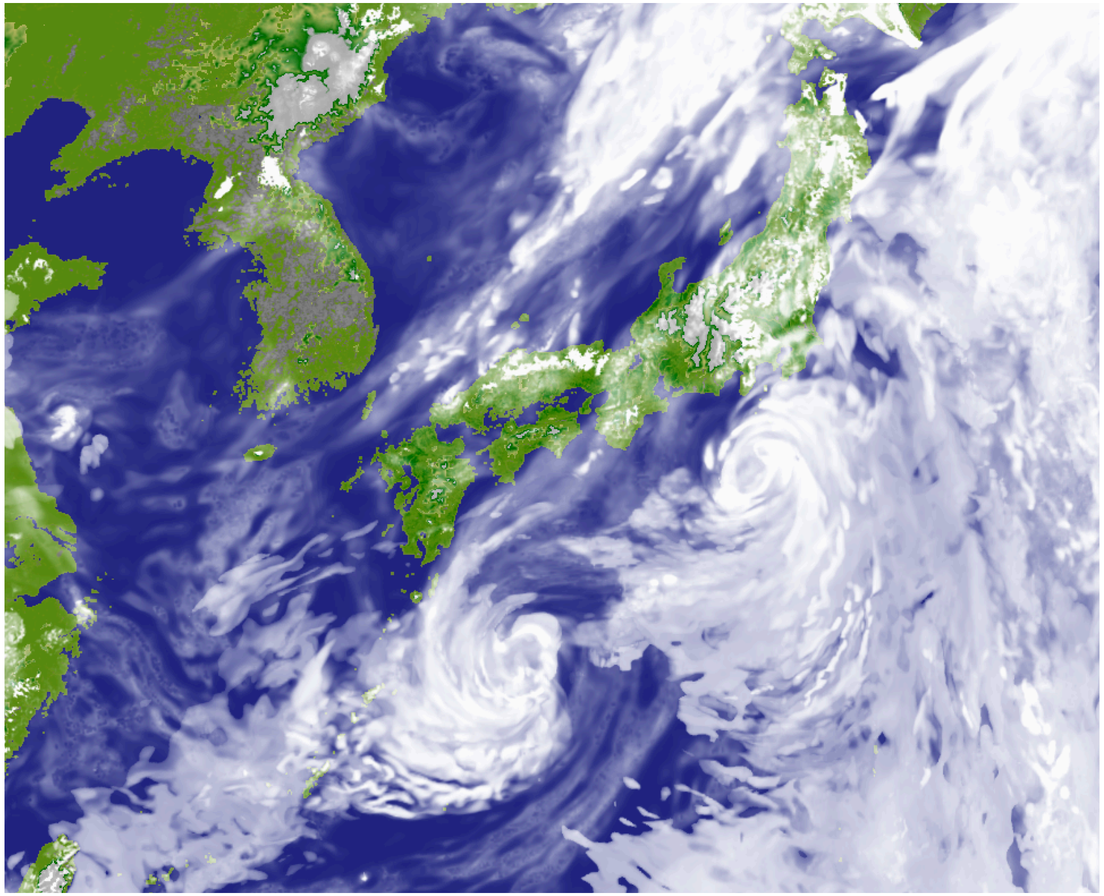
Hybrid Fortran

High Productivity GPU Porting Framework Applied to Japanese Weather Prediction Model



Michel Müller

supervised by
Takayuki Aoki

Tokyo Institute of Technology

AOKI Lab.
Tokyo Institute of Technology

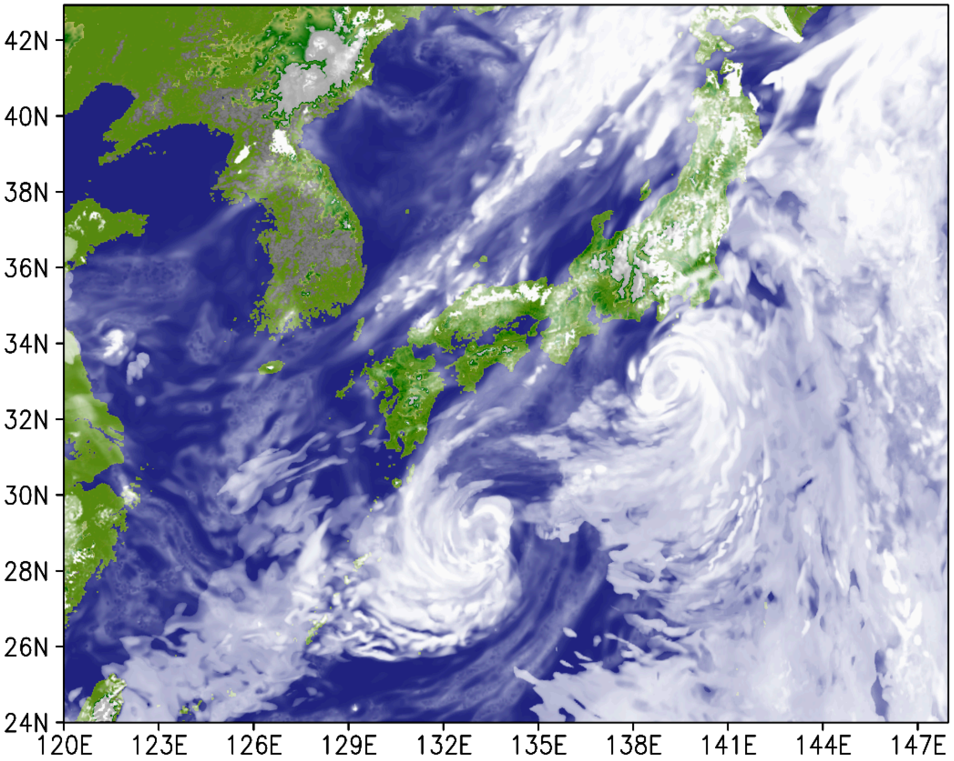
Outline

1. Motivation & Problem Description
2. Proposed Solution
3. Example & Application Status
4. Code Transformation
5. Performance- & Productivity Results
6. Conclusion

ASUCA

What is ASUCA? [6]

- Non-hydrostatic weather prediction model
- Main Japanese mesoscale weather model, in production since end of 2014
- Dynamical + physical core
- Regular grid:
  horizontal domain IJ,
  vertical domain K
- Mostly parallelizeable in IJ, K is mostly sequential



Cloud cover result with ASUCA using a 2km resolution grid and real world data

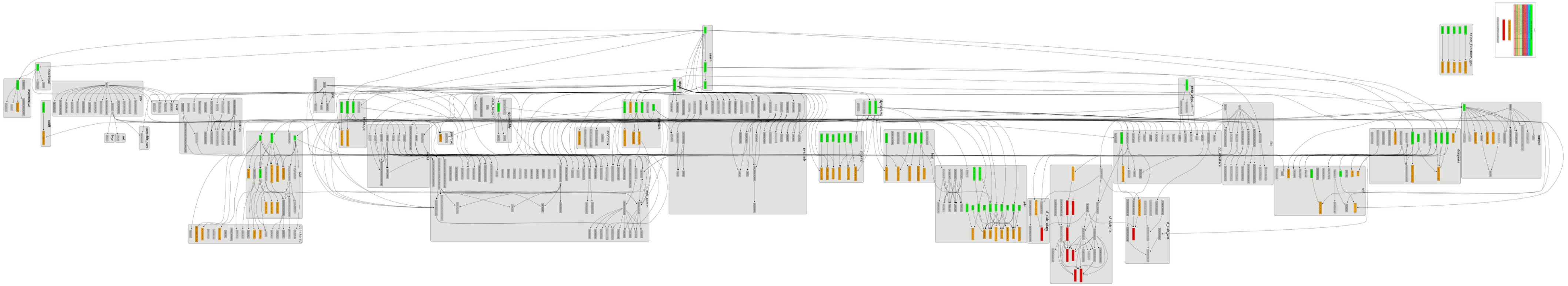
Goals of Hybrid ASUCA

- Performant GPU Implementation
- Low code divergence
- Code as close to original as possible - keep Fortran

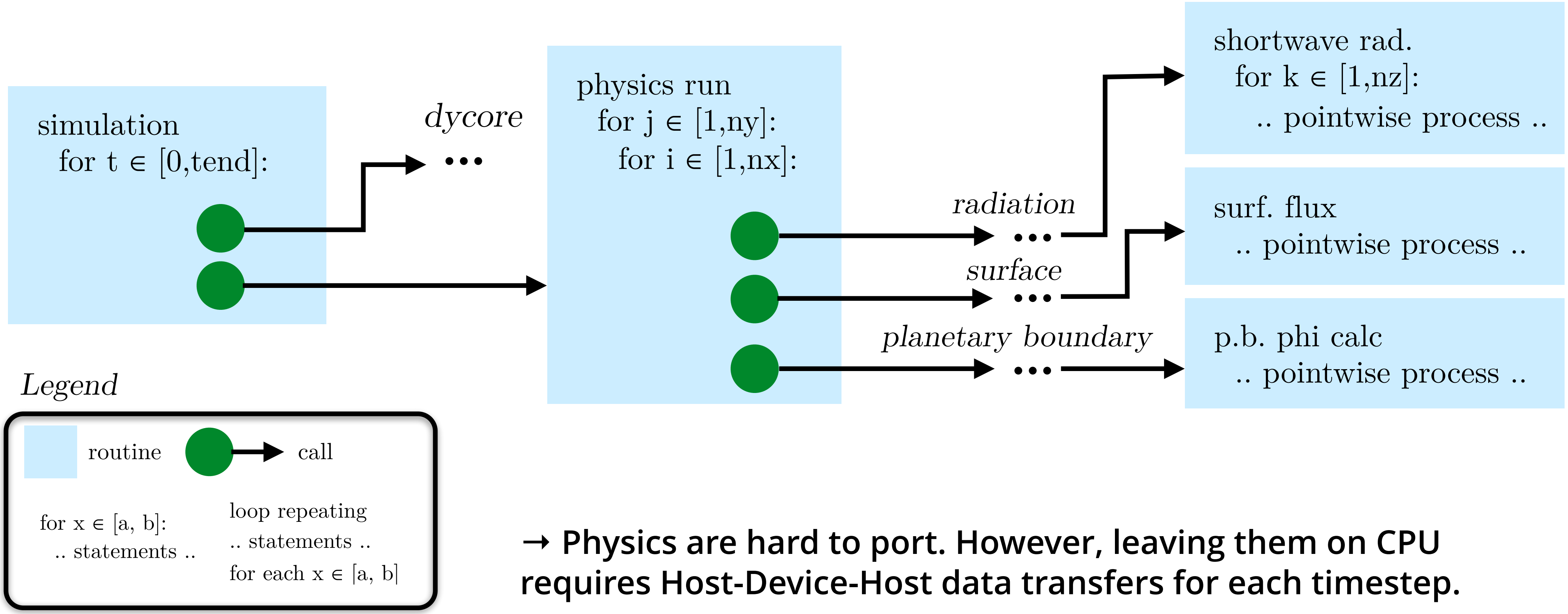
[6] Kawano K., Ishida J. and Muroi C.: “Development of a New Nonhydrostatic Model ASUCA at JMA”, 2010

ASUCA… another point of view

- 155k LOC
- 338 kernels
- one lonely Fortran GPU programmer



ASUCA



```
simulation
  for t ∈ [0,tend]:
```

dycore

•••

```
physics run
  for j ∈ [1,ny]:
    for i ∈ [1,nx]:
```

radiation •••

surface •••

planetary boundary •••

```
shortwave rad.
  for k ∈ [1,nz]:
    .. pointwise process ..
```

```
surf. flux
  .. pointwise process ..
```

```
p.b. phi calc
  .. pointwise process ..
```

Legend

routine

call

```
for x ∈ [a, b]:
  .. statements ..
```

loop repeating
.. statements ..
for each x ∈ [a, b]

→ **Physics are hard to port. However, leaving them on CPU requires Host-Device-Host data transfers for each timestep.**

Focus Problems

1. Code Granularity

2. Memory Layout

Focus Problems

1. Code Granularity

2. Memory Layout

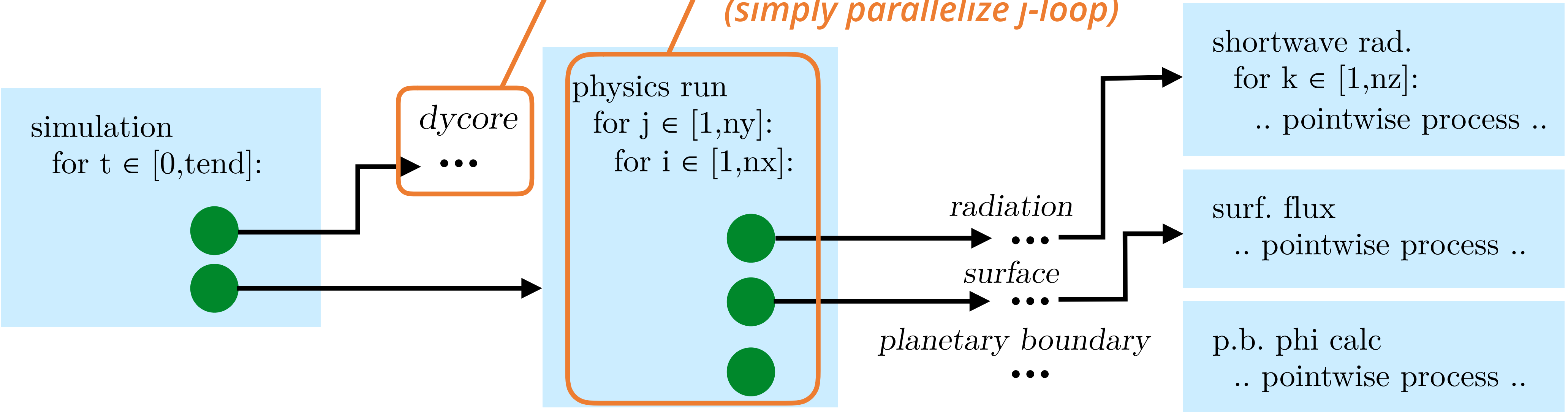
Definition of granularity:

The amount of work done by one thread.

For our purposes, we distinguish between two types of granularity:

a) ***runtime defined granularity***

b) ***code defined granularity***



fine code granularity
→*GPU friendly*

coarse code granularity
→*GPU unfriendly, performant on CPU*
(simply parallelize j-loop)

simulation
for t ∈ [0,tend]:

dycore
•••

physics run
for j ∈ [1,ny]:
for i ∈ [1,nx]:

radiation
•••

surface
•••

planetary boundary
•••

shortwave rad.
for k ∈ [1,nz]:
.. pointwise process ..

surf. flux
.. pointwise process ..

p.b. phi calc
.. pointwise process ..

Focus Problems

1. Code Granularity

Definition of granularity:

The amount of work done by one thread.

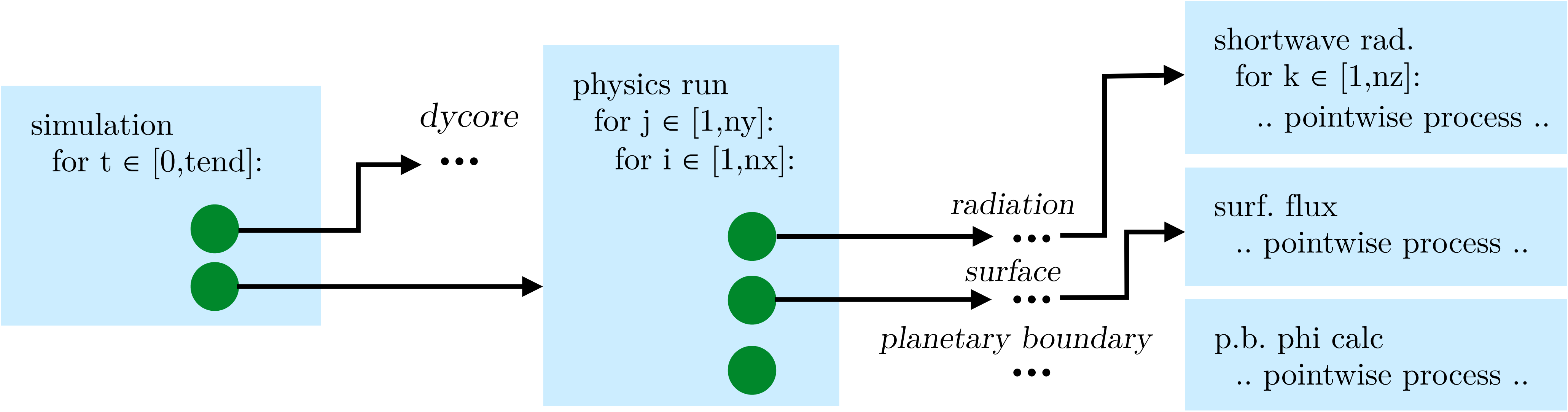
For our purposes, we distinguish between two types of granularity:

a) *runtime defined granularity*

b) *code defined granularity*

2. Memory Layout

- Regular grid → Fortran’s multi-dimensional arrays offer a simple to use and efficient data structure
- Performant layout on CPU: Keep fast varying vertical domain in cache → k-first
  Example stencil in original code:
  `A_out(k,i,j) = A(k,i,j) + A(k,i-1,j) …`
- GPU: Requires i-first or j-first for coalesced access



simulation
for t ∈ [0,tend]:

dycore
…

physics run
for j ∈ [1,ny]:
for i ∈ [1,nx]:

radiation
…

surface
…

planetary boundary
…

shortwave rad.
for k ∈ [1,nz]:
.. pointwise process ..

surf. flux
.. pointwise process ..

p.b. phi calc
.. pointwise process ..

OpenACC is not high level enough for this usecase.

What others* do …

1. Code Granularity

Kernel fusion in backend

- ➡ User needs to refine coarse kernels manually at first.
- ➡ Difficult to manage across functions and modules in a deep call-tree

2. Memory Layout

Stencil DSL abstraction in frontend

- ➡ Rewrite of point-wise code necessary

---

* [1] Shimokawabe T., Aoki T. and Onodera, N.: "High-productivity framework on GPU-rich supercomputers for operational weather prediction code ASUCA", 2014

[2] Fuhrer O. et al..: "Towards a performance portable, architecture agnostic implementation strategy for weather and climate models", 2014

… and what we propose

1. Code Granularity

Abstraction in frontend

- ◉ We assume the number of parallel loop constructs to be small (ASUCA: 200-300).

➡ Rewrite of these structures is manageable.

2. Memory Layout

Transformation in backend

- ◉ Manual rewrite of memory access patterns is time consuming and error-prone.

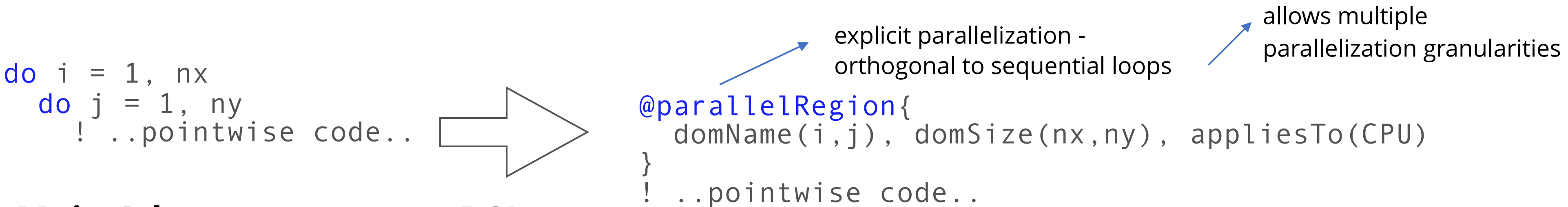
➡ We automate this process in backend.

In case of ASUCA:

1. Reordering of K-I-J to I-J-K
2. Due to granularity change for physics: Auto-privatization (I-J extension) of thread-local scalars and vertical arrays

→ Hybrid Fortan

- A language extension for Fortran
- A code transformation for targeting GPU and multi-core CPU parallelizations with the same codebase; Produces CUDA Fortran, OpenACC and OpenMP parallel versions in backend.
- Goal: Making GPU retargeting of existing Fortran code as productive as possible
- Idea: Combine strengths of DSLs and Directives



```
do i = 1, nx
  do j = 1, ny
    ! ..pointwise code..
```

```
@parallelRegion{
  domName(i,j), domSize(nx,ny), appliesTo(CPU)
}
! ..pointwise code..
```

explicit parallelization - orthogonal to sequential loops

allows multiple parallelization granularities

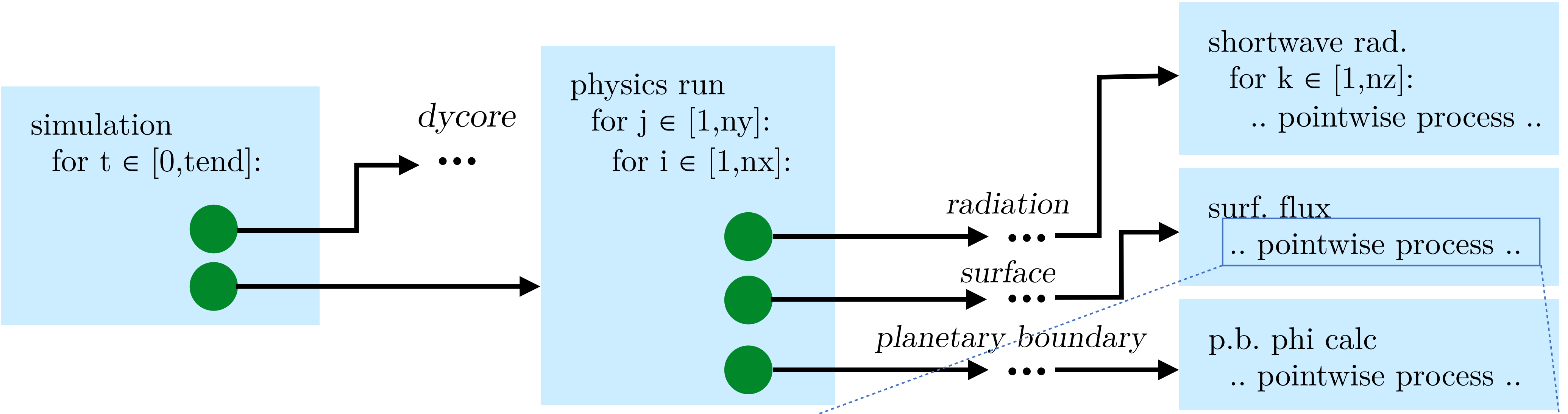
Main Advantages versus DSLs

- No change of programming language necessary
- Code with coarse parallelization granularity can easily be ported

Main Advantages versus Directives (e.g. OpenACC)

- Memory layout is abstracted —> Optimized layouts for GPUs and CPUs
- No rewrite and/or code duplication necessary for code with coarse parallelization granularity

Example



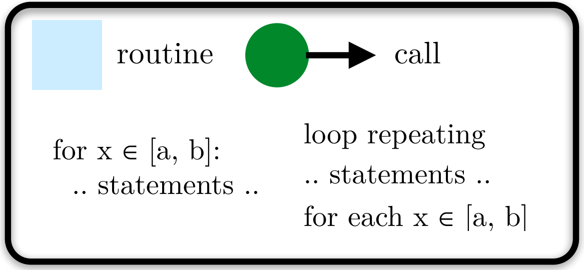
Example reference code from surface flux

Data parallelism not exposed at this layer of code —> coarse grained parallelization

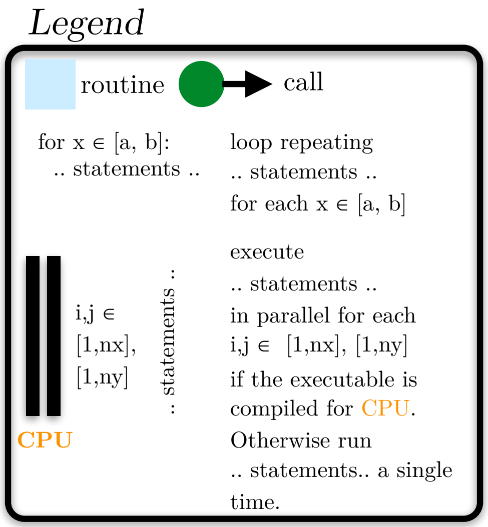
```
lt = tile_land
if (tlcvr(lt) > 0.0_r_size) then
  call sf_slab_flx_land_run(&
    ! ... inputs and further tile variables omitted
    & taux_tile_ex(lt), tauy_tile_ex(lt) &
    & )

  u_f(lt) = sqrt(sqrt(taux_tile_ex(lt) ** 2 + tauy_tile_ex(lt) ** 2))
else
  taux_tile_ex(lt) = 0.0_r_size
  tauy_tile_ex(lt) = 0.0_r_size
  ! ... further tile variables omitted
end if
```

Legend



Example using Hybrid Fortran



Legend

routine

call

for x ∈ [a, b]:
.. statements ..

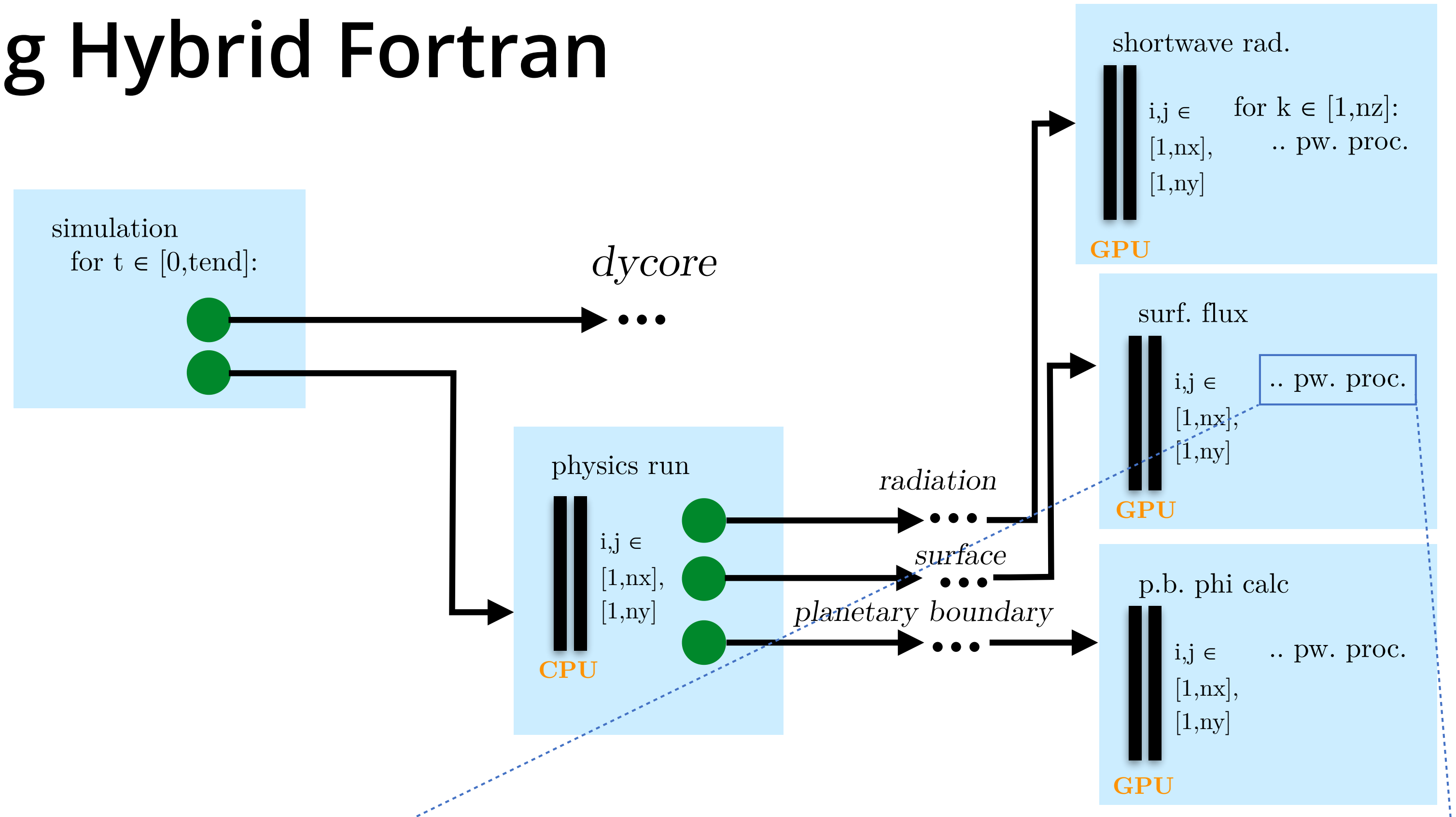
loop repeating
.. statements ..
for each x ∈ [a, b]

i,j ∈
[1,nx],
[1,ny]

.. statements ..

CPU

execute
.. statements ..
in parallel for each
i,j ∈ [1,nx], [1,ny]
if the executable is
compiled for CPU.
Otherwise run
.. statements.. a single
time.



simulation
for t ∈ [0,tend]:

dycore

physics run

i,j ∈
[1,nx],
[1,ny]

CPU

radiation

surface

planetary boundary

shortwave rad.

i,j ∈
[1,nx],
[1,ny]

for k ∈ [1,nz]:
.. pw. proc.

GPU

surf. flux

i,j ∈
[1,nx],
[1,ny]

.. pw. proc.

GPU

p.b. phi calc

i,j ∈
[1,nx],
[1,ny]

.. pw. proc.

GPU

example code from surface flux using Hybrid Fortran

Pointwise code can be reused as is - Hybrid Fortran rewrites this code automatically to apply fine grained parallelism by using the `appliesTo` clause and the global call graph.

```
@parallelRegion{appliesTo(GPU), domName(i,j), domSize(nx,ny)}
lt = tile_land
if (tlcvr(lt) > 0.0_r_size) then
  call sf_slab_flx_land_run(&
    ! ... inputs and further tile variables omitted
    & taux_tile_ex(lt), tauy_tile_ex(lt) &
    & )

  u_f(lt) = sqrt(sqrt(taux_tile_ex(lt) ** 2 + tauy_tile_ex(lt) ** 2))
else
  taux_tile_ex(lt) = 0.0_r_size
  tauy_tile_ex(lt) = 0.0_r_size
  ! ... further tile variables omitted
end if
! ... sea tiles code and variable summing omitted
@end parallelRegion
```

Example using Hybrid Fortran

surface flux example including data specifications

- `autoDom` —> **extend existing data domain specification with parallel domain given by** `@domainDependant` **directive**
- `present` —> **data is already present on device**

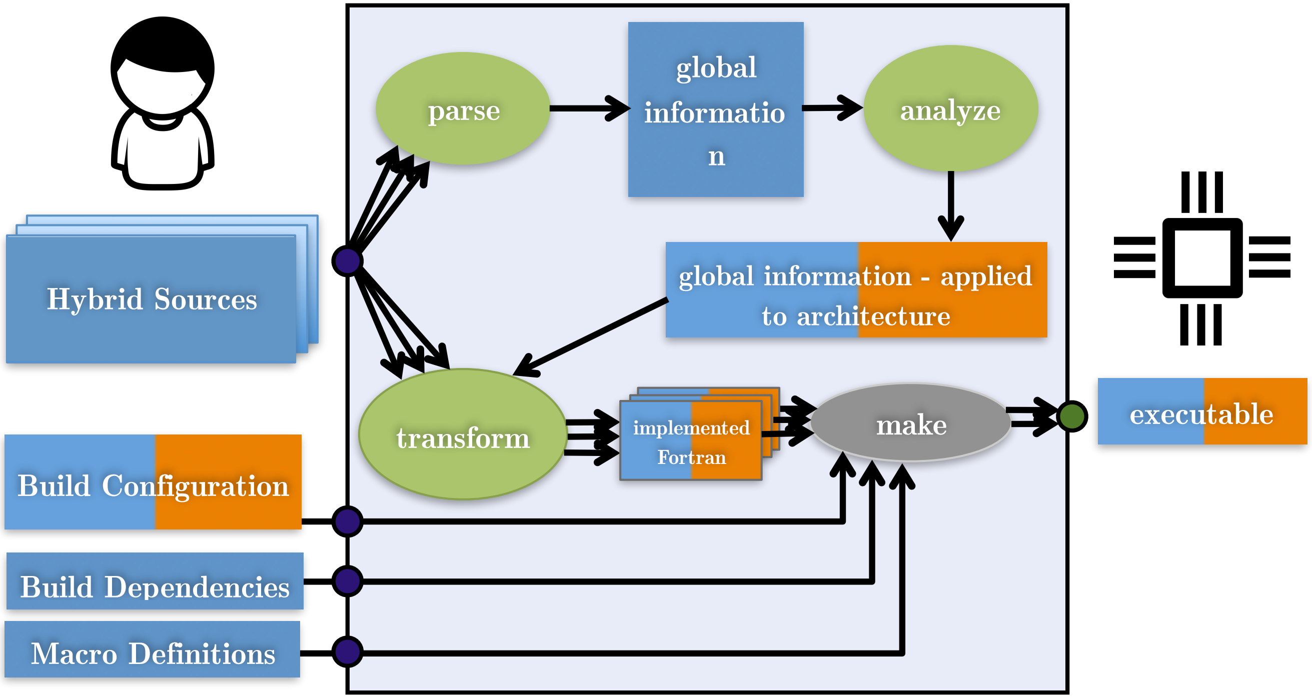
```
@domainDependant{domName(i,j), domSize(nx,ny), attribute(autoDom, present)}
tlcvr, taux_tile_ex, tauy_tile_ex, u_f
@end domainDependant

@parallelRegion{appliesTo(GPU), domName(i,j), domSize(nx,ny)}
lt = tile_land
if (tlcvr(lt) > 0.0_r_size) then
  call sf_slab_flx_land_run(&
    ! ... inputs and further tile variables omitted
    & taux_tile_ex(lt), tauy_tile_ex(lt) &
    & )

  u_f(lt) = sqrt(sqrt(taux_tile_ex(lt) ** 2 + tauy_tile_ex(lt) ** 2))
else
  taux_tile_ex(lt) = 0.0_r_size
  tauy_tile_ex(lt) = 0.0_r_size
  ! ... further tile variables omitted
end if
! ... sea tiles code and variable summing omitted
@end parallelRegion
```

Code Transformation Process

🟦 hybrid file
🟦🟧 file with CPU+GPU version
● python
● GNU Make
● output
● input
👤 user facing
machine facing
legend



1. Process macros in input
2. Sanitize input
   deleting whitespace & comments, merging continued lines
3. Parse global call graph (“parse”)
4. Apply user-defined target-specific parallelization granularity to call graph (“analyze”)
5. Parse module data specifications
6. Link module data spec. to routines where data is imported
7. Generate global application model
   intermediate representation, contains modules, routines and code regions, each linked with all relevant user code and meta information
8. Transform code for target architecture (“transform”)
   implementation class per routine with hooks called for each detected pattern that requires transformation
9. Sanitize output
   split lines that are too long for Fortran standard
10. Process macros in output
    implementation of memory layout

Analysis Step: CPU

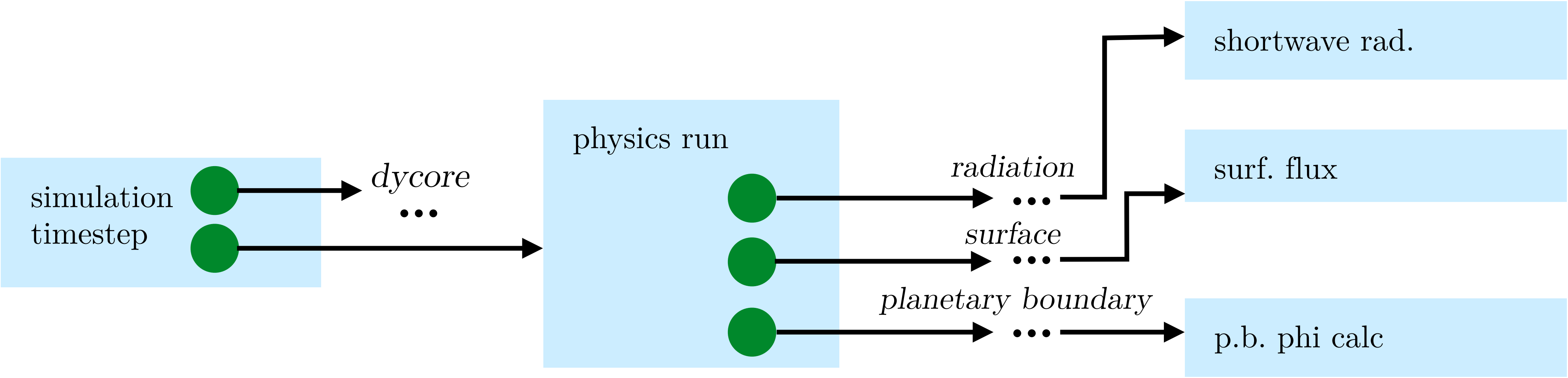
Legend

🟦 routine

● → call

for x ∈ [a, b]:
.. statements ..

loop repeating
.. statements ..
for each x ∈ [a, b]



simulation timestep

dycore •••

physics run

radiation •••

surface •••

planetary boundary •••

shortwave rad.

surf. flux

p.b. phi calc

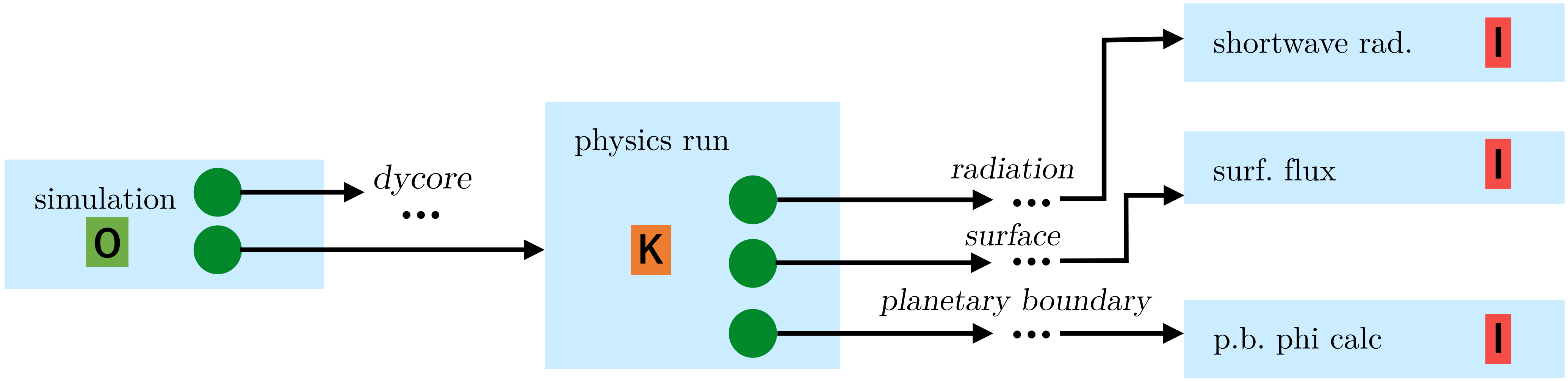
Analysis Step: CPU

Legend

- 🟦 routine
- ●→ call
- O "outside" —> routine calling kernel routines
- K "kernel" routines
- I "inside" —> routine called inside kernel

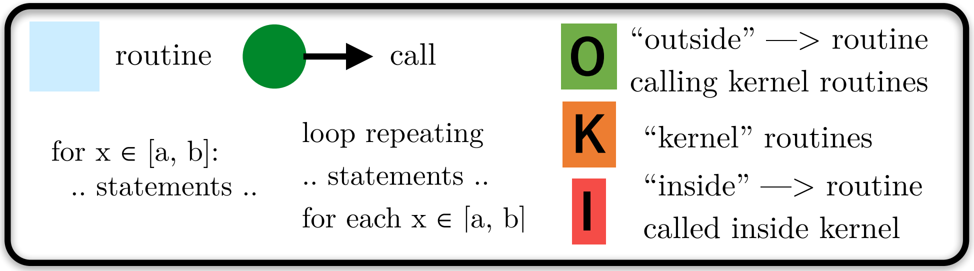
```
for x ∈ [a, b]:
    .. statements ..
```

loop repeating .. statements .. for each x ∈ [a, b]



Analysis Step: GPU

Legend



routine

call

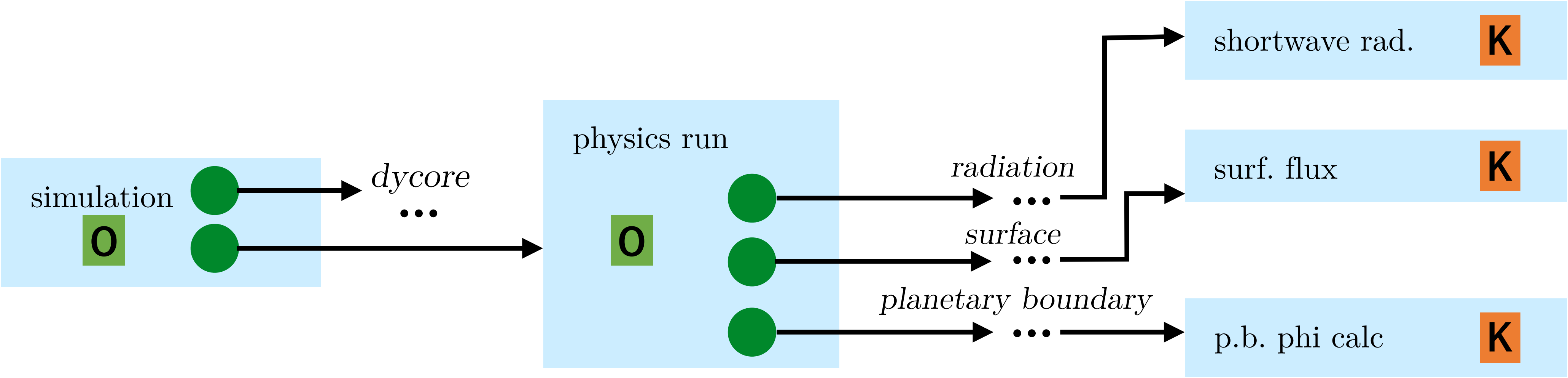
for x ∈ [a, b]:
.. statements ..

loop repeating
.. statements ..
for each x ∈ [a, b]

O “outside” —> routine calling kernel routines

K “kernel” routines

I “inside” —> routine called inside kernel



simulation **O**

dycore …

physics run **O**

radiation …

surface …

planetary boundary …

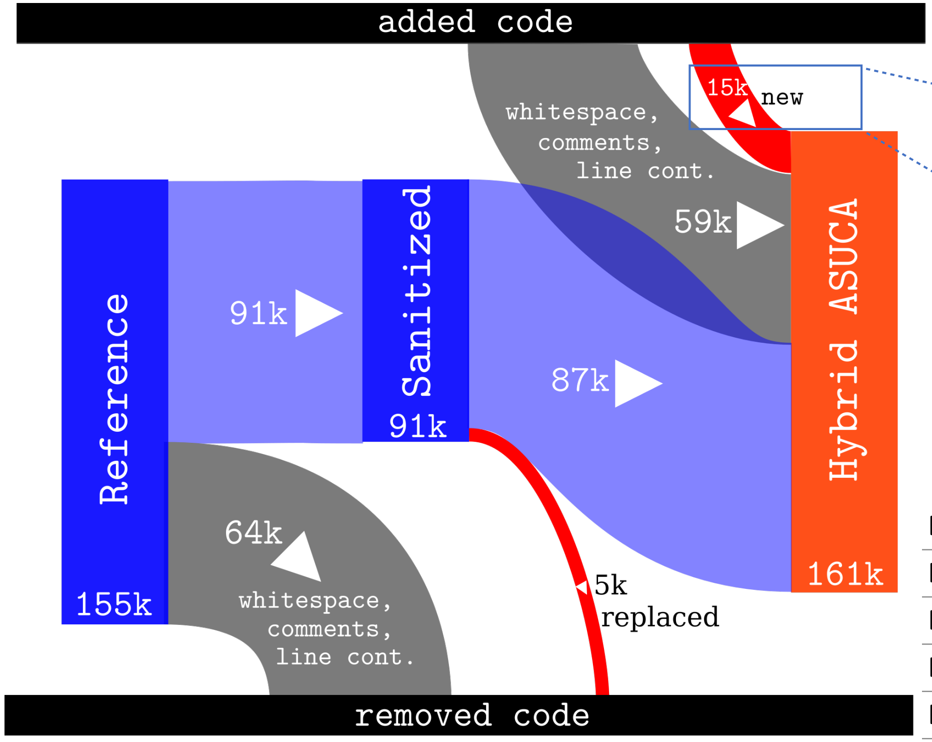
shortwave rad. **K**

surf. flux **K**

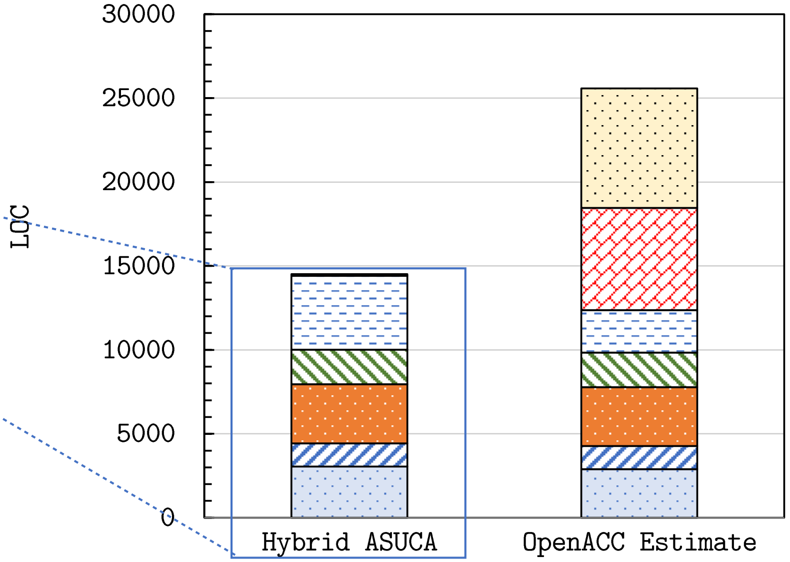
p.b. phi calc **K**

ASUCA: Productivity

Code Reuse and Changes



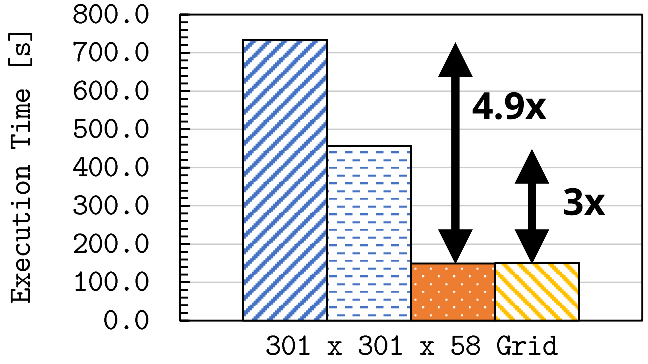
Comparison with OpenACC Estimate



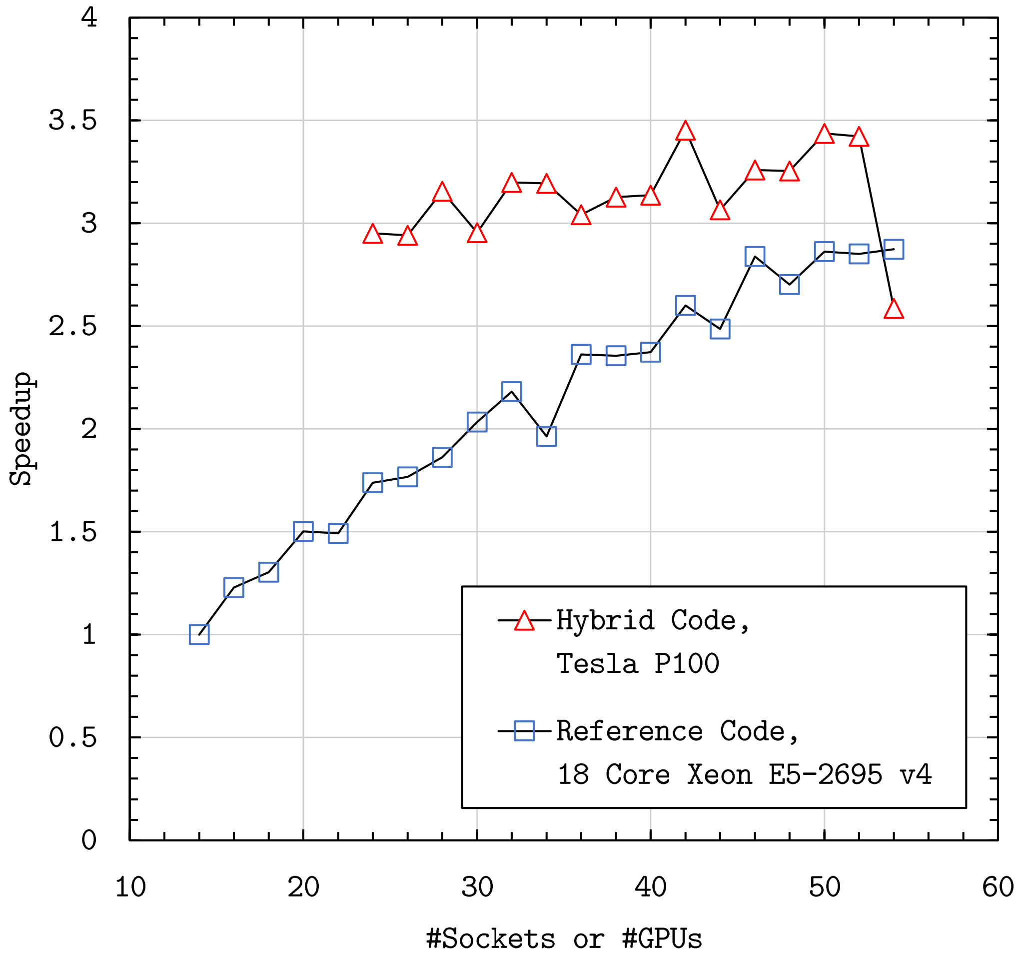
	Hybrid ASUCA	OpenACC Estimate
☐ CPU-only physics	0	7122
☒ storage order macros	116	6098
☐ parallelization & data layout DSL	4398	2521
▧ long-wave radiation	2059	2059
■ modified data spec./init	3519	3519
▨ routine & call signatures	1381	1381
☐ other	3046	2884

ASUCA: Performance

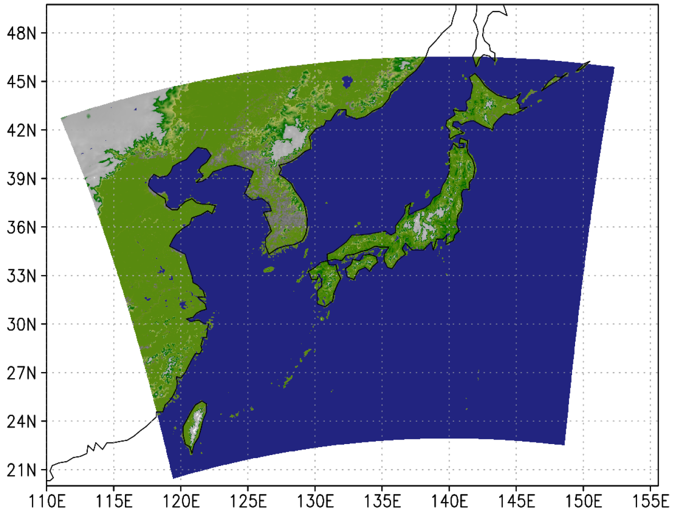
Kernel performance on reduced Grid (301 x 301 x 58) →



▨	ASUCA Reference, 4 x 6-core Xeon X5670	734.0
▤	ASUCA Reference, 1 x 18-core Xeon E5-2695 v4	456.7
🟧	Hybrid ASUCA, 4 x Tesla K20x	148.9
▧	Hybrid ASUCA, 1 x Tesla P100	151.1



← **Strong scaling results on Reedbush-H, 1581 x 1301 x 58 Grid (Japan and surrounding region)**

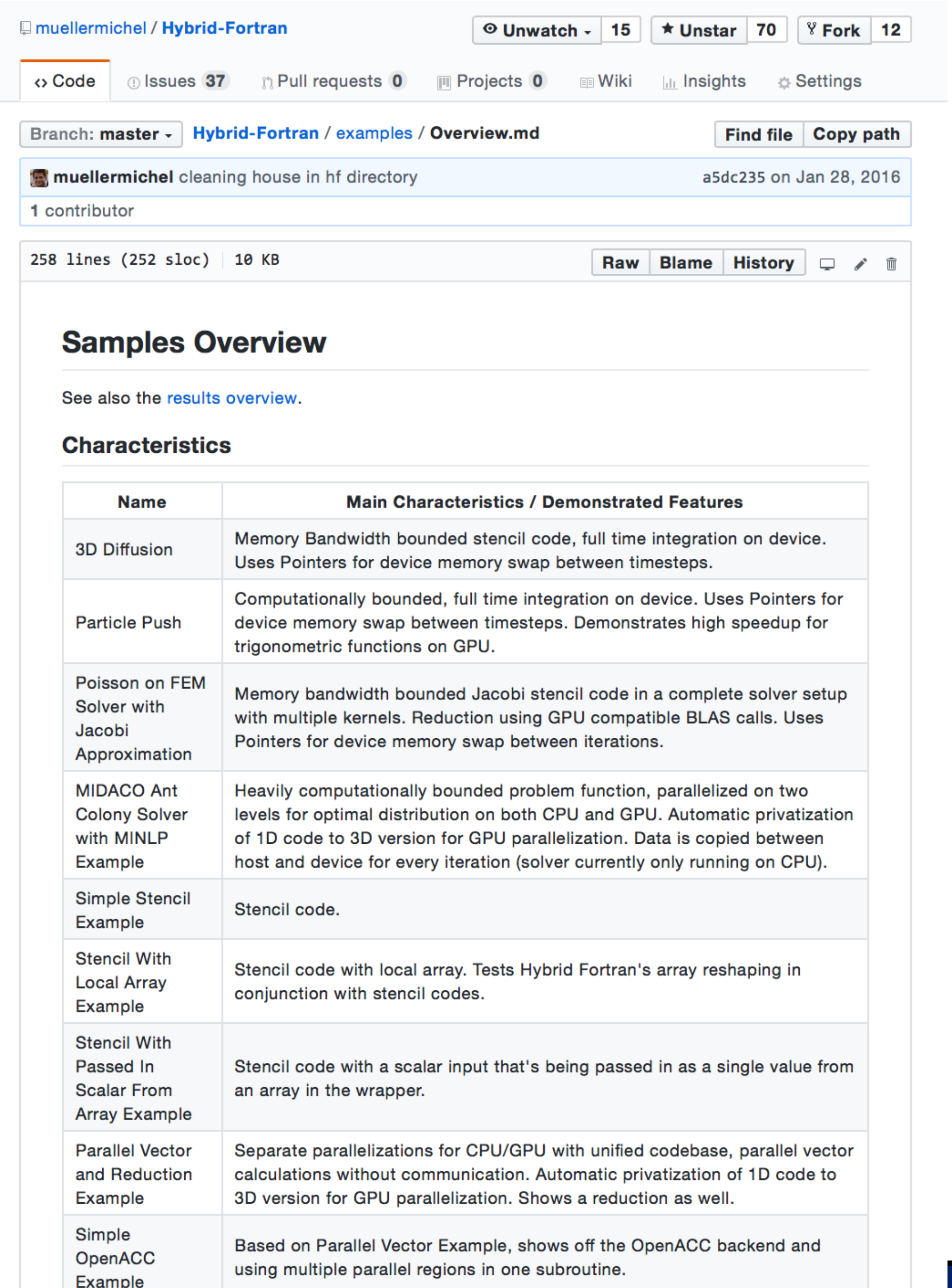


Hybrid Fortran on GitHub

21 Sample Codes

LGPL License

PDF Documentation



muellermichel cleaning house in hf directory a5dc235 on Jan 28, 2016

1 contributor

258 lines (252 sloc) 10 KB

Samples Overview

See also the results overview.

Characteristics

Name	Main Characteristics / Demonstrated Features
3D Diffusion	Memory Bandwidth bounded stencil code, full time integration on device. Uses Pointers for device memory swap between timesteps.
Particle Push	Computationally bounded, full time integration on device. Uses Pointers for device memory swap between timesteps. Demonstrates high speedup for trigonometric functions on GPU.
Poisson on FEM Solver with Jacobi Approximation	Memory bandwidth bounded Jacobi stencil code in a complete solver setup with multiple kernels. Reduction using GPU compatible BLAS calls. Uses Pointers for device memory swap between iterations.
MIDACO Ant Colony Solver with MINLP Example	Heavily computationally bounded problem function, parallelized on two levels for optimal distribution on both CPU and GPU. Automatic privatization of 1D code to 3D version for GPU parallelization. Data is copied between host and device for every iteration (solver currently only running on CPU).
Simple Stencil Example	Stencil code.
Stencil With Local Array Example	Stencil code with local array. Tests Hybrid Fortran's array reshaping in conjunction with stencil codes.
Stencil With Passed In Scalar From Array Example	Stencil code with a scalar input that's being passed in as a single value from an array in the wrapper.
Parallel Vector and Reduction Example	Separate parallelizations for CPU/GPU with unified codebase, parallel vector calculations without communication. Automatic privatization of 1D code to 3D version for GPU parallelization. Shows a reduction as well.
Simple OpenACC Example	Based on Parallel Vector Example, shows off the OpenACC backend and using multiple parallel regions in one subroutine.

Conclusions

- A performant GPU port for a meso-scale weather prediction model has been achieved (physics + dynamics).
- Using Hybrid Fortran, 85% of the ported code is a direct copy of the original - without counting whitespace, comments and line continuations.
- Overall, the code size has grown by less than 4%.
- A library of code examples has been constructed and Open Sourced together with Hybrid Fortran.