Kernel Tuner Tutorial – Advanced Topics

netherlands eScience center

Advanced topics session outline

- Performance portable applications
- Optimization strategies
- Observers
- Fourth hands-on session
- Closing

Performance portable applications

Performance portability

- The property that an application performance similarly on different hardware
- Auto-tuning may be used to achieve performance portability, if an application has been tuned on different hardware and we can select the right kernel based on the hardware at hand
- Kernel configuration selection can be done compile-time or run-time, based on earlier obtained tuning results

store_results

- The `store_results` function can be used to store information about the best performing configurations of a tunable kernel

```
from kernel_tuner.integration import store_results

store_results("vector_add_results.json", "vector_add", "vector_add.cu",
              tune_params, size, results, env, top=3)
```

- Stores the (e.g.) top 3% of tuning results for the specified combination of `problem_size` and GPU (retrieved from `env`) to the JSON file
  - The new results are appended to the JSON file
  - Results for the same `problem_size` and GPU are updated

Compile-time kernel selection

- Performs kernel selection at compile time
- Main advantage:
  - Can be done with very limited changes to the host application
- Limitation:
  - Limited to only selecting kernels based on properties known at compile-time, e.g. the target GPU

Compile-time kernel selection example

```python
from kernel_tuner.integration import store_results, create_device_targets

store_results("results.json", "vector_add", "vector_add.cu", tune_params, size, results, env)
create_device_targets("vector_add.h", "results.json")
```

vector_add.h

```c
/* header file generated by Kernel Tuner, do not modify by hand */
#pragma once
#ifndef kernel_tuner /* only use these when not tuning */

#ifdef TARGET_A100_PCIE_40GB
#define block_size_x 672

#elif TARGET_RTX_A6000
#define block_size_x 160

#else /* default configuration */
#define block_size_x 352
#endif /* GPU TARGETS */

#endif /* kernel_tuner */
```

Compile-time kernel selection example

```python
from kernel_tuner.integration import store_results, create_device_targets

store_results("results.json", "vector_add", "vector_add.cu", tune_params, size, results, env)
create_device_targets("vector_add.h", "results.json")
```

Kernel Tuner always inserts
#define kernel_tuner
When compiling kernels for
benchmarking

vector_add.h

```c
/* header file generated by Kernel Tuner, do not modify by hand */
#pragma once
#ifndef kernel_tuner /* only use these when not tuning */

#ifdef TARGET_A100_PCIE_40GB
#define block_size_x 672

#elif TARGET_RTX_A6000
#define block_size_x 160

#else /* default configuration */
#define block_size_x 352
#endif /* GPU TARGETS */

#endif /* kernel_tuner */
```

Compile-time kernel selection example

```python
from kernel_tuner.integration import store_results, create_device_targets

store_results("results.json", "vector_add", "vector_add.cu", tune_params, size, results, env)
create_device_targets("vector_add.h", "results.json")
```

This `block_size_x` value showed best performance on the A100

vector_add.h

```c
/* header file generated by Kernel Tuner, do not modify by hand */
#pragma once
#ifndef kernel_tuner /* only use these when not tuning */

#ifdef TARGET_A100_PCIE_40GB
#define block_size_x 672

#elif TARGET_RTX_A6000
#define block_size_x 160

#else /* default configuration */
#define block_size_x 352
#endif /* GPU TARGETS */

#endif /* kernel_tuner */
```

Compile-time kernel selection example

```python
from kernel_tuner.integration import store_results, create_device_targets

store_results("results.json", "vector_add", "vector_add.cu", tune_params, size, results, env)
create_device_targets("vector_add.h", "results.json")
```

vector_add.h

```c
/* header file generated by Kernel Tuner, do not modify by hand */
#pragma once
#ifndef kernel_tuner /* only use these when not tuning */

#ifdef TARGET_A100_PCIE_40GB
#define block_size_x 672

#elif TARGET_RTX_A6000
#define block_size_x 160

#else /* default configuration */
#define block_size_x 352
#endif /* GPU TARGETS */

#endif /* kernel_tuner */
```

This `block_size_x` value showed best performance on the A6000

Compile-time kernel selection example

```python
from kernel_tuner.integration import store_results, create_device_targets

store_results("results.json", "vector_add", "vector_add.cu", tune_params, size, results, env)
create_device_targets("vector_add.h", "results.json")
```

vector_add.h

```c
/* header file generated by Kernel Tuner, do not modify by hand */
#pragma once
#ifndef kernel_tuner /* only use these when not tuning */

#ifdef TARGET_A100_PCIE_40GB
#define block_size_x 672

#elif TARGET_RTX_A6000
#define block_size_x 160

#else /* default configuration */
#define block_size_x 352
#endif /* GPU TARGETS */

#endif /* kernel_tuner */
```

This `block_size_x` value showed best performance overall, on all GPUs

Compile-time kernel selection example

In vector_add.cu:

```
#include "vector_add.h"
```

In Makefile:

```
TARGET_GPU = `nvidia-smi --query-gpu="gpu_name" --format=csv,noheader | sed -E 's/[^[:alnum:]]+/_/g'`
CU_FLAGS = -DTARGET_${TARGET_GPU}

vector_add.o: vector_add.cu
	nvcc ${CU_FLAGS} -c $< -o $@
```

Typing ‘make’ will now use different `block_size_x` values on A100, A6000, and on other GPUs

Run-time kernel selection

- More flexible, allows also to select kernels based on data size or other properties
- Requires more significant modification of the host application
- Depends on the programming language of the host application

Run-time kernel selection Python

In Python:

```python
from kernel_tuner.integration import store_results

store_results("vector_add_results.json", "vector_add", "vector_add.cu", tune_params, size, results, env)
```

In the Python host application:

```python
from kernel_tuner import kernelbuilder

# create a kernel using the stored results
vector_add = kernelbuilder.PythonKernel(kernel_name, kernel_string, n, args,
                                        results_file=test_results_file)

# call the kernel
vector_add(c, a, b, n)
```

Run-time kernel selection C++ example

In Python:

```python
from kernel_tuner.integration import store_results

store_results("vector_add_results.json", "vector_add", "vector_add.cu", tune_params, size, results, env)
```

In vector_add.cpp:

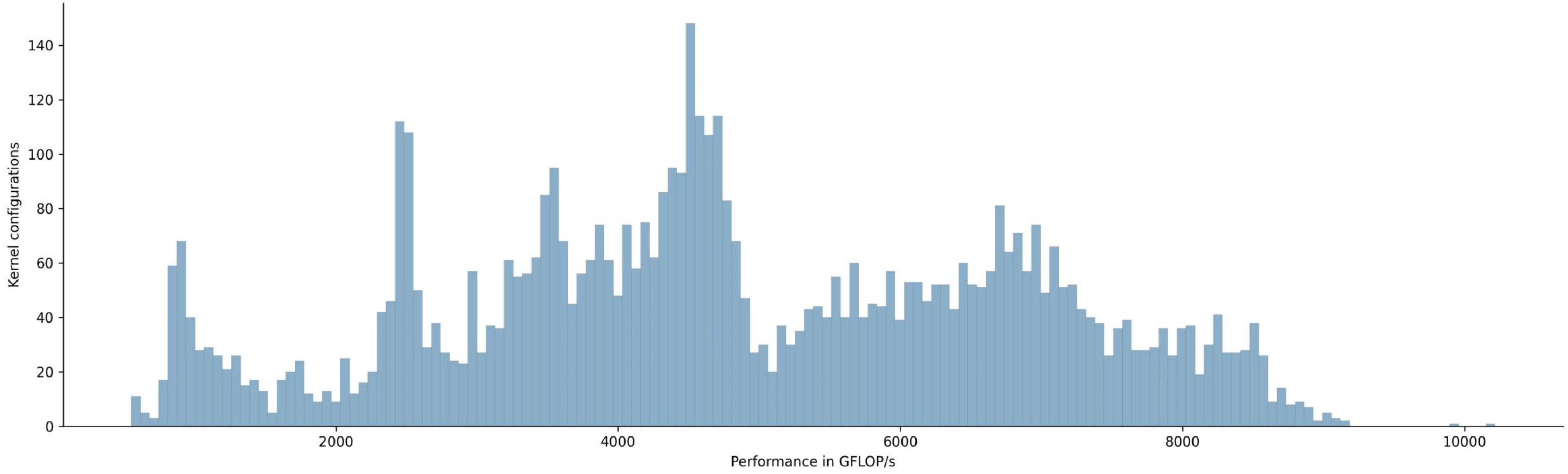
```cpp
#include "kernel_launcher.h"
using namespace kernel_launcher;


auto vector_add = CudaKernel<float*, float*, float*, int>::compile_best_for_current_device(
            "vector_add_results.json", 800000000, "vector_add.cu", {"-std=c++11"});

int grid_size = (n + vector_add.get_block_dim().x - 1) / vector_add.get_block_dim().x;

vector_add(grid_size)(dev_C, dev_A, dev_B, n);
```

Uses Kernel Launcher header-only C++ library: https://github.com/stijnh/kernel_launcher

Optimization strategies

Large search space of kernel configurations

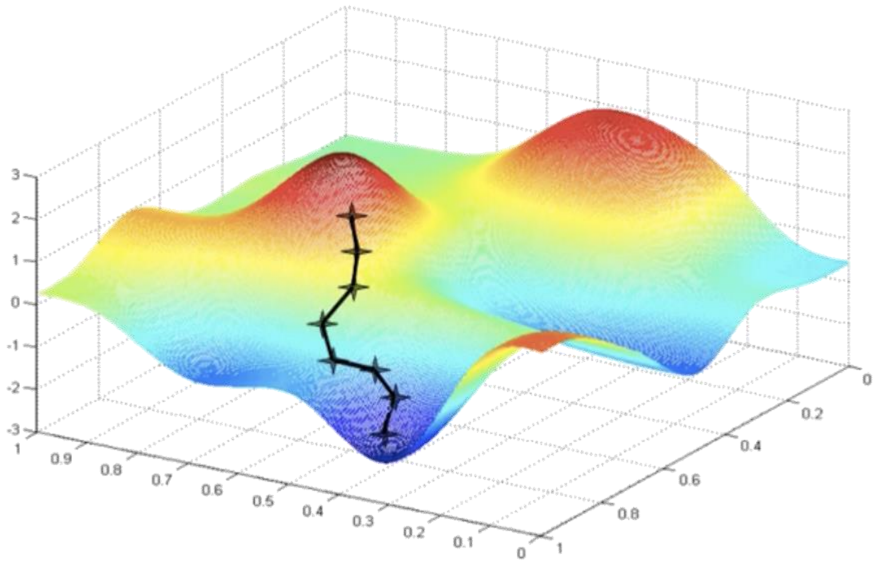
Auto-tuning a Convolution kernel on Nvidia A100



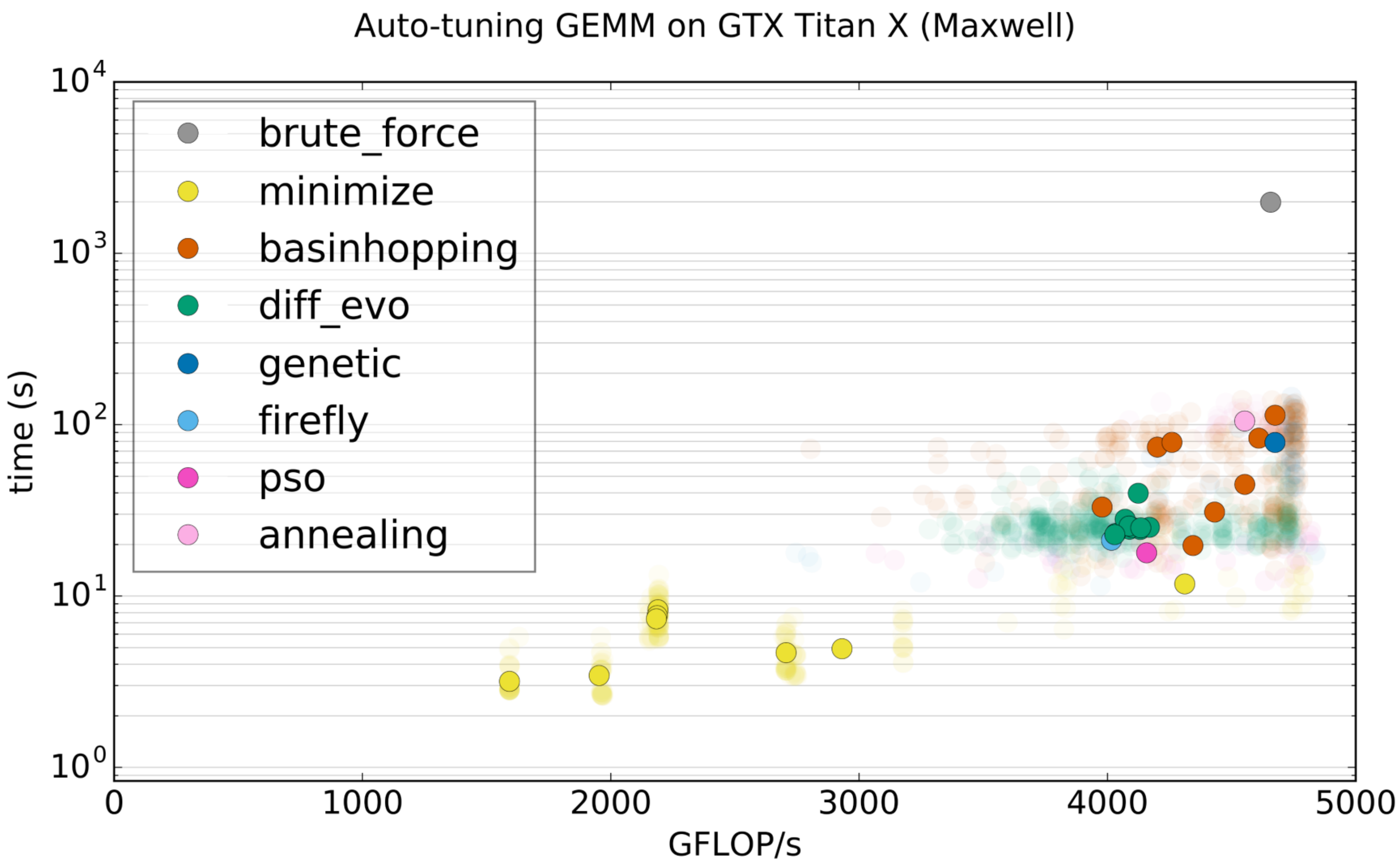
Optimization strategies in Kernel Tuner

- 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
  - …



Speeding up auto-tuning



Your mileage may vary

- Active topic of research
- Different optimizers seem to perform differently for certain 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 all a matter of how much time you have versus how strongly you want guarantees of finding an optimal configuration
- Experiment!

How to use a search strategy

- 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`

Observers

Observers introduction

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

Observer base class

```python
class BenchmarkObserver(ABC):
    """Base class for Benchmark Observers"""

    def register_device(self, dev):
        """Sets self.dev, for inspection by the observer at various points during benchmarking"""
        self.dev = dev

    def before_start(self):
        """before start is called every iteration before the kernel starts"""
        pass

    def after_start(self):
        """after start is called every iteration directly after the kernel was launched"""
        pass

    def during(self):
        """during is called as often as possible while the kernel is running"""
        pass

    def after_finish(self):
        """after finish is called once every iteration after the kernel has finished execution"""
        pass

    @abstractmethod
    def get_results(self):
        """ 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.
        """
        pass
```

NVMLObserver

- NVML is the NVIDIA Management Library for monitoring and 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)
```

GPU Memory management

- Kernel Tuner reuses the same data on the GPU for benchmarking all kernel configurations
- When you use output verification, GPU data is refreshed from the host to the GPU right before calling the kernel once for output verification, before benchmarking
- This assumes the kernels are idempotent or at least that running the kernel multiple times on the same data does not significantly impact performance

Breadth First Search (BFS)

- For some kernels, like BFS, this assumption does not hold
- The BFS kernel in the Rodinia Benchmark Suite works as follows:
  - Threads are created for each node in the graph, each thread checks the `g_graph_mask` Boolean array to see if it is active in this computation
  - If so, it iterates over all edges of this node to set the `g_updating_graph_mask` for all neighbors to true and sets its own `g_graph_mask` to false
  - A separate kernel is used to update `g_graph_mask` based on the `g_updating_graph_mask`
- What happens when you call the first kernel multiple times?

Tuning BFS with Kernel Tuner

- Problem:
  - We cannot execute the first BFS kernels multiple times on the same data
- Not working solutions:
  - Use iterations=1, does not solve our problem because data is reused for multiple kernel configurations
  - Enable output verification and set iterations=1, does not work either because output verification calls the kernel once outside of benchmarking
- Real solution: Use an observer!

BFSObserver

```
class BFSObserver(BenchmarkObserver):

    def __init__(self, args):
        self.args = args

    def before_start(self):
        for i, arg in enumerate(self.args):
            if not arg is None:
                self.dev.memcpy_htod(self.dev.allocations[i], arg)

    def get_results(self):
        return {}
```

dev is set by Kernel Tuner to each observer to have access to the device functions interface of the backend

Tuning the BFS kernel

```
...

args = [starting, no_of_edges, graph_edges, g_graph_mask, g_updating_graph_mask,
        g_graph_visited, g_cost, num_of_nodes]

refresh_args = [None, None, None, g_graph_mask, g_updating_graph_mask, None, g_cost, None]

bfs_observer = BFSObserver(refresh_args)

results, env = tune_kernel("bfs_kernel", "kernels.cu",
                           problem_size, args, tune_params, metrics=metrics,
                           compiler_options=cp, answer=answer, grid_div_x=grid_div_x,
                           verbose=True, observers=[bfs_observer])
```

Fourth hands-on session

Advanced hands-on

- The fourth hands-on notebook is:
  - https://github.com/benvanwerkhoven/kernel_tuner_tutorial/blob/master/hands-on/cuda/03_Kernel_Tuner_Advanced.ipynb
- The goal of this hands-on is to experiment with search optimization strategies and custom observers
  - Copy the notebook to your Google Colab and work there
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors

Optional hands-on

- Done with the fourth hands-on already?
- Keep playing with this notebook
    - https://github.com/benvanwerkhoven/kernel_tuner_tutorial/blob/master/hands-on/cuda/Kernel_Tuner_Tutorial.ipynb
- Keep experimenting with your own code
- Feel free to ask questions to instructors and mentors

Closing remarks

Kernel Tuner – developed open source

- We are developing Kernel Tuner as an open source project
- GitHub repository:
  - https://github.com/benvanwerkhoven/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

Feature Roadmap

- Allow tuning objectives other than time
- Remote/parallel tuning
- Multi-objective optimization
- Further support for integrating kernels into applications
- API for plotting/analysis of tuning results
- Tuning compiler flags

Contributions are welcome!

- Contributions can come in many forms: tweets, blog posts, issues, pull requests
- Before making larger changes, please create an issue to discuss

- For the full contribution guide, please see: https://benvanwerkhoven.github.io/kernel_tuner/contributing.html

Thanks to all contributors!










And many more!

Netherlands eScience Center

- Collaborate with us:
  - Open calls
  - Partner in EU projects
  - Direct collaboration
  - See: https://www.esciencecenter.nl/collaborate-with-us/
- We are hiring:
  - Research Software Engineers (RSEs)
  - See: https://www.esciencecenter.nl/vacancies/

Funding

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).

This work is funded in part by the Netherlands eScience Center.

Thanks!

If you have any further questions or would like to reach out, please feel free to contact me at:

Ben van Werkhoven
b.vanwerkhoven@esciencecenter.nl