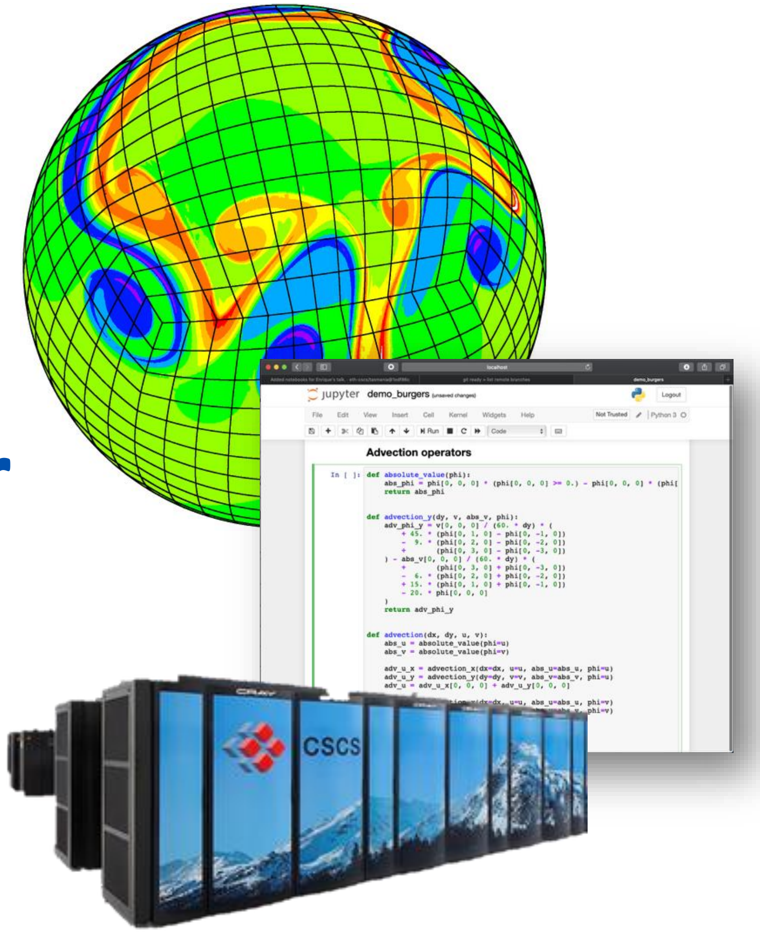
# High Performance Computing for Weather and Climate (HPC4WC)

Content: High-Level Programming

Lecturer: Christoph Müller, Oliver Fuhrer

Block course 701-1270-00L

Summer 2025



# Supercomputer Architecture

(Numbers are for Alps and vary from system to system)

## Day 1

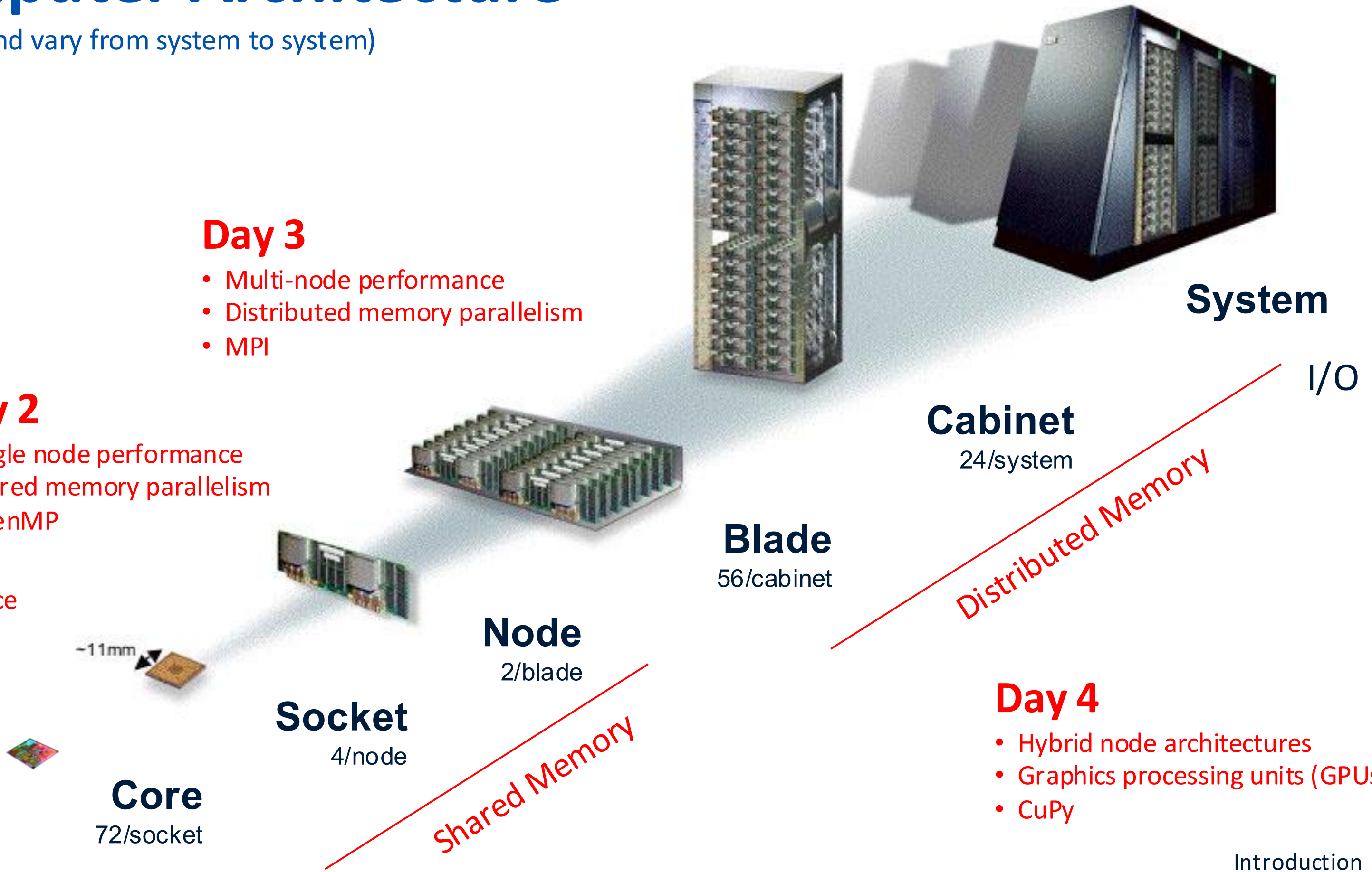
- Single core performance
- Caches

## Day 2

- Single node performance
- Shared memory parallelism
- OpenMP

## Day 3

- Multi-node performance
- Distributed memory parallelism
- MPI




## Day 4

- Hybrid node architectures
- Graphics processing units (GPUs)
- CuPy

# Future of HPC in Weather and Climate?

**Yesterday**

x86 CPU

MPI

Fortran

OpenMP

**Today**

C++

GPU

Python

Domain-specific languages

ML

**Tomorrow**

ML

Specialized hardware?

ASIC?

GT4Py?

# Learning Goals

- Understand what a domain-specific language (DSL) is.
- Understand how a DSL helps in writing hardware-agnostic and maintainable code without sacrificing performance.
- Be able to apply a DSL to a stencil program from a weather and climate model.

# Typical Workflow

| | | |
|---|---|---|
| solver.py | `import numpy as np` | Fast prototyping in Python (or MATLAB) |
| solver.F90 | `#include <vector>` | Naïve implementation in a compiled language (e.g. F90, C++) |
| solver_omp.F90 | `#pragma omp parallel` | Multi-threaded version using OpenMP |
| solver_omp_mpi.F90 | `MPI_Init(&argc, &argv);` | Going multi-node with MPI (possibly blended with OpenMP) |
| solver.cu | `cudaMalloc(&d_x, …));` | CUDA version for impressive single-node performance |
| solver_mpi.cu | `MPI_Recv(d_x, …);` | CUDA-aware MPI: getting the best out of hybrid systems |

# Possible Scenarios

**What if…**

…we want to introduce a modification at the algorithmic/numerical level?

…our application has a broad user community and it must run efficiently on a variety of platforms?

…our code consists of thousands (if not millions) lines of code?

**The explosion of hardware architectures made this development model obsolete!**

# A Real-Case Example: ICON

- Global and regional weather and climate model developed by the **ICON Partnership** and many other contributers
- Run operationally by 8 national weather services, used for global and regional production climate simulations and used by several academic institutions as a research tool.
- Four main target architectures: x86 CPUs, NVIDIA and AMD GPUs and NEC vector CPUs
- Around 1.6 M lines of F90 code and 0.45 M lines of C/C++ code.
- Cost of porting the full code base to GPU: approx. 30 programmer-years!

# Separation of Concerns

| Domain expert | Performance expert |
| --- | --- |
| Answer scientific research questions | Write optimized code for target platform |
| Declarative programming style:<br>Focus on **what** you want to do | Imperative programming style:<br>Focus on **how** to do it |
| Common data access interface:<br>e.g. data[i, j, k] | Storage and memory allocation:<br>e.g. C-layout vs F-layout |
| Computation kernels:<br>Calculations for a single grid point | Control structure (e.g. for loops):<br>Optimized data traversal |
| Individual operators (“grains”) | Final computation:<br>Detect and exploit parallelism b/w grains |

# Overarching Goals (The 3 P's)

- **Productivity**
  Easy to implement.
  Easy to **read**.
  Easy to **maintain**.

- **Performance**
  Is **fast**.

- **Portability**
  Single **hardware-agnostic** application code.
  Runs efficiently on **different hardware** targets.

# Domain Specific Languages (DSLs)

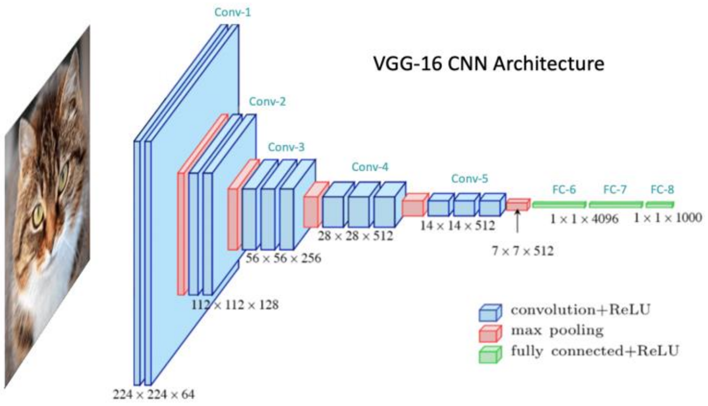
- Programming language tailored for a specific class of problems.
- Higher level of abstraction w.r.t. a general purpose language.
- Intended to be used by domain experts, who may not be fluent in programming.
- Abstractions and notations much aligned to concepts and rules from the domain.

# Domain Specific Languages (DSLs)

- Programming language tailored for a specific class of problems.
- Higher level of abstraction w.r.t. a general purpose language.
- Intended to be used by domain experts, who may not be fluent in programming.
- Abstractions and notations much aligned to concepts and rules from the domain.
- Some examples:
    - Typesetting: LaTeX
    - Machine Learning: PyTorch, JAX
    - Scientific Computing: Kokkos, FEniCS
    - Fluid Dynamics: OpenFOAM
    - Image Processing: Halide, Taichi
    - Stencils: Devito, GT4Py, Exo

# Example: VGG-16 in PyTorch

```
# Block 1
nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, stride=1, padding='same'),
nn.ReLU(),
nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding='same'),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2),

# Block 2
nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding='same'),
nn.ReLU(),
nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding='same'),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2),

# Block 3
nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding='same'),
nn.ReLU(),
nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding='same'),
nn.ReLU(),
nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1, padding='same'),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2),
```



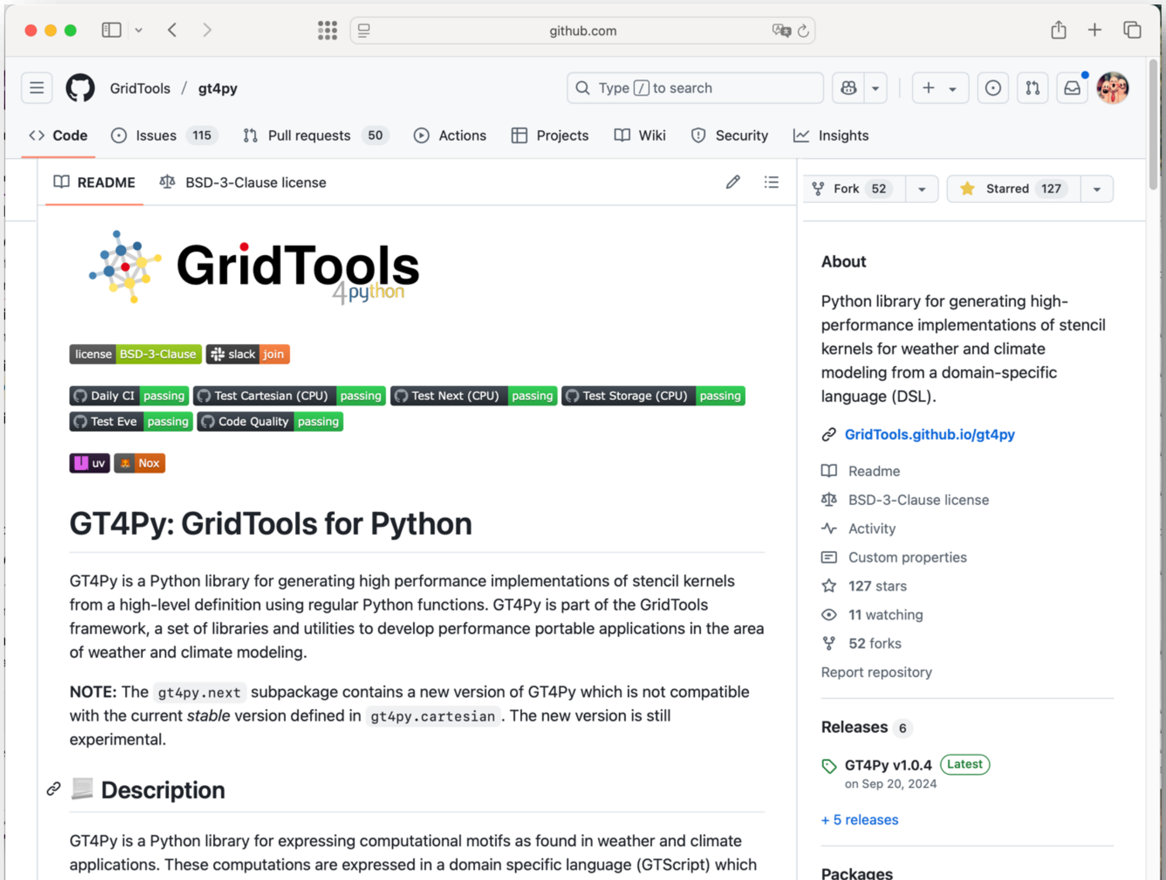

Imagine this was written as naive C++ code

- Loops, cache blocking, MPI, …

# Example: Single Conv2d layer in CUDA

```
#define TILE_WIDTH 16
#define K 3
#define RADIUS (K / 2)

__global__
void conv2d_shared_optimized(const float* __restrict__ input,
                             const float* __restrict__ kernel,
                             float* __restrict__ output,
                             int H, int W)
{
    // Shared memory tile: (TILE_WIDTH + 2 * RADIUS)^2 to hold halo
    __shared__ float tile[TILE_WIDTH + 2 * RADIUS][TILE_WIDTH + 2 * RADIUS];

    // Thread and global coordinates
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int row_o = blockIdx.y * TILE_WIDTH + ty;
    int col_o = blockIdx.x * TILE_WIDTH + tx;

    int row_i = row_o - RADIUS;
    int col_i = col_o - RADIUS;

    // Shared memory index
    if (row_i >= 0 && row_i < H && col_i >= 0 && col_i < W) {
        tile[ty][tx] = input[row_i * W + col_i];
    } else {
        tile[ty][tx] = 0.0f; // zero-padding
    }

    // Ensure all threads have loaded their tile
    __syncthreads();

    // Compute only within output tile (excluding halo region)
    if (ty >= RADIUS && ty < TILE_WIDTH + RADIUS &&
        tx >= RADIUS && tx < TILE_WIDTH + RADIUS &&
        row_o < H && col_o < W) {

        float sum = 0.0f;

        #pragma unroll
        for (int i = 0; i < K; ++i) {
            #pragma unroll
            for (int j = 0; j < K; ++j) {
                sum += kernel[i * K + j] * tile[ty - RADIUS + i][tx - RADIUS + j];
            }
        }

        output[row_o * W + col_o] = sum;
    }
}
```

# GT4Py


github.com

GridTools / gt4py

Type / to search

Code Issues 115 Pull requests 50 Actions Projects Wiki Security Insights

README BSD-3-Clause license

Fork 52 Starred 127

GridTools 4python

license BSD-3-Clause slack join

Daily CI passing Test Cartesian (CPU) passing Test Next (CPU) passing Test Storage (CPU) passing

Test Eve passing Code Quality passing

uv Nox

## GT4Py: GridTools for Python

GT4Py is a Python library for generating high performance implementations of stencil kernels from a high-level definition using regular Python functions. GT4Py is part of the GridTools framework, a set of libraries and utilities to develop performance portable applications in the area of weather and climate modeling.

**NOTE:** The `gt4py.next` subpackage contains a new version of GT4Py which is not compatible with the current *stable* version defined in `gt4py.cartesian`. The new version is still experimental.

## Description

GT4Py is a Python library for expressing computational motifs as found in weather and climate applications. These computations are expressed in a domain specific language (GTScript) which

**About**

Python library for generating high-performance implementations of stencil kernels for weather and climate modeling from a domain-specific language (DSL).

GridTools.github.io/gt4py

Readme

BSD-3-Clause license

Activity

Custom properties

127 stars

11 watching

52 forks

Report repository

**Releases** 6

GT4Py v1.0.4 Latest

on Sep 20, 2024

+ 5 releases

**Packages**

# GT4Py

- High-performance implementation of a stencil kernel from a high-level definition.
- GT4Py is a domain specific **library** which exposes a domain specific **language** to express the stencil logic.
- GT4Py is embedded in Python (**eDSL**).
  - Legal Python syntax semantics, can be executed directly in Python.
- GT4Py = **G**rid**T**ools **For P**ython
  - Harnessing the C++ GridTools ecosystem to generate native implementations of the stencils.
- Emphasis on tight integration with scientific Python stack.

# What Does The GT4Py DSL Need?

```
import numpy as np

def laplacian_np(in_field):
    out_field = np.zeros_like(in_field)
    nx, ny, nz = in_field.shape
    for i in range(1, nx - 1):
        for j in range(1, ny - 1):
            for k in range(0, nz):
                out_field[i, j, k] = (
                    - 4 * in_field[i, j, k]
                    + in_field[i - 1, j, k]
                    + in_field[i + 1, j, k]
                    + in_field[i, j - 1, k]
                    + in_field[i, j + 1, k] )
    return out_field
```

Input, output, and possibly temporary 3D fields

Nested loops iterating along both horizontal and vertical directions

Math operations

Indices and offsets

# Laplacian in GT4Py

```python
import gt4py.next as gtx
import numpy as np

I = gtx.Dimension("I")
J = gtx.Dimension("J")
K = gtx.Dimension("K", kind=gtx.DimensionKind.VERTICAL)

IJKField = gtx.Field[gtx.Dims[I, J, K], gtx.float64]

@field_operator
def _grad_norm(in_field: IJKField) -> IJKField:
    lap_field = (
        -4.0 * in_field
        + in_field(I - 1)
        + in_field(I + 1)
        + in_field(J - 1)
        + in_field(J + 1)
    )

    return lap_field
```

# Laplacian in GT4Py

```
import gt4py.next as gtx
import numpy as np

I = gtx.Dimension("I")
J = gtx.Dimension("J")
K = gtx.Dimension("K", kind=gtx.DimensionKind.VERTICAL)

IJKField = gtx.Field[gtx.Dims[I, J, K], gtx.float64]

@field_operator
def _grad_norm(in_field: IJKField) -> IJKField:
    lap_field = (
        -4.0 * in_field
        + in_field(I - 1)
        + in_field(I + 1)
        + in_field(J - 1)
        + in_field(J + 1)
    )

    return lap_field
```

# Laplacian in GT4Py

```
import gt4py.next as gtx
import numpy as np

I = gtx.Dimension("I")
J = gtx.Dimension("J")
K = gtx.Dimension("K", kind=gtx.DimensionKind.VERTICAL)

IJKField = gtx.Field[gtx.Dims[I, J, K], gtx.float64]

@field_operator
def _grad_norm(in_field: IJKField) -> IJKField:
    lap_field = (
        -4.0 * in_field
        + in_field(I - 1)
        + in_field(I + 1)
        + in_field(J - 1)
        + in_field(J + 1)
    )

    return lap_field
```

# Laplacian in GT4Py

```
import gt4py.next as gtx
import numpy as np

I = gtx.Dimension("I")
J = gtx.Dimension("J")
K = gtx.Dimension("K", kind=gtx.DimensionKind.VERTICAL)

IJKField = gtx.Field[gtx.Dims[I, J, K], gtx.float64]

@field_operator
def _grad_norm(in_field: IJKField) -> IJKField:
    lap_field = (
        -4.0 * in_field
        + in_field(I - 1)
        + in_field(I + 1)
        + in_field(J - 1)
        + in_field(J + 1)
    )

    return lap_field
```

# Laplacian in GT4Py

```
import gt4py.next as gtx
import numpy as np

I = gtx.Dimension("I")
J = gtx.Dimension("J")
K = gtx.Dimension("K", kind=gtx.DimensionKind.VERTICAL)

IJKField = gtx.Field[gtx.Dims[I, J, K], gtx.float64]

@field_operator
def _grad_norm(in_field: IJKField) -> IJKField:
    print(“Hello”)  # ERROR
    lap_field = (
        -4.0 * in_field
        + in_field(I - 1)
        + in_field(I + 1)
        + in_field(J - 1)
        + in_field(J + 1)
    )

    return lap_field
```

# Laplacian in GT4Py

```python
import gt4py.next as gtx
import numpy as np

I = gtx.Dimension("I")
J = gtx.Dimension("J")
K = gtx.Dimension("K", kind=gtx.DimensionKind.VERTICAL)

IJKField = gtx.Field[gtx.Dims[I, J, K], gtx.float64]

@field_operator
def _grad_norm(in_field: IJKField) -> IJKField:
    lap_field = (
        -4.0 * in_field
        + in_field(I - 1)
        + in_field(I + 1)
        + in_field(J - 1)
        + in_field(J + 1)
    )

    return lap_field
```

# Laplacian in GT4Py

```
import gt4py.next as gtx
import numpy as np

I = gtx.Dimension("I")
J = gtx.Dimension("J")
K = gtx.Dimension("K", kind=gtx.DimensionKind.VERTICAL)

IJKField = gtx.Field[gtx.Dims[I, J, K], gtx.float64]

@field_operator
def _grad_norm(in_field: IJKField) -> IJKField:
    lap_field = (
        -4.0 * in_field
        + in_field(I - 1)
        + in_field(I + 1)
        + in_field(J - 1)
        + in_field(J + 1)
    )

    return lap_field
```

# Laplacian in GT4Py

```
import gt4py.next as gtx
import numpy as np

I = gtx.Dimension("I")
J = gtx.Dimension("J")
K = gtx.Dimension("K", kind=gtx.DimensionKind.VERTICAL)

IJKField = gtx.Field[gtx.Dims[I, J, K], gtx.float64]

@field_operator
def _grad_norm(in_field: IJKField) -> IJKField:
    lap_field = (
        -4.0 * in_field
        + in_field(I - 1)
        + in_field(I + 1)
        + in_field(J - 1)
        + in_field(J + 1)
    )

    return lap_field
```

# Laplacian in GT4Py

```
import gt4py.next as gtx
import numpy as np

I = gtx.Dimension("I")
J = gtx.Dimension("J")
K = gtx.Dimension("K", kind=gtx.DimensionKind.VERTICAL)

IJKField = gtx.Field[gtx.Dims[I, J, K], gtx.float64]

@field_operator
def _grad_norm(in_field: IJKField) -> IJKField:
    lap_field = (
        -4.0 * in_field
        + in_field(I - 1)
        + in_field(I + 1)
        + in_field(J - 1)
        + in_field(J + 1)
    )

    return lap_field
```

- No loops
- No concrete domains (this stencil can be applied to any compute domain)
- No optimization code such as OpenMP, MPI, …

# Backends

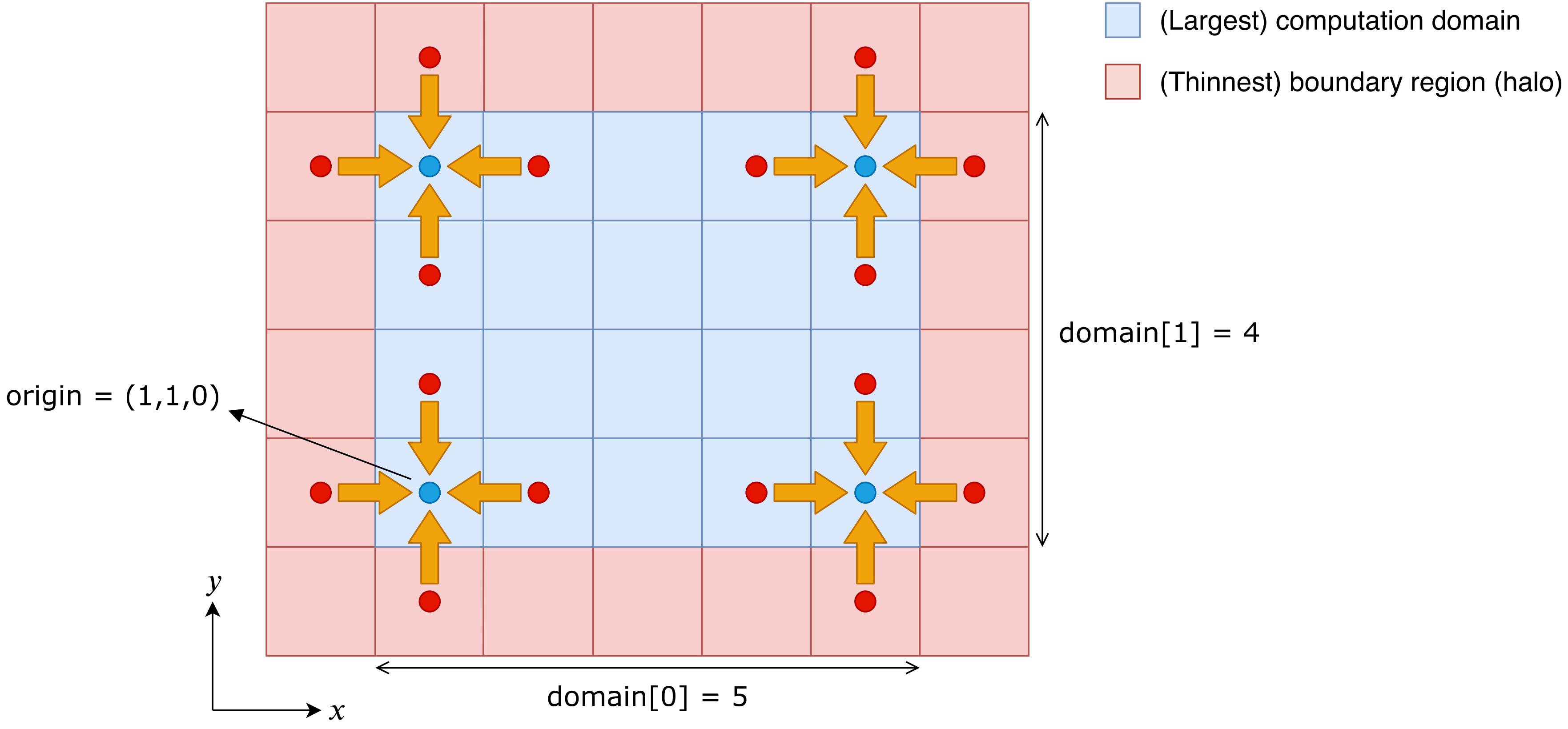
- A stencil needs to be instantiated for a given **backend**:

```
backend = gtx:gtfn_cpu
laplacian = laplacian_defs.with_backend(backend)
```

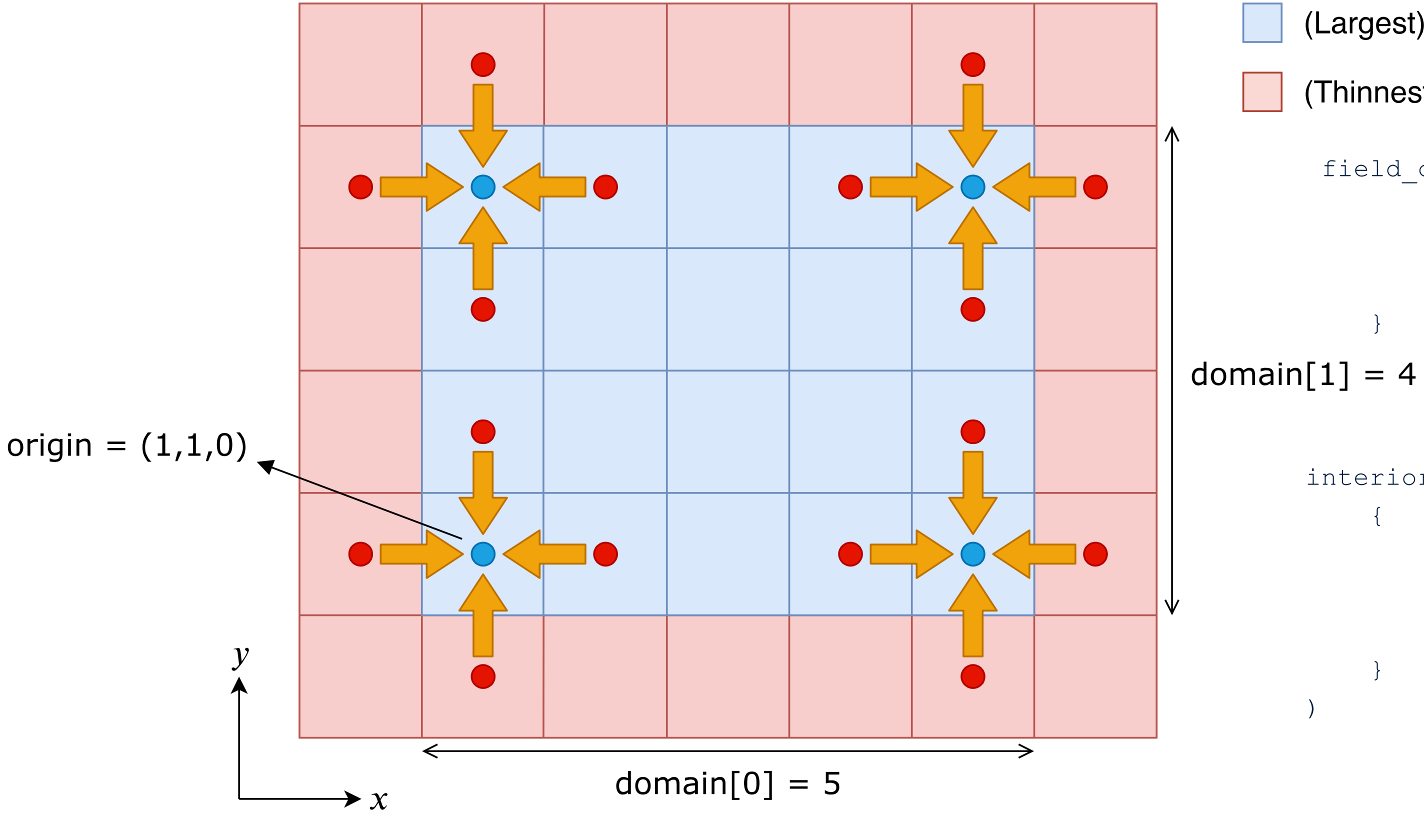
- Backends target different purposes, needs, and computer architectures:
  - Python: None (embedded execution for prototyping, debugging);
  - C++: gtx:gtfn_cpu (x86), gtx:gtfn_gpu (NVIDIA GPU).

- For non-Python backends, compilation consists of three steps:
  1) Generate optimized code for the target architecture (cached in .gt4py_cache).
  2) Compile the automatically generated code.
  3) Build Python bindings to that code.

# Region of application



# Region of application



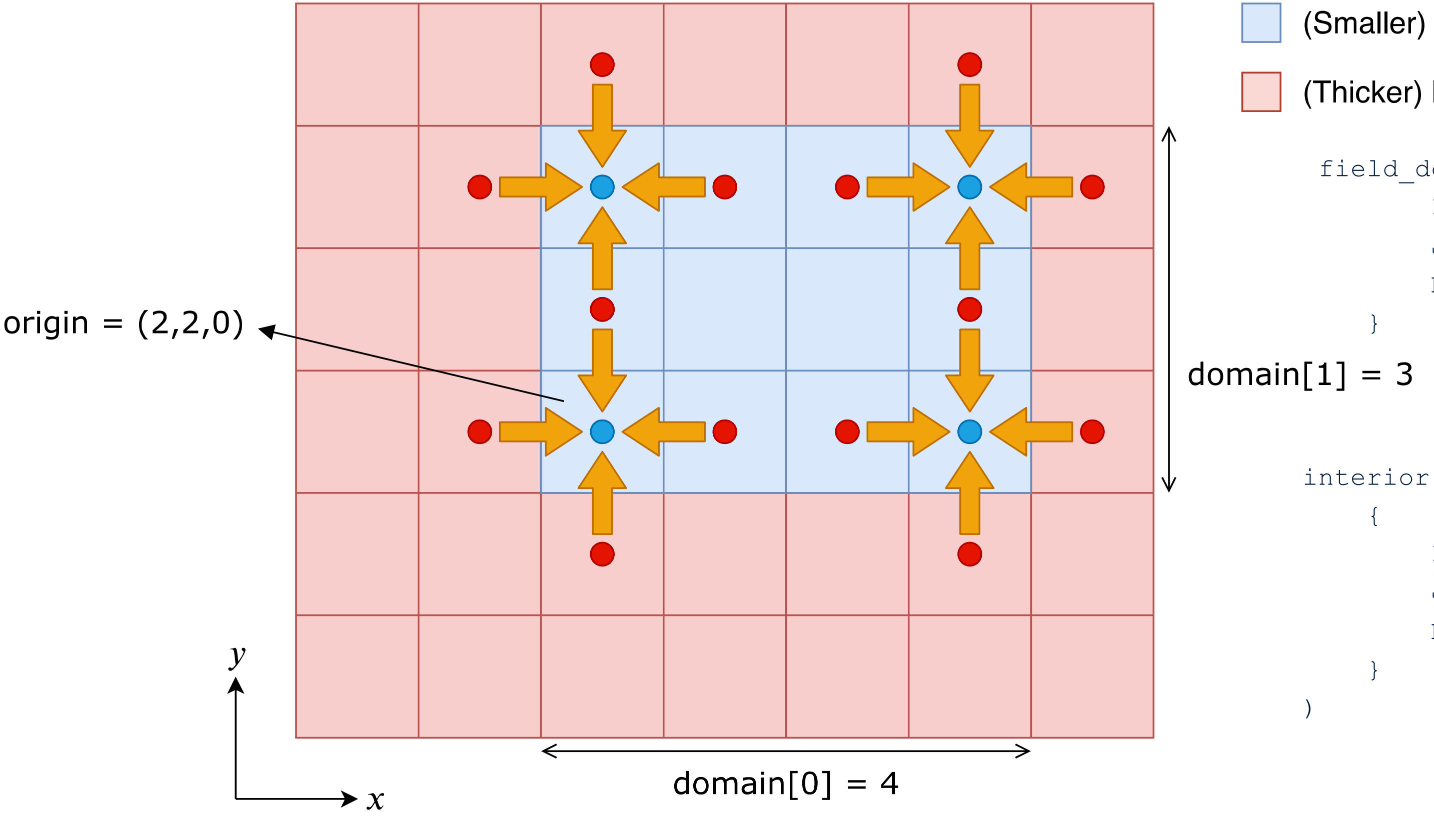

(Largest) computation domain

(Thinnest) boundary region (halo)

```
field_domain = {
        I: (0, 5 + 2),
        J: (0, 4 + 2),
        K: (0, 10),
    }
```

```
interior = gtx.domain(
    {
        I: (1, 6),
        J: (1, 5),
        K: (0, 10),
    }
)
```

# Region of application




(Smaller) computation domain

(Thicker) boundary region (halo)

```
field_domain = {
        I: (0, 5 + 2),
        J: (0, 4 + 2),
        K: (0, 10),
    }
```

```
interior = gtx.domain(
    {
        I: (2, 6),
        J: (2, 5),
        K: (0, 10),
    }
)
```

# Field

- A field operator is a callable object which can be invoked on GT4Py fields.
- Fields have optimal memory **strides**, **alignment** and **padding**.
- gtx provides functionalities to allocate fields …

```
nx, ny, nz = 128, 128, 64
num_halo = 2
field_domain = {I: (-num_halo, nx + num_halo), J: (-num_halo, ny + num_halo), K: (0, nz)}

out_field = gtx.zeros(field_domain, dtype=gtx.float64, allocator=backend)
```

… and convert NumPy arrays into valid fields:

```
in_field = gtx.as_field(np.randon.rand(nx+2*num_halo, ny+2*num_halo, nz),
domain=field_domain, dtype=f64, allocator=backend)
```

# Field

- Fields can be accessed as NumPy arrays:

```
in_field.ndarray[0, 0, 0] = 4.
print(in_field.ndarray[0, 0, 0])
# Output: 4.0
```

# Running

- Running computations is as simple as a function call:

```
laplacian(
        in_field=in_field,
        out=out_field,
        domain=interior,
)
```

Pass all arguments to field operator which are listed in the interface

# Running

- Running computations is as simple as a function call:

```
laplacian(
    in_field=in_field,
    out=out_field,
    domain=interior,
)
```

Required additional output argument

# Running

- Running computations is as simple as a function call:

```
laplacian(
        in_field=in_field,
        out=out_field,
        domain=interior,
)
```

Optional computation domain

# Running

- Running computations is as simple as a function call:

```
laplacian(
        in_field=in_field,
        out=out_field,
        domain=interior,
)
```

- `out` now contains the results of the computation.

# Weather and Climate on DSLs

- Several models (FV3, FVM and ICON) being ported to GT4Py
- Other approaches
  - COSMO (MeteoSwiss) dynamical was re-written in C++ using GridTools library.
  - E3SM (US DOE) using the Kokkos library for on-node parallelism.
  - LFric (UK MetOffice)
- Who knows what the future will bring…

# Disadvantages of a DSL

- Lack of generality: A DSL is not a complete ontology!
- Debugging on the generated code.
- Cost of developing and maintaining the DSL compiler toolchain.

# Conclusions

- High-level programming techniques hide the complexities of the underlying architecture to the end user.
- DSL allows to target multiple platforms without polluting the application code with hardware-specific boilerplate code.
- GT4Py is a Python framework to write performance portable applications in the weather and climate area. It ships with a DSL to write stencil computations.

# Lab Exercises

**01-GT4Py-motivation.ipynb**

- Compare NumPy, CuPy and GT4Py on the sum-diff and Laplacian stencil (demo).

**02-GT4Py-concepts.ipynb**

- Digest the main concepts of GT4Py.
- Get familiar with writing, compiling and running stencils.
- Get insights on the internal data-layout of the storages.

**03-GT4Py-stencil2d.ipynb**

- Step-by-step porting of stencil2d.py to GT4Py.
- Write two alternative versions of stencil2d-gt4py.py

# References

Broad introduction to DSLs:
https://www.jetbrains.com/mps/concepts/domain-specific-languages/

GT4Py repository:
https://github.com/GridTools/gt4py