GPU Acceleration of a Global Atmospheric Model by Python combining with CUDA

Ki-Hwan Kim

KIAPS (Korea Institute of Atmospheric Prediction Systems)

2016.8.13

PYCON
APAC 2016

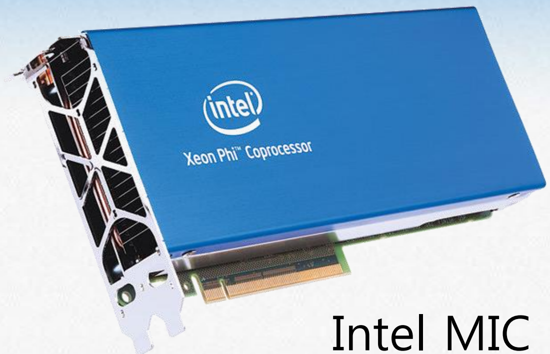
작년 파이콘 발표


PYCON Korea 2015
KIAPS
www.kiaps.org
Python의 계산 성능 향상을 위해
Fortran, C, CUDA-C, OpenCL-C
코드들과 연동하기
김기환
Korea Institute of
Atmospheric Prediction Systems
(KIAPS)
(재)한국형수치예보모델개발사업단

Modern Processors

- **Massively parallel**
- **High performance**
- **Low power consumption**

NVIDIA GPU



Intel MIC

CPU

AMD GPU

FPGA

How can we utilize modern processors?

- **Directive based tools in Fortran environment**
  - OpenACC
  - OpenMP (since v4.0)
  - Compiler can generate code for all targets
  - **Easy to start, but difficult to achieve high performance**

OpenACC

```
PROGRAM main
 INTEGER :: a(N), b(N)
 <stuff>
 !$acc parallel loop &
 & device_type(nvidia) num_gangs(200) &
 & device_type(host) num_gangs(16)
 DO i = 1,N
   a(i) = a(i) + rhs(i)
 END DO
 !$acc end parallel loop
 <stuff>
END SUBROUTINE main
```

OpenMP

```
PROGRAM main
 INTEGER :: a(N), b(N)
 <stuff>
 !$omp target teams distribute &
 & num_teams(x)
 DO i = 1,N
   a(i) = a(i) + rhs(i)
 END DO
 !$omp end target parallel do
 <stuff>
END SUBROUTINE main
```

Example codes from Cray

How can we utilize modern processors?

- **Native languages for each processor**

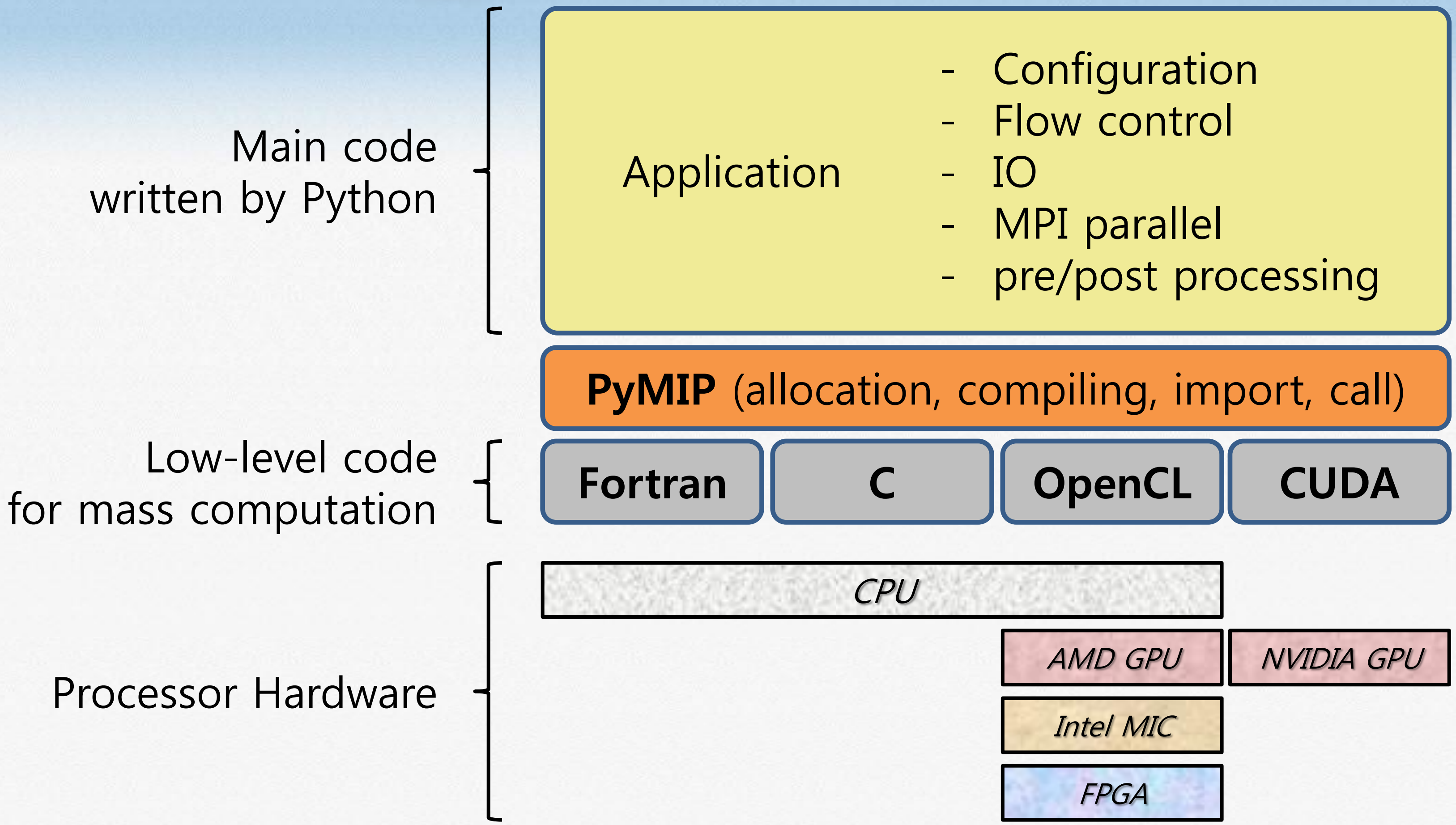
processors	CPU	GPU (NVIDIA)	GPU (AMD)	MIC (INTEL)	FPGA
Native Languages	C Fortran OpenCL …	**CUDA**	**OpenCL**	C Fortran **OpenCL**	VHDL Verilog **OpenCL**

- Native language is advantageous to achieve peak performance.
- **Is there a way to integrate various codes written by native languages?**

Suggestion: PyMIP

- **Py**thon based **M**achine **I**ndependent **P**latform
- **Easily switching hardware platforms with same programming interface**
- Provide two components
  - Generalized array variable
  - JIT compiling for low-level codes (Fortran, C, CUDA, OpenCL)
- Low-level codes can be tunable for peak performance

Layer structure of PyMIP



Simple example - DAXPY

Double precision $A \cdot X + Y$

```
import numpy as np

n = 2**20
a = np.random.rand()
x = np.random.rand(n)
y = np.random.rand(n)

y[:] = a*x + y
```

Simple example – DAXPY (Fortran, C)

daxpy_core.f90

```fortran
SUBROUTINE daxpy(n, a, x, y)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  REAL(8), INTENT(IN) :: a
  REAL(8), INTENT(IN) :: x(n)
  REAL(8), INTENT(INOUT) :: y(n)
  INTEGER :: i

  DO i=1,n
    y(i) = a*x(i) + y(i)
  END DO
END SUBROUTINE
```

daxpy_core.c

```c
void daxpy(int n, double a, double *x, double *y) {
    // size and intent of array arguments for f2py
    // x :: n, in
    // y :: n, inout
    int i;

    for (i=0; i<n; i++) {
        y[i] = a*x[i] + y[i];
    }
}
```

Simple example – DAXPY (OpenCL, CUDA)

daxpy_core.cl

```
//#pragma OPENCL EXTENSION cl_amd_fp64 : enable

__kernel void daxpy(int n, double a, __global double *x, __global double *y) {
    int gid = get_global_id(0);

    if (gid >= n) return;
    y[gid] = a*x[gid] + y[gid];
}
```

daxpy_core.cu

```
__global__ void daxpy(int n, double a, double *x, double *y) {
    int gid = blockDim.x * blockIdx.x + threadIdx.x;

    if (gid >= n) return;
    y[gid] = a*x[gid] + y[gid];
}
```

Simple example – DAXPY with PyMIP

daxpy_main.py

```python
import numpy as np
from numpy.testing import assert_array_almost_equal as aa_equal
from pymip import DevicePlatform

n = 2**20
a = np.random.rand()
x = np.random.rand(n)
y = np.random.rand(n)
ref = a*x + y

# CPU_F90, CPU_C, CPU_OpenCL, NVIDIA_GPU_CUDA
platform = DevicePlatform('CPU', 'F90')

xx = platform.ArrayAs(x)
yy = platform.ArrayAs(y)

src = open(__file__.replace('daxpy_core.'+platform.code_type)).read()
libpath = platform.source_compile(src)
lib = platform.load_library(libpath)
func = platform.get_function(lib, 'daxpy')

# (int32, float64, float64 array, float64 array)
func.prepare('idoo', n, a, xx, yy, gsize=n)
func.prepared_call()

aa_equal(ref, yy.get(), 15)
```

Setup

Set target device

Allocation

Compile & import

Call subroutine

Check result

Simple example – DAXPY with PyMIP

daxpy_main.py

Setup / Set target device / Allocation / Compile & import / Call subroutine / Check result

```
import numpy as np
from numpy.testing import assert_array_almost_equal as aa_equal
from pymip import DevicePlatform

n = 2**20
a = np.random.rand()
x = np.random.rand(n)
y = np.random.rand(n)
ref = a*x + y

# CPU_F90, CPU_C, CPU_OpenCL, NVIDIA_GPU_CUDA
platform = DevicePlatform('CPU', 'F90')

xx = platform.ArrayAs(x)
yy = platform.ArrayAs(y)

src = open(__file__.replace('daxpy_core.'+platform.code_type)).read()
libpath = platform.source_compile(src)
lib = platform.load_library(libpath)
func = platform.get_function(lib, 'daxpy')

# (int32, float64, float64 array, float64 array)
func.prepare('idoo', n, a, xx, yy, gsize=n)
func.prepared_call()

aa_equal(ref, yy.get(), 15)
```

Simple example – DAXPY with PyMIP

daxpy_main.py

Setup

Set target device

Allocation

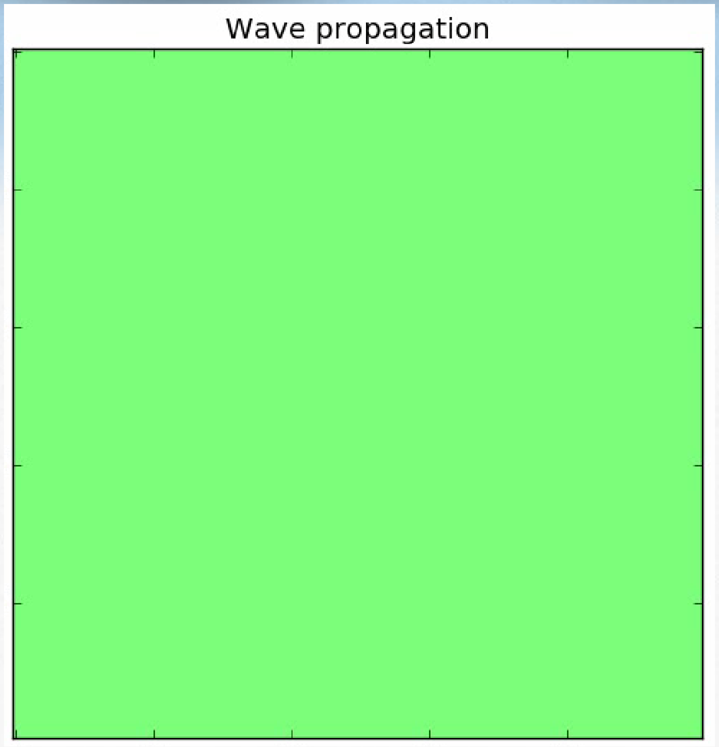
Compile & import

Call subroutine

Check result

```
import numpy as np
from numpy.testing import assert_array_almost_equal as aa_equal
from pymip import DevicePlatform

n = 2**20
a = np.random.rand()
x = np.random.rand(n)
y = np.random.rand(n)
ref = a*x + y

# CPU_F90, CPU_C, CPU_OpenCL, NVIDIA_GPU_CUDA
platform = DevicePlatform('NVIDIA_GPU', 'CUDA')

xx = platform.ArrayAs(x)
yy = platform.ArrayAs(y)

src = open(__file__.replace('daxpy_core.'+platform.code_type)).read()
libpath = platform.source_compile(src)
lib = platform.load_library(libpath)
func = platform.get_function(lib, 'daxpy')

# (int32, float64, float64 array, float64 array)
func.prepare('idoo', n, a, xx, yy, gsize=n)
func.prepared_call()

aa_equal(ref, yy.get(), 15)
```

Simple performance test – 2D wave propagation



- 2D wave equation

$$\frac{\partial^2 u}{\partial t^2} = a^2 \nabla^2 u = a^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

- Numerics: Central finite-difference

$$u\Big|_{i,j}^{n+1} = \left(a \frac{\Delta_t}{\Delta_x} \right)^2 \left(u\Big|_{i-1,j}^{n} + u\Big|_{i+1,j}^{n} + u\Big|_{i,j-1}^{n} + u\Big|_{i,j+1}^{n} - 4u\Big|_{i,j}^{n} \right) + 2u\Big|_{i,j}^{n} - u\Big|_{i,j}^{n-1}$$

Simple performance test – 2D wave propagation

- **CPU** (Intel E5-2690, ICC v15.0.3)

	F90	C	PyMIP (F90)	PyMIP (C)
wallclock	3m 57s	3m 53s	4m 7s	4m 2s
	237 s	233 s	247 s	242 s

- **CPU** (Intel Xeon X5675)

	PyMIP (OpenCL)
wallclock	2m 32s
	152 s

- **GPU** (NVIDIA Tesla M2090)

	PyMIP (CUDA)
wallclock	37 s

[Setup] 10,000x10,000 grid, 1,200 time steps

Real example – Global atmospheric model

- Calculate the global atmospheric flow
- Required for weather/climate prediction
- Main components
  - **Dynamical core**
  - Physics process
  - Data assimilation

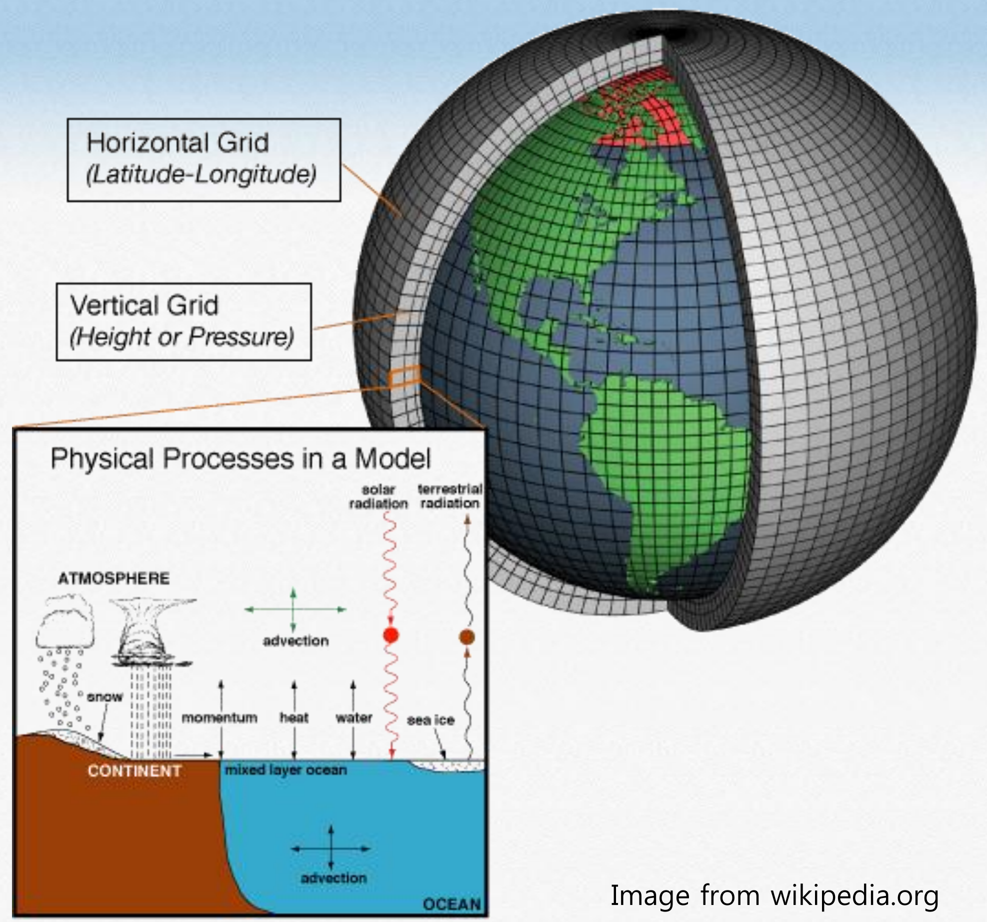


Image from wikipedia.org

Real example – Global atmospheric model

- GEOS 5 (Goddard Earth Observing System)
- Global Atmospheric model which has been developing in NASA
- Resolution: 7 km
- Time step: 30 minutes
- Visualizations by Greg Shirah on August 10, 2014

Governing equations of a dynamical core

***V. Bjerknes** (1904) pointed out for the first time that there is a complete set of*
***7 equations with 7 unknowns** that governs the evolution of the atmosphere:*

Momentum $$\frac{d\mathbf{v}}{dt} = -\alpha\nabla p - \nabla\phi + \mathbf{F} - 2\Omega\times\mathbf{v}$$

Mass $$\frac{\partial\rho}{\partial t} = -\nabla.(\rho\mathbf{v})$$

Ideal-gas law $$p = \rho RT$$

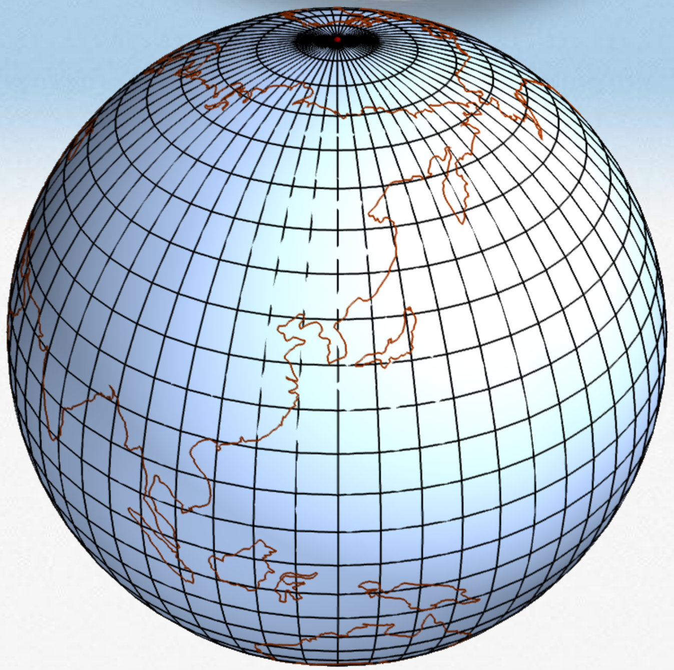
Energy $$\frac{ds}{dt} = C_p\frac{1}{\theta}\frac{d\theta}{dt} = \frac{Q}{T}$$

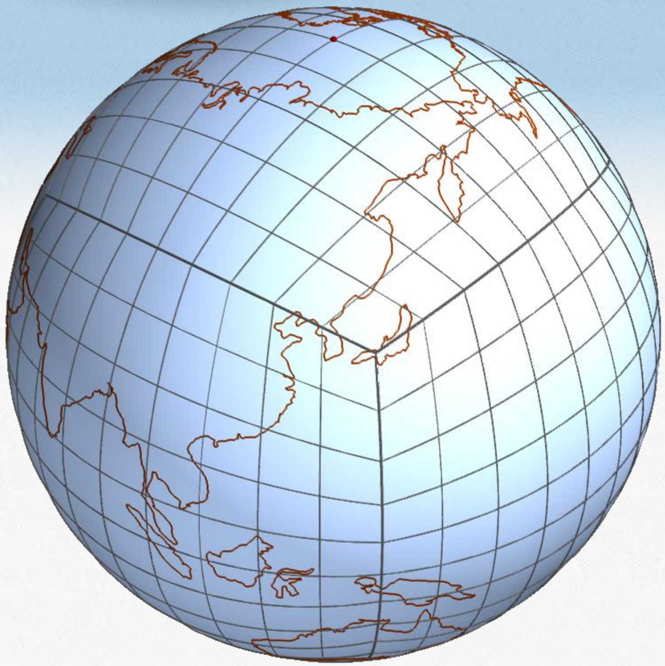
Moisture $$\frac{dq}{dt} = E - C$$

7 equations, 7 unknowns (u, v, w, T, p, s, q) ➔ **solvable**

from Song-you Hong, KIAPS

Grid on the sphere





Lat-lon grid

- **Most popular**
- **Orthogonal coordinate system**
- **Polar singularity → Low parallel efficiency**

Cubed-sphere grid

- **No polar singularity**
- **Suitable for Spectral Element Method**
- **Non-orthogonal coordinate system**

Numerical method for the dynamical core

❐ Spatial derivative → Spectral Element method

- Divide domain to elements ← Finite Element Method
- Polynomial expansion in each element ← Spectral Method
- High parallel efficiency

$$\frac{\partial \psi}{\partial t} + \nabla \cdot \mathbf{p} = 0$$

$$\frac{\partial \psi_{ij}}{\partial t} = -\left[\sum_{k=0}^{N} D_{ik} \left(p_x\right)_{kj} + \sum_{k=0}^{N} \left(p_y\right)_{ik} D_{kj}^{T}\right]$$

$$D_{ij} = \frac{\partial L_j(\xi_i)}{\partial \xi} = \begin{cases} \frac{1}{x_i - x_j} \frac{P_N(x_i)}{P_N(x_j)} & i \neq j \\ -\frac{N(N+1)}{4} & i = j = 0 \\ \frac{N(N+1)}{4} & i = j = N \\ 0 & 0 < i = j < N \end{cases}$$

❐ Temporal derivative → 3rd Runge-Kutta Method

Count code lines of KIM model

KIM v2.2.15 without Core_SH and External codes

	files	lines	code	ratio (%)
/	532	**238601**	**143878**	100.00%
/Model	139	85637	53050	36.87%
/Shared	77	73064	44707	**31.07%**
/Tools	317	81055	47160	32.78%

/Shared/Base	5	563	275	0.19%
/Shared/Framework	12	9457	5085	3.53%
/Shared/Grid	35	39072	26087	18.13%
/Shared/IoModule	19	18430	10713	7.45%
/Shared/Parallel	5	5515	2535	1.76%

Replace with Python codes

/Model/AtmosModel/DynamicalCore	22	12983	8225	**5.72%**
/Model/AtmosModel/PhysicsPackage	109	70447	43416	30.18%

/Model/AtmosModel/DynamicalCore/Core_SW	14	8666	5641	**3.92%**

Convert to C, CUDA, OpenCL
(**60~70 % of wallclock time**)

Subroutines for KIM dynamical core

parallel	Generate cubed-sphere grid
	Domain decomposition
	DSS communication

derivative	divergence_sphere()
	gradient_sphere()
	vorticity_sphere()
	divergence_sphere_wk()
	laplace_sphere_wk()
	gradient_sphere_wk_testconv()
	curl_sphere_wk_testconv()
	vlaplace_sphere_wk()

big step	calc_divv()
	calc_ww()
	calc_cq_alt_php()
	init_zero()
	rhs_ph()
	horizontal_pressure_gradient()
	pg_buoy_w()
	coriolis_wVort()
	curvature_wVort()
	update_tendency()
	advance_scalar_moist()
	advance_scalar_trace()
	calc_p_rho_phi()
	horizontal_diffusive_Spn()
	horizontal_explicit_diffusion()
	horizontal_explicit_diffusion_moist()
	horizontal_explicit_diffusion_trace()
	vertical_explicit_diffusion()
	horizontal_explicit_diffusion_ss()
	horizontal_explicit_diffusion_moist_ss()
	horizontal_explicit_diffusion_trace_ss()
	limiter_moist_trace()

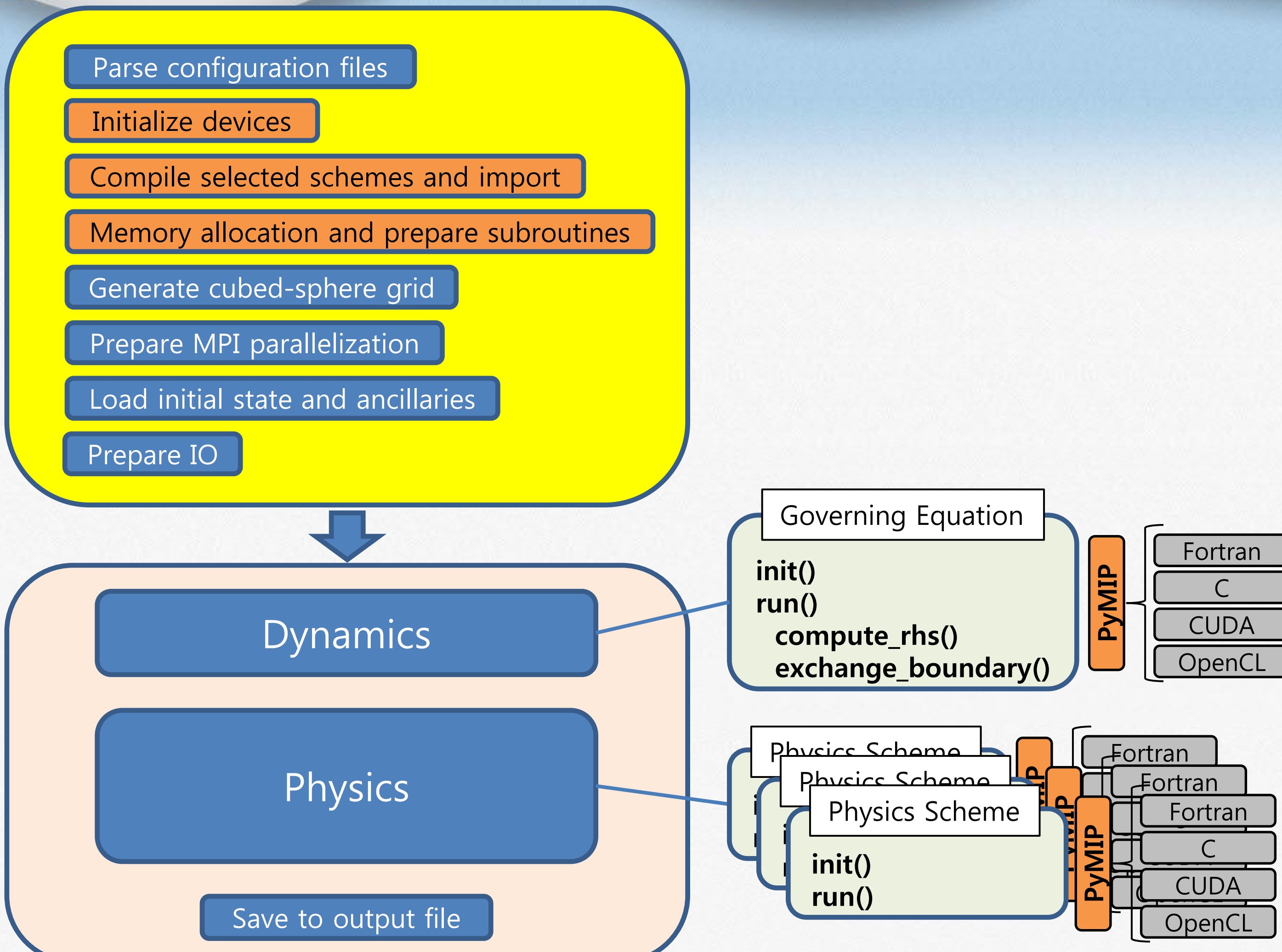
small step	small_step_pre()
	calc_p_rho()
	calc_coef_w()
	advance_uv()
	calc_divv_small()
	advance_ww_pds()
	calc_mu()
	advance_t()
	advance_w()
	sumflux()
	small_step_finish()

advect	advect_uv_kinetic()
	advect_w()
	advect_scalar()
	advect_scalar_moist()

Workflow of framework based on PyMIP

Setup

Time Loop



Result on CPU and GPU

KIM v2.3

[ne30np4(~100km), 1 day prediction]

name	ncalls	nranks	mean_time	std_dev	wallmax (rank)		wallmin (rank)	
Total	16	16	13682.964	0.009	13682.975 (	10)	13682.953 (	0)
IniGrid	16	16	1.612	0.010	1.618 (	11)	1.583 (	15)
IniAtmosModel	16	16	2.596	0.007	2.607 (	9)	2.586 (	8)
RunAtmosModel	16	16	13678.514	0.007	13678.521 (	0)	13678.506 (	9)
RunDynamicalCore	15360	16	13610.672	1.478	13612.058 (	6)	13607.065 (	0)
RunCore_SW	15360	16	13610.668	1.478	13612.055 (	6)	13607.063 (	0)
WriteDynamicsOutput(Run)	15360	16	65.516	1.457	69.010 (	8)	64.125 (	6)
WriteOutputAPIs(Run)	15360	16	2.294	0.006	2.301 (	6)	2.278 (	0)

CPU : GAON2(1 node, 2 sockets, 16 cores)
Intel Xeon E5-2690 (92.8 GFLOP/s, 51.2 GB/s, 2012.Q1)

Python+CUDA

	1 GPU	2 GPU	3 GPU	4 GPU
real	30m42.478s	38m23.651s	27m43.319s	20m42.388s
user	22m47.228s	57m34.304s	63m7.244s	63m38.328s
sys	7m55.364s	19m10.336s	19m58.800s	19m7.636s

1 GPU x7.4 — w/out MPI

2 GPU x5.9 **3 GPU x8.2** **4 GPU x11.0** — with MPI

GPU : Bricks(1 node, 4 GPU)
NVIDIA TESLA M2090 (665.6 GFLOP/s, 177.6 GB/s, 2011.Q3)

Summary

- Suggestion of a new methodology for various modern processors

- The new methodology is based on Python
  - Integrate codes written by native languages (Fortran, C, CUDA, OpenCL)
  - Catch both productivity and high performance

- It works well in a big realistic problem as the global atmospheric model.

- Future plan
  - Porting of the dynamical core to Intel MIC
  - Performance analysis and tuning
  - **Porting of the Physics components**

(재)한국형수치예보모델개발사업단이

현대과학의 한계를 뛰어넘는 새로운 도전을 시작합니다.

Thank You

GLOBAL CENTER OF NUMERICAL WEATHER PREDICTION MODELING

KIAPS

KOREA INSTITUTE OF ATMOSPHERIC PREDICTION SYSTEMS

Speed up with memory optimization

parallel	generate cubed-sphere grid	
	domain decomposition	
	DSS communication	

derivative	divergence_sphere()	
	gradient_sphere()	
	vorticity_sphere()	
	divergence_sphere_wk()	
	laplace_sphere_wk()	
	gradient_sphere_wk_testconv()	
	curl_sphere_wk_testconv()	
	vlaplace_sphere_wk()	

big step	calc_divv()	6.8 %
	calc_ww()	-
	calc_cq_alt_php()	-
	init_zero()	-
	rhs_ph()	13.9 %
	horizontal_pressure_gradient()	22.7 %
	pg_buoy_w()	-
	coriolis_wVort()	3.1 %
	curvature_wVort()	-
	update_tendency()	-
	advance_scalar_moist()	-
	advance_scalar_trace()	-
	calc_p_rho_phi()	-
	horizontal_diffusive_Spn()	12.0 %
	horizontal_explicit_diffusion()	5.3 %
	horizontal_explicit_diffusion_moist()	-
	horizontal_explicit_diffusion_trace()	-
	vertical_explicit_diffusion()	-
	horizontal_explicit_diffusion_ss()	5.3 %
	horizontal_explicit_diffusion_moist_ss()	-
	horizontal_explicit_diffusion_trace_ss()	-
	limiter_moist_trace()	-

small step	small_step_pre()	-
	calc_p_rho()	-
	calc_coef_w()	-
	advance_uv()	14.4 %
	calc_divv_small()	-
	advance_ww_pds()	-
	calc_mu()	22.7 %
	advance_t()	9.0 %
	advance_w()	-
	sumflux()	28.1 %
	small_step_finish()	-

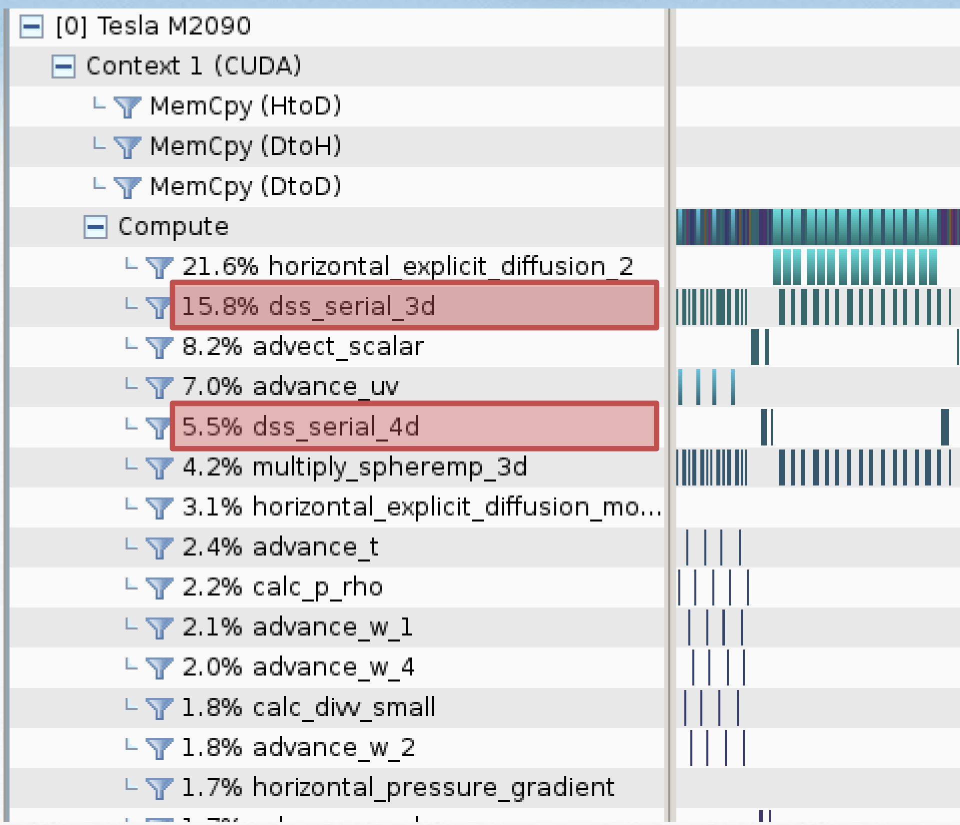
advect	advect_uv_kinetic()	2.6 %
	advect_w()	-
	advect_scalar()	-
	advect_scalar_moist()	-

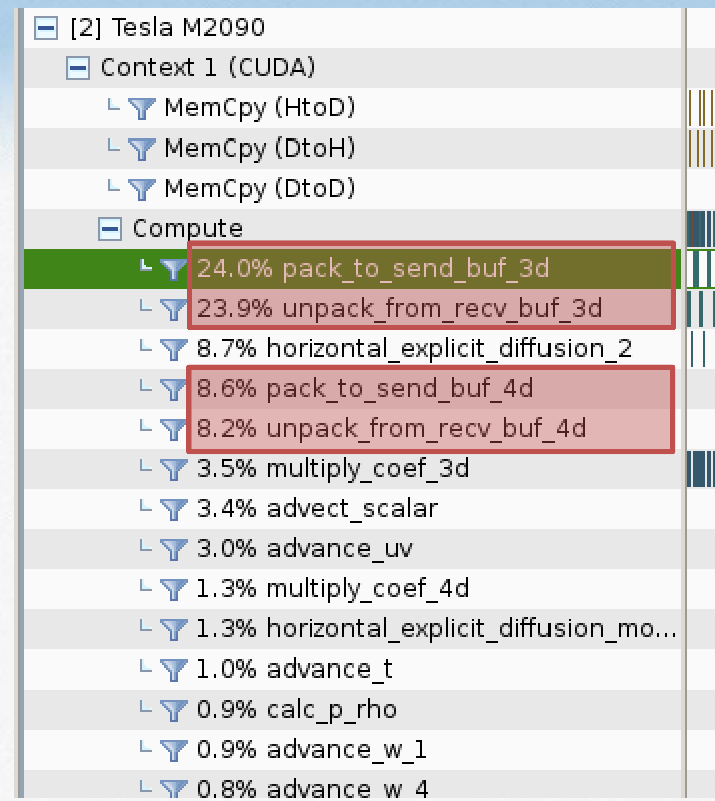
Total speed up

➔ 3.6 %

Bottleneck

Profiling using nvprof





21.3 % for DSS → **(64.7+α) % for DSS**

without MPI **with MPI**

Bottleneck

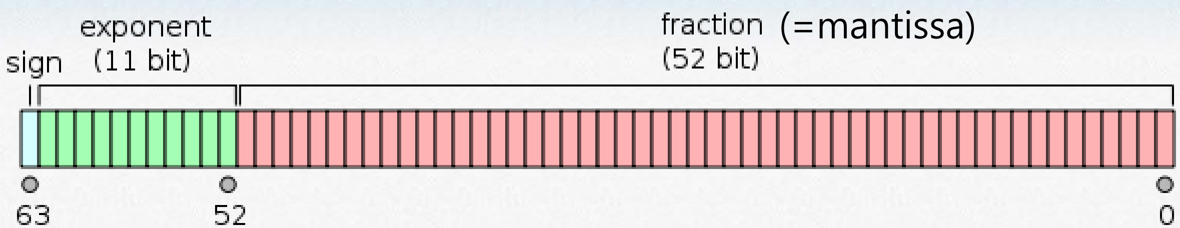
```
__global__ void pack_to_send_buf_3d(int shift_gid, int nelem,
        int nk, int nvar, int send_map_size, int local_src_size,
        int *dsts, int *srcs,
        double *send_buf, double *local_buf,
        double *var, int ivar) {

    int tid = threadIdx.x;
    int idx = blockDim.x * blockIdx.x + tid + shift_gid;

    // indices
    if (idx >= send_map_size) return;

    // local
    int dst, src;
    int ie, j, i, k, midx;

    dst = dsts[idx];
    src = srcs[idx];

    ie = src/NP2;
    j = ( src%NP2 )/NP;
    i = src%NP;

    if ( idx < local_src_size ) {
        for (k=0; k<nk; k++) {
            midx = ie*nk*NP2 + k*NP2 + j*NP + i;
            local_buf[dst*nk*nvar + ivar*nk + k] = var[midx];
        }
    }
    else {
        for (k=0; k<nk; k++) {
            midx = ie*nk*NP2 + k*NP2 + j*NP + i;
            send_buf[dst*nk*nvar + ivar*nk + k] = var[midx];
        }
    }
}
```

```
__global__ void unpack_from_recv_buf_3d(int shift_gid, int nelem,
        int nk, int nvar, int udsts_size, int local_buf_size,
        int *udsts, int *start_idxs, int *end_idxs, int *srcs,
        double *recv_buf, double *local_buf,
        double *var, int ivar) {

    int tid = threadIdx.x;
    int idx = blockDim.x * blockIdx.x + tid + shift_gid;

    // indices
    if (idx >= udsts_size) return;

    // local
    int dst, src;
    int ie, j, i, k, l, midx;
    double tmp;

    dst = udsts[idx];
    ie = dst/NP2;
    j = ( dst%NP2 )/NP;
    i = dst%NP;

    for (k=0; k<nk; k++) {
        tmp = 0;
        for (l=start_idxs[idx]; l<=end_idxs[idx]; l++) {
            src = srcs[l];

            if (src < local_buf_size)
                tmp += local_buf[src*nk*nvar + ivar*nk + k];
            else
                tmp += recv_buf[(src-local_buf_size)*nk*nvar +
ivar*nk + k];
        }

        midx = ie*nk*NP2 + k*NP2 + j*NP + i;
        var[midx] = tmp;
    }
}
```

Consistency check

How do we compare floating-point numbers?

IEEE 754 double floating-point format



$$\boxed{\boldsymbol{x = (mantissa) \times 2^{(exponent)}}}$$

12345.678901234567 → $0.7535204407491 8012 \times 2^{14}$
12345.678901234000 → $0.7535204407491 4548 \times 2^{14}$

0.12345678901234567 → $0.98765431209876542 \times 2^{-3}$
0.12345678901234000 → $0.98765431209872001 \times 2^{-3}$

0.00012345678901234567 → $0.50567900779456787 \times 2^{-12}$
0.00012345678901234000 → $0.50567900779454467 \times 2^{-12}$

← Using numpy.frexp(x)

First, compare exponents

Then, count same digit of mantissas