Workshop on

Domain-Specific Languages for Performance-Portable Weather and Climate Models

Content: Introduction & Basic Concepts I
(stencil definition, fields & storages, compilation & execution)

Presenter: Oliver Fuhrer

>>> print(“Welcome!”)

Jeremy McGibbon

Lucas Harris (GFDL)

Johann Dahm

Oliver Elbert

Eddie Davies

Oliver Fuhrer

Rhea George

Tobias Wicky

Christopher Kung (NASA)

GFDL Operations (Garrett Power, Louis Wust, …)

GFDL Front Office (Dale “Chief Walton, …)

NOAA RDHPCS (Frank Indiviglio, Eric Schnepp, …)

MSU HPC User Support

Agenda

Time (EST)	What
10:00 am	**Session A** (Presentation) **Hands-on A**
1:30 pm	**Session B** (Discussion and presentation) **Hands-on B**
4:30 pm	**Session C** (Wrap-up and discussion) **Hands-on C**

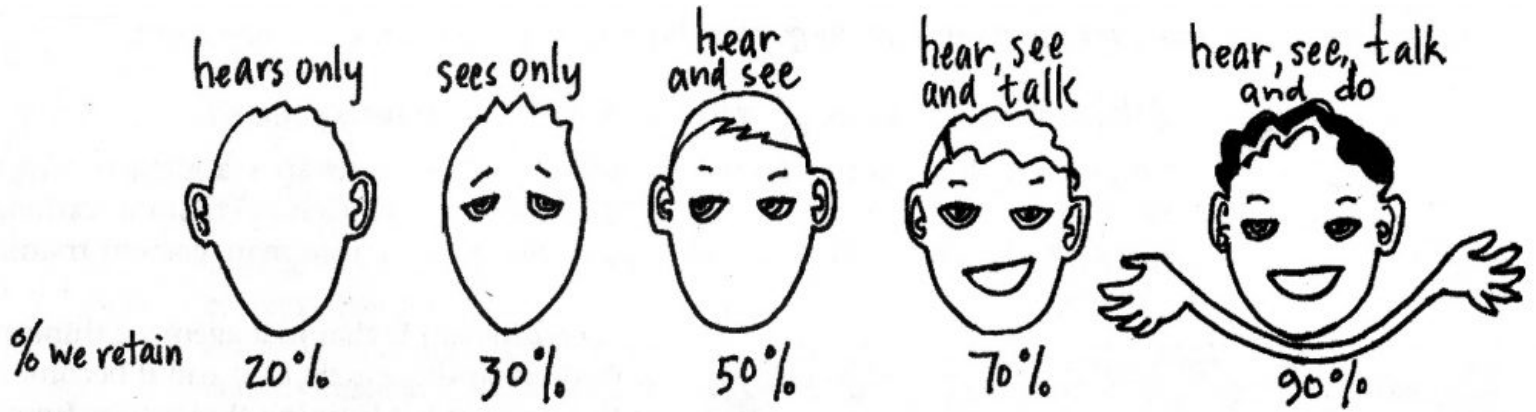
Google Meet

Presentations
Discussion
Ad-hoc Meetings

slack

Questions
Discussion

Hear, See, Talk, Do



Short History

May 2019	Decision of collaboration with GFDL
June 2019	Kickoff
January 2019	Teams complete
	Hard work
October 2019	FV3 port using DSL validates Several GFS parametrizations ported using DSL
on-going	Refactor. Performance.

Goals of Workshop

- Learn what a domain-specific language (DSL) can do / cannot do.
- Understand how a DSL helps in writing hardware-agnostic, maintainable code without sacrificing performance.
- Experience an interactive, Python-based development environment on a supercomputer for model development

What is a domain-specific language?

Domain-specific Language

A domain-specific language (DSL) is a programming language with concepts tailored to a specific class of problems.

By sacrificing generality, DSLs can strike a better balance of performance, portability and productivity.

Taxonomy of Abstractions

Level of abstraction

Domain-specific languages & abstractions

Kernels & Schedule

HPC programming languages & libraries

(here there be dragons…)

$$\nabla^2 u$$

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{j+1,j} - 2u_{i,j} + u_{j-1,j}}{\Delta y^2}$$

```
on domain:
 - 4 u[] + u[i+1] + u[i-1] + u[j+
```

```
for i = 1, ni
  for j = 1, nj
    for k = 1, nk
      - 4 u[i,j,k] +
                   +
```

```
!$acc parallel presen
!$acc loop vector col
for i = 1, ni
  for j = 1, nj
```

C++, Fortran
MPI, OpenMP
OpenACC

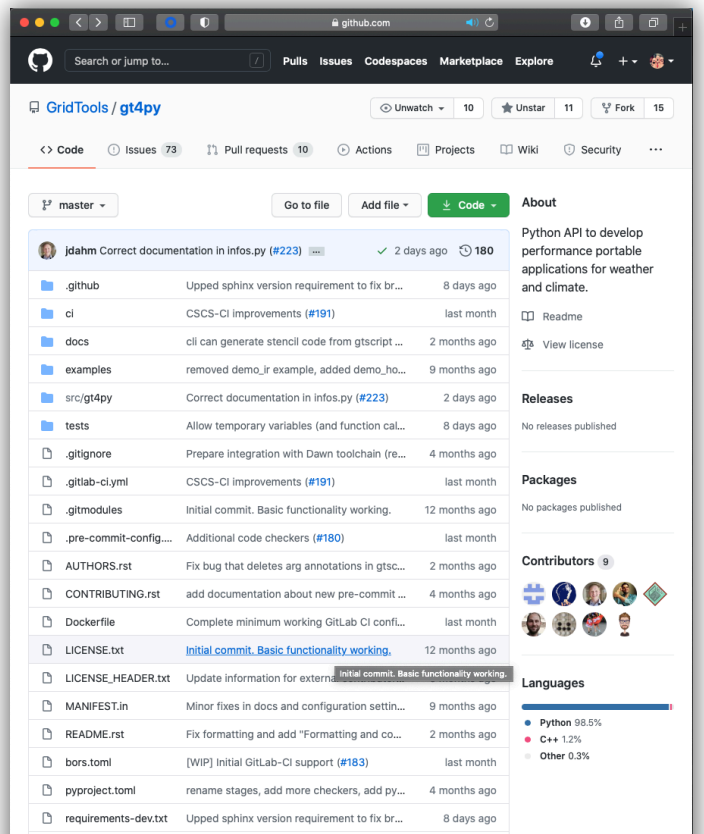
Kokkos
RAJA

GridTools C++

GT4Py

GT4Py

- Open-source project under active development by CSCS and Vulcan
  https://github.com/GridTools/gt4py
- On-going collaboration with MeteoSwiss, ECMWF, and others
- Part of the GridTools ecosystem of tools and libraries for weather and climate
- GT4Py is a domain-specific *library* which exposes a domain-specific *language* (GTScript) to express stencil computations.
- GTScript is embedded in Python (eDSL)
- Emphasis on tight integration with scientific Python stack



Example: Laplacian

Compute the horizontal Laplacian of a three-dimensional field on a regular, Cartesian grid

$$f_{\text{out}} = \nabla_h^2 f_{\text{in}}$$

2nd-order finite difference approximation assuming $\Delta x = \Delta y = 1$

```
real(kind=8) :: in_field(ni, nj, nk)
real(kind=8) :: out_field(ni, nj, nk)

integer :: i, j, k

do k = 1, nk
  do j = 1 + nhalo, nj - nhalo
    do i = 1 + nhalo, ni - nhalo
      out_field(i, j, k) =           &
        -4.0d0 * 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)
    end do
  end do
end do
```

Definition function

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

Definition function (“Stencil”)

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

DSL provides additional keywords
(import optional; standard Python objects)

Definition function

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

Computations / stencils are defined as regular (named) function

Definition function

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

Input and output fields

Definition function

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

Field descriptors as mandatory type annotations

Data Types

- GT4Py supports standard Python and numpy data types
  - `bool`, `int`, `float`
  - `np.bool`, `np.int32`, `np.int64`, `np.float32`, `np.float64`
- Automatic upcasting from integer types to floating point types

Definition function

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

Computations must be inside a computation context. Think of it as being a vertical loop (k).

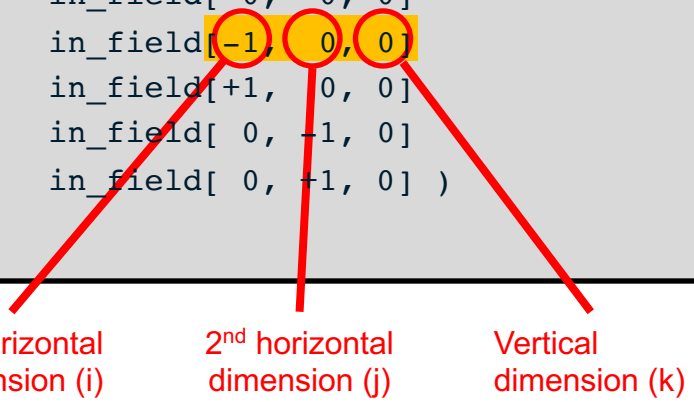
Definition function

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

Each statement (or **stage**) can be thought of as a horizontal loop (ij)

Definition function

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```



Neighboring points are accessed using offsets

Definition function

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
    with computation(PARALLEL), interval(...):
        in_field = out_field
```

There can be multiple computations and/or intervals

Definition function

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval

def laplacian_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

No explicit loops!
No explicit data storage order!
No return statement!

Compilation

```
from gt4py import gtscript

backend = “numpy”
laplacian = gtscript.stencil(backend, laplacian_def)
```

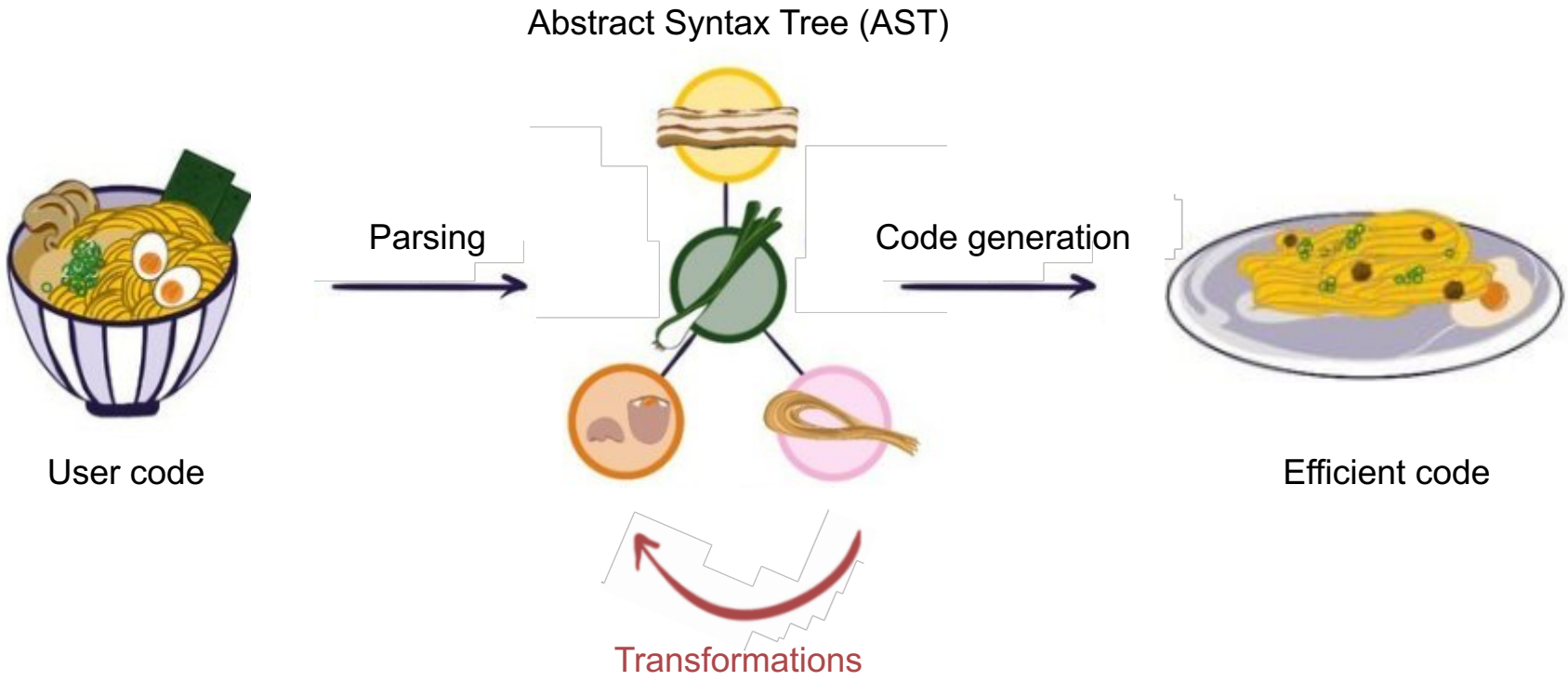
A stencil needs to be compiled for a given backend

1. Generate optimized code for target architecture
2. Compile the generated code
3. Build Python bindings to that code and link dynamically

Different backends are available.

“debug” and “numpy” are Python backends useful for development.

Domain-specific Compiler



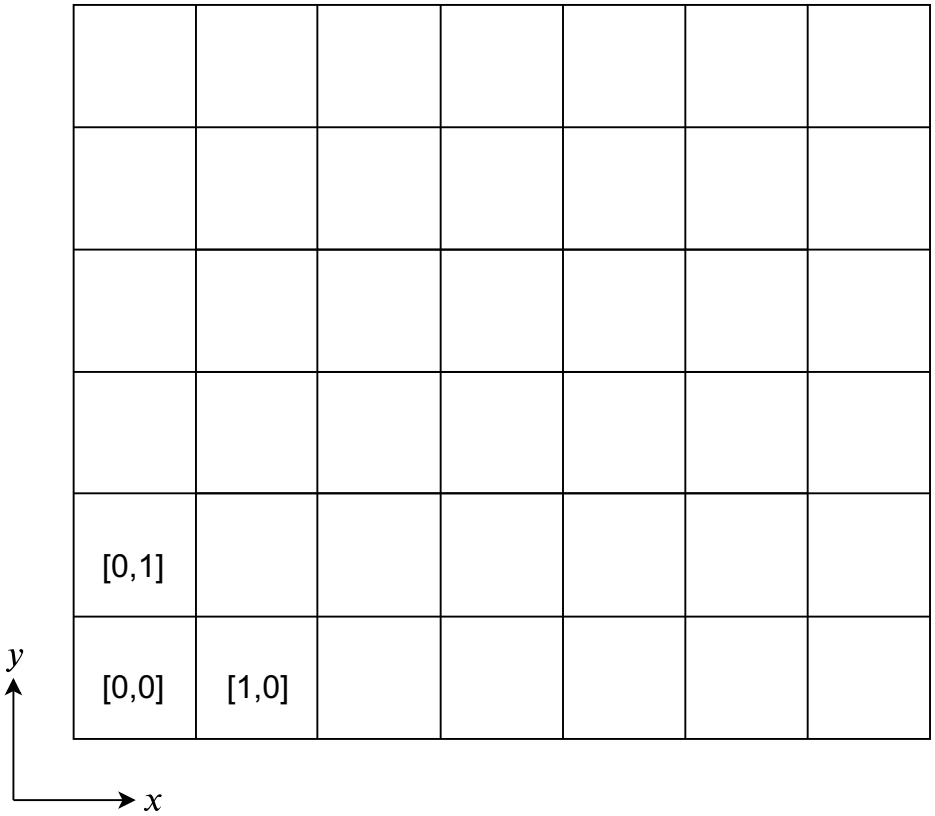
Decorator

```
import numpy as np
from gt4py.gtscript import Field, PARALLEL, computation, interval, stencil

@stencil(backend="numpy")
def laplacian(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (
            - 4. * in_field[ 0,  0, 0]
            +      in_field[-1,  0, 0]
            +      in_field[+1,  0, 0]
            +      in_field[ 0, -1, 0]
            +      in_field[ 0, +1, 0] )
```

Decorator to compile directly

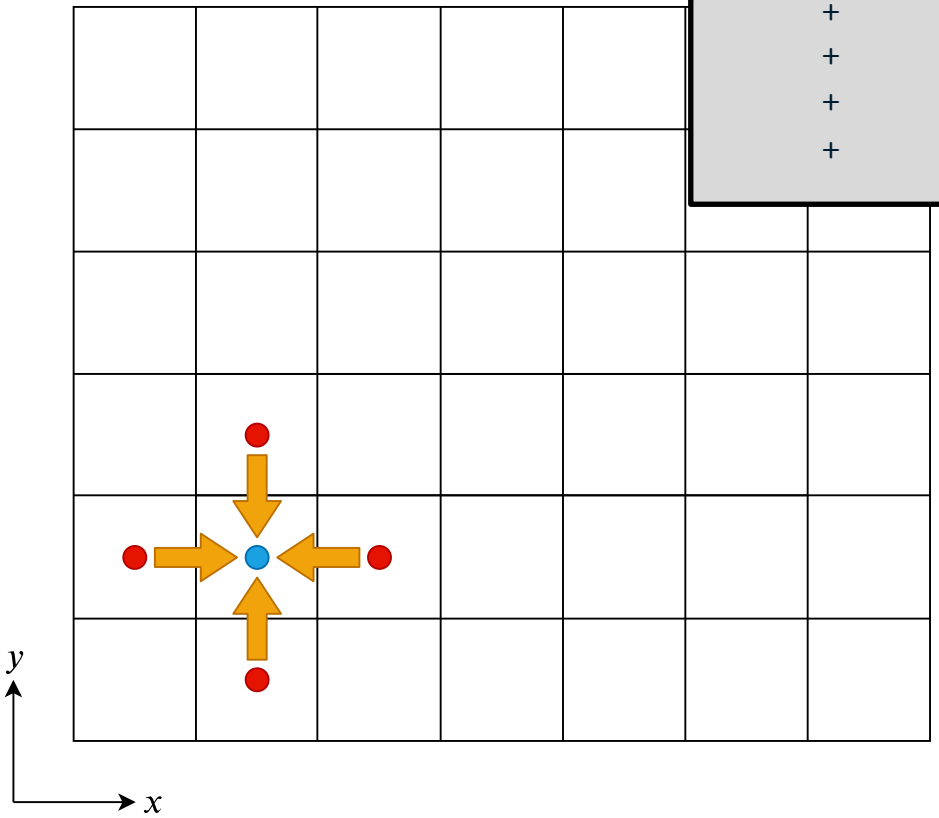
What is missing?

Horizontal Coordinates (IJ)

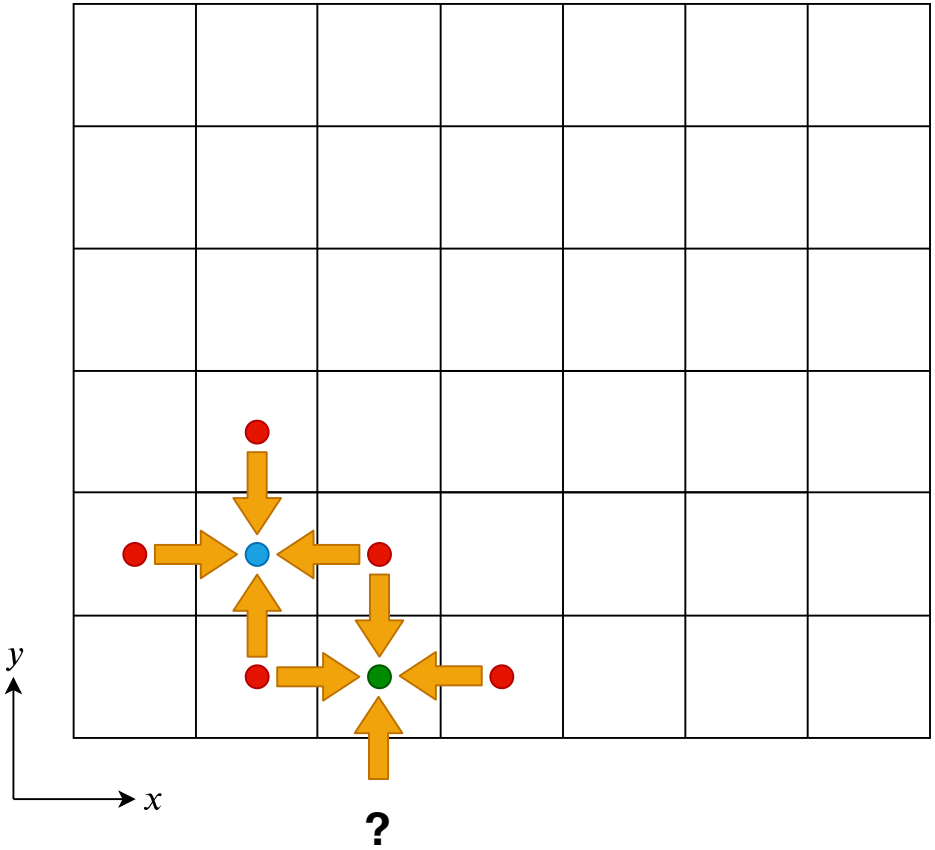


Horizontal Dependencies

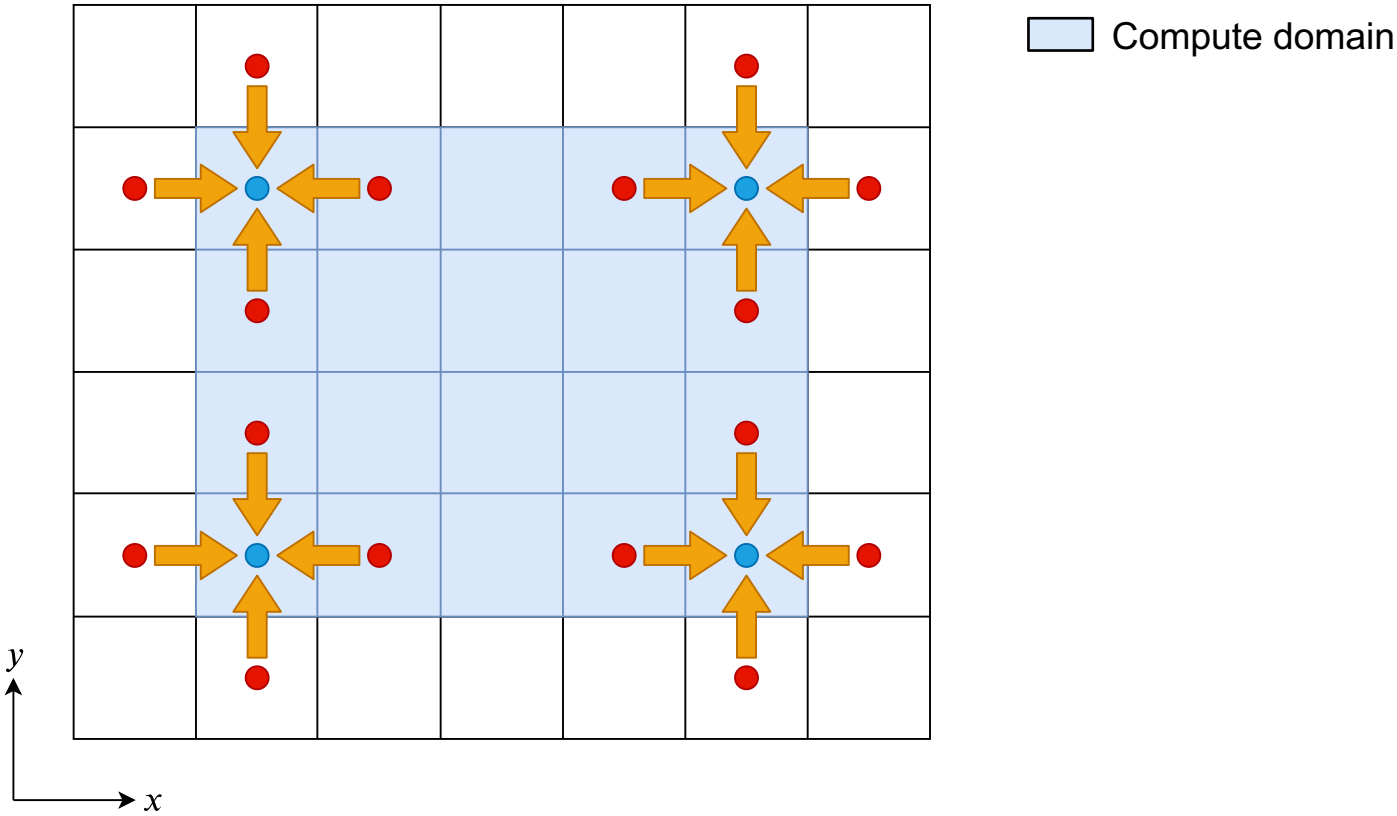
```
out_field = (
    - 4. * in_field[ 0,  0, 0]
     +      in_field[-1,  0, 0]
     +      in_field[+1,  0, 0]
     +      in_field[ 0, -1, 0]
     +      in_field[ 0, +1, 0] )
```



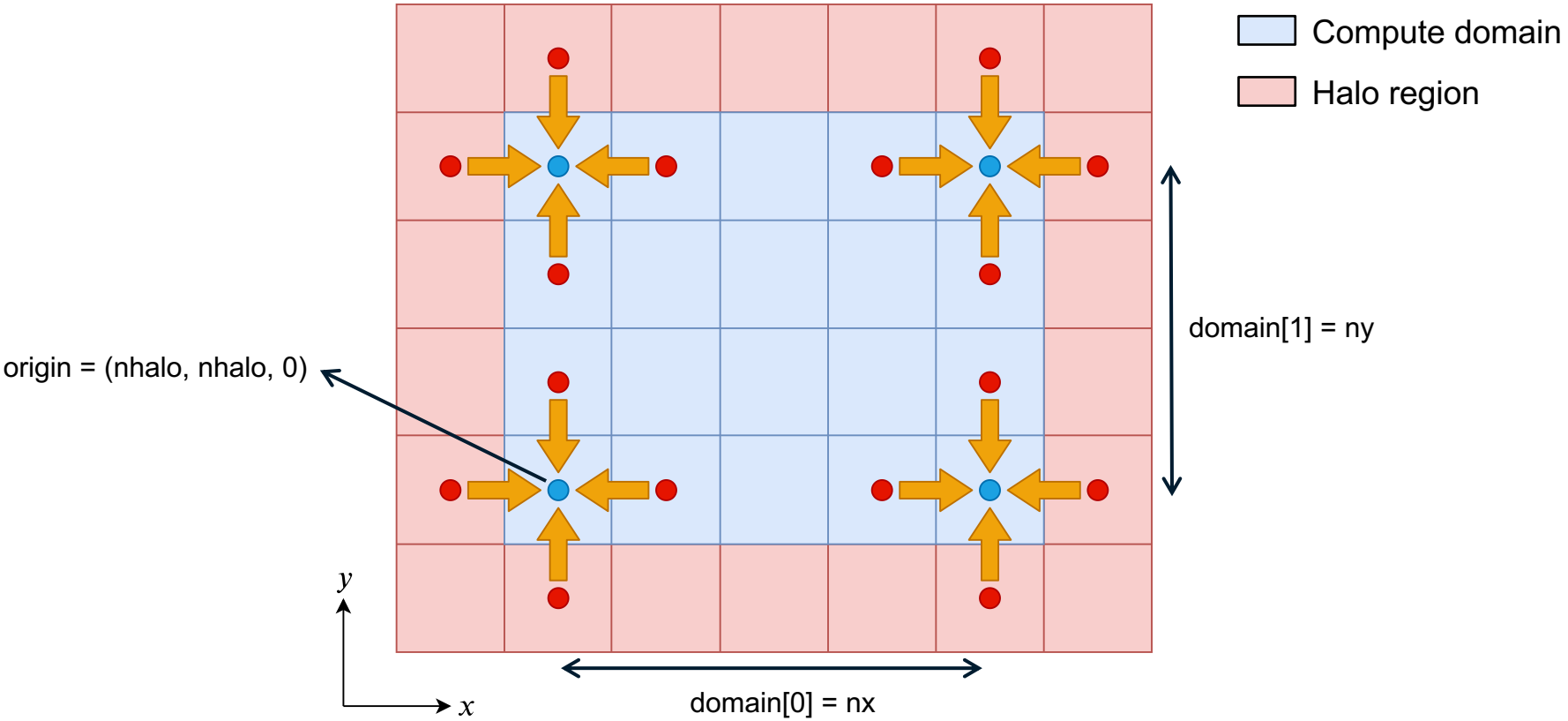
Out-of-bounds Access



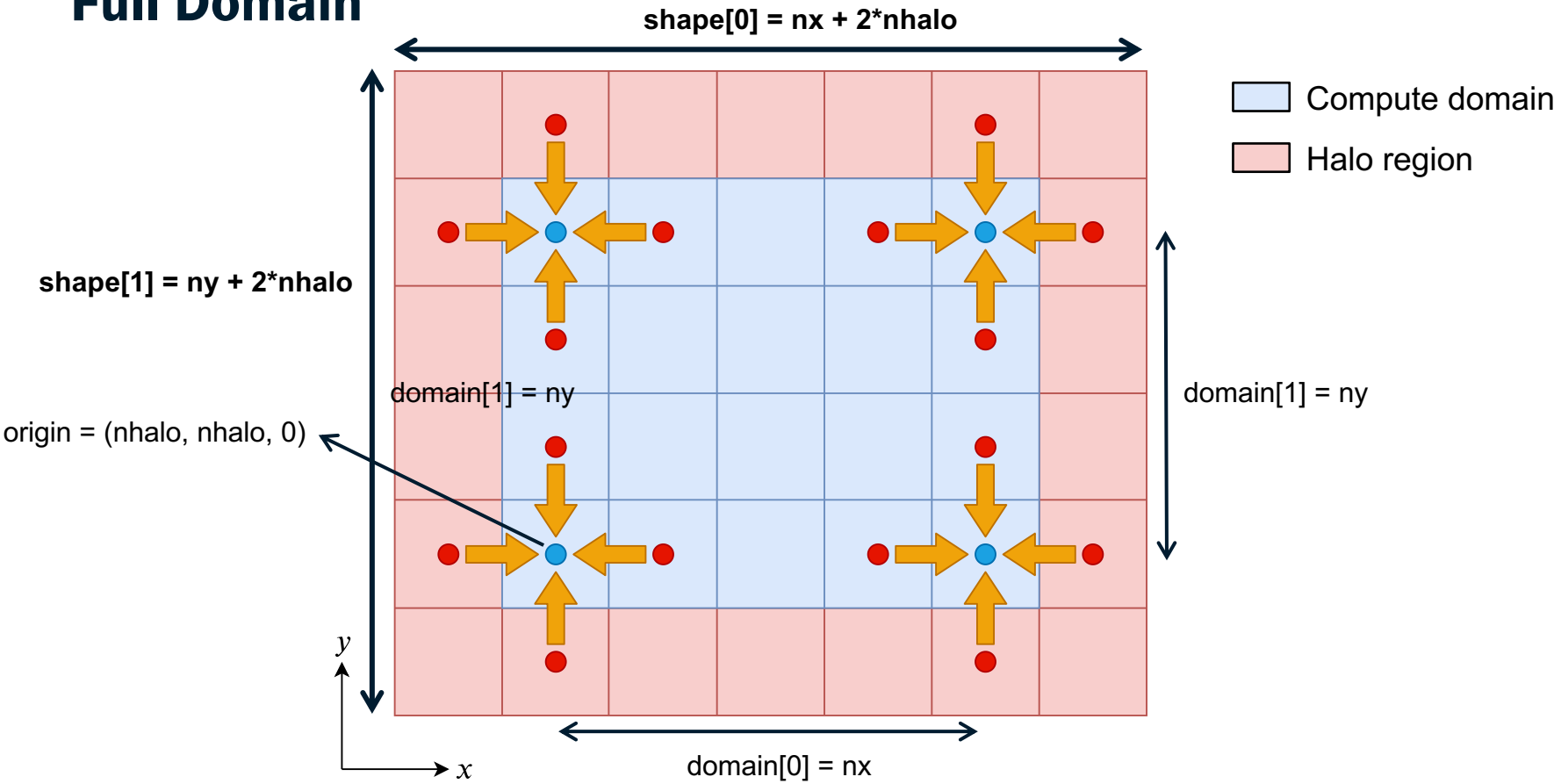
Compute Domain



Halo Region


Compute domain
Halo region
origin = (nhalo, nhalo, 0)
domain[1] = ny
domain[0] = nx
y
x

Full Domain


shape[0] = nx + 2*nhalo
shape[1] = ny + 2*nhalo
domain[1] = ny
origin = (nhalo, nhalo, 0)
domain[1] = ny
Compute domain
Halo region
y
x
domain[0] = nx

Storages

```
nhalo = 3
nx, ny, nz = 128, 128, 64
shape = (nx + 2*nhalo, ny + 2*nhalo, nz)
alignment = (nhalo, nhalo, 0)
out_field = gt.storage.zeros(backend, alignment, shape, dtype=np.float64)
```

- Backend-optimal **storage order**, **alignement** and **padding**
- Behave (almost) like numpy arrays
- Safe host and device copies for convenience

```
out_field.synchronize() # ensures both host and device copy are up-to-date
out_field.host_to_device() # copy from host to device
out_field.device_to_host() # copy from device to host
```

Storages: Usage

```
# create storage from NumPy array
in_field = gt.storage.from_array(
    np.random.rand(nx + 2*nhalo, ny + 2*nhalo, nz),
    backend, origin, shape, dtype=np.float64
)

# also supported are empty(), ones(), copy(), ...

# storages can be accessed as NumPy arrays
in_field[0, 0, 0] = 42.
print(in_field[0, 0, 0])

# inspect the storage
in_field.shape
in_field.backend
in_field.strides
```

Execution

```
origin = (nhalo, nhalo, 0)
domain = (nx, ny, nz)
laplacian(
    in_field=in_field, out_field=out_field,
    origin=origin, domain=domain
)
```

- The compilation returns a callable object (`laplacian`) which must be invoked with GT4Py storages
- `out_field` now contains the result of the computation

Execution

```
origin = (nhalo, nhalo, 0)
domain = (nx, ny, nz)
laplacian(
    in_field=in_field, out_field=out_field,
    origin=origin, domain=domain
)
```

- The compilation returns a callable object (`laplacian`) which must be invoked with GT4Py storages
- `out_field` now contains the result of the computation

Storages holding the data are bound to the symbols used in the stencil definition.

Execution

```
origin = (nhalo, nhalo, 0)
domain = (nx, ny, nz)
laplacian(
    in_field=in_field, out_field=out_field,
    origin=origin, domain=domain
)
```

- The compilation returns a callable object (`laplacian`) which must be invoked with GT4Py storages
- `out_field` now contains the result of the computation

Origin and extent of the compute domain define where results are computed.

Execution

```
origin = (nhalo, nhalo, 0)
domain = (nx, ny, nz)
laplacian(
    in_field=in_field, out_field=out_field,
    origin=origin, domain=domain
)
```

- The compilation returns a callable object (`laplacian`) which must be invoked with GT4Py storages
- `out_field` now contains the result of the computation

Current Limitation

- All storages have to be 3D and the same size
- Individual dimensions can be set to 1 to achieve 2D and 1D storages (see later)

Questions?

Hands-on Session

Now it's your turn!

Session-1A.1.ipynb to work on

- Apply concepts presented in this session
- Generate and inspect code for a stencil using different backends

See you on Slack!