

Energy-Efficient GPU Computing

Ben van Werkhoven

Alessio Sclocco

Stijn Heldens

Floris-Jan Willemsen

netherlands
eSciencecenter



Universiteit
Leiden
The Netherlands

Introduction: Ben van Werkhoven

Ben van Werkhoven

Assistant Professor at LIACS, Leiden University

b.j.c.van.werkhoven@liacs.leidenuniv.nl



Research interests:

High-Performance Computing, GPU programming, automatic performance tuning (auto-tuning), optimization techniques, energy efficiency, mixed-precision and accuracy, performance modeling, concurrency and multi-threading, research software engineering

Background:

- 2010-2014 PhD “Scientific Supercomputing with Graphics Processing Units” at VU Amsterdam
- 2014-2023 Research Software Engineer at the Netherlands eScience Center
- 2023-now Assistant Professor at LIACS, Leiden University

Introduction: Alessio Sclocco

Research Software Engineer @ Netherlands eScience Center

a.sclocco@esciencecenter.nl

Background:

- 2011-2012 junior researcher at VU Amsterdam
 - Working on GPUs for radio astronomy
- 2012-2017 PhD "Accelerating Radio Astronomy with Auto-Tuning" at VU Amsterdam
 - Under the supervision of professors Henri Bal and Rob van Nieuwpoort
- 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
- 2017-2023 Research Software Engineer at the Netherlands eScience Center
 - Radio astronomy, climate modeling, biology, natural language processing, high-energy physics



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 candidate 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



Introduction: Floris-Jan Willemssen

- Floris-Jan Willemssen
 - PhD Candidate at LIACS (Leiden University) and Netherlands eScience Center
 - Research focusses on optimization algorithms in auto-tuning and efficient search space resolution
 - Get in touch via fjwillemssen.com or f.j.willemsen@esciencecenter.nl



13:30	–	13:35	Opening and welcome
13:35	–	14:00	Introduction Energy Efficient GPU Computing
14:00	–	14:15	Introduction to auto-tuning with Kernel Tuner
14:15	–	14:30	First hands-on session
14:30	–	15:00	Code optimization techniques for energy efficiency
15:00	–	15:30	Coffee break
15:30	–	15:45	Second hands-on session
15:45	–	16:15	Mixed precision programming techniques
16:15	–	16:30	Third hands-on session
16:30	–	16:45	Optimizing GPU core clock frequency
16:45	–	16:55	Fourth hands-on session
16:55	–	17:00	Closing remarks

- This tutorial is organized in four parts, each part consisting of a presentation followed by a hands-on exercise
- The slides are verbose on purpose, and you can use them as references while working on the exercises or after the tutorial
- We will use Google Colab for the hands-on sessions, so you don't need to have access to a GPU or install anything locally
 - But you are free to experiment on your own system
- Please download the latest version of slides and hands-on notebooks here:
 - https://github.com/KernelTuner/kernel_tuner_tutorial

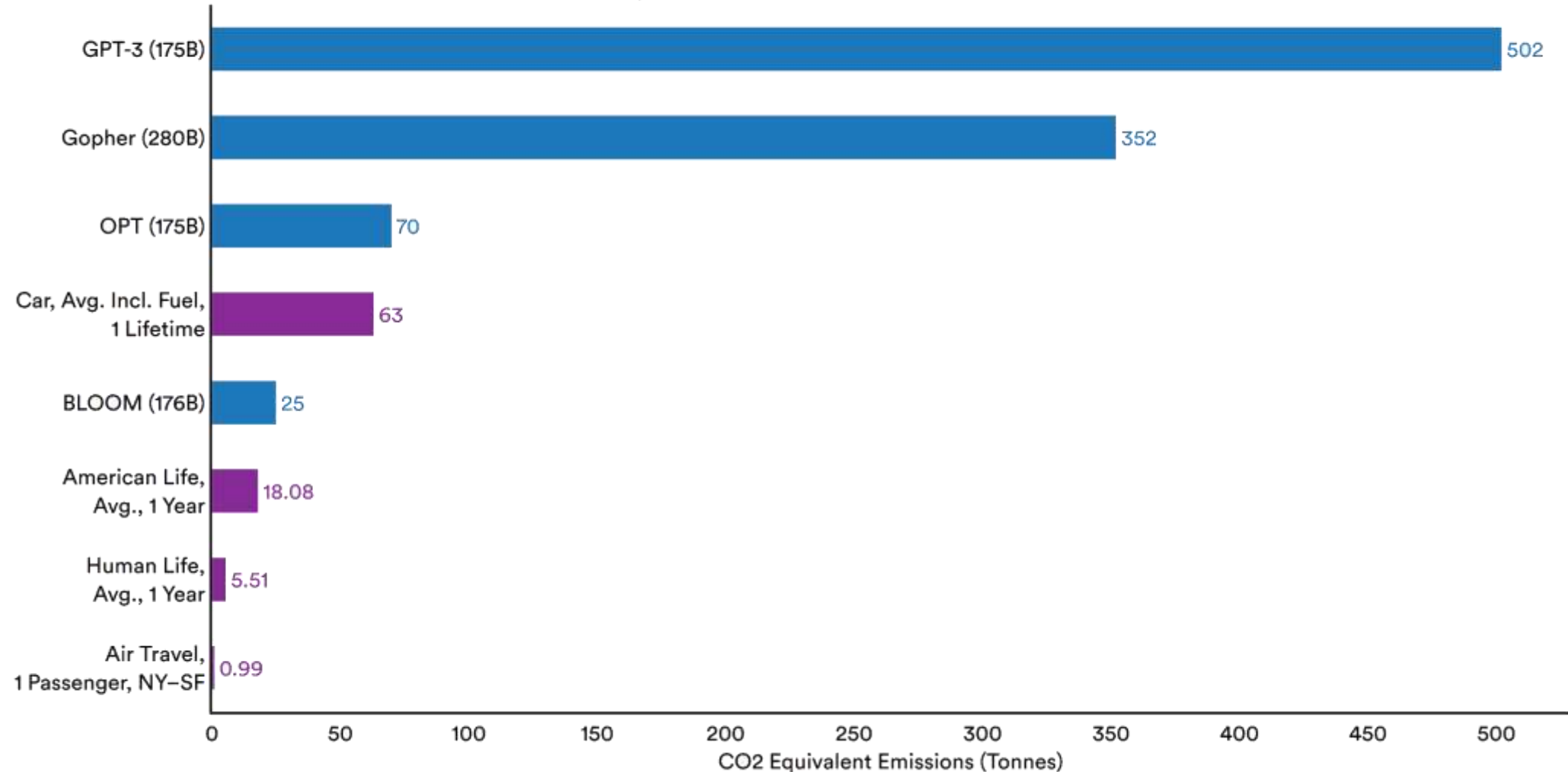
Energy Efficient GPU Computing



LLM Training emissions

CO2 Equivalent Emissions (Tonnes) by Selected Machine Learning Models and Real Life Examples, 2022

Source: Luccioni et al., 2022; Strubell et al., 2019 | Chart: 2023 AI Index Report



Source: Stanford AI Index report 2023

What is 500 tons of CO₂?

Roughly equal to:

- 8,268 tree seedlings grown for 10 years
- \$80000 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 ...



Energy cost of supercomputers

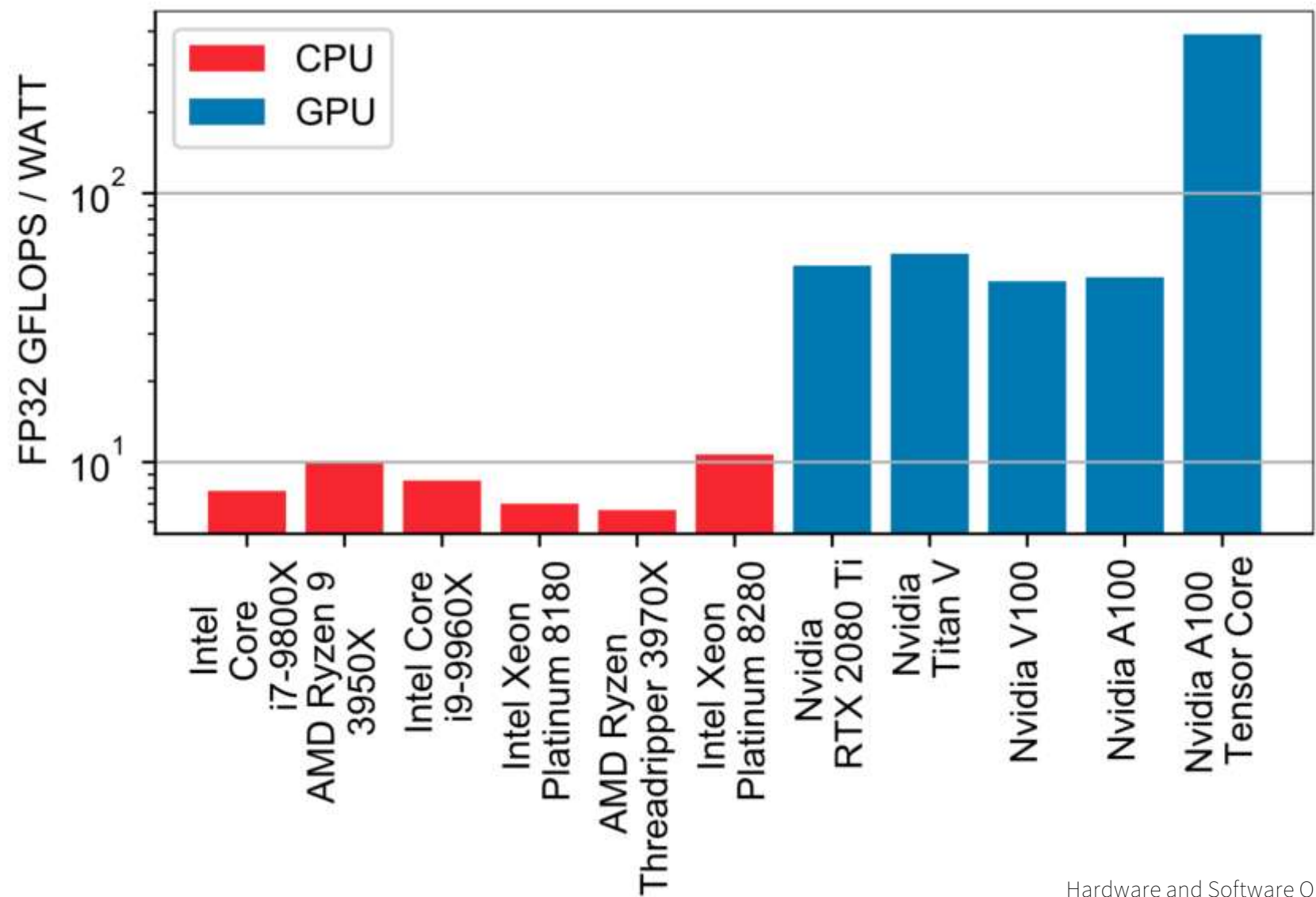
Frontier: #1 in TOP500 list (Jun 2023)

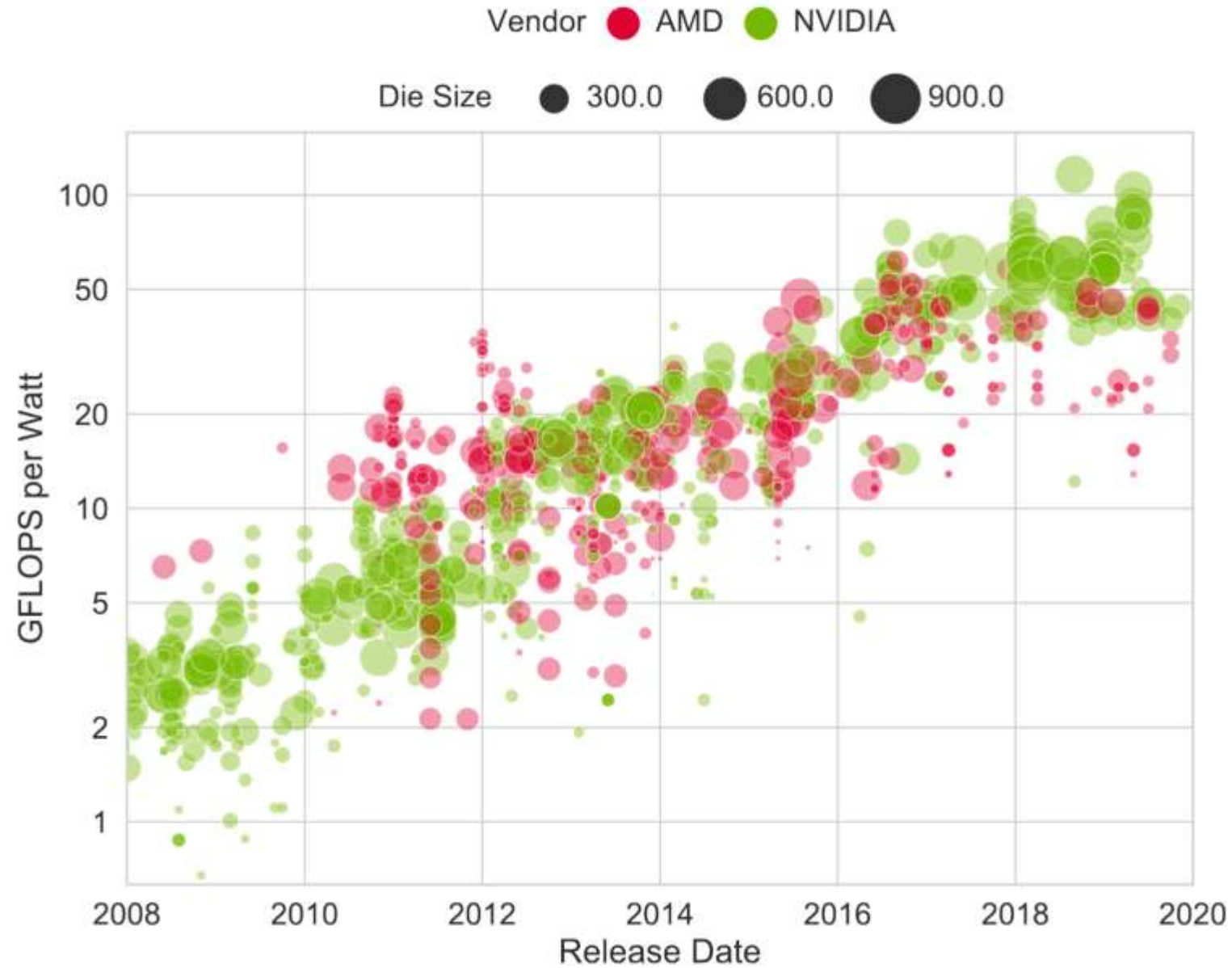
- #6 Green500 (Jun 2023)
- 20 Megawatt continuously
- \$40 million annual electricity bill
- 100,000 metric tons of CO2 annually
- ~20,000 cars on the road for a year in US

Summit: (#5, Frontier's predecessor)

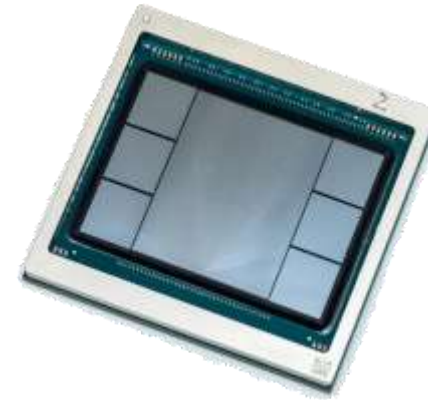
- 64% of energy is consumed by GPUs

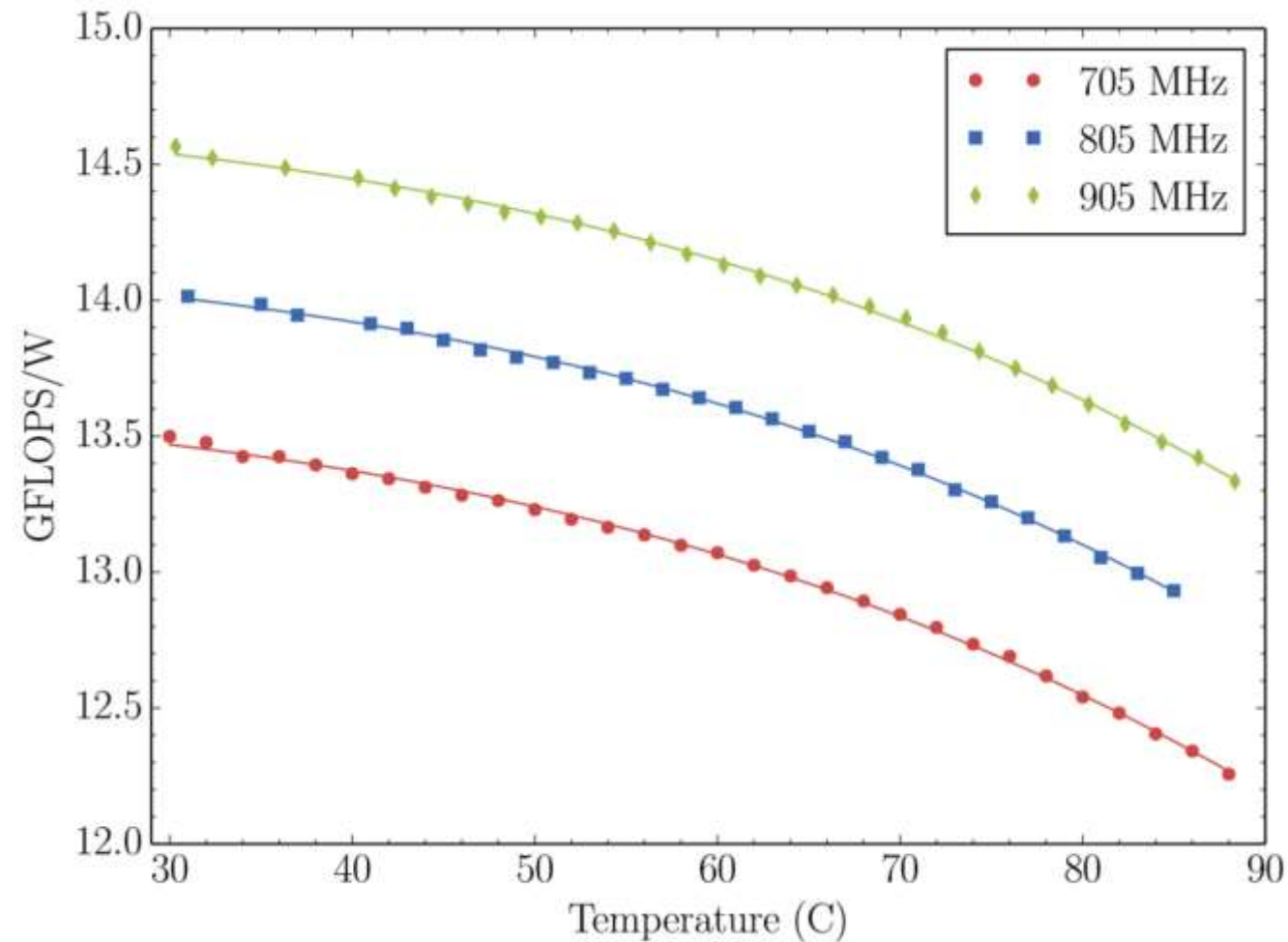






- Nvidia H100 GPU:
 - Energy: 350 Watt
 - Surface: 8.14 cm²
 - Heat dissipation: 43.0 Watt/cm²
- Light bulb:
 - Energy: 100 Watt
 - Surface: 15 cm²
 - Heat dissipation: 6.7 Watt/cm²
- Electric cooker:
 - Energy: 1800 Watt
 - Surface: 1017 cm²
 - Heat dissipation: 1.8 Watt/cm²



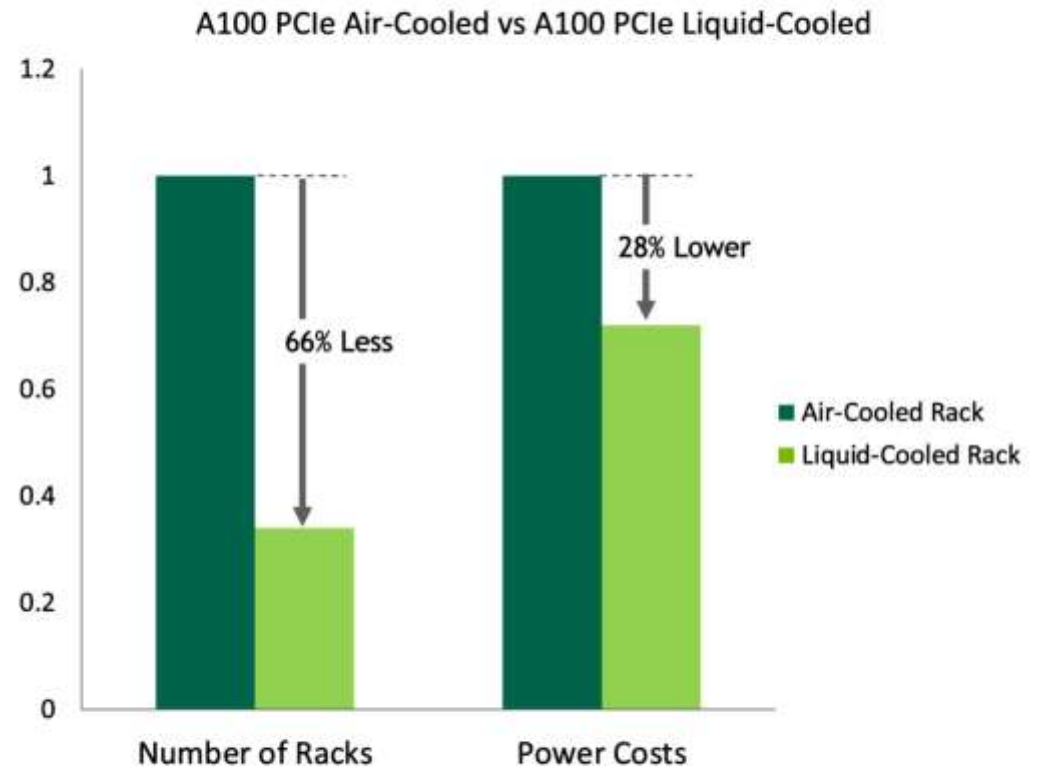


Hotter GPUs can be ~7% less energy efficient

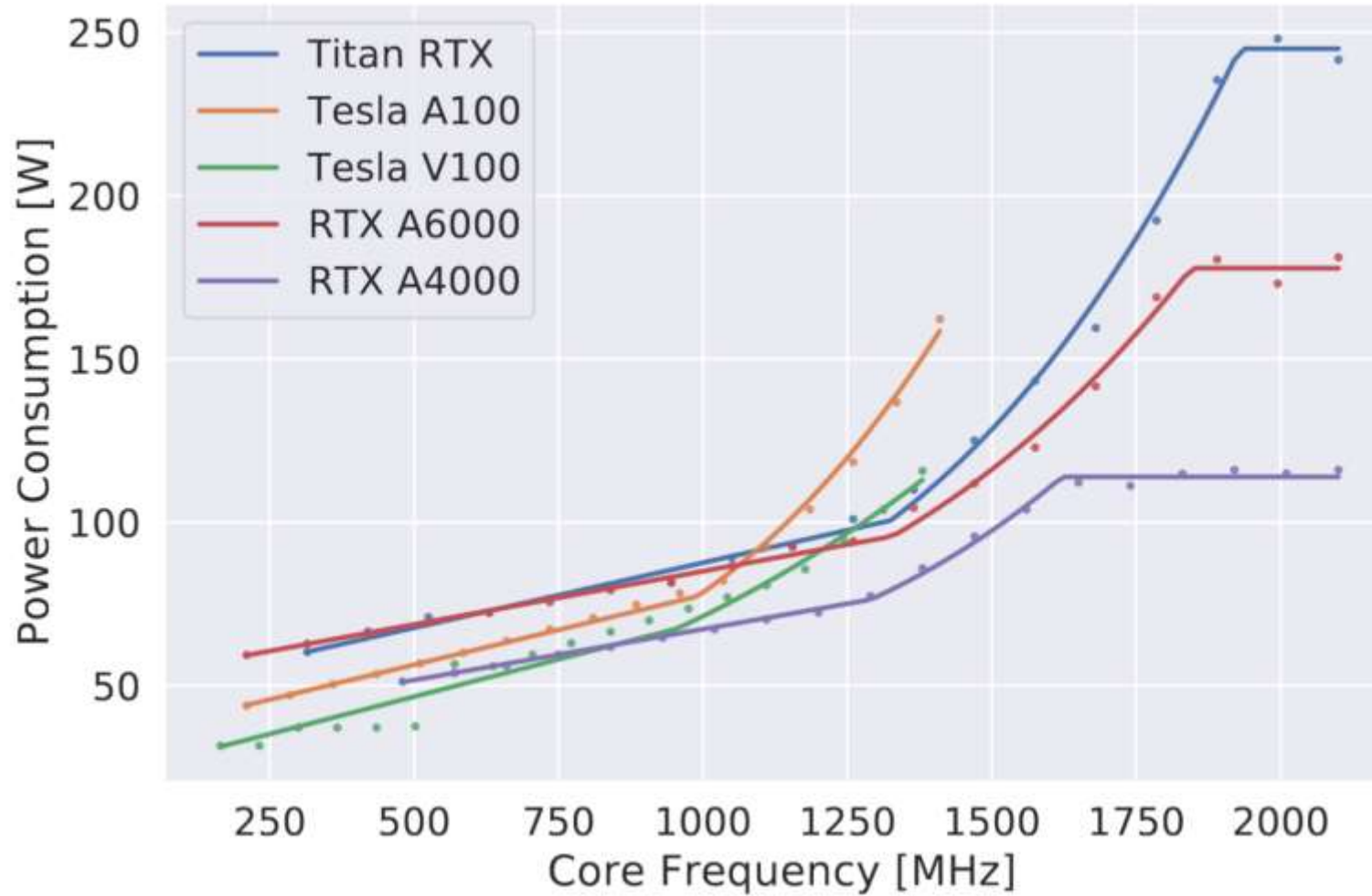
Results obtained with xGPU (radio astronomy correlator) on Nvidia K20

- 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

RACK LEVEL COST REDUCTION



Configuration:
2000 servers each with 2x CPU | 192GB | 1TB SSD | 2x A100 80GB
Air-cooled and liquid-cooled GPUs each at 300W TDP and same performance characteristics
Air-cooled infrastructure @ 1.6 PUE; Liquid-cooled infrastructure @ 1.15 PUE
15KW Air-Cooled Rack | 30KW Liquid-Cooled Rack | Power costs = \$0.2 per KWhr

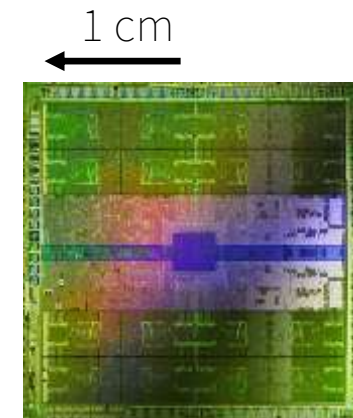


- Moving data around is 20x more expensive than computing on it

Estimations for Nvidia H100:

- A single double-precision Fused Multiply-Add¹: 13.7 pJ
- Moving the operands (4x 64-bits) for 10 mm within chip²: 294.4 pJ (21x more energy)

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



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

²Based 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

13:30	–	13:35	Opening and welcome
13:35	–	14:00	Introduction Energy Efficient GPU Computing
14:00	–	14:15	Introduction to auto-tuning with Kernel Tuner
14:15	–	14:30	First hands-on session

1.

14:30	–	15:00	Code optimization techniques for energy efficiency
15:00	–	15:30	Coffee break
15:30	–	15:45	Second hands-on session

2.

15:45	–	16:15	Mixed precision programming techniques
16:15	–	16:30	Third hands-on session

3.

16:30	–	16:45	Optimizing GPU core clock frequency
16:45	–	16:55	Fourth hands-on session
16:55	–	17:00	Closing remarks

13:30	–	13:35	Opening and welcome
13:35	–	14:00	Introduction Energy Efficient GPU Computing
14:00	–	14:15	Introduction to auto-tuning with Kernel Tuner
14:15	–	14:30	First hands-on session

1.

14:30	–	15:00	Code optimization techniques for energy efficiency
15:00	–	15:30	Coffee break
15:30	–	15:45	Second hands-on session

2.

15:45	–	16:15	Mixed precision programming techniques
16:15	–	16:30	Third hands-on session

3.

16:30	–	16:45	Optimizing GPU core clock frequency
16:45	–	16:55	Fourth hands-on session
16:55	–	17:00	Closing remarks

Introduction to Auto-Tuning with Kernel Tuner



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

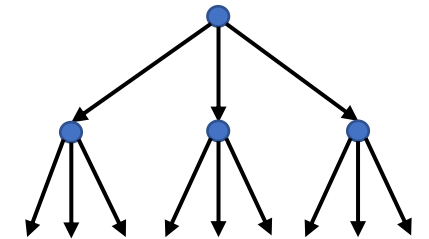
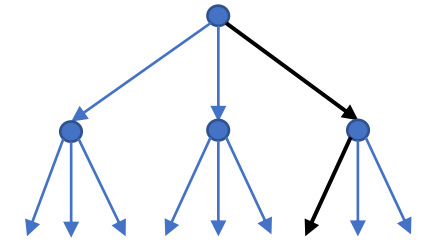
- Different mappings of the problem to threads and thread blocks
- Different data layouts in different memories (shared, constant, ...)
- Different ways of exploiting special hardware features
- Thread block dimensions
- Code optimizations that may be applied or not
- Work per thread in each dimension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

Challenge:

- 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

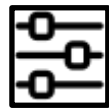
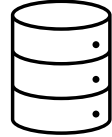


Python script



1

Kernel Data Parameters



Kernel Tuner

Compiles &
benchmarks

2



3

Optimal
parameter
values

Kernel Tuner

A tool for automatic performance tuning of GPU kernels

- Developed open source (Apache 2.0)
- Funded by several national 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

https://github.com/KernelTuner/kernel_tuner

```
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)
```

Growing Kernel Tuner ecosystem

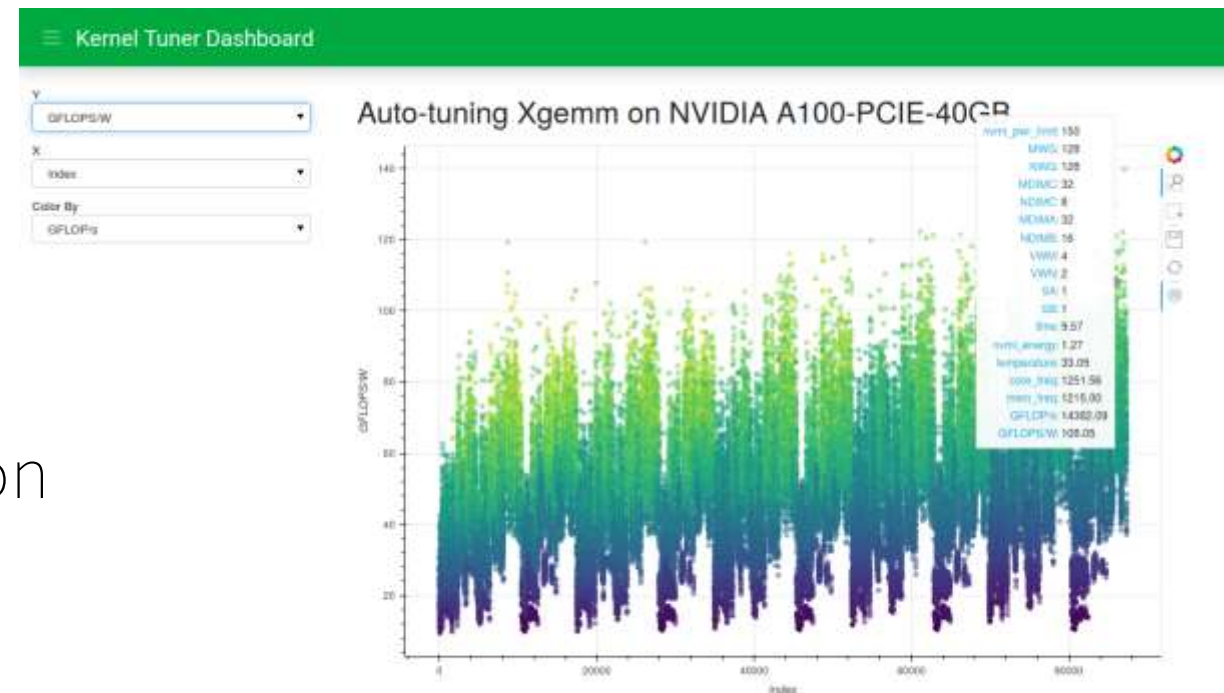
Kernel Launcher

C++ magic to integrate auto-tuned kernels into C++ applications

Kernel Float

C++ data types for mixed-precision GPU kernel programming

Kernel Dashboard



<https://github.com/KernelTuner/>

Auto-tuning as numerical optimization

- Construct a search space \mathcal{X} from all tunable parameters P_1, P_2, \dots, P_n and all possible values D_1, D_2, \dots, D_n satisfying all user-defined constraints C_1, C_2, \dots, C_m :

$$\mathcal{X} = CSP\langle P, D, C \rangle$$

- Let $f(x)$ be the execution time of kernel configuration $x \in \mathcal{X}$
- Treat the problem as a numerical optimization problem

$$x_{opt} = \arg \min f(x)$$

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, ...

Algorithm Column beats Row - convolution feval <= 200

BasinHopping	0	4	5	5	13	19	9	8	4	9	9	10	4	6	10
BestILS	6	0	2	2	13	19	5	7	3	7	7	9	6	8	12
BestMLS	6	2	0	1	12	16	10	8	5	6	10	11	7	11	11
BestTabu	10	9	12	0	16	21	14	16	5	15	18	13	12	14	20
DifferentialEvolution	3	4	4	1	0	17	7	9	3	5	4	1	1	8	8
DualAnnealing	1	0	0	0	1	0	4	2	0	0	1	0	0	1	3
FirstILS	4	1	1	1	7	16	0	3	1	2	6	8	5	9	7
FirstMLS	5	0	0	0	10	14	5	0	1	3	8	8	6	8	7
FirstTabu	7	6	8	2	13	20	12	10	0	9	14	13	7	11	14
GLS	5	0	0	1	8	15	5	3	0	0	6	6	4	7	6
GeneticAlgorithm	2	0	2	0	3	17	7	6	1	4	0	1	0	6	3
ParticleSwarm	5	7	5	3	6	19	10	8	2	8	8	0	1	8	9
RandomSampling	9	11	11	7	16	24	13	12	7	13	13	16	0	14	11
SMAC4BB	1	5	6	4	11	20	9	7	2	8	8	7	1	0	9
SimulatedAnnealing	3	0	0	0	6	16	5	2	0	1	3	3	1	6	0

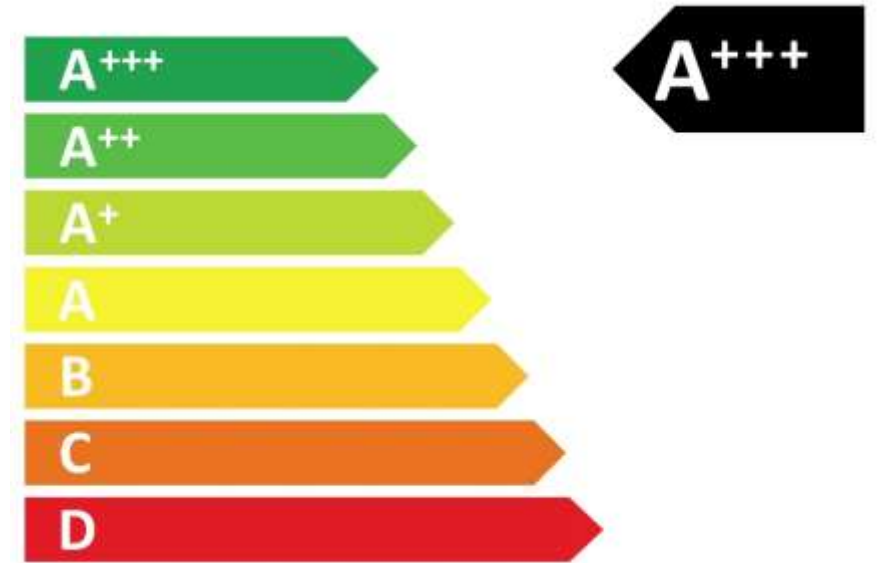
BasinHopping
BestILS
BestMLS
BestTabu
DifferentialEvolution
DualAnnealing
FirstILS
FirstMLS
FirstTabu
GLS
GeneticAlgorithm
ParticleSwarm
RandomSampling
SMAC4BB
SimulatedAnnealing

Optimizing Energy Efficiency

Minimize energy consumption instead of only the execution time

Kernel Tuner supports:

- Measuring power consumption during tuning
 - GPU built-in current sensors
 - External power measurement hardware
- Custom objectives to optimize for time, energy or any user-defined metric
- Performance models for model-steered auto-tuning



- 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)

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

- Prerequisites:
 - Python 3.8 or newer
 - CUDA or OpenCL device with necessary drivers and compilers installed
- To install Kernel Tuner:
 - **`pip install kernel_tuner`**
- For more information:
 - https://kerneltuner.github.io/kernel_tuner/latest/install.html
- Note: installation on your system is not required for the hands-on sessions

Hands-on



- The first hands-on notebook is:
 - https://colab.research.google.com/github/KernelTuner/kernel_tuner_tutorial/blob/master/energy/00_Kernel_Tuner_Introduction.ipynb
- The goal of this hands-on is to:
 - Install and run Kernel Tuner
 - Measure energy consumption of our kernels
 - View the results using Kernel Tuner Dashboard
- [Open the notebook in Google Colab](#) and work there
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors

Change runtime type in Colab

00-Kernel_Tuner-Introduction.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text Copy to Drive

Kernel Tuner Tutorial

Introduction Hands-on

Welcome to the first hands-on of the Kernel Tuner, a tool that helps you optimize your Python, and **tune** our first CUDA kernel.

To install the latest version of `kernel_tuner`, make sure to have both `numpy` and `cupy` installed.

```
[ ] %pip install kernel_tuner
```

After installing all necessary packages, we will create a simple kernel.

```
[ ] import numpy as np
import kernel_tuner as kt
```

Before using Kernel Tuner, we will create a simple kernel. This simple kernel is called `vector_add`.

```
[ ] %%writefile vector_add_kernel.cu

__global__ void vector_add(float * c, float * a, float * b, int n) {
    int i = (blockIdx.x * blockDim.x) + threadIdx.x;

    if ( i < n ) {
        c[i] = a[i] + b[i];
    }
}
```

The execution of the cell above created the file `vector_add_kernel.cu` containing the source code of our kernel.

Connected to Python 3 Google Compute Engine backend (GPU)

Change runtime type

Runtime type

Python 3

Hardware accelerator ?

☐ CPU ☒ T4 GPU ☐ A100 GPU ☐ V100 GPU

☐ TPU

Want access to premium GPUs? [Purchase additional compute units](#)

Cancel Save

Select a GPU

- Observers
 - To observe quantities other than execution time
 - Measurements are stored in results, but not printed to screen
- Metrics
 - Allows user to create derived metrics
 - Specified as keys in a dict using lambda functions
 - Always printed to screen, also stored in results
- Cache files
 - Allows Kernel Tuner to continue from previous runs, where it left off
 - Allows Dashboard to visualize tuning results during/after the run

Code Optimization Techniques for Energy Efficiency



- Modify the kernel source code to improve performance or tunability
- Effects on performance can be different on different GPUs or different input data
- You can tune:
 - Enabling or disabling an optimization
 - The parameters introduced by certain optimizations
- You often need to combine multiple different optimizations with specific tunable parameter values to arrive at optimal performance

- In March 2023, we published a literature review summarizing the last decade of code optimizations for GPU programming
 - We describe which optimizations are used in literature and how they are used
- *Optimization Techniques for GPU Programming*
Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri Bal
ACM Computing surveys 2023
<https://dl.acm.org/doi/abs/10.1145/3570638>

- Coalescing memory accesses
- Host/device communication
- Kernel fusion
- Loop blocking
- Loop unrolling
- Prefetching
- Recomputing values
- Reducing atomics
- Reducing branch divergence
- Reducing redundant work
- Reducing register usage
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- Varying work per thread
- Vectorization

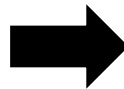
- Coalescing memory accesses
- Host/device communication
- Kernel fusion
- Loop blocking
- Loop unrolling
- Prefetching
- Recomputing values
- Reducing atomics
- Reducing branch divergence
- Reducing redundant work
- Reducing register usage
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- Varying work per thread
- Vectorization

Merge one or more kernels into one kernel

- Why?
 - Reduces data movements between off-chip DRAM and GPU registers
 - Moving data around is more expensive than computing on it
- How?
 - Fuse the kernel arguments and computations of two kernels into one
 - Demote a kernel to a **__device__** function and call it from another kernel
 - *Temporal fusion*: merge multiple calls of the same kernel into one

```
// c = a+b
vector_add<<<grid, threads>>>(c, a, b, n);
// e = c+d
vector_add<<<grid, threads>>>(e, c, d, n);
```

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

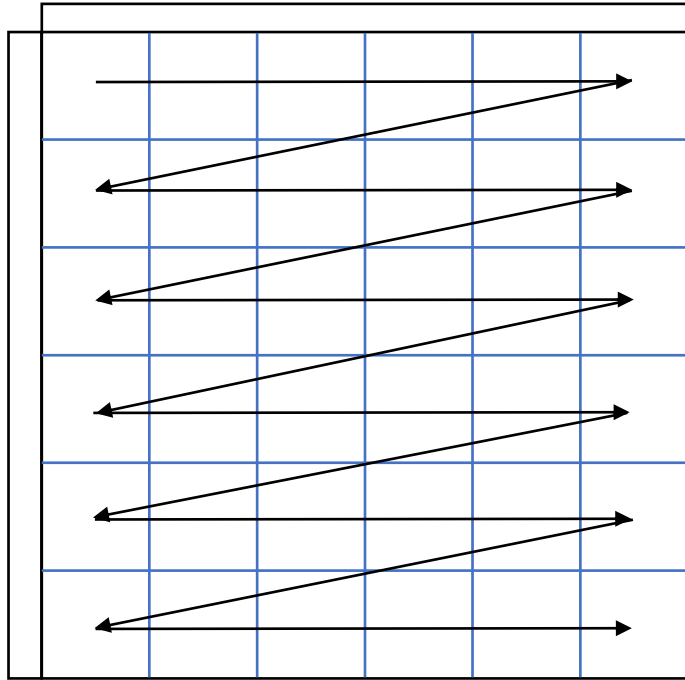


```
// e = a+b+d
vector_3add<<<grid, threads>>>(e, a, b, d, n);
```

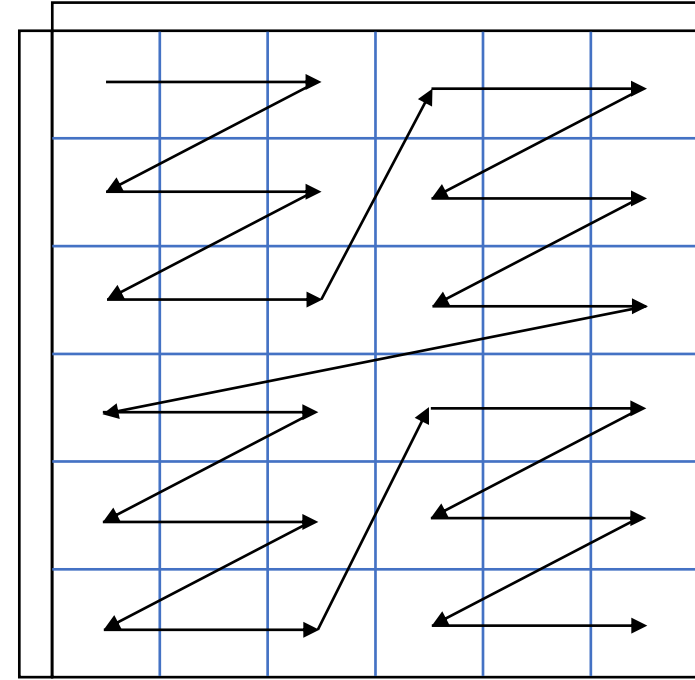
```
__global__
void vector_3add(float *d, float *a, float *b,
                float *c, int n) {
    int i = (blockIdx.x*blockDim.x)+threadIdx.x;
    if (i < n) {
        d[i] = a[i] + b[i] + c[i];
    }
}
```

Modify the structure of one or more loops to work in blocks over the data

- Why?
 - Increases spatial / temporal locality
 - Reduces the ‘working set’ of the algorithm
- How?
 - Change the order of computations and data accesses in nested loops
 - Usually nearly doubles the number of for-loops in the code
 - Outer-loops iterate over the blocks
 - Inner-loops iterate within each block



```
for (int j=0; j<ny; j++) {
    for (int i=0; i<nx; i++) {
        ...[j*nx + i]
    }
}
```



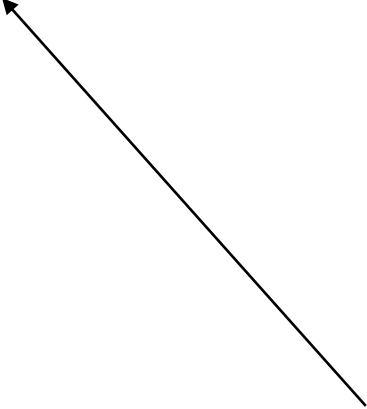
```
for (int j=0; j<ny; j+=nyb) {
    for (int i=0; i<nx; i+=nxb) {
        for (int jb=0; jb<nyb; jb++) {
            for (int ib=0; ib<nxb; ib++) {
                ...[(j+jb)*nx + (i+ib)]
            }
        }
    }
}
```

Reduce the number of iterations of a loop

- Why?
 - Increases instruction-level-parallelism
 - Reduces loop overhead instructions
- How?
 - Replicate the contents of a for-loop n times, increase loop counter by n
 - In the early days, only manually or with a code generator
 - Compiler does this now: **#pragma unroll <value>**
 - In CUDA, value has to be an integer constant expression
 - 0 is not allowed and gives an error, 1 means unrolling is disabled

...

```
#pragma unroll loop_unroll_factor_k  
for (int k=0; k<n; k++) {  
    ...  
}
```



The compiler can unroll this loop if **n** is known at compile-time. The **loop_unroll_factor_k** parameter should be a divisor of the loop counter **n**

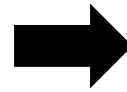
Reduce the number of registers per thread required by the kernel

- Why?
 - Registers are an important and limited SM resource and are likely to limit occupancy
 - Allows to increase the tunable range of thread block dimensions
- How?
 - Compiling constant values into your code rather than keeping them in registers (e.g. using templates or tunable parameters)
 - Limiting or disabling loop unrolling is very effective in reducing register usage
 - In kernels that do many different things, split the kernel
 - Enabling register spilling with compiler flag **-maxrregcount=N** or tuning the number of blocks per SM using the kernel **__launch_bounds__()**

Reducing register usage

```
__global__ void
```

```
some_kernel(...)  
{  
    ...  
}
```



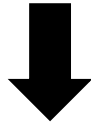
```
__global__ void  
__launch_bounds__(block_size_x*block_size_y*block_size_z,  
                  blocks_per_sm)  
some_kernel(...)  
{  
    ...  
}
```

Increase or decrease the amount of work per thread (or thread block) and adjust the number of threads and thread blocks accordingly

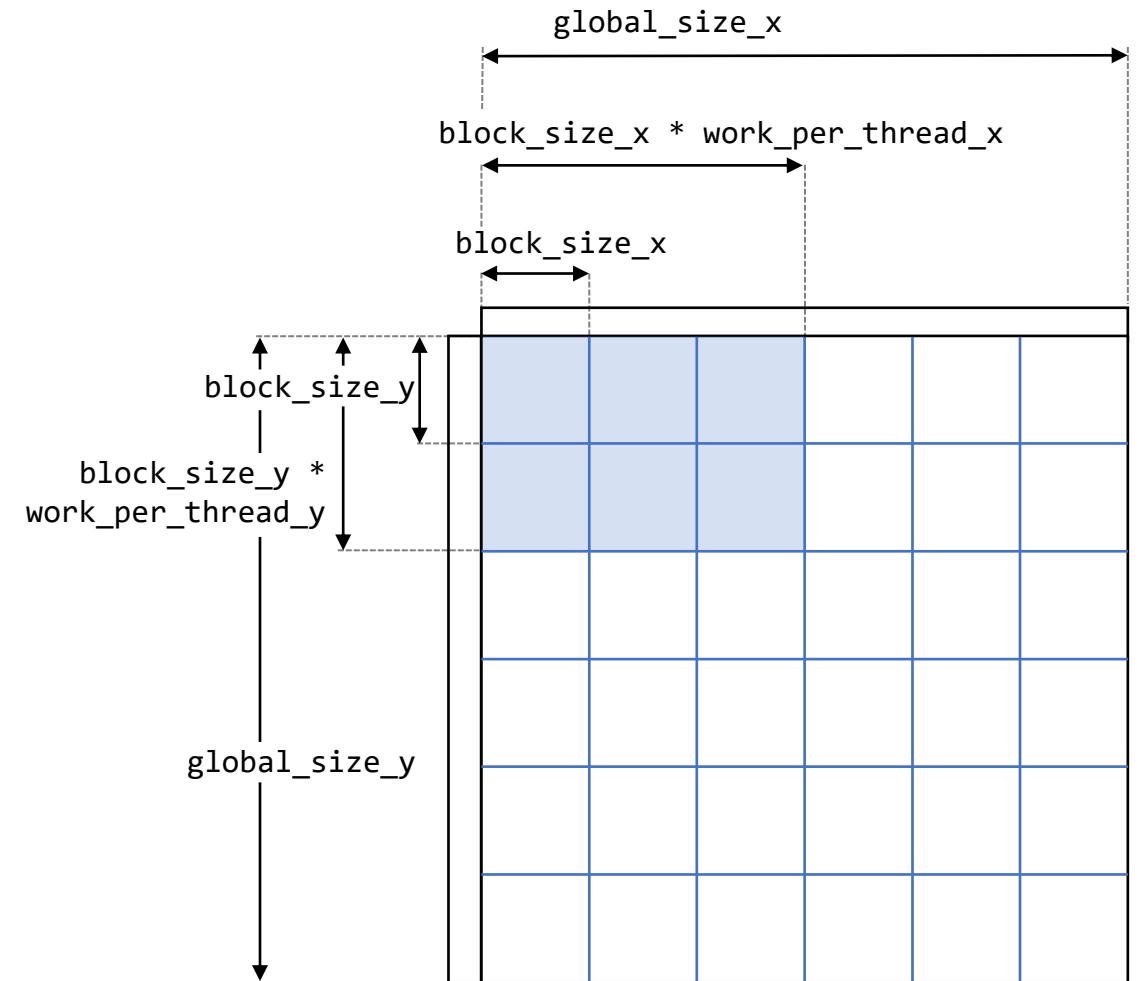
- Why?
 - Increasing work per thread often increases data reuse and locality
 - Reduces redundant instructions previously executed by other threads
 - Increases instruction-level parallelism and possibly increases register usage
- How?
 - Reduce number of threads blocks in total, but increase the work per thread block
 - Bring down number of threads within the block, but keep the amount of work equal

Varying work per thread

```
...  
#pragma unroll  
for (kb = 0; kb < block_size_x; kb++) {  
    sum += sA[ty][kb] * sB[kb][tx];  
}
```



```
...  
#pragma unroll  
for (kb = 0; kb < block_size_x; kb++) {  
    #pragma unroll  
    for (int j = 0; j < work_per_thread_x; j++) {  
        sum[j] += sA[ty][kb] * sB[kb][tx + j * block_size_x];  
    }  
}
```



- Coalescing memory accesses
- Host/device communication
- Kernel fusion
- Loop blocking
- Loop unrolling
- Prefetching
- Recomputing values
- Reducing atomics
- Reducing branch divergence
- Reducing redundant work
- Reducing register usage
- Reformatting input data
- Using a specific memory space
- Using warp shuffle instructions
- Varying work per thread
- Vectorization

Hands-on



- The second hands-on notebook is:
 - https://colab.research.google.com/github/KernelTuner/kernel_tuner_tutorial/blob/master/energy/01_Code_Optimizations_for_Energy.ipynb
- The goal of this hands-on is to:
 - See an example of Kernel Fusion
 - Compare the energy consumption of different kernels
- [Open the notebook in Google Colab](#) and work there
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors

Mixed Precision Programming Techniques



- Prevalent is double precision (64 bit); GPUs also support lower precision

Double (64 bit)



Single (32 bit)



FP24 (24 bit)



TensorFloat (19 bit)



Half (16 bit)



Bfloat (16 bit)



Minifloat (8 bit)



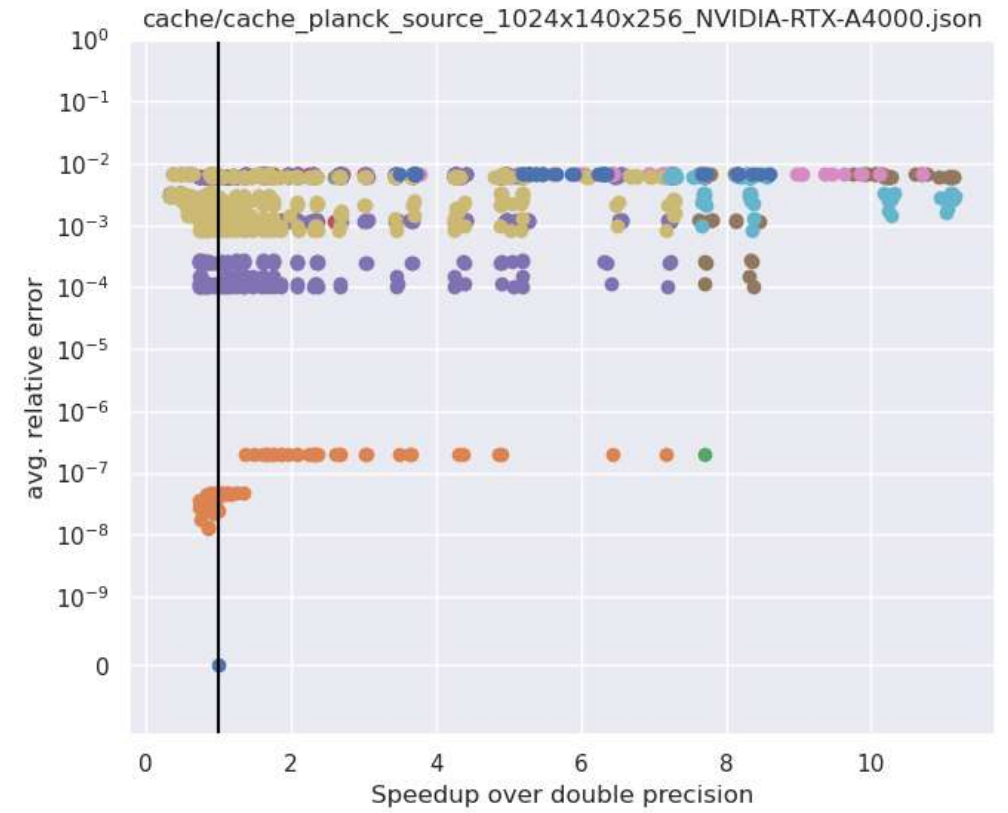
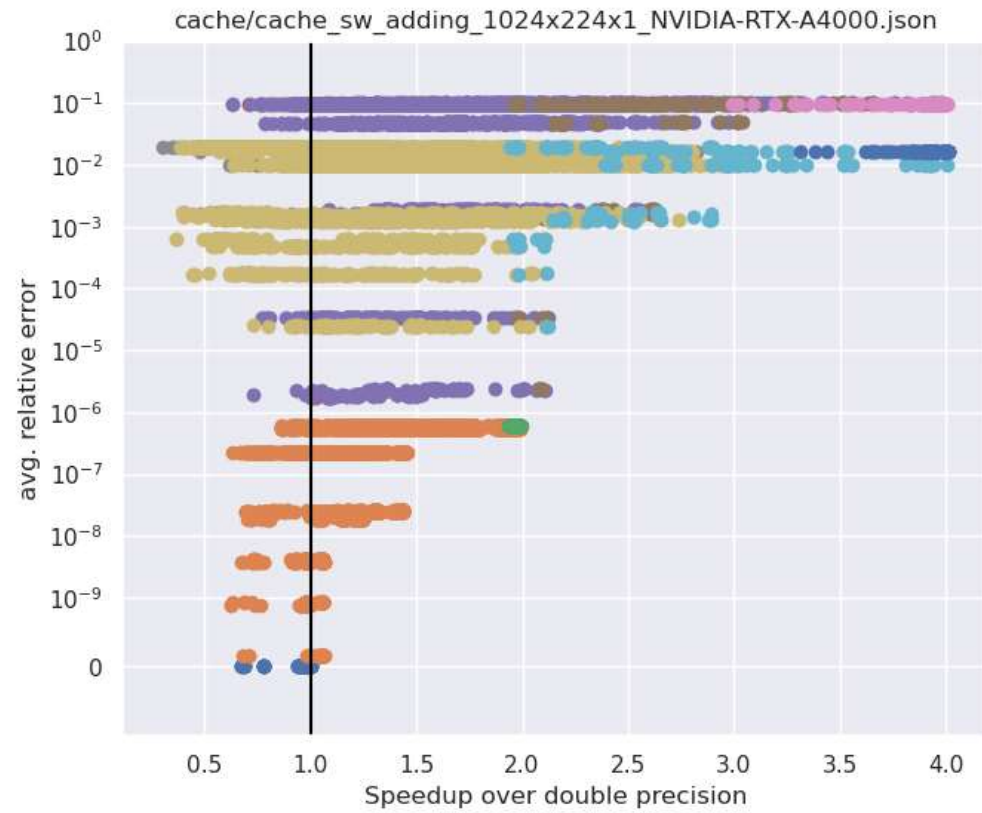
- Low precision has many benefits 😊!
 - Faster computation
 - Less compute cycles required, especially double precision is often slow
 - Lower memory footprint
 - Less bits required per number
 - Better cache utilization
 - Higher cache hit rates
 - Higher *effective* memory bandwidth
 - More numbers per second
 - Lower register usage
 - Increases GPU occupancy, thus performance
 - All these points also increase energy efficiency

- But, at the cost of loss in precision 😞

[illegible]

- Core idea:
 - What if we mix different precision levels in one application?
 - Use different floating-point types for different variables in code
- Leads to trade-off between accuracy and performance
 - Lower precision typically results in higher performance
 - Need to find balance between error and speedup
- What precision should be used for each variable?
 - Ideally, we want maximum performance for an acceptable error
 - Auto-tuning to the rescue!

Example radiation solver

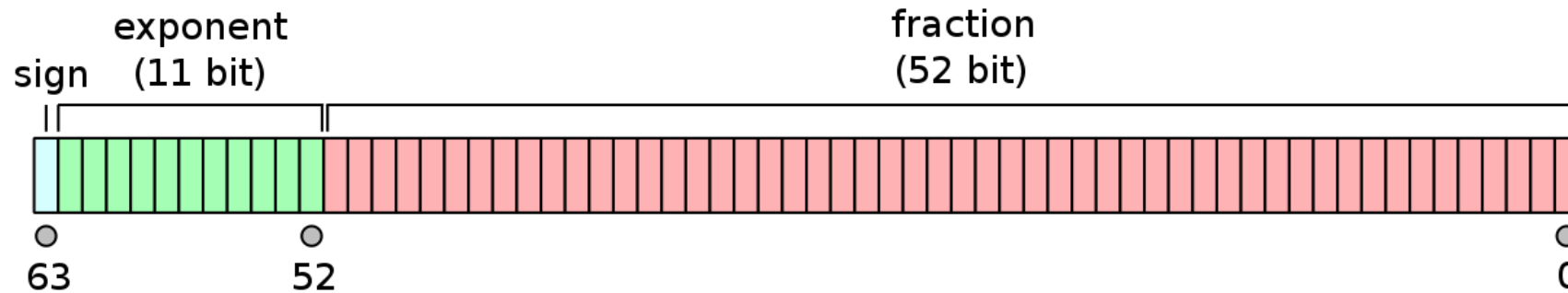


- IEEE 754 standard is implemented in all architectures
- Floating-point number consists of three parts:
 - S: sign (+ or -)
 - M: mantissa/significand
 - E: exponent
- Floating-point number represented using exponential format:
 - $(-1)^S \times M \times 2^E$
 - Example: $+1.42 \times 2^3$ means $S=+1$, $M=1.42$, $E=3$
 - Where $1 \leq M < 2$, which makes representation unique
 - There are also non-normal numbers: NaN, Inf, subnormal

- Sign bit (1 bit)
- Mantissa/significand (**A** bits)
 - Determines number of significant digits
 - Results rounded to number of decimal places
 - Example: A=23 means ~7 decimal places
- Exponent (**B** bits)
 - Determines range of numbers
 - Numbers outside range become *zero* or *infinity*
 - Example: **B=8** means range is $\sim 10^{-38}$ to $\sim 10^{38}$
- Total size: 1 + A + B bits

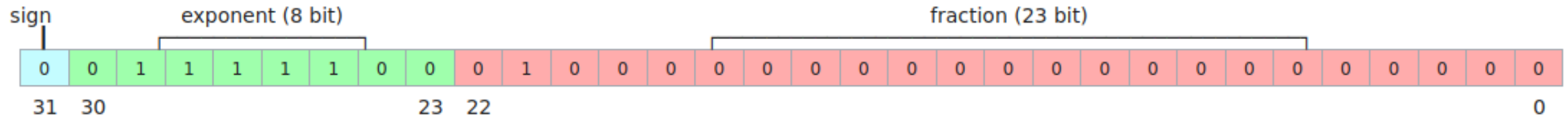
Type	$\sqrt{2}$
A=52 (Float64)	1.41421356237309
A=23 (Float32)	1.414213
A=10 (Float16)	1.414
A=2 (Float8)	1.5





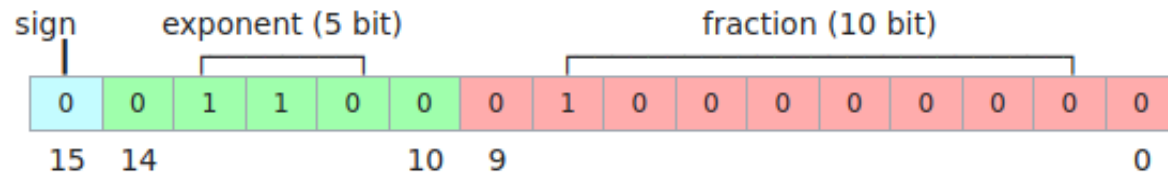
- **double** precision (64 bits) prevalent in scientific computing
- GPUs typically slow on double arithmetic
 - Except the scientific/datacenter-rated GPUs

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
double	64	11	52	2.2e-308	1.8e+308	15	1 + 2.22e-16



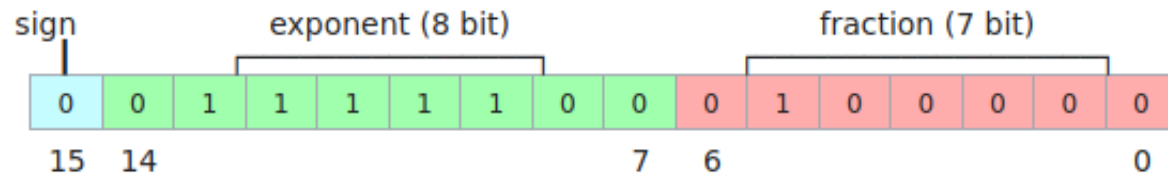
- Single precision (32 bits) balances accuracy and throughput
- Widely used in graphics and general GPU applications

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
float	32	8	23	1.2e-38	3.4e+38	6	1.000000119



- Introduced with NVIDIA's Pascal architecture (2016)
- Double computational throughput of float
- Limited range, reasonable accuracy

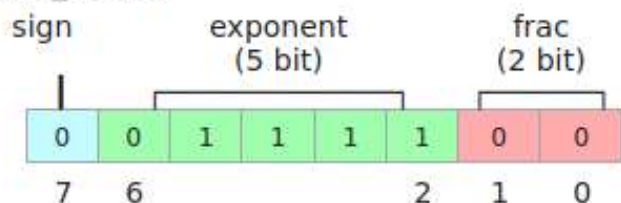
Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
half	16	5	10	0.000061	65536	3	1.00097



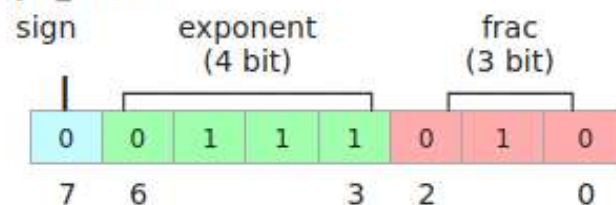
- "Brain" Floating-point. Introduced by Google Brain project
- Introduced with NVIDIA's Ampere architecture (2020)
- Large range, limited accuracy

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
bfloat	16	8	7	1.2e-38	3.4e+38	2	1.00781

fp8_e5m2



fp8_e4m3



- Introduced with NVIDIA's Hopper architecture (2022)
- Two flavors: 5+2 bits or 4+3 bits
- No arithmetic functions, only conversions

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
fp8_e4m3	8	4	3	0.015625	256	1	1.125
fp8_e5m2	8	5	2	0.000061	65536	0	1.25

Type name	Total bits	Exponent bits	Significant bits	Smallest normal	Biggest normal	Decimal places	1+Epsilon
double	64	11	52	2.2e-308	1.8e+308	15	1 + 2.22e-16
float	32	8	23	1.2e-38	3.4e+38	6	1.000000119
half	16	5	10	0.000061	65536	3	1.00097
bfloat	16	8	7	1.2e-38	3.4e+38	2	1.00781
fp8_e4m3	8	4	3	0.015625	256	1	1.125
fp8_e5m2	8	5	2	0.000061	65536	<1	1.25

- Create type aliases in kernels
 - C: use preprocess #define
 - C++: use template parameters
- Available data types in CUDA
 - **double** and **float** are predefined
 - **__half** found in **<cuda_fp16.h>**
 - **__nv_bfloat16** found in **<cuda_bf16.h>**
 - **__nv_fp8_eXmY** found in **<cuda_fp8.h>**

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



```
#define TYPE_A float
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
    int n,
    const TYPE_A* A,
    const TYPE_B* B,
    TYPE_C* C
) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n)
        C[i] = A[i] + B[i];
}
```

```
#include <cuda_fp16.h>

#define TYPE_A __half
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
    int n,
    const TYPE_A* A,
    const TYPE_B* B,
    TYPE_C* C
) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n)
        C[i] = A[i] + B[i];
}
```

```
#include <cuda_fp16.h>

#define TYPE_A __half
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
    int n,
    const TYPE_A* A,
    const TYPE_B* B,
    TYPE_C* C
) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n)
        C[i] = A[i] + B[i];
}
```

Does not compile 😞!

kernel.cu(15): error: no operator
"+" matches these operands
operand types are: __half + float

```
#include <cuda_fp16.h>

#define TYPE_A __half
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
    int n,
    const TYPE_A* A,
    const TYPE_B* B,
    TYPE_C* C
) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n)
        C[i] = A[i] + B[i];
}
```

Does not compile 😞!

kernel.cu(15): error: no operator
"+" matches these operands
operand types are: __half + float

- No type promotion
 - Cannot mix types in binary operations
- Some operations require intrinsics
 - `__hdiv()`, `__hsin()`, `__hfmad()`
- Missing operations
 - No `__htan()`?
- Missing or awkward type conversion
 - `__nv_cvt_bfloat16raw2_to_fp8x2`
 - No `fp8` to `double`?
 - No `half` to `bfloat16`?

```
__global__ void kernel(const __half* input, float constant, float* output) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    __half in0 = input[2 * i + 0];  
    __half in1 = input[2 * i + 1];  
    __half2 a = __halves2half2(in0, in1);  
    float b = float(constant);  
    __half c = __float2half(b);  
    __half2 d = __half2half2(c);  
    __half2 e = __hadd2(a, d);  
    __half f = __low2half(e);  
    __half g = __high2half(e);  
    float out0 = __half2float(f);  
    float out1 = __half2float(g);  
    output[2 * i + 0] = out0;  
    output[2 * i + 1] = out1;  
}
```

```
__global__ void kernel(const __half* input, float constant, float* output) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    __half in0 = input[2 * i + 0];
    __half in1 = input[2 * i + 1];
    __half2 a = __halves2half2(in0, in1);
    float b = float(constant);
    __half c = __float2half(b);
    __half2 d = __half2half2(c);
    __half2 e = __hadd2(a, d);
    __half f = __low2half(e);
    __half g = __high2half(e);
    float out0 = __half2float(f);
    float out1 = __half2float(g);
    output[2 * i + 0] = out0;
    output[2 * i + 1] = out1;
}
```



```
#include "kernel_float.h"
namespace kf = kernel_float;

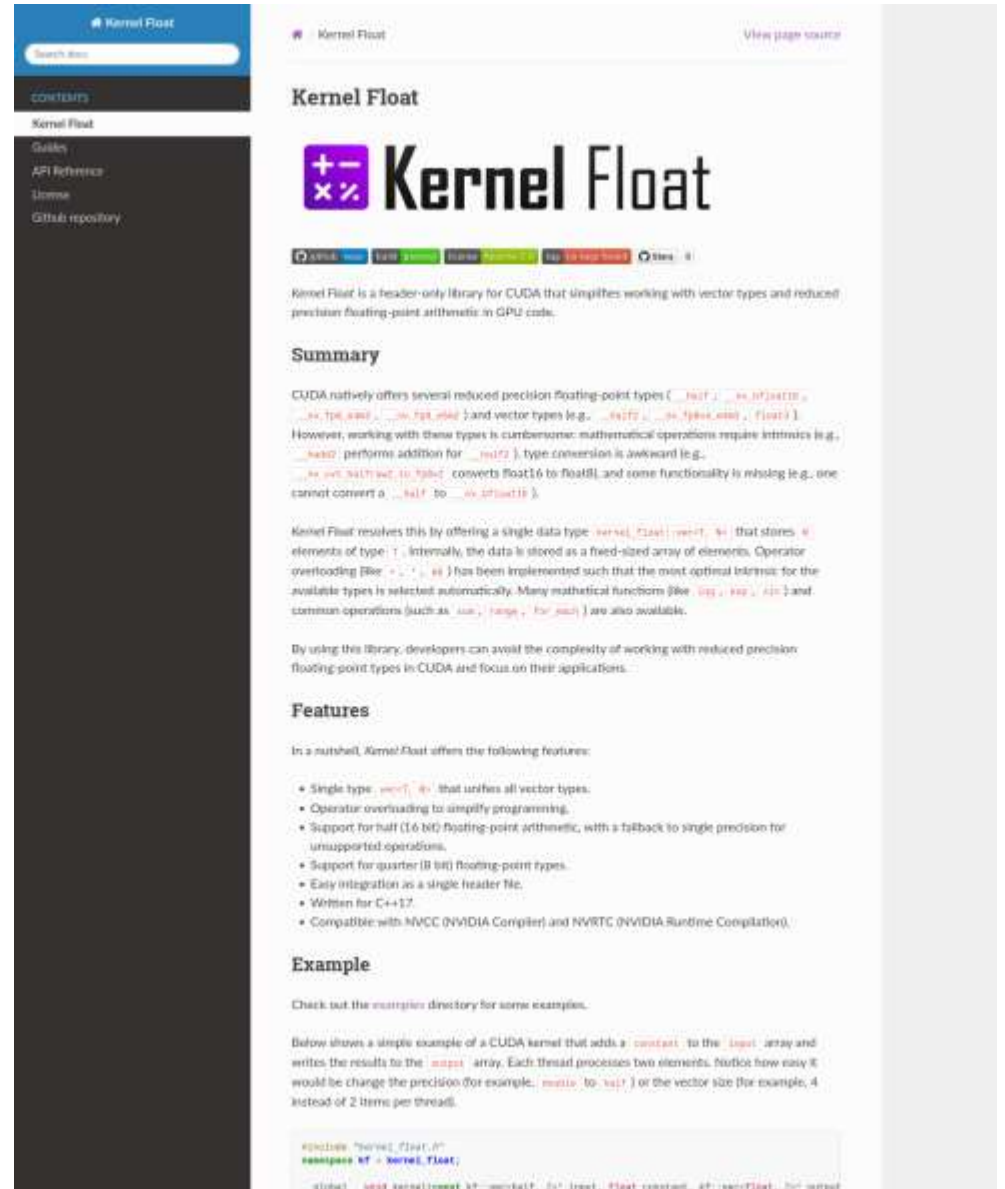
__global__ void kernel(const kf::vec<half, 2>* input, float constant, kf::vec<float, 2>* output) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    output[i] = input[i] + kf::cast<half>(constant);
}
```



Kernel Float

https://github.com/KernelTuner/kernel_float

- Header-only C++ library to simplify mixed precision GPU programming
- Offers single type: **vec<T, N>**
 - **N** elements of type **T**
 - Auto selects optimal storage format
- Offers all mathematical operations
 - Auto selects best intrinsic
 - Fallback to single precision for missing operations



The screenshot shows the GitHub repository for Kernel Float. The left sidebar contains a search bar and a table of contents with links to Kernel Float, Guides, API Reference, License, and GitHub repository. The main content area displays the repository name, a logo with mathematical symbols, and a summary of the library. The summary explains that Kernel Float is a header-only library for CUDA that simplifies working with vector types and reduced precision floating-point arithmetic. It mentions that CUDA natively offers several reduced precision floating-point types and vector types, but working with them is cumbersome. Kernel Float resolves this by offering a single data type, `vec<T, N>`, that stores `N` elements of type `T`. It also lists features such as single type, operator overloading, support for half and quarter floating-point types, easy integration as a single header file, and compatibility with NVCC and NVRTC. An example section shows a simple CUDA kernel that adds a constant to an input array and writes the results to an output array.

Kernel Float

Kernel Float is a header-only library for CUDA that simplifies working with vector types and reduced precision floating-point arithmetic in GPU code.

Summary

CUDA natively offers several reduced precision floating-point types (e.g., `__half`, `__half2`, `__half4`, `__half8`) and vector types (e.g., `__half2`, `__half4`, `__half8`, `__float2`). However, working with these types is cumbersome: mathematical operations require intrinsics (e.g., `__half2` performs addition for `__half2`), type conversion is awkward (e.g., `__half2` to `__half4`), and some functionality is missing (e.g., one cannot convert a `__half` to `__half2`).

Kernel Float resolves this by offering a single data type, `vec<T, N>`, that stores `N` elements of type `T`. Internally, the data is stored as a fixed-sized array of elements. Operator overloading (like `+`, `*`, `^`) has been implemented such that the most optimal intrinsic for the available types is selected automatically. Many mathematical functions (like `sin`, `exp`, `log`) and stream operations (such as `sum`, `range`, `for_each`) are also available.

By using this library, developers can avoid the complexity of working with reduced precision floating-point types in CUDA and focus on their applications.

Features

In a nutshell, Kernel Float offers the following features:

- Single type, `vec<T, N>`, that unifies all vector types.
- Operator overloading to simplify programming.
- Support for half (16-bit) floating-point arithmetic, with a fallback to single precision for unsupported operations.
- Support for quarter (8-bit) floating-point types.
- Easy integration as a single header file.
- Written for C++17.
- Compatible with NVCC (NVIDIA Compiler) and NVRTC (NVIDIA Runtime Compiler).

Example

Check out the `examples` directory for some examples.

Below shows a simple example of a CUDA kernel that adds a `constant` to the `input` array and writes the results to the `output` array. Each thread processes two elements. Notice how easy it would be to change the precision (for example, `vec<float, 2>`) or the vector size (for example, 4 instead of 2 items per thread).

```
#include "kernel_float.h"
constexpr kf = kernel_float;

__global__ void kernel(const kf::vec<float, 2> &input, float constant, kf::vec<float, 2> &output)
```



```
#define TYPE_A float
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
    int n,
    const TYPE_A* A,
    const TYPE_B* B,
    TYPE_C* C
) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n)
        C[i] = A[i] + B[i];
}
```

```
#include "kernel_float.h"

#define TYPE_A float
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
    int n,
    const kernel_float::vec<TYPE_A, 1>* A,
    const kernel_float::vec<TYPE_B, 1>* B,
    kernel_float::vec<TYPE_C, 1>* C
) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n)
        C[i] = A[i] + B[i];
}
```

```
#include "kernel_float.h"

#define TYPE_A __half
#define TYPE_B float
#define TYPE_C float

__global__ void vector_add(
    int n,
    const kernel_float::vec<TYPE_A, 1>* A,
    const kernel_float::vec<TYPE_B, 1>* B,
    kernel_float::vec<TYPE_C, 1>* C
) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n)
        C[i] = A[i] + B[i];
}
```

- Kernel Float automatically uses vector intrinsics
 - Requires using **kernel_float::vec<T, N>** with $N \geq 2$
- Several types benefit from vectorization!
 - **half** and **bfloat** require vectorized intrinsics for high throughput
 - Vectorized memory operations
 - Vectorized integer operations
 - ...

```
#include "kernel_float.h"

#define TYPE_A float
#define TYPE_B float
#define TYPE_C __half
#define VECTOR_SIZE 1

__global__ void vector_add(
    int n,
    const kernel_float::vec<TYPE_A, VECTOR_SIZE>* A,
    const kernel_float::vec<TYPE_B, VECTOR_SIZE>* B,
    kernel_float::vec<TYPE_C, VECTOR_SIZE>* C
) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i * VECTOR_SIZE < n)
        C[i] = A[i] + B[i];
}
```

```
#include "kernel_float.h"

#define TYPE_A float
#define TYPE_B float
#define TYPE_C half
#define VECTOR_SIZE 2

__global__ void vector_add(
    int n,
    const kernel_float::vec<TYPE_A, VECTOR_SIZE>* A,
    const kernel_float::vec<TYPE_B, VECTOR_SIZE>* B,
    kernel_float::vec<TYPE_C, VECTOR_SIZE>* C
) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i * VECTOR_SIZE < n)
        C[i] = A[i] + B[i];
}
```

- Accuracy vs performance trade-off
 - What type should we use for each variable?
 - Ideally want high performance with low error
- Variables datatypes and kernel parameters both affect performance
 - Usually heavily intertwined, we cannot tune them separately
- Leads to large search-space, for example:
 - 10 variables and 4 precision levels: $4^{10} = 1$ million options
 - 8 parameters with each 6 options: $6^8 = 1$ million options
 - Total: 1 trillion configurations!

Kernel Tuner offers native support for *accuracy tuning*

- Step 1: Add tunable floating-point types as tuning parameters
- Step 2: Wrap inputs/outputs in **TunablePrecision** objects
- Step 3: provide reference output as **answer**
- Step 4: Add **AccuracyObserver**

See the example:

- [examples/cuda/accuracy.py](#)

- The **TunablePrecision** wrapper tells Kernel Tuner that type of input/output arguments depends on a tunable parameter
- Before benchmarking, data converted to provided data types
- During benchmarking, kernel is passed pointer of correct data type
- [Advanced] The general **Tunable** object allows arbitrary conversions

- The **AccuracyObserver** measures the error and adds a metric
- Supports 10+ well-known metrics
 - Root mean square error (**RMSE**)
 - Mean relative error (**rel**)
 - Mean absolute error (**abs**)
 - Maximum relative error (**max**)
 -
 - Custom metrics are also possible!
- The best error metric is application-dependent
- Compatible with other observers!

Hands-on

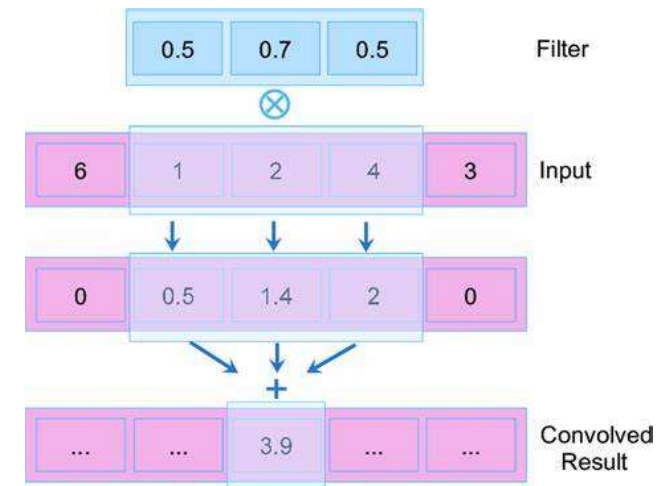


- The third hands-on notebook is:
 - https://colab.research.google.com/github/KernelTuner/kernel_tuner_tutorial/blob/master/energy/02_Mixed_precision_programming.ipynb

- The goal of this hands-on is to:
 - Tune a signal convolution kernel with mixed precision types
 - Experiment with the accuracy-performance trade-off

- [Open the notebook in Google Colab](#) and work there

- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors



```
# The tunable types. Currently, the code is only
tune_params = dict()
tune_params["OUTPUT_TYPE"] = ["double"]
tune_params["INPUT_TYPE"] = ["double"]
tune_params["FILTER_TYPE"] = ["double"]
tune_params["ACCUM_TYPE"] = ["double"]

# Other tunable parameters
tune_params["block_size x"] = [128, 256]
tune_params["VECTOR_SIZE"] = [1, 2, 4]
tune_params["PREFETCH_INPUT"] = [0, 1]
tune_params["UNROLL_LOOP"] = [0, 1]
```

Optimizing GPU Core Clock Frequency



Measuring power consumption with NVML

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

Advantages:

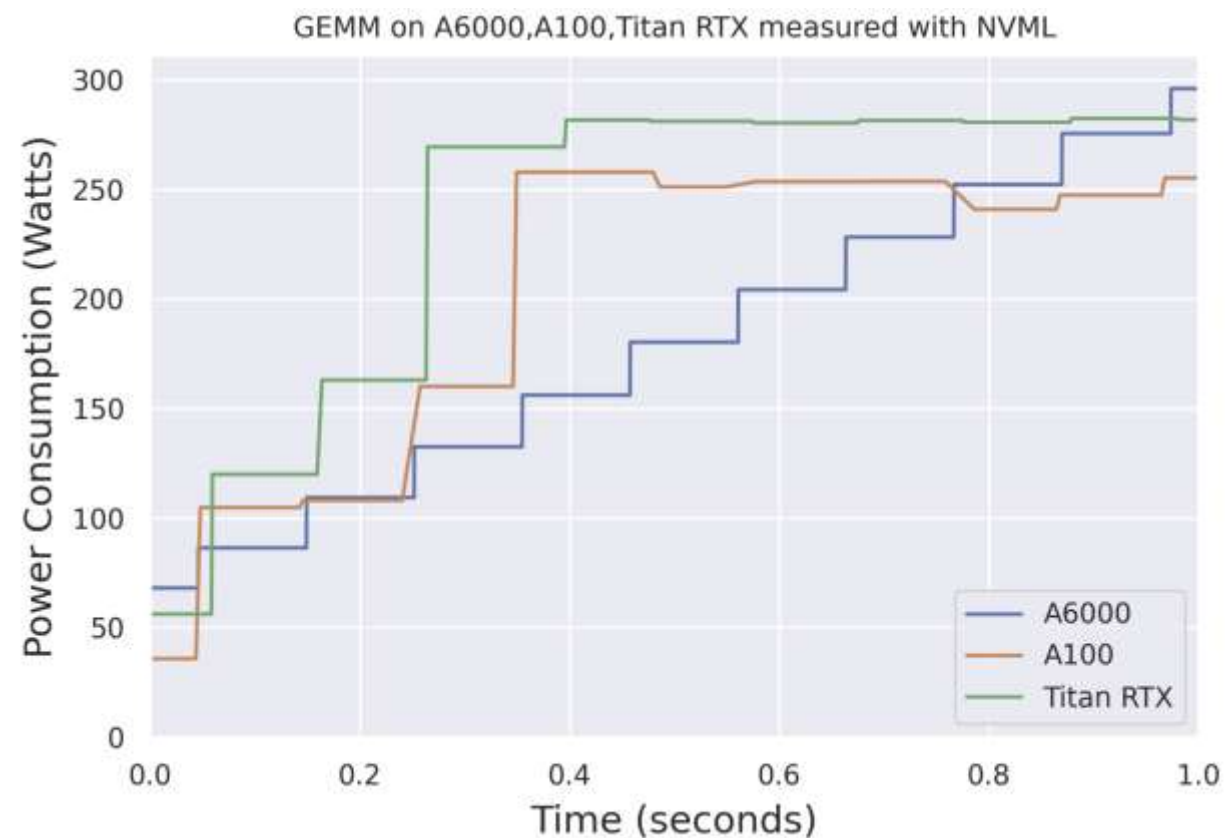
- Highly available

Disadvantages:

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

Current solution:

- Measure power while continuously running the kernel for one second

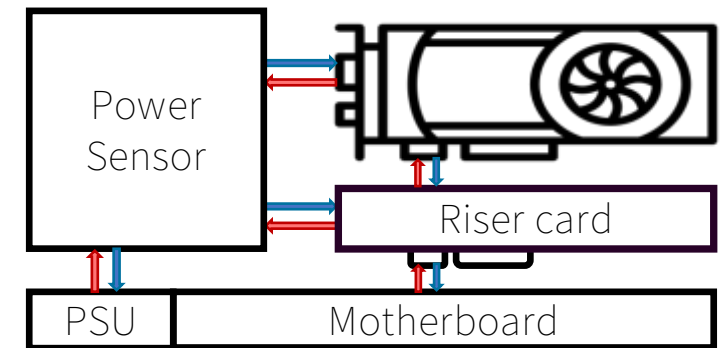


Pros:

- Instantaneous power readings
- Time resolution: 2.8 KHz
- Open source: <https://gitlab.com/astron-misc/PowerSensor>

Cons:

- Some assembly required
 - You need to build the hardware!



Supported in Kernel Tuner, using PowerSensorObserver

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`**: try out different power limits
 - **`nvml_gr_clock`**: set the GPU core clock frequency
 - **`nvml_mem_clock`**: set the GPU memory clock frequency
 - Setting these requires root privileges

- Kernel Tuner has helper functions to setup tunable parameters:

In `kernel_tuner.observers.nvml`:

- **`get_nvml_pwr_limits(device, n=None, quiet=False)`:**
 - Device is the device ordinal as reported by `nvidia-smi`
 - **`n`** is the number of evenly-spaced values to tune
 - if unspecified returns values spaced 5 Watts apart
- **`get_nvml_gr_clocks(device, n=None, quiet=False)`:**
 - **`n`** is the number of evenly-spaced values to tune
 - If unspecified, all supported core clocks are returned

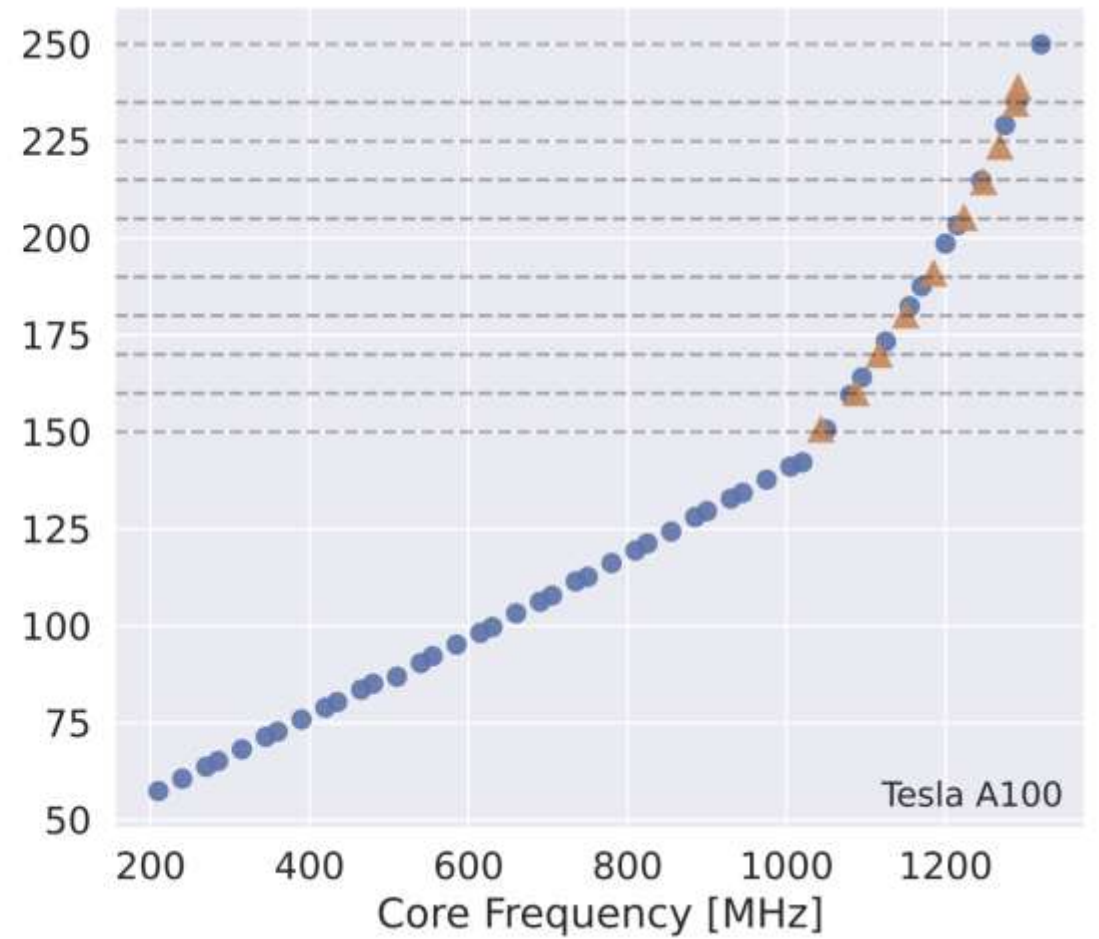
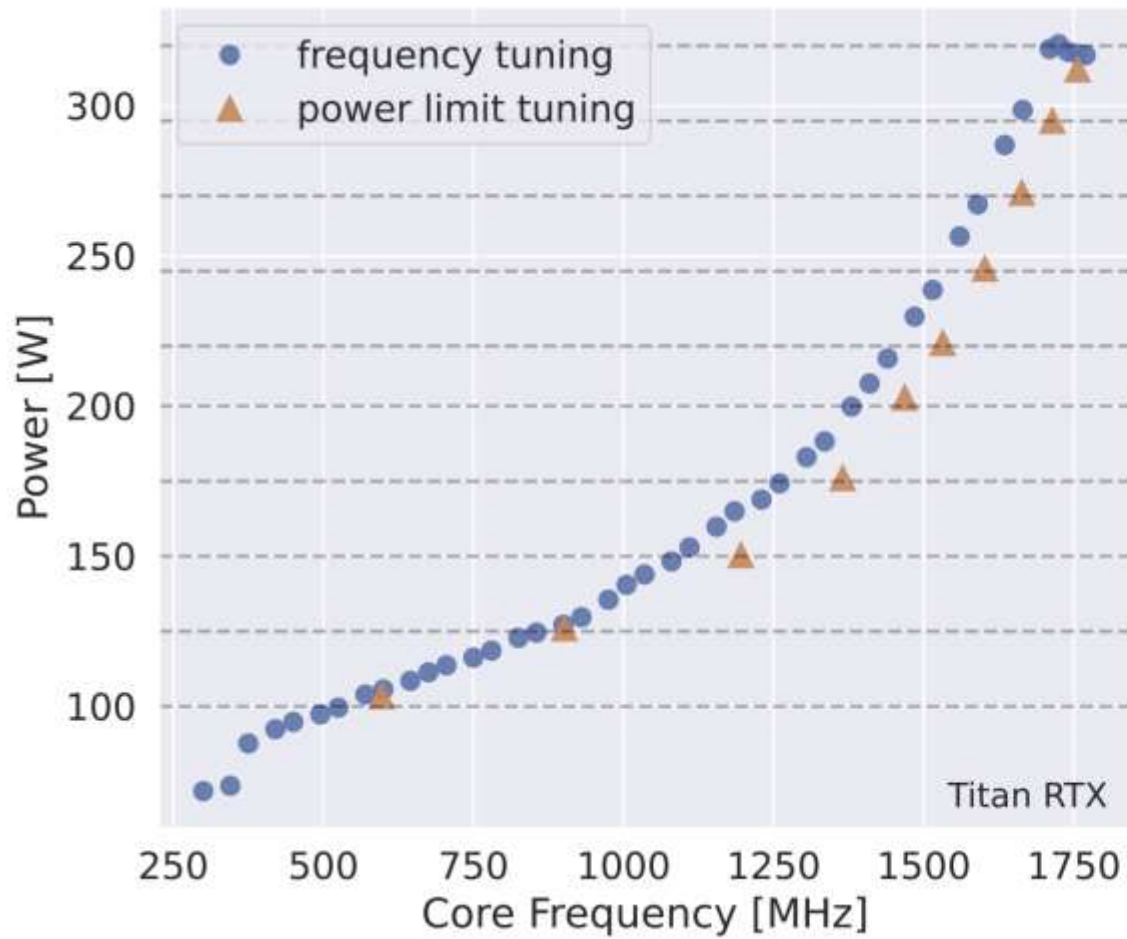
Many tunable parameters affect compute performance and/or energy efficiency

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 unfortunately require root privileges for the latest generations of Nvidia GPUs

Tuning CLBlast GEMM using frequency or power limit tuning



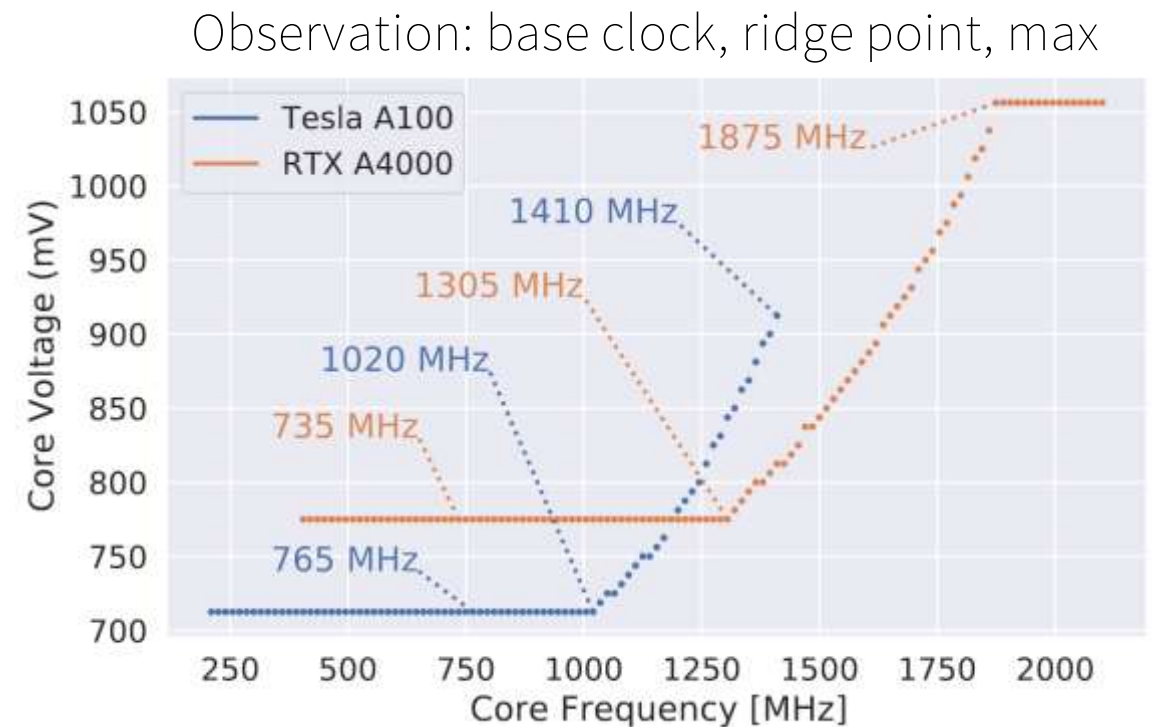
Advantages of power capping:

- 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

- 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’

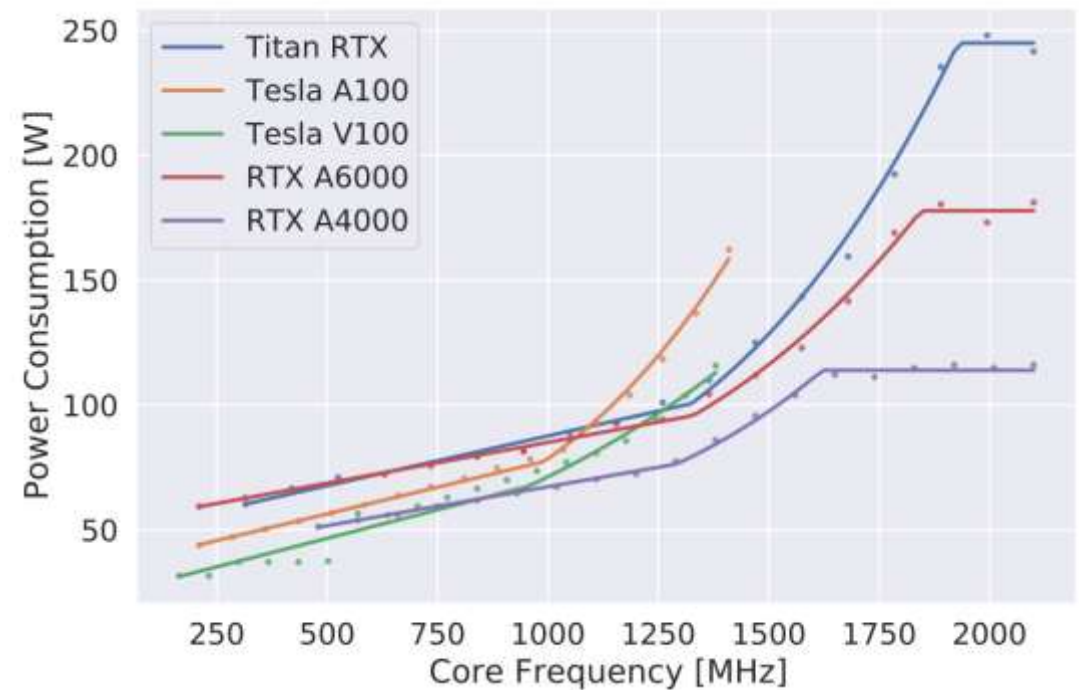


- Not every GPU reports core voltages, but we can estimate the voltage using a simple power model
- When we fix all parameters and vary the clock frequency, we can approximate power consumption using:

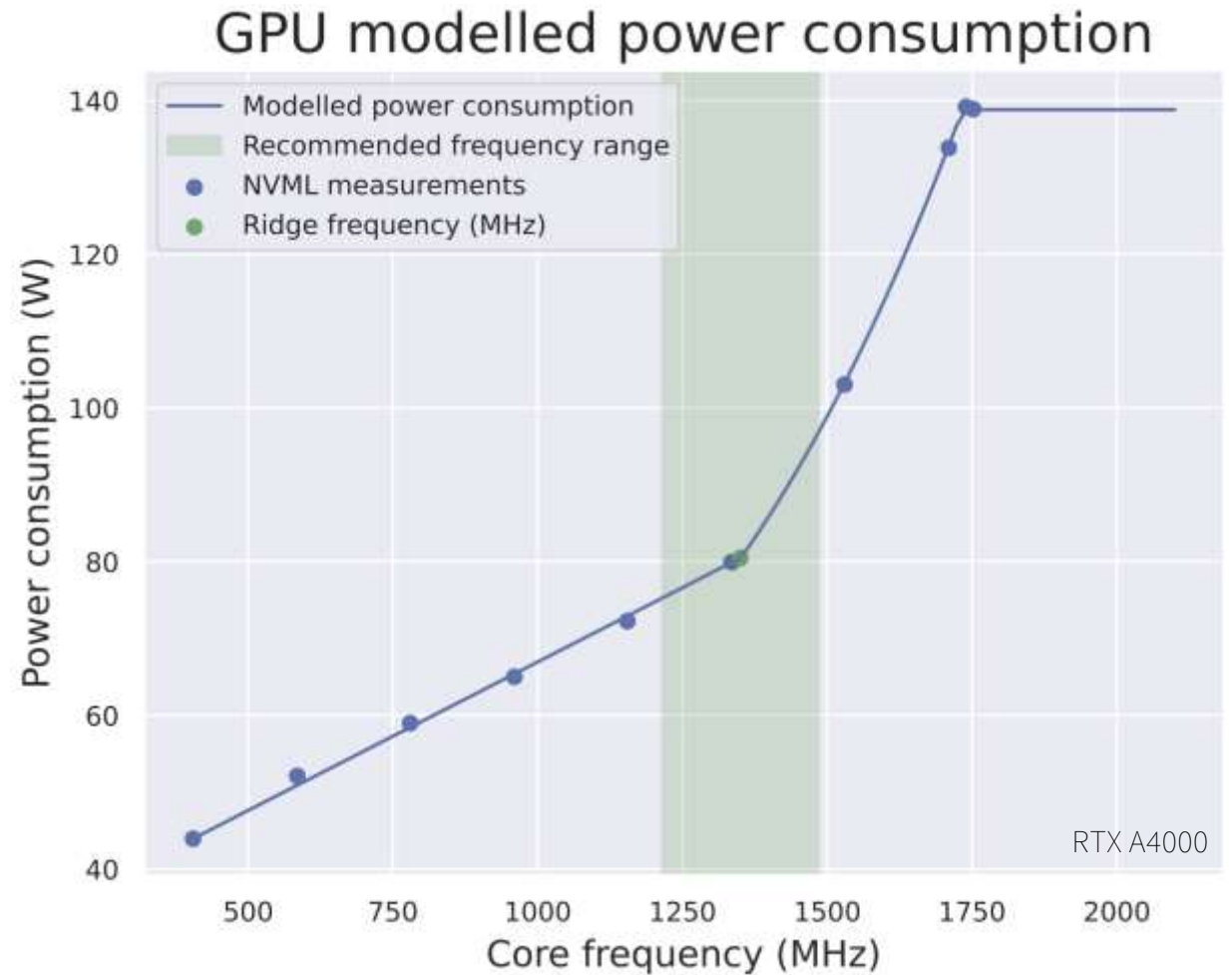
$$P_{load} = \min(P_{max}, P_{idle} + \alpha * f * v^2)$$

- And identify the GPUs ‘ridge point’ frequency in this way

Modeled Power Consumption



- Use performance model to limit the frequency range for tuning
- Reduces the search space by ~80% on average



- 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["nvm1_energy"]
```

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

Hands-on



- The fourth hands-on notebook is:
 - https://colab.research.google.com/github/KernelTuner/kernel_tuner_tutorial/blob/master/energy/03_energy_efficient_computing.ipynb
- The goal of this hands-on is to:
 - Tune a kernel to minimize the execution time or the energy consumption
 - Use an optimization strategy
 - Compare different energy optimization strategies
- [Open the notebook in Google Colab](#) and work there
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors

Closing Remarks

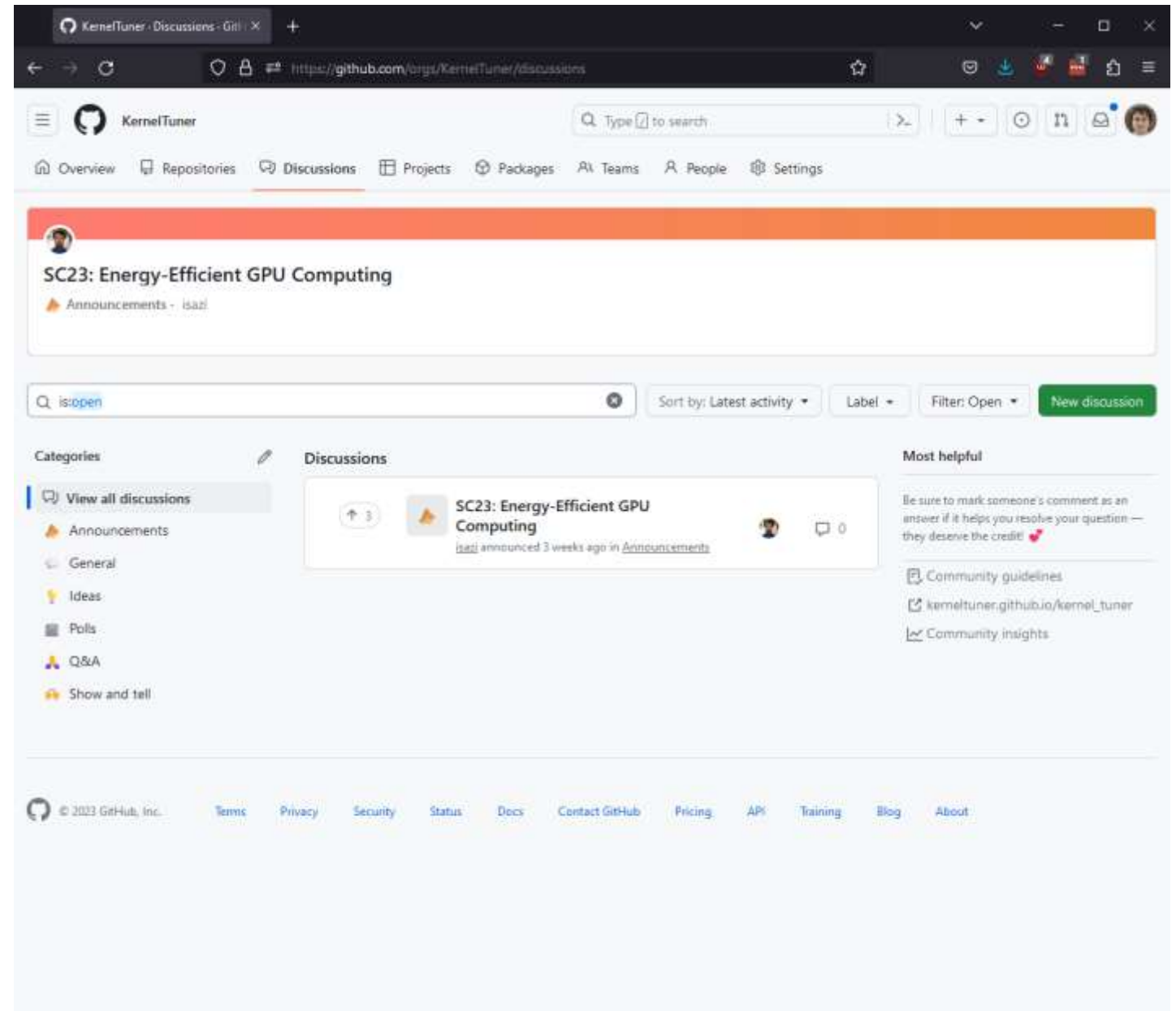


- We are developing Kernel Tuner as an open-source project
- GitHub repository:
 - https://github.com/KernelTuner/kernel_tuner
 - License: Apache 2.0
- If you use Kernel Tuner in a project, please cite the paper:
 - B. van Werkhoven, Kernel Tuner: A search-optimizing GPU code auto-tuner, *Future Generation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

- 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://kerneltuner.github.io/kernel_tuner/stable/contributing.html

Join the discussion!

- We have a discussion board on GitHub!



- Kernel Launcher: C++ library for creating optimal-performance portable CUDA applications
S. Heldens, B. van Werkhoven
International Workshop on Automatic Performance Tuning (iWAPT2023) co-located with IPDPS2023
- Optimization Techniques for GPU Programming
Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri Bal
*ACM Computing surveys*2023
- Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning
Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, K. Joost Batenburg
International Workshop on Performance Modeling, Benchmarking and Simulation of High-Performance Computer Systems (PMBS) at Supercomputing (SC22) 2022
- Bayesian Optimization for auto-tuning GPU kernels
F.J. Willemsen, R.V. van Nieuwpoort, B. van Werkhoven
International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) at Supercomputing (SC21) 2021
- Kernel Tuner: A search-optimizing GPU code auto-tuner
B. van Werkhoven
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).
- The COMPAS project has received funding from the Netherlands eScience Center (NLESC.OEC.2022.001).
- 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



Thanks!

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

Ben van Werkhoven

b.van.werkhoven@liacs.leidenuniv.nl