Workshop on

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

Content: Basic Concepts III
(functions, vertical loops, intervals, lower dim fields)

Presenter: Eddie Davis

Learning Goals

- Writing modular code with reusable GTScript functions.
- Understanding the different loop orders in stencil computations.
- Using intervals to apply different computations at boundary conditions.
- Applying this knowledge to implement patterns like tridiagonal solves.

Code Reuse in Fortran

sfc_sice.F (3-layer sea-ice scheme)

```
525   !  --- ...  compute ice temperature
526
527           bi   = hfd(i)
528           ai   = hfi(i) - sneti(i) + ip - tice(i)*bi  ! +v sol input here
529           k12  = ki4*ks / (ks*hice(i) + ki4*snowd(i))
530           k32  = (ki+ki) / hice(i)
531
532           wrk    = one / (dt6*k32 + dici*hice(i))
533           a10    = dici*hice(i)*dt2i + k32*(dt4*k32 + dici*hice(i))*wrk
534           b10    = -di*hice(i) * (ci*stsice(i,1) + li*tfi/stsice(i,1))  &
535       &          * dt2i - ip                                            &
536       &          - k32*(dt4*k32*tfw + dici*hice(i)*stsice(i,2)) * wrk
537
538           wrk1  = k12 / (k12 + bi)
539           a1    = a10 + bi * wrk1
540           b1    = b10 + ai * wrk1
541           c1    = dili * tfi * dt2i * hice(i)
542
543           stsice(i,1) = -(sqrt(b1*b1 - 4.0d0*a1*c1) + b1)/(a1+a1)
544           tice(i) = (k12*stsice(i,1) - ai) / (k12 + bi)
545
546           if (tice(i) > tsf) then
547             a1 = a10 + k12
548             b1 = b10 - k12*tsf
549             stsice(i,1) = -(sqrt(b1*b1 - 4.0d0*a1*c1) + b1)/(a1+a1)
550             tice(i) = tsf
551             tmelt   = (k12*(stsice(i,1)-tsf) - (ai+bi*tsf)) * delt
```

Quadratic formula!

Why do we not use a function call?

Lap(Lap()) Revisited

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

def diffusion_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        alpha = 1.0 / 32.0
        tmp_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_field = (
            - 4. * tmp_field[ 0, 0, 0]
            +      tmp_field[-1, 0, 0]
            +      tmp_field[ 1, 0, 0]
            +      tmp_field[0, -1, 0]
            +      tmp_field[ 0, 1, 0])
        out_field = in_field[0, 0, 0] - alpha * out_field[0, 0, 0]
```

Functions

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

@gtscript.function
def laplacian(in_field: Field[np.float64]):
    return (- 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])

def diffusion_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        alpha = 1.0 / 32.0
        tmp_field = laplacian(in_field)
        out_field = laplacian(tmp_field)
        out_field = in_field[0, 0, 0] - alpha * out_field[0, 0, 0]
```

Functions

```
@gtscript.function
def function(in_1[: Type_1], …, in_n[: Type_n]):
    out_1 = stmt_1
    …
    [tmp_i = stmt_j]
    …
    out_m = stmt_m
    return out_1, …, out_m
```

- Functions allow repeated code blocks to be defined in one place
- Statements in functions are inlined before code generation so there is no call overhead
- Functions can have varying numbers of inputs and outputs that can be fields or scalars
- Function inputs are immutable

Functions do not contain computations or intervals.

Functions

```
@gtscript.function
def function(in_1[: Type_1], …, in_n[: Type_n]):
    out_1 = stmt_1
    …
    [tmp_i = stmt_j]
    …
    out_m = stmt_m
    return out_1, …, out_m
```

- Functions allow repeated code blocks to be defined in one place
- Statements in functions are inlined before code generation so no call overhead
- Functions can have varying numbers of inputs and outputs that can be fields or scalars
- Function inputs are immutable

Decorator to declare a function

Functions

```
@gtscript.function
def function(in_1[: Type_1], …, in_n[: Type_n]):
    out_1 = stmt_1
    …
    [tmp_i = stmt_j]
    …
    out_m = stmt_m
    return out_1, …, out_m
```

- Functions allow repeated code blocks to be defined in one place
- Statements in functions are inlined before code generation so no call overhead
- Functions can have varying numbers of inputs and outputs that can be fields or scalars
- Function inputs are immutable

Type hints are optional in the arguments of functions (fields or scalars)

Functions

```
@gtscript.function
def function(in_1[: Type_1], …, in_n[: Type_n]):
    out_1 = stmt_1
                …
    [tmp_i = stmt_j]
                …
    out_m = stmt_m
    return out_1, …, out_m
```

- Functions allow repeated code blocks to be defined in one place
- Statements in functions are inlined before code generation so no call overhead
- Functions can have varying numbers of inputs and outputs that can be fields or scalars
- Function inputs are immutable

Function body is a sequence of statements (or conditionals).

Functions

```
@gtscript.function
def function(in_1[: Type_1], …, in_n[: Type_n]):
    out_1 = stmt_1
    …
    [tmp_i = stmt_j]
    …
    out_m = stmt_m
    return out_1, …, out_m
```

- Functions allow repeated code blocks to be defined in one place
- Statements in functions are inlined before code generation so no call overhead
- Functions can have varying numbers of inputs and outputs that can be fields or scalars
- Function inputs are immutable

Functions end with a return statement

Function Examples

```
@gtscript.function
def d_dx(f, dx):
  return (f[+1, 0, 0] - f)/dx

@gtscript.function
def avg_x(f):
  return 0.5 * (f[+1, 0, 0] - f)

@gtscript.function
def flux_divergence(fx, fy, area):
  return (fx - fx[1, 0, 0]
        + fy - fy[0, 1, 0]) / area

@gtscript.function
def interpolate_4ord_x(x, a1, a2):
  return a1 * (v + v[1, 0, 0])
       + a2 * (v[-1, 0, 0] + v[2, 0, 0])
```

Functions are the workhorse feature that allow for code reuse and modularity.

Models can build a library of basic numerical operators which can be re-used throughout the code.

Current limitations

- Nested function calls are not supported, e.g., `lap(lap(in_field))`
- Recursive functions are not allowed.
- Return statements are required.
- Call depth is limited to 6 calls.

Questions?

Vertical Loops

```
def diffusion_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        alpha = 1.0 / 32.0
        tmp_field = laplacian(in_field)
        out_field = laplacian(tmp_field)
        …
```

Loop Order

Pseudo Code

```
parfor k in range(k_begin, k_end):
    parfor i,j in …:
        alpha = 1.0 / 32.0
        tmp_field[i, j, k] = laplacian(in_field, i, j, k)
        out_field[i, j, k] = laplacian(tmp_field, i, j, k)
        …
```

Vertical Loops

```
def diffusion_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(FORWARD), interval(...):
        alpha = 1.0 / 32.0
        tmp_field = laplacian(in_field)
        out_field = laplacian(tmp_field)
        …
```

Loop Order

Pseudo Code

```
for k in range(k_begin, k_end):
    parfor i,j in …:
        alpha = 1.0 / 32.0
        tmp_field[i, j, k] = laplacian(in_field, i, j, k)
        out_field[i, j, k] = laplacian(tmp_field, i, j, k)
        …
```

Vertical Loops

```
def diffusion_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(BACKWARD), interval(...):
        alpha = 1.0 / 32.0
        tmp_field = laplacian(in_field)
        out_field = laplacian(tmp_field)
        …
```

Loop Order

Pseudo Code

```
for k in range(k_end - 1, k_begin - 1, -1):
    parfor i,j in …:
        alpha = 1.0 / 32.0
        tmp_field[i, j, k] = laplacian(in_field, i, j, k)
        out_field[i, j, k] = laplacian(tmp_field, i, j, k)
        …
```

Intervals

```
def diffusion_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        alpha = 1.0 / 32.0
        tmp_field = laplacian(in_field)
        out_field = laplacian(tmp_field)
        …
```

Loop Interval

Pseudo Code

```
parfor k in range(k_begin, k_end):
    parfor i,j in …:
        alpha = 1.0 / 32.0
        tmp_field[i, j, k] = laplacian(in_field, i, j, k)
        out_field[i, j, k] = laplacian(tmp_field, i, j, k)
        …
```

Intervals

```
def diffusion_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(1, None):
        alpha = 1.0 / 32.0
        tmp_field = laplacian(in_field)
        out_field = laplacian(tmp_field)
        …
```

Loop Interval

Pseudo Code

```
parfor k in range(k_begin + 1, k_end):
    parfor i,j in …:
        alpha = 1.0 / 32.0
        tmp_field[i, j, k] = laplacian(in_field, i, j, k)
        out_field[i, j, k] = laplacian(tmp_field, i, j, k)
        …
```

Intervals

```
def diffusion_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(0, -1):
        alpha = 1.0 / 32.0
        tmp_field = laplacian(in_field)
        out_field = laplacian(tmp_field)
        …
```

Loop Interval

Pseudo Code

```
parfor k in range(k_begin, k_end - 1):
    parfor i,j in …:
        alpha = 1.0 / 32.0
        tmp_field[i, j, k] = laplacian(in_field, i, j, k)
        out_field[i, j, k] = laplacian(tmp_field, i, j, k)
        …
```

Intervals

```
def diffusion_def(in_field: Field[np.float64], out_field: Field[np.float64]):
    with computation(PARALLEL), interval(-2, None):
        alpha = 1.0 / 32.0
        tmp_field = laplacian(in_field)
        out_field = laplacian(tmp_field)
        …
```

Loop Interval

Pseudo Code

```
parfor k in range(k_end - 2, k_end):
    parfor i,j in …:
        alpha = 1.0 / 32.0
        tmp_field[i, j, k] = laplacian(in_field, i, j, k)
        out_field[i, j, k] = laplacian(tmp_field, i, j, k)
        …
```

How about an example?

Vertical Advection

- Vertical grid levels are unevenly distributed in many climate models
- Grid cells close to surface typically are only ~20 m thick
- Vertical advection is often solved implicitly to avoid unduly limiting the time step size

$$\frac{\partial \phi}{\partial t} = \frac{\partial w\phi}{\partial z}$$

  - where z is the height
  - w is the vertical velocity

- Solved using the Crank-Nicholson scheme coupled with vertically centered differences

$$\frac{\phi_k^{n+1} - \phi_k^n}{\Delta t} = \frac{1}{2}\left(\frac{w_{k+1}^n \phi_{k+1}^n - w_{k-1}^n \phi_{k-1}^n}{2\Delta z} + \frac{w_{k+1}^{n+1} \phi_{k+1}^{n+1} - w_{k-1}^{n+1} \phi_{k-1}^{n+1}}{2\Delta z} \right)$$

Vertical Advection

- If *w* is known at each grid point at time step *n* + 1, this yields a tridiagonal system for φ^{n+1}
- General Form:

$$\begin{cases} b_1\phi_1^{n+1} + c_1\phi_2^{n+1} = d_1 \\ a_k\phi_{k-1}^{n+1} + b_k\phi_k^{n+1} + c_k\phi_{k+1}^{n+1} = d_k, \qquad k = 2, \ldots, n_z - 1 \\ a_{n_z}\phi_{n_z-1}^{n+1} + b_{n_z}\phi_{n_z}^{n+1} = d_{n_z} \end{cases}$$

- Matrix Form:

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n_z-1} \\ 0 & & & a_{n_z} & b_{n_z} \end{bmatrix} \begin{bmatrix} \phi_1^{n+1} \\ \phi_2^{n+1} \\ \vdots \\ \phi_{n_z}^{n+1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n_z} \end{bmatrix}$$

Tridiagonal Solve

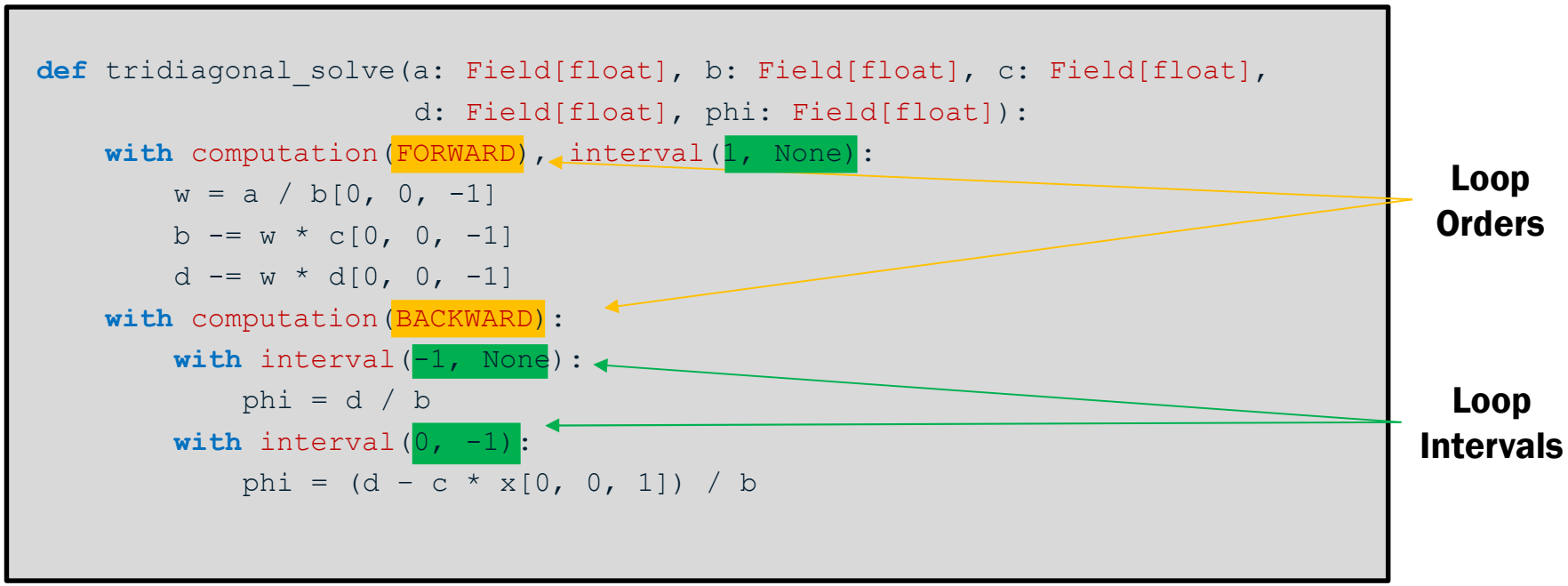
- Tridiagonal systems can be efficiently solved using the Thomas algorithm
- Two sweeps:
    1. **Forward** loop to update *b* and *d*
    2. **Backward** substitution loop

```
for k in 1 .. N:
    w = a[k] / b[k - 1]
    b[k] = b[k] - w * c[k - 1]
    d[k] = d[k] - w * d[k - 1]

phi[N] = d[N] / b[N]
for k in N - 1 .. 1, step -1:
    phi[k] = (d[k] - c[k] * x[k + 1]) / b[k]
```

Tridiagonal Solve

```
def tridiagonal_solve(a: Field[float], b: Field[float], c: Field[float],
                      d: Field[float], phi: Field[float]):
    with computation(FORWARD), interval(1, None):
        w = a / b[0, 0, -1]
        b -= w * c[0, 0, -1]
        d -= w * d[0, 0, -1]
    with computation(BACKWARD):
        with interval(-1, None):
            phi = d / b
        with interval(0, -1):
            phi = (d - c * x[0, 0, 1]) / b
```



Lower Dimensional Fields

- Just as loop orders and intervals can be customized so can data fields
- Fields can be customized by data type or axes
- Field descriptors are defined as `Field[<data_type>, <axes>]`
    1. The *data_type* is either a Python built-in or NumPy data type (default = *float*)
    2. The axes are a combination of the uppercase characters *I, J*, *K* that correspond to the *x*, *y*, and *z* directions, respectively (default = *IJK*)
- Examples:
    1. `Field[float, IJK]`
    2. `Field[float, IJ]`
    3. `Field[np.int, K]`

Counts and Sums

```
def count_and_sum(q: Field[float], dp: Field[float], dm: Field[float],
                  zfix: Field[int, IJ], zsum: Field[float, IJ]):
    with computation(PARALLEL), interval(…):
        zfix = 0
        zsum = 0.0
    with computation(FORWARD):
        with interval(1, -1):
            if (q < 0.0) and (q[0, 0, -1] < 0.0):
                zfix += 1
                q += (q[0, 0, -1] * dp[0, 0, -1]) / dp
            dm = q * dp
        with interval(1, None):
            zsum += dm
            if (zfix > 0):
                q = zsum * dm / dp
```

2D Integer Field

2D Float Field

Implicit Fall Example

```
subroutine implicit_fall(dt, ktop, kbot, ze, vt, dp, q, precip, m1)
   ...
   do k = ktop, kbot
      dz(k) = ze(k) - ze(k + 1)
      dd(k) = dt * vt(k)
      q(k) = q(k) * dp(k)
   enddo
   ! sedimentation: non - vectorizable loop
   qm(ktop) = q(ktop) / (dz(ktop) + dd(ktop))
   do k = ktop + 1, kbot
      qm(k) = (q(k) + dd(k - 1) * qm(k - 1)) / (dz(k) + dd(k))
   enddo
   ! qm is density at this stage
   do k = ktop, kbot
      qm(k) = qm(k) * dz(k)
   enddo
   ! output mass fluxes: non - vectorizable loop
   m1(ktop) = q(ktop) - qm(ktop)
   do k = ktop + 1, kbot
      m1(k) = m1(k - 1) + q(k) - qm(k)
   enddo
   precip = m1(kbot)
   ! update:
   do k = ktop, kbot
      q(k) = qm(k) / dp(k)
   enddo
end subroutine implicit_fall
```

```
def implicit_fall(ze: Field, dp: Field, q: Field, precip: Field,
                  m1: Field, dt: float):
    with computation(FORWARD), interval(…):
        dz = ze - ze[0, 0, 1]
        dd = dt * vt
        q *= dp

    # sedimentation: non - vectorizable loop
    with computation(FORWARD):
        with interval(0, 1):
            qm = q / (dz + dd)
        with interval(1, None):
            qm = (q + dd[0, 0, -1] * qm[0, 0, -1]) / (dz + dd)

    # qm is density at this stage
    with computation(PARALLEL), interval(…):
        qm *= dz

    # output mass fluxes
    with computation(FORWARD), interval(…):
        with interval(0, 1):
            m1 = q - qm
        with interval(1, None):
            m1 = m1[0, 0, -1] + q - qm
        with interval(-1, None):
            precip = m1
    # update:
    with computation(PARALLEL), interval(…):
        q = qm / dp
```

Hands-on Session

Session-1B.ipynb to work on today

- Function examples and limitations
- Forward and backward loops
- Working with 2D and 3D fields

See you on Slack! **Next huddle at 4:30 pm EST.**