GPU TECHNOLOGY CONFERENCE

# Challenges in Accelerating Operational Weather and Climate Codes

Peter Strazdins, The Australian National University
Peter.Strazdins [at] cs.anu.edu.au

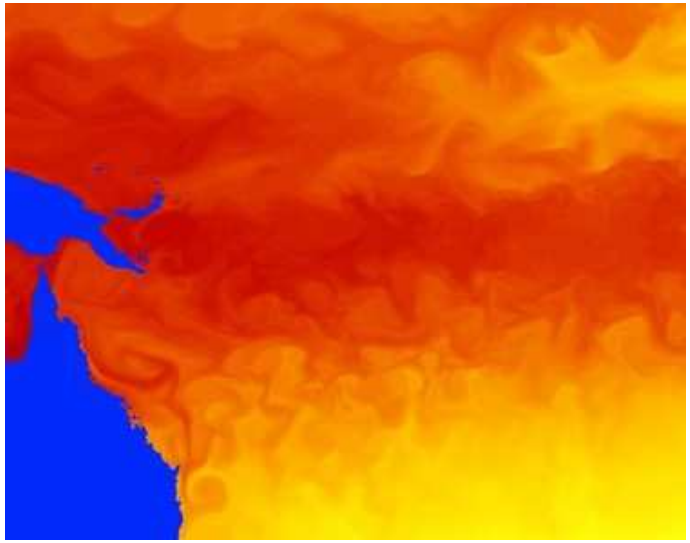
# 1 Talk Overview

- operational weather codes at Australia's Bureau of Meteorology (BoM)
- characteristics of operational weather codes
- case study: the UK Met Office Unified Model
  - typical usage, code configuration & structure, performance analysis
- accelerating the ANUGA tsunami propagation code (work with Zhe Weng)
  - naive approach: CUDA
  - advanced approach: CUDA with relative debugging
  - advanced approach: directive-based (OpenHMPP)
  - results: Merimbula workload, performance & productivity comparison
- acceleration of current weather and climate models
  - current status, challenges and approaches
- conclusions

# 2 Operational Codes at Australia's Bureau of Meteorology

Various codes are needed for regional and global operational forecasts:

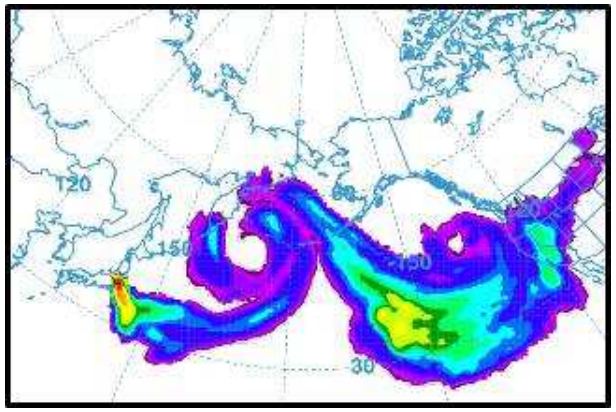
- MOST (Method of Splitting Tsunami): tsunami prediction (NOAA)
- HySplit: air parcel trajectories and dispersion (NOAA and BoM)
- Storm Surge / ROMS (Regional Ocean Modeling System) (UCLA)
- MOM (Modular Ocean Model), currently uses version 4 (NOAA)



- 4DVAR - data assimilation (UK Met Office)
- MetUM:local/global atmosphere model (UK Met Office)
  - currently using the EndGame 'dynamical core'
  - the GungHo dynamical core to be considered by end of decade

# 3 Characteristics of Operational Weather Codes

- very large and complex code bases (almost all Fortran!)
  - require significant expertise in order to configure and run meaningfully
    - often unit testing is not available
    - quality of results crucial but often difficult to determine
      (often use bit-reproducibility)



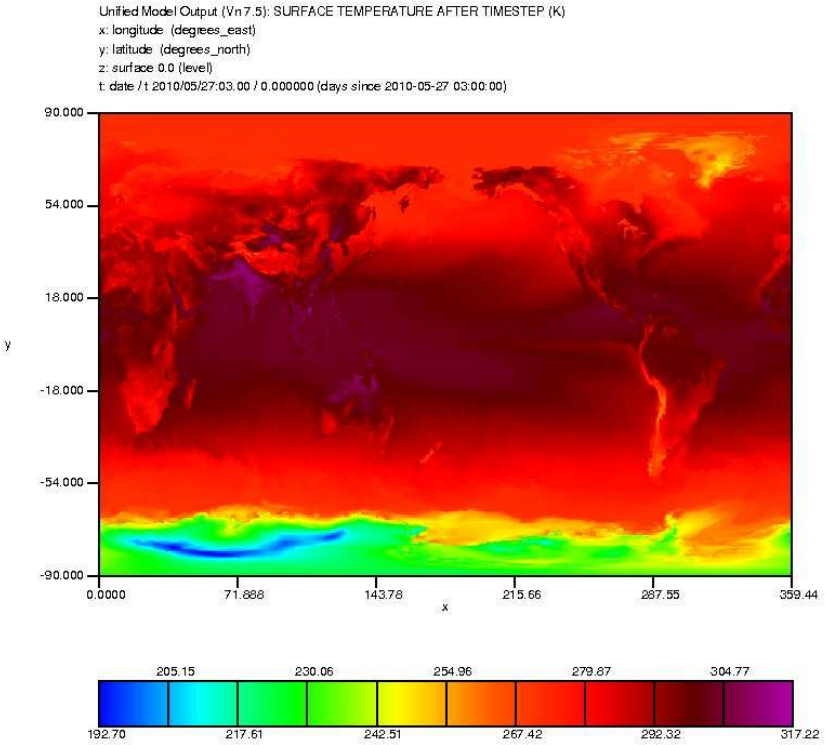
- workloads for accurate forecasts involve very large scale calculations requiring parallel processing
  - typically memory-bound and floating-point intensive
  - halo exchanges are an important operation
  - parallelized for the message-passing (MPI) paradigm
  - some have hybrid (OpenMP) ||ism – unclear if has consistent benefit
- 'port' to accelerators is often incomplete or yet to be attempted

# 4 The Unified Model in Aust. Weather and Climate Simulations

- the Met Office Unified Model (MetUM) is a (global) atmospheric model developed by the UK Met Office from early '90s
- for weather, BoM currently uses a a N320L70 ($640 \times 481 \times 70$) atmosphere grid
  - wish to scale up to a N512L70 ($1024 \times 769 \times 70$) grid
  - operational target: 24 hr simulation in 500s on $<$ 1K cores (10-day 'ensemble' forecasts)
  - doubling the grid resolution increases 'skill' but is $\leq 8\times$ the work!
- climate simulations currently use a use N216L85 grid
  - ACCESS project makes many (long) runs for IPCC
    - common infrastructure: atmosphere: MetUM (96 cores); ocean: NEMO, sea ice: CICE, coupler: OASIS (25 cores)
  - next-generation medium-term models to N320L70
- note: (warped) 'cylindrical' grids are easier to code but problematic ...

# 5 The MetOffice Unified Model (EndGame)

- configuration via UMUI tool creates a directory with (conditionally-compiled) source codes + data files (for a particular grid)
  - no unit tests are available with standard distributions
  - main input file is a 'dump' of initial atmospheric state (1.5GB for N320L70)
  - 'namelist' files for $\approx 1000$ run-time settable parameters
  - in operational runs, periodically records statistics via the STASH sub-system




- partition evenly the EW & NS dimensions of the atmosphere grid on a $P \times Q$ (MPI) process grid

# 6 Unified Model Code Structure and Internal Profiler

- codes in Fortran-90 (mostly F77; $\approx 900$ KLOC) with cpp (include common blocks, commonly used parameter sub-lists (100's!), etc)
- main routine `u_model()`, reads dump file & repeatedly calls `atm_step()`
  - dominated by Helmholtz $P$-$T$ solver (GCR on a tridiagonal linear system)
  - comprises a preconditioner + iterative solver (typically requiring 30–45 iterations)
- internal profiling module can be activated via 'namelist' parameters
  - reports number of calls and totals across all processes, e.g.

| | ROUTINE | MEAN | MEDIAN | SD | % of mean | MAX | ... |
|---|---|---|---|---|---|---|---|
| 1 | PE_Helmholtz | 206.97 | 206.98 | 0.05 | 0.02% | 207.02 | ... |
| 3 | ATM_STEP | 36.39 | 38.53 | 9.46 | 25.99% | 44.60 | ... |
| 4 | SL_Thermo | 25.38 | 26.60 | 3.45 | 13.58% | 31.15 | ... |
| 5 | READDUMP | 24.18 | 24.36 | 1.12 | 4.62% | 24.37 | ... |
| ... | | | | | | | |

  - due to global sync. when a timer starts, can estimate load imbalance

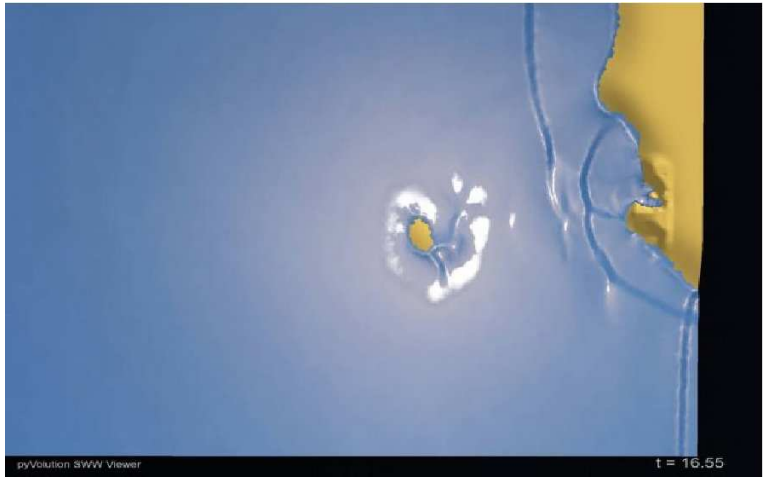
# 7 Individual Subroutine Scalability (N512L70 grid)

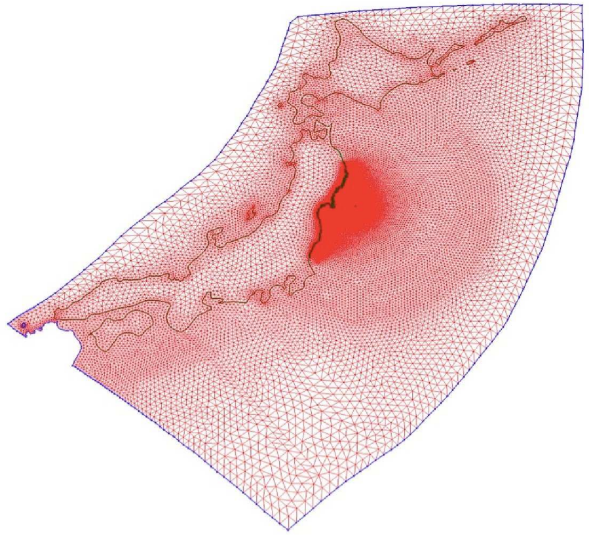
| function | UM7.5+PS24 on 1536 cores | | |
|---|---|---|---|
| | speedup (vs 16 cores) | % of total time | % imbalance |
| PE_Helmholtz | 61.6 | 48 | 0 |
| atm_step | 11.4 | 25 | 20 |
| SL_Full_wind | 49.1 | 13 | 46 |
| SL_Thermo | 53.3 | 9 | 19 |
| Convect | 66.2 | 5 | 37 |
| SF_EXPL | 35.4 | 1 | 75 |
| Atmos_Physics2 | 52.9 | 1 | 28 |
| total: | 55.8 | 100 | 30 |

- data taken on Vayu cluster: two quad-core 2.93 GHz X5570 nodes with single plane QDR Infiniband
- while most time taken in relatively few top-level routines, most of these involve many subroutines!
- the (underestimated) load imbalance + comm. is significant in most
- *extremely* memory intensive: needed enlarged stack, 96 GM /node for small jobs, pinning more memory for large core counts, . . .
- analysis via IPM indicates loads/stores & L1$/L2$/L3$ misses completely accounts for computation time in most routines (0.5 – 2 GLOPs)

# 8 ANUGA Tsunami Propagation and Inundation Modelling

- website: ANUGA; open source: Python, C and MPI
- shallow water wave equation, takes into account friction & bed elevation
  - 2D triangles of variable size according to topography and interest (un-structured mesh)
  - time step determined by triangle size and wave speed
- sim. on 40M cell Tohoku tsunami: super-lin. speedup to 512 cores on K



# 9 ANUGA Code Structure

- computationally-intensive code to be accelerated (5 KLOC Python and 3 KLOC C) is in the `Domain` class
- simplified time evolution loop:

```
def evolve(self, yieldstep, ... )
    ...
    while (self.get_time < self.finaltime):
        self.evolve_one_euler_step(yieldstep, self.finaltime)
        self.apply_fractional_steps()
        self.distribute_to_vertices_and_edges()
        self.update_boundary()
        ...
    ...
def evolve_one_euler_step(self, yieldstep, finaltime):
    self.compute_fluxes()
    self.compute_forcing_terms()
    self.update_timestep(yieldstep, finaltime)
    self.update_conserved_quantities()
    self.update_ghosts()
```

# 10 Acceleration via CUDA: Naive Approach

- profiling the sequential ANUGA via the Python Call Graph module revealed most time spent in 4 C functions (including `compute_fluxes`)
  - suggested strategy of replacing each with one or more CUDA kernels
- use a sub-class of `Domain`, `Basic_GPU_domain`:

```
def evolve(self, yieldstep, ...):
    if self.using_gpu:
        ...
        self.equip_kernel_fns() # load & compile all CUDA kernels
    ... # remaining code similar to original evolve()
def compute_fluxes(self):
    if self.using gpu:
        ... # allocate and copy to GPU the 22 in & in/out fields
        self.compute_fluxes_kernel_function(..., self.timestep)
        self.gravity_kernel_function(...)
        ... # copy from GPU the 5 in/out fields; free GPU fields
        self.flux_time = min(self.timestep)
    else:
        Domain.compute_fluxes(self);
```

# 11 Acceleration via CUDA: Advanced Approach

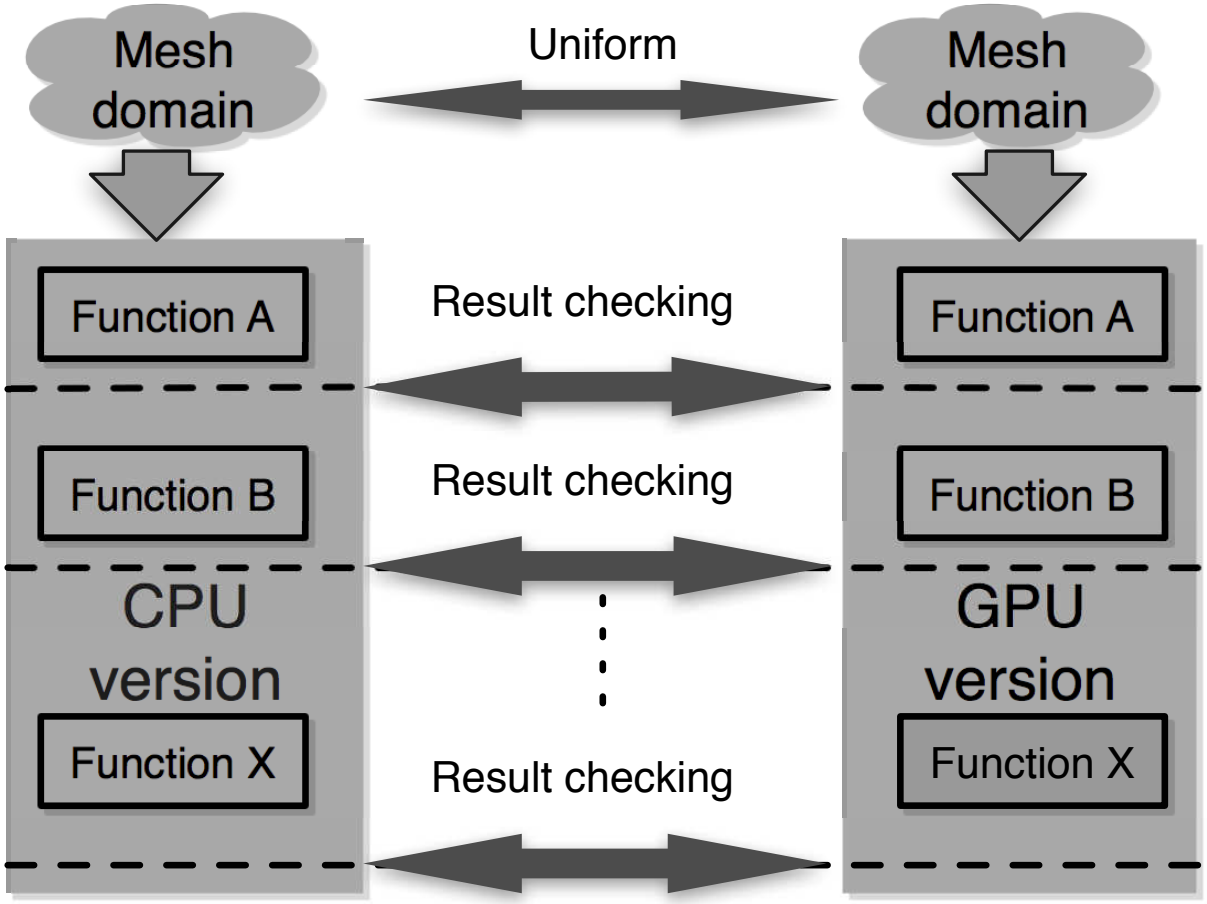
- copying of data on each major function maintained data consistency and permitted incremental ||ization but had very serious overheads!!
- use a sub-class `Advanced_GPU_domain` which:
  - at the beginning of simulation, allocate & copy over all the field data (some 50 vectors) to GPU instead
  - this data is kept on GPU throughout calculation (requires device memory to be large enough to hold all data simultaneously)
  - only as little of the data as required for control purposes copied back to host during simulation (i.e. the `timestep` array)
- memory coalescing and other optimizations were also applied to the kernels
- required *all* Python methods manipulating the fields to be written in CUDA!
  - no longer possible to incrementally add and test kernels!
  - needs a debugging strategy to isolate the faulty kernels

# 12 Background: Relative Debugging

- general debugging technique (Abramson 1995):
  regularly compare the execution of an new (optimized) program against its previously existing reference version
- consists of 4 steps
  1. specify the target data & where in the two programs they should be the same
  2. the relative debugger controls the execution of two programs and compares data at the specified points
  3. upon an error, the developer refines the assertions to isolate the region causing the error
  4. when this region becomes 'small enough' for feasible error location, use traditional debugging methods
- current implementations support large-scale MPI programs, but not devices

# 13 Host-Device Debugging: Overview

- test data fields after each (bottom-level) Python method
- only these methods may manipulate field data



# 14 General Python-Based Relative Debugging Method

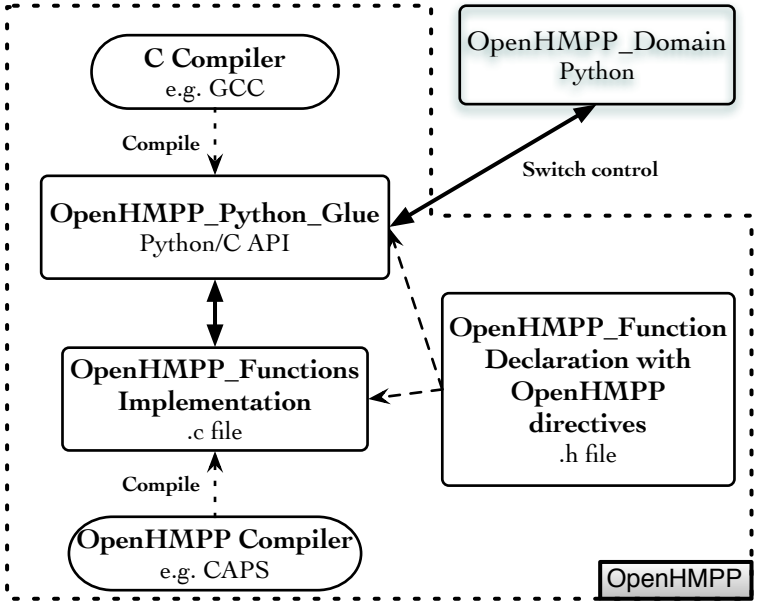
- co-testing in the CUDA implementation

```
def evolve(self, yieldstep, ...):
    if self.using_gpu:
        if self.co_testing:
            self.cotesting_domain = deep_copy(self)
            self.cotesting_domain.using_gpu = False
            self.cotesting_domain.pair_testing = False
        ... # as before
    self.decorate_test_check_point()
    ... # remainder of code is as before
```

- Python decorators used to wrap the original method with one implementing relative debugging and are added to the target function
- successfully identified kernel errors such as incorrect arguments, exceeded thread blocks, overflowed array indices, data-dependency violations etc

# 15 Acceleration via OpenHMPP: Naive Approach

Required the writing of OpenHMPP - Python glue to interface, most of `evolve()` must be done in C




- the sub-class `HMPP_domain` had to be written manually as a C `struct`!
- naive approach simply involved adding directives to transfer fields at call/return and codelet directive to parallelize:

```
#pragma hmpp gravity codelet, target=CUDA &
#pragma hmpp & args[*].transfer=atcall
void gravity(int N, double stageEdgeVs[N], ...
    int k, k3; double hh[3], ...;
#pragma hmppcg gridify(k), private(k3, hh, ...
#pragma hmppcg & global(stageEdgeVs, ...)
    for (k=0; k<N; k++) { k3=k*3; ... }
}
```

# 16 Acceleration via OpenHMPP: Advanced Approach

- field data is marked as mirrored, with a manual transfer

```
#pragma hmpp gravity codelet ..., transfer=atcall, &
#pragma hmpp & args[stageEdgeVs, ...].mirror, &
#pragma hmpp & args[stageEdgeVs, ...].transfer=manual
void gravity(int N, double stageEdgeVs[N], ...){...}
```

- C `evolve()` allocates device data for 67 fields and sets up mirrors on CPU fields (copying at start)

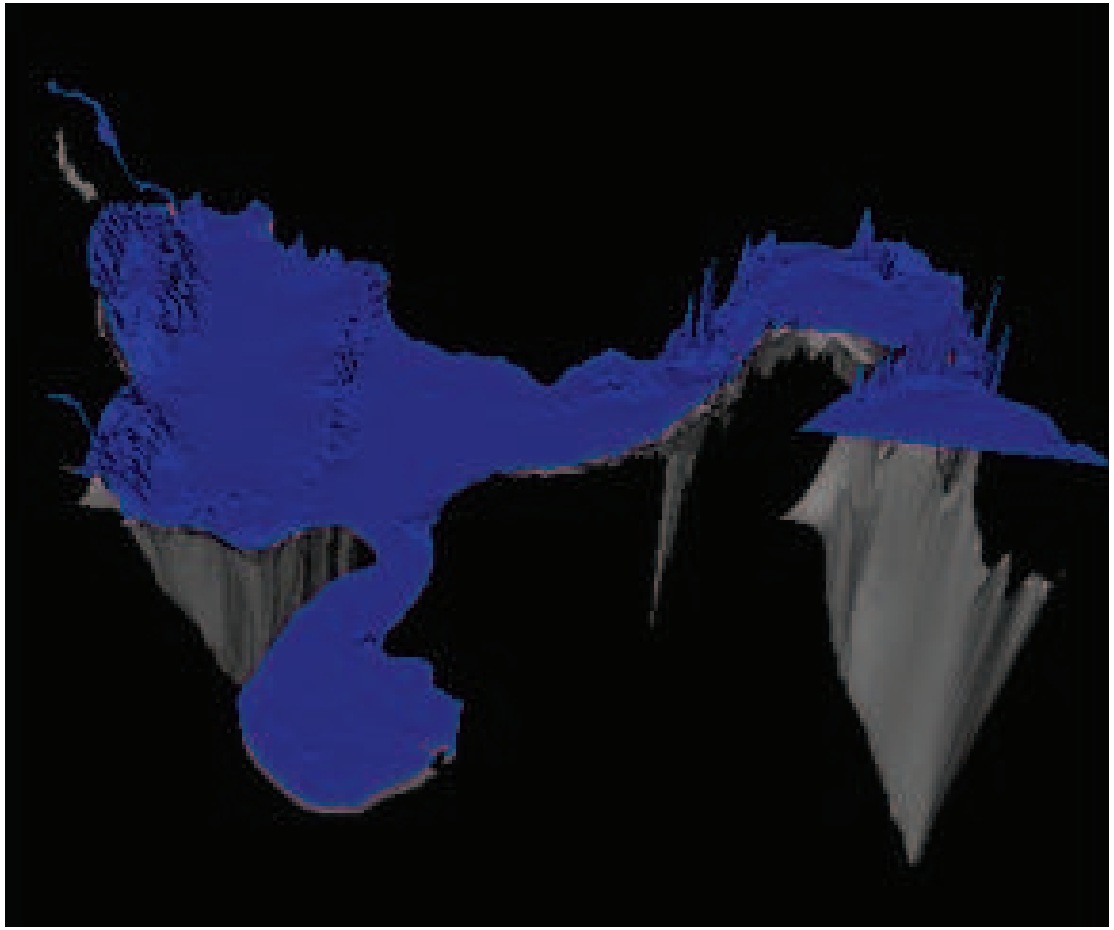
```
double evolve(struct domain *D, ...) {
  int N = D->number_of_elements, ...;
  double *stageEdgeVs = D->stageEdgeVs;
# pragma hmpp cf_central allocate, data[stageEdgeVs], size={N}
  ...
# pragma hmpp advancedload data[..., stageEdgeVs, ...]
```

- this is linked to the callsite where it is (first) used:

```
double compute_fluxes(struct domain *D) {
  int N = D->number_of_elements,
# pragma hmpp cf_central callsite
  compute_fluxes_central(N, ..., D->stage_edge_values, ...);
# pragma hmpp gravity callsite
  gravity(N, D->stage_edge_values, ...); ... }
```

# 17 Results: The Merimbula Workload

- performance tests run on a a real-world dataset with 43,200 grid entities
- inundation simulation of a coastal lake over 300 virtual seconds
- 34s simulation time on a single Phenom core



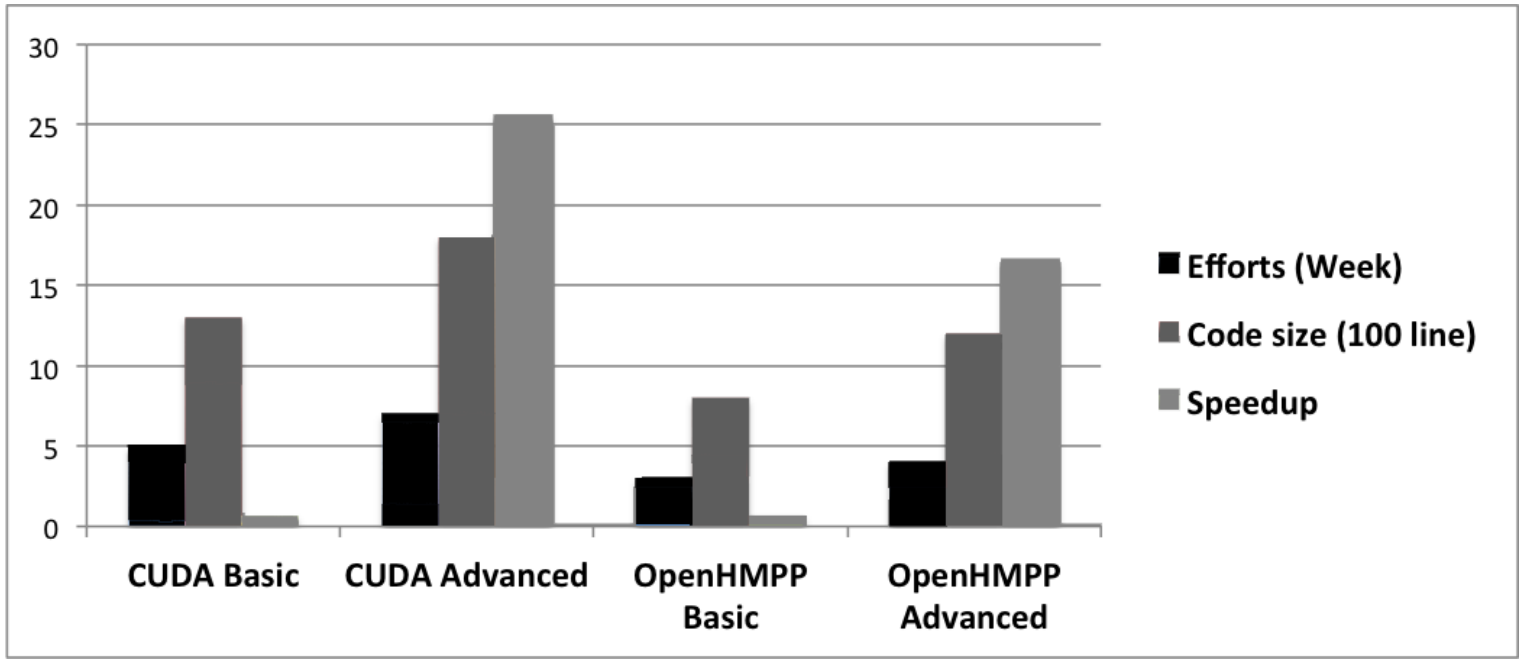
# 18 Results: Speedup

- all results are on a NVIDIA GeForce GTX480 and AMD Phenom(tm) II X4945

| | speedup | | |
|---|---|---|---|
| | naive approach | advanced approach | advanced + rearrangement |
| CUDA | 0.64 | 25.8 | 28.1 |
| OpenHMPP | 0.66 | 16.3 | N/A |
| OpenHMPP | 0.66 | 16.3 | N/A |

- 'rearrangement': kernels rearranged for unstructured mesh coalesced memory accesses
  - on average, this improved kernel execution time by 20%
- PyCUDA prepared call important: without it, advanced approach speedup only 13.3
- kernel-by-kernel comparison CUDA vs OpenHMPP: most of the difference on only 2 of the kernels (including `compute_fluxes()`)

# 19 Results: Productivity




| | | CUDA | OpenHMPP |
|---|---|---|---|
| • speedup of advanced approaches: | per 100 LOC: | 1.4 | 1.4 |
| | per person-week: | 3.5 | 4.0 |

- (implementation by Zhe Weng, no prior experience, CUDA done 1st)

# 20 Lessons from ANUGA

- relatively small code base but still 0.5 year FTE effort
- memory intensivity $\Rightarrow$ unacceptable host-device memory overheads
- keeping data fields on the device $\Rightarrow$ good speedups, CUDA 1.75$\times$ faster
  - however, had to make kernels out of even the minor functions
  - OpenHMPP's data mirroring & asynchronous transfers made this easier

  but OpenHMPP had slightly better productivity:
  - for CUDA, host-device relative debugging was required
    - Python's OO features (sub-classes, deep copy and attribute inspection / iteration) made this relatively easy
    - cannot easily adopt this approach to Fortran codes!
- where incremental ||ization is not possible, some systematic methodology such as host-device relative debugging will be needed

# 21 Status of Acceleration of Weather and Climate Models

- large amount of collaboration worldwide ongoing
  - good progress and results on WRF (US), Cosmo (Switzerland) and NICAM (Japan) atmosphere models
  - OpenACC with Fortran is the dominant approach; CUDA-F popular
  - recent technical developments (CUDA unified memory, NVLink, OpenACC 2.0, Pascal - 3D memory) will play an important role
- for BoM's operational codes:
  - ROMS (Regional Ocean Modeling System): successful CUDA port at CalPoly (2013); status of codes unknown
  - MetUM/GungHo: ongoing work with NVIDIA and STFC (UK); base codes not yet operational
  - MOM (Modular Ocean Model): NOAA/GFDL has interest in an OpenACC implementation
  - MOST (tsunami prediction), HySplit (air parcel trajectories) and MetUM/Endgame: to be started

# 22 Challenges and Approaches in Accelerating Weather Codes

- requires a hybrid, and flexible multi-level parallelization approach
  - e.g. Parallel Ocean Program (POP) sub-blocking
  - ideally wish to partition work over host and (multiple) device cores
  - domain-specific approaches, e.g. the Physis Stencil Framework, may be required
- what general techniques can be developed to improve efficiency
  - halo-exchanges: coalescing, wide halos
  - can memory bandwidth be effectively reduced?
  - will explicit solvers become viable over implicit solvers with accelerators?
- support is needed to accelerate large legacy codes:
  - the Semi-Automated Kernel Generator (KGEN) can support generating and verifying a kernel
  - a (semi-) automated unit test generator is particularly needed for codes like MetUM

# 23 Conclusions

- working with such codes and systems is hard!
  - 'bleeding edge' technology, pushing memory limits, lack of testing infrastructure, cumbersome and huge legacy code systems . . .
- memory intensivity likely requires the port of large proportions of code
  - unlikely that incremental parallelization will lead to satisfactory results
  - directive-based approaches with some 'smarts' seem a promising approach
  - support for testing and verification is also needed
- codes like MetUM have some scaling (communication overhead + load imbalance) issues
  - this makes gaining worthwhile acceleration more difficult!

  As well, memory hierarchy capacity and bandwidth is particularly critical.
- a lot of research (and hard work!) is needed before BoM codes run on accelerated supercomputers!


# THANK YOU

JOIN THE CONVERSATION

#GTC15