

GPU code optimization and auto-tuning made easy with Kernel Tuner: *A hands-on, bring your own code tutorial*

Ben van Werkhoven and Alessio Sclocco

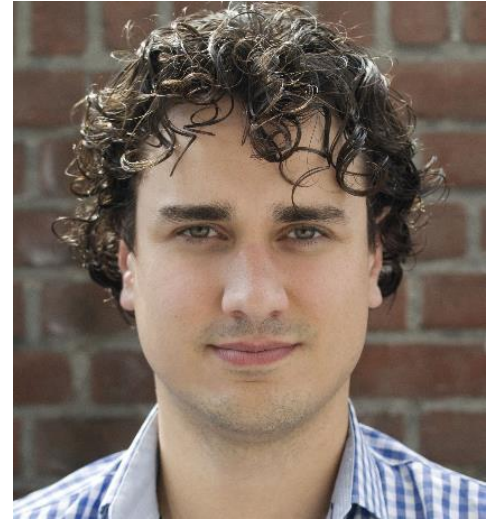
Netherlands eScience Center

netherlands
eScience center

November 7, 2022

Introduction: Ben van Werkhoven

Ben van Werkhoven
Senior Research Engineer
Netherlands eScience Center
b.vanwerkhoven@esciencecenter.nl



Background:

- 2010-2014 PhD “Scientific Supercomputing with Graphics Processing Units” at the VU University Amsterdam in the group of prof. Henri Bal
- 2014-now working at the Netherlands eScience Center as the GPU expert in many different scientific research projects

GPU Programming since early 2009, worked on applications in computer vision, digital forensics, climate modeling, particle physics, geospatial databases, radio astronomy, and localization microscopy



Introduction: Alessio Sclocco

Alessio Sclocco

Research Software Engineer

Netherlands eScience Center

a.sclocco@esciencecenter.nl



Background:

- 2017-2022: Research Software Engineer at the Netherlands eScience Center
 - Radio astronomy, climate modeling, biology, natural language processing, physics
- 2019: visiting scholar at Nanyang Technological University in Singapore
- 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
- 2012-2017: PhD "Accelerating Radio Astronomy with Auto-Tuning" at VU Amsterdam
- 2011-2012: junior researcher at VU Amsterdam
 - Working on GPUs for radio astronomy



Outline for the day

10:00	–	10:05	Opening and welcome
10:05	–	10:30	Introduction to auto-tuning with Kernel Tuner
10:30	–	10:45	First hands-on session
10:45	–	11:15	Getting started with Kernel Tuner: integrating code, multi-dimensional problems, user-defined metrics
11:15	–	11:45	Second hands-on session
11:45	–	13:00	Lunch
13:00	–	13:40	Intermediate topics: GPU code optimizations, output verification, search space restrictions, caching tuning results
13:40	–	14:10	Third hands-on session
14:10	–	14:45	Advanced topics: creating performance portable applications, search optimization strategies, custom observers
14:45	–	15:00	Fourth hands-on session
15:00	–	16:00	Drinks

Administrative announcements

- We will have four sessions in which we start with introducing some new concepts and follow with a hands-on exercise
- The hands-on exercises include example kernels, but you are also welcome to experiment with your own code
- We will use SURF JupyterHub for the hands-on, so you don't need to have access to a GPU or install anything locally
- You can download the slides and the hands-on notebooks here:
 - https://github.com/KernelTuner/kernel_tuner_tutorial



Outline for this session

- Overview of auto-tuning technologies
- Introduction to Kernel Tuner
- Kernel Tuner installation and setup
- First hands-on session

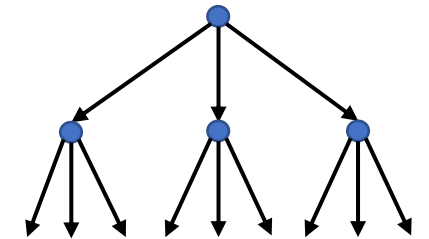
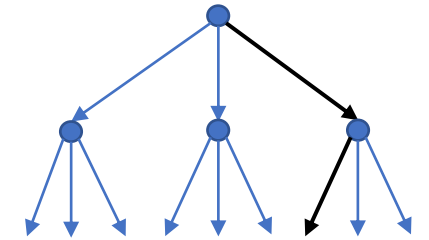


Overview of auto-tuning technologies



Manual optimization versus auto-tuning

- 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 versions of the code



Compiler vs software auto-tuning

- Compiler auto-tuning
 - Does not modify the source code
 - Needs to work with assumptions the compiler can take
 - Usually tunes compiler flags for the application as a whole
- Software auto-tuning
 - Works with a code template or code generator
 - Can tune more diverse code optimizations and even competing implementations of the same algorithm
 - Usually tunes individual kernels or pipelines of kernels



Model-based versus empirical tuning

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



Run-time vs compile-time auto-tuning

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



Summary

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



Kernel Tuner



Auto-tuning GPU applications

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
- Work per thread in each dimension
- Loop unrolling factors
- Overlapping computation and communication
- ...

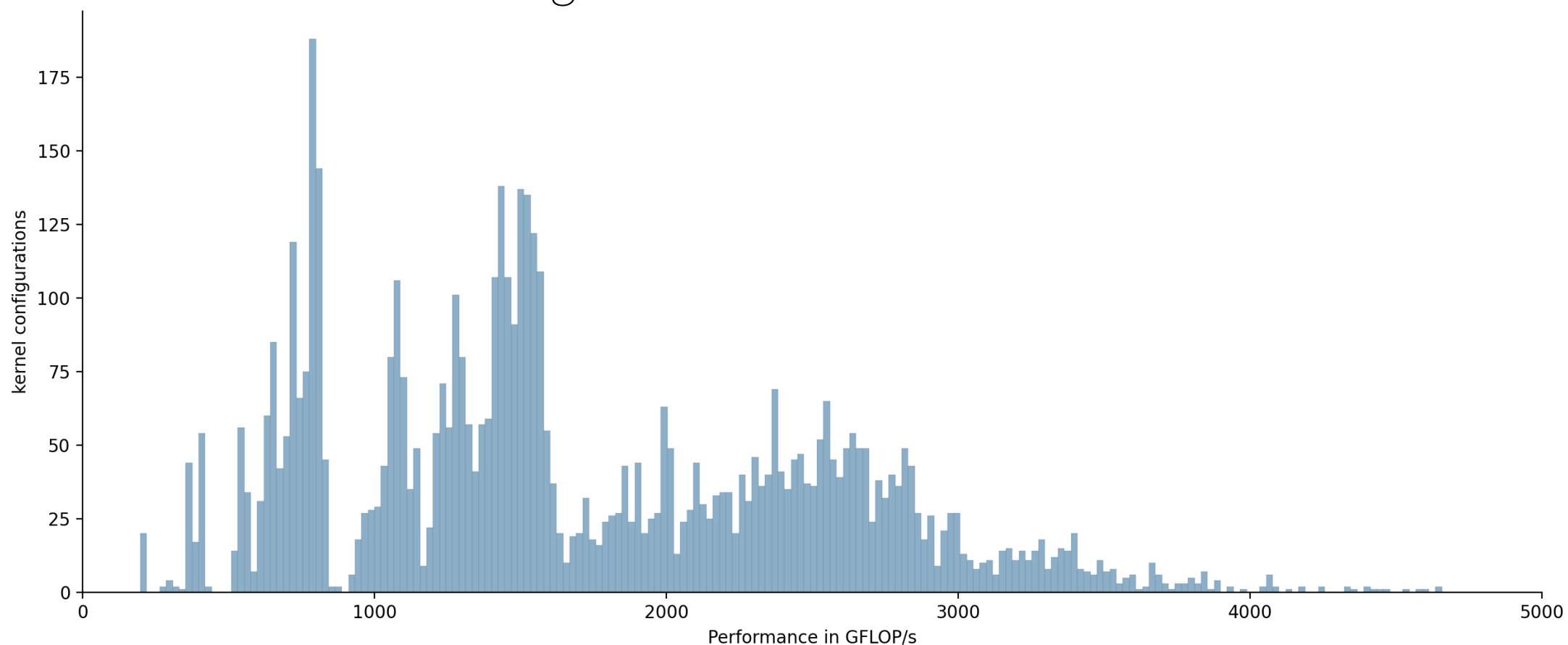
Problem:

- Creates a very large design space



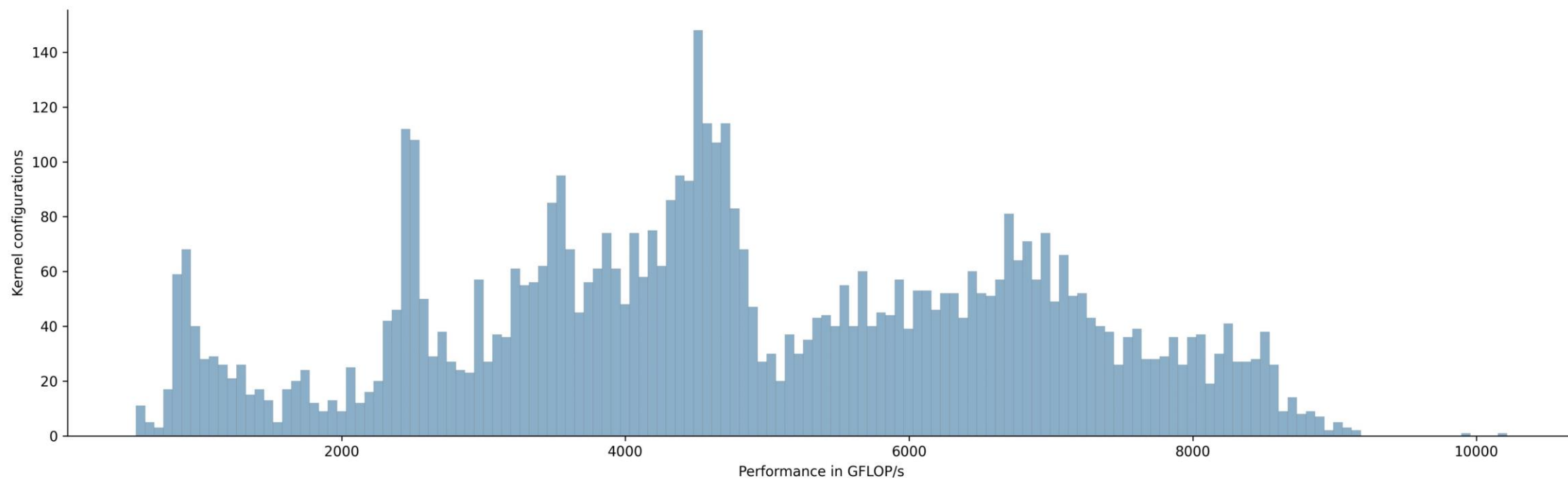
Large search space of kernel configurations

Auto-tuning a GEMM kernel on GTX Titan X



Large search space of kernel configurations

Auto-tuning a Convolution kernel on Nvidia A100



Kernel Tuner

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
- Using code generators
- Unit testing GPU code
- ...

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)
```



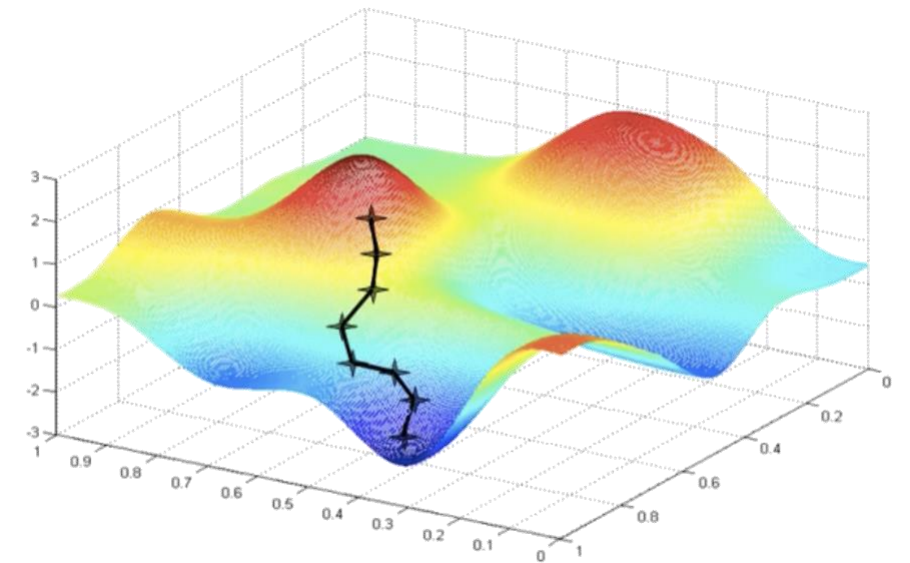
What it does

- Compute the Cartesian product of all tuning parameters
 - Remove instances that fail any of the restrictions
- For each instance in the parameter space (brute force tuning):
 - 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 with the averaged run time for each instance

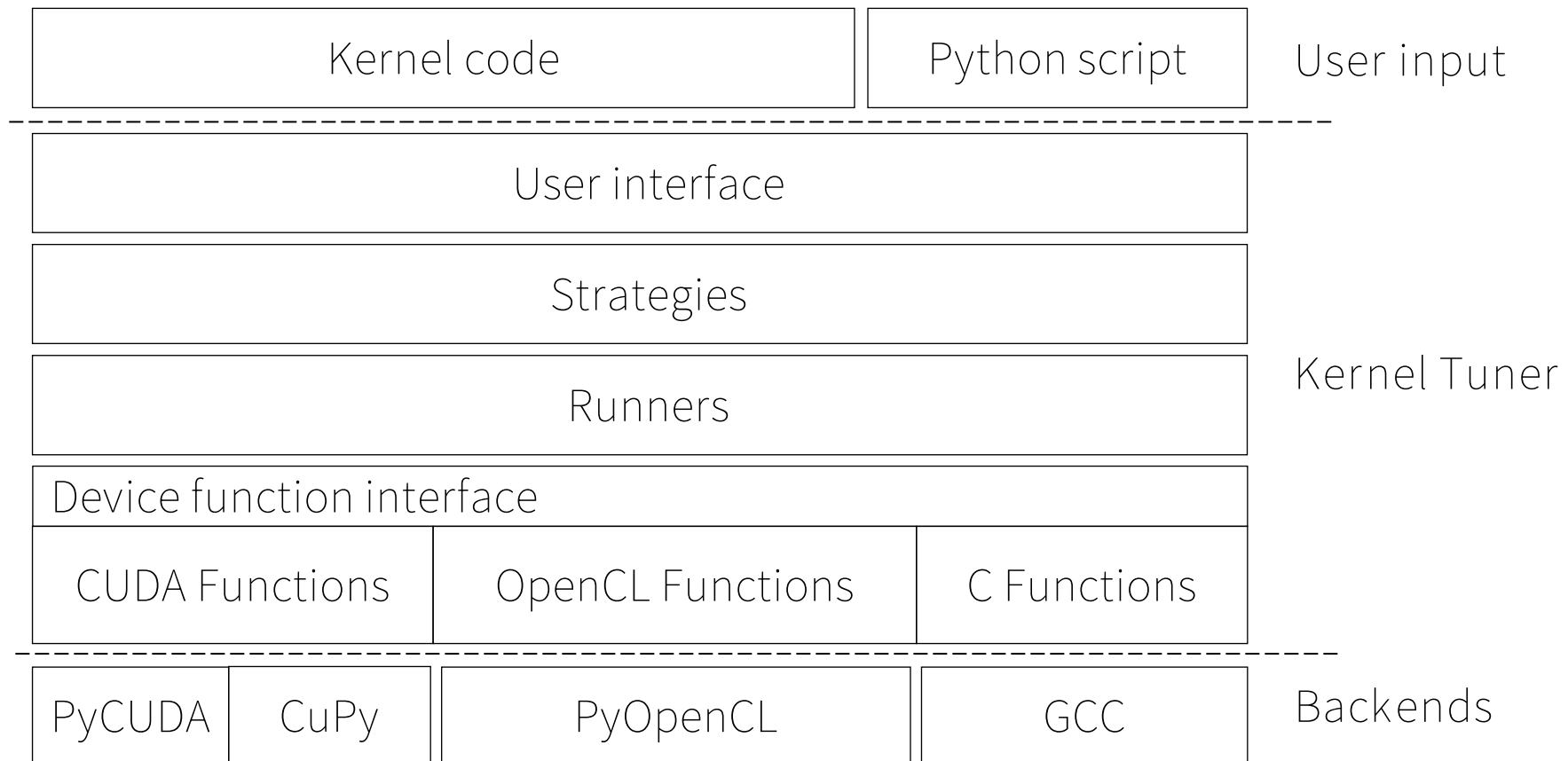


Optimization strategies in Kernel Tuner

- 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
 - ...



Kernel Tuner architecture



Installation on your system

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



First hands-on session



Introduction hands-on

- The first hands-on notebook is:
 - https://github.com/KernelTuner/kernel_tuner_tutorial/blob/master/hands-on/cuda/00_Kernel_Tuner_Introduction.ipynb
- The goal of this hands-on is to **import** and **run** Kernel Tuner
 - You will find the notebook in your JupyterHub environment
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors



Optional hands-on

- Done with the first hands-on already?
- Play with this other notebook
 - https://github.com/KernelTuner/kernel_tuner_tutorial/blob/master/hands-on/cuda/Kernel_Tuner_Tutorial.ipynb
- Start experimenting with your own code
- Feel free to ask questions to instructors and mentors



Getting started outline

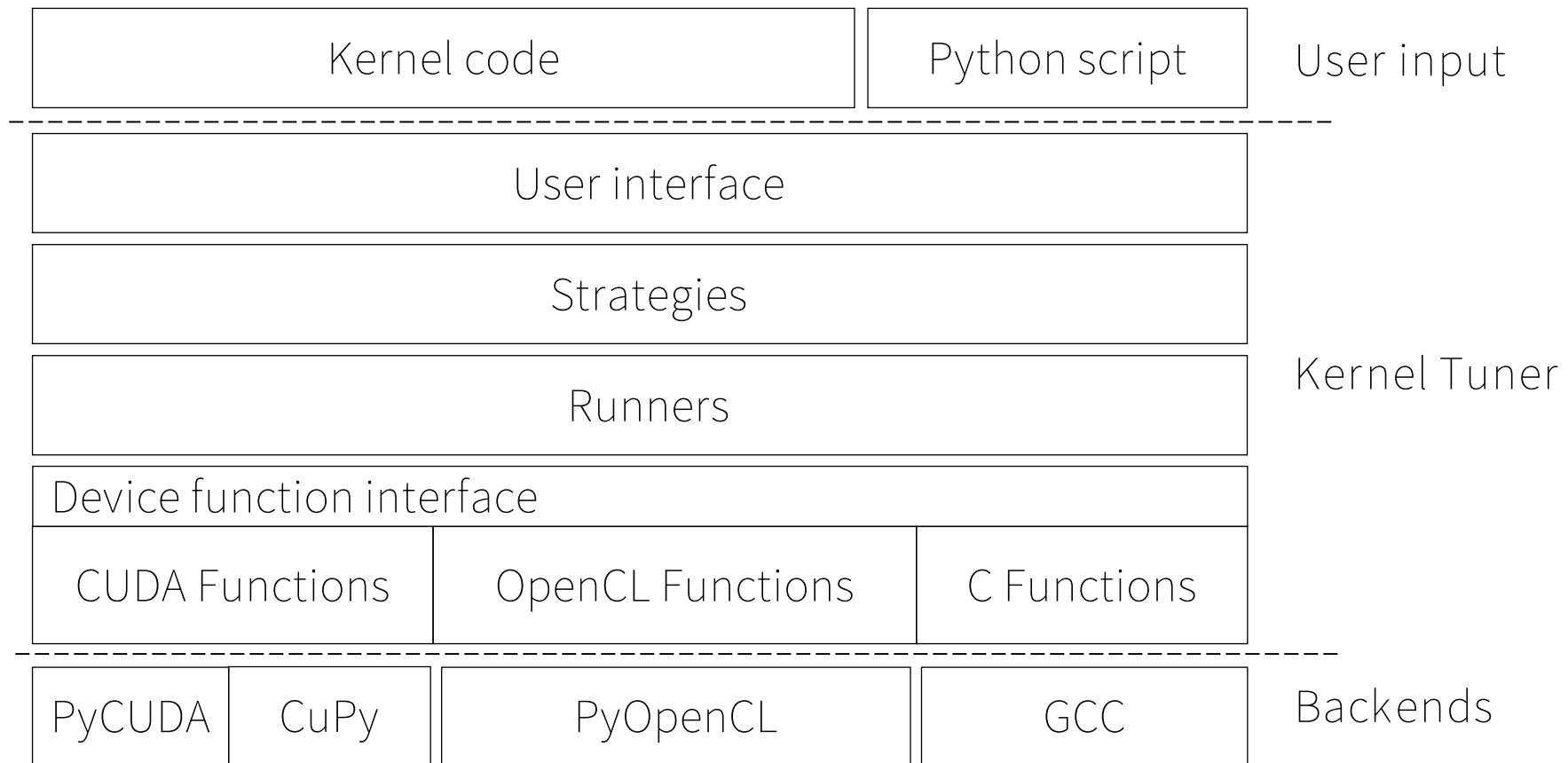
- Integrating code into Kernel Tuner
- Grid and thread block dimensions
- User-defined metrics
- Second hands-on session
- Lunch break



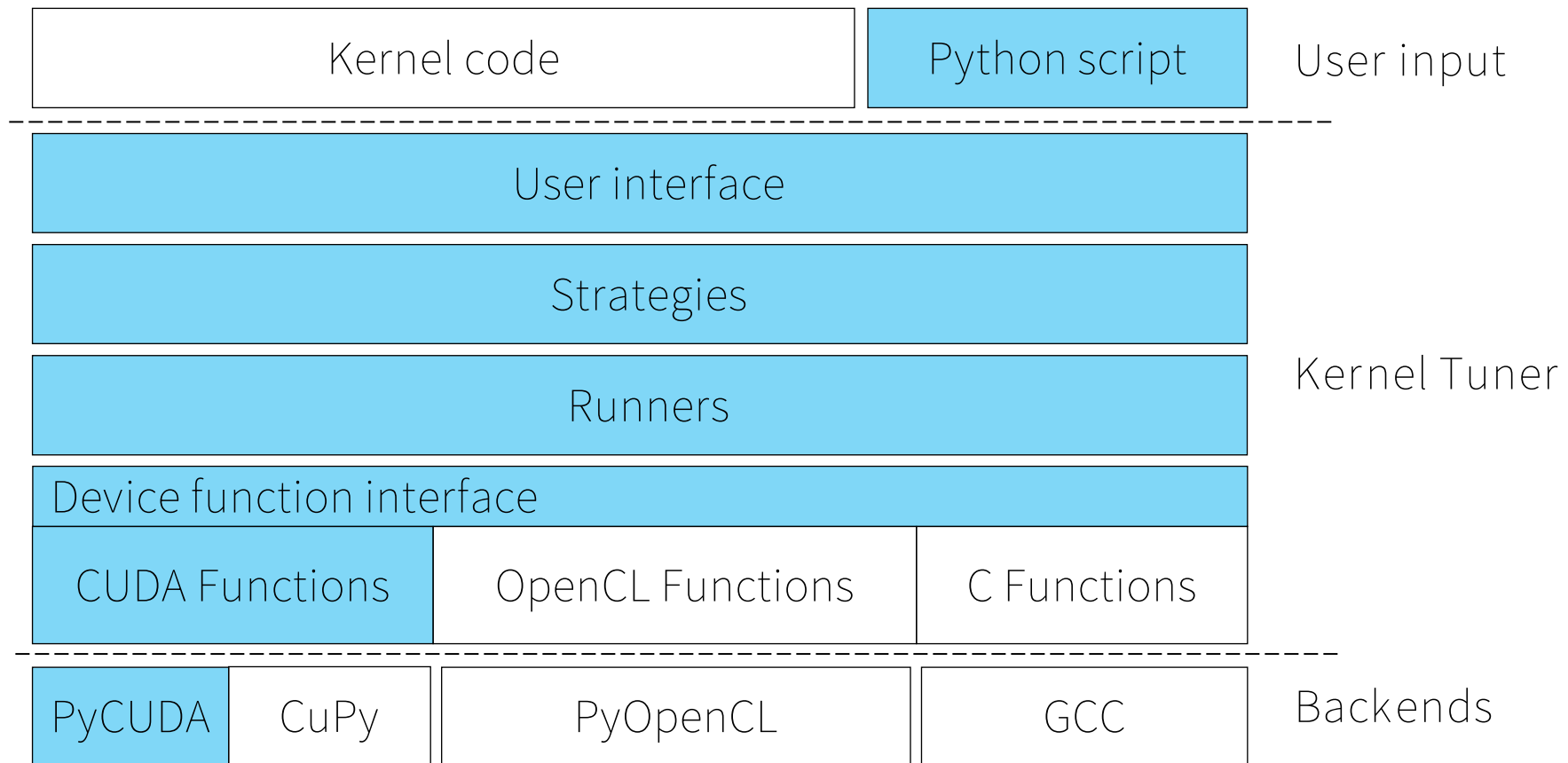
Kernel Tuner code integration



Kernel Tuner architecture



Kernel Tuner architecture



Kernel Tuner compiles and benchmarks many kernel configurations

- We need to tell Kernel Tuner everything that is needed to compile and run our kernel, including 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, so we need to load/create the 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 auxiliary 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.



Specifying Kernel source code

- The 2nd positional argument of **tune_kernel()** is **kernel_source**
- **kernel_source** can be a string with the code, filename, or a function
 - The function option is useful for [code generators](#) or when using a templating engine such as [jinja](#)
- Kernel Tuner automatically detects the programming language and selects a backend
 - Defaults are PyCUDA for CUDA and PyOpenCL for OpenCL
 - If you want to select a different backend use the **lang** option, e.g. **lang="CuPy"** to select the CuPy backend



Kernel compilation

- Kernel Tuner will compile the same kernel repeatedly with different parameters inserted
- If your code includes many headers with all kinds of template expansions the compilation time may become prohibitive
- Recommendation: isolate device code from the rest of your application
 - Easy trick is to put kernel code in separate source files and include these in the host code where needed



Kernel arguments

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



Kernel argument types

While NumPy arrays can be of mixed types to mimic more complex types, these can be difficult to reconstruct correctly in Python.

It may be difficult to use Kernel Tuner on kernels with custom types for input arrays instead of simple arrays of primitive types.

A general performance recommendation for GPU programming is to not use arrays of structs if possible.



Using a struct as kernel argument

- Using Python's built-in **struct.pack()** and Numpy's **np.frombuffer()**

```
import struct
import numpy as np
import kernel_tuner as kt

def create_receive_spec_struct():
    ...
    packstr = struct.pack('iiiiiiiiiiPPi?fffi?0l', #the 0l at the end ensures padding to the next long (8bytes)
                          nSamples, nSamplesIQ, nSlowTimeSamples, nChannels,
                          nTX, nRepeats, nFastTimeSamples, rfSize, mNRows, mNRowsIQ,
                          nActiveChannels, 0, 0, 0, isIQ, Fs, FsIQ, Fc, nBuffers, initialized)
    return np.frombuffer(np.array(packstr), np.dtype((np.void, len(packstr))))[0]

receive_spec = create_receive_spec_struct()

...

args = [bf, rf, receive_spec, recon]

kt.tune_kernel(kernel_name, kernel_source, problem_size, args, tune_params,
               compiler_options=compiler_options, lang="CUDA", restrictions=restrict)
```



Overview

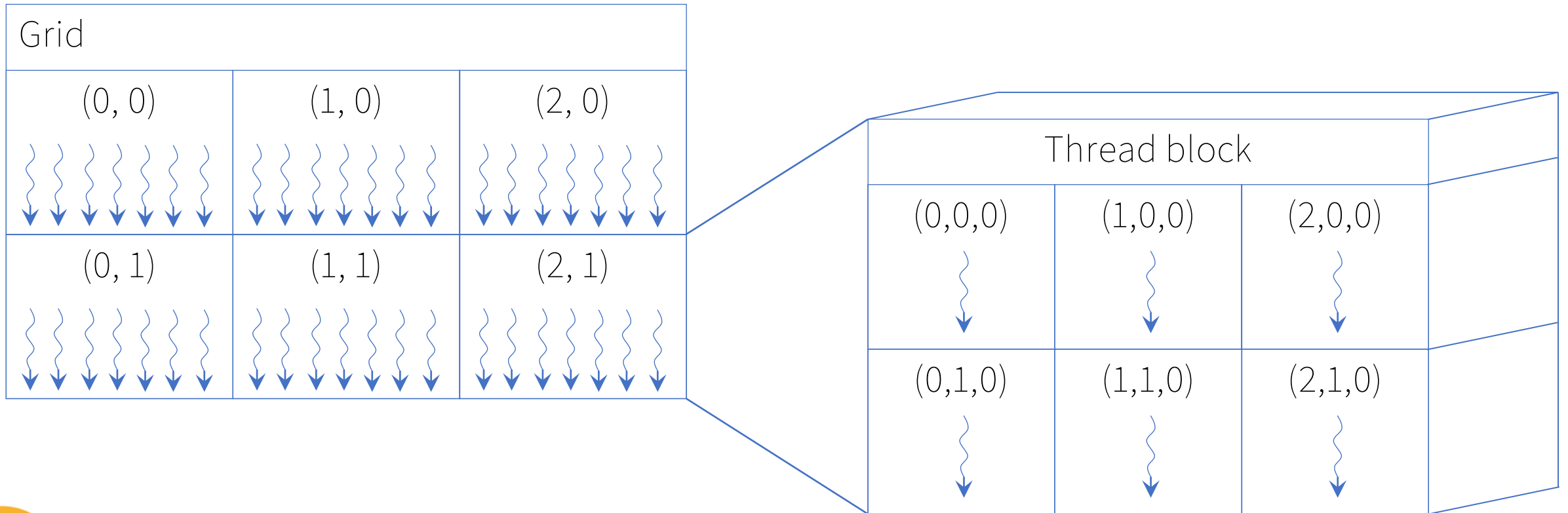
- For tuning CUDA kernels, Kernel Tuner uses PyCUDA, CuPy, or cuda-python
 - PyCUDA forces kernels to have **extern "C"** linkage
 - Templates supported using on-the-fly wrapper generation by Kernel Tuner
 - CuPy fully supports C++ and templates, but no host code is allowed
 - Cuda-Python needs **extern "C"** linkage, host code not allowed
- General recommendations:
 - Use simple types for kernel arguments
 - Isolate device code from host code to simplify separate compilation
 - Using arrays of structs is not recommended for performance



Grid and Thread block dimensions



CUDA Thread Hierarchy



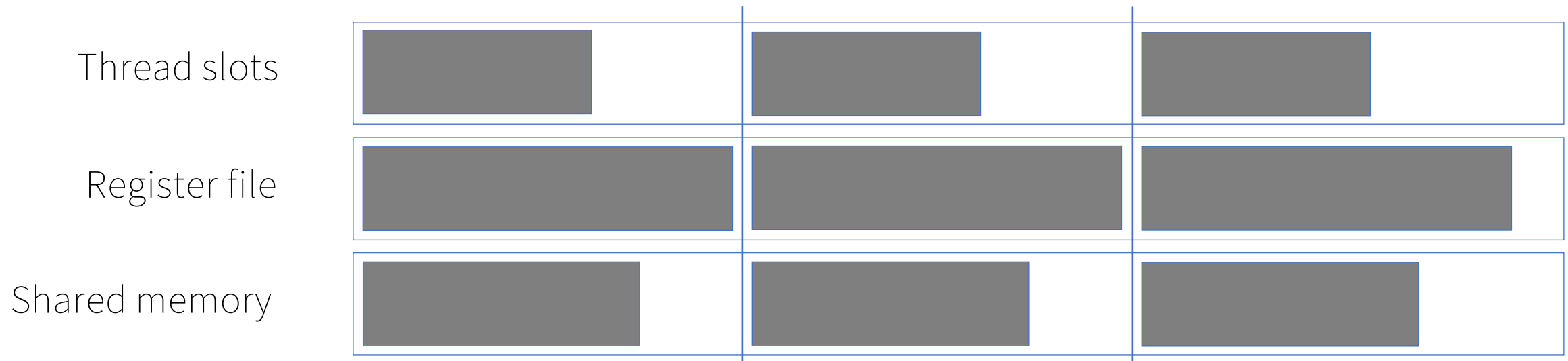
Choosing thread block dimensions

- 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?



Why do thread block dimensions matter so much?

- The GPU consists of several (1 to 80) *streaming multiprocessors* (SMs)
- The SMs are fully independent and contain several resources:
 - Register file, Shared memory, Thread Slots, and Thread Block slots
- SM resources are dynamically partitioned among the thread blocks that execute concurrently on the SM, resulting in a certain *occupancy*



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]}

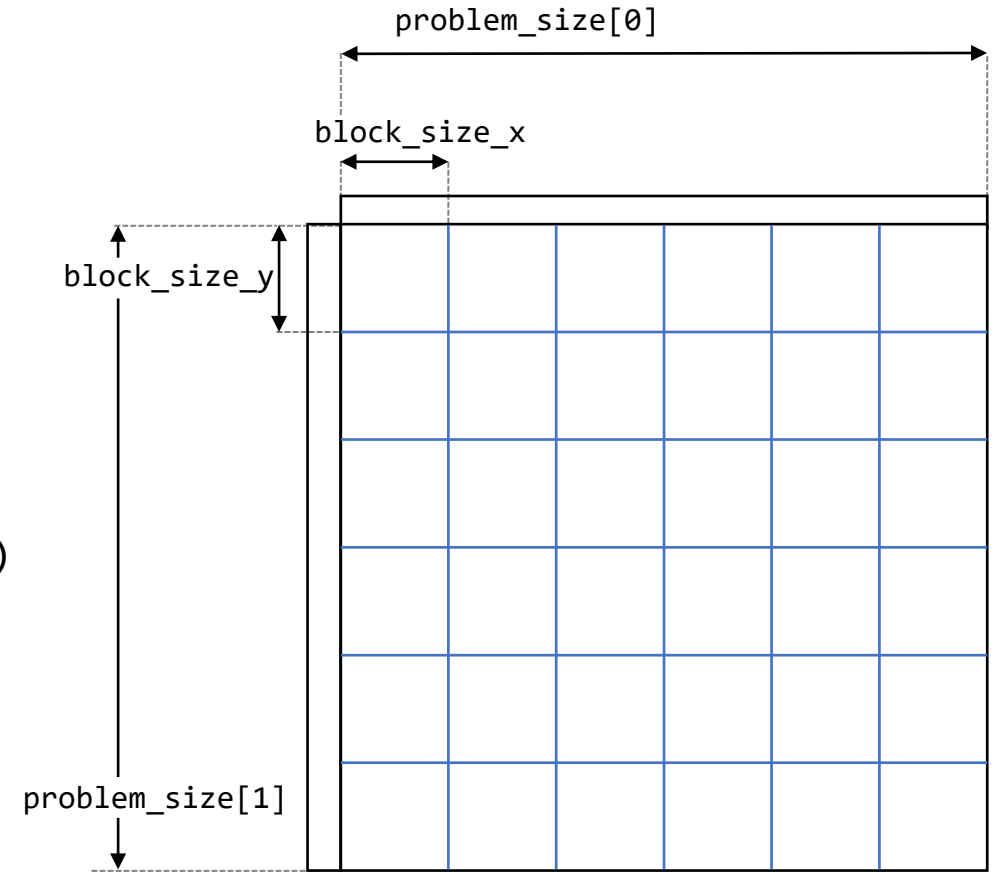
tune_kernel("vector_add", kernel_string, n, args, tune_params)
```



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

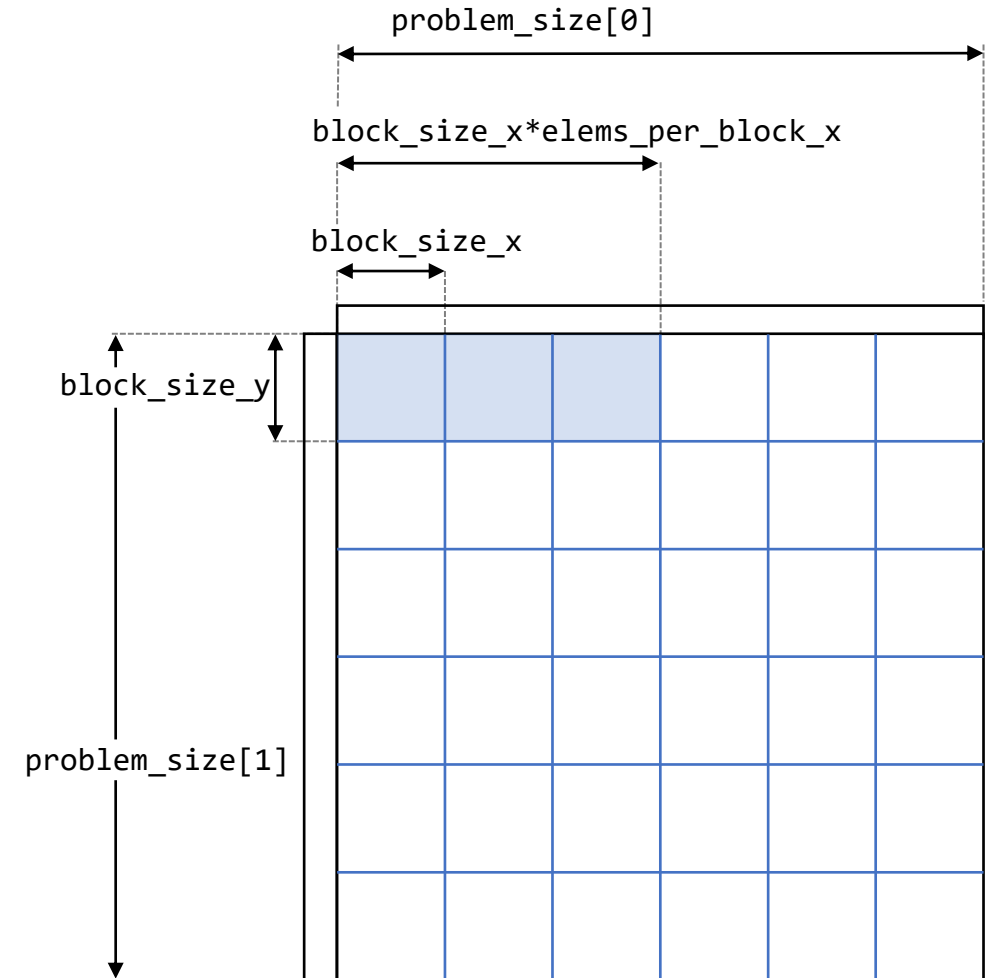
Specifying grid dimensions

- You specify the **problem_size**
- **problem_size** describes the dimensions across which threads are created
- By default, the grid dimensions are computed as:
 - $\text{grid_size_x} = \text{ceil}(\text{problem_size_x} / \text{block_size_x})$



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



problem_size

- 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 = 800000000
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 = []
```



User-defined metrics



Metrics

- Kernel Tuner only reports time during tuning in milliseconds (ms)
 - This is the averaged time of (by default 7 iterations), you can change the number of iterations using the **iterations** optional argument
 - 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



User-defined metrics

- 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)
```



Second hands-on session



Getting started hands-on

- The second hands-on notebook is:
 - https://github.com/KernelTuner/kernel_tuner_tutorial/blob/master/hands-on/cuda/01_Kernel_Tuner_Getting_Started.ipynb
- The goal of this hands-on is to experiment with **tunable grid dimensions** and **user defined metrics**
 - You will find the notebook in your JupyterHub environment
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors



Optional hands-on

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



Intermediate topics session outline

- GPU Optimizations
- Output verification
- Search space restrictions
- Caching tuning results
- Third hands-on session
- Break



GPU optimizations



GPU code optimizations

- Modify the kernel code in an attempt 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



Overview of GPU Optimizations

- 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



Overview of GPU Optimizations

- 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



(Partial) Loop Unrolling

- Why?
 - Increases instruction-level-parallelism
 - Reduces loop overhead instructions
- How?
 - In the early days, only manually or with a code generator
 - Compiler does this now: **#pragma unroll <value>**
 - In CUDA, value has to be integer constant expression
 - 0 is not allowed, 1 means unrolling is disabled
 - Remember, Kernel Tuner inserts parameters with **#define**
 - Parameters that start with **loop_unroll_factor_** are inserted as integer constant expressions instead, on 0 KT removes line with pragma



Partial loop unrolling

...

```
#pragma unroll loop_unroll_factor_nlay  
for (int ilay=0; ilay<nlay; ++ilay) {  
    ...  
}
```

The compiler can unroll this loop if **nlay** is known at compile-time. The **loop_unroll_factor_nlay** parameter should be a divisor of **nlay**.



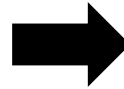
Reducing register usage

- 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 are very effective ways of reducing register usage
 - In kernels that do many different things, splitting the kernel may help to cut down register usage
 - Enabling register spilling with compiler flag **-maxrregcount=N** or tuning the number of blocks per SM using the kernel **__launch_bounds__()**



Reducing register usage

```
template<typename TF> __global__
void some_kernel(const int ncol,
                 const int nlay,
                 const int ngpt,
                 const int top_at_1, TF* flux_dn)
{
    const int icol = blockIdx.x*blockDim.x + threadIdx.x;
    const int igpt = blockIdx.y*blockDim.y + threadIdx.y;
    if ( (icol < ncol) && (igpt < ngpt) )
    {
        if (top_at_1)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

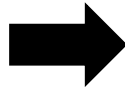


```
template<typename TF, int top_at_1> __global__
void some_kernel(const int ncol,
                 const int nlay,
                 const int ngpt,
                 TF* flux_dn)
{
    const int icol = blockIdx.x*blockDim.x + threadIdx.x;
    const int igpt = blockIdx.y*blockDim.y + threadIdx.y;
    if ( (icol < ncol) && (igpt < ngpt) )
    {
        if (top_at_1)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```



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

In Python:

```
tune_params["blocks_per_sm"] = [1, 2, 3, 4, 5, 6, 7, 8]
```



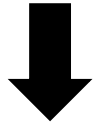
Varying work per thread

- 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

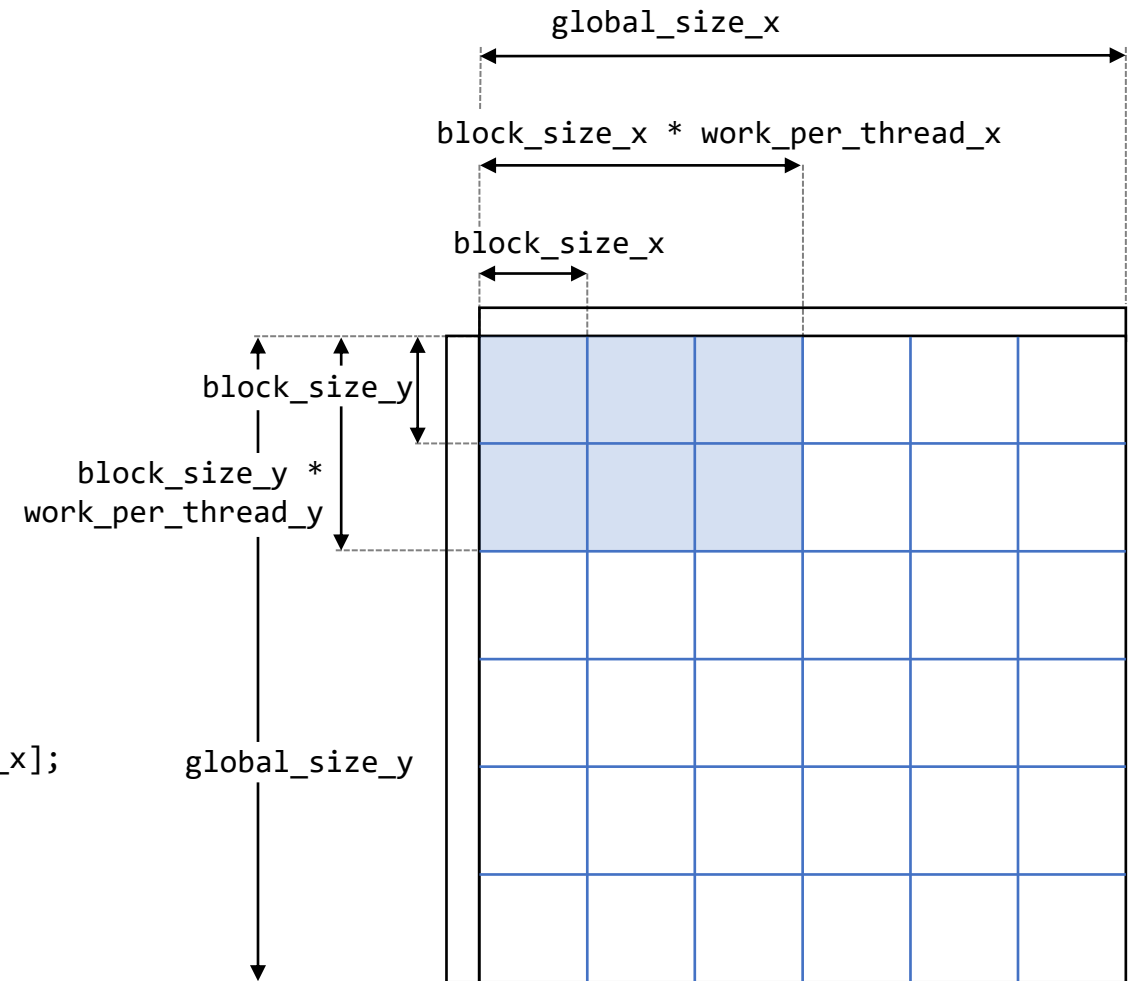


Varying work per thread

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



Vectorization

- Why?
 - Reduces the instructions needed to fetch data from global memory
 - Improves memory throughput
 - Often also increases work per thread and instruction-level parallelism
 - May increase register usage
- How?
 - Using wider data types (e.g. **float2** or **float4** instead of **float**)
 - Vector length can be tuned



Vectorization


```
#if (vector==1)
#define floatvector float
#elif (vector == 2)
#define floatvector float2
#elif (vector == 4)
#define floatvector float4
#endif

__global__ void sum_floats(float *sum_global, floatvector *array, int n) {

    int x = blockIdx.x * block_size_x + threadIdx.x;
    if (x < n/vector) {
        floatvector v = array[x];

        ...
    }

    ...
}
```

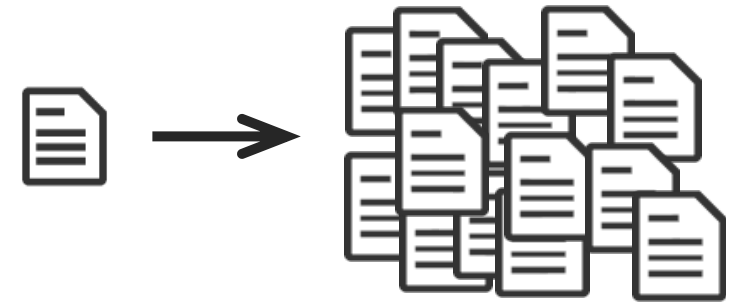


Output verification



Programming tunable applications

- 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!



Output verification

- 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, but first a simple example



Simple answer example

```
args = [c, a, b, n]
```

```
answer = [a+b, None, None, None]
```

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



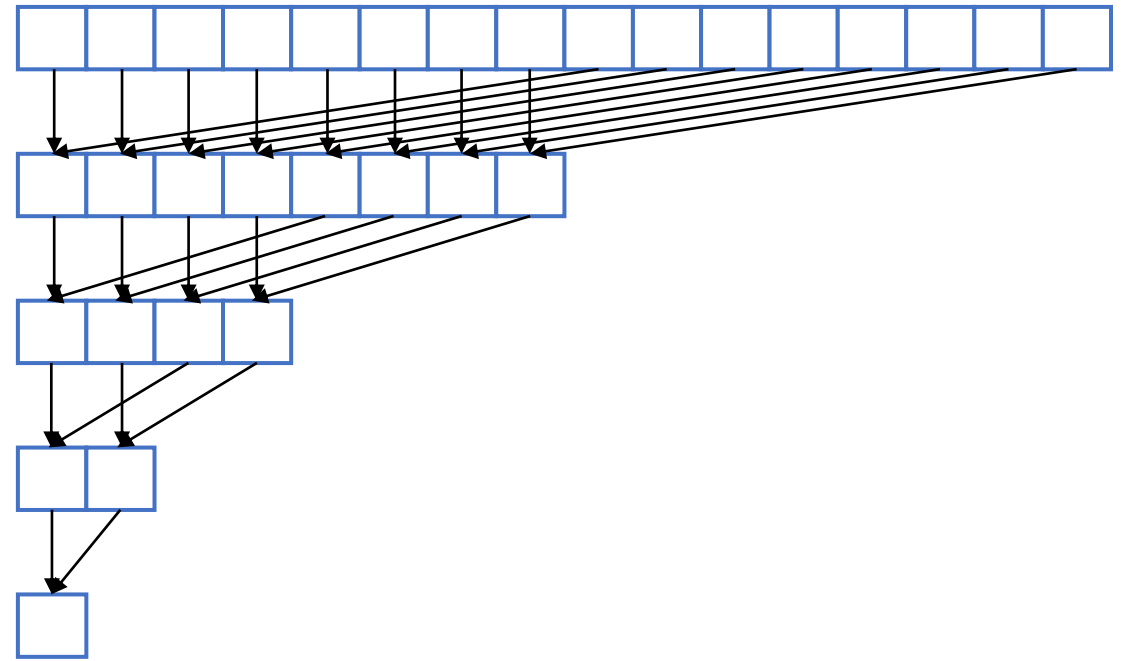
Custom verify functions

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



Custom verify example - reduction

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



Custom verification function - wrong

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

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

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



Custom verification function

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

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

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

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



Search space restrictions



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["vector"] = [1, 2, 4]  
tune_params["use_shared_mem"] = [0, 1]
```

- 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

- In this example:
 - We have a parameter that controls a loop count, **tile_size_x**
 - And we want to also tune the partial loop unrolling factor of that loop, using a parameter named **loop_unroll_factor_x**
- Kernel Tuner considers the Cartesian product of all possible values of both parameters as the search space
- But only configurations in which **loop_unroll_factor_x** is a divisor of **tile_size_x** are valid



Partial loop unrolling example

```
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]
```

```
restrictions = lambda p: p["loop_unroll_factor_x"] <= p["tile_size_x"] and  
                        p["tile_size_x"] % p["loop_unroll_factor_x"] == 0
```



Caching tuning results



Caching

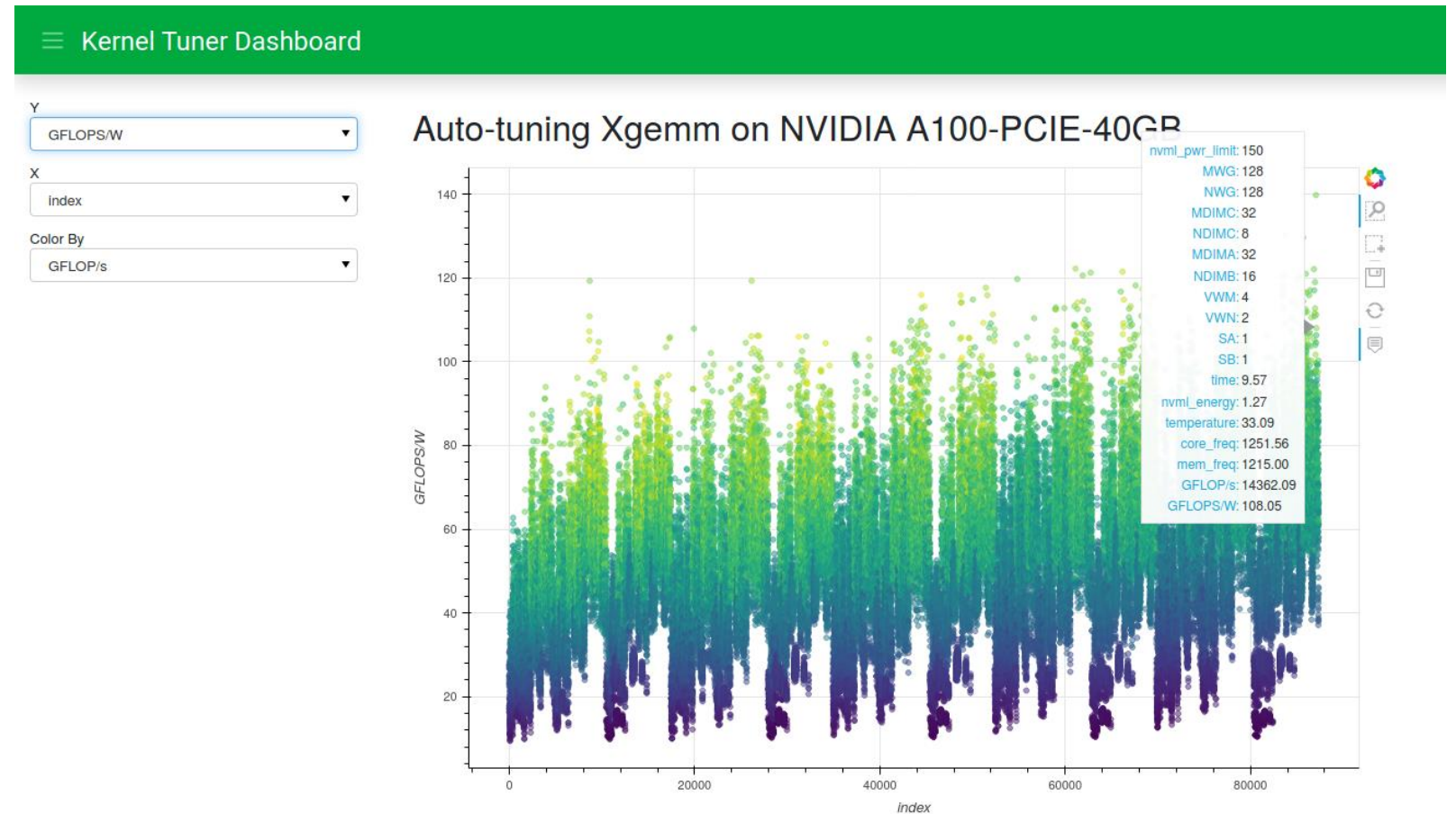
- Tuning large search spaces can take very long
- You might need to stop and continue later on
- Caching is enabled by passing a filename to the **cache** option
- Kernel Tuner will append new results to the cache directly after benchmarking a kernel configuration
- Kernel Tuner detects existing (possibly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



Kernel Dashboard

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

Live visualizations
to monitor your
Kernel Tuner runs



Simulation runner

- In the next part of this tutorial, we will look into using optimization strategies
- Cache files can also be used to quickly benchmark different optimization strategies or tune hyperparameters
- To use the simulation runner set **simulation_mode=True** with an existing **cache** file that contains information on *all* configurations in the search space



Third hands-on session



Intermediate hands-on

- The third hands-on notebook is:
 - https://github.com/KernelTuner/kernel_tuner_tutorial/blob/master/hands-on/cuda/02_Kernel_Tuner_Intermediate.ipynb
- The goal of this hands-on is to experiment with **search space restrictions, caching, and output verification**
 - You will find the notebook in your JupyterHub environment
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors



Optional hands-on

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



Advanced topics session outline

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



Performance portable applications



Performance portability

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



store_results

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

```
from kernel_tuner.integration import store_results
```

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

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



Compile-time kernel selection

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



Compile-time kernel selection example

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

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



vector_add.h

```
/* header file generated by Kernel Tuner, do not modify by hand */  
#pragma once  
#ifndef kernel_tuner /* only use these when not tuning */  
  
#ifdef TARGET_A100_PCIE_40GB  
#define block_size_x 672  
  
#elif TARGET_RTX_A6000  
#define block_size_x 160  
  
#else /* default configuration */  
#define block_size_x 352  
#endif /* GPU TARGETS */  
  
#endif /* kernel_tuner */
```

Compile-time kernel selection example

```
from kernel_tuner.integration import store_results, create_device_targets

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

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



vector_add.h

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

#ifdef TARGET_A100_PCIE_40GB
#define block_size_x 672

#elif TARGET_RTX_A6000
#define block_size_x 160

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

#endif /* kernel_tuner */
```

Compile-time kernel selection example

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

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

This **block_size_x** value
showed best performance
on the A100



vector_add.h

```
/* header file generated by Kernel Tuner, do not modify by hand */  
#pragma once  
#ifndef kernel_tuner /* only use these when not tuning */  
  
#ifdef TARGET_A100_PCIE_40GB  
#define block_size_x 672  
  
#elif TARGET_RTX_A6000  
#define block_size_x 160  
  
#else /* default configuration */  
#define block_size_x 352  
#endif /* GPU TARGETS */  
  
#endif /* kernel_tuner */
```

Compile-time kernel selection example

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

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

This **block_size_x** value
showed best performance
on the A6000



vector_add.h

```
/* header file generated by Kernel Tuner, do not modify by hand */  
#pragma once  
#ifndef kernel_tuner /* only use these when not tuning */  
  
#ifdef TARGET_A100_PCIE_40GB  
#define block_size_x 672  
  
#elif TARGET_RTX_A6000  
#define block_size_x 160  
  
#else /* default configuration */  
#define block_size_x 352  
#endif /* GPU TARGETS */  
  
#endif /* kernel_tuner */
```

Compile-time kernel selection example

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

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

vector_add.h

```
/* header file generated by Kernel Tuner, do not modify by hand */  
#pragma once  
#ifndef kernel_tuner /* only use these when not tuning */  
  
#ifdef TARGET_A100_PCIE_40GB  
#define block_size_x 672  
  
#elif TARGET_RTX_A6000  
#define block_size_x 160  
  
#else /* default configuration */  
#define block_size_x 352  
#endif /* GPU TARGETS */  
  
#endif /* kernel_tuner */
```

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



Compile-time kernel selection example

In `vector_add.cu`:


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



In Makefile:

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

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



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

Run-time kernel selection

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



Run-time kernel selection in Python

In Python:

```
from kernel_tuner.integration import store_results
```

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



In the Python host application:

```
from kernel_tuner import kernelbuilder
```

```
# create a kernel using the stored results
```

```
vector_add = kernelbuilder.PythonKernel(kernel_name, kernel_string, n, args,  
                                         results_file="vector_add_results.json")
```

```
# call the kernel
```

```
vector_add(c, a, b, n)
```

Kernel Launcher

https://github.com/KernelTuner/kernel_launcher

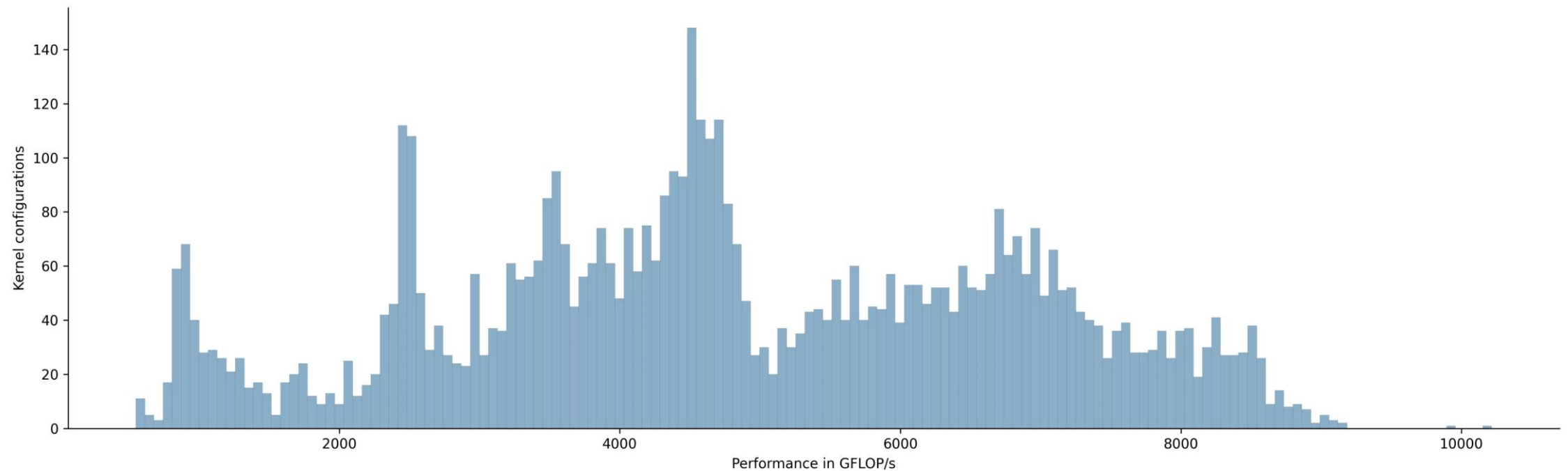
- C++ magic to integrate auto-tuned kernels into C++ applications
- Specify tunable parameters and host code in a single code base
- Automatically:
 - Extract real input/output data for tuning during application execution
 - Use Kernel Tuner to tune kernels and store results in wisdom files
 - At run-time, select and compile the best kernel configuration depending on the hardware and problem size

Optimization strategies



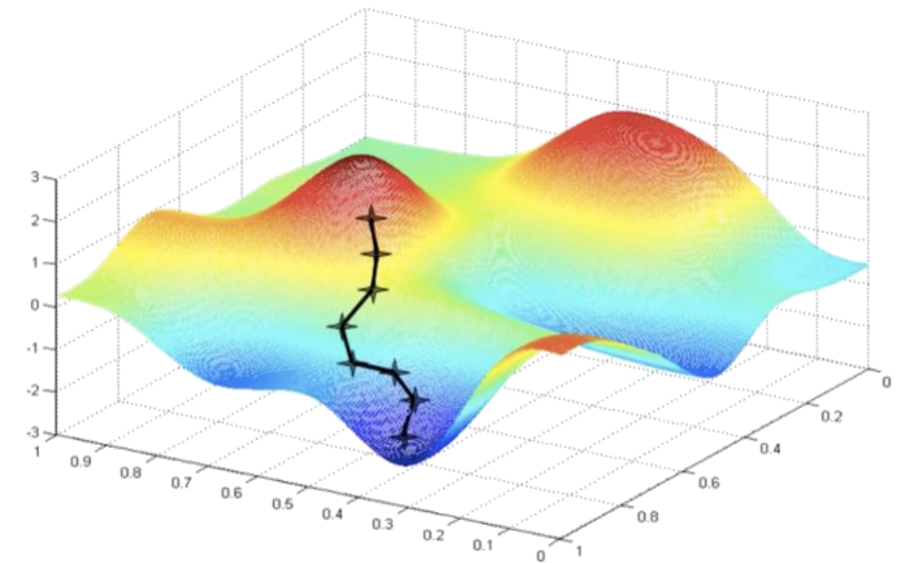
Large search space of kernel configurations

Auto-tuning a Convolution kernel on Nvidia A100

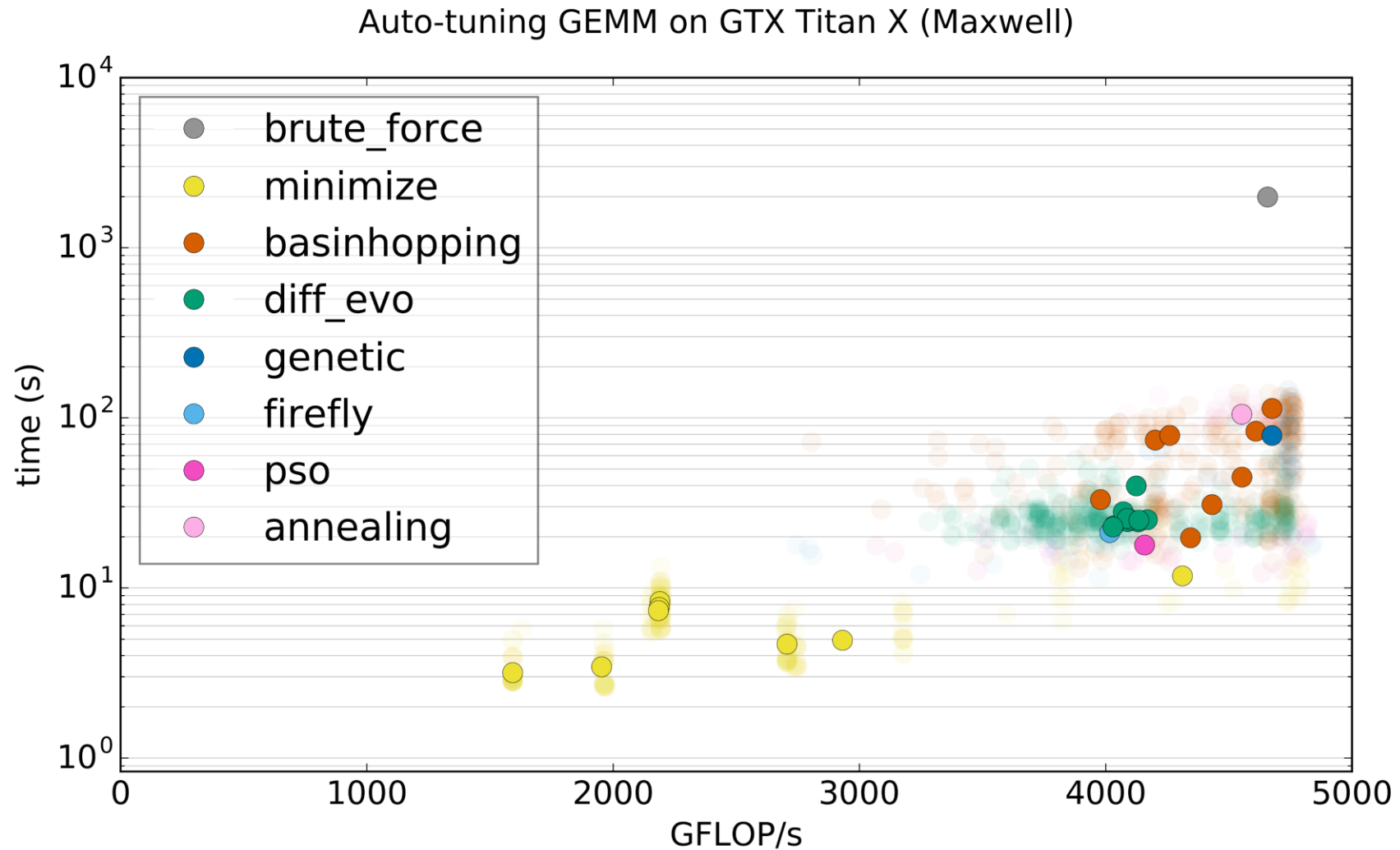


Optimization strategies in Kernel Tuner

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



Speeding up auto-tuning



Your mileage may vary

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



How to use a search strategy

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



Observers



Observers introduction

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



Creating your own observers

Simply extend the BenchmarkObserver base class to create your own Observer

Only **get_results()** is mandatory to implement

More information and documentation:

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

```
class kernel_tuner.observers.BenchmarkObserver
```

Base class for Benchmark Observers

after_finish()

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

after_start()

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

before_start()

before start is called every iteration before the kernel starts

during()

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

abstract get_results()

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

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

register_device(dev)

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



NVMLObserver

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



NVMLObserver example

...

```
tune_params["nvm1_pwr_limit"] = [250, 225, 200, 175]
```

```
nvmlobserver = NVMLObserver(["nvm1_energy", "temperature"])
```

```
metrics = OrderedDict()
```

```
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)
```



Custom tuning objectives

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

```
metrics["GFLOPS/W"] = lambda p: (size/1e9) / p["nvm1_energy"]
```

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



Fourth hands-on session



Advanced hands-on

- The fourth hands-on notebook is:
 - https://github.com/KernelTuner/kernel_tuner_tutorial/blob/master/hands-on/cuda/03_Kernel_Tuner_Advanced.ipynb
- The goal of this hands-on is to experiment with **search optimization strategies** and **custom observers**
 - You will find the notebook in your JupyterHub environment
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors



Optional hands-on

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



Closing remarks



Kernel Tuner – developed open source

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



Feature Roadmap

- Remote/parallel tuning
- Mixed-precision and accuracy tuning
- Data set sensitive tuning
- Multi-objective optimization
- Tuning compiler flags



Contributions are welcome!

- Contributions can come in many forms: tweets, blog posts, issues, pull requests
- Before making larger changes, please create an issue to discuss
- For the full contribution guide, please see:
 - https://KernelTuner.github.io/kernel_tuner/stable/contributing.html



Kernel Tuner Developers

- Alessio Sclocco (eScience Center)
- Stijn Heldens (eScience center)
- Floris-Jan Willemsen (eScience Center)
- Richard Schoonhoven (CWI)
- Willem Jan Palenstijn (CWI)
- Bram Veenboer (Astron)
- Ben van Werkhoven (eScience Center)
- And many more contributors, see [GitHub](#) for the full list



Publications

- Optimization Techniques for GPU Programming
Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri Bal
ACM Computing surveys (accepted for publication)
- 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
- Benchmarking optimization algorithms for auto-tuning GPU kernels
Richard Schoonhoven, Ben van Werkhoven, K. Joost Batenburg
IEEE Transactions on Evolutionary Computation (TEVC) 2022
- Kernel Tuner: A search-optimizing GPU code auto-tuner,
B. van Werkhoven
Future Generation Computer Systems, 2019



Funding

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

ESiWACE2 has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 823988.



The ConFu project is funded by the Netherlands eScience Center (file number 00020223-A).

Netherlands eScience Center

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



Thanks!

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

Ben van Werkhoven

b.vanwerkhoven@esciencecenter.nl

