Workshop on

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

Content: Basic Concepts II
(horizontal dependencies, temporaries, backends & performance)

Presenter: Tobias Wicky

Learning goals for this session

- Understand the GT4Py execution model
- Understand how horizontal loop bounds work
- Hands-on with different backends and performance

Execution Model (aka “Parallel Model”)

The **execution model** specifies the behavior of elements of a programming language.

By applying the execution model, one can derive the behavior of a program that was written in that programming language.

When “reading code”, we essentially apply the execution model to the code.

See https://github.com/GridTools/concepts/wiki/GTScript-Parallel-model

Execution Model: Vertical

GT4Py

```
@stencil(backend=backend)
def copy_stencil( a: Field[float],
                  b: Field[float]):
  with computation(PARALLEL), interval(...):
      b = a
```

Python pseudocode

```
for k_ in random.sample(range(k, K)):
  b[i:I, j:J, k_] = a[i:I, j:J, k_]
```

- PARALLEL computation has no specified order in k
- Other vertical loop orders exist (see later!)
- Data races can occur

Parallel loop in k-direction

Execution Model: Vertical

GT4Py

```
@stencil(backend=backend)
def vertical_sum( a: Field[float]):
  with computation(PARALLEL), interval(...):
      a = a[0, 0, -1]
```

Python pseudocode

```
for k_ in random.sample(range(k, K)):
  a[i:I, j:J, k_] = a[i:I, j:J, k_-1] + 1
```

- PARALLEL computation has no specified order in k
- Other vertical loop orders exist (see later!)
- Data races can occur

Race condition!
(GT4Py will issue an error)

Execution Model: Horizontal

GT4Py

```
def average_stencil( a: Field[float],
                     b: Field[float]):
  with computation(PARALLEL), interval(...):
      b = 0.5 * (a[1, 0, 0] + a[-1, 0, 0])
```

Python pseudocode

```
for k_ in range(k, K):
  b[i:I, j:J, k_] = 0.5 * (
      a[i+1:I+1, j:J, k_] +
      a[i-1:I-1, j:J, k_]
  )
```

- Horizontal (ij-direction) execution policy for a single statement is always parallel
- Data-races can occur with horizontal dependencies

Execution Model: Horizontal

GT4Py

```
def average_stencil( a: Field[float],
                     b: Field[float]):
  with computation(PARALLEL), interval(...):
      a = 0.5 * (a[1, 0, 0] + a[-1, 0, 0])
```

Python pseudocode

```
for k_ in range(k, K):
  a[i:I, j:J, k_] = 0.5 * (
     a[i+1:I+1, j:J, k_] +
     a[i-1:I-1, j:J, k_]
  )
```

- Horizontal (ij-direction) execution policy for a single statement is always parallel
- Data-races can occur with horizontal dependencies

Execution Model: Summary

- *computations* are executed sequentially in the order they appear in the code
- vertical *intervals* are executed sequentially in the order defined by the *iteration policy* of the *computation*
- every vertical *interval* is executed as a sequential for-loop over the K-range following the order defined by the iteration policy

Note: The DSL compiler often catches unsafe / illegal code,
but checks are not complete.

Horizontal Compute Domains

GT4Py

```
def average_stencil( a: Field[float],
                     b: Field[float]):
  with computation(PARALLEL), interval(...):
      a = 1.
      b = 0.5 * (a[1, 0, 0] + a[-1, 0, 0])
```

Python pseudocode

```
for k_ in range(k, K):
  a[i-1:I+1, j:J, k_] = 1.
  b[i:I, j:J, k_] = 0.5 * (
      a[i+1:I+1, j:J, k_] +
      a[i-1:I-1, j:J, k_]
  )
```

- Horizontal (ij-direction) execution policy for a single statement is always parallel
- Horizontal loop bounds are deduced automatically

Horizontal loops are deduced automatically by dependency and offset analysis.

Horizontal Compute Domains

GT4Py

```
def average_stencil( a: Field[float],
                     b: Field[float]):
  with computation(PARALLEL), interval(...):
      b = 0.5 * (a[1, 0, 0] + a[-1, 0, 0])
      a = 1.
```

Python pseudocode

```
for k_ in range(k, K):
  b[i:I, j:J, k_] = 0.5 * (
      a[i+1:I+1, j:J, k_] +
      a[i-1:I-1, j:J, k_]
  )
  a[i:I, j:J, k_] = 1.
```

- Horizontal (ij-direction) execution policy for a single statement is always parallel
- Horizontal loop bounds are deduced automatically

Horizontal loops are deduced automatically by dependency and offset analysis.

Horizontal Compute Domains: Example

```
def c_sw(...):
  with computation(PARALLEL), interval(...):
    ut = dt2 * ut * dy * sin_sg3[-1, 0, 0]
    vt = dt2 * vt * dx * sin_sg4[0, -1, 0]
    delpc = upwind_step(delp, ut, vt, rarea, 1., 1.)
    ptc = upwind_step(pt, ut, vt, rarea, delp, delpc)
    wc = upwind_step(w, ut, vt, rarea, delp, delpc)
    # compute kinetic energy (ke)
    ucc = uc[1, 0, 0]
    vcc = vc[0, 1, 0]
    ke = 0.5 * dt2 * (ua * ucc + va * vcc)                    [(-1, 0), (-1, 0)]
    # compute absolute vorticity (vort)
    fx = uc * dxc
    fy = vc * dyc
    vort = fx[0, -1, 0] - fx - fy[-1, 0, 0] + fy
    vort = fc + rarea_c * vort
    # transport absolute vorticity
    fy1 = dt2 * (v - uc * cosa_u) / sina_u                    [(0, 0), (-1, 0)]
    fy = fy1[0, -1, 0]                                        [(0, 0), (0, 0)]
    fx1 = dt2 * (u - vc * cosa_v) / sina_v                    [(-1, 0), (0, 0)]
    fx = fx1[-1, 0, 0]                                        [(0, 0), (0, 0)]
    # update time-centered winds on the C-grid
    uc = uc + fy1 * fy + rdxc * (ke[-1, 0, 0] - ke)           [(0, 0), (0, 0)]
    vc = vc - fx1 * fx + rdyc * (ke[0, -1, 0] - ke)           [(0, 0), (0, 0)]
```

Temporary Variables (aka "Temporaries")

GT4Py

```
def average_stencil( a: Field[float]):
  with computation(PARALLEL):
    with interval(...):
      tmp = 0.5 * (a[1, 0, 0] + a[-1, 0, 0])
      a = tmp
```

tmp is a field!

Python pseudocode

```
for k_ in range(k, K):
  tmp[i:I, j:J, k_] = 0.5 * (
      a[i+1:I+1, j:J, k_] +
      a[i-1:I-1, j:J, k_]
  )
  a[i:I, j:J, k_] = tmp[i:I, j:J, k_]
```

- Use temporaries to store intermediate results and avoid race-conditions
- Temporary variables are automatically typed, dimensioned, and allocated

Temporary variables hold intermediate results in a computation.

Temporary Variables (aka “Temporaries”)

GT4Py

```
def average_stencil( a: Field[float]):
  with computation(PARALLEL):
    with interval(...):
      tmp = 1.
      a = 0.5*(tmp[1, 0, 0] + tmp[-1, 0, 0])
```

tmp is extended in i-direction!

Python pseudocode

```
for k_ in range(k, K):
  tmp[i-1:I+1, j:J, k_] = 1.
  a[i:I, j:J, k_] = 0.5 * (
     tmp[i+1:I+1, j:J, k_] +
     tmp[i-1:I-1, j:J, k_]
  )
```

- Use temporaries to store intermediate results and avoid race-conditions
- Temporary variables are automatically typed, dimensioned, and allocated

Horizontal loops are deduced automatically by dependency and offset analysis.

Temporary Variables (aka “Temporaries”)

GT4Py

```
def average_stencil( a: Field[float]
                     b: Field[float]):
  with computation(PARALLEL):
    with interval(...):
      tmp = 0.5 * (a[1, 0, 0] + a[-1, 0, 0])
      b = tmp
```

tmp can be a scalar!
(if statements are fused)

Loop based Python

```
for k_ in range(k, K):
  for i_ in range(i, I):
    for j_ in range(j, J):
      tmp = 0.5 * (
          a[i_+1, j_, k_] +
          a[I_-1, j_, k_]
      )
      b[i_, j_, k_] = tmp
```

- Optimizers will fuse loops and choose the most efficient dimensionality and memory location for temporaries

User should think of temporaries as fields, but their shape may be different in generated code

Temporary Variables: Example

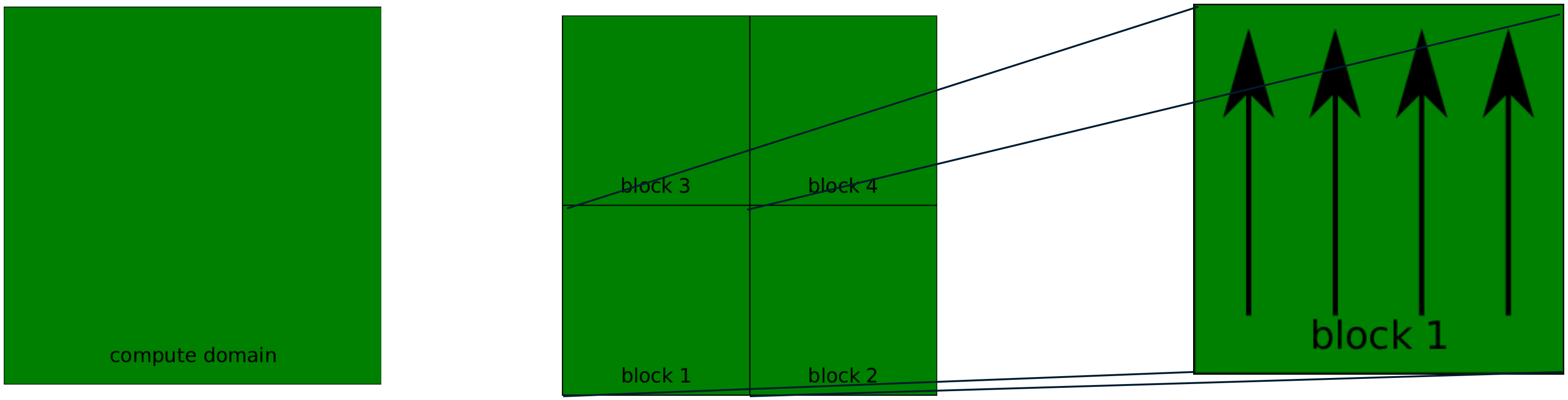
```
def c_sw(...):
  with computation(PARALLEL), interval(...):
    ut = dt2 * ut * dy * sin_sg3[-1, 0, 0]
    vt = dt2 * vt * dx * sin_sg4[0, -1, 0]
    delpc = upwind_step(delp, ut, vt, rarea, 1., 1.)
    ptc = upwind_step(pt, ut, vt, rarea, delp, delpc)
    wc = upwind_step(w, ut, vt, rarea, delp, delpc)
    # compute kinetic energy (ke)
    ucc = uc[1, 0, 0]
    vcc = vc[0, 1, 0]
    ke = 0.5 * dt2 * (ua * ucc + va * vcc)
    # compute absolute vorticity (vort)
    fx = uc * dxc
    fy = vc * dyc
    vort = fx[0, -1, 0] - fx - fy[-1, 0, 0] + fy
    vort = fc + rarea_c * vort
    # transport absolute vorticity
    fy1 = dt2 * (v - uc * cosa_u) / sina_u
    fy = fy1[0, -1, 0]
    fx1 = dt2 * (u - vc * cosa_v) / sina_v
    fx = fx1[-1, 0, 0]
    # update time-centered winds on the C-grid
    uc = uc + fy1 * fy + rdxc * (ke[-1, 0, 0] - ke)
    vc = vc - fx1 * fx + rdyc * (ke[0, -1, 0] - ke)
```

Backends

	debug	numpy	gtx86	gtmc	gtcuda	Prototypes
parallelism	explicit ijk-loops	vectorized syntax	OpenMP for ij-blocking	OpenMP for ij-blocking and i-vectorization	CUDA blocks and threading in ij-direction	CUDA Kokkos AMD ...
storage order	IJK	IJK	KIJ	IJK	IJK	

Backends: Differences

From the execution model:
Horizontal (ij-direction) execution policy for a single statement is always parallel



coarse-grain parallelism

fine-grain parallelism

Backends: Loop-Order

From the execution model:
computations are executed sequentially in the order they appear in the code

Backends: Loop-Order

From the execution model:
computations are executed sequentially in the order they appear in the code

```
def average_stencil( a: Field[float]
                     b: Field[float]):
  with computation(PARALLEL), interval(...):
     b = 0.5 * (a[1, 0, 0] + a[-1, 0, 0])
```

```
for k_ in range(k, K):
  for i_ in range(i, I):
    for j_ in range(j, J):
      b = 0.5 * (
          a[i_+1, j_, k_] +
          a[I_-1, j_, k_]
      )
```

```
for i_ in range(i, I):
  for j_ in range(j, J):
    for k_ in range(k, K):
      b = 0.5 * (
          a[i_+1, j_, k_] +
          a[I_-1, j_, k_]
      )
```

Hands-on Session

Session-1A.2.ipynb for the following hands-on session

- Apply temporaries
- Investigate runtime and performance
- Understand execution model and storage layouts

See you on Slack! **Next huddle at 11:30 am EST.**