Optimizing GPU code with Kernel Tuner

Alessio Sclocco, Stijn Heldens, Ben van Werkhoven

netherlands eScience center

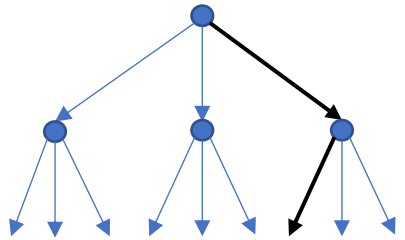
To maximize GPU code performance, you need to find the best combination of:

- 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
- …

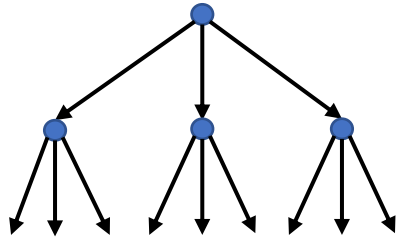
Problem:

- Creates a very large search space

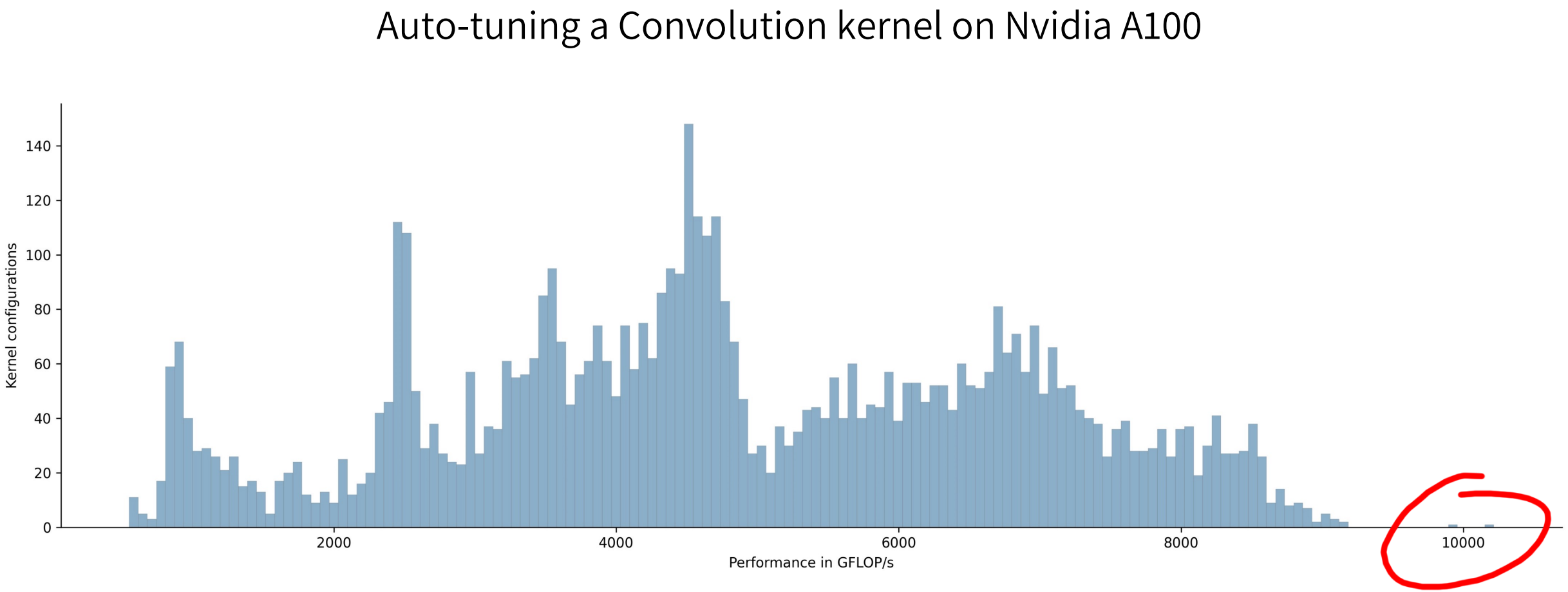
- Optimizing code manually, you iteratively perform:
  - Modify the code
  - Run a few benchmarks
  - Revert or accept the change



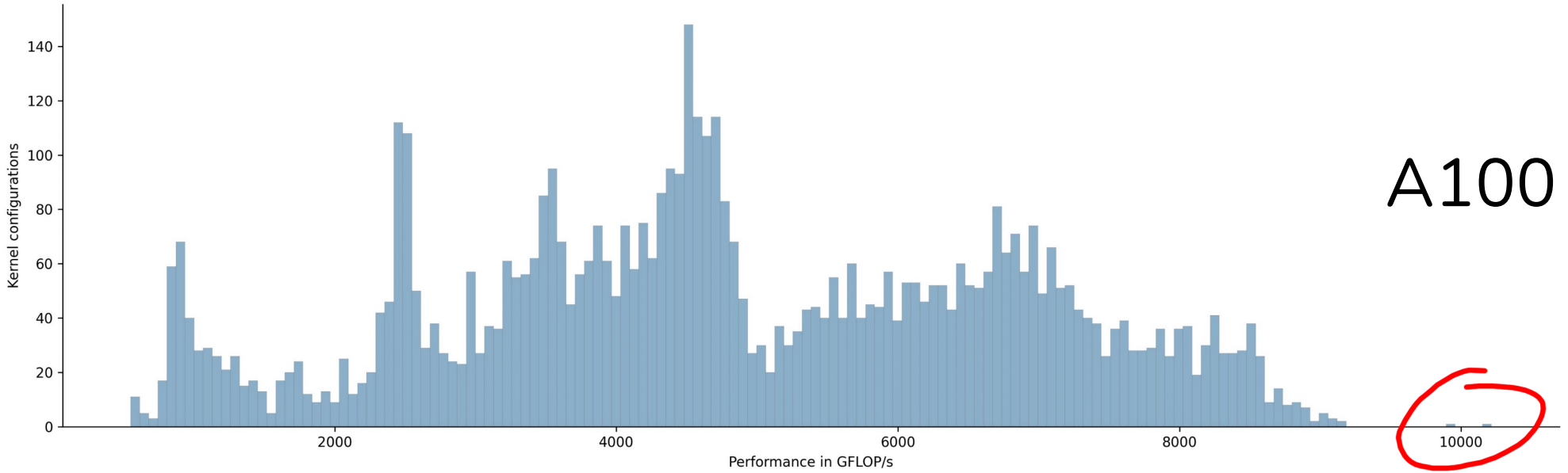
- With auto-tuning you:
  - Write a templated version of your code or a code generator
  - Benchmark the performance of all code variants



Large search space of kernel configurations



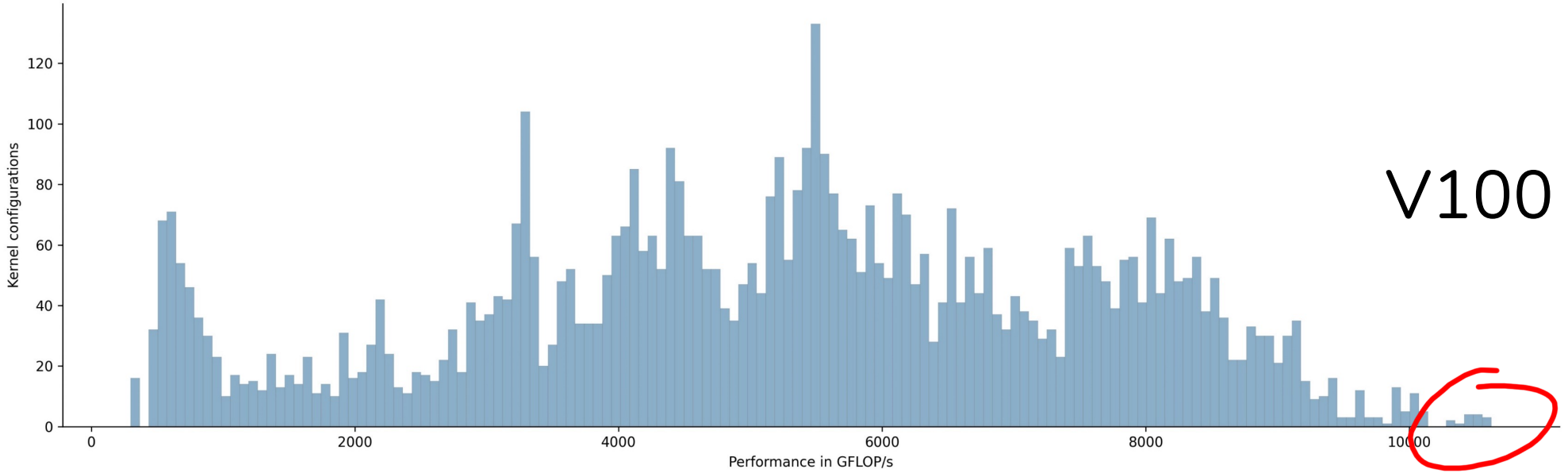
On different GPUs ...



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

Kernel Tuner

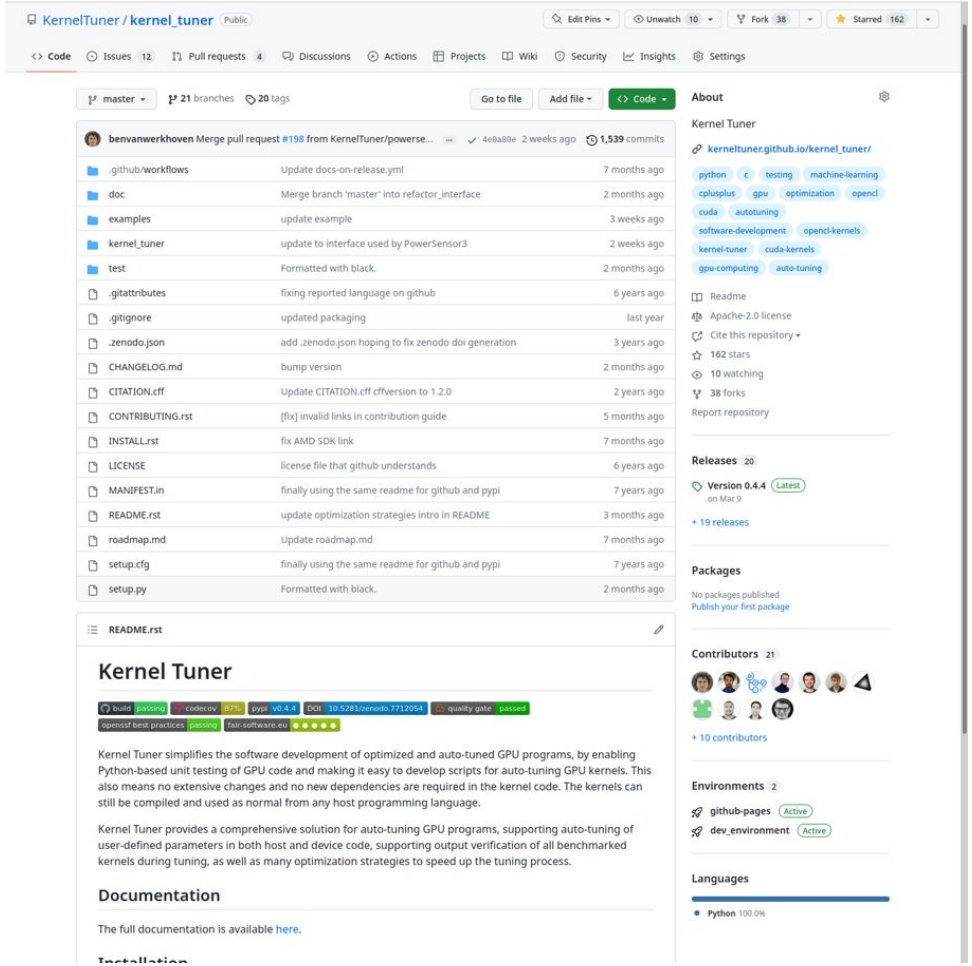
Kernel Tuner is a Python tool for tuning and testing GPU kernels

Easy to use:

- Can be used directly on existing kernels and code generators
- Inserts no dependencies in the kernels or host application
- Kernels can still be compiled with regular compilers

Supports:

- Tuning functions in OpenCL, CUDA, C, and Fortran
- Large number of effective search optimizing algorithms
- Output verification for auto-tuned kernels and pipelines
- 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 for the full list
  - https://github.com/KernelTuner/kernel_tuner

Minimal example

```
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)
```

What Kernel Tuner does

- Creates the search space:
  - Computed as the Cartesian product of all values of all tunable parameters

- 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 everything that is needed to compile and run our kernel:
  - This includes source code and compiler options
  - This is 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 allocates GPU memory and moves data in and out of the GPU for you

Kernel Tuner supports the following types for kernel arguments:

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

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

Specifying thread block dimensions

- 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:
  - `params["block_size_x"] = [32, 64, 128, 256]`

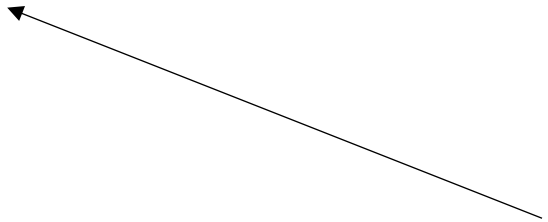
- You can use different names for these by passing the `block_size_names` option using a list of strings
- Note: when you change the thread block dimensions, the number of thread blocks used to launch the kernel generally changes as well

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 allows 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
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)
```



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

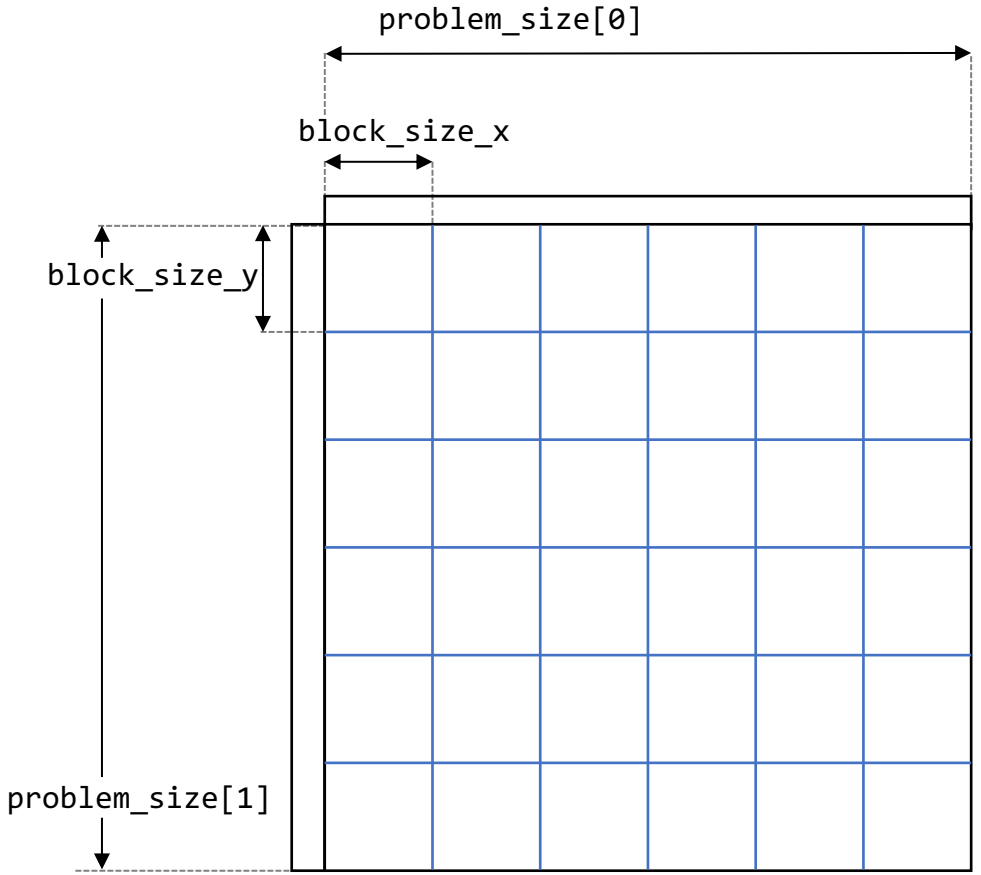
Vector add example

```
import numpy
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)
```

n is the number of elements in our array, the number of thread blocks depends on both n and `block_size_x`

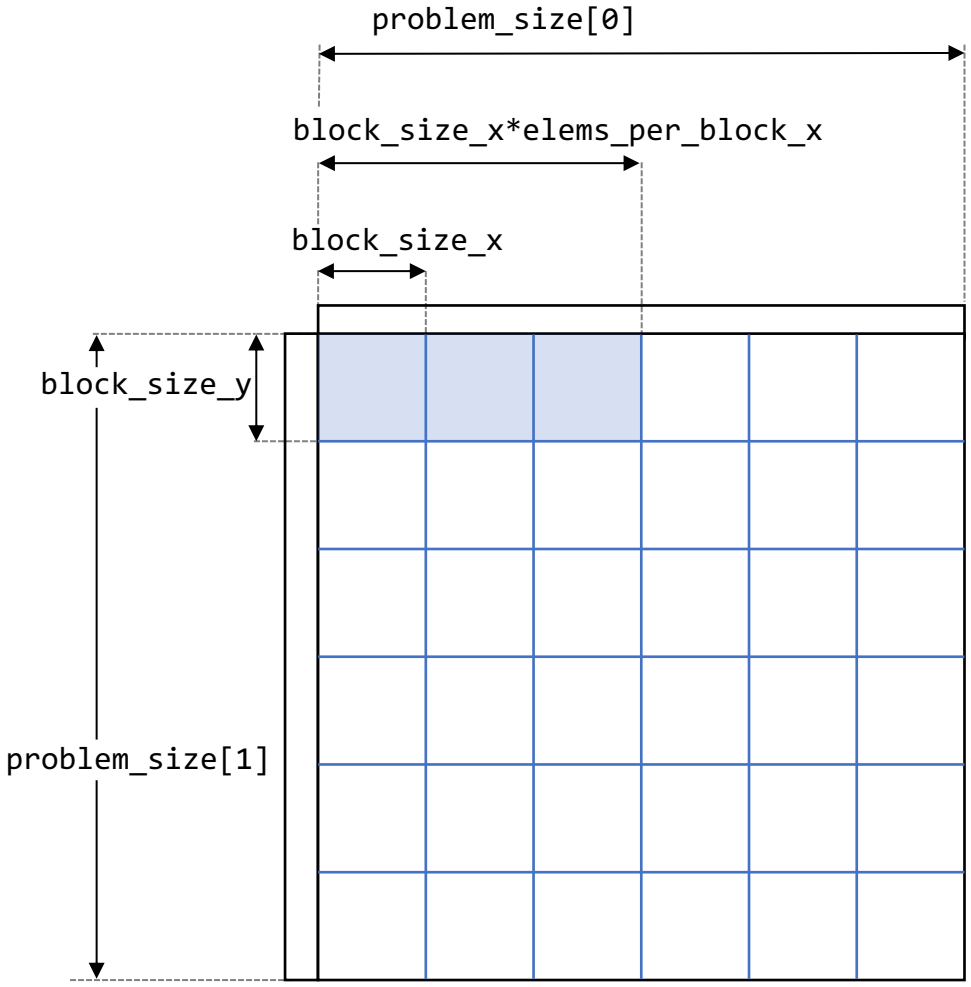
Specifying grid dimensions

- 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)`



Grid divisor lists

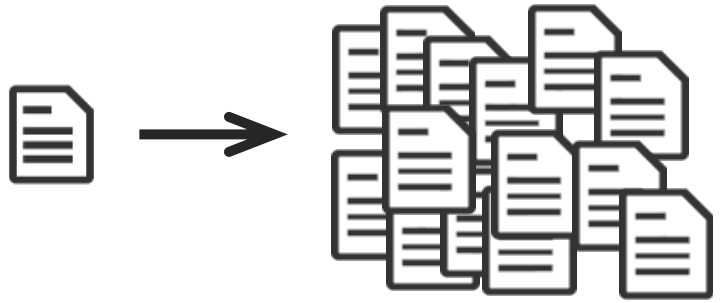
- Other parameters, or none at all, may also affect the grid dimensions
- Grid divisor lists control how `problem_size` is divided to compute the grid size
- Use the optional arguments:
  - `grid_div_x`, `grid_div_y`, and `grid_div_z`
- You may disable this feature by explicitly passing empty lists as 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
- Use strings to derive `problem_size` from a tunable parameter
- May also be a (lambda) function that takes a dictionary of tunable parameters and returns a tuple of integers
- For example, `reduction.py`:

```
size = 800_000_000
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"
grid_div_x = []
```

- 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 kernels while tuning!



- When you pass a reference `answer` to `tune_kernel`:
  - Kernel Tuner will run the kernel once before benchmarking and compare the kernel output against the reference `answer`
  - The `answer` is a list that matches the kernel arguments in number, shape, and type, but contains `None` for input arguments
  - 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 by defining your own verification function

Restricting the search

- 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["tile_size_x"] = [1, 2, 3, 4, 5, 6, 7, 8]
tune_params["loop_unroll_factor_x"] = [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

Dependent parameters example

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

- In this example:
  - One parameter controls a loop count: `tile_size_x`
  - Another parameter controls the partial loop unrolling factor of that loop: `loop_unroll_factor_x`
- By default, Kernel Tuner considers search space to be the Cartesian product of all possible combinations of all values for all parameters
- But only configurations in which `loop_unroll_factor_x` is a divisor of `tile_size_x` are valid

Partial loop unrolling example

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

# define a lambda function that returns True when a configuration is valid
restrict = lambda p: p["loop_unroll_factor_x"] <= p["tile_size_x"] and
                     p["tile_size_x"] % p["loop_unroll_factor_x"] == 0

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

- During tuning, Kernel Tuner reports the execution time of each configuration
  - The reported time is in milliseconds
  - This is the averaged time of, by default, 7 iterations
    - You can change the number of iterations using the `iterations` optional argument
  - Actually, all individual execution times will be returned by `tune_kernel`, but only the average is printed to screen
- You may want to use a metric different from time to compare kernel configurations

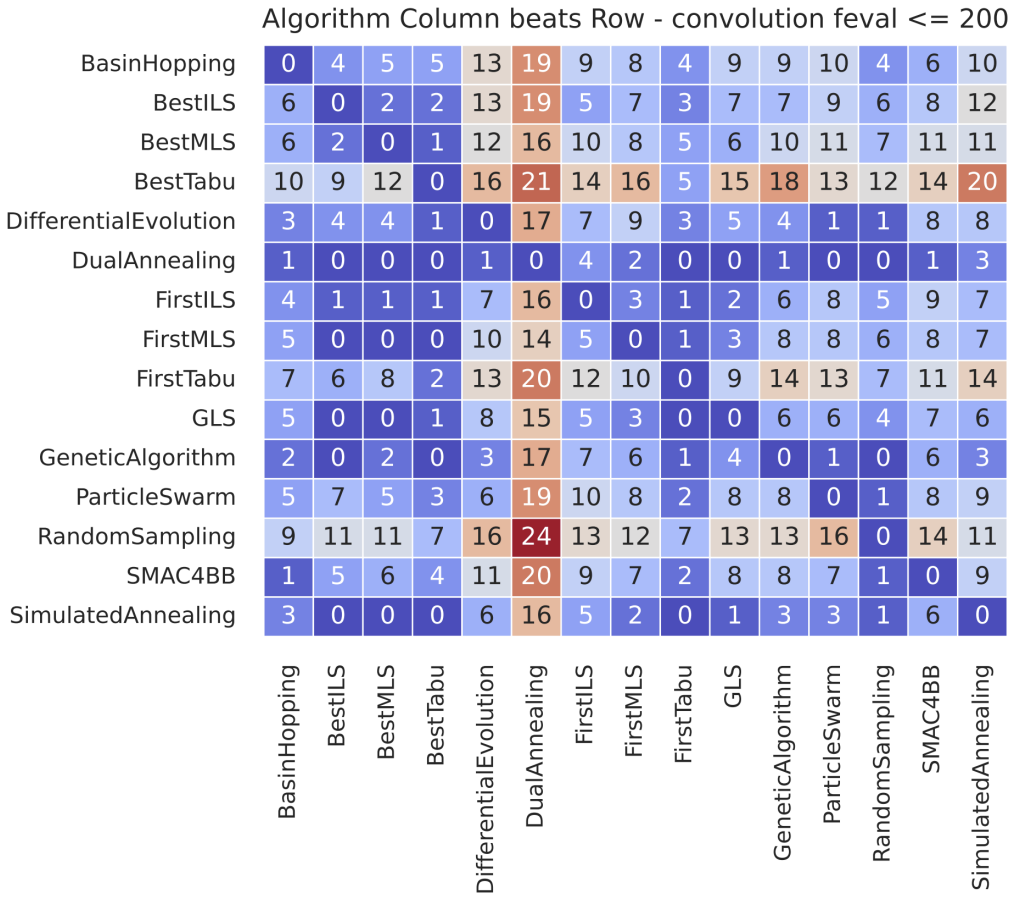
- Are composable, and therefore the order matters, so they are passed using a Python `OrderedDict`
- The `key` is the name of the metric, and the `value` is a function that computes it
- For example:

```
from collections import OrderedDict
metrics = OrderedDict()
metrics["time_s"] = lambda p : (p["time"] / 1000)
```

- 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 may result 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="strategy_name"`, where `"strategy_name"` is any of:
  - `"brute_force"`: Brute force search
  - `"random_sample"`: random search
  - `"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

- Code variants and tunable parameters describe a *search space*
- Auto-tuners automatically explore the search space
  - Possibly using *optimization algorithms*

- Kernel Tuner
  - Uses Python to tune GPU kernels
  - Describe ‘problem size’, let tuner try different grid and thread block dimensions
  - Output verification is used to test kernels while tuning
  - Restrictions encode dependencies between tunable parameters
  - Time is the default metric, but users can define their own
  - Optimization algorithms can be used to reduce the tuning time

netherlands eScience center

Optimizing energy efficiency

The impact of training ML models

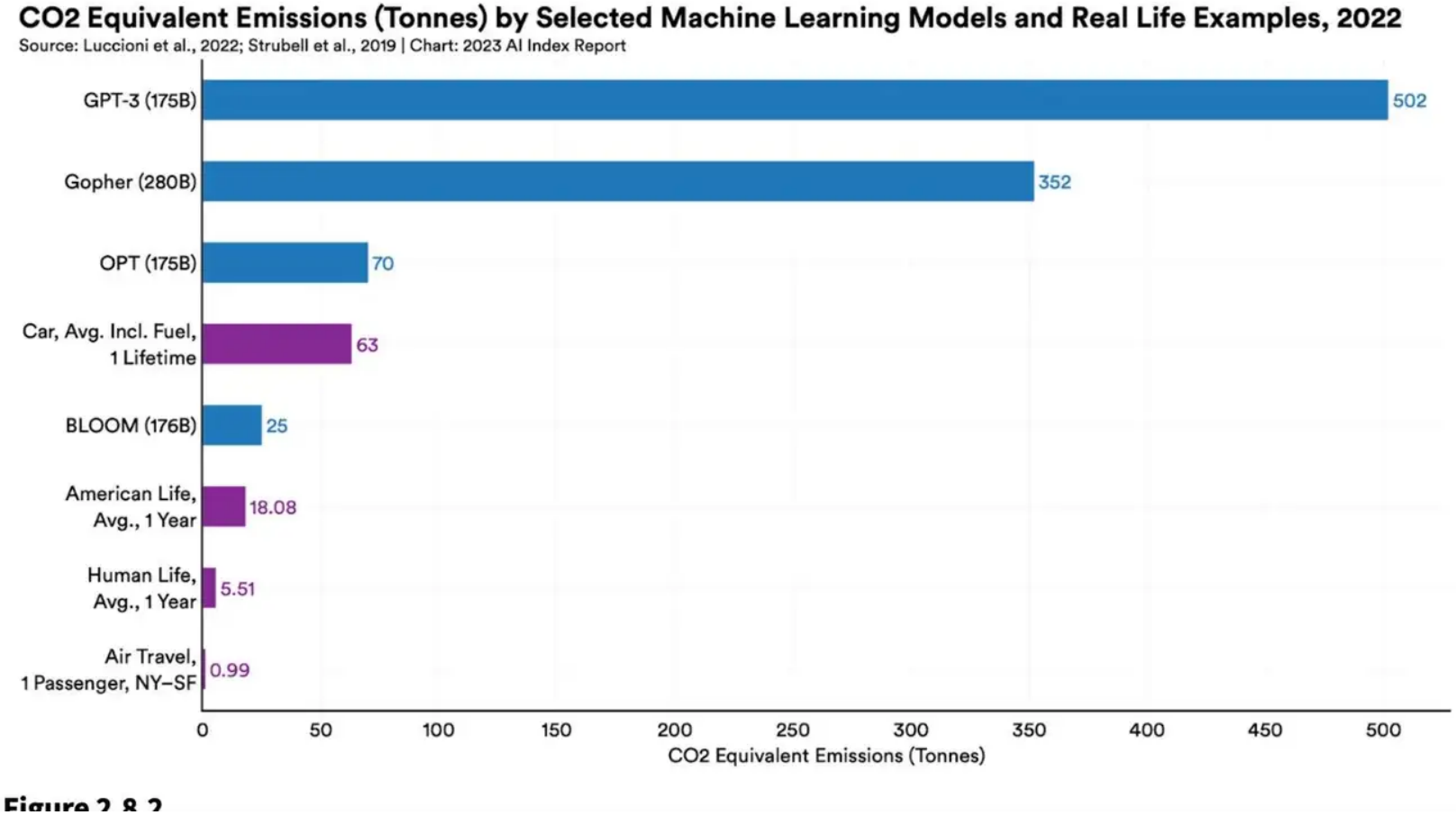
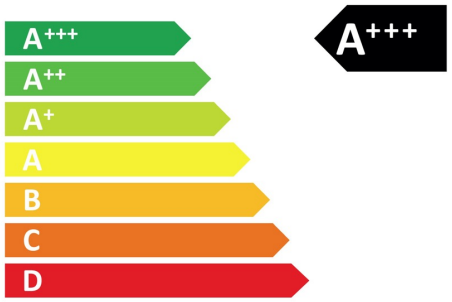


Figure 2.8.2

Stanford University Human-Centered Artificial Intelligence. *Artificial Intelligence Index Report 2023*. https://aiindex.stanford.edu/wp-content/uploads/2023/04/HAI_AI-Index-Report_2023.pdf



Minimize energy consumption instead of the execution time

Auto-tuning GPU code for energy efficiency, challenges:

- How to measure power consumption during tuning?
- Is there a difference between optimizing for time or energy?
- How effective are optimization algorithms at optimizing for energy?
- Can we narrow down the search space?
- Which methods to support? Frequency tuning or power capping?
- How much energy can be gained in practice?

Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

- GFLOPS/W is a widely-used metric for **energy efficiency**, also used by Green500
- GFLOPS (or GFLOP/s) is a measure of **computational throughput**
  - billions of floating-point operations per second
- We can compute GFLOP/s as: the total number of floating-point operations (in billions) divided by the kernel execution time in seconds
- Watt (W) is a measure of **power**, equal to energy in Joule (J) per second (s)

$$\frac{GFLOPS}{W} = \frac{GFLOP/s}{J/s} = \frac{GFLOP}{J}$$

Remember: Observers

- Observers extend the benchmarking capabilities of Kernel Tuner
- Allows users to subscribe to certain events during benchmarking
- Also used internally for measuring time
- Kernel Tuner implements observers for measuring power consumption:
  - **PowerSensorObserver**
  - **NVMLObserver**
  - **PMTObserver**

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

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

Nvidia Management Library (NVML)

Allows to measure several quantities during tuning:

- Power consumption, core frequency, core voltage, memory frequency, GPU temperature, and energy consumption

Provides an interface within Kernel Tuner to NVML

- Enables new tunable parameters:
  - `nvml_pwr_limit`, `nvml_gr_clock`, `nvml_mem_clock`
  - Setting these may require root privileges

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, Kernel Tuner's optimization strategy minimizes `time`
- But there is also support for using a custom tuning `objective`
- 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")
```

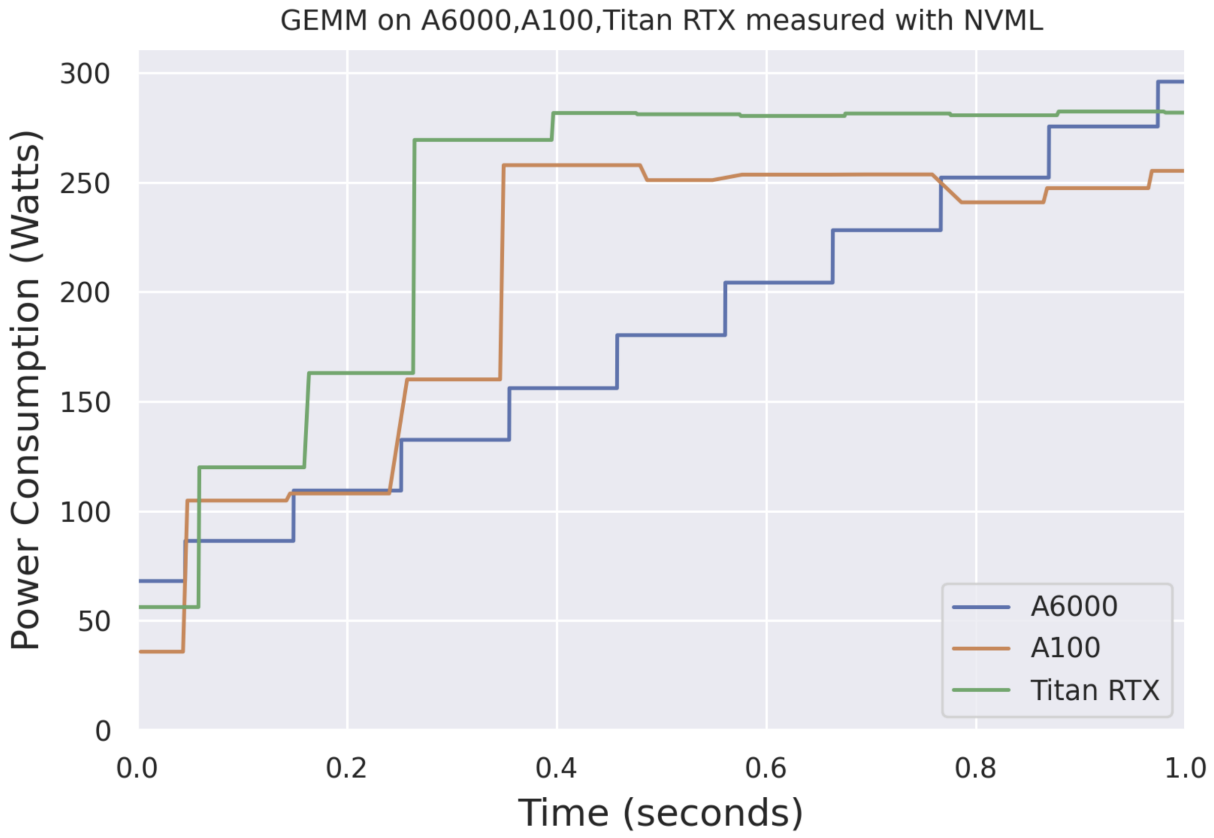
Measuring power consumption with NVML

NVML can observe GPU temperature, core and memory clocks, core voltage, and power

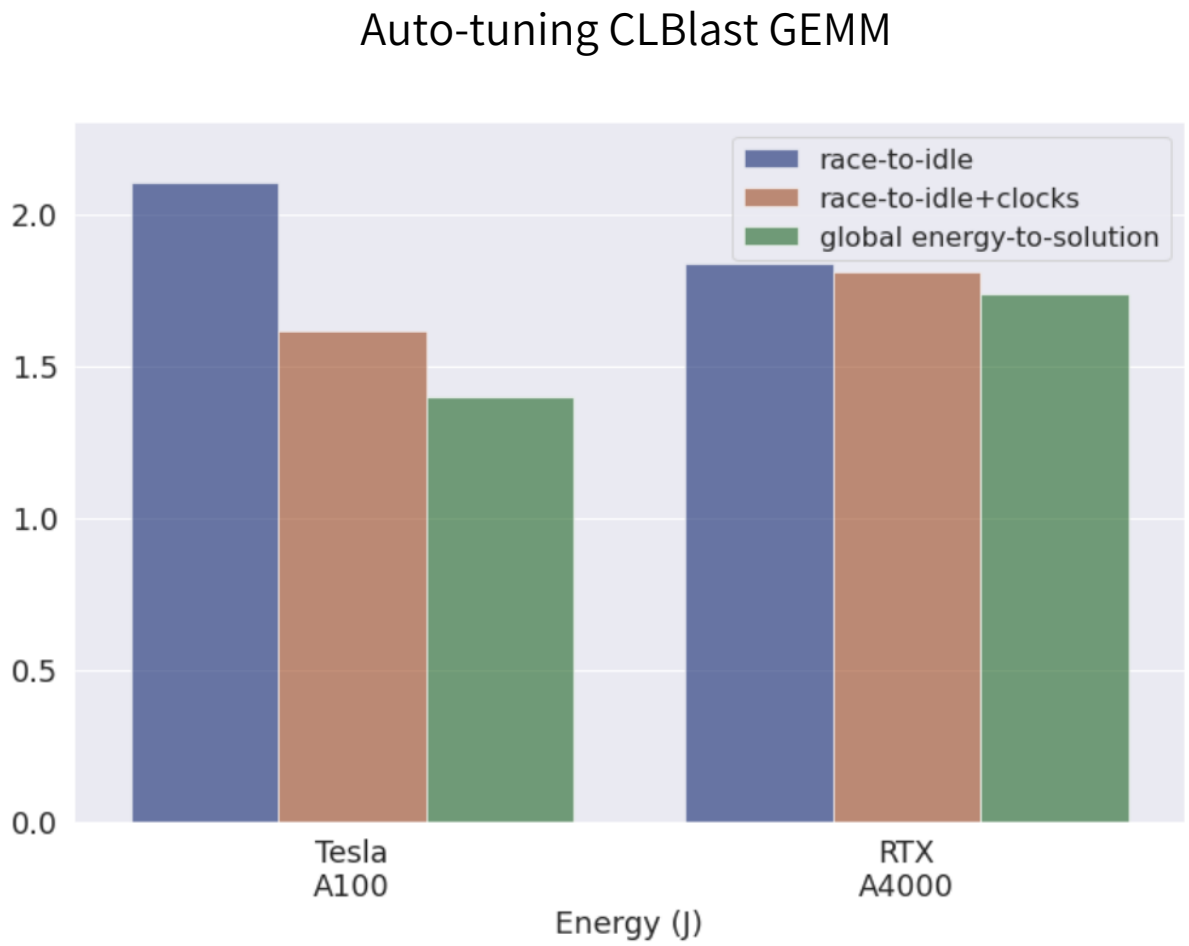
Advantages:
- Highly available

Disadvantages:
- Returns time-averaged power, not instantaneous power consumption
- Limited time resolution

Current solution:
- Measure power for one second while repeatedly running the kernel



- *Race-to-idle:* use the device for the shortest duration
- *Energy-to-solution:* use the fewest Joules to complete the operation



Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

Power limit vs frequency tuning

Many tunable parameters affect compute performance and/or energy efficiency

But we can also:

- Limit the GPU **clock frequency**, allow GPU to vary power consumption
- Limit the GPU **power consumption**, allow GPU to determine clock frequency

Both methods require root privileges for the latest generations of Nvidia GPUs

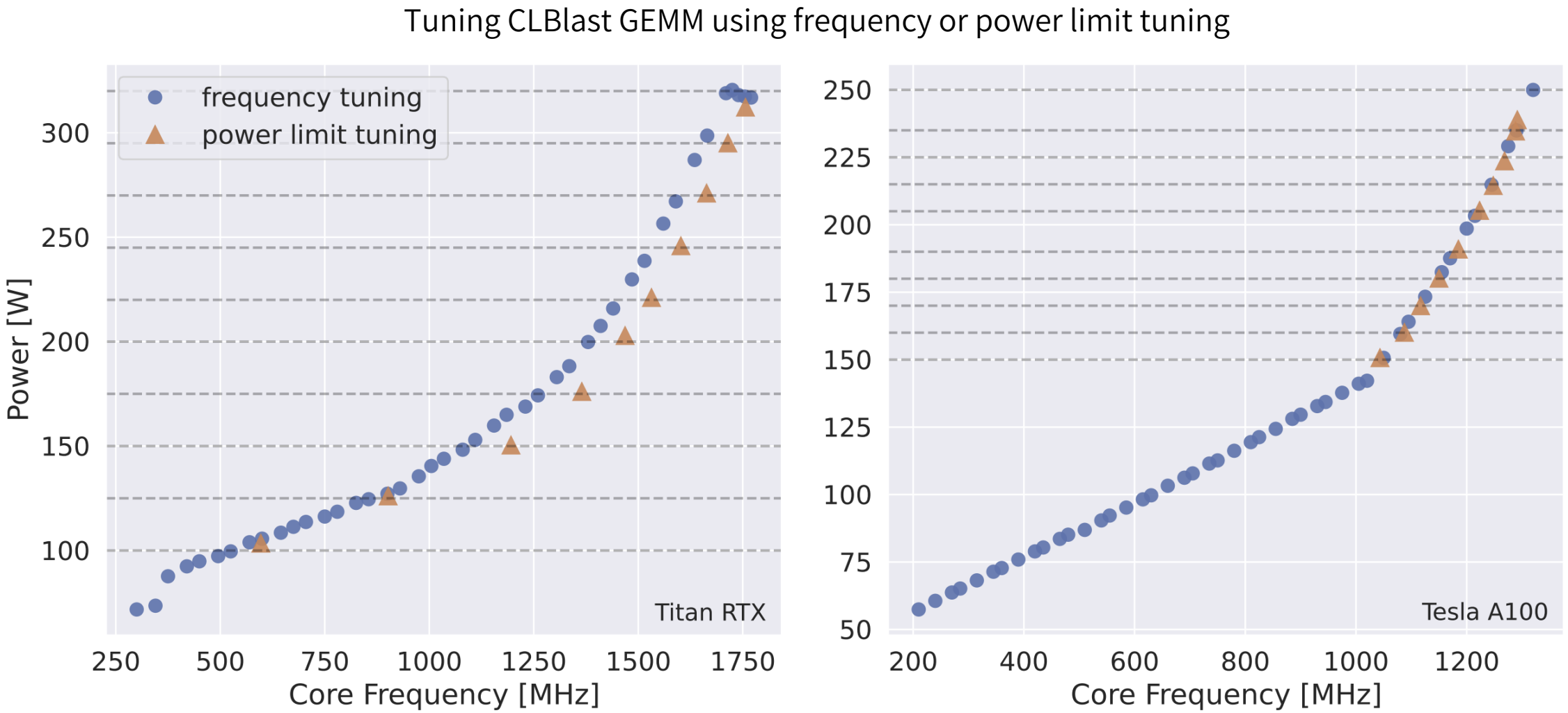
Advantages of **power capping**:

- Potentially more effective, GPU may also lower memory clock
- Reliable method in face of limited power

Advantages of **frequency tuning**:

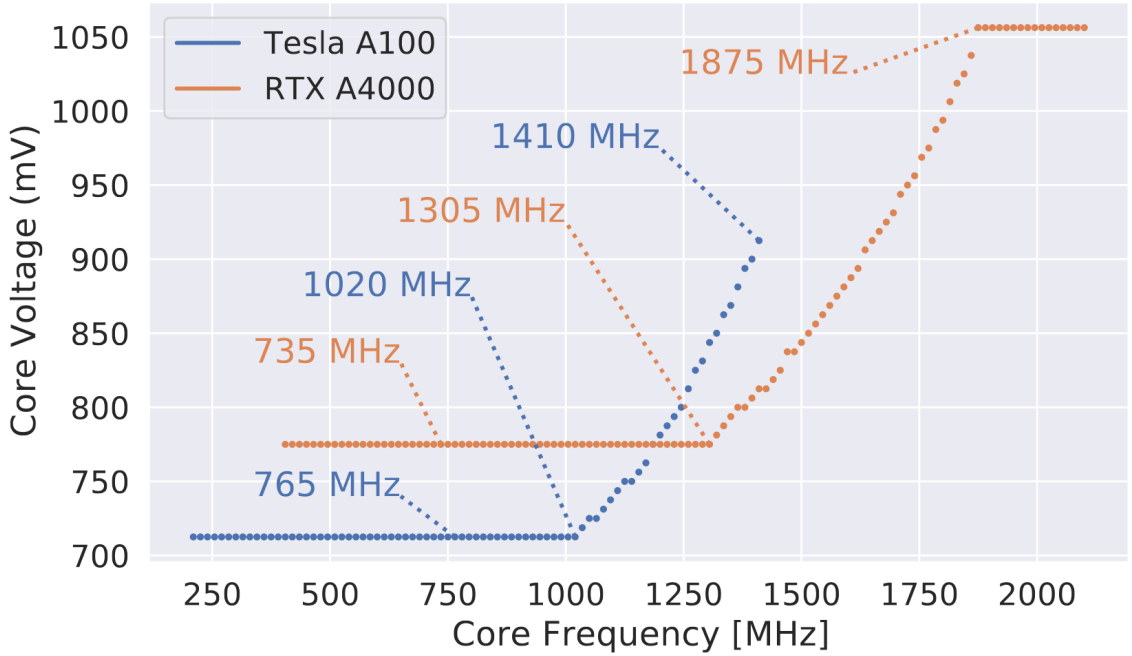
- Especially on A100, frequency tuning enables a wider power range
- Fixing the clock frequency also improves measurement stability

Frequency-power relation



- GPUs rapidly ramp up voltage when clock frequency increases beyond a certain point
- This point appears to be a sweet spot in the trade-off between energy consumption and compute performance
- We call this point the ‘ridge point’

Observation: base clock, ridge point, max



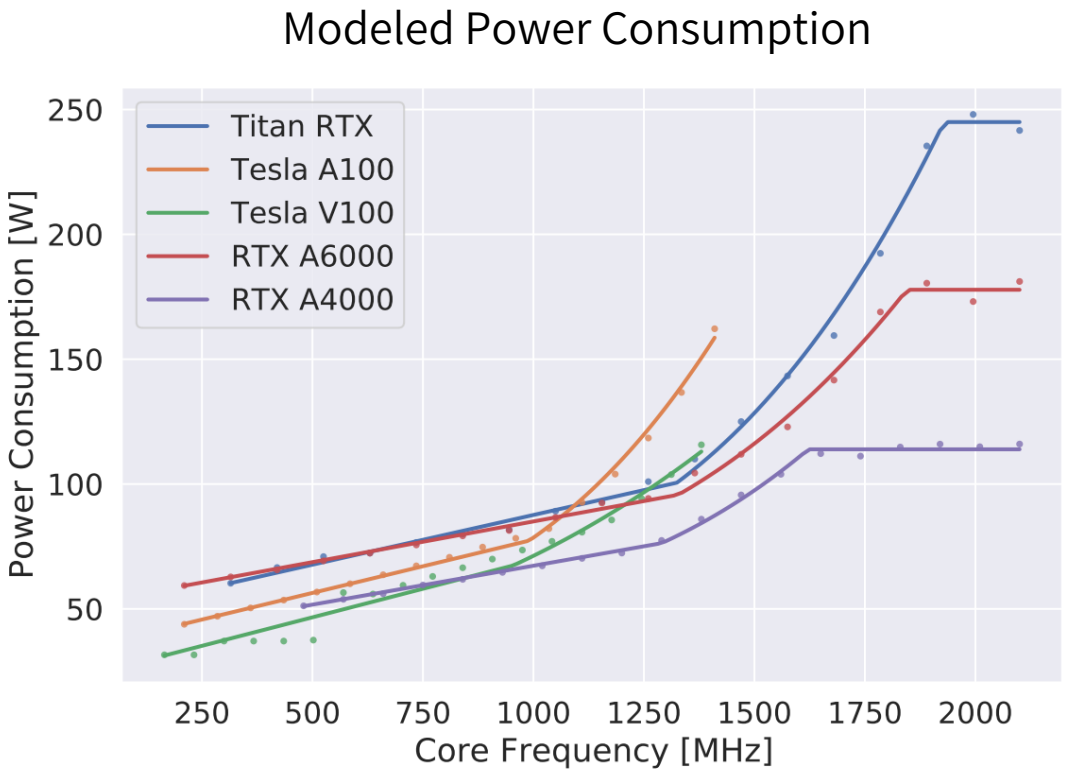
Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

A simple power consumption model

- Not every GPU reports core voltages, but we can estimate the voltage using a simple power model
- When we fix all parameters and vary the clock frequency, we can approximate power consumption using:

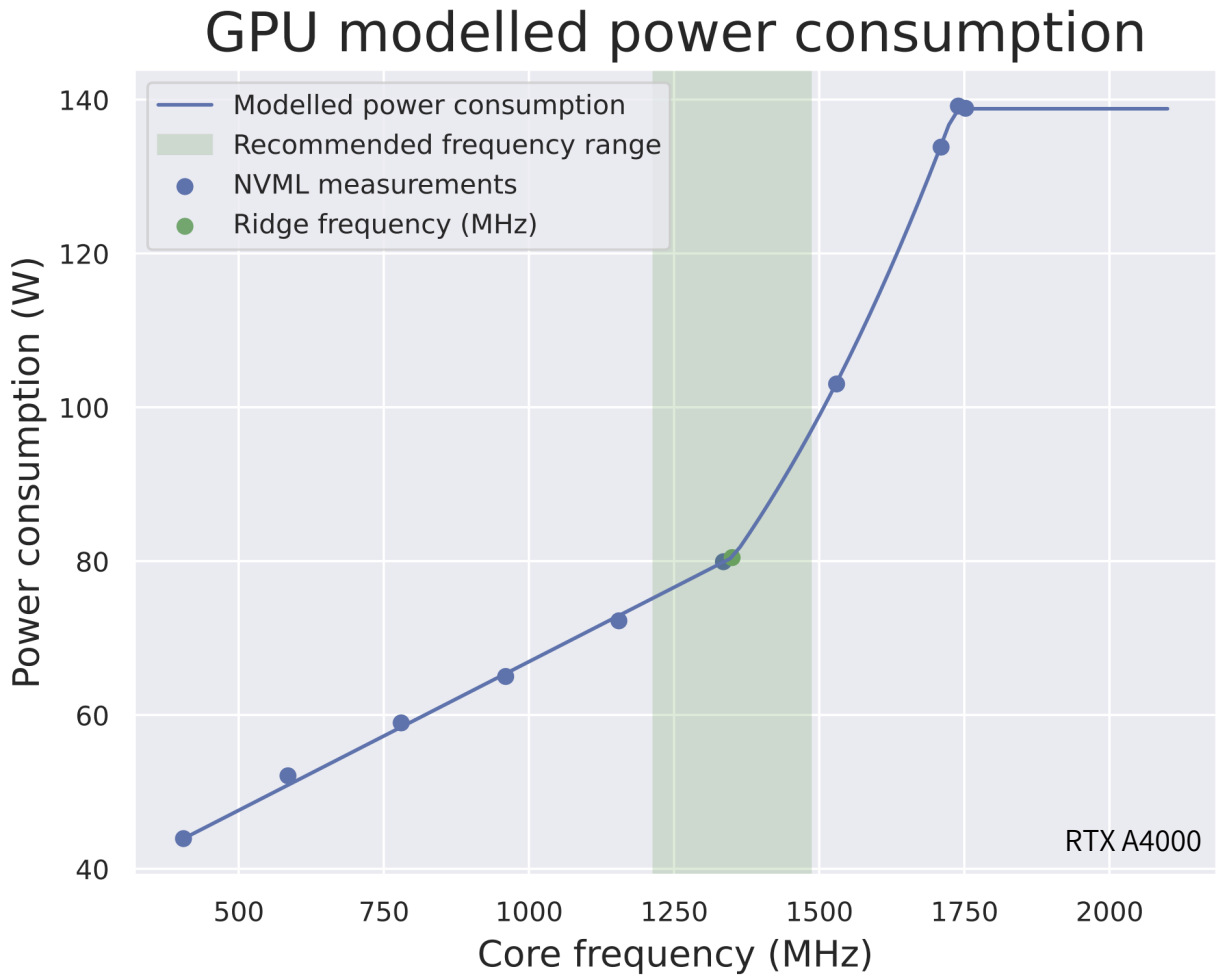
$$P_{load} = \min(P_{max}, P_{idle} + \alpha * f * v^2)$$

- And identify the GPUs 'ridge point' frequency in this way



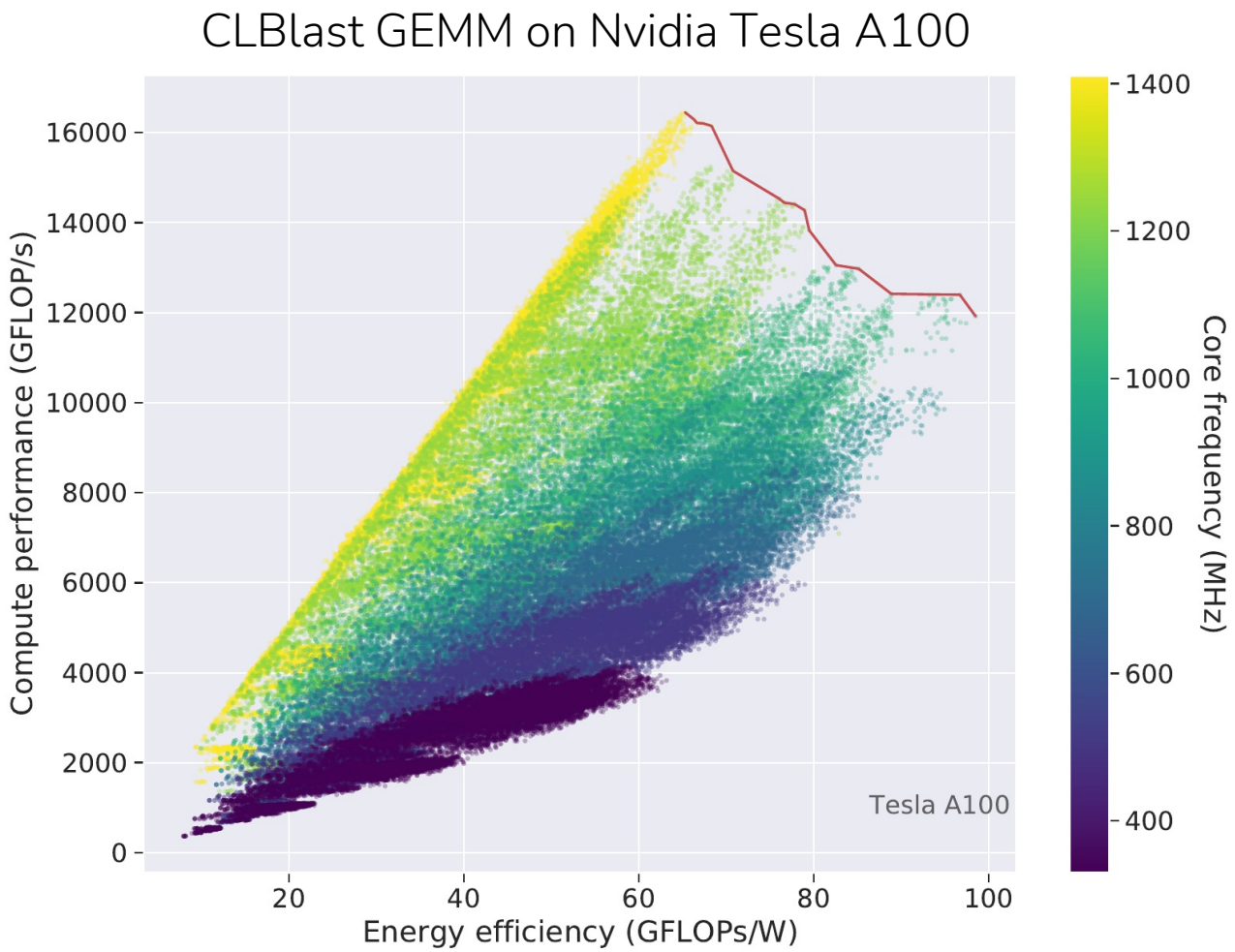
Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

- Use performance model to limit the frequency range for tuning
- Support implemented in Kernel Tuner 0.4.4
- Reduces the search space by ~80% on average



https://github.com/KernelTuner/kernel_tuner/blob/master/examples/cuda/going_green_performance_model.py

Practical energy gains



We can gain 50.9% energy efficiency, by trading in 27.5% performance

Future research directions

- Improving NVML power measurement accuracy and stability
- Multi-objective optimization
- AMD support: HIP/ROCm
- Tuning accuracy and mixed-precision to explore GPU energy-accuracy trade-offs

Auto-tuning Energy Efficiency: overview

- GPU Energy measurements:
  - NVML (slower, but highly available), PowerSensor2 (faster, but assembly required)
- Energy is clearly a different optimization objective, distinct from time
- Model-steered auto-tuning can effectively optimize GPU energy efficiency
  - On average gain 42.0% energy efficiency, trading in 24.3% compute performance

- All presented methods available in Kernel Tuner:
  - http://github.com/KernelTuner/kernel_tuner
- Full paper:
  - *Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning*
    R. Schoonhoven, B. Veenboer, B. van Werkhoven, K. J. Batenburg
    International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) at Supercomputing (SC22) 2022
  - https://arxiv.org/abs/2211.07260

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