Green and Efficient Data Science

Alessio Sclocco, Stijn Heldens

netherlands eScience center

Introduction: Alessio Sclocco

Research Software Engineer @ Netherlands eScience Center

a.sclocco@esciencecenter.nl

Background:

- 2011-2012 researcher at VU Amsterdam
  - Working on GPUs for radio astronomy
- 2012-2017 PhD "Accelerating Radio Astronomy with Auto-Tuning" at VU Amsterdam
- 2015-2016 scientific programmer at ASTRON, the Netherlands Institute for Radio Astronomy
  - Designing and developing a real-time GPU pipeline for the Westerbork radio telescope
- 2019 visiting scholar at Nanyang Technological University in Singapore
  - Real-time tracking of social insects
- 2017-now Research Software Engineer at the Netherlands eScience Center
  - Radio astronomy, climate modeling, biology, natural language processing, high-energy physics, medical imaging

Introduction: Stijn Heldens

- Stijn Heldens (s.heldens@esciencecenter.nl)
  - Research Software Engineer (RSE)
  - Research interests in HPC include:
    GPU programming, parallel algorithms,
    distributed systems, and scalable programming models.
  - background:
    - 2022-now, Research Software Engineer at Netherlands eScience Center
    - 2018-2022, PhD on scalable programming models
    - 2016-2017, Researcher at University of Twente
    - 2015-2016, Researcher at Delft University of Technology
    - 2012-2015, MSc Computer science at VU University Amsterdam

Outline for the day

13:00	–	13:30	Walk-in
13:30	–	13:35	Introduction by the organizers
13:35	–	13:50	Introduction to Graphics Processing Units and GPU programming
13:50	–	15:00	Using CuPy and Numba to accelerate your Python code
15:00	–	15:15	Coffee break
15:15	–	16:00	Introduction to CUDA
16:00	–	16:30	Optimizing code with auto-tuning
16:30	–	16:55	Energy-efficient computing
16:55	–	17:00	Wrap-up

Acknowledgments

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.

Introduction to GPU Computing

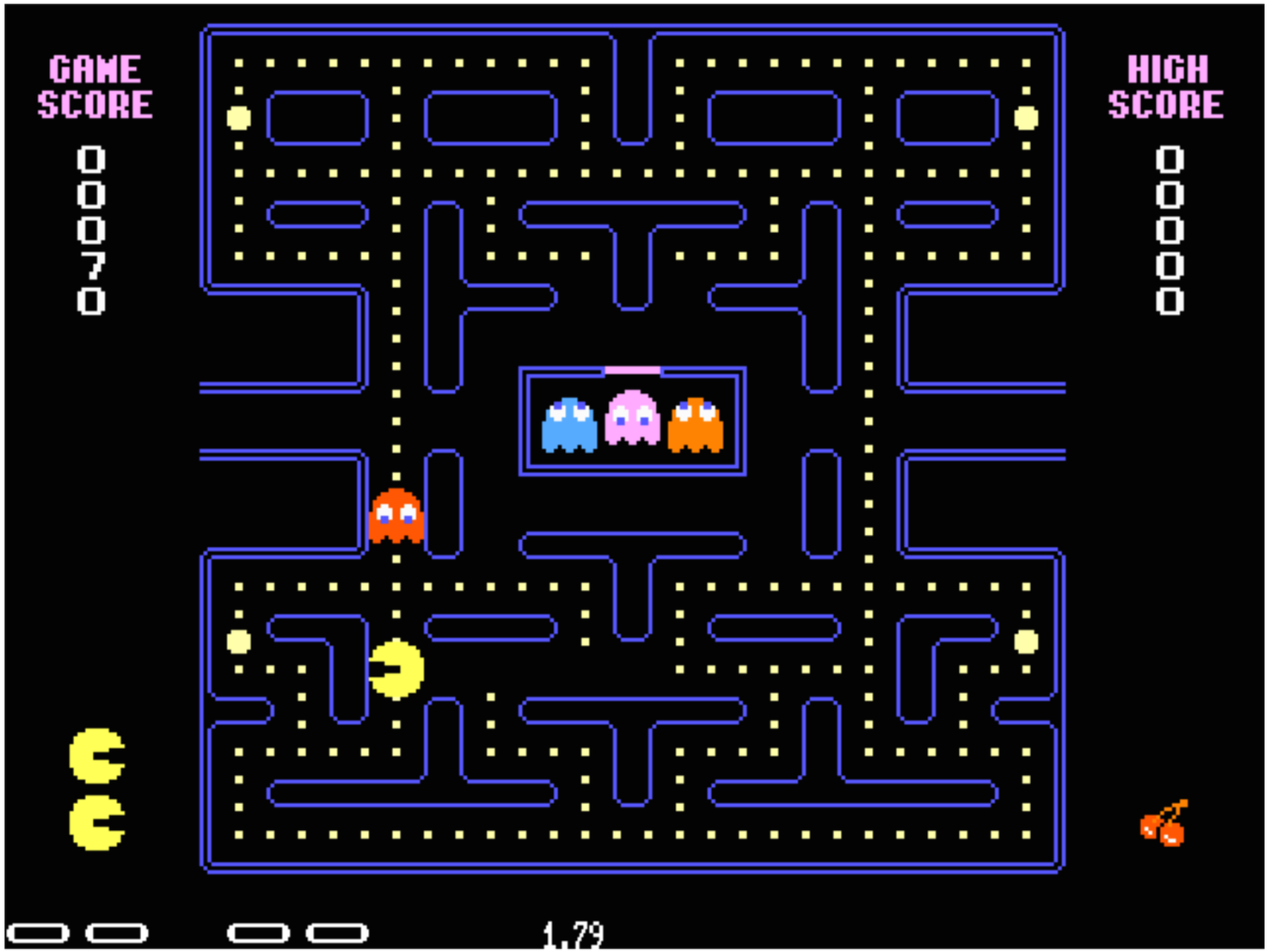
netherlands eScience center

What is a GPU?

- Graphics Processing Unit –
  The computing chip on a graphics card

- GPGPU – General Purpose computing on GPUs
  - Using GPUs for more than graphics

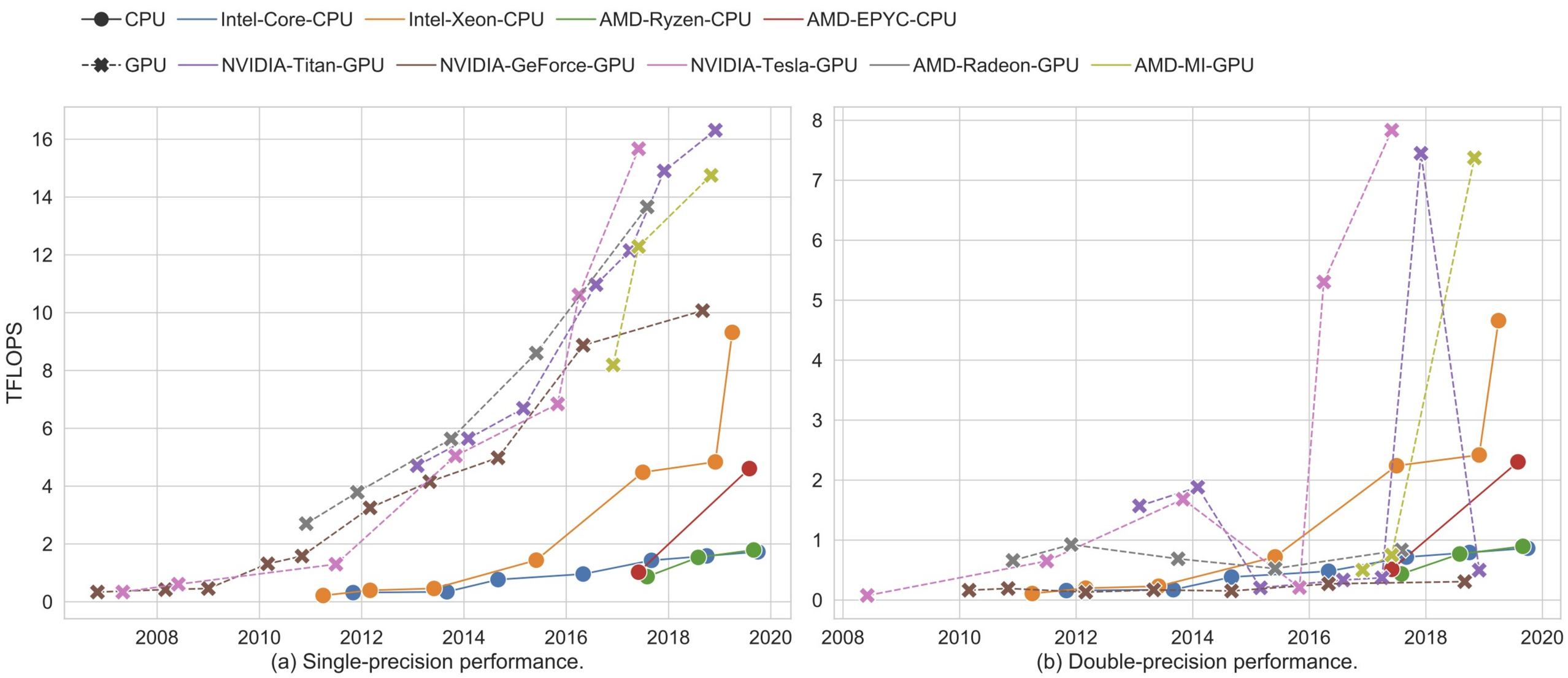
Graphics in 1980



Graphics in 2000

Graphics today

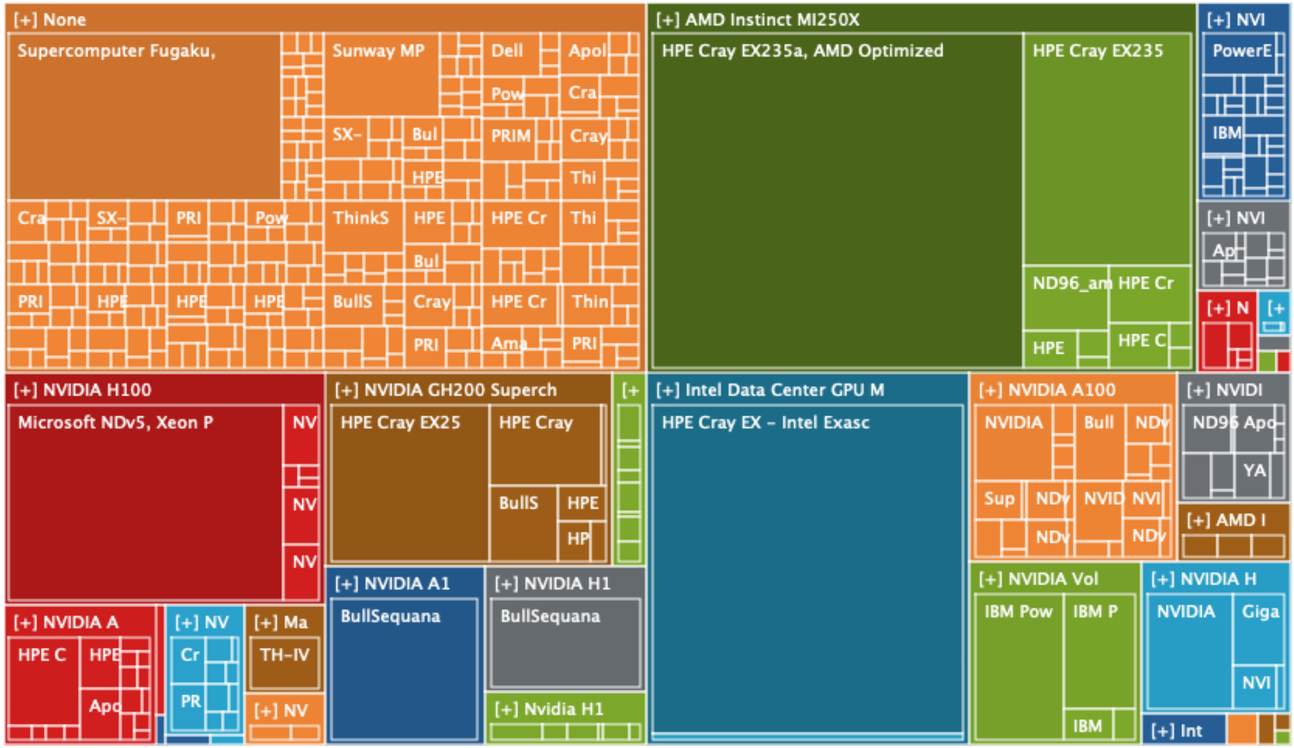
Performance Comparison: CPUs vs GPUs



Summarizing CPU and GPU Design Trends with Product Data
Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli, 2020

GPUs in the high-performance computing landscape

- Graphics Processing Units (GPUs) are widely used to accelerate HPC applications
- On the June 2024 edition of the Top500 list:
  - The fastest supercomputer (Frontier) have GPUs
  - The two (known) exascale machines (Frontier, Aurora) have GPUs
  - The top 3 systems have GPUs
  - 9 of the top 10 systems have GPUs



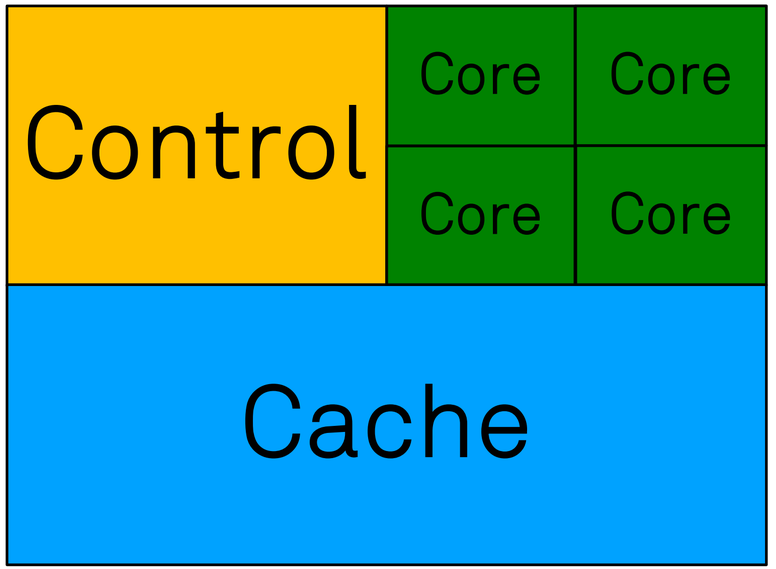
The World's Fastest Supercomputer: Frontier

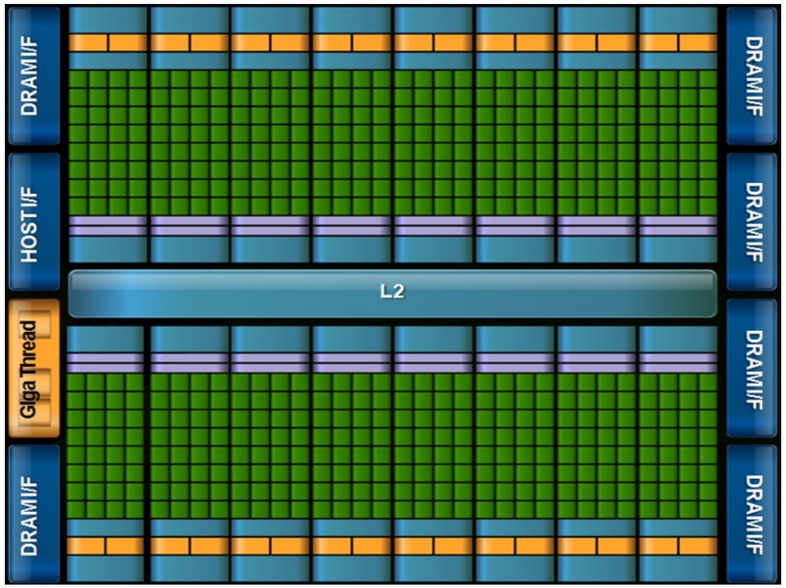
Number 1 in TOP500 list (Jun 2024)

- Peak compute performance:
  - 1.6 ExaFLOPS = 1.6×10^{18} floating point operations per second

- 9472 nodes with:
  - 1 CPU (AMD EPYC 64C)
  - 4 GPUs (AMD Instinct MI250X)

- Different goals produce different designs
  - GPU assumes workload is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: minimize latency experienced by 1 thread
  - Big on-chip caches
  - Sophisticated control logic
- GPU: maximize throughput of all threads
  - Multithreading can hide latency, so no big caches
  - Control logic
    - Much simpler
    - Less: share control logic across many threads



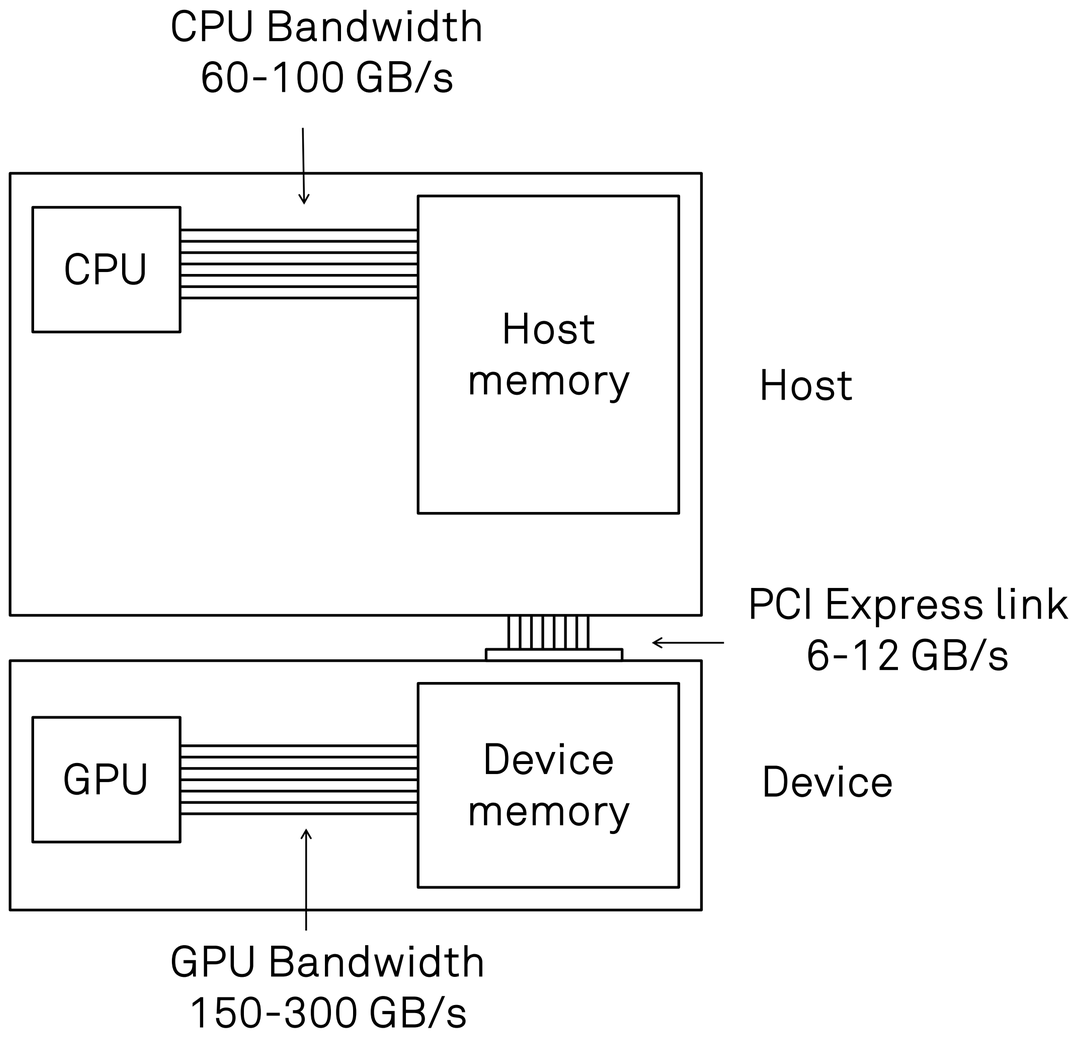


The computer architecture is very different:

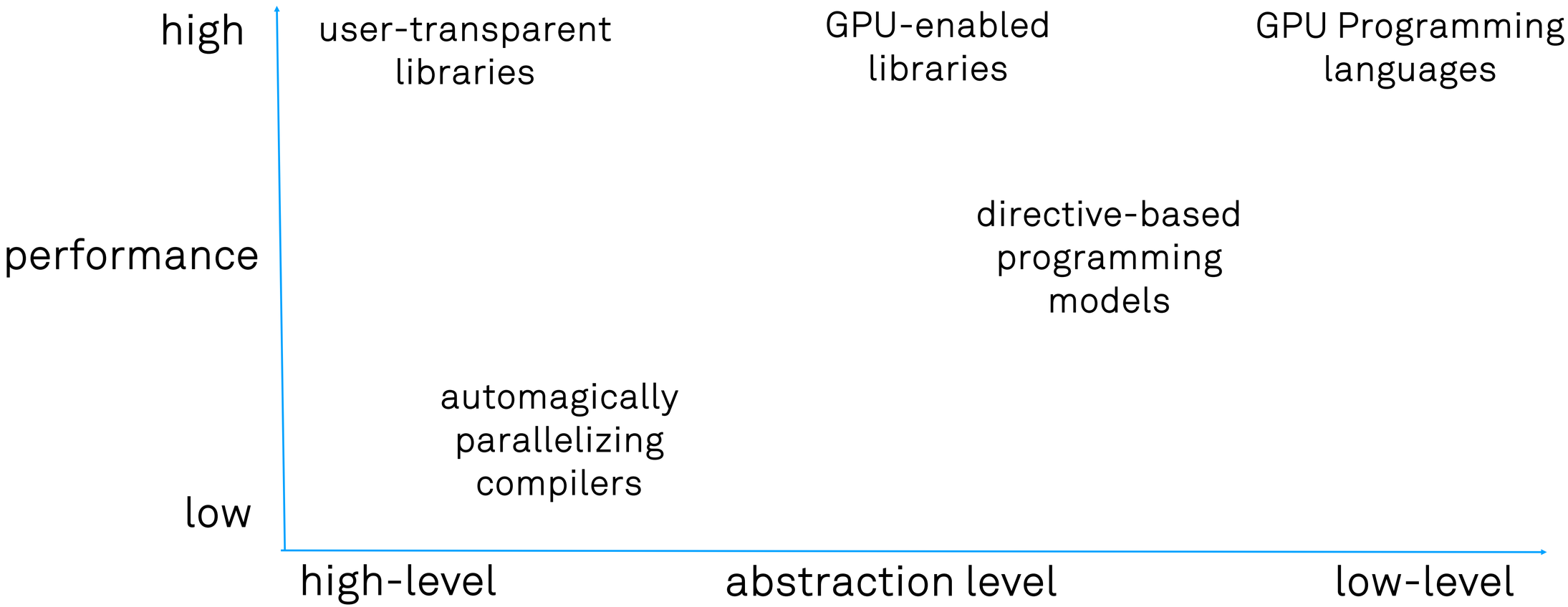
- Algorithms need to be parallelized and mapped to the hardware
- Requires software to be rewritten in specialized programming language
- Optimizing for compute performance requires knowledge about hardware

GPUs are on separate devices:

- Have to deal with separate memory space, limited bandwidth between host and device memory



GPU programming techniques



Optimizing code with auto-tuning

netherlands eScience center

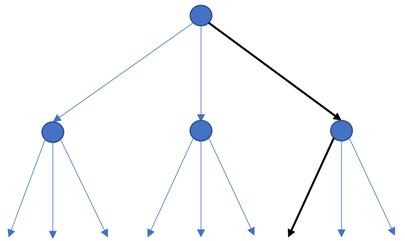
To maximize GPU code performance, you need to find the best combination of:

- 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
- …

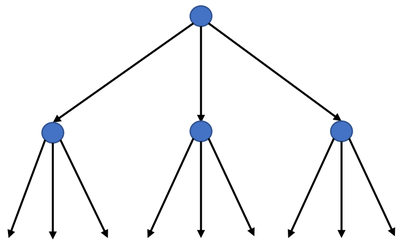
Problem:

- Creates a very large search space

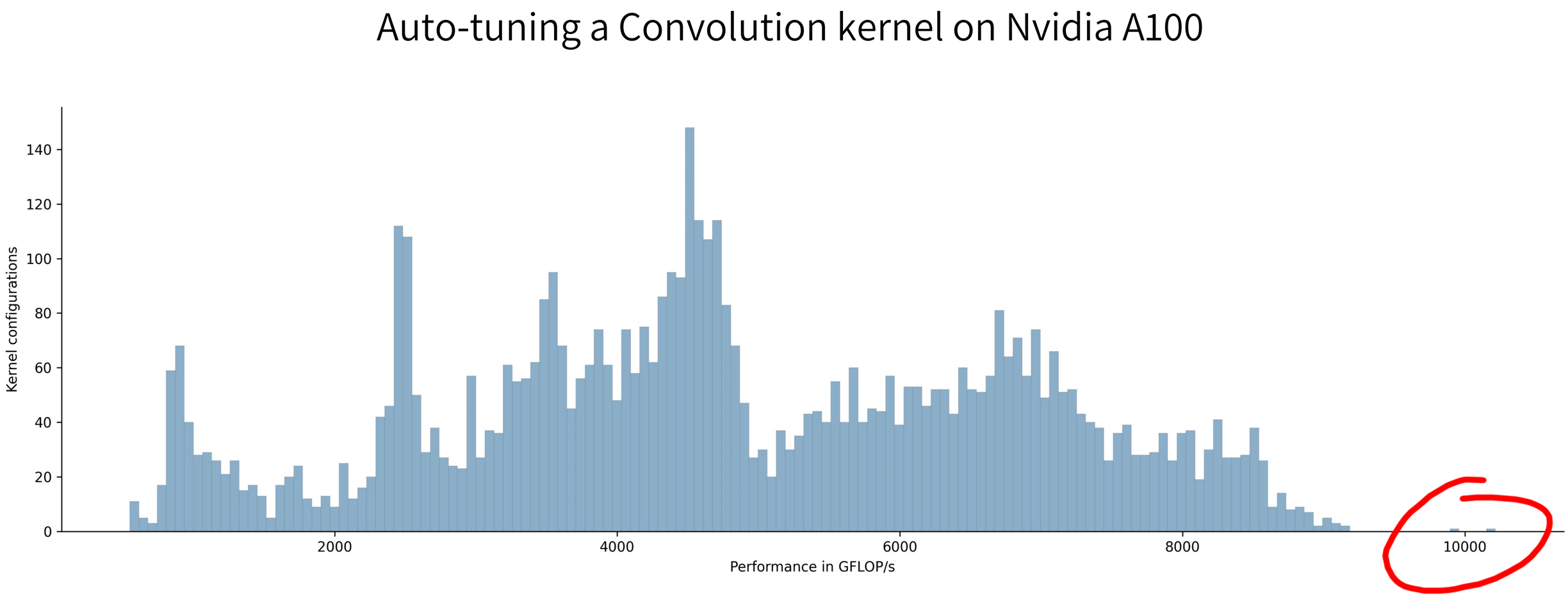
- Optimizing code manually, you iteratively perform:
  - Modify the code
  - Run a few benchmarks
  - Revert or accept the change



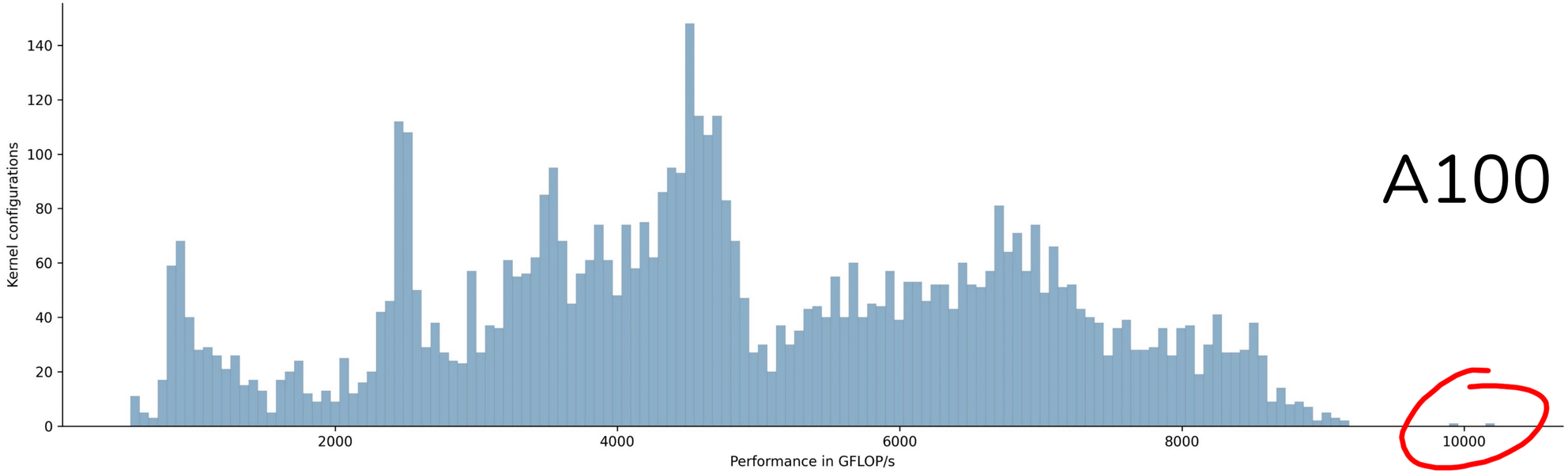
- With auto-tuning you:
  - Write a templated version of your code or a code generator
  - Benchmark the performance of all code variants



Large search space of kernel configurations



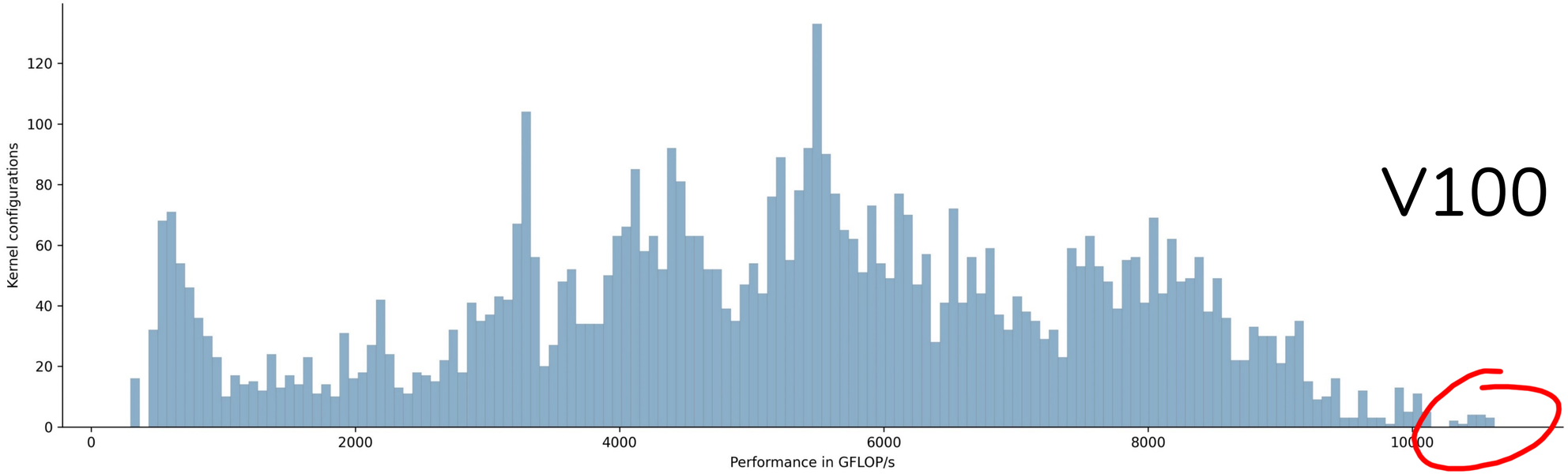
On different GPUs ...



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

A tool for automatic performance tuning of GPU kernels

- Developed open source (Apache 2.0)
- Funded by several national (NL) and European projects
- Used by 10+ eScience center projects, and 10+ other universities & organizations
- Supports:
  - CUDA, HIP, OpenCL, C, Fortran, OpenACC
  - 20+ search optimization algorithms
  - Energy measurement of GPU kernels
  - Many different use cases

netherlands eScience center · CWI · ASTRON · Universiteit Leiden The Netherlands

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

Choosing thread block dimensions

- 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]`

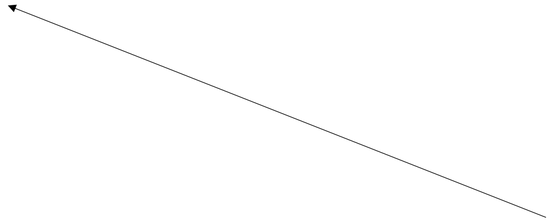
- You can use different names for these by passing the `block_size_names` option using a list of strings
- Note: when you change the thread block dimensions, the number of thread blocks used to launch the kernel generally changes as well

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)

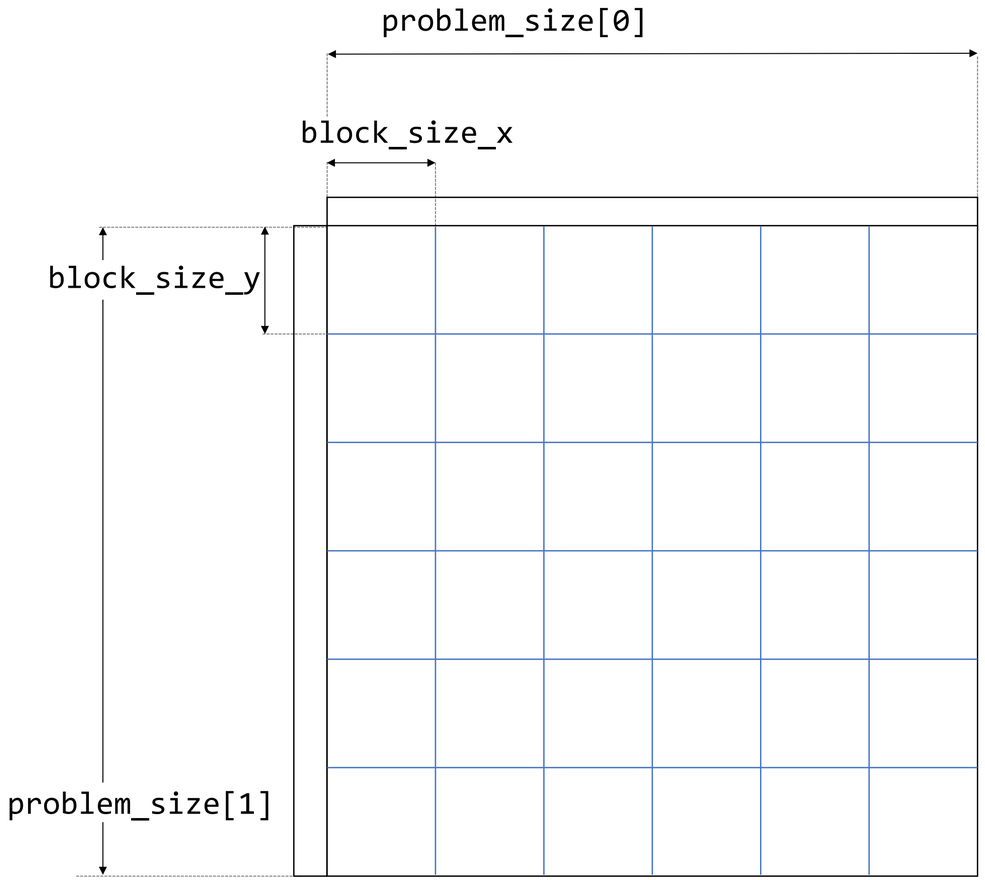
Vector add example

```
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`

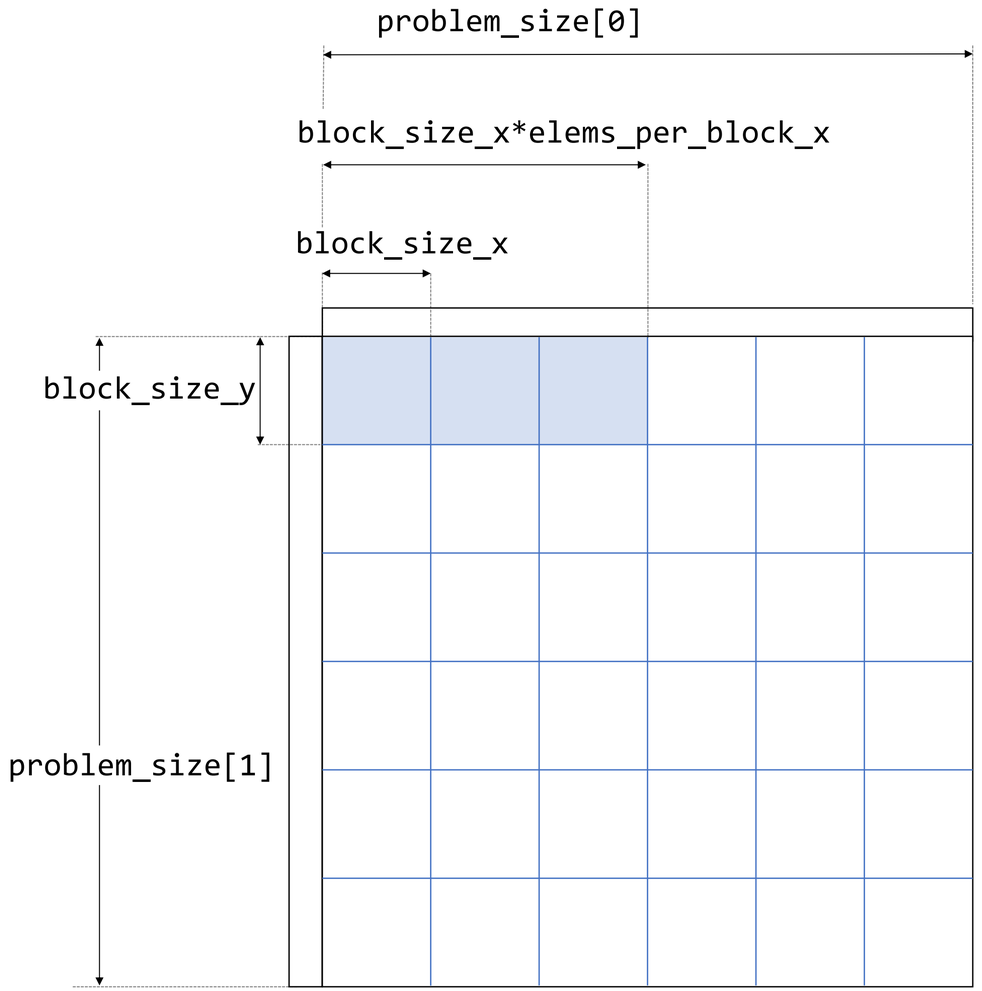
Specifying grid dimensions

- 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

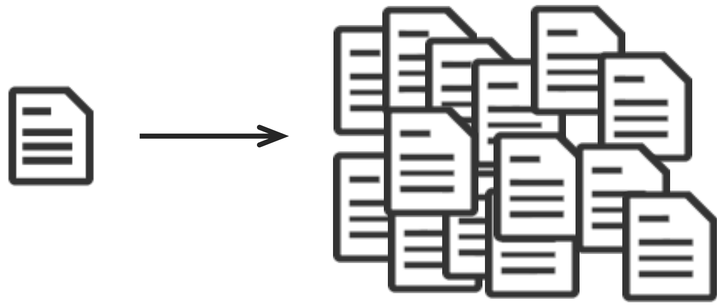
- Other parameters, or none at all, may also affect the grid dimensions
- Grid divisor lists control how `problem_size` is divided to compute the grid size
- Use the optional arguments:
  - `grid_div_x`, `grid_div_y`, and `grid_div_z`
- You may disable this feature by explicitly passing empty lists as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

restrict = ["loop_unroll_factor_x <= tile_size_x",
            "tile_size_x % loop_unroll_factor_x == 0"]

# pass our constraints 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

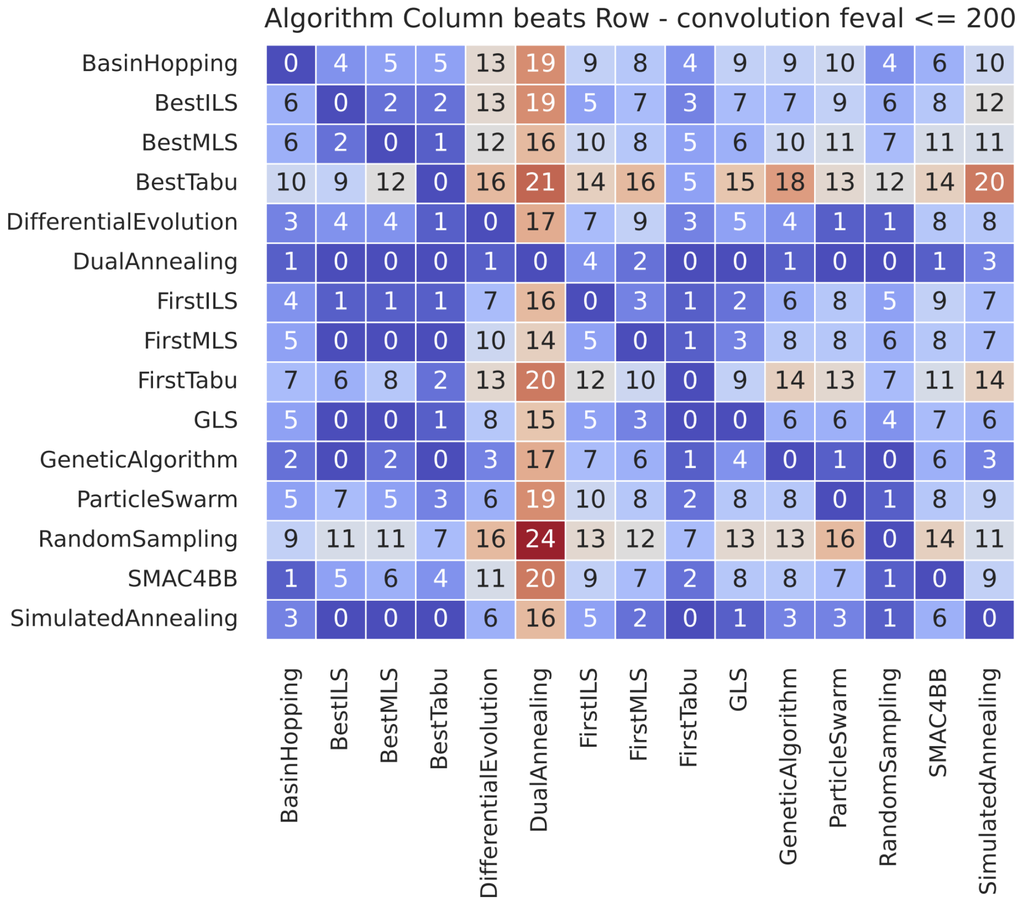
- Are composable, and therefore the order matters, so they are passed using a Python `dict`
- The `key` is the name of the metric, and the `value` is a function that computes it
- For example:

```
metrics = dict()
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

Hands-on Session

netherlands eScience center

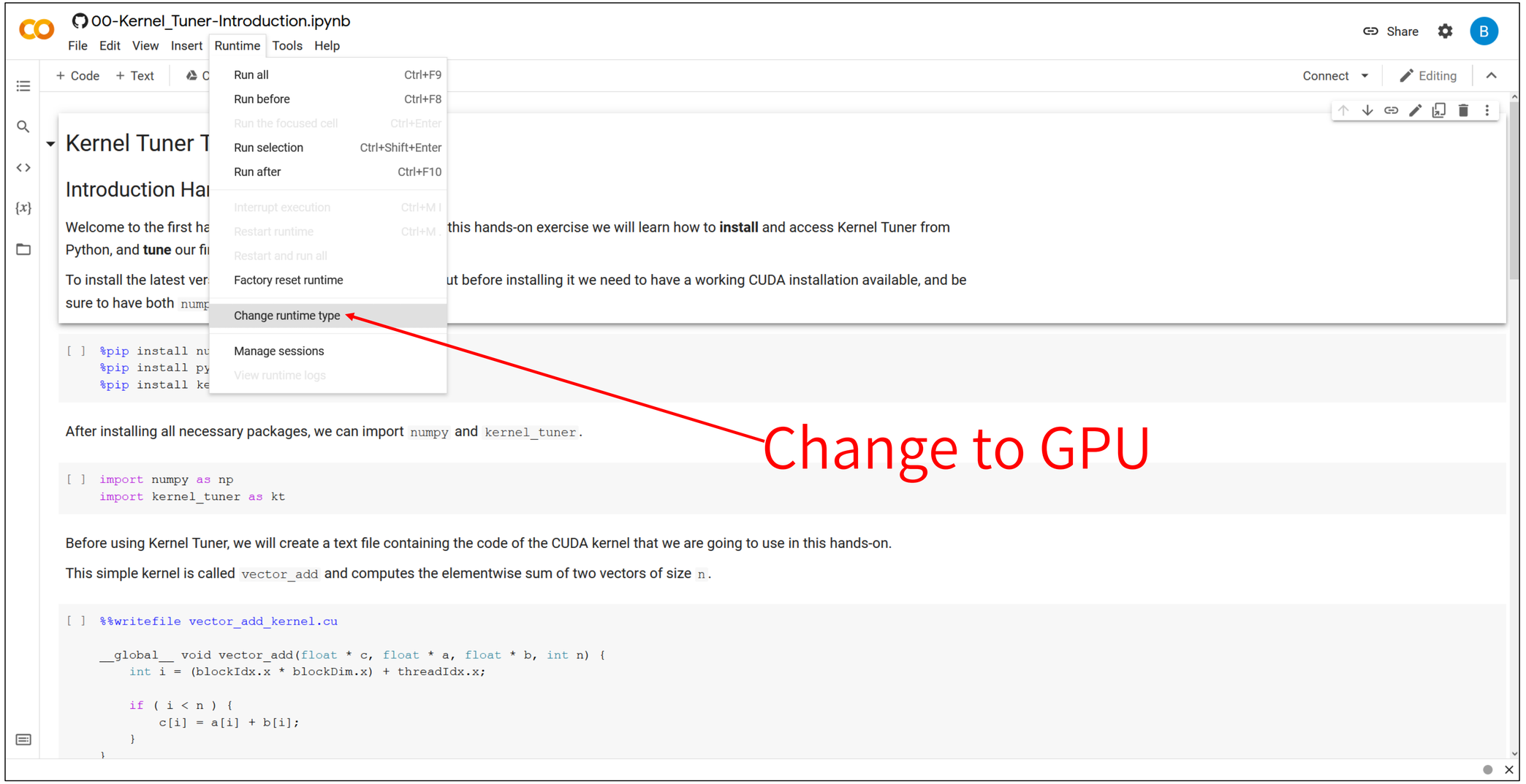
- The goal of this hands-on is to tune a Convolution CUDA kernel
  - Open the notebook in your Google Colab and work there
  - https://tinyurl.com/kerneltunertutorial

- 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

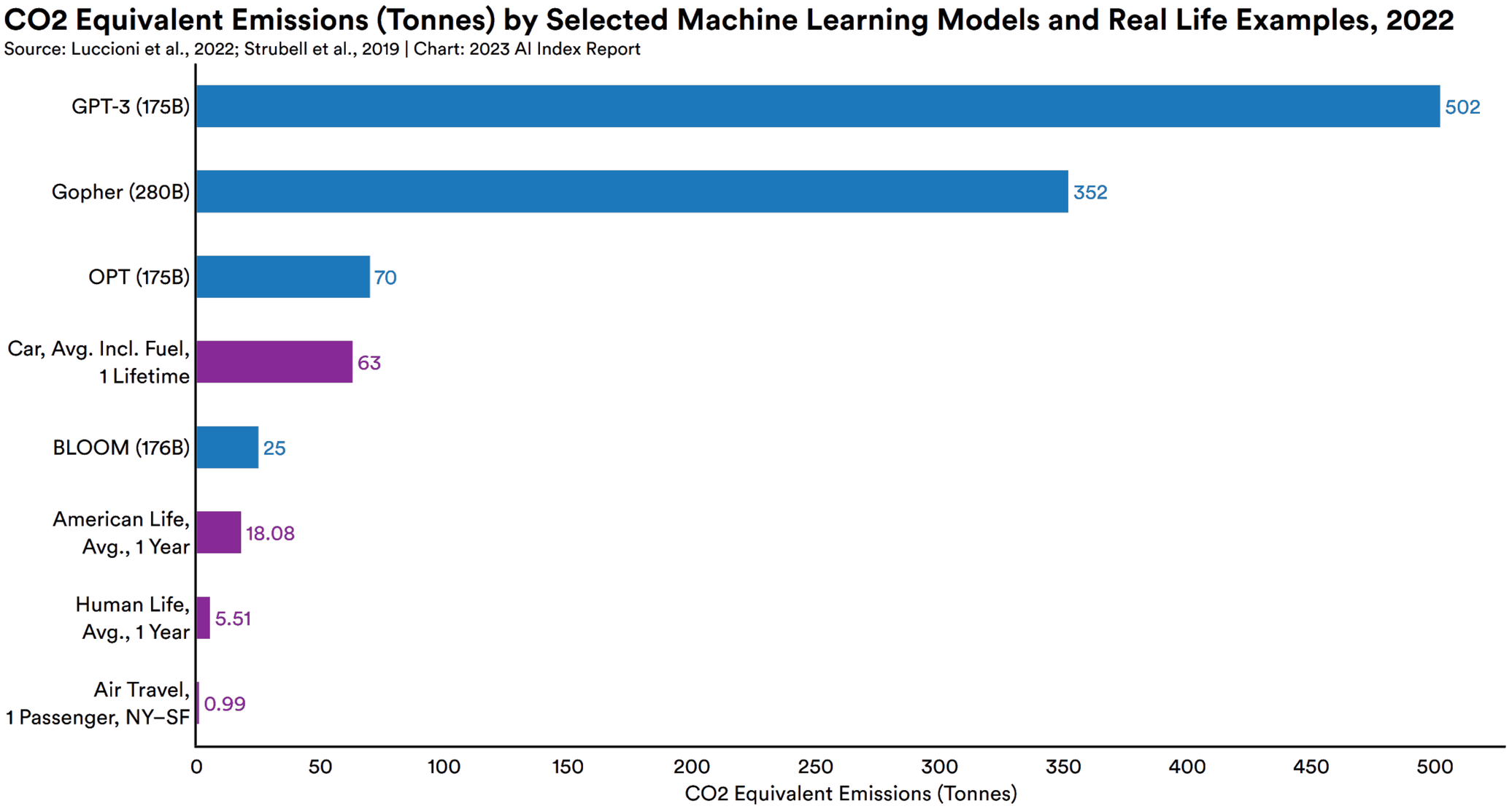
Change runtime type in Colab



Energy-efficient GPU Computing

netherlands eScience center

LLM Training emissions



Source: Stanford AI Index report 2023

What is 500 tons of CO2?

Roughly equal to:

- 8,268 tree seedlings grown for 10 years
- $80,000 in electricity bill
- 63 homes' energy use for a year in the US
- 111 passenger cars driving around for a year in the US

- Less than 2 days of running the Frontier supercomputer …

https://www.epa.gov/energy/greenhouse-gas-equivalencies-calculator

Energy cost of supercomputers

Frontier: #1 in TOP500 list (Jun 2024)

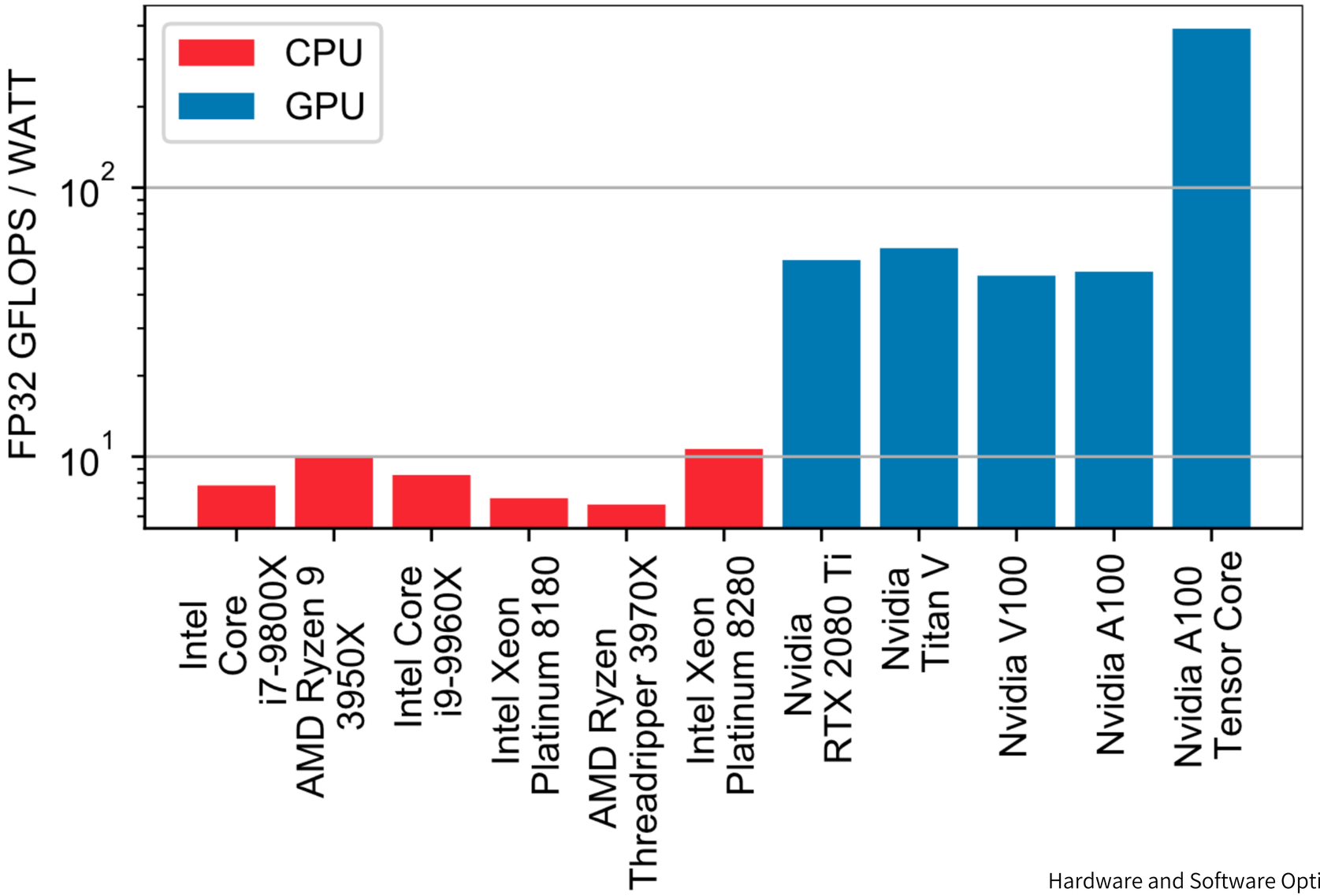
- #13 Green500 (Jun 2024)
- 20 Megawatt continuously
- $40 million annual electricity bill
- 100,000 metric tons of CO2 annually
- ~20,000 cars on the road for a year in US

Summit: (#9, Frontier's predecessor)

- 64% of energy is consumed by GPUs

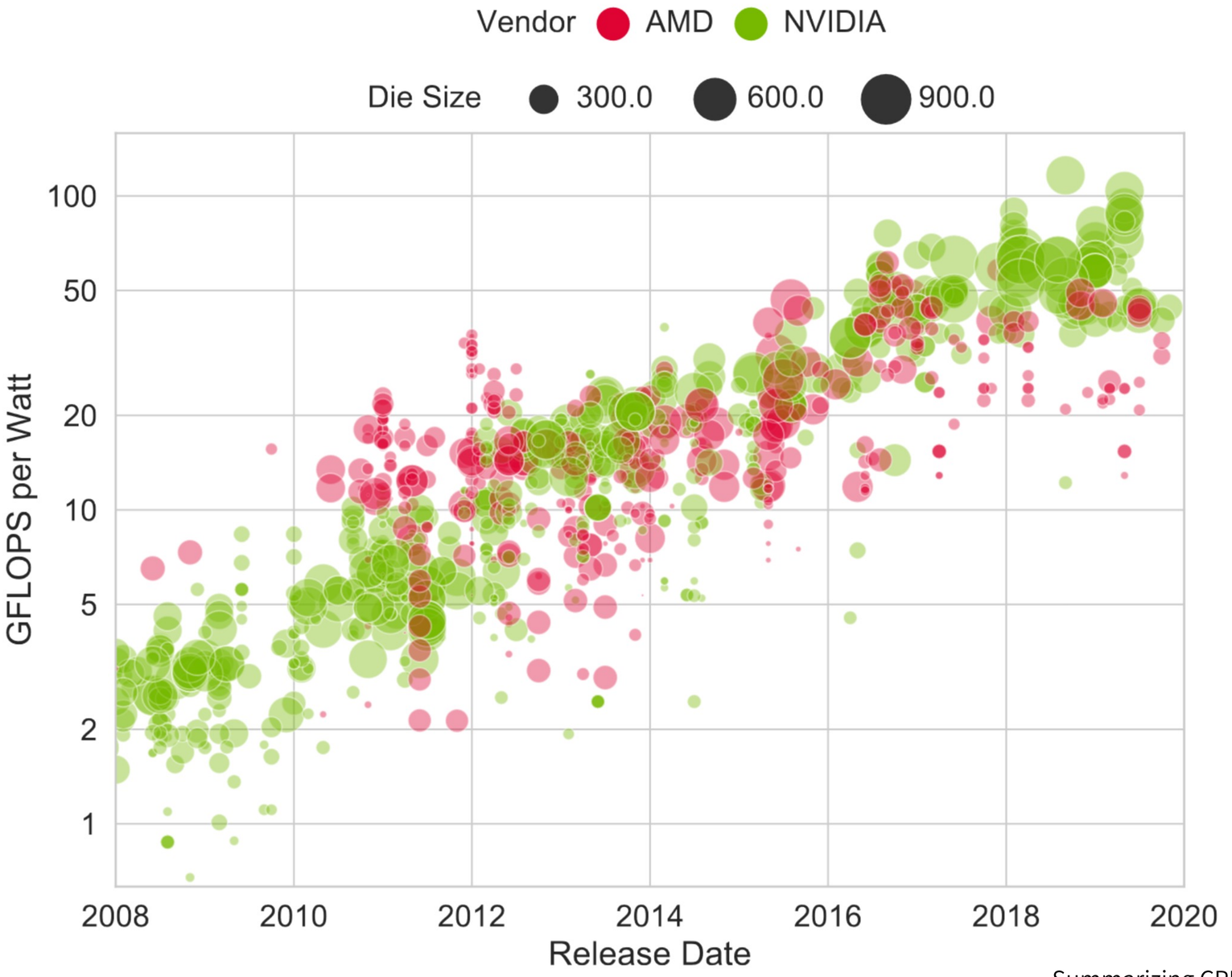
Efficient Computation through Tuned Approximation
David Keyes, SIAG/SC Supercomputing Spotlights 2022
Autotuning based on frequency scaling toward energy efficiency of blockchain algorithms on graphics processing units M. Stachowski, A. Fiebig, and T. Rauber, Journal of Supercomputing, 2020.

GFLOPs/W for different architectures



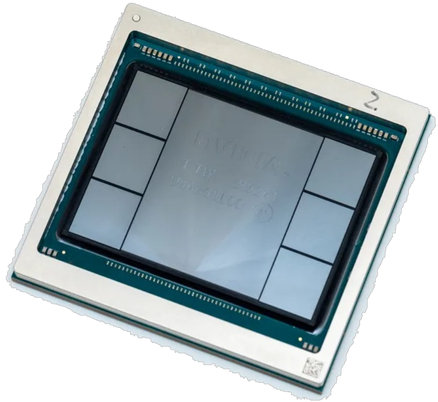
Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead
Capra et al. 2020 IEEE Access

Energy Efficiency of GPUs

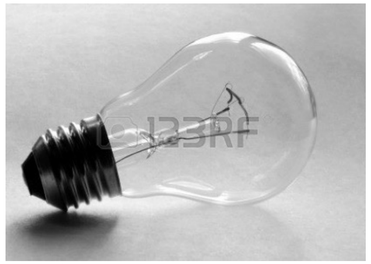


Summarizing CPU and GPU Design Trends with Product Data
Sun et al. 2020

- Nvidia H100 GPU:
  - Energy: 350 Watt
  - Surface: 8.14 cm^2
  - Heat dissipation: 43.0 $Watt/cm^2$

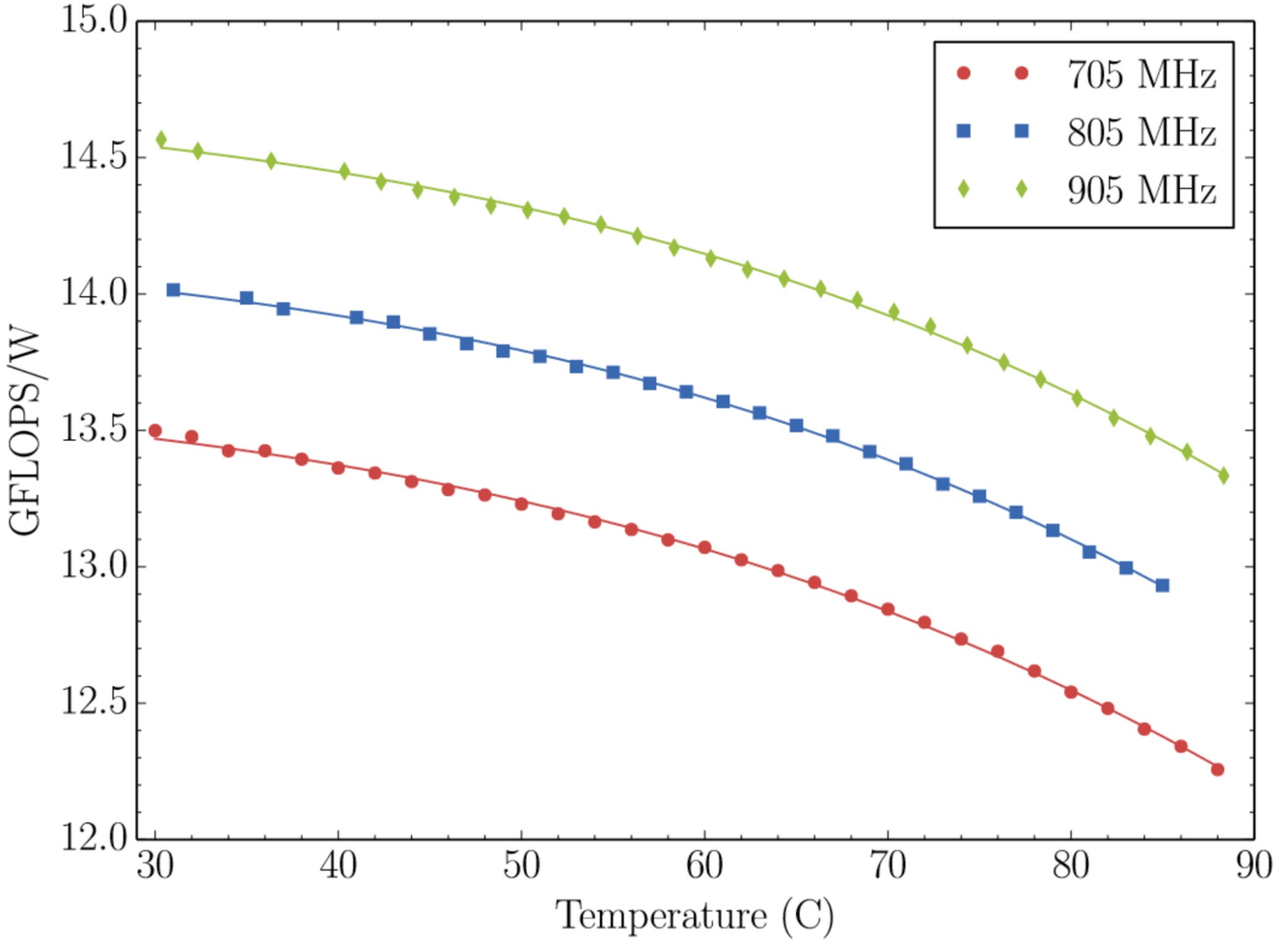


- Light bulb:
  - Energy: 100 Watt
  - Surface: 15 cm^2
  - Heat dissipation: 6.7 $Watt/cm^2$



- Electric cooker:
  - Energy: 1800 Watt
  - Surface: 1017 cm^2
  - Heat dissipation: 1.8 $Watt/cm^2$

GPU Temperature – Energy Efficiency relation

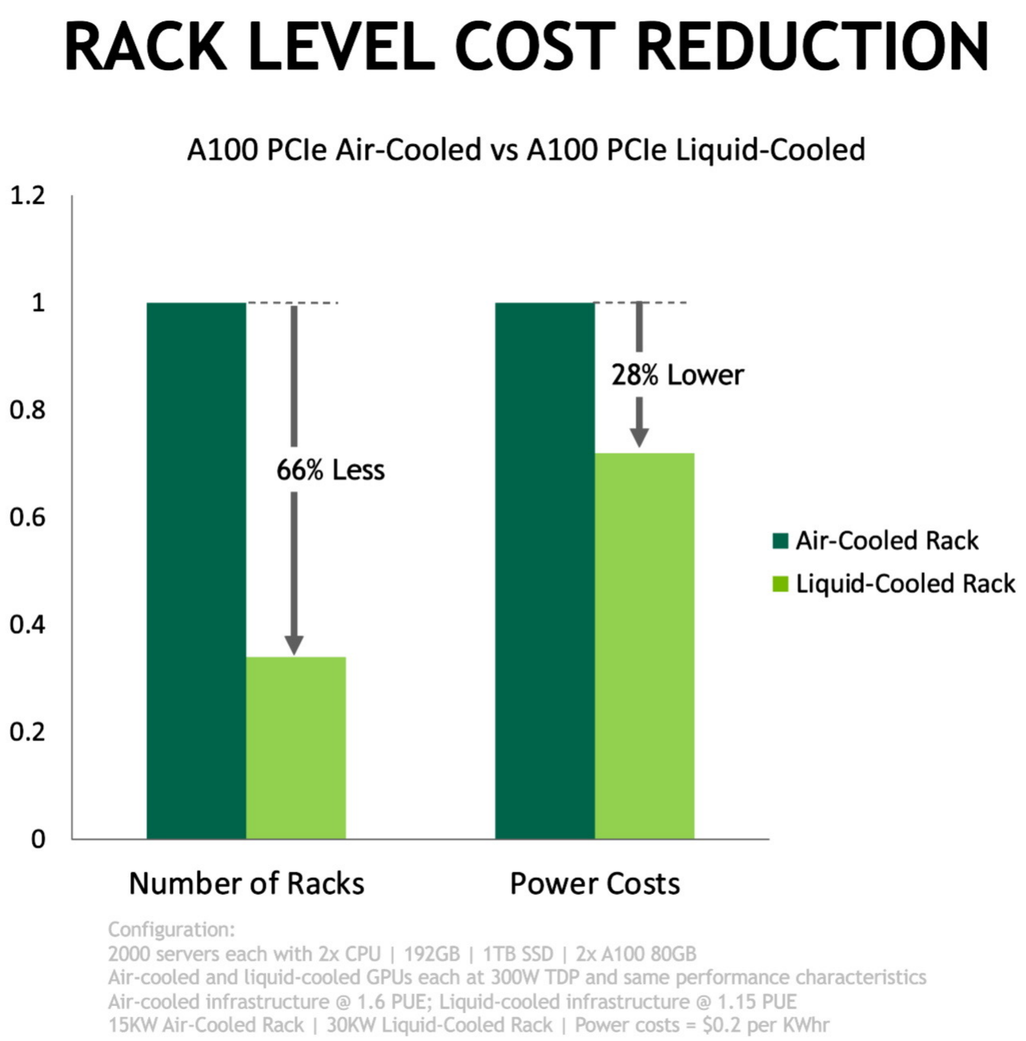


Hotter GPUs can be ~7% less energy efficient

Results obtained with xGPU (radio astronomy correlator) on Nvidia K20

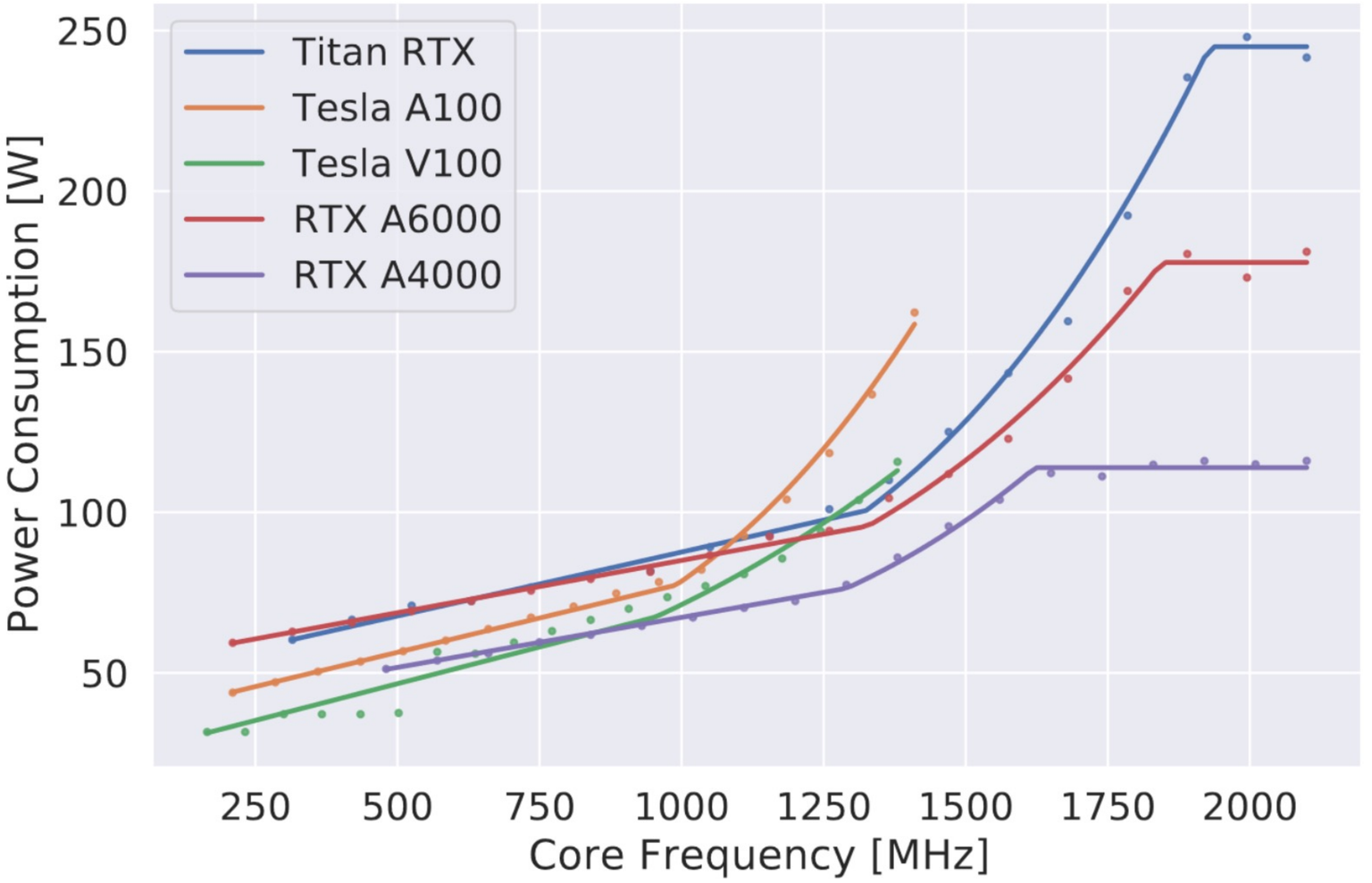
Optimizing performance-per-watt on GPUs in high performance computing
Price et al. 2016

- Liquid cooling is more energy friendly than air cooling
- But as the efficiency difference between hot and cold GPUs is ~7%, you probably shouldn't overdo the cooling



https://blogs.nvidia.com/blog/2022/05/23/liquid-cooled-gpus-computex/

Clock frequency power relation



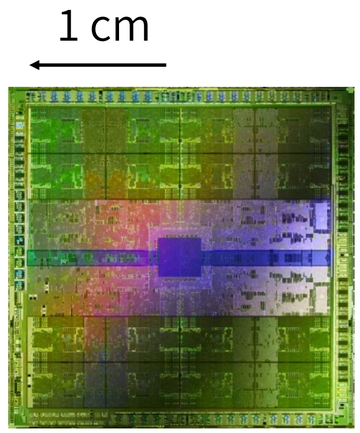
Going Green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
PMBS workshop at SC22 2022

- Moving data around is 20x more expensive than computing on it

Estimations for Nvidia H100:

```
mad.f64  %f1, %f2, %f3, %f0; // c += a*b;
```

- A single double-precision Fused Multiply-Add[1]: 13.7 pJ
- Moving the operands (4x 64-bits) for 10 mm within chip[2]: 294.4 pJ (21x more energy)



[1]Based on 25.6 TFLOP/s peak at 350 Watt TDP
https://www.nvidia.com/en-us/data-center/h100/

[2]Based on 115 femtojoule/bit/mm
GPUs and the Future of Parallel Computing
Keckler et al. 2011

Three strategies for energy efficient GPU Computing:

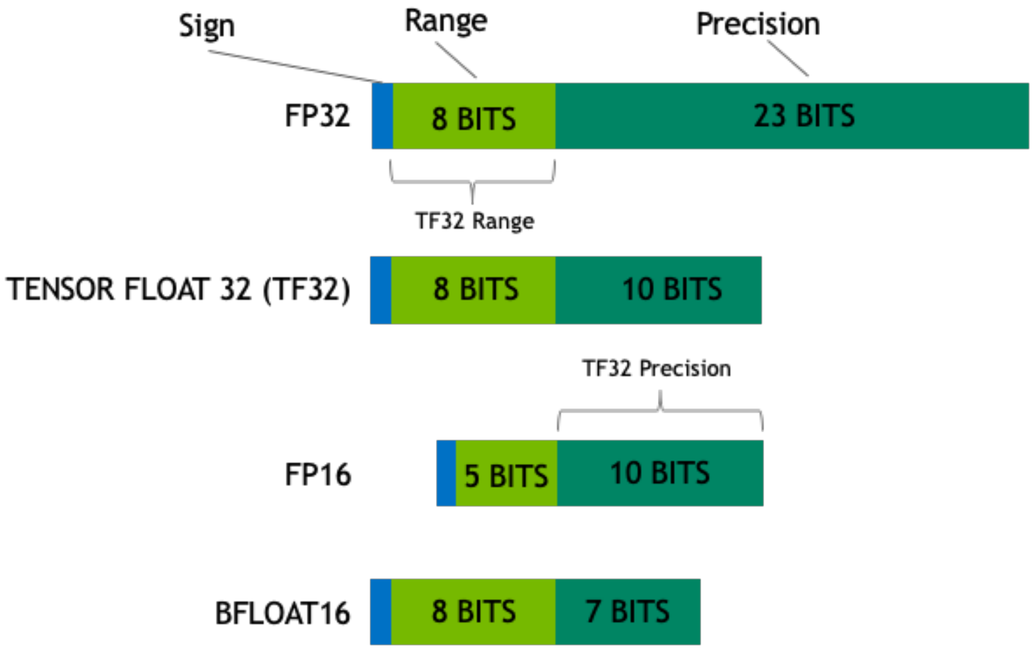
1. Use for shorter amount of time
2. Minimize data movements
3. Optimize device settings

Three strategies for energy efficient GPU Computing:

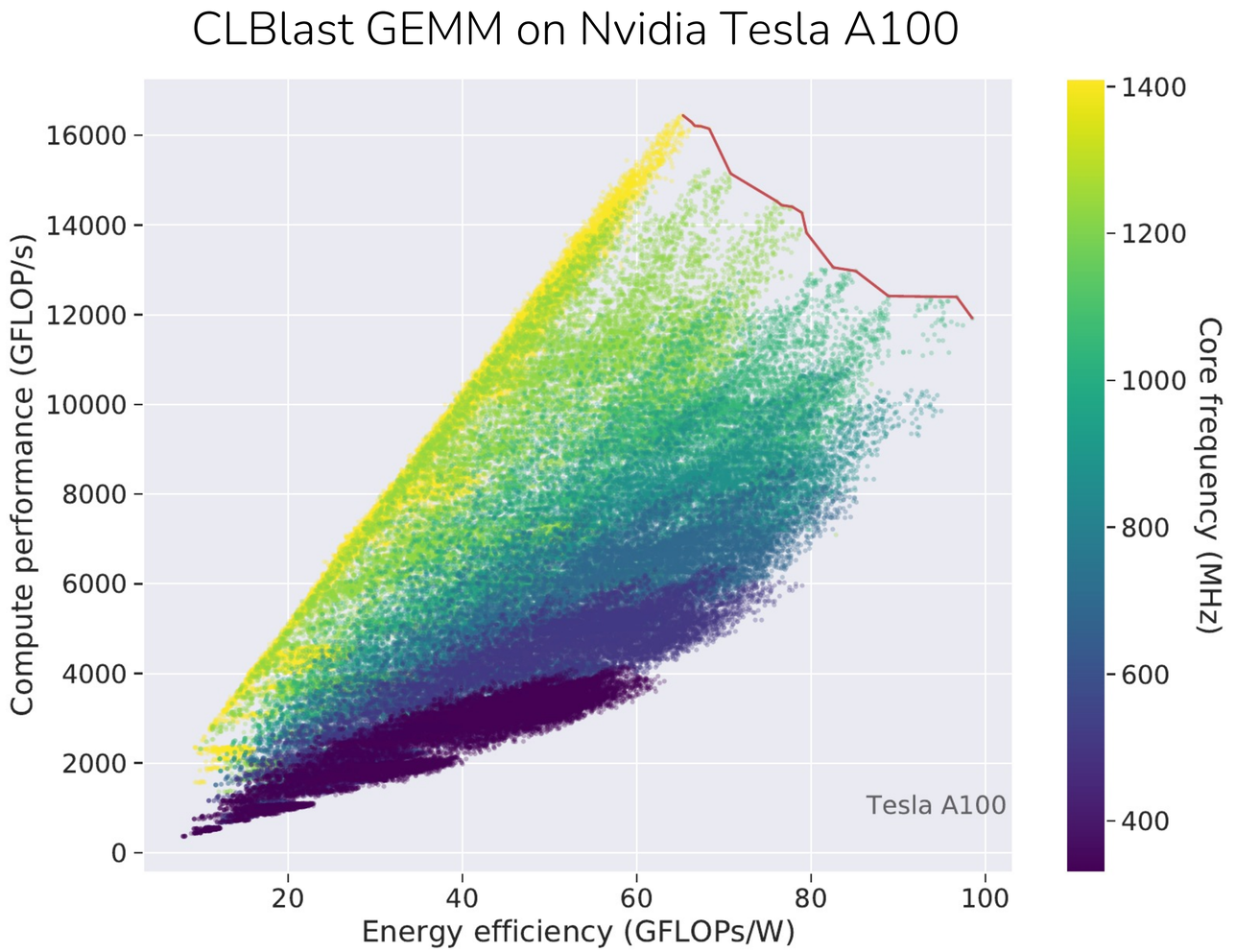
1. Use for shorter amount of time → Optimize application performance
2. Minimize data movements → Lower/mixed precision techniques
3. Optimize device settings → Optimize clock frequency

Mixed-precision arithmetic

- **Lower precision** gives reduction in compute and storage
  - Example: **double** to **half** precision gives **4x** improvement
  - At cost of increase in **error**
- **Core idea:**
  - What if we **mix different** precision levels in one application?
  - Different floating-point types for different variables in code
- Leads to **trade-off** between **accuracy** and **performance**
  - What precision should be used for each variable?
  - Ideally, we want **maximal performance** with **minimal error**
  - **Auto-tuning** to the rescue!



Optimize clock frequency



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