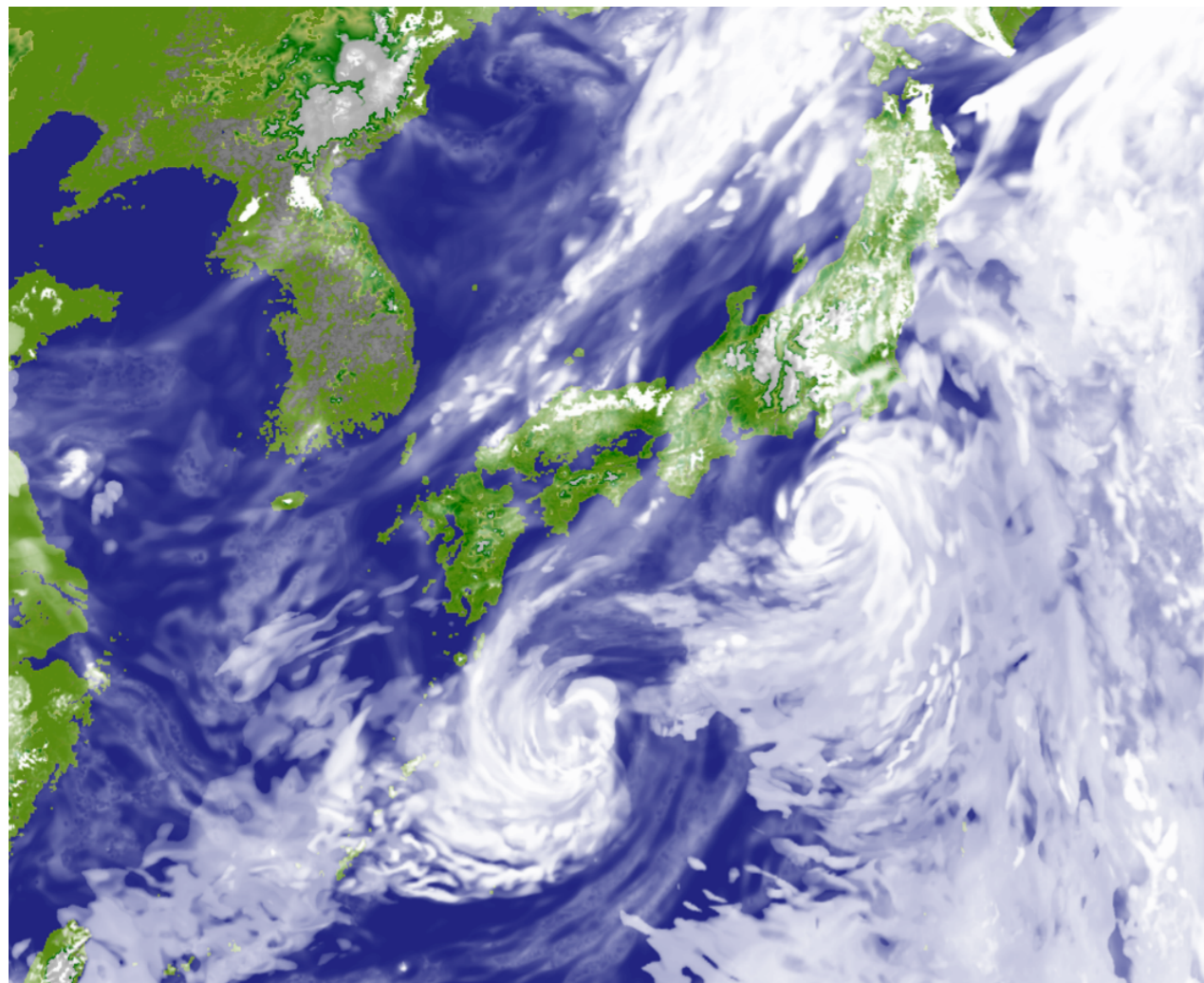

Hybrid Fortran

High Performance & Productivity for GPU Numerics

Talk



Michel Müller

Postdoctoral Researcher ETH Zurich
Dr. Eng., Tokyo Institute of Technology

2018-11-01

Outline

1. Introduction
2. Method
3. Application
4. Performance
5. Conclusion

NWP and Computational Performance

- Increase in computational performance allows increasing grid resolution.
 - During last decade this allows resolution of increasingly small cloud formations in dynamical core.
 - Typically applied finite-volume and finite-difference based discretization methods are bottlenecked by memory bandwidth in the dynamics.
- ➔ Hardware architectures with high memory bandwidth are sought.

... however Dennard scaling does not.

Dennard scaling: Power density of micro transistors proportional to area.

→ Clock frequency/single threaded perf. scales inverse proportionally to transistor size

Since 90nm process technology (~2004-2005), Dennard scaling does not hold anymore.

Leakage currents increasingly limit advancements in single threaded performance.

Latency- versus Throughput Oriented Processing

Latency: Time elapsed between initiation and completion of a task.

Throughput: Total amount of work completed per unit time.

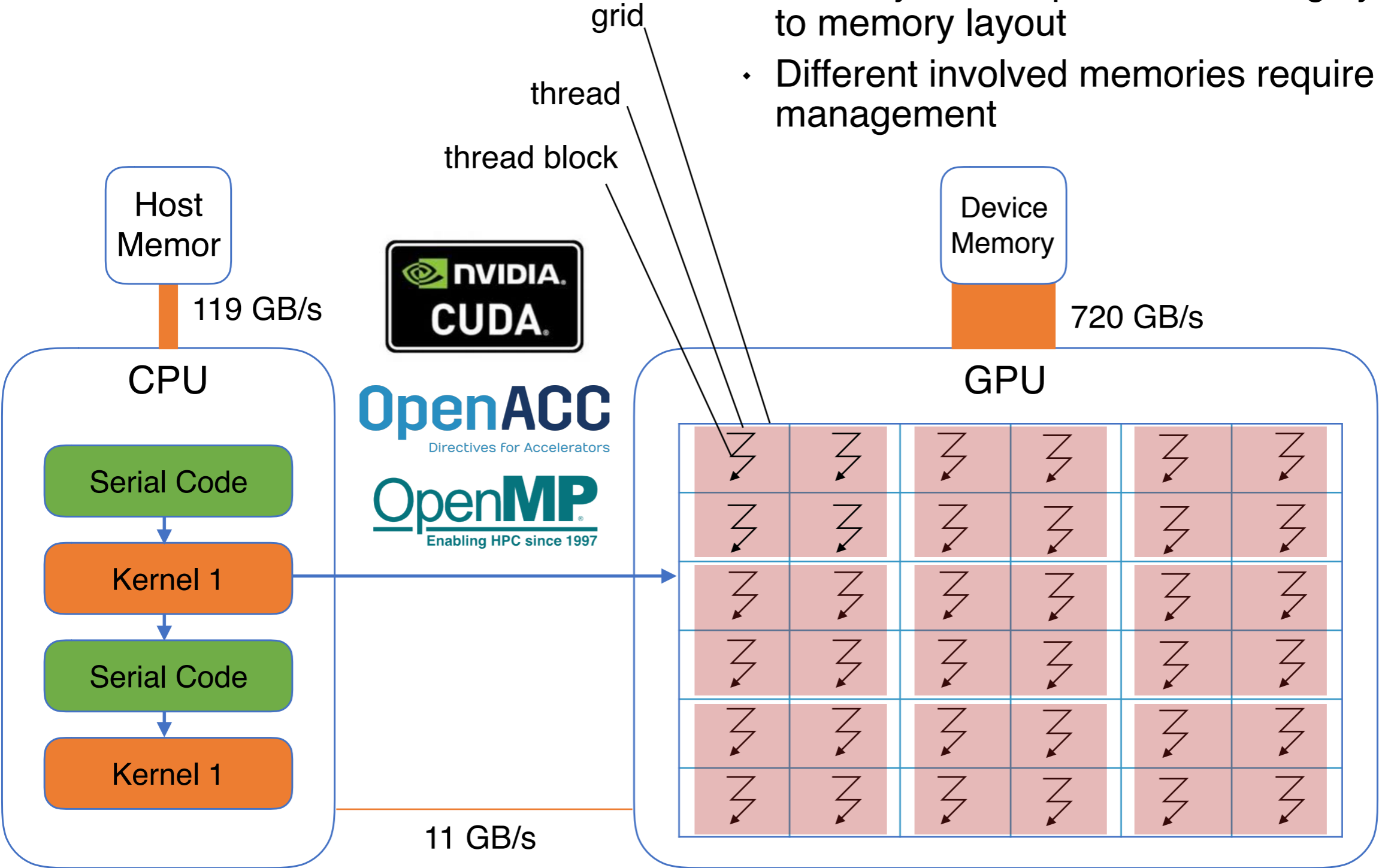
- Due to end of Dennard scaling:
 - ➔ shift from latency-oriented processor design to throughput-oriented
 - ➔ applications only profit when adapted accordingly

GPU Computing

- Graphics Processing Units (GPUs) are a popular type of throughput-oriented processors.
- Today has many applications outside of graphics.
- Applications need to be highly parallelizeable, as GPUs have a high latency to complete a single task compared to CPUs.

GPU Computing

- High memory bandwidth
- Support for branching, 64bit FP
- Fine-grained parallelism
- Memory access performance highly sensitive to memory layout
- Different involved memories require management



ASUCA NWP Model

What is ASUCA?

- ``Asuca is a System based on a Unified Concept for Atmosphere''
- fully compressible, non-hydrostatic weather prediction model
- regional scale - as depicted in Figure 1.2
- one of main operational forecast models in Japan, in production since 2014
- spatial discretization: finite-volume method on Arakawa-C-type rectangular grid
- time discretization:
 - third-order Runge-Kutta based iteration scheme for advection and Coriolis force
 - time-splitting method, employing secondary third-order Runge-Kutta iteration with short time step for sound- and gravity waves
- vertical-only models for parametrization of radiation, planetary boundary layer and surface physical processes

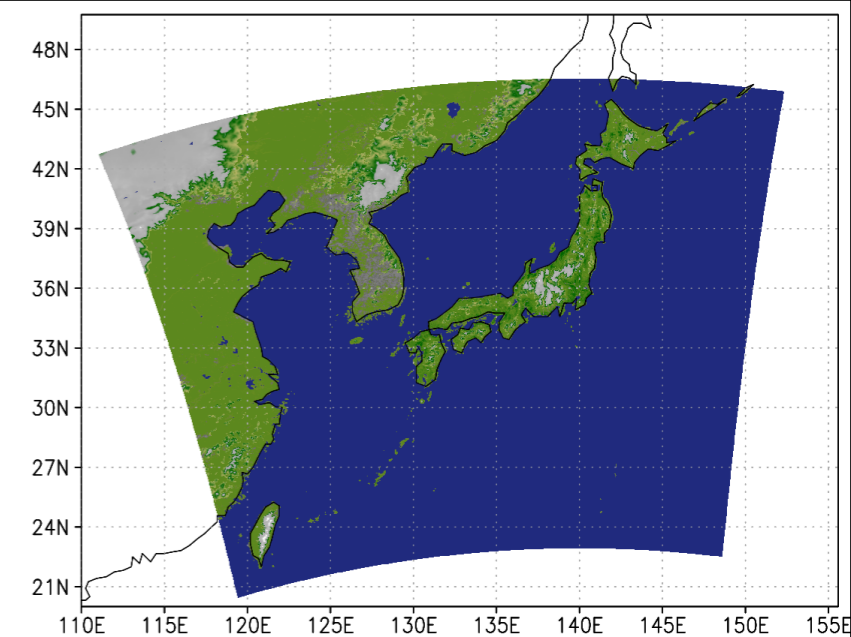


Figure 1.2: ASUCA's model simulation boundaries

GPUs for Numerical Weather Prediction

- GPUs offer high memory bandwidth, which is in high demand in NWP.
 - GPUs are an attractive target architecture.
- Major problems to solve for existing regular grid NWP codes:
 - Memory layout needs change
 - Code granularity in physical processes too coarse for GPU
- Existing methods to solve these problems:
 - Only apply GPU to dynamical core.
 - Rewrite Fortran code using C++ templates for architecture specialization.
 - Code divergence between CPU and GPU to solve granularity issues.
- Unsatisfactory to maintain a unified, coherent and efficient code base in Fortran (the standard in NWP)
- For ASUCA, a solution with none of these drawbacks was sought.

Background

- ✓ paradigm shift towards throughput oriented design
- ✓ GPUs attractive for NWP (high mem. bandwidth)
- ✓ productivity and maintainability of GPU approaches lacking

Motivation

- ✓ Many of today's NWP- and climate models cannot make efficient use of high-throughput architectures. We want to find and prove easily adoptable approach.

Goal

- ✓ GPU port for "ASUCA" NWP model in Fortran with minimal code divergence / minimal learning

Contributions

- new granularity abstraction and memory layout transformation method
- applied to ASUCA, resulting in >3x speedup in kernel performance and >2x reduction in processors required for a full scale run with real data
- method unique in increasing productivity for porting coarse-grained codes to GPU

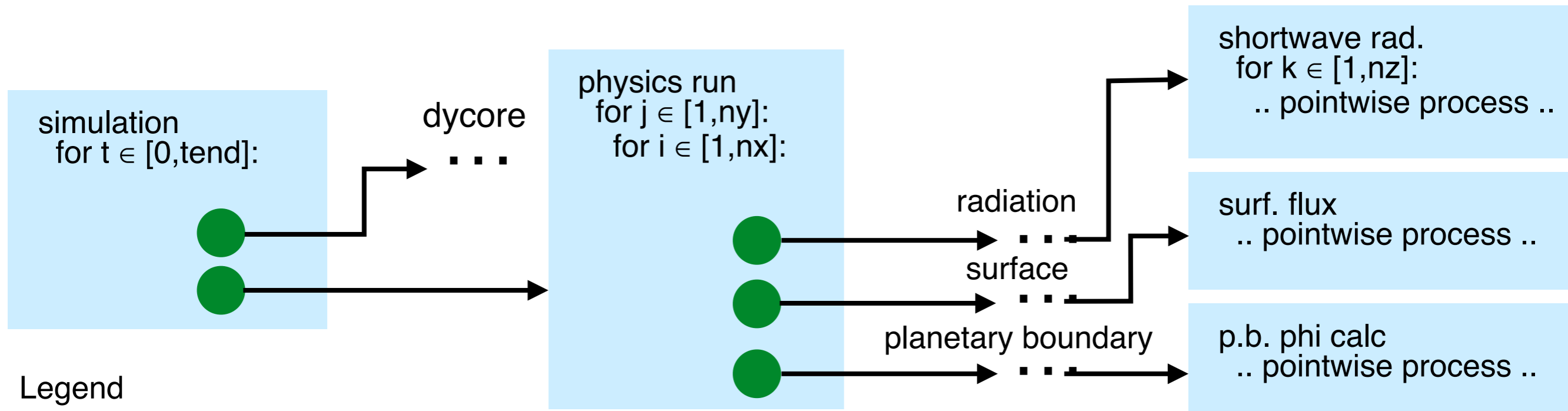
2. Method

- Granularity Abstraction
- Memory Layout & Regions
- Code Transformation

Assumptions for Design

- Mainly used data structure is Fortran arrays of different dimensions and data types.
- Kernels are data parallel.
- Existing inter-node / inter-GPU communication code can be reused.

ASUCA Code Structure



Legend

routine
 → call

for x ∈ [a, b]:
 .. statements ..

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

- ➔ Physics difficult to port
- ◆ Applying GPU only to dynamical core requires expensive host-device-communication for every timestep

Key Problems

1. Code Granularity

Definition of granularity:

The amount of work done by one thread.

fine-grained: low amount of work per thread

coarse-grained: high amount of work per thread

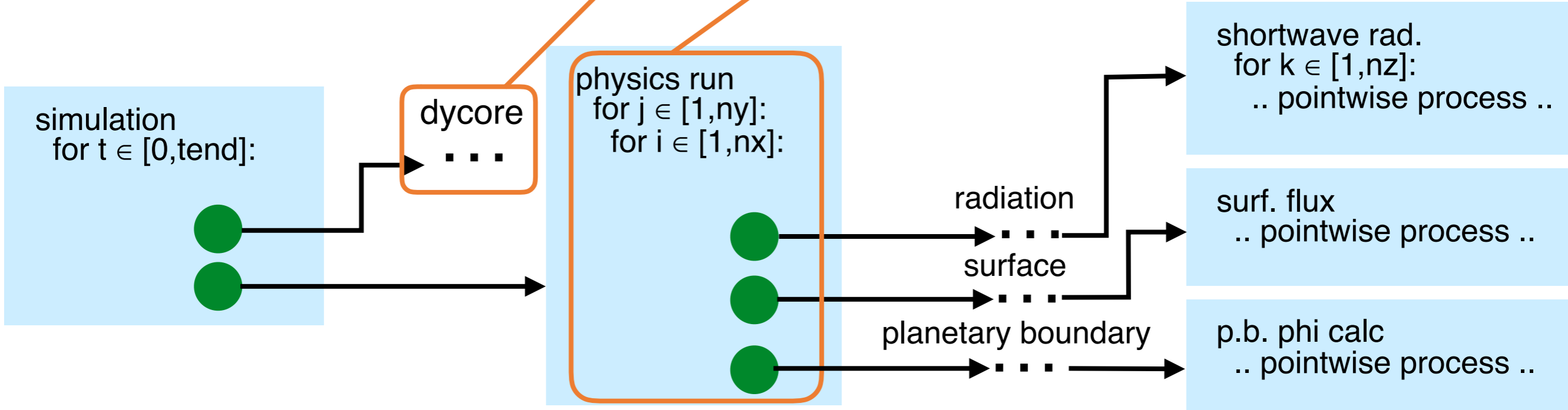
Two types of granularity:

a) runtime defined

b) code defined

fine code granularity
→ GPU friendly

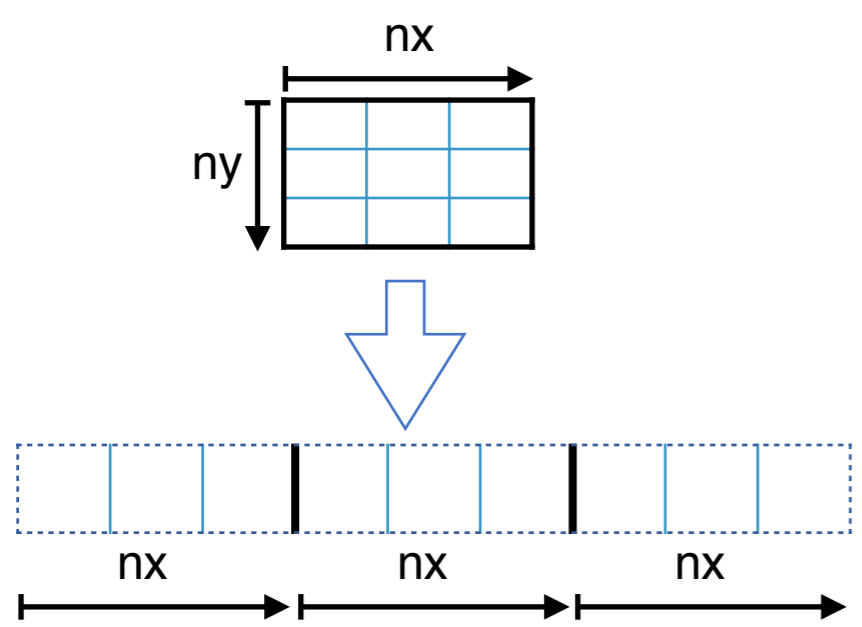
coarse code granularity
→ GPU unfriendly, performant on CPU



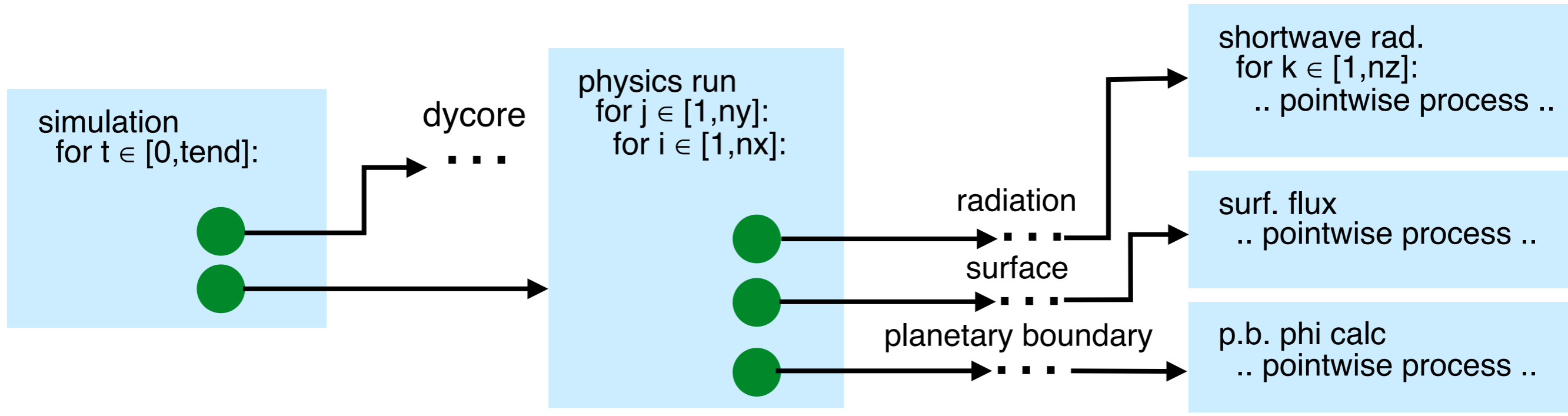
2. Method

Key Problems

2. Memory Layout



- Performant layout on CPU: Keep fast varying vertical domain in cache → k-first
 Example stencil in original code:
 $A_out(k,i,j) = A(k,i,j) + A(k,i-1,j) \dots$
- GPU: Requires i-first or j-first for coalesced access



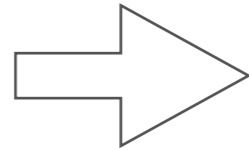
Hybrid Fortran

Main ideas:

- Allow efficient many-core GPU port while maintaining multi-core CPU compatibility
- Delegate parallelization boilerplate to framework
- Allow multiple parallelization granularities for the same code
- Transform memory layout for each target architecture

Parallelization & Granularity Abstraction

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



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

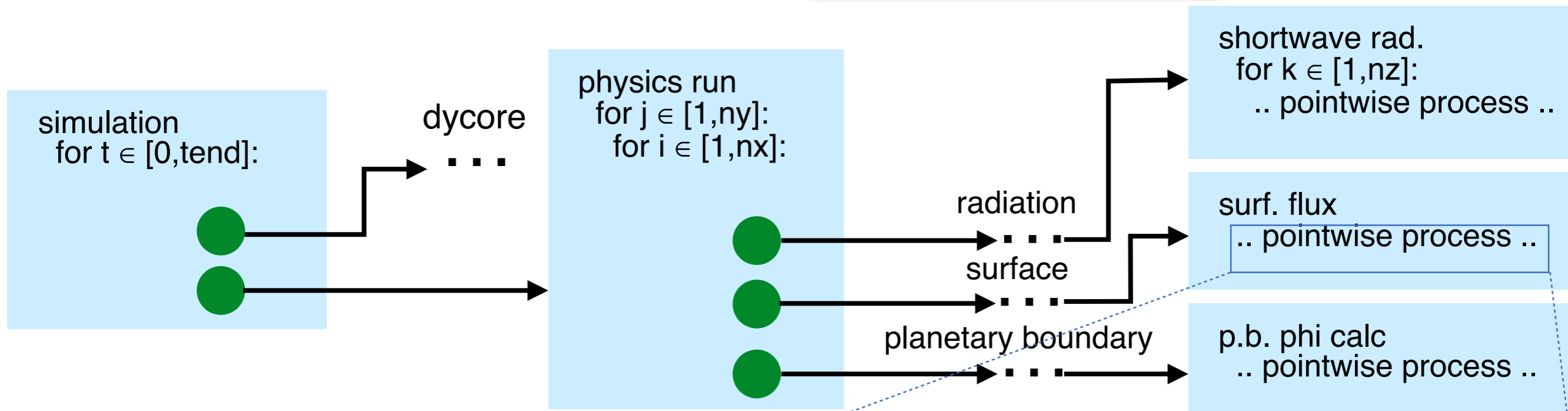
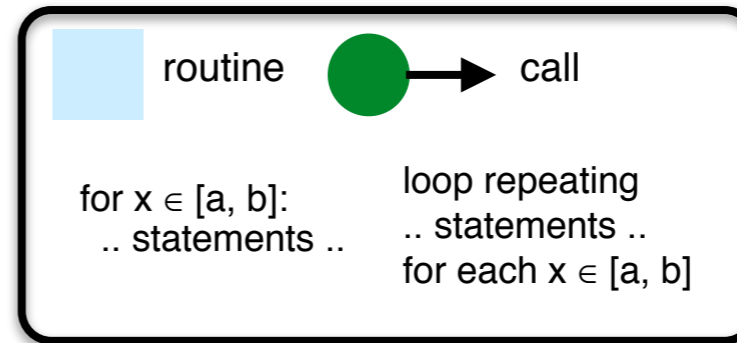
explicit parallelization -
orthogonal to
sequential loops

allows multiple
parallelization
granularities

Creates CUDA Fortran, OpenACC or CPU
multicore-OpenMP based parallelization,
depending on backend.

Example Physical Process

Legend



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_flux_land_run(&
    ! ... inputs and further tile variables omitted
    & tau_x_tile_ex(lt), tau_y_tile_ex(lt) &
    & )

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


Data Specifications

Data specifications:

- autoDom: extend existing data domain specification with parallel domain given by @domainDependant directive.
 - domName, domSize attributes specify horizontal extension of data domain
- present: data is already present on device.
 - requires counterpart specification at data region boundaries with transferHere attribute

```
@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
```

Transformed Code

Example surface flux kernel transformed with OpenACC backend.

```
!$acc kernels deviceptr(taux_tile_ex) deviceptr(tauy_tile_ex) &
!$acc& deviceptr(tlcvr) deviceptr(u_f)
!$acc loop independent vector(CUDA_BLOCKSIZE_Y)
outerParallelLoop0: do j=1,ny
!$acc loop independent vector(CUDA_BLOCKSIZE_X)
  do i=1,nx
    ! *** loop body *** :
    lt = tile_land
    if (tlcvr( AT(i,j,lt) )> 0.0_r_size) then
      call sf_slab_flx_land_run(&
        ! ... inputs and further tile variables omitted
        & taux_tile_ex( AT(i,j,lt) ), tauy_tile_ex( AT(i,j,lt) )
        &
        & )
      u_f( AT(i,j,lt) )= sqrt(sqrt(taux_tile_ex( AT(i,j,lt) )** 2 + &
        & tauy_tile_ex( AT(i,j,lt) )** 2))
    else
      taux_tile_ex( AT(i,j,lt) )= 0.0_r_size
      tauy_tile_ex( AT(i,j,lt) )= 0.0_r_size
      ! ... further tile variables omitted
    end if
    ! ... sea tiles code and variable summing omitted
  end do
end do outerParallelLoop0
!$acc end kernels
```

OpenACC parallelization

data specifications

block size macros

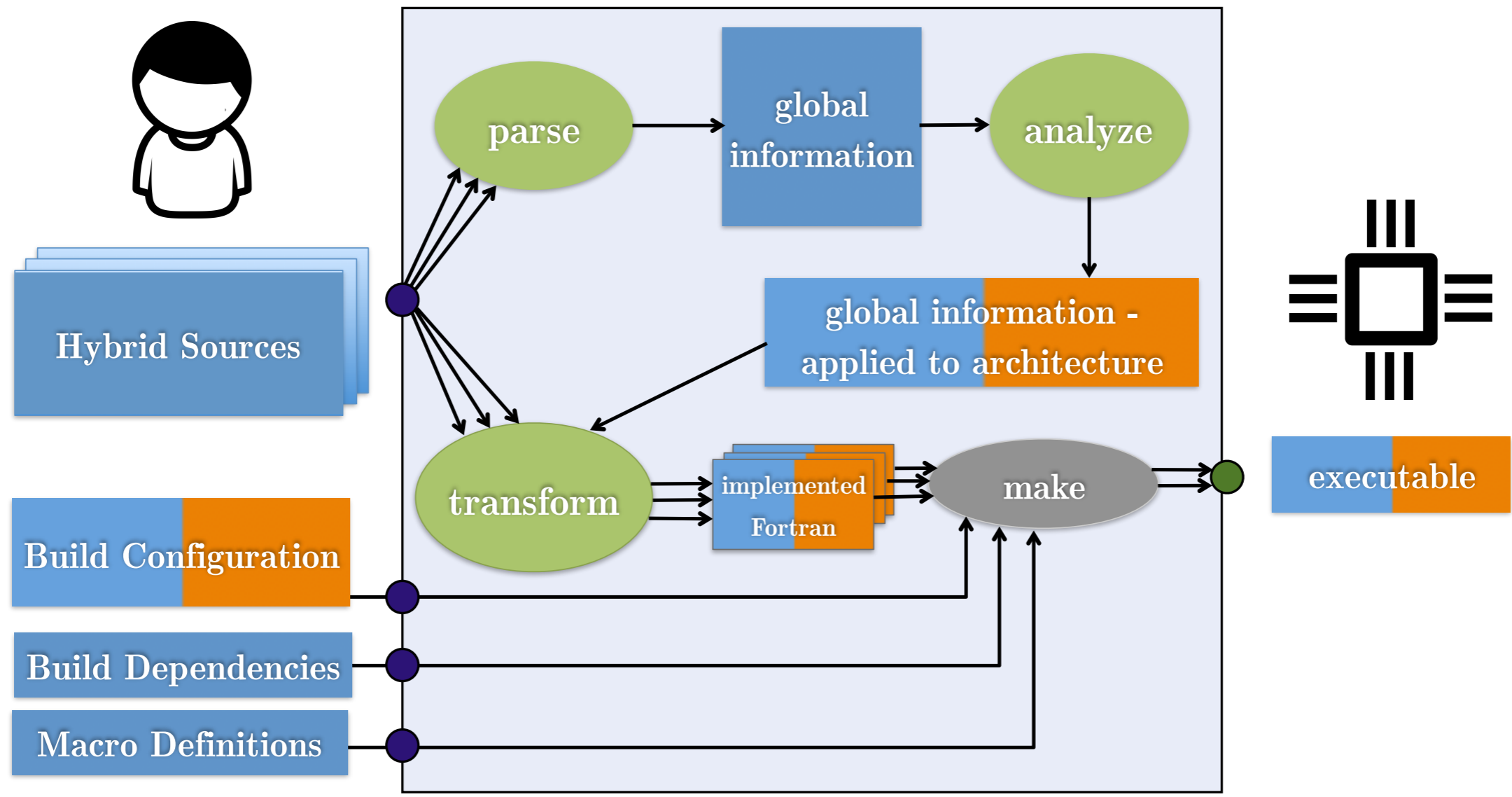
parallel loops

horizontal domain extension of data








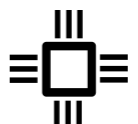
storage ordering macro

line breaks

Transformation Process

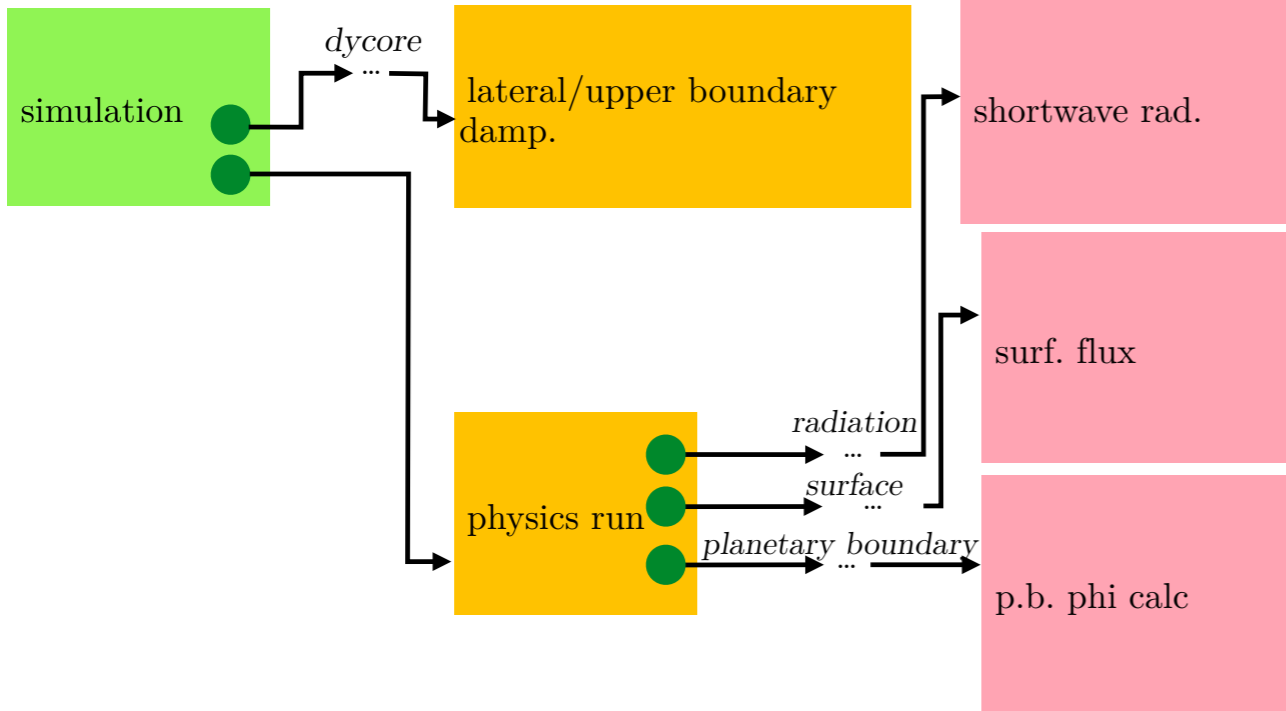


legend

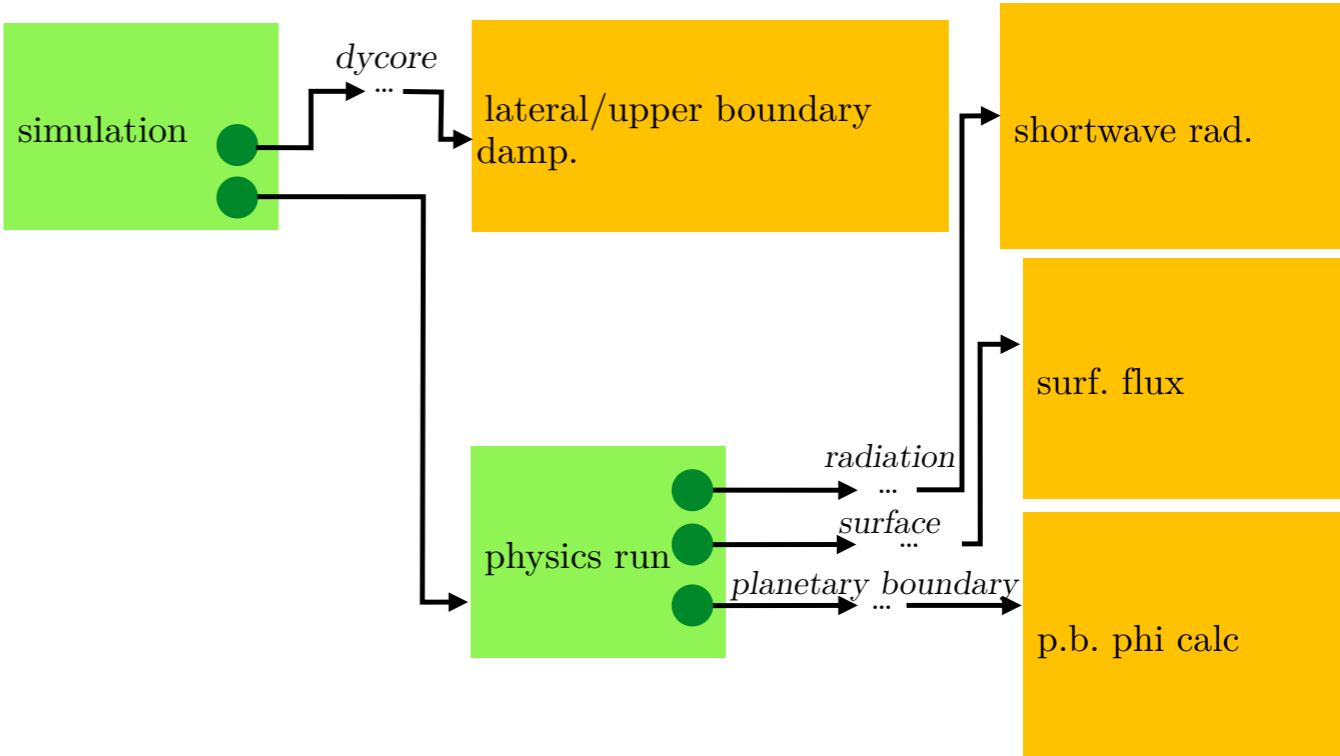
 hybrid file	 file with CPU+GPU version	 python	 output	 user facing
		 GNU Make	 input	 machine facing

Callgraph Analysis

CPU Version



GPU Version



Legend

	routine outside		routine inside
	routine with region		routine inside
	region		region
	call		call

Limitations

- code for programmable caches on GPU (“shared memory”, “texture memory”) is not generated by tool.
- relies on standard subroutines, e.g. Fortran function construct not supported for code running on GPU.

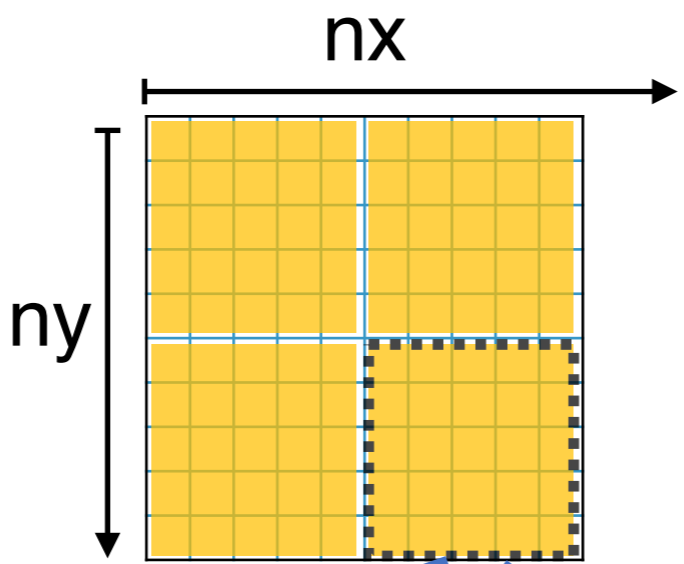
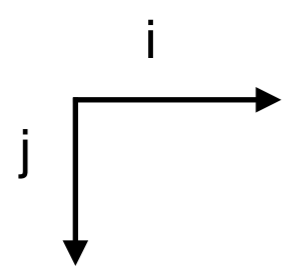
Contributions

- ✓ new granularity abstraction and memory layout transformation method
- **applied to ASUCA**, resulting in >3x speedup in kernel performance and >2x reduction in processors required for a full scale run with real data
- method unique in increasing productivity for porting coarse-grained codes to GPU

3. Application

- ◆ **Hybrid ASUCA Implementation**
- ◆ **Productivity Results**

Parallelization

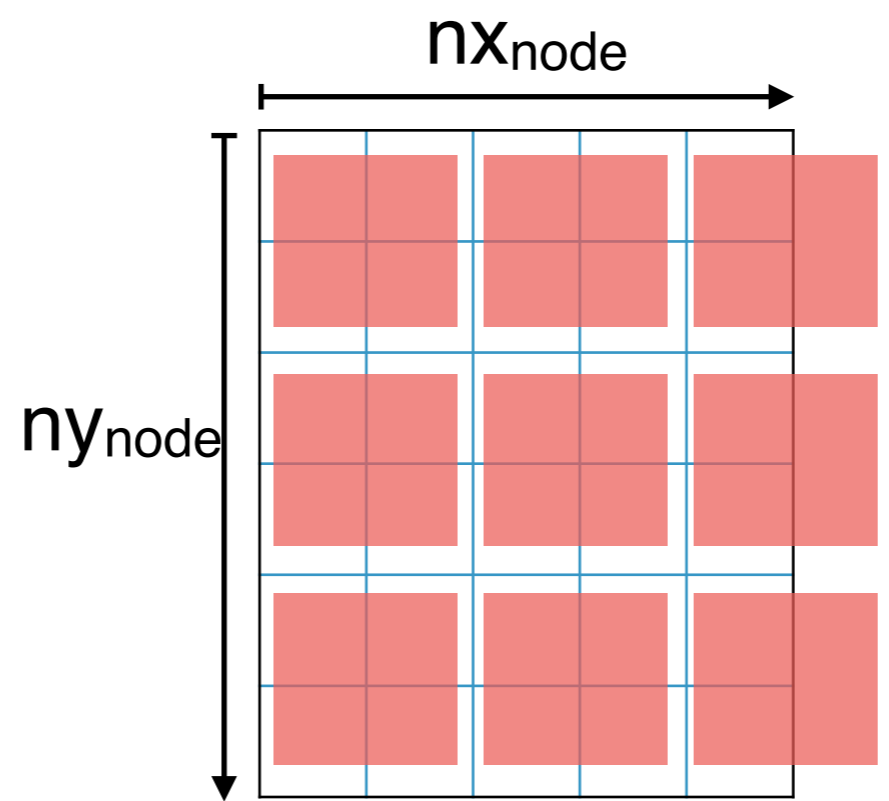
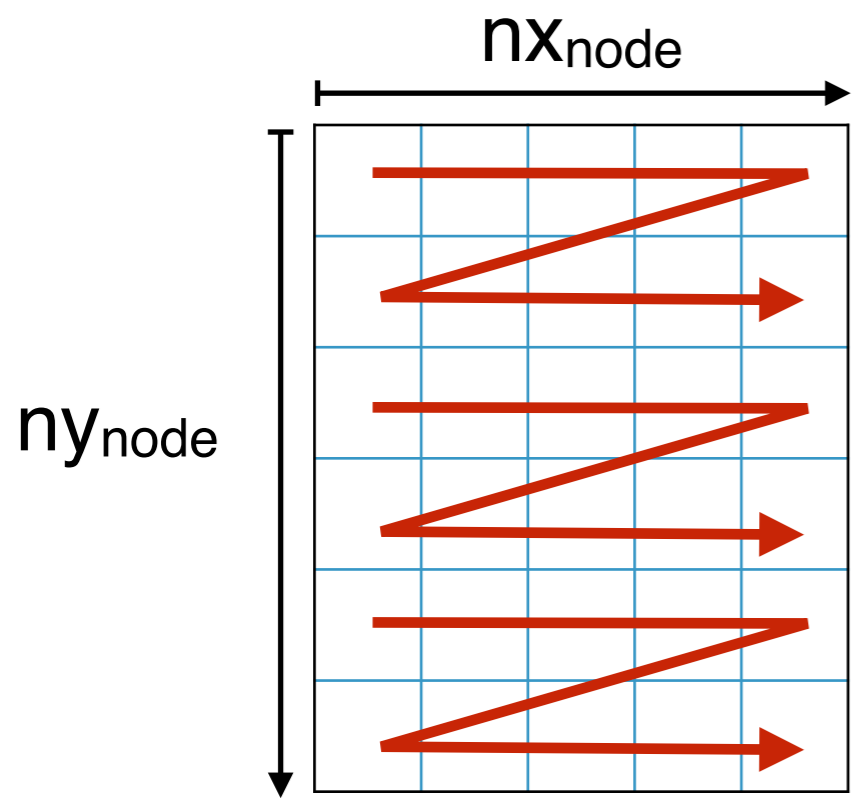


Legend

- grid
- MPI rank domain
- CPU thread & marching direction
- GPU thread block

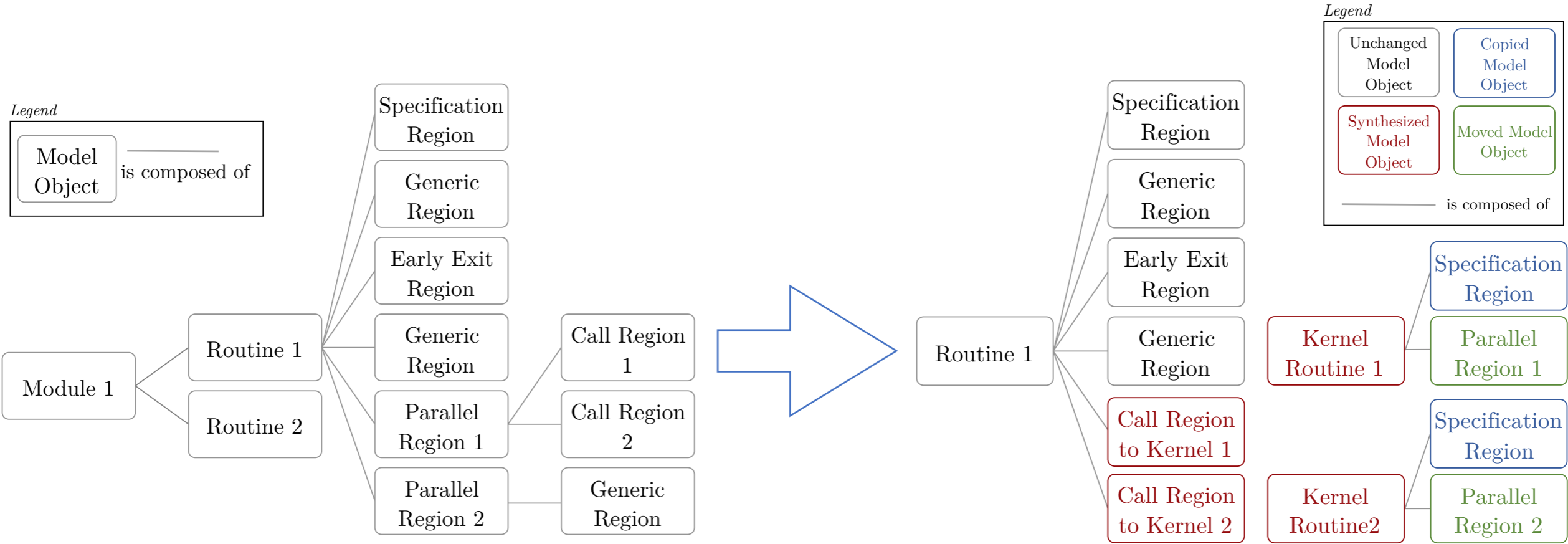
CPU

GPU



Dynamical Core

- ASUCA’s dynamical core contains many “tight parallel loops”, i.e. fine-grained parallelism.
- CUDA Fortran compiler was most stable during development.
 - ➔ Chosen as main backend.
 - ➔ Transformed code must have separate routines per kernel.
- ➔ To facilitate tight parallel loops, Hybrid Fortran employs routine splitting.
 - ➔ Existing code becomes compatible with CUDA Fortran backend.



Physical Processes

- Original physical process library from JMA adapted for GPU (MSM0705 model) provides column-wise models for:
 - Radiation, (solar, optical cloud absorption, atmospheric reflection and absorption)
 - For efficient use of GPU, memory footprint of indirect radiation effects was reduced by 10x by using ad-hoc computations for each long-wave band rather than storing temporary data of all bands.
 - Planetary boundary layer model
 - Wind momentum-, sensible heat- and latent heat surface fluxes
- Kessler-type warm rain model
- Hybrid Fortran's adaptive parallelization granularity used to generate GPU version

physical process kernel

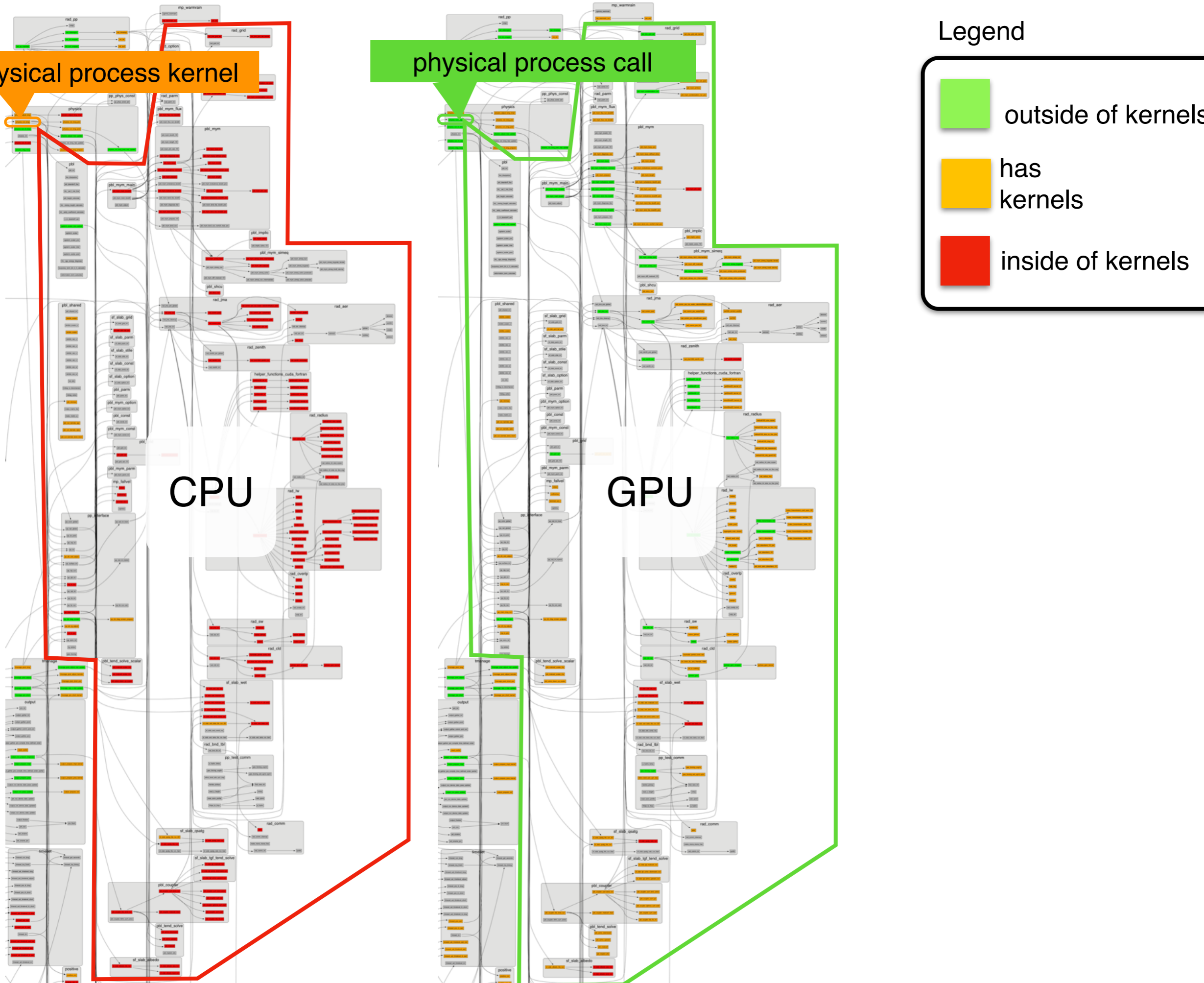
physical process call

Legend

- outside of kernels
- has kernels
- inside of kernels

CPU

GPU



Column-wise Courant-Friedrichs-Lewy Convergence

- Precipitation module uses separate CFL condition per column.
- Due to granularity shift, column-wise CFL convergence requires change from simple loop break to reduction kernel and masking array (cfl_reached).

```
@domainDependant{
  attribute(autoDom, present), domName(i,j), domSize(nx,ny)
}
cfl_reached, dt_rk_rest, ...
@end domainDependant

! ... initialization of variables

timestep_sed: do
  ! ... Runge-Kutta based iterative solution to sedimentation

  @parallelRegion{appliesTo(GPU), domName(i, j), domSize(nx, ny)}
  if ( dt_rk_rest < dt_rk_rest_epsln ) then
    cfl_reached = .true.
  end if
@end parallelRegion

  call all_true_for_xy_plane(cfl_reached, all_cfl_reached)

  if (all_cfl_reached) then
    exit timestep_sed
  end if
end do timestep_sed
```

Verification

- Hybrid ASUCA uses 64bit FP arithmetic throughout.
- Normalized root mean square error was tested continuously for pressure, moment and temperature variables. Stays within $1E-9$.
- Performed tests include:
 - Radiation test.
 - Physical process test for radiation, planetary boundary layer and surface.
 - Two-dimensional “warm bubble” test.
 - Various application configurations with real data, including full scale test on $1581 \times 1301 \times 58$ grid (2km resolution).

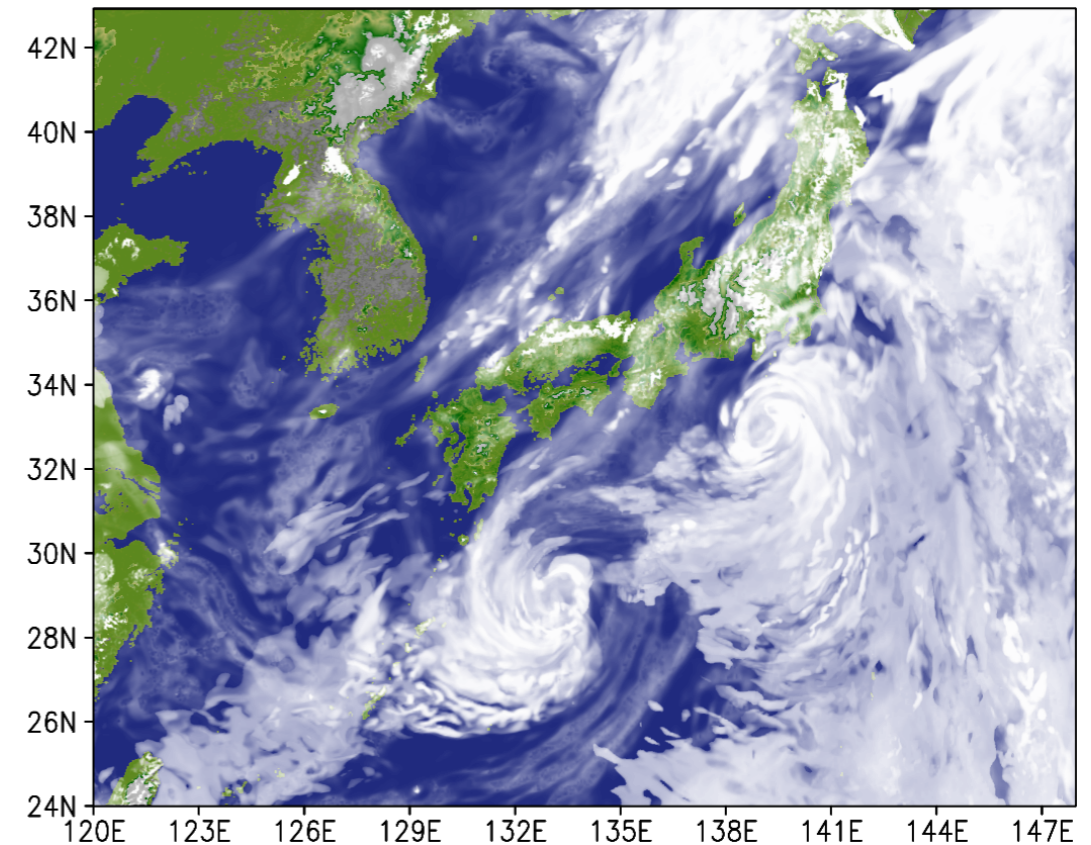
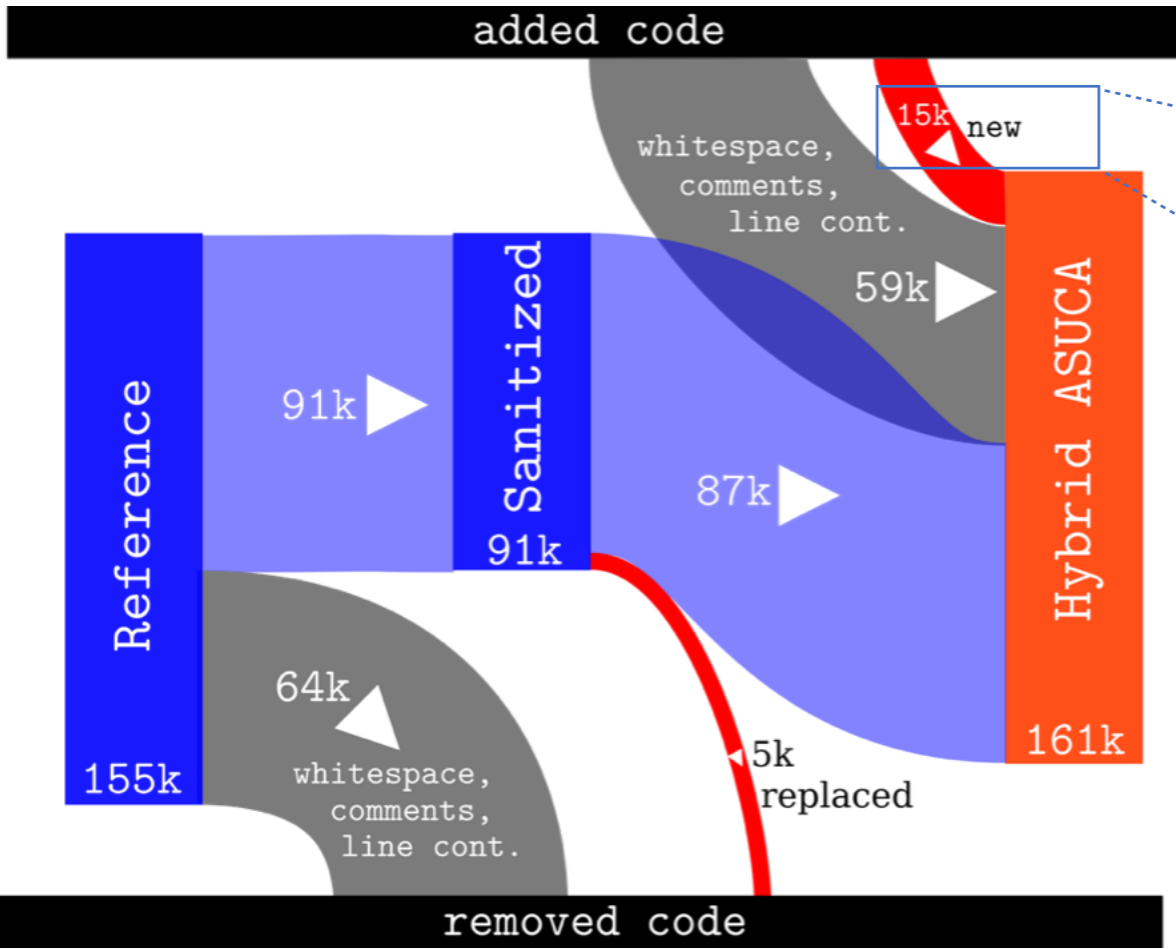


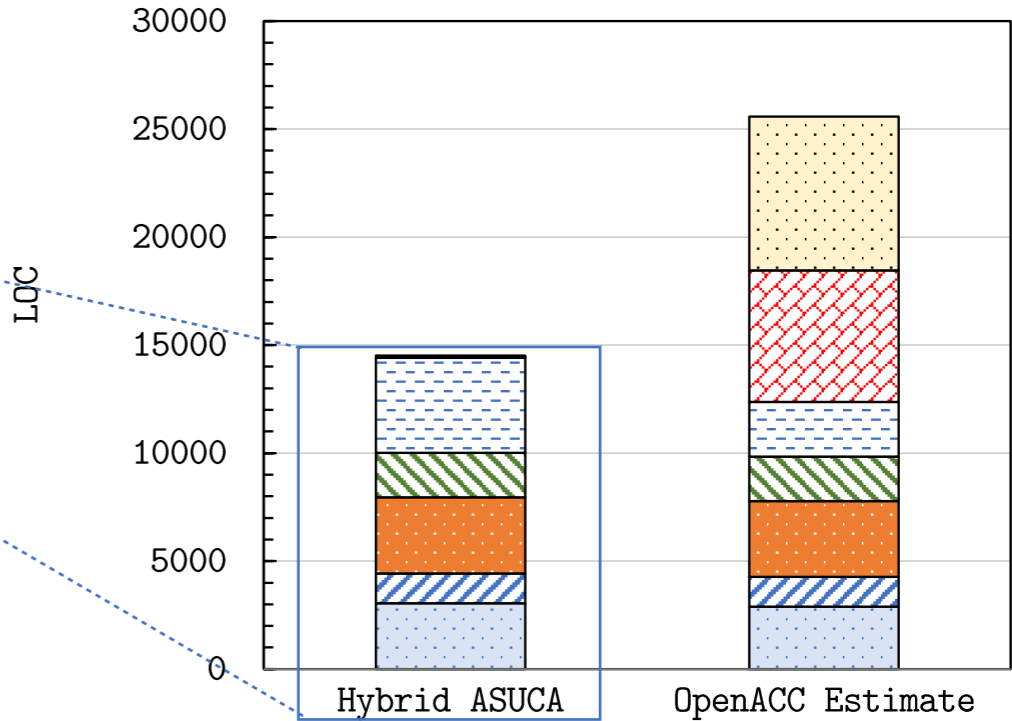
Figure 3.5: Total cloud cover result for ASUCA using 2km resolution grid and real initial data

Productivity Results

Code Reuse and Changes



Comparison with 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

4. Performance

Performance Model: Reduced Weather Application

$$\frac{\partial T}{\partial t} = \kappa \nabla^2 T + f$$

T	temperature
κ	thermal diffusivity
f	radiation heat

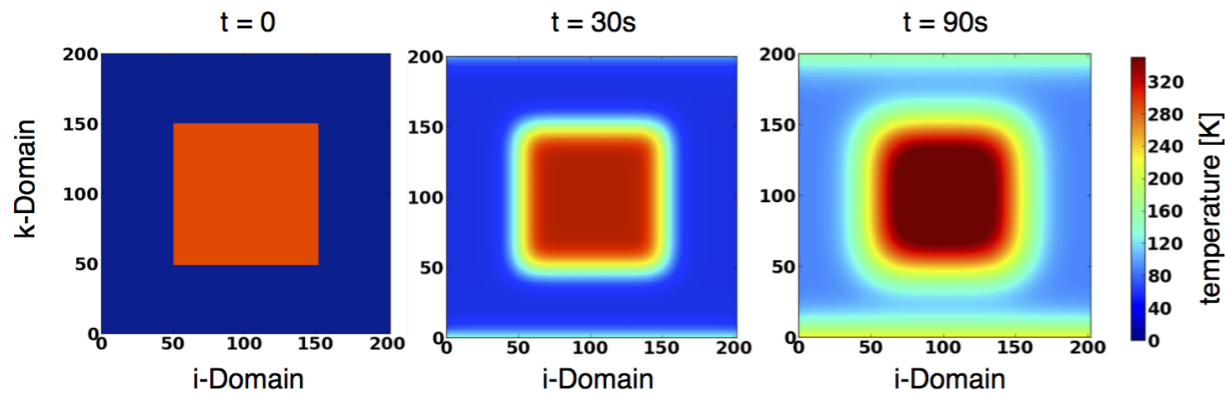
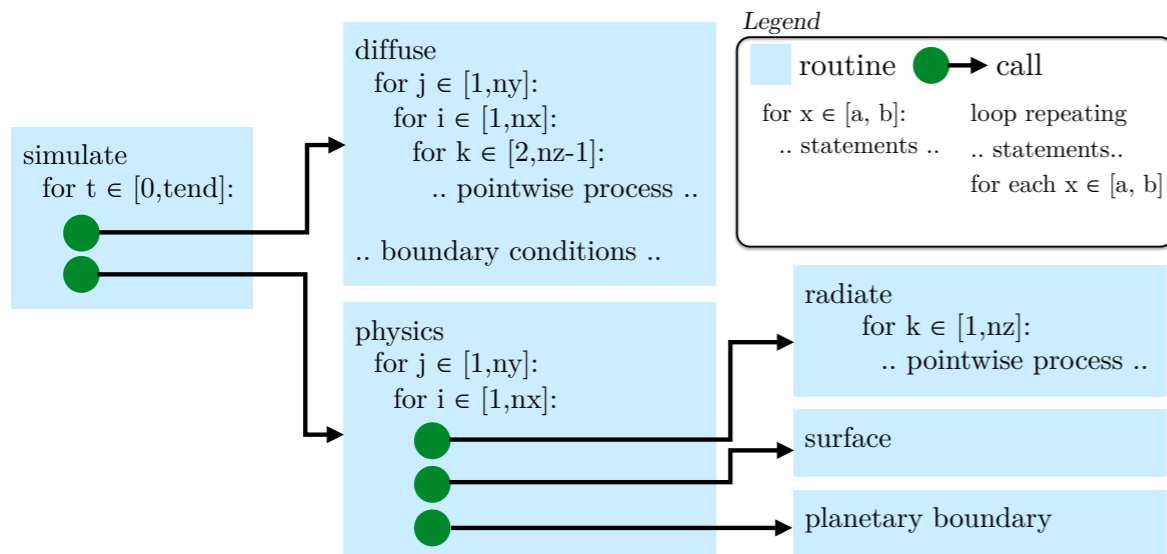


Figure 4.1: Output at $j = 100$.



- diffuse: 7-point Von-Neumann-type stencil, 0.125 FLOP/B DP
- radiate: 0.0625 FLOP/B DP
- ➔ memory bandwidth bounded on all architectures (e.g. system balance on P100: 7.8 FLOP/B, 6-core Westmere: 2.8 FLOP/B)

$$t_D = \underbrace{\frac{\Delta t_{output}}{\Delta t_{timestep}}}_{\text{\#timesteps between output}} \left(n_x \cdot n_y \cdot n_z \cdot \underbrace{\left(\frac{b \cdot m_{sa}}{BW_D} + \frac{b \cdot m_{HtoD}}{BW_{HtoD}} \right)}_{\text{pointwise inner diffusion}} + n_y \cdot n_z \cdot \underbrace{\frac{m_{ra}}{RA_D}}_{\text{pointwise diffusion boundary}} \right)$$

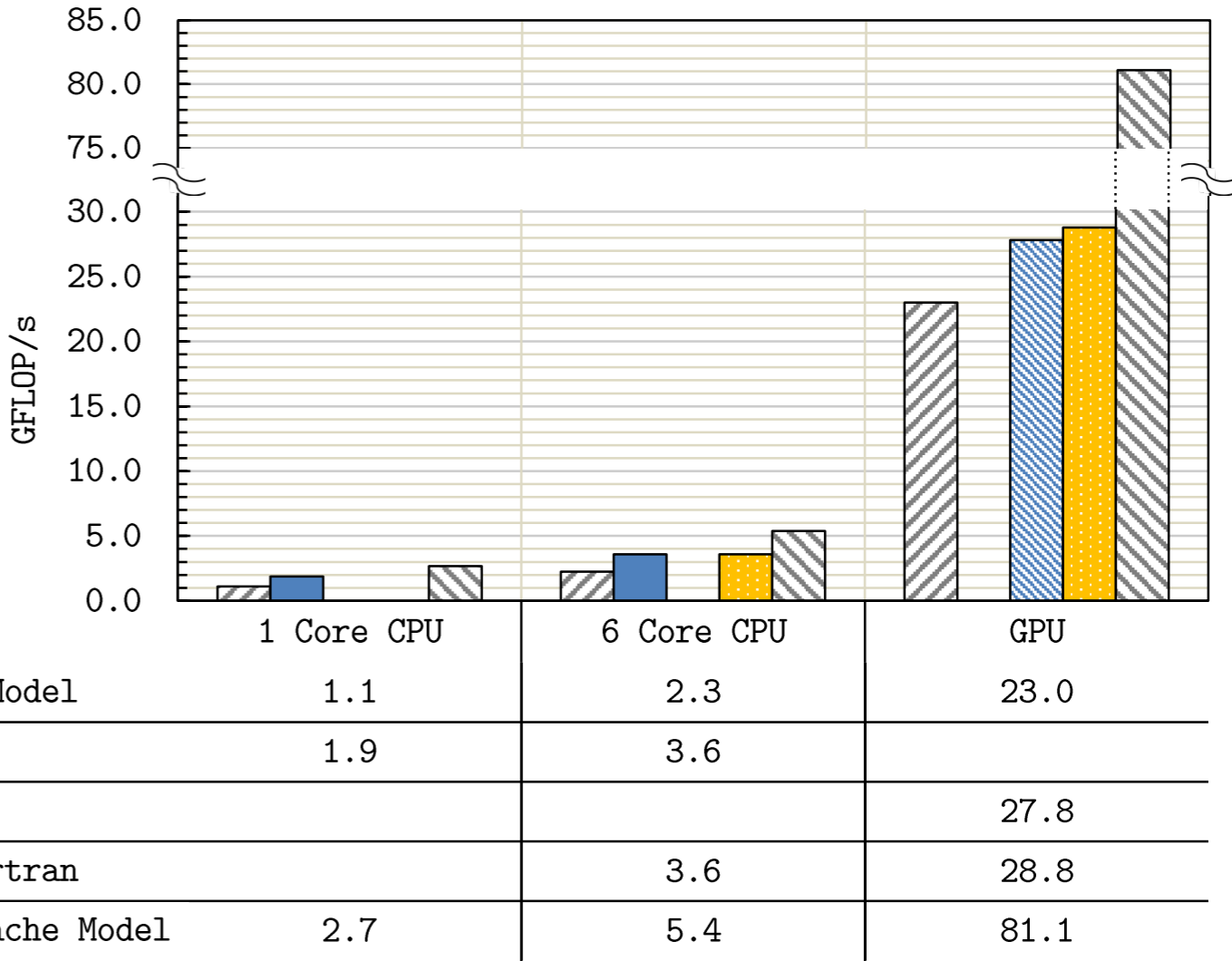
pointwise host/device I/O

Results: Reduced Weather Application

	IJK Order	KIJ Order
CPU Single Core	1.73s	1.28s
GPU (OpenACC) (Fastest Implementation)	0.10s	0.77s

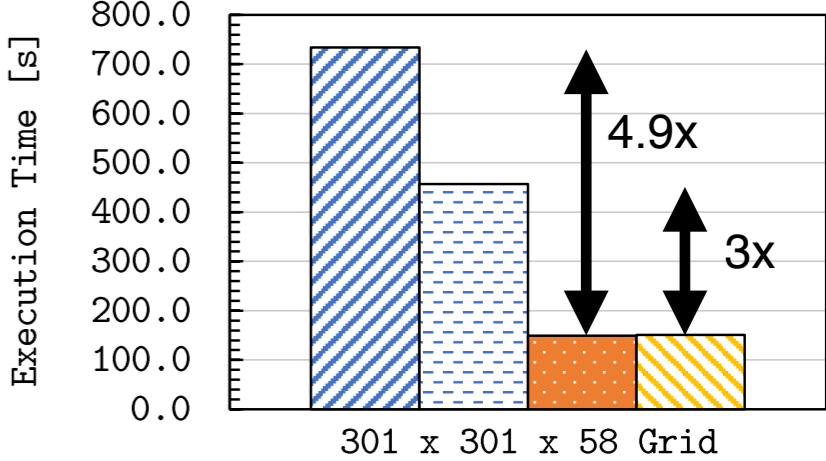
← Influence of storage order on execution time

Performance of reduced weather app. for separately implemented, vs. Hybrid Fortran generated, vs. model on 256x256x256 grid, 100 timesteps (fastest implementation)

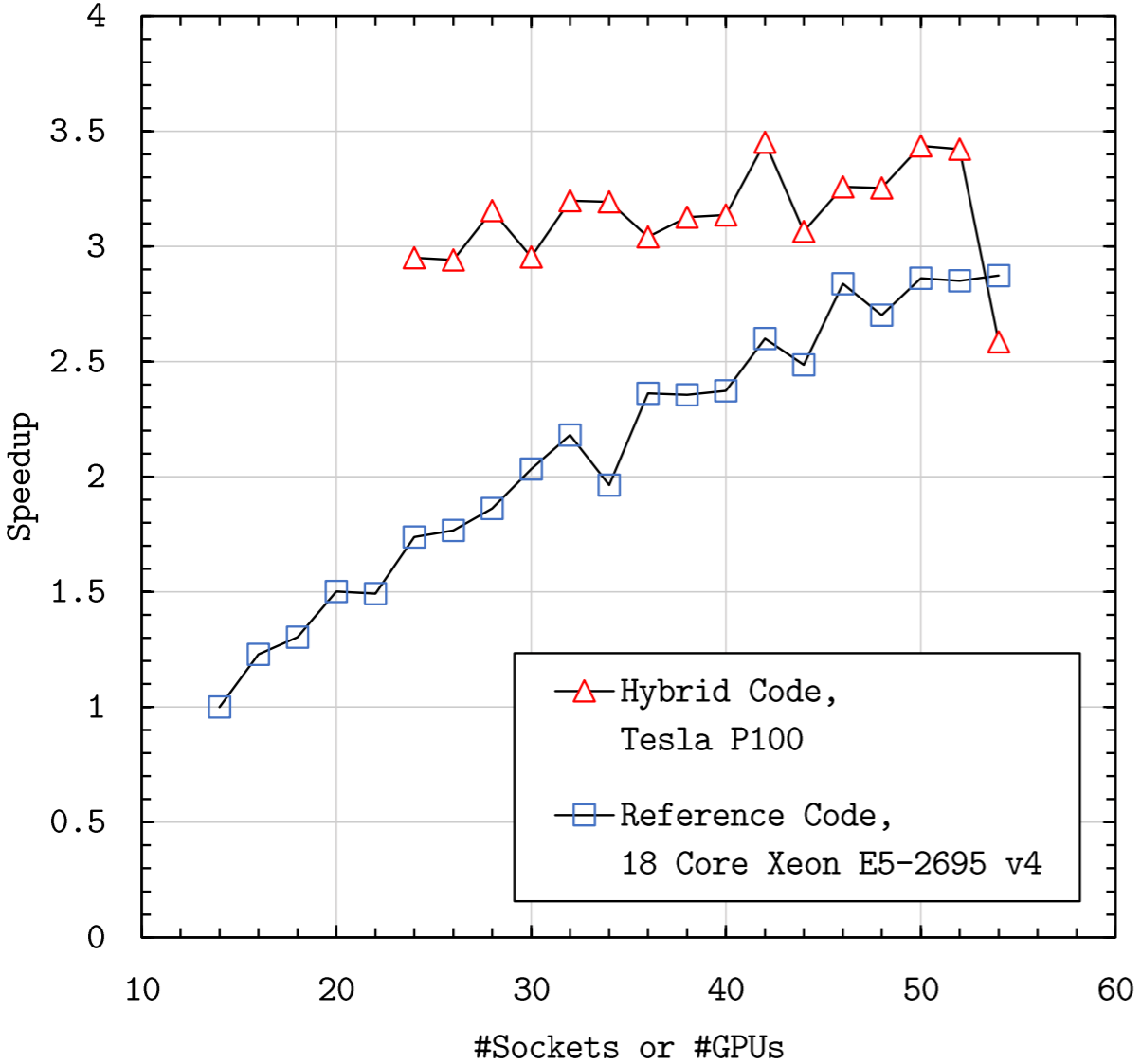


Results: Hybrid ASUCA

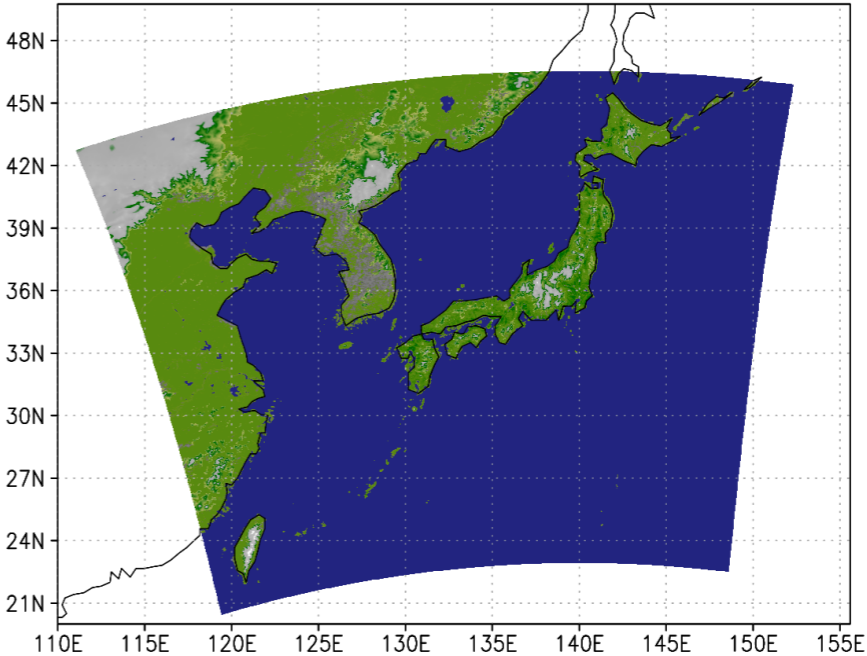
Kernel performance on reduced Grid (301 x 301 x 58) →



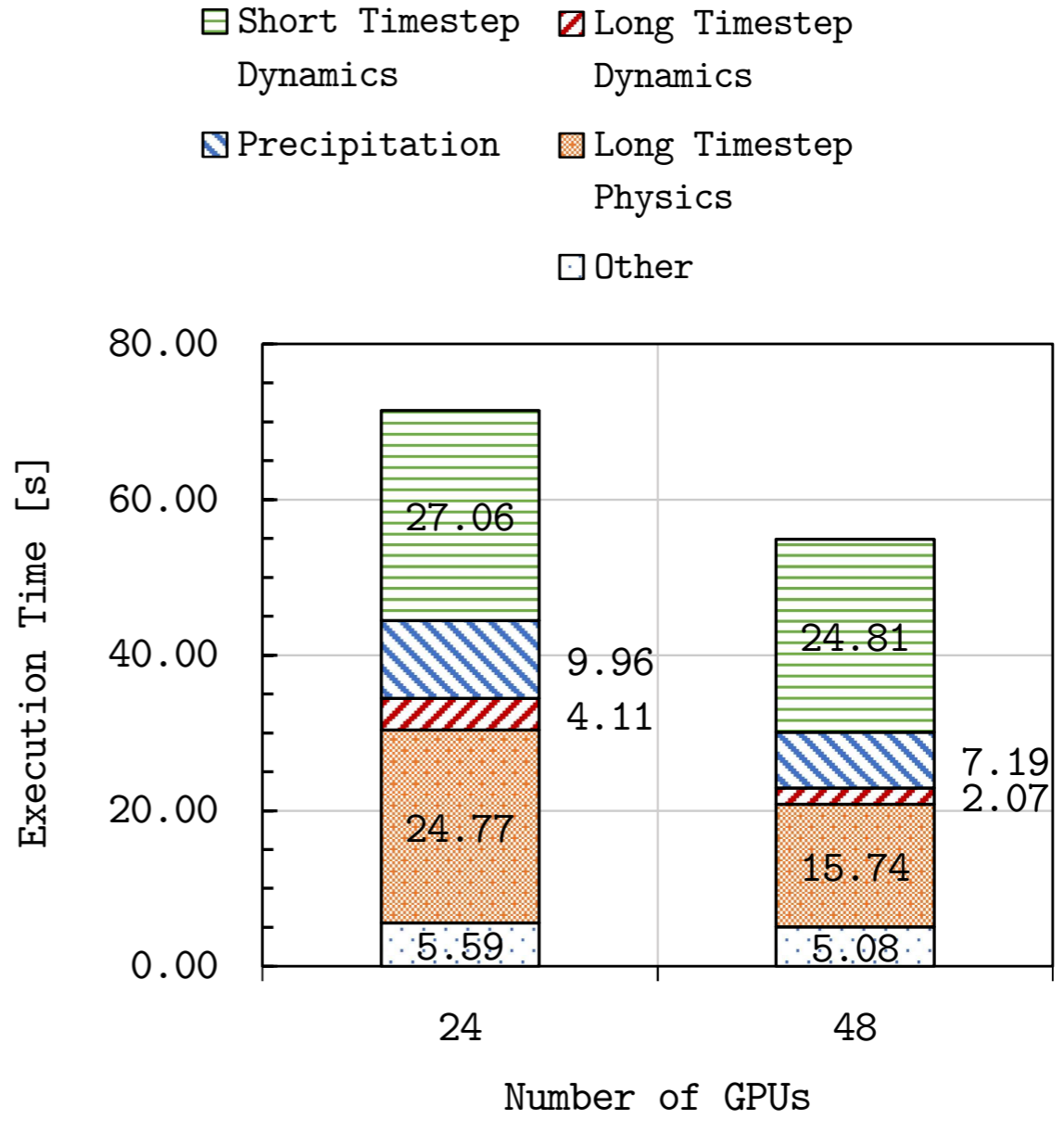
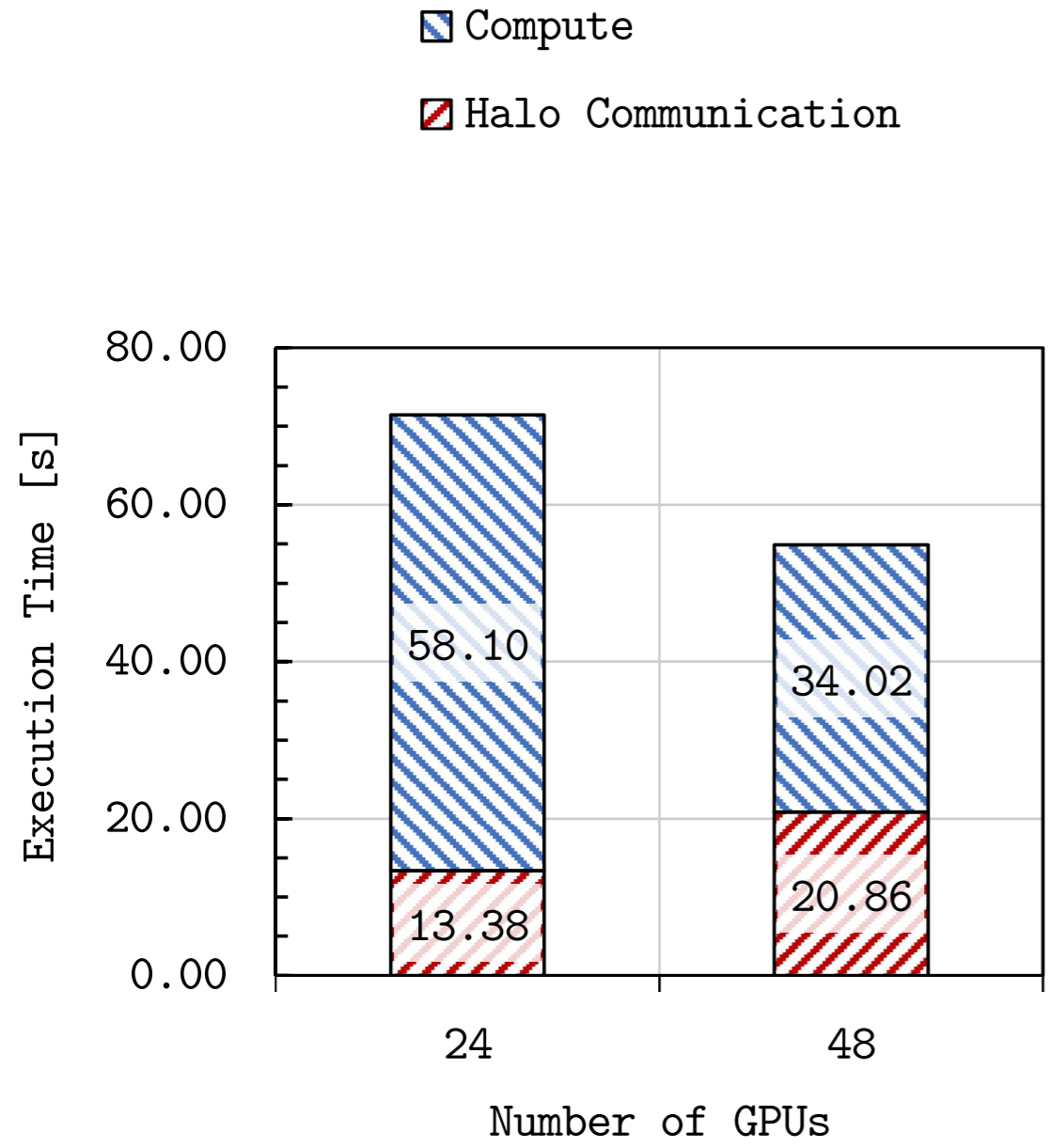
ASUCA Reference, 4 x 6-core Xeon X5670	734.0
ASUCA Reference, 1 x 18-core Xeon E5-2695 v4	456.7
Hybrid ASUCA, 4 x Tesla K20x	148.9
Hybrid ASUCA, 1 x Tesla P100	151.1



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



Results: Hybrid ASUCA

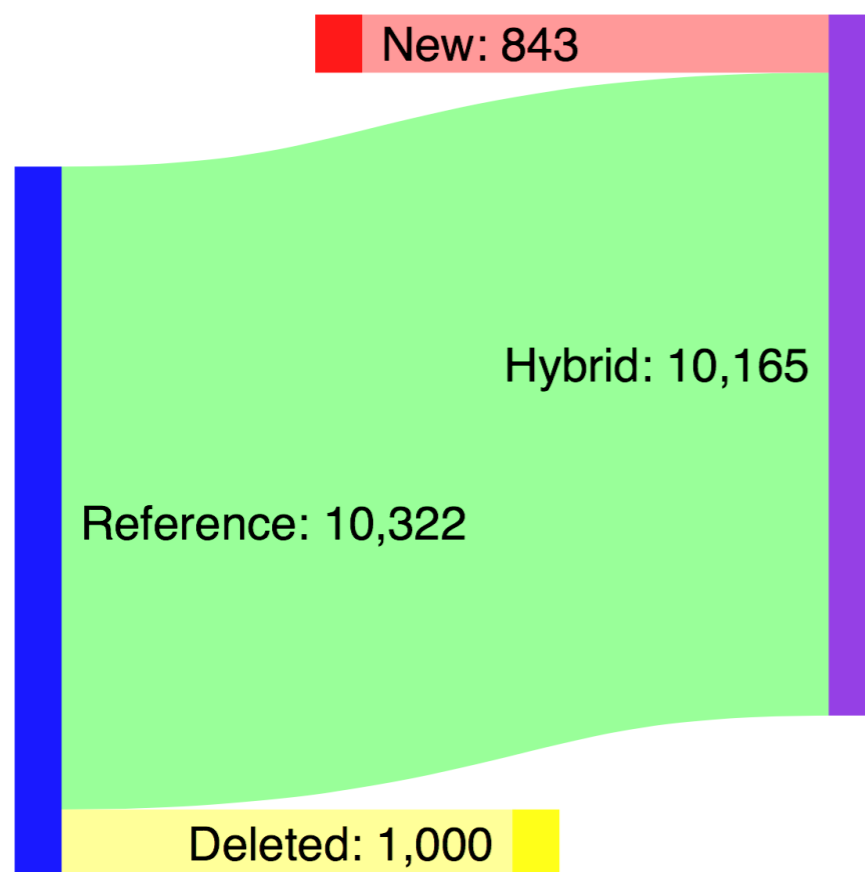


Impact of communication and modules for strong scaling on 1581 x 1301 x 58 ASUCA Grid, using 2x P100 GPU per node (TSUBAME 3)

Example Application: NICAM Physics

- Cloud microphysics
- Precipitation of rain, snow, graupel
- 111 loops to parallelize
- Due to timing issues and influenza: Roughly one week to work on this benchmark
- Hours logged: ~31.3.

number of lines of code:



	Runtime [s]
Reference, 2x 14-core Broadwell [1]	0.595
Hybrid, 2x 14-core Broadwell [2]	0.941
Hybrid, 1x P100 GPU [3]	0.232

5. Conclusion

Summary

Background

- ✓ paradigm shift towards throughput oriented design
- ✓ GPUs attractive for NWP (high mem. bandwidth)
- ✓ productivity and maintainability of GPU approaches lacking

Motivation

- ✓ Many of today's NWP- and climate models cannot make efficient use of high-throughput architectures. We want to find and prove easily adoptable approach.

Goal

- ✓ GPU port for "ASUCA" NWP model in Fortran with minimal code divergence / minimal learning

Contributions

- ✓ new granularity abstraction and memory layout transformation method
- ✓ applied to ASUCA, resulting in >3x speedup in kernel performance and >2x reduction in processors required for a full scale run with real data
- ✓ method unique in increasing productivity for porting coarse-grained codes to GPU

On all previous projects applying high-throughput architectures to NWP and climate models [27]:

“All these approaches were effectively addressing fine-grained parallelism in some way or other without addressing coarser grained concurrency, and all involved various levels of "intrusion" into code, from adding/ changing codes, to complete rewrites or translations.”

Prof. Bryan Lawrence

Professor of Weather and Climate Computing
Director of Models and Data @ NCAS

[27] Lawrence, Bryan N., et al. "Crossing the Chasm: How to develop weather and climate models for next generation computers?", under review for Geosci. Model Dev. (2017).

On how ACME model (DOE) cannot share a single source code for CPU and GPU due to register pressure[16]:

“The only remedy for this at present is to break the kernel up into multiple kernels. (...) On the CPU one would want to keep an element loop fused together for caching reasons.”

Dr. Matthew R. Norman
Computational Climate Scientist
Oak Ridge National Laboratory

[16] Norman, Matthew R., Azamat Mametjanov, and Mark Taylor. "Exascale Programming Approaches for the Accelerated Model for Climate and Energy." (2017).

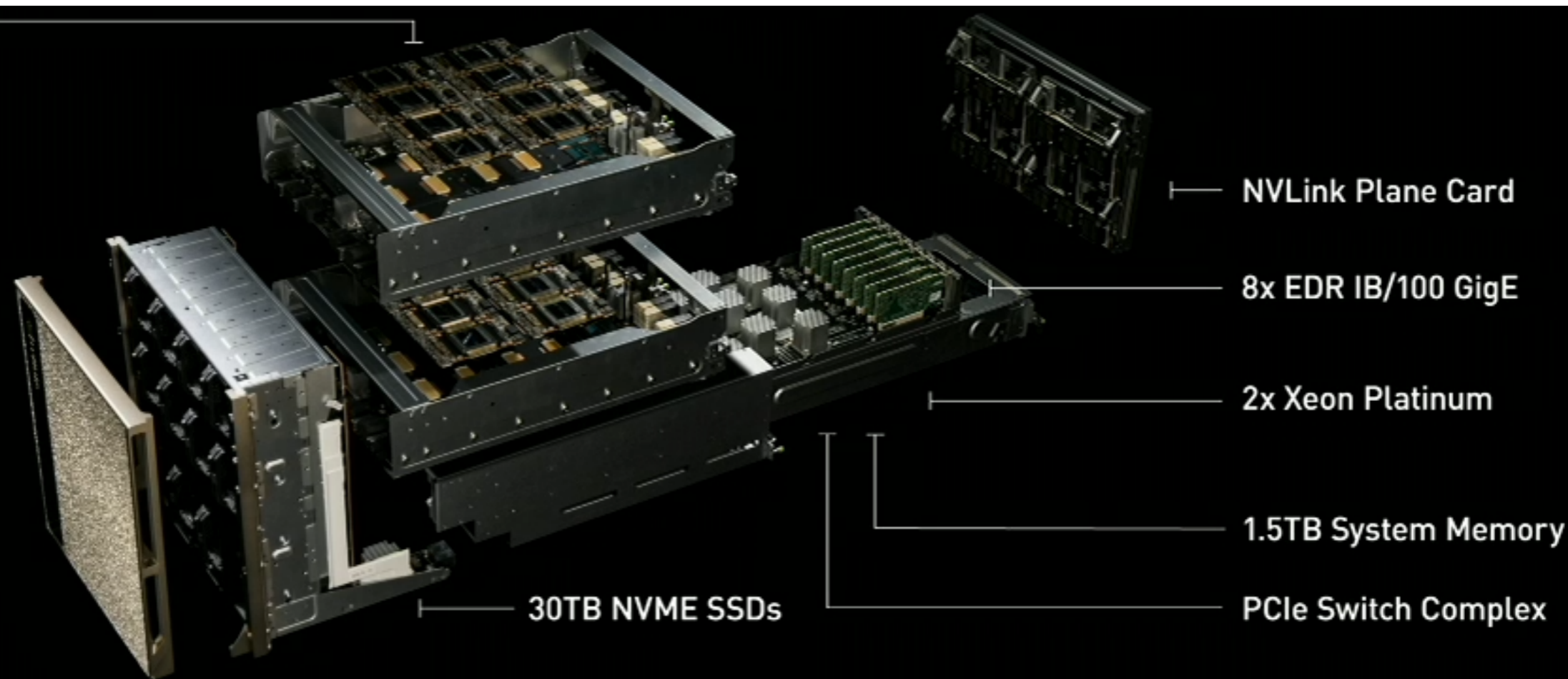
Concluding Remarks

- All previous projects porting NWP and climate models to high-throughput architectures had to choose between
 - complete rewrite (maximum learning),
 - code divergence (poor maintainability),
 - efficiency loss on at least one architecture (poor performance).
- This work shows a new approach, which has many potential applications beyond GPU and beyond NWP.
 - Hybrid Fortran is Open Source and can be applied directly where suitable.
 - Method as documented can be replicated in other applications, even if Hybrid Fortran is not used.

Outlook

- NVIDIA introduced DGX-2 - a 400k USD GPU system
- Thesis: Operational 2km ASUCA on a single DGX-2 possible
 - 16x Tesla V100s totaling 512GB HBM with unified address space
 - Halo communication entirely through 900 GB/s NVSwitch

16x Tesla V100 32GB
12x NVSwitch



Thank you for your attention.