GPU code optimization and automatic performance tuning made easy with Kernel Tuner

Alessio Sclocco, Stijn Heldens, Floris-Jan Willemsen, Ben van Werkhoven

netherlands eScience center

Administrative announcements

- We will have a presentation to introduce auto-tuning and Kernel Tuner, followed by a hands-on session to experiment with what we learned today
- The hands-on session include an example kernel, but you are also welcome to experiment with your own GPU code
- We will use Google Colab for the hands-on session, so you don't need to have access to a GPU or install anything locally
- You can download the slides and the hands-on notebook here:
  - https://github.com/KernelTuner/kernel_tuner_tutorial

netherlands eScience center

Overview of auto-tuning technologies

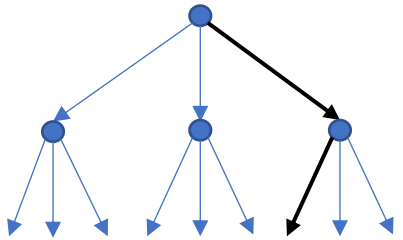
To maximize GPU code performance, you need to find the best combination of:

- Different mappings of the problem to threads and thread blocks
- Different data layouts in different memories (shared, constant, …)
- Different ways of exploiting special hardware features
- Thread block dimensions
- Code optimizations that may be applied or not
- Work per thread in each dimension
- Loop unrolling factors
- Overlapping computation and communication
- …

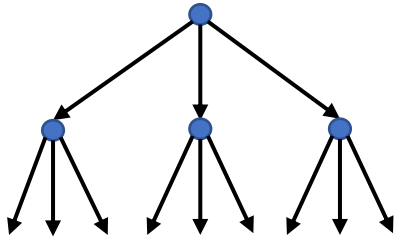
Problem:

- Creates a very large search space

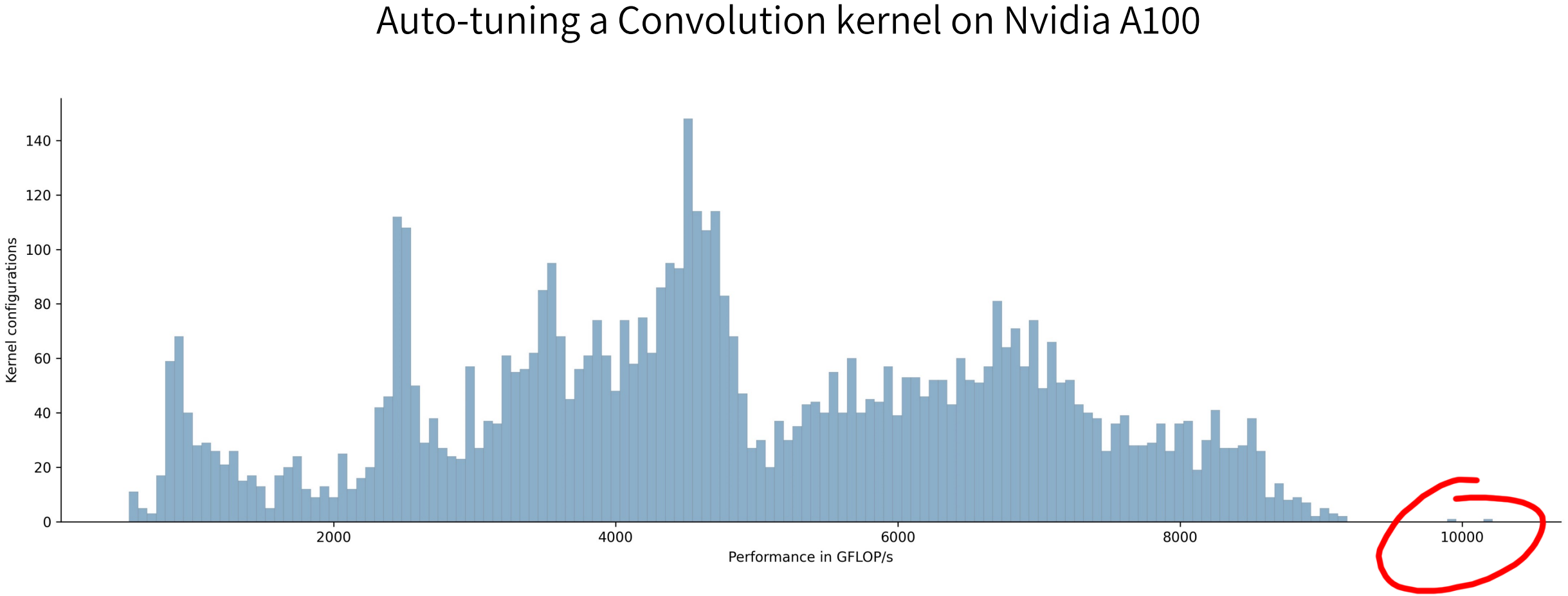
- Optimizing code manually you iteratively:
  - Modify the code
  - Run a few benchmarks
  - Revert or accept the change



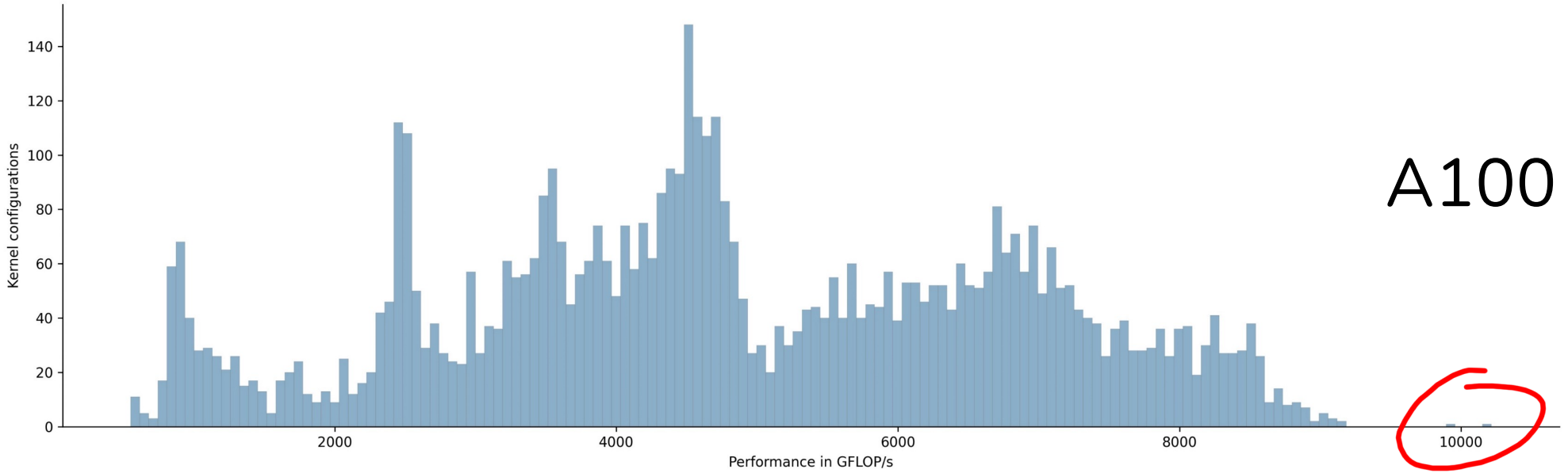
- With auto-tuning you:
  - Write a templated version of your code or a code generator
  - Benchmark the performance of all code variants



Large search space of kernel configurations



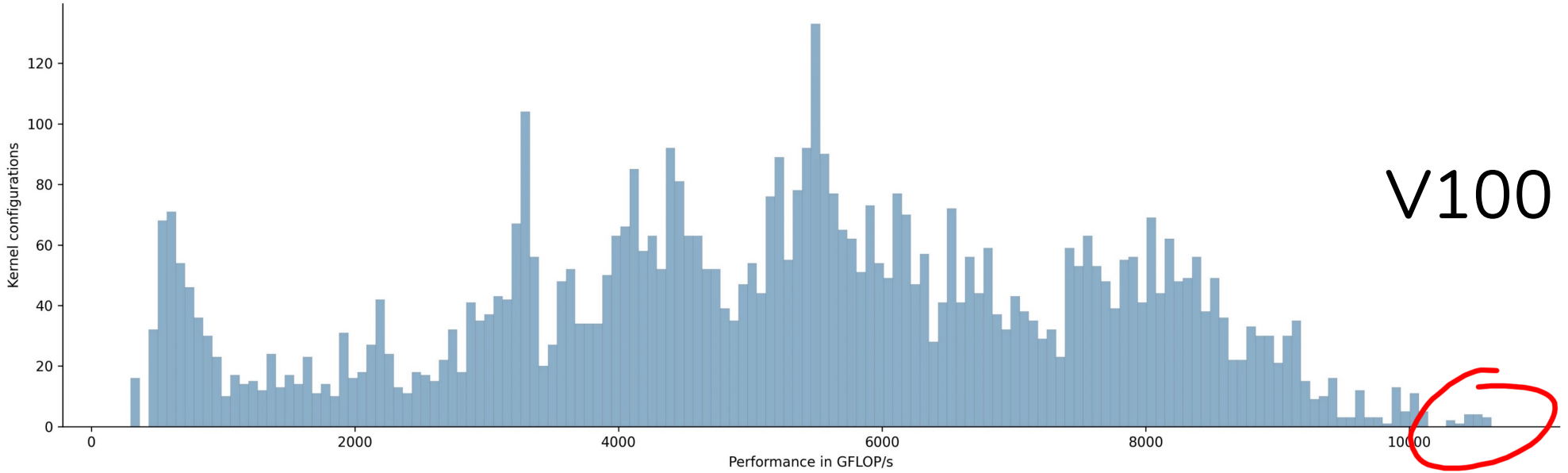
On different GPUs ...



A100

top 5% on A100

8,16,2,2,1,0
8,32,2,1,1,0



V100

top 5% on V100

4,32,5,1,1,0
4,32,5,1,1,1
32,4,1,4,0,0
32,4,1,4,0,1
32,4,1,4,1,0
32,4,1,4,1,1
64,2,1,4,0,0
64,2,1,4,1,0
64,4,1,4,0,0
64,4,1,4,0,1
64,4,1,4,1,0
64,4,1,4,1,1
128,2,1,8,0,0
128,2,1,8,1,0

- Compiler auto-tuning
  - Does not modify the source code
  - Needs to work with assumptions the compiler can take
  - Usually tunes compiler flags for the application as a whole

- Software auto-tuning
  - Works with a code template or code generator
  - Can tune more diverse code optimizations and even competing implementations of the same algorithm
  - Usually tunes individual kernels or pipelines of kernels

- Model-based auto-tuning
  - Requires a model or oracle to predict the performance of different kernel configurations on the target hardware
  - Typically used for relatively simple, regular, and well-understood kernels, such as stencils or Fourier transforms
  - Models are difficult to construct and generalize to any kernel or any hardware
- Empirical auto-tuning
  - Uses benchmarking to determine performance of different configurations
  - Easier to implement, but requires access to the hardware, compilation, and benchmarking
  - Optimization strategies are used to speedup tuning process

- Run-time auto-tuning
  - Performs benchmarking while the application is running
    - May slow down the application when tuning takes place
  - Host application needs substantial modification
  - Good when performance strongly depends on input data
  - Performance measurements may not be reliable
- Compile-time auto-tuning
  - Allows to separate tuning code from the host application
  - Host application can be in any programming language and is usually not modified to support tuned kernels
  - Requires kernels to be compiled in isolation
  - Good for working on kernels with big optimization spaces in isolation

- Advantages of auto-tuning over manual optimization are:
  - Allows to see past local minima and unexpected interactions between different code optimizations
  - Simply rerun the tuner for different hardware
- Software auto-tuning is more flexible and powerful than only tuning compiler flags
- Tuning using performance models is much more efficient, but difficult to achieve
- Use run-time tuning for when the input data strongly influences kernel performance or when kernels are difficult to separate from host application
- Use compile-time tuning for heavily-optimized kernels that can be compiled and benchmarked in isolation
- Kernel Tuner is an **empirical, compile-time, software auto-tuner**

netherlands eScience center

Introduction to Kernel Tuner

Kernel Tuner

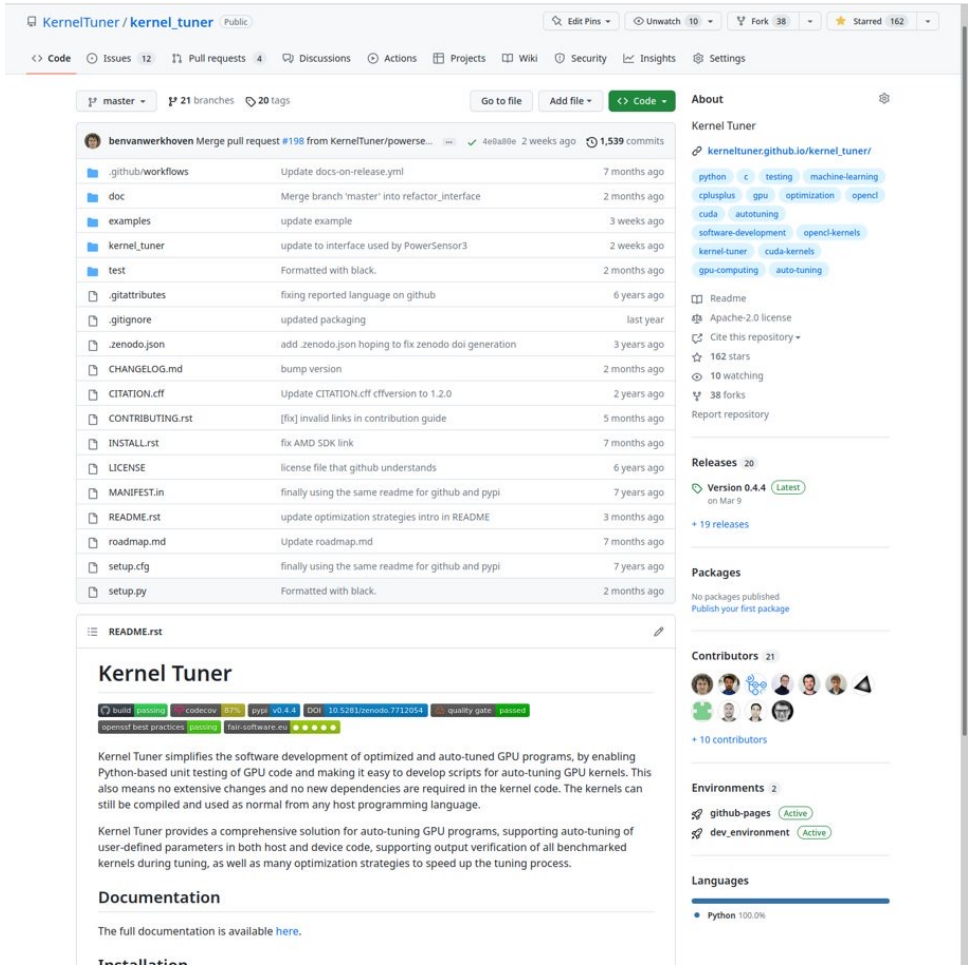
Kernel Tuner is a Python tool for tuning and testing GPU kernels

Easy to use:

- Usable with existing kernels and code generators
- No dependencies required in kernels or host application
- Kernels can still be compiled with regular compilers

Supports:

- Tuning code in OpenCL, CUDA, C, or Fortran
- Large number of effective search strategies
- Output verification for each benchmarked kernels
- Tuning parameters in both host and device code
- Allows unit testing GPU code from Python
- …



https://github.com/KernelTuner/kernel_tuner

Kernel Tuner Developers

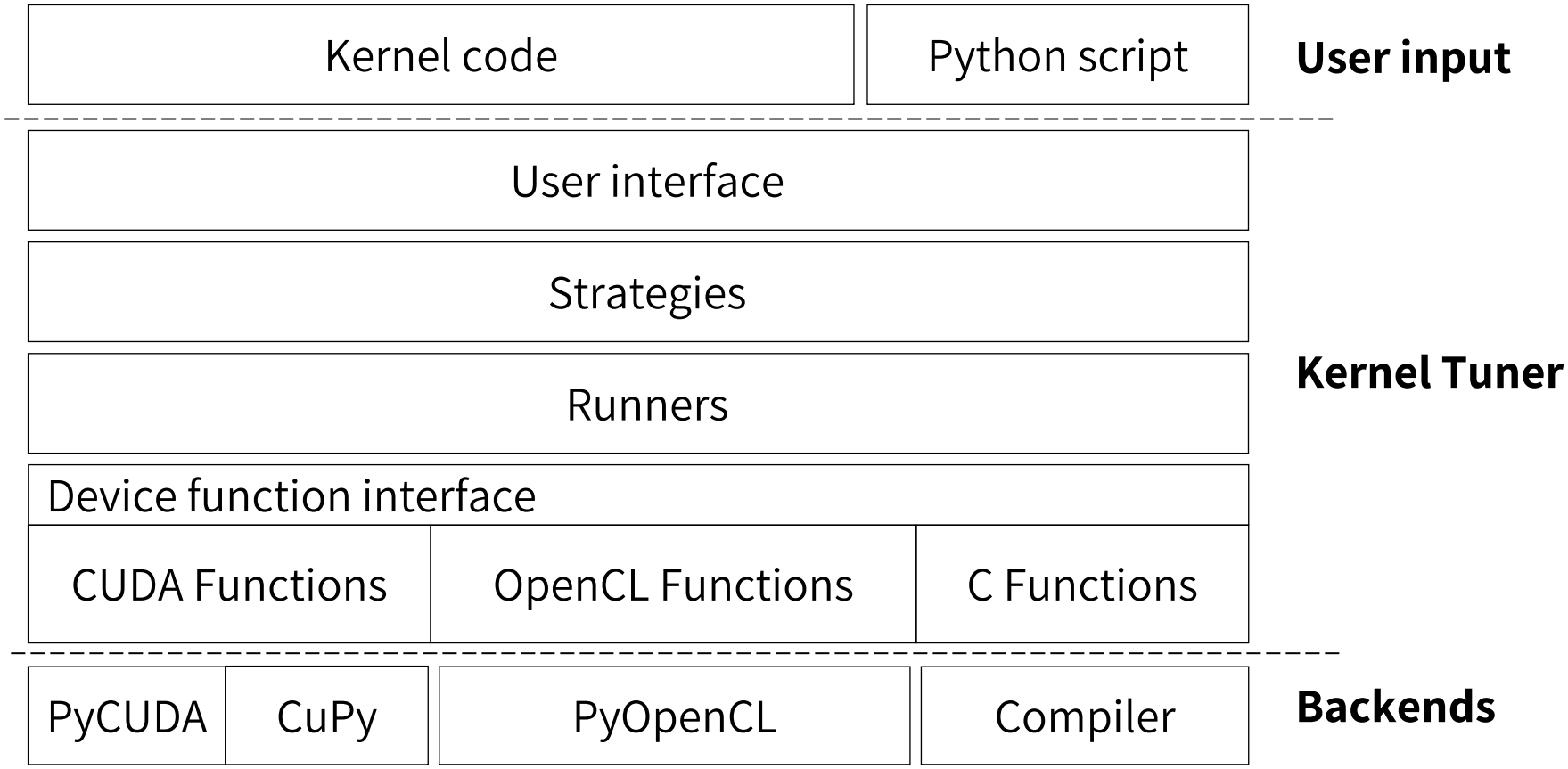
- Alessio Sclocco (eScience Center)
- Stijn Heldens (eScience center)
- Floris-Jan Willemsen (eScience Center)
- Richard Schoonhoven (CWI)
- Willem Jan Palenstijn (Leiden University)
- Bram Veenboer (Astron)
- Ben van Werkhoven (Leiden University)

- And many more contributors, see GitHub:
  - https://github.com/KernelTuner/kernel_tuner

Where does Kernel Tuner fit in the auto-tuning taxonomy?

- Kernel Tuner is:
  - **Empirical**
    - optimization methods may use models internally
  - **Compile-time**
    - or preferably even development-time
  - **Software-based**
  - Easy to **integrate** into existing applications
  - Capable of tuning **discrete** parameters in kernel code

Kernel Tuner architecture



Minimal example of tuning a CUDA kernel

```
import numpy
from kernel_tuner import tune_kernel

kernel_string = """
__global__ void vector_add(float *c, float *a, float *b, int n) {
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        c[i] = a[i] + b[i];
    }
}"""

n = numpy.int32(1e7)
a = numpy.random.randn(n).astype(numpy.float32)
b = numpy.random.randn(n).astype(numpy.float32)
c = numpy.zeros_like(b)
args = [c, a, b, n]
tune_params = {"block_size_x": [32, 64, 128, 256, 512]}

tune_kernel("vector_add", kernel_string, n, args, tune_params)
```

- Create the search space:
  - Computed as the Cartesian product of all values of all tunable parameters
  - Remove configurations that fail any of the restrictions

- Use optimization algorithm to select configurations:
  - For each selected configuration:
    - Insert preprocessor definitions for each tuning parameter
    - Compile the kernel created for this instance
    - Benchmark the kernel
    - Store the averaged execution time

- Return the full data set

Kernel Tuner compiles and benchmarks many kernel configurations

- We need to tell Kernel Tuner how to compile and run our kernel:
  - This includes source code and compiler options
  - Easier if your kernel code can be compiled separately, so without including many other files
- Kernel Tuner is written in Python:
  - We need to load/create the kernel's input/output data in Python

`kernel_tuner.tune_kernel`(*kernel_name, kernel_source, problem_size, arguments, tune_params, grid_div_x=None, grid_div_y=None, grid_div_z=None, restrictions=None, answer=None, atol=1e-06, verify=None, verbose=False, lang=None, device=0, platform=0, smem_args=None, cmem_args=None, texmem_args=None, compiler=None, compiler_options=None, log=None, iterations=7, block_size_names=None, quiet=False, strategy=None, strategy_options=None, cache=None, metrics=None, simulation_mode=False, observers=None*)

Tune a CUDA kernel given a set of tunable parameters

Parameters:

- **kernel_name** (*string*) – The name of the kernel in the code.
- **kernel_source** (*string or list and/or callable*) –
  The CUDA, OpenCL, or C kernel code. It is allowed for the code to be passed as a string, a filename, a function that returns a string of code, or a list when the code needs auxilliary files.
  To support combined host and device code tuning, a list of filenames can be passed. The first file in the list should be the file that contains the host code. The host code is assumed to include or read in any of the files in the list beyond the first. The tunable parameters can be used within all files.
  Another alternative is to pass a code generating function. The purpose of this is to support the use of code generating functions that generate the kernel code based on the specific parameters. This function should take one positional argument, which will be used to pass a dict containing the parameters. The function should return a string with the source code for the kernel.

Kernel Tuner automatically allocates GPU memory and moves data in and out of the GPU

Supports the following types for kernel arguments:

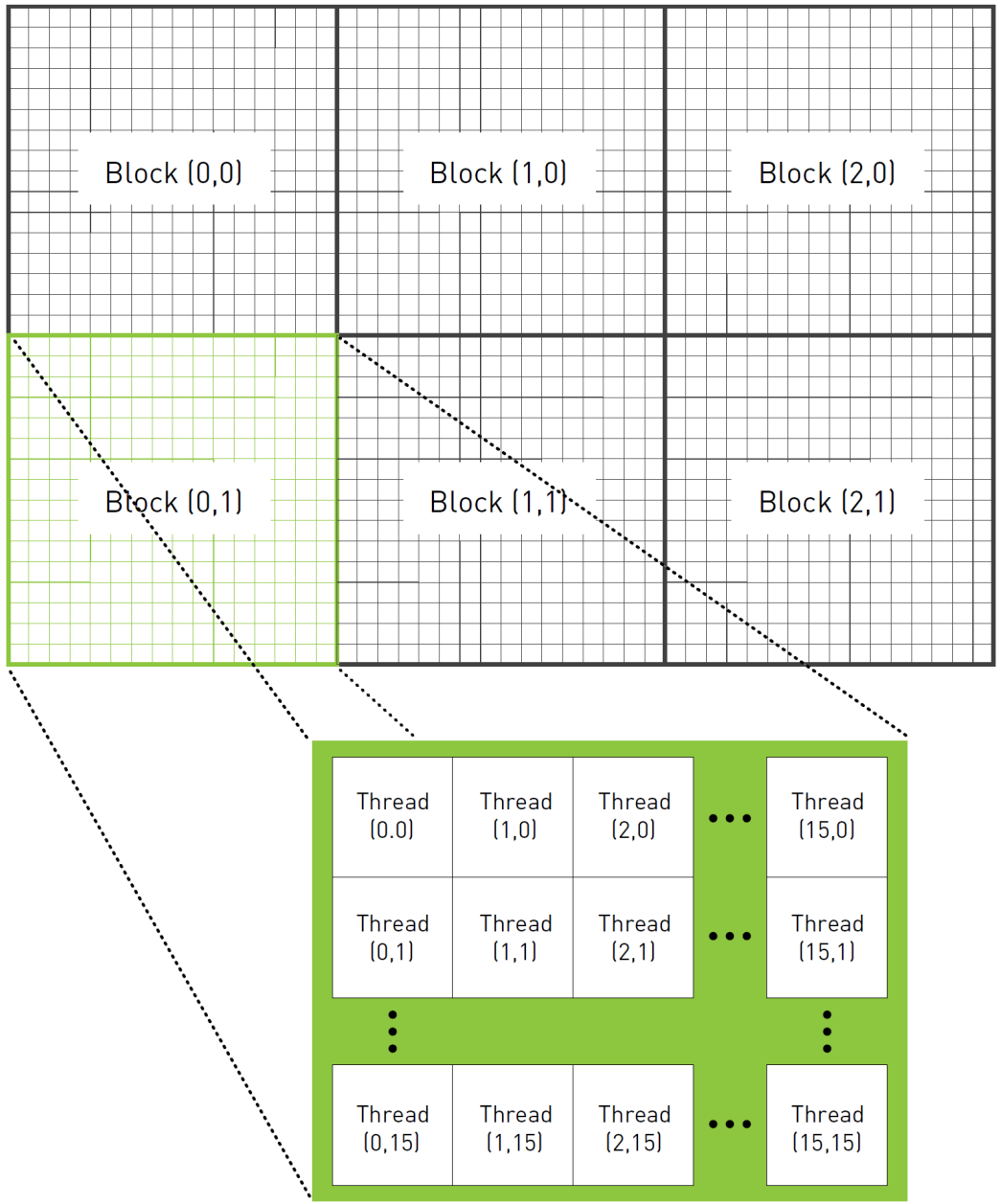
- NumPy scalars (`np.int32`, `np.float32`, ...)
- NumPy ndarrays
- CuPy arrays
- Torch tensors

- Prerequisites:
  - Python 3.8 or newer
  - CUDA or OpenCL device with necessary drivers and compilers installed
  - PyCUDA, PyOpenCL, or CuPy installed
- To install Kernel Tuner:
  - pip install kernel_tuner
- For more information:
  - https://kerneltuner.github.io/kernel_tuner/stable/install.html
- **Note**: this is not required for the hands-on session

netherlands eScience center

Grid and Thread block dimensions

CUDA Thread Hierarchy



- Almost all GPU kernels can be written in a form that allows for varying thread block dimensions
- Changing thread block dimensions affects performance, but not the result
- The question is, how to determine the optimal setting?

- In Kernel Tuner, you specify the possible values for thread block dimensions of your kernel using special tunable parameters:
  - `block_size_x, block_size_y, block_size_z`
- For each, you may pass a list of values this parameter can take:
  - `tune_params["block_size_x"] = [32, 64, 128, 256]`

- You can use different names for these by passing the `block_size_names` option using a list of strings
- Note: when you change the thread block dimensions, the number of thread blocks used to launch the kernel generally changes as well

Access thread block dimensions at compile-time

- Kernel Tuner automatically inserts a block of `#define` statements to set values for `block_size_x`, `block_size_y`, and `block_size_z`
- You can use these values in your code to access the thread block dimensions as compile-time constants
- This is generally a good idea for performance, because
  - Loop conditions may use the thread block dimensions, fixing the number of iterations at compile-time enables the compiler to unroll the loop and optimize the code
  - Shared memory declarations can use the thread block dimensions, e.g. for compile-time sizes multi-dimensional data

Vector add example

```
import numpy
import kernel_tuner

kernel_string = """
__global__ void vector_add(float *c, float *a, float *b, int n) {
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        c[i] = a[i] + b[i];
    }
}"""

n = numpy.int32(1e7)
a = numpy.random.randn(n).astype(numpy.float32)
b = numpy.random.randn(n).astype(numpy.float32)
c = numpy.zeros_like(b)
args = [c, a, b, n]
tune_params = {"block_size_x": [32, 64, 128, 256, 512]}

kernel_tuner.tune_kernel("vector_add", kernel_string, n, args, tune_params)
```

Here we specify the tunable values that kernel tuner should try for **`block_size_x`**

Vector add example

```
import numpy
import kernel_tuner

kernel_string = """
__global__ void vector_add(float *c, float *a, float *b, int n) {
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        c[i] = a[i] + b[i];
    }
}"""

n = numpy.int32(1e7)
a = numpy.random.randn(n).astype(numpy.float32)
b = numpy.random.randn(n).astype(numpy.float32)
c = numpy.zeros_like(b)
args = [c, a, b, n]
tune_params = {"block_size_x": [32, 64, 128, 256, 512]}

kernel_tuner.tune_kernel("vector_add", kernel_string, n, args, tune_params)
```

Notice how we can use `block_size_x` in our `vector_add` kernel code, while it is actually not defined (yet)

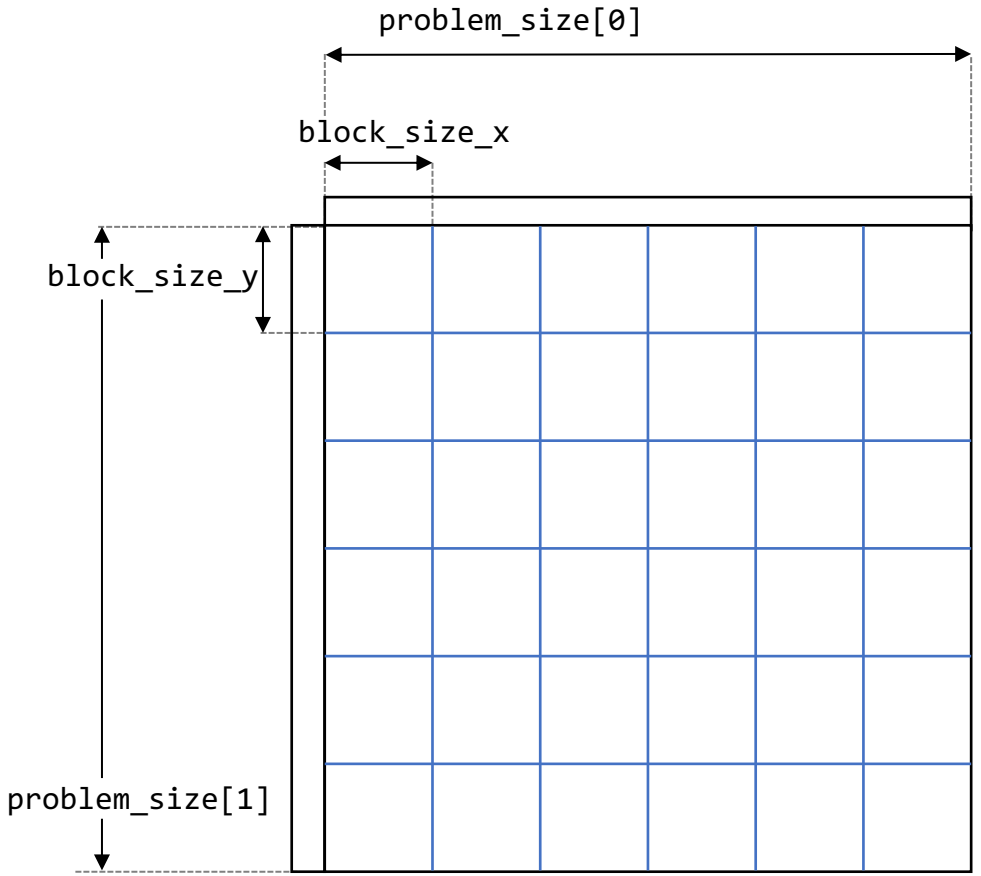
Vector add example

```
import numpy
import kernel_tuner

kernel_string = """
__global__ void vector_add(float *c, float *a, float *b, int n) {
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        c[i] = a[i] + b[i];
    }
}"""

n = numpy.int32(1e7)
a = numpy.random.randn(n).astype(numpy.float32)
b = numpy.random.randn(n).astype(numpy.float32)
c = numpy.zeros_like(b)
args = [c, a, b, n]
tune_params = {"block_size_x": [32, 64, 128, 256, 512]}

kernel_tuner.tune_kernel("vector_add", kernel_string, n, args, tune_params)
```

n is the number of array elements, the number of thread blocks depends on both **n** and **block_size_x**

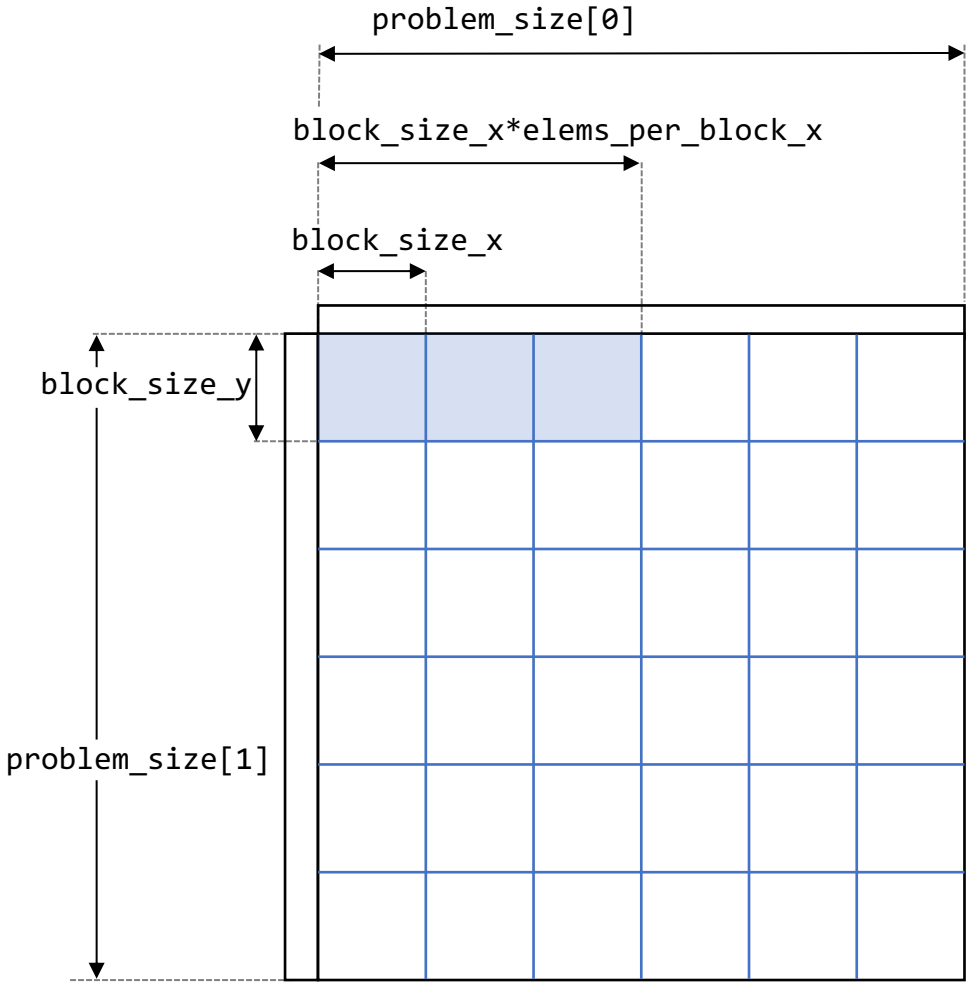
Specifying grid dimensions

- In Kernel Tuner, you specify the `problem_size`
- `problem_size` describes the dimensions across which threads are created
- By default, the grid dimensions are computed as:
  - `grid_size_x = ceil(problem_size_x / block_size_x)`
- `Example:`
  - `problem_size_x = 90`
  - `block_size_x = 16`
  - `grid_size_x = ceil(90 / 16) = ceil(5.625) = 6`



Grid divisor lists

- Other parameters, or none at all, may also affect the grid dimensions
- Grid divisor lists control how `problem_size` is divided to compute the grid size
- Use the optional arguments:
  - `grid_div_x`, `grid_div_y`, and `grid_div_z`
- You may disable this feature by explicitly passing empty lists as grid divisors, in which case `problem_size` directly sets the grid dimensions



- The `problem_size` is usually a single integer or a tuple of integers
- May also be a (tuple of) tunable parameter(s)
- May also be a (lambda) function that takes a dictionary of tunable parameters and returns a tuple of integers
- For example, `reduction.py`:

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
tune_params["num_blocks"] = [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]
problem_size = "num_blocks" # alternative: `lambda p: p["num_blocks"]`
grid_div_x = []
```

netherlands eScience center

Search space restrictions

- By default, the search space is the Cartesian product of all possible combinations of tunable parameter values
- Example:

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512]
tune_params["block_size_y"] = [1, 2, 3, 4, 5, 6, 7, 8]
tune_params["block_size_z"] = [1, 2, 3, 4, 5, 6, 7, 8]
```

- However, for some tunable kernels:
  - there are tunable parameters that depend on each other
  - only certain combinations of tunable parameter values are valid

- For example:
  - Parameter controls the block size: `block_size_{x, y, z}`
  - But the total threads per block cannot exceed a maximum
- By default, Kernel Tuner considers search space to be the Cartesian product of all possible combinations of all values for all parameters
- What if we only want configurations in which the number of threads per block does not exceed 512?

Restriction example

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512]
tune_params["block_size_y"] = [1, 2, 3, 4, 5, 6, 7, 8]
tune_params["block_size_z"] = [1, 2, 3, 4, 5, 6, 7, 8]

# define a lambda function that returns True when a configuration is valid
restrict = lambda p: p["block_size_x"] * p["block_size_y"] * \
                     p["block_size_z"] <= 512

# pass our lambda function to the restrictions option
kernel_tuner.tune_kernel(..., restrictions=restrict, ...)
```

Partial loop unrolling example

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512]
tune_params["block_size_y"] = [1, 2, 3, 4, 5, 6, 7, 8]
tune_params["block_size_z"] = [1, 2, 3, 4, 5, 6, 7, 8]

# or: define a list of expressions that returns True when a configuration is valid
restrict = ["block_size_x * block_size_y * block_size_z <= 512"]


# pass our lambda function to the restrictions option
kernel_tuner.tune_kernel(..., restrictions=restrict, ...)
```

netherlands eScience center

User-defined metrics

- Kernel Tuner only reports time during tuning in milliseconds (ms)
  - This is the averaged time of (by default) 7 iterations, you can change the number of iterations using the `iterations` optional argument
  - Only the average time is printed to the screen, but all individual execution times will be returned by `tune_kernel` for further analysis
- You may want to use a metric different from time to compare kernel configurations

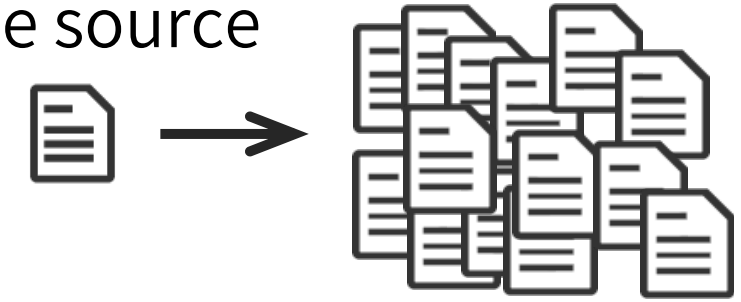
- Are composable, and therefore the order matters, so they are passed using a Python `OrderedDict`
- The `key` is the name of the metric, and `value` is a function that computes it
- For example:

```
from collections import OrderedDict
my_metrics = OrderedDict()
my_metrics["time_sec"] = lambda p : p["time"] / 1000.0
my_metrics["GFLOPS"] = lambda p : total_gflops / p["time_sec"]
kernel_tuner.tune_kernel(..., metrics=my_metrics, ...)
```

netherlands eScience center

Output verification

- When working with tunable code you are essentially maintaining many different versions of the same program in a single source



- It may happen that certain combinations of tunable parameters lead to versions that produce incorrect results
- Kernel Tuner can **verify** the output of each kernel while tuning!

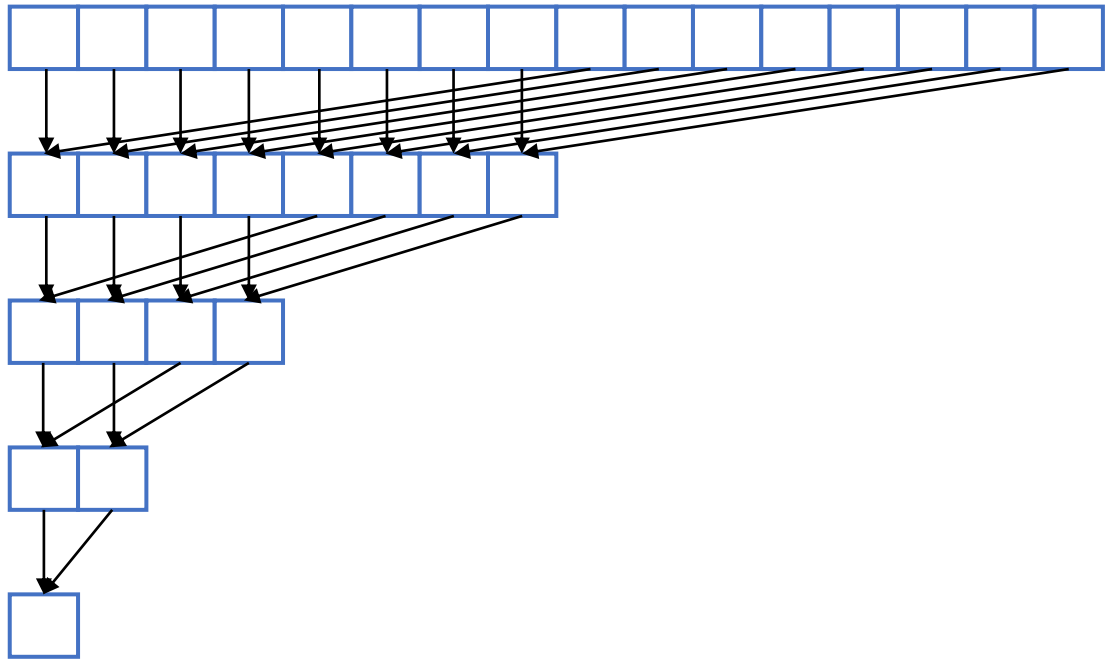
- When you pass a reference `answer` to `tune_kernel`:
  - The `answer` is a list that matches the kernel arguments in number, shape, and type, but contains `None` for input arguments
  - Kernel Tuner will run the kernel once before benchmarking and compare the kernel output against the reference `answer`
  - By default, Kernel Tuner will use `np.allclose()` with an absolute tolerance of `1e-6` to compare the state of all kernel arguments in GPU memory that have non-`None` values in the `answer` array
- And of course, you can modify this behavior, but first a simple example

Simple answer example

```
args = [c, a, b, n]
answer = [a+b, None, None, None]


tune_kernel("vector_add", kernel_string, size, args, tune_params,
            answer=answer, atol=1e-3)
```

- For some kernels the default verification functionality is not enough
- For example when output is different for different tunable parameters
- You can pass a function to the `verify` optional argument of `tune_kernel()`
- The verify function should take 3 arguments: a reference, the result, and a tolerance

- Say we have reduction kernel in which all thread blocks as a group iterate over the input
- Then each thread block computes a thread-block-wide partial sum
- A second kernel is used to sum all partial sums to a single summed value



Custom verification function - wrong

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
tune_params["num_blocks"] = [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]
problem_size = "num_blocks"


args = [sum_x, x, n]
reference = [numpy.sum(x), None, None]



tune_kernel("sum_floats", kernel_string, problem_size, args, tune_params, grid_div_x=[],
            verbose=True, answer=reference)
```

Custom verification function

```
tune_params["block_size_x"] = [32, 64, 128, 256, 512, 1024]
tune_params["num_blocks"] = [32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]
problem_size = "num_blocks"


args = [sum_x, x, n]
reference = [numpy.sum(x), None, None]

def verify_partial_reduce(cpu_result, gpu_result, atol=None):
    return numpy.isclose(cpu_result[0], numpy.sum(gpu_result[0]), atol=atol)


tune_kernel("sum_floats", kernel_string, problem_size, args, tune_params, grid_div_x=[],
            verbose=True, answer=reference, verify=verify_partial_reduce)
```

netherlands eScience center

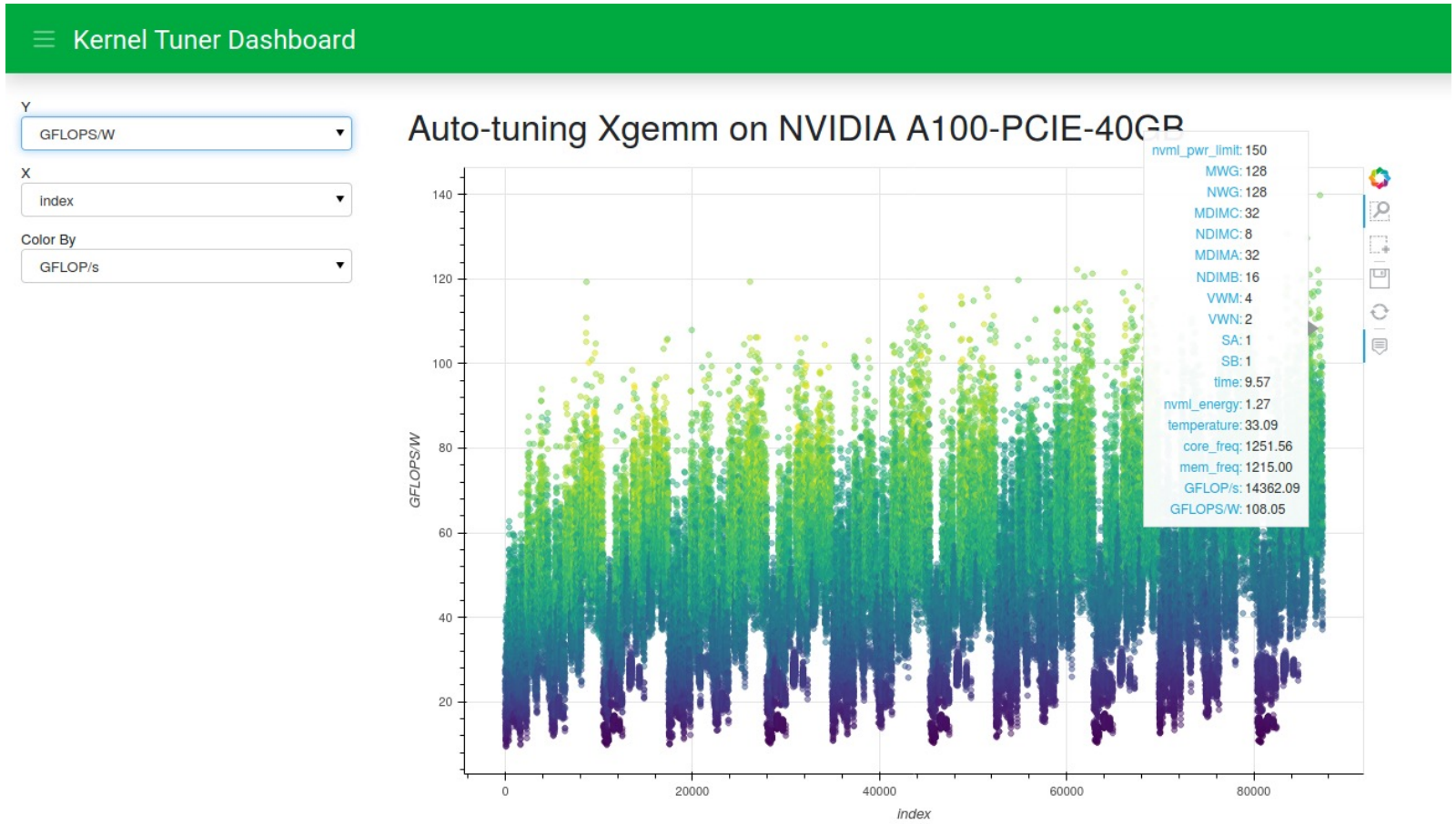
Caching tuning results

- Tuning large search spaces can take very long
- You might need to stop and continue later on
- Caching is enabled by passing a filename to the `cache` option
- Kernel Tuner will append new results to the cache directly after benchmarking a kernel configuration
- Kernel Tuner detects existing (possibly incomplete) cache files and automatically resumes tuning where it had left off

Kernel Dashboard

https://github.com/KernelTuner/dashboard

Live visualizations to monitor your Kernel Tuner runs

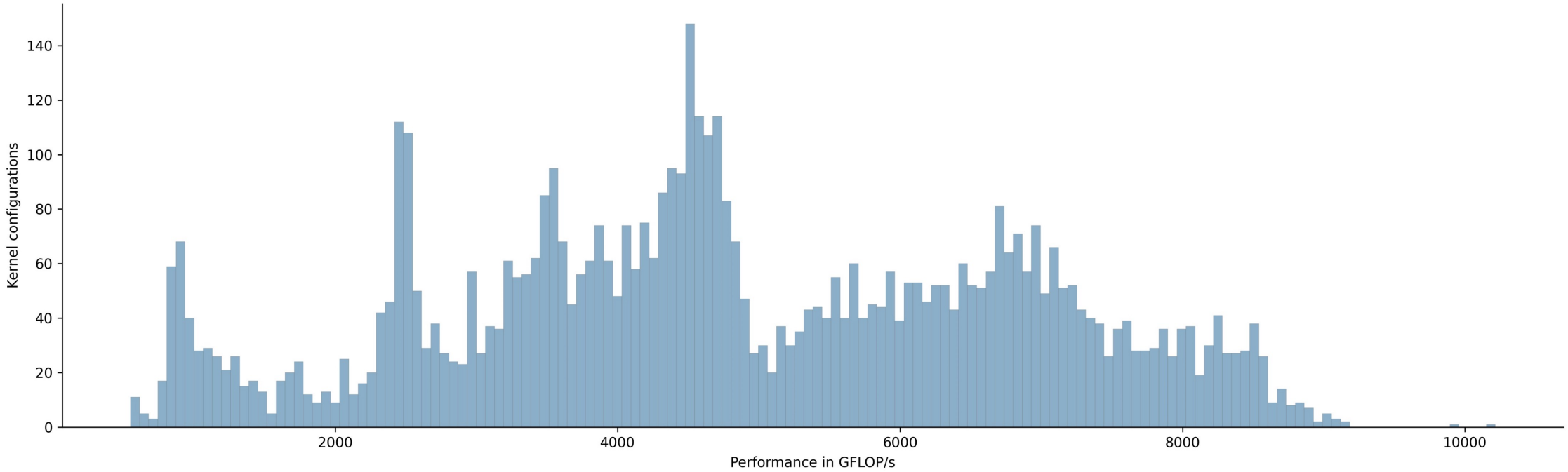


netherlands eScience center

Optimization strategies

Large search space of kernel configurations

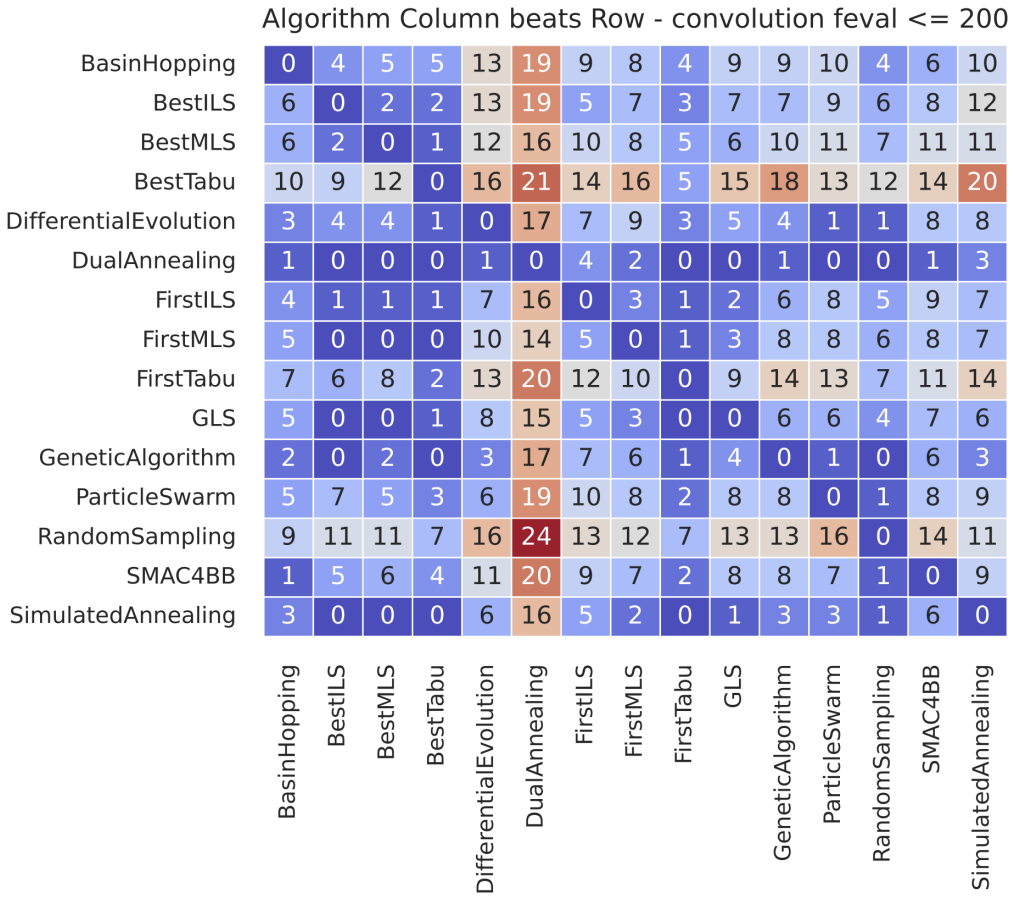
Auto-tuning a Convolution kernel on Nvidia A100



- Real-world search spaces can be very large
  - And an empirical auto-tuner needs to benchmark each configuration in the space
    - Often more than once to build robust statistics
- This results in long tuning time
  - In a recent paper using Kernel Tuner it took over 15 days to tune a highly optimized matrix multiply kernel
- The default strategy (*brute force*) explores the full configuration space
- To avoid this, we can apply some optimization strategies to tuning itself
  - Find a "*good enough*" configuration without exploring the whole space

Optimization strategies in Kernel Tuner

- Local optimization
  - Nelder-Mead, Powell, CG, BFGS, L-BFGS-B, TNC, COBYLA, and SLSQP
- Global optimization
  - Basin Hopping, Simulated Annealing, Differential Evolution, Genetic Algorithm, Particle Swarm Optimization, Firefly Algorithm, Bayesian Optimization, Multi-start local search, Iterative local search, Dual Annealing, Random search, …



Benchmarking optimization algorithms for auto-tuning GPU kernels
Richard Schoonhoven, Ben van Werkhoven, K. Joost Batenburg
IEEE Transactions on Evolutionary Computation (TEVC) 2022

- By passing `strategy="string_name"`, where `"string_name"` is any of:
  - `"brute_force"`: Brute force search
  - `"random_sample"`: random search
  - `"minimize"`: minimize using a local optimization method
  - `"basinhopping"`: basinhopping with a local optimization method
  - `"diff_evo"`: differential evolution
  - `"genetic_algorithm"`: genetic algorithm optimizer
  - `"mls"`: multi-start local search
  - `"pso"`: particle swarm optimization
  - `"simulated_annealing"`: simulated annealing optimizer
  - `"firefly_algorithm"`: firefly algorithm optimizer
  - `"bayes_opt"`: Bayesian Optimization
- Note that nearly all methods have specific options or hyperparameters that can be set using the `strategy_options` argument of `tune_kernel`

- https://kerneltuner.github.io/kernel_tuner/stable/optimization.html

- **Active** topic of research
- Different optimizers perform differently for different combinations of tunable kernel + GPU + input
- Nearly all methods are **stochastic**, meaning that they do not always return the global optimum or even the same result
- It is a **trade-off** between how much time you want to spend versus how strongly you want guarantees of finding an optimal configuration
- **Experiment**!

- Optimization strategies can be quickly benchmarked or tuned using cache files
- To use the simulation runner set `simulation_mode=True` with an existing `cache` file that contains information on ***all*** configurations in the search space

netherlands eScience center

Observers and energy efficiency

- **Observers** allow to modify the behavior during benchmarking and measure quantities other than time
- It loosely follows the **observer programming pattern**, allowing an observer object to observe certain events
- Also used **internally** for measuring time in the various backends

Creating your own observers

Simply extend the `BenchmarkObserver` base class to create your own `Observer`

Only `get_results()` is mandatory to implement

More information and documentation: https://kerneltuner.github.io/kernel_tuner/stable/observers.html

class **kernel_tuner.observers.BenchmarkObserver**

Base class for Benchmark Observers

after_finish()

after finish is called once every iteration after the kernel has finished execution

after_start()

after start is called every iteration directly after the kernel was launched

before_start()

before start is called every iteration before the kernel starts

during()

during is called as often as possible while the kernel is running

abstract **get_results**()

get_results should return a dict with results that adds to the benchmarking data

get_results is called only once per benchmarking of a single kernel configuration and generally returns averaged values over multiple iterations.

register_device(*dev*)

Sets self.dev, for inspection by the observer at various points during benchmarking

- **NVML** is the NVIDIA Management Library for monitoring/managing GPUs
- Kernel Tuner's **NVMLObserver** supports the following observable quantities: `"power_readings", "nvml_power", "nvml_energy", "core_freq", "mem_freq", "temperature"`
- If you pass an NVMLObserver, you can also use the following special tunable parameters to benchmark GPU kernels under certain conditions: `nvml_pwr_limit, nvml_gr_clock, nvml_mem_clock`
- Requires NVML, nvidia-ml-py3, and certain features may require root access

NVMLObserver example

```
...
tune_params["nvml_pwr_limit"] = [250, 225, 200, 175]


nvmlobserver = NVMLObserver(["nvml_energy", "temperature"])


metrics = OrderedDict()
metrics["GFLOPS/W"] = lambda p: (size/1e9) / p["nvml_energy"]


results, env = tune_kernel("vector_add", kernel_string, size, args,
                           tune_params, observers=[nvmlobserver],
                           metrics=metrics, iterations=32)
```

- By default, the optimization strategy minimizes `time`
- You can specify a custom tuning `objective` to be used instead to maximize or minimize
- The objective can be any observed quantity or user-defined metric

```
metrics["GFLOPS/W"] = lambda p: (size/1e9) / p["nvml_energy"]


results, env = tune_kernel("vector_add", kernel_string, size, args,
                           tune_params, observers=[nvmlobserver],
                           metrics=metrics, iterations=32,
                           objective="GFLOPS/W")
```

netherlands eScience center

Hands-on session

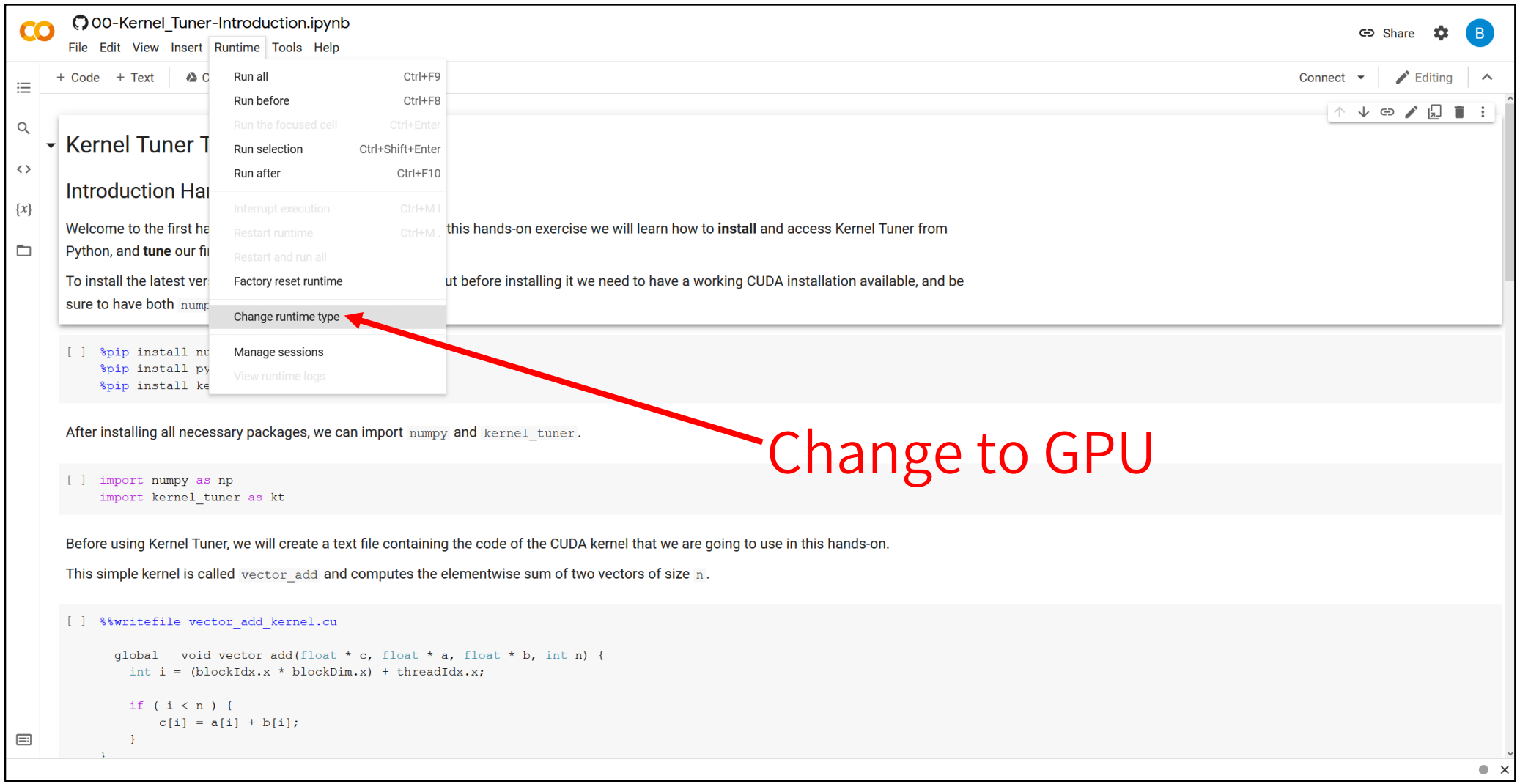
- The goal of this hands-on is to tune a Convolution CUDA kernel
  - Open the notebook in your Google Colab and work there
  - https://tinyurl.com/kerneltunertutorial

- Please follow the instructions in the Notebook

- Feel free to ask questions to instructors and mentors

- The notebook is located here:
  - https://github.com/KernelTuner/kernel_tuner_tutorial/blob/master/hands-on/cuda/Kernel_Tuner_Tutorial.ipynb

Change runtime type in Colab



netherlands eScience center

Closing remarks

- We are developing Kernel Tuner as an open-source project
- GitHub repository:
  - https://github.com/KernelTuner/kernel_tuner
  - License: Apache 2.0
- If you use Kernel Tuner in a project, please cite the paper:
  - B. van Werkhoven, Kernel Tuner: A search-optimizing GPU code auto-tuner, *Future Generation Computer Systems,* 2019

Acknowledgments

- The CORTEX project has received funding from the Dutch Research Council (NWO) in the framework of the NWA-ORC Call (file number NWA.1160.18.316).

- ESiWACE3 is funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Spain, Netherlands, Germany, Sweden, Finland, Italy and France, under grant agreement No 1010930