Kernel Tuner Tutorial – Getting Started

netherlands eScience center

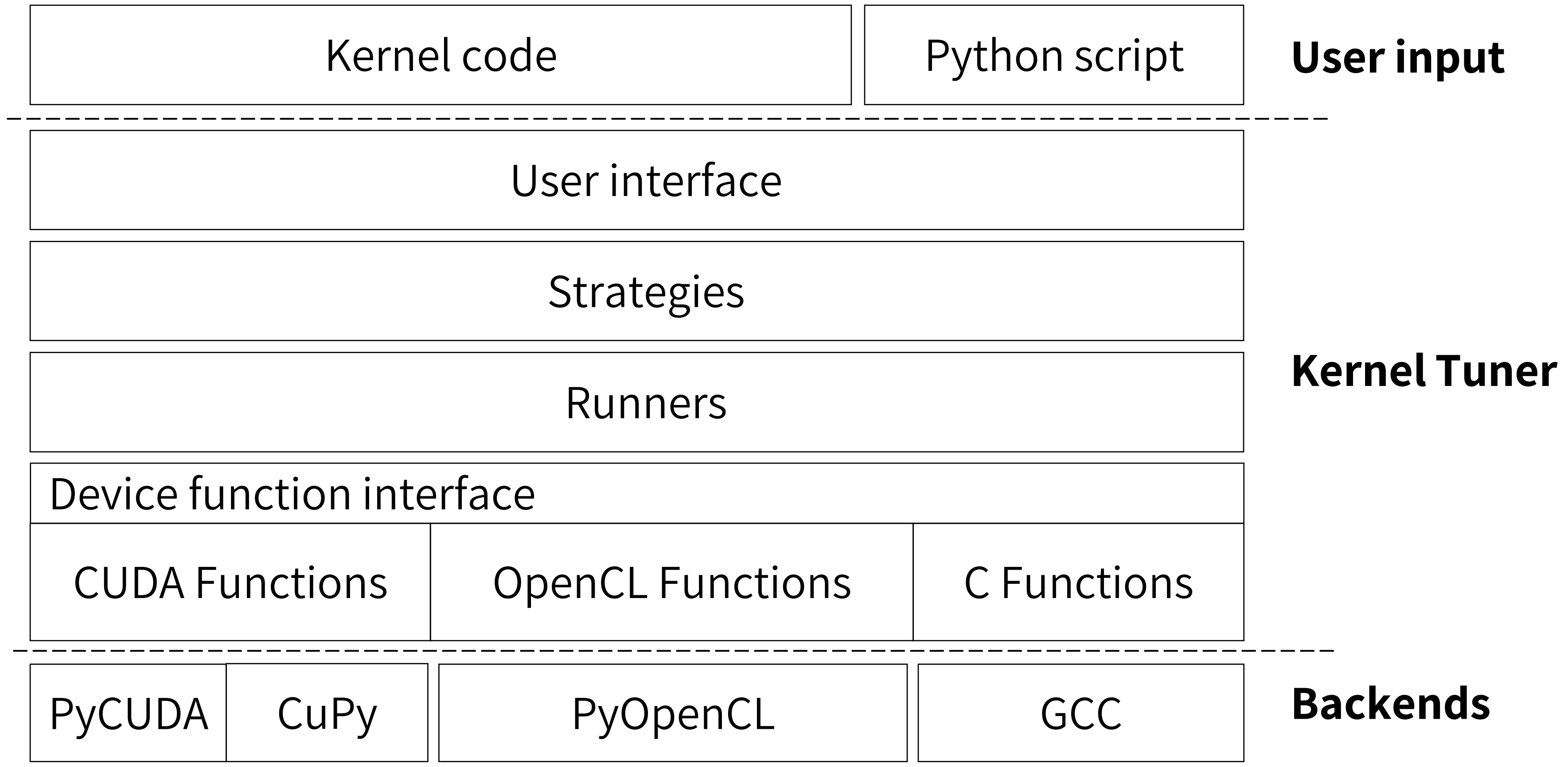
Getting started outline

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

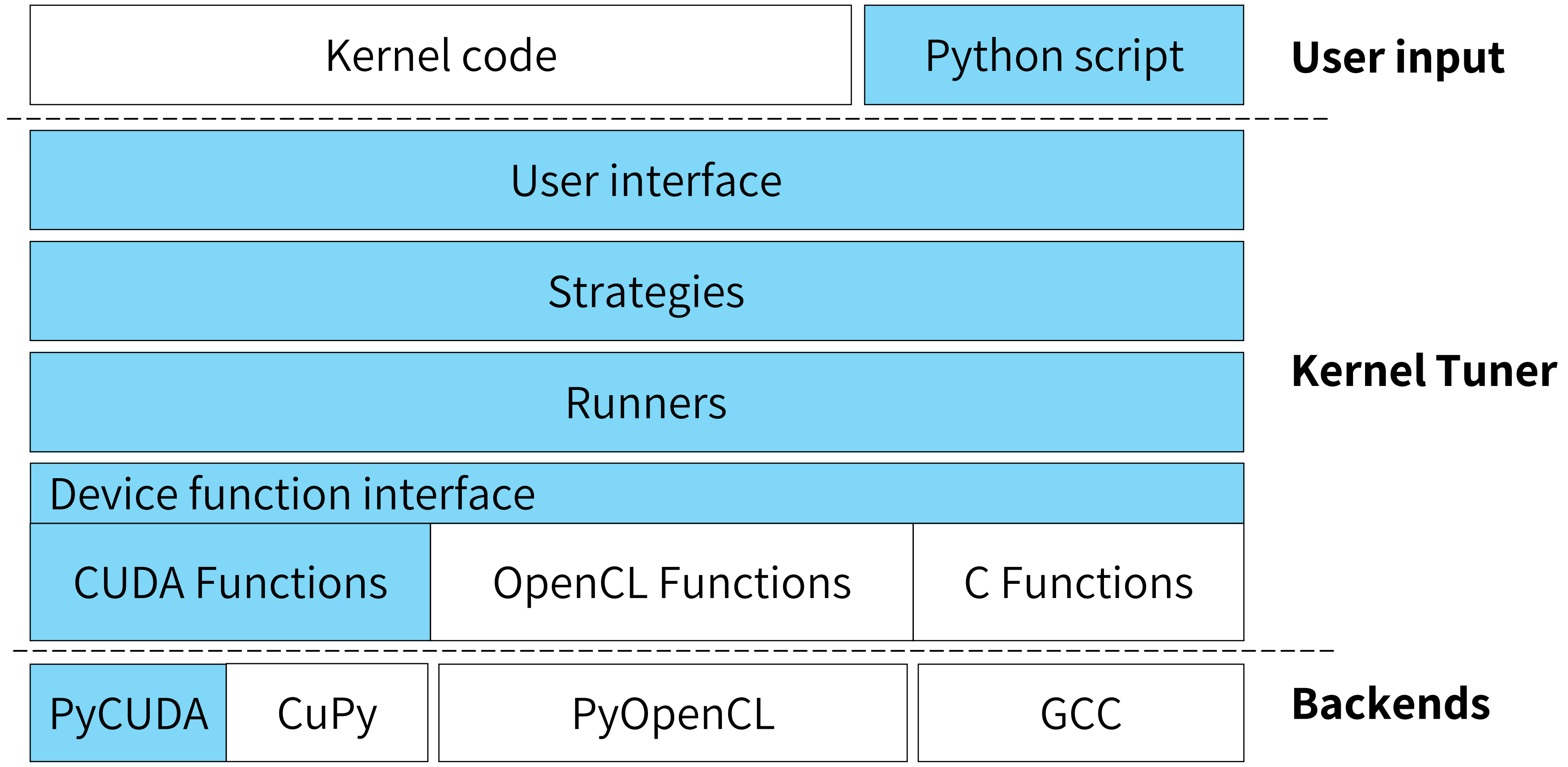
netherlands eScience center

Kernel Tuner code integration

Kernel Tuner architecture

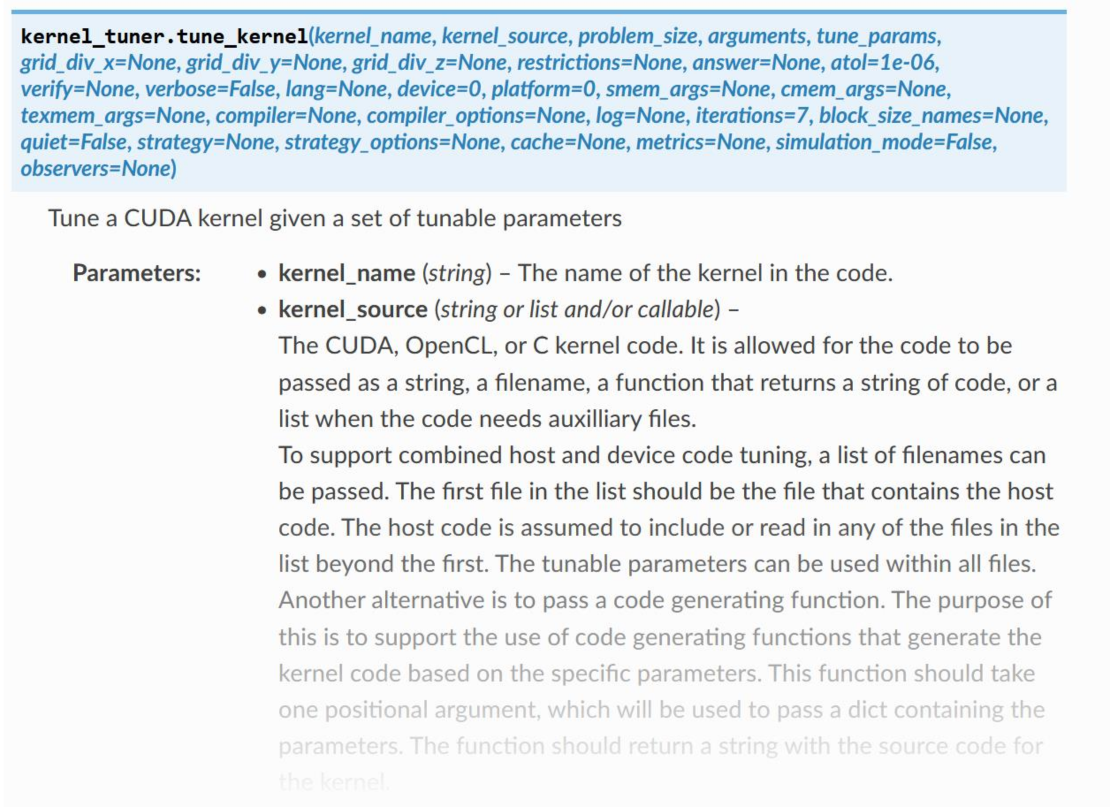


Kernel Tuner architecture



Kernel Tuner compiles and benchmarks many kernel configurations

- So 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 auxilliary files.
  To support combined host and device code tuning, a list of filenames can be passed. The first file in the list should be the file that contains the host code. The host code is assumed to include or read in any of the files in the list beyond the first. The tunable parameters can be used within all files.
  Another alternative is to pass a code generating function. The purpose of this is to support the use of code generating functions that generate the kernel code based on the specific parameters. This function should take one positional argument, which will be used to pass a dict containing the parameters. The function should return a string with the source code for the kernel.

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, if you want to select a different backend use the lang option, e.g. `lang="CuPy"` to select the CuPy backend

PyCUDA backend

Uses NVCC to compile CUDA kernels and then does a lookup based on the kernel name, therefore:

- CUDA kernels must have **`extern "C"`** linkage
- CUDA kernels therefore cannot be templated

- Kernel Tuner has a workaround that detects template usage and provides some support for templated kernels by generating a wrapper kernel on-the-fly
- The wrapper kernel will have **`extern "C"`** linkage

CuPy backend

- CuPy uses NVRTC for runtime compilation
- Kernels may have slightly different performance and register usage compared to NVCC compilation
- Fully supports C++ and templates for kernels

- NVRTC does not allow host code in any of the source files
- General recommendation is to isolate kernels into separate source files and include those files when needed in the host application

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 use **not use arrays of structs** whenever you can.

Custom type workarounds

- Call a C function or a different CUDA kernel that initializes data, e.g. using run_kernel to construct data before passing it to tune_kernel
- Dump data to binary and read in the data in Python
- Write a wrapper kernel that converts primitive types to custom types and in turn calls the kernel
- Rewrite kernels to use simple arrays of primitive types

Kernel compilation

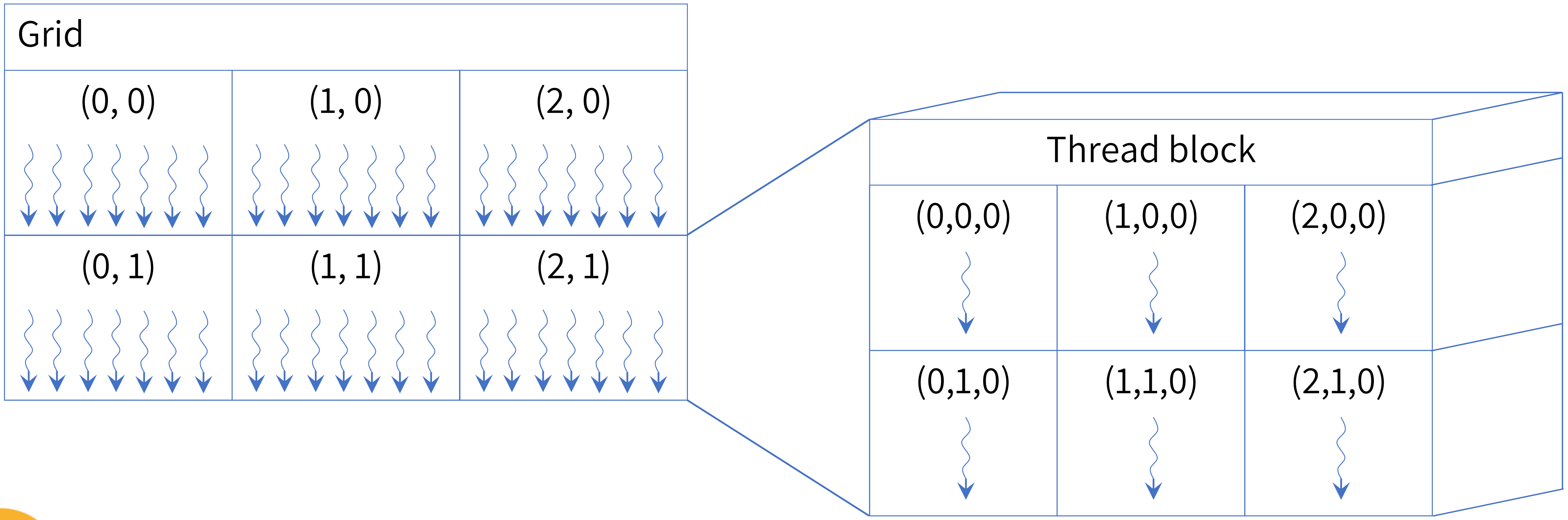
- Kernel Tuner will compile the same kernel over and over again with different parameters inserted
- If your code includes many headers with all kinds of template expansions the compilation time may become prohibitive
- Recommendation: isolate device code from the rest of your application
    - Easy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

Summary

- For tuning CUDA kernels, Kernel Tuner uses either PyCUDA or CuPy
- PyCUDA forces kernels to have **`extern "C"`** linkage
    - Limited templates support by on-the-fly wrapper generation by Kernel Tuner
- CuPy fully supports C++ and templates, but no host code is 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

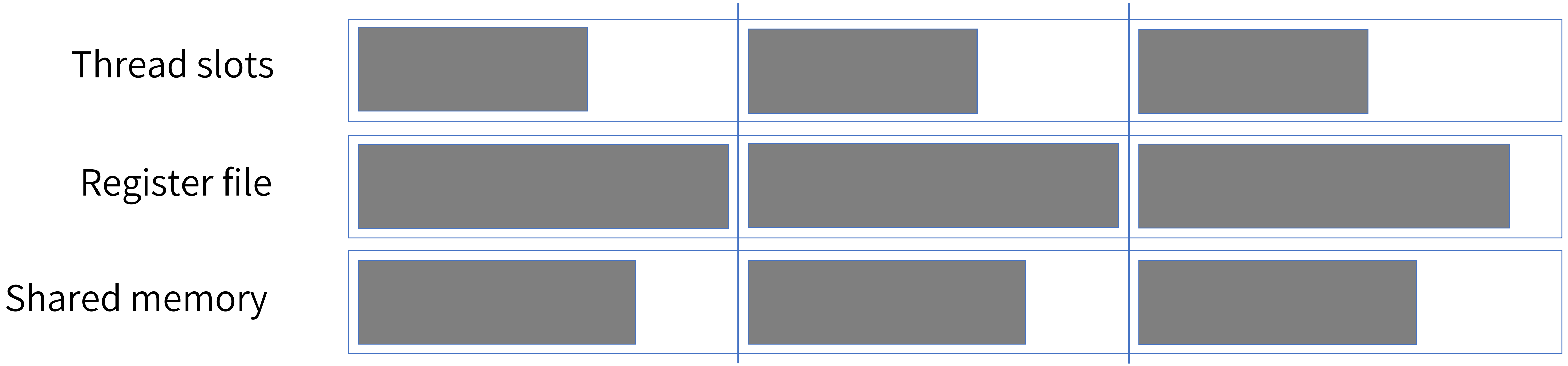


Choosing thread block dimensions

- 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 to tune_kernel

- 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

Specify 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)

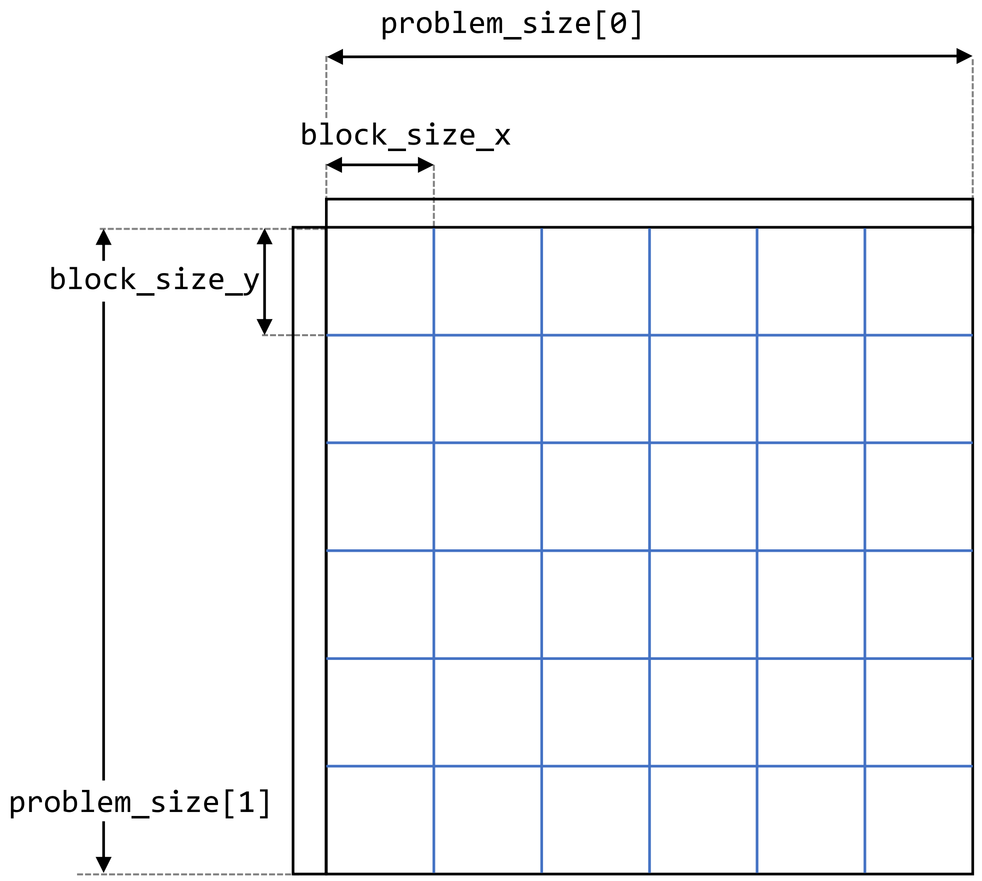
Vector add example

```
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`**

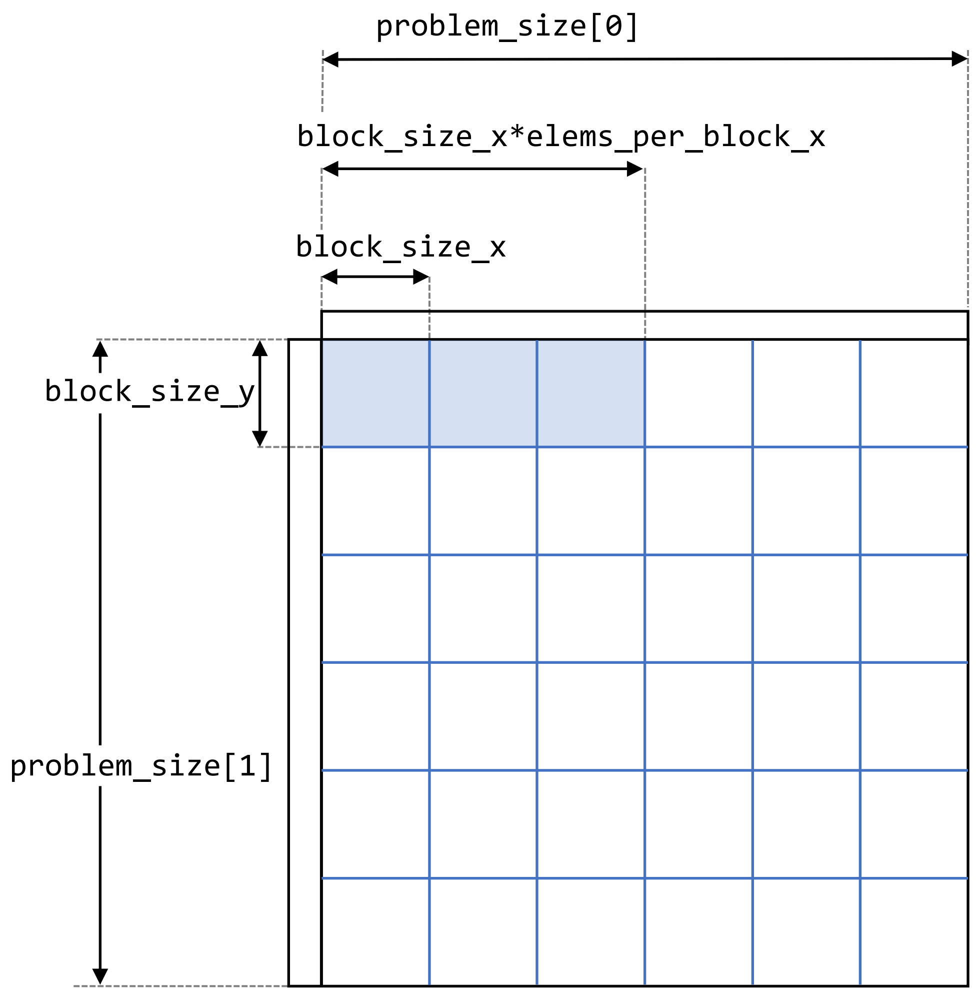
Specifying grid dimensions

- You specify the `problem_size`
- `problem_size` describes the dimensions across which threads are created
- By default, the grid dimensions are computed as:
  - `grid_size_x = ceil(problem_size_x / block_size_x)`



Grid divisor lists

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



problem_size

- The problem size is usually a single integer or a tuple of integers
- But you may also use strings to use a tunable parameter
- `problem_size` 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/benvanwerkhoven/kernel_tuner_tutorial/blob/master/hands-on/cuda/01_Kernel_Tuner_Getting_Stared.ipynb
- The goal of this hands-on is to experiment with tunable grid dimensions and user defined metrics
  - Copy the notebook to your Google Colab and work there
- Please follow the instructions in the Notebook
- Feel free to ask questions to instructors and mentors

Optional hands-on

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