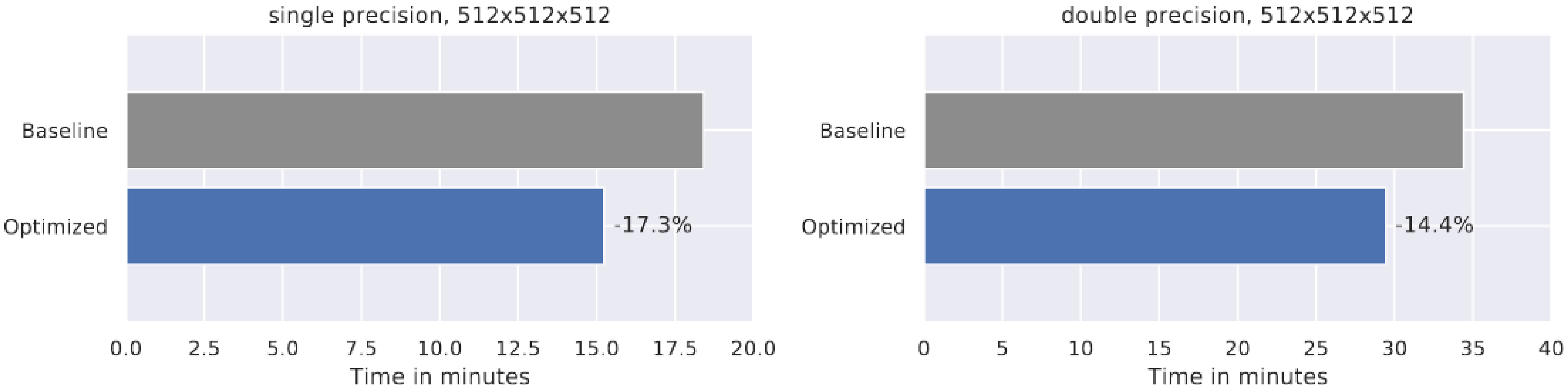
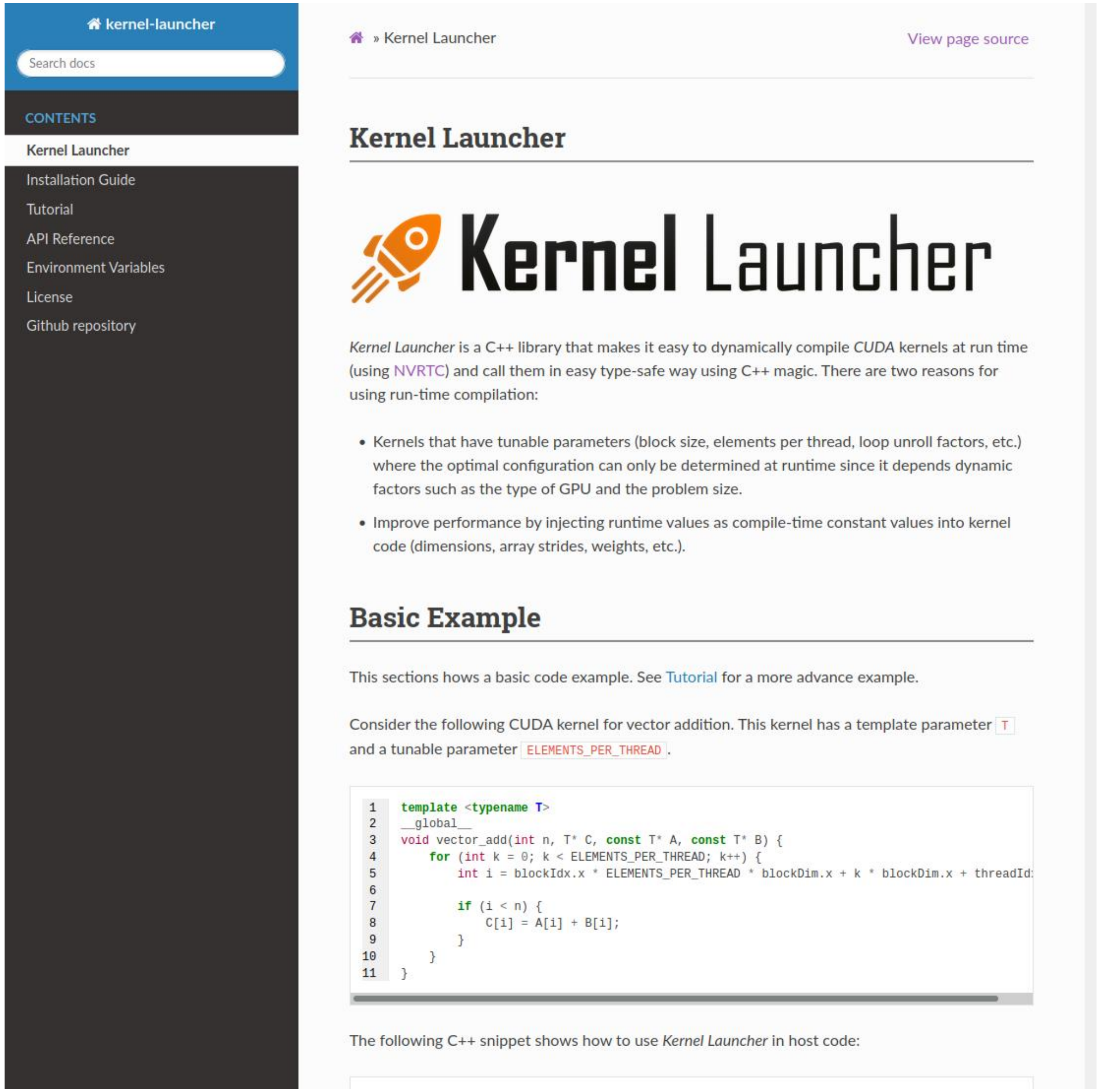


Figure 4: Run-time in minutes of full application for 5000 time iterations.

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

Search docs

CONTENTS

- Kernel Launcher
- Installation Guide
- Tutorial
- API Reference
- Environment Variables
- License
- Github repository

» 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 hows 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`.

```
template <typename T>
__global__
void vector_add(int n, T* C, const T* A, const T* B) {
    for (int k = 0; k < ELEMENTS_PER_THREAD; k++) {
        int i = blockIdx.x * ELEMENTS_PER_THREAD * blockDim.x + k * blockDim.x + threadId

        if (i < n) {
            C[i] = A[i] + B[i];
        }
    }
}
```

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).