Acceleration of a Python-based Tsunami Modelling Application via CUDA and OpenHMPP

Zhe Weng and Peter Strazdins*,
Computer Systems Group,
Research School of Computer Science,
The Australian National University

(slides available from http://cs.anu.edu.au/~Peter.Strazdins/seminars)

The 15th Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2014),
Phoenix, Arizona, 23 May 2014

THE AUSTRALIAN NATIONAL UNIVERSITY

1 Talk Overview

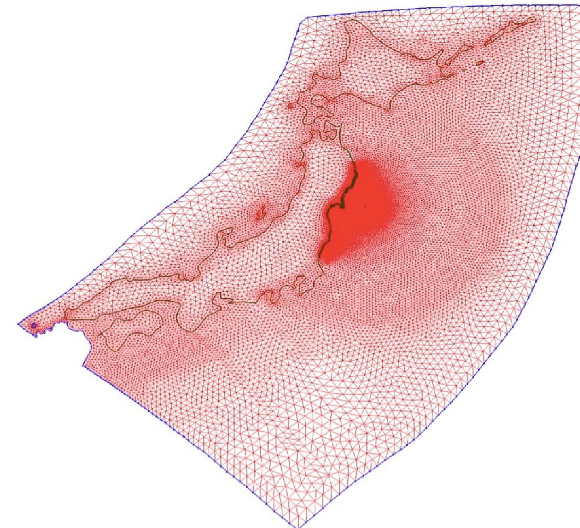
- the Python-based ANUGA Tsunami Modelling Application
  - code structure and challenges to parallelize
- background
  - relative debugging
  - PyCuda and OpenHMPP
- acceleration via CUDA
  - naive & advanced approaches
  - host-device relative debugging via OO support
- acceleration via OpenHMPP
  - naive & advanced approaches
- results: Merimbula workload, performance and productivity comparison
- conclusions and future work

2 Motivation and Contributions

- unstructured meshes – highly efficient technique for irregular problems
  - only perform detailed calculations where needed
  - however, code becomes complex and the computation memory-intensive
    - indirect data accesses required, fewer FLOPs/ memory access
- debugging on devices is hard! (have less support than on host)
  - usually require host $\leftrightarrow$ device data transfers, themselves error-prone
- contributions: for Python/C based unstructured mesh applications:
  - show how to effectively port for kernel-based and directive-based paradigms
    - minimizing memory transfers is the key!
    - in the former, show how to overcome inherent debugging difficulties: host-device form of relative debugging
    - relatively easy implementation with OO support
  - comparison of their performance and productivity
  - code available at http://code.google.com/p/anuga-cuda

3 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
  - time step determined by triangle size and wave speed
- sim. on 40M cell Tohoku tsunami: super-lin. speedup to 512 cores on K



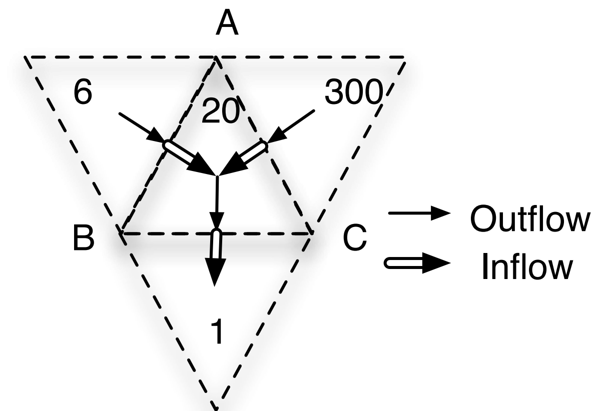
4 ANUGA Code Structure

- 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()
```

5 ANUGA: Parallelization Issues

- `compute_fluxes()` calculates the flux values on the edge of triangles
- a characteristic calculation in unstructured grid applications
- sequential algorithm optimization: the inflow = - outflow (of corresponding edge of neighbor)
  - creates a significant saving, but a subtle inter-iteration dependency
  - will cause error with naive parallelization



6 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, 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

7 Background: PyCuda and OpenHMPP

- PyCuda: a seamless Python/CUDA API
  - allows full access to CUDA, invoking kernels in the normal way
  - runtime needs to work out types of arguments
    - can be avoided via specifying via prepared invocations
- OpenHMPP: a directive-based programming standard to specify computations to be offloaded to devices and data transmission
  - these are kernel-based (placed beside functions)
  - automatic (slow) or manual (fast but error-prone) data transfer strategy
  - also have codelet generator directives: enhance code generation (loop-based)

8 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`:

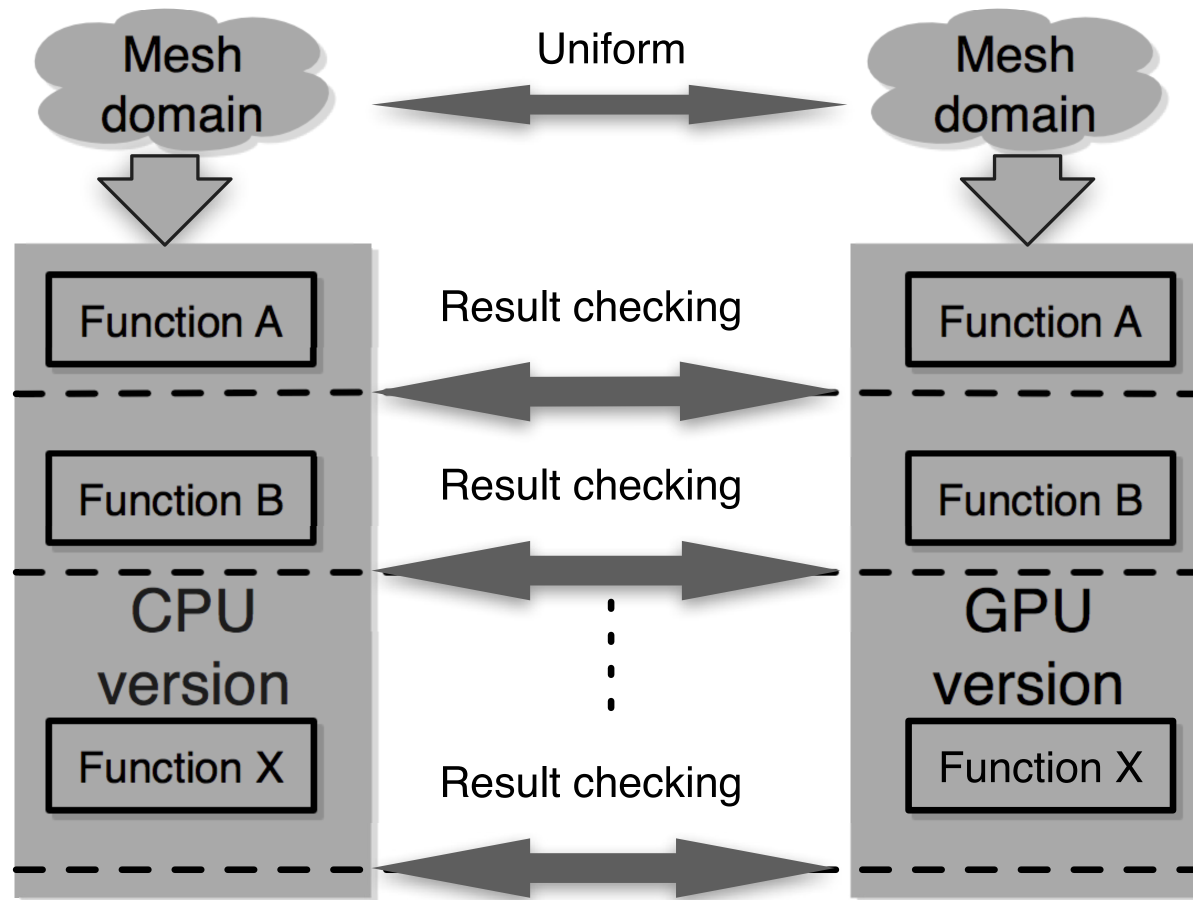
```
def evolve(self, yieldstep, ...):
    if self.using_gpu:
        ...
        self.equip_kernel_fns() # load & compile all CUDA kernels
    ... # remaining code similar to original evolve()
def compute_fluxes(self):
    if self.using gpu:
        ... # allocate and copy to GPU the 22 in & in/out fields
        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);
```

9 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

10 Host-Device Debugging: Overview

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



11 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:

```
def decorate_check_point(self, check_input=False):
    for name, fn in self.iter_attributes():
        if callable(fn) and hasattr(fn, ‘‘checkpointable’’) :
            fn.__func__.original = getattr(Domain, name)
            self.add_check_point(name, fn, check_input)
```

12 General Python-Based Relative Debugging Method (II)

- finally we add the Python decorators to the target function

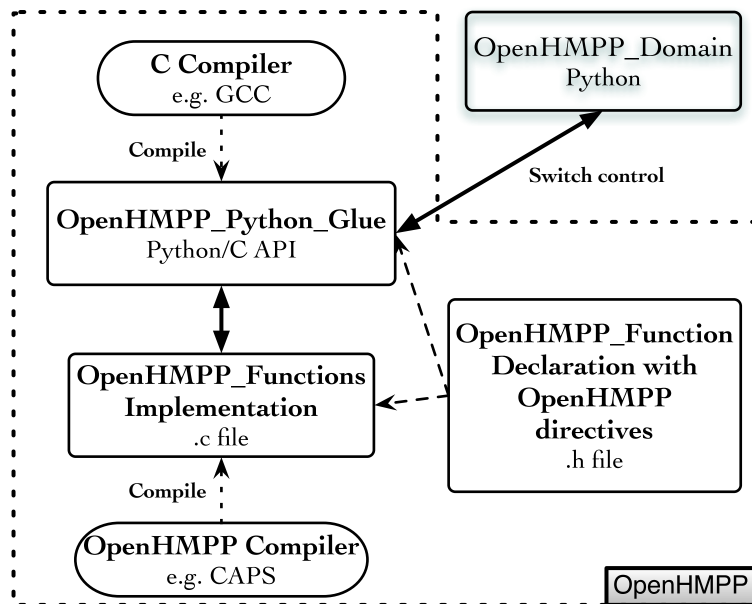
```
def add_check_point(self, name, fn, check_input=False):
    def check_point(*args, **kv):
        fn(*args, **kv) # invoke target function
        # now invoke reference function
        fn.original(self.cotesting_domain, *args, **kv)
        self.check_all_data(name)

    if self.cotesting == True:
        setattr(self, name, check_point)
```

- also need to manually update control variables, i.e. the current time, in the reference mesh
- also can check data before call, to determine if error in the Python work-flow
- successfully identified kernel errors such as incorrect arguments, ex-ceeded thread blocks, overflowed array indices, data-dependency viola-tions etc

13 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; ... }
}
```

14 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)

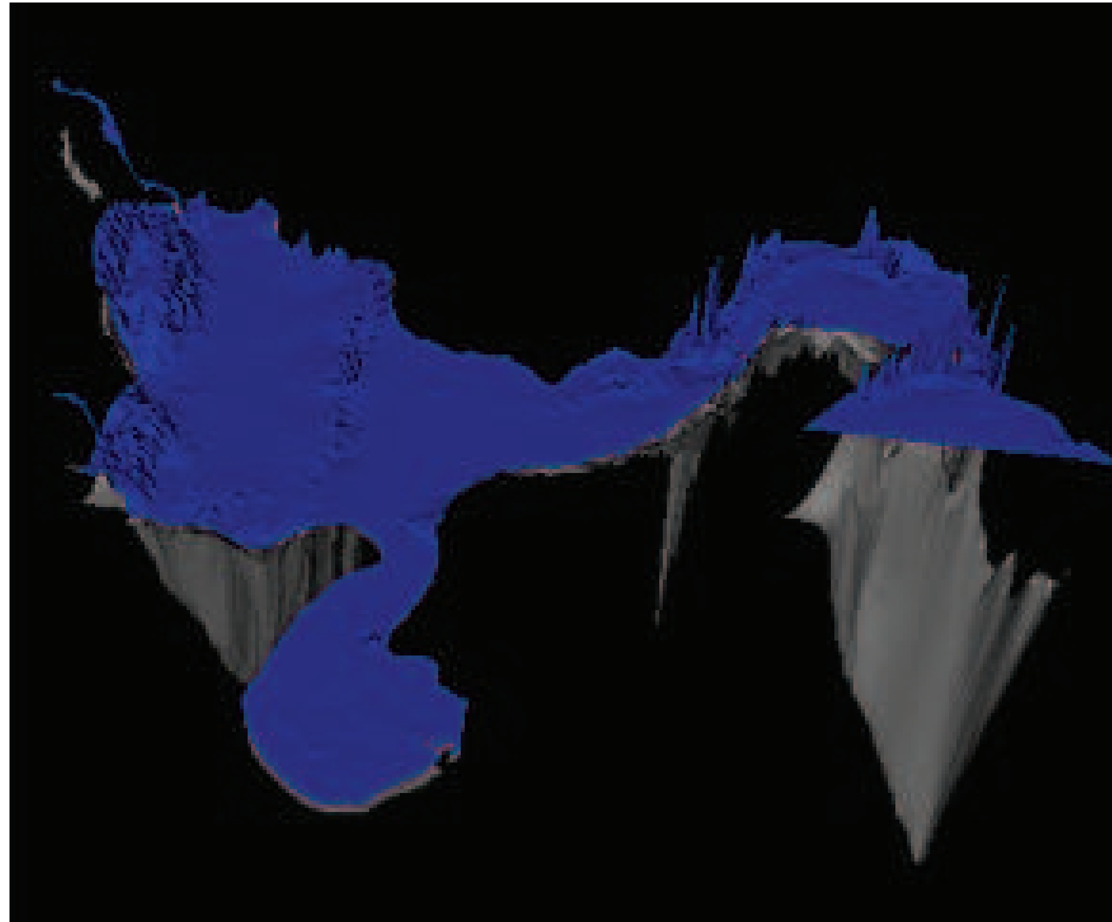
```
double evolve(struct domain *D, ...) {
  int N = D->number_of_elements, ...;
  double *stageEdgeVs = D->stageEdgeVs;
# pragma hmpp cf_central allocate, data[stageEdgeVs], size={N}
  ...
# pragma hmpp advancedload data[..., stageEdgeVs, ...]
```

- 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, ...); ... }
```

15 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



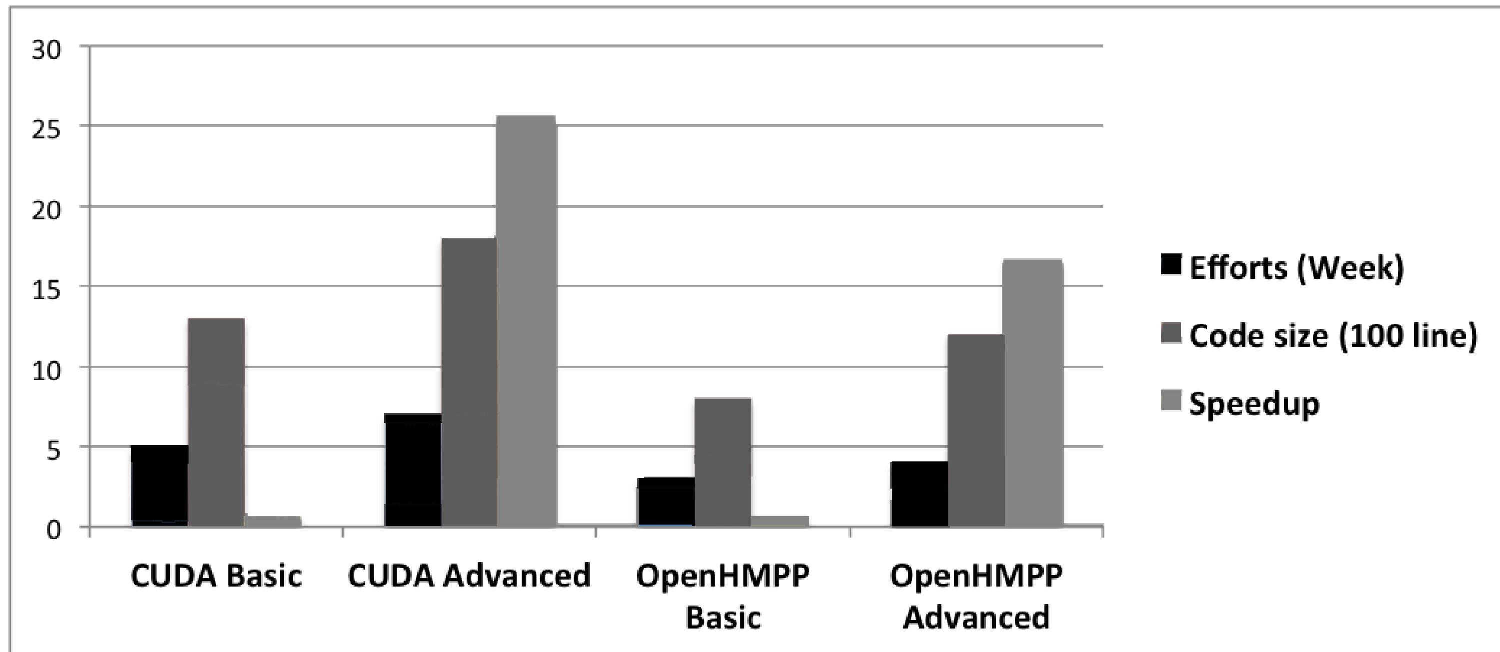
16 Results: Speedup

- 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()`)

17 Results: Productivity



- speedup of advanced approaches:

	CUDA	OpenHMPP
per 100 LOC:	1.4	1.4
per person-week:	3.5	4.0

- (implementation by 1st author, no prior experience, CUDA 1st)

18 Conclusions

- ANUGA is a sophisticated and complex unstructured mesh application
- easy to port naively but unacceptable host-device memory overheads
- keeping data fields on the device $\Rightarrow$ good speedups, CUDA 1.75$\times$ faster
  - optimizations to coalesce memory accesses was important to CUDA
  - OpenHMPP's data mirroring & asynchronous transfers made this easy

  but OpenHMPP had slightly better productivity:
  - PyCUDA made this easy, but prepared call technique necessary
  - lack of similar OpenHMPP interface $\Rightarrow$ more code being ported to C
  - for CUDA, host-device relative debugging was required
    - Python's OO features (sub-classes, deep copy and attribute inspection / iteration) made this relatively easy
    - approach is generic and can be adopted by other applications
- where incremental $||$ation is not possible, host-device relative debugging shows promise!

Thank You!! ...Questions???