Workshop on

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

Content: Additional Concepts I
(compile-time vs. runtime, conditionals, math functions)

Presenter: Oliver Elbert

Learning goals for this session

- External variables and compile-time
- Understand use of conditionals in stencils
- Familiarity with math functions in stencils

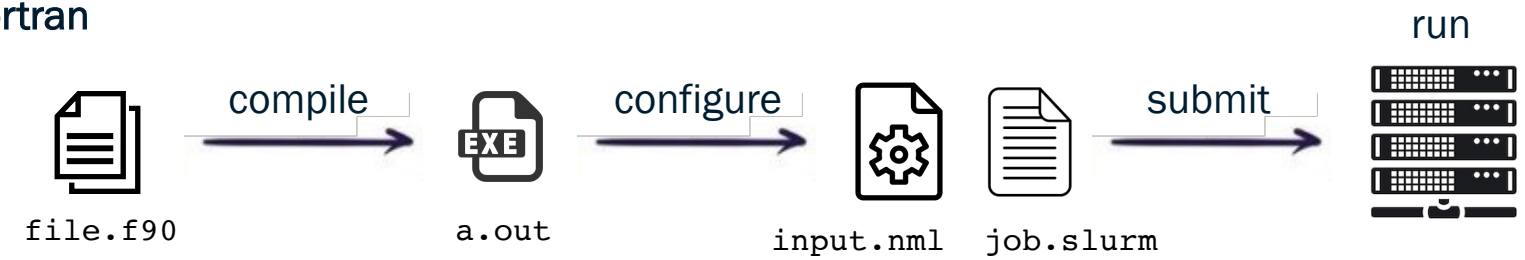
Compile-time values

Because GT4Py is compiled just-in-time, any constant can be evaluated at compile-time, which allows the compiler to further optimize the code.

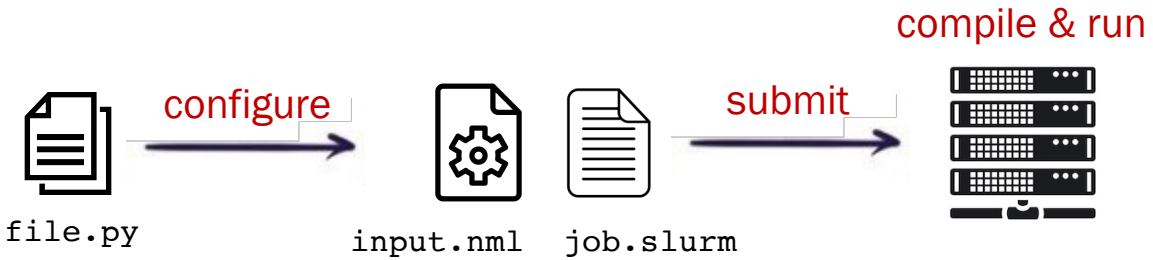
Compile-time vs. Run-time

The meaning of compile-time is a flavor of just-in-time (JIT) in Python

Fortran



Python DSL



When is Compile-time?

```
@gtscript.stencil(backend=backend)
def my_stencil(x: sd, dx: float):
    with computation(PARALLEL), interval(...)
    …
```

vs

```
def my_stencil_fn(x: sd, dx: float):
    with computation(PARALLEL), interval(...)
    …

stencil_call_1 = gtscript.stencil(
    definition = my_stencil_fn
    backend = backend
)
```

Stencils are compiled by invoking `gtscript.stencil()` either when you define the stencil or later.

If you decorate when you define the function you use the decorated function:
`my_stencil(in_field, x)`

If you explicitly call `gtscript.stencil()` later, then use the stencil you define in the call:
`stencil_call_1(in_field, x)`

Externals and Compile-time Values

```
C = 3
@gtscript.stencil(backend=backend)
def my_stencil(x: sd, dx: float):
    with computation(PARALLEL), interval(...)
    x[0, 0, 0] = x[0, 0, 0] + dx * C
```

Compiles to:

```
parfor k in range(kmin, kmax):
    parfor i, j in ...:
        x[i, j, k] = x[i, j, k] + dx * 3
```

Variables in GT4Py are bound to either:

- Fields – read/write at run-time
- Parameters – scalars, read at run-time
- Externals – read at compile-time

Fields and parameters are passed as arguments to a stencil function

Externals are substituted during compilation

External values are frozen when the `gtscript.stencil()` decorator is invoked, and will not be reevaluated even if the external changes later in the code

Externals and Compile-time Values

```
def avg(a: Field, b: Field):
    from __externals__ import I_OFFSET
    with computation(PARALLEL),
interval(...)
        a = 0.5 * (b + b[I_OFFSET, 0, 0])

avg_right_stencil = gtscript.stencil(
    definition = avg,
    backend = backend
    externals = {"I_OFFSET": 1}
)

avg_left_stencil = gtscript.stencil(
    definition = avg,
    backend = backend
    externals = {"I_OFFSET": -1}
)
```

External values are frozen when the `gtscript.stencil()` decorator is invoked, and will not be reevaluated even if the external changes later in the code

If you want to change the external value you have to re-decorate the stencil and pass it explicitly, equivalent to defining a different stencil in the compiled code.

This allows you to re-use Python code while changing the numerics, such as for corners and edges.

Externals and Compile-time Values

```
def avg(a: Field, b: Field):
    from __externals__ import I_OFFSET
    with computation(PARALLEL),
interval(...)
        a = 0.5 * (b + b[I_OFFSET, 0, 0])

avg_right_stencil = gtscript.stencil(
    definition = avg,
    backend = backend
    externals = {"I_OFFSET": 1}
)

avg_left_stencil = gtscript.stencil(
    definition = avg,
    backend = backend
    externals = {"I_OFFSET": -1}
)
```

External values are frozen when the `gtscript.stencil()` decorator is invoked, and will not be reevaluated even if the external changes later in the code

If you want to change the external value you have to re-decorate the stencil and pass it explicitly, equivalent to defining a different stencil in the compiled code.

This allows you to re-use Python code while changing the numerics, such as for corners and edges.

*You can also use externals for control flow

Questions?

Conditionals in GT4Py Stencils

In a highly parallel context, conditional statements can cause ambiguity with off-center operations. How does GT4Py handle these issues?

Why can conditionals be problematic?

Conditionals (Motivation)

sw_core.F90

scalar, domain-decomposition dependent

scalar, namelist-dependent

field, flow-dependent

```
305     if ( sw_corner ) vort(1,    1) = vort(1,    1) + fy(0,   1)
306     if ( se_corner ) vort(npx  ,1) = vort(npx,  1) - fy(npx, 1)
307     if ( ne_corner ) vort(npx,npy) = vort(npx,npy) - fy(npx,npy)
308     if ( nw_corner ) vort(1,  npy) = vort(1,  npy) + fy(0,  npy)
309
310     if ( hydrostatic ) then
311         do j=js-1,jep1+1
312             do i=is-1,iep1
313                 if ( vt(i,j) > 0. ) then
314                     fy1(i,j) = delp(i,j-1)
315                     fy(i,j) =   pt(i,j-1)
316                 else
317                     fy1(i,j) = delp(i,j)
318                     fy(i,j) =   pt(i,j)
319                 endif
320                 fy1(i,j) =  vt(i,j)*fy1(i,j)
321                 fy(i,j) = fy1(i,j)* fy(i,j)
322             enddo
323         enddo
324     end if
```

Conditionals (Scalar)

```
if c_scalar < 0.:
    a[0, 0, 0] = 1.
    b = a[1, 0, 0]
```

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially

Is equivalent to:

```
parfor i, j in ...:
    cond = c_scalar < 0.

parfor i,j in ...:
    if cond:
        a[i, j, k] = 1.

parfor i,j in ...:
    if cond:
        b[i, j, k] = a[i+1, j, k]
```

Conditionals (Field)

```
if c[1, 0, 0] < 0.:
    a[0, 0, 0] = 1.
    b = c[1, 0, 0]
```

Is equivalent to:

```
parfor i, j in ...:
    mask[i, j] = c[i+1, j, k] < 0.

parfor i, j in ...:
    if mask[i, j]:
        a[i, j, k] = 1.

parfor i, j in ...:
    if mask[i, j]:
        b[i, j, k] = c[i+1, j, k]
```

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially
- Field conditionals are handled with a 2D mask

Conditionals (Read/Write-Field)

```
if a[1, 0, 0] < 0.:
    a[0, 0, 0] = 2.
```

Is allowed, and is equivalent to:

```
parfor i, j in ...:
    mask[i, j] = a[i+1, j, k] < 0.

parfor i, j in ...:
    if mask[i, j]:
        a[i, j, k] = 2.
```

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially
- Field conditionals are handled with a mask
  - Self updates are ok
  - Currently only with numpy backend

Conditions (Vertical Loop Order)

```
with computation(FORWARD):
    with interval(1, None):
        if a[0, 0, -1] < 0.:
            a[0, 0, 0] = 2.

[-1, -1, -1] => [-1, 2, -1]
```

is different from

```
with computation(BACKWARD)
    with interval(1, None):
        if a[0, 0, -1] < 0.:
            a[0, 0, 0] = 2.

[-1, -1, -1] => [-1, 2, 2]
```

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially
- Field conditionals are handled with a 2D mask
- For k-dimension, conditionals are evaluated in compute order

Conditionals (Compile-time)

```
if __INLINED(ADV_ORD < 3):
    a[0, 0, 0] = 1.
    b = c[1, 0, 0]
else:
    a[0, 0, 0] = 2.
    b = c[0, 1, 0]
```

Is equivalent to (`ADV_ORD < 3`):

```
parfor i, j in ...:
    a[i, j, k] = 1.
parfor i,j in ...:
    b[i, j, k] = c[i+1, j, k]
```

...or (`ADV_ORD >= 3`):

```
parfor i,j in ...:
    a[i, j, k] = 2.
parfor i,j in ...:
    b[i, j, k] = c[i, j+1, k]
```

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially
- Field conditionals are handled with a mask
- For k-dimension, conditionals are evaluated in compute order
- **If the conditional is known at compile time, compiler can remove ”dead code”**

Conditionals (Compile-time)

```
XVAL = 0
@gtscript.stencil()
def my_stencil(a: sd):
    with computation(PARALLEL), interval(...)
        if XVAL < 3:
            a[0, 0, 0] = 1.
        else :
            a[0, 0, 0] = 2.
```

compiles to

```
parfor i, j in ...:
    cond = 0 < 3
parfor i, j in ...:
    if cond:
        a[i, j, k] = 1.
parfor i, j in ...:
    if not cond:
        a[i, j, k] = 2.
```

Just because externals are read does not mean that conditionals are evaluated at compile-time

Inlining the conditional using `__INLINED(X)` will do this and only generates the relevant branch of the conditional –otherwise you'll have `if 0 < 3` in the compiled code

Conditionals (Compile-time)

```
XVAL = 0
@gtscript.stencil()
def my_stencil(a: sd):
    with computation(PARALLEL), interval(...)
        if __INLINED(XVAL < 3):
            a[0, 0, 0] = 1.
        else :
            a[0, 0, 0] = 2.
```

compiles to

```
parfor i, j in ...:
    a[i, j, k] = 1.
```

Just because externals are read does not mean that conditionals are evaluated at compile-time

Inlining the conditional using `__INLINED(X)` will do this and only generates the relevant branch of the conditional –otherwise you'll have `if 0 < 3` in the compiled code

Conditionals (Compile-time)

```
def my_stencil(a: sd, b: sd, c: sd):
    from __externals__ import ADV_ORD
    with computation(PARALLEL),
interval(...)
        if __INLINED(ADV_ORD == 2):
            a[0, 0, 0] = 1
            b = c[1, 0, 0]
        else :
            a[0, 0, 0] = 2
            b = c[0, 1, 0]

stencil_call_1 = gtscript.stencil(
    definition = my_stencil,
    externals = {"ADV_ORD": 2}
)
stencil_call_2 = gtscript.stencil(
    definition = my_stencil,
    externals = {"ADV_ORD": 4}
)
```

As an example, you can implement multiple advection schemes this way while re-using code, and still optimize at compile-time

Conditionals (Horizontal Dependencies)

```
if c_scalar < 0.:
    a[0, 0, 0] = a[1, 0, 0]
```

Is a race condition (not just in conditionals)
Instead try:

```
if c_scalar < 0.:
    tmp = a[1, 0, 0]
    a[0, 0, 0] = tmp
```

Which gets translated to:

```
parfor i, j in ...:
    cond = c_scalar < 0.
parfor i, j in ...:
    if cond:
        tmp[i, j, k] = a[i+1, j, k]
parfor i, j in ...:
    if cond:
        a[i, j, k] = tmp[i, j, k]
```

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially
- Field conditionals are handled with a mask
- For k-dimension, conditionals are evaluated in compute order
- If the conditional is known at compile time, compiler can remove ”dead code”
- **Be careful about ambiguities**

Conditionals (Horizontal Dependencies)

```
if c[0, 0, 0] < 0.:
    a[0, 0, 0] = 1.
    b = a[1, 0, 0]
```

Neighboring grid points can be in different branches at the same time!

To remove ambiguity, use:

```
if c[0, 0, 0] < 0.:
    a[0, 0, 0] = 1.
if c[0, 0, 0] < 0.:
    b = a[1, 0, 0]
```

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially
- Field conditionals are handled with a mask
- For k-dimension, conditionals are evaluated in compute order
- If the conditional is known at compile time, compiler can remove ”dead code”
- Be careful about ambiguities
  - Cannot write to and read with an offset on the parallel axis inside one field conditional

Conditionals Inside Stencils

```
with computation(FORWARD) interval(...)
    if c[0, 0, 0] < 0.:
        a[0, 0, 0] = 1
        b = a[0, 0, 1]
```

The vertical loop order is specified, so no ambiguity with k-offsets

vs.

```
with computation(PARALLEL), interval(...):
    if c[0, 0, 0] < 0:
        a[0, 0, 0] = 1
        b = a[0, 0, 1]
```

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially
- Field conditionals are handled with a mask
- For k-dimension, conditionals are evaluated in compute order
- If the conditional is known at compile time, compiler can remove ”dead code”
- Be careful about ambiguities
  - Cannot write to and read with an offset on the parallel axis inside one field conditional
  - Unambiguous if compute order is specified though

Conditionals (Ternary)

```
with computation(PARALLEL), interval(...):
    x[0, 0, 0] = dt * ut[0, 0, 0]
    crx[0, 0, 0] = x if x < 0. else 0.
```

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially
- Field conditionals are handled with a mask
- For k-dimension, conditionals are evaluated in compute order
- If the conditional is known at compile time, compiler can remove ”dead code”
- Be careful about ambiguities
- **Ternary if statements are fine too**

Conditionals Inside Stencils

Examples

This comes up in multiple places inside FV3 (e.g. remapping, selecting advection scheme)

In the hands-on session we will try adding a flux limiter to a diffusion stencil

- A statement inside a conditional is executed in parallel over i and j
- Statements are executed sequentially
- Field conditionals are handled with a mask
- For k-dimension, conditionals are evaluated in compute order
- If the conditional is known at compile time, compiler can remove ”dead code”
- Be careful about ambiguities
- k dimension follows compute order
- Ternary if statements are fine too

Questions?

Math Functions

The compiler has access to Python math functions and builtins, and will select the appropriate version for the backend you choose.

Math Functions

```
with computation(PARALLEL), interval(...)
    x = max(x, abs(y[0, 0, -1])
    pkz = exp(k * log(R * x / dz))
```

```
abs                     min
max                     mod
sin                     cos
tan                     asin
acos                    atan
sqrt                    exp
log                     isfinite
isinf                   isnan
floor                   ceil
trunc
```

- GT4py has access to basic python and math functions

Math Functions

Example

"Filling in" negative tracer values uses both conditionals and builtin functions

We can try an example in both parallel and ordered axes

- GT4py has access to basic python and math functions

Questions?

Let’s put it all together!

Hands-on Session

Now it's your turn!

Session-2A.ipynb

- Hands-on with conditionals and externals
- Using conditional statements to limit diffusive fluxes
- Filling negative values in a field with builtins and conditional statements

See you on Slack! **Next huddle at 1:30 pm EST.**