Case Studies in the GPU Acceleration of Two Earth Sciences Applications

Peter Strazdins,
Computer Systems Group,
Research School of Computer Science,
The Australian National University


MSI Special Year 2019 in Computational Mathematics
Challenges in High Performance Computing workshop
The Australian National University, 02 Sep 2019
1 Alternate Title: A Tale of Two Honours Students

• an introduction to Graphics Processing Units
  (excerpts from the Parallel Systems Graphic Processing Units lecture)
• part 1: accelerating the ANUGA tsunami model (Zhe Weng)
  (thanks also to Steve Roberts, ANU)
• part 2: accelerating the HYSPLIT air concentration model (Fan Yu)
  (thanks also to Joerg Henrichs and Tim Pugh, Bureau of Meteorology)
Part 1 Overview: Accelerating the ANUGA Tsunami Modelling Application

- 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
3 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 ↔ 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
4 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
5 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:
  
  ```python
  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()
  ```
6 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 neighbour)
  - Creates a significant saving, but a subtle inter-iteration dependency
- Will cause error with naive parallelization
7 Background: Relative Debugging

- general debugging technique (Abramson 1995): regularly compare the execution of a 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
8  **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)
9 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:
  
  ```python
  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);
  ```
10 Acceleration via CUDA: Advanced Approach

- copying of data on each major function maintained data consistency and permitted incremental update 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
11 Host-Device Debugging: Overview

- test data fields after each (bottom-level) Python method
- only these methods may manipulate field data
12 General Python-Based Relative Debugging Method

• **co-testing in the CUDA implementation**

```python
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()
    ... # as before
```

• **Python decorators used to wrap the original method with one implementing relative debugging:**

```python
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)
```
General Python-Based Relative Debugging Method (II)

• **finally we add the Python decorators to the target function**

```python
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 workflow

• successfully identified kernel errors such as incorrect arguments, exceeded thread blocks, overflowed array indices, data-dependency violations etc
14 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:

```c
#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; ...
    }
}
```

C Compiler
e.g. GCC

OpenHMPP_Python_Glue
Python/C API

OpenHMPP_Functions
Implementation
.c file

OpenHMPP_Domain
Python

Switch control

OpenHMPP_Compiler
e.g. CAPS

OpenHMPP_Function
Declaration with
OpenHMPP directives
.h file
15 Acceleration via OpenHMPP: Advanced Approach

- field data is marked as mirrored, with a manual transfer
  
  ```c
  #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)
  
  ```c
  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:
  
  ```c
  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, ...); ... }
  ```
Results: The Merimbula Workload

- performance tests run on 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
17 Results: Speedup

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

<table>
<thead>
<tr>
<th></th>
<th>naive approach</th>
<th>speedup advanced approach</th>
<th>advanced + rearrangement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>0.64</td>
<td>25.8</td>
<td>28.1</td>
</tr>
<tr>
<td>OpenHMPP</td>
<td>0.66</td>
<td>16.3</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- ‘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()`)
18 Results: Productivity

- speedup of advanced approaches: per 100 LOC: 1.4 1.4
  per person-week: 3.5 4.0

- (implementation by 1st author, no prior experience, CUDA 1st)
Part 1: 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
  - OpenHMPP had slightly better productivity:
    - PyCUDA made this easy, but the 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 incrementation is not possible, host-device relative debugging shows promise!
Part 2 Overview: Accelerating the HYSPLIT Air Concentration Model

- background: the HYSPLIT application
- shared memory parallelization
  - program analysis and particle loop refactorization
  - OpenMP parallelization
  - retaining bit reproducibility
- GPU parallelization
  - naive approach; coarse grain parallelization
- performance
  - OpenMP
  - CUDA – naive and course-grain parallelization
  - coding effort
- conclusions and future work
21 The HYSPLIT Application

- the Hybrid Single-Particle Lagrangian Integrated Trajectory model is a 4th-generation model from NOAA’s ARL over 30 years; has 1000’s users
- computes air parcel trajectories as well as transport dispersion, deposition and chemical transformation
- requires gridded meteorological data on a latitude-longitude grid
- has MPI parallelization
  - only suitable for coarse-grain process-level parallelization
The HYSPLIT Application (II)

- has two primary models: air trajectories and air concentration; the latter is more computationally intensive – we concentrate on this
  - pollutants released at certain positions and times are modelled by particles or puffs
  - on each time step, it iterates over all the particles or puffs and performs 5 steps to each: advection, interpolation, dispersion, deposition and concentration calculation
- the model calculates the distribution of these pollutants according to meteorological data (e.g. wind speed) and the properties of the pollutants
Shared Memory Parallelization: Particle Loop Refactoring

- from program profiling, > 80% execution time was spent in the interpolation, dispersion and advection steps inside a particle loop
- this logically *embarrassingly parallel* loop has two features which prohibit parallelization:
  - sub-grid re-load: conditional on sub-grids loaded from a previous iteration
  - the concentration calculation: many irregular data dependencies
24 Particle Loop Refactoring (II)

- we need to split the particle loop into a serial loop (sub-grid re-load, plus other file I/O operations), a parallel loop followed by a serial loop (concentration)
  - as the sub-grid re-load, advection and interpolation steps were originally in a subroutine, this had to be first in-lined
- further problem: a variable $A$ set for particle $i$ in the first serial loop must have the same value for particle $i$ in the other loops
  - solution: duplicate every such variable with an array across all particles, replace references to $A$ with $A[i]$
  - these are set in the first loop
- the meteorological sub-grids similarly need to be duplicated
  As these are a few MB each, only copies of the differing sub-grids are stored, rather than one for each particle
25 OpenMP Parallelization

- after all the above refactoring work, the three sub-loops can be safely executed one-by-one
- now an !$OMP PARALLEL DO can be safely inserted just before the parallel sub-loop
- the application can be just as trivially parallelized using Pthreads etc
26 Retaining Bit Reproducibility

- this is extremely important in practice for operational codes!
- the dispersion step requires one Gaussian random number generated per particle
  These are similarly generated by the first loop and be saved in an per-particle array
- for the CUDA implementation, we compile with the \texttt{-emu} flag to ensure IEEE-standard results for \texttt{exp()} and \texttt{sqrt()}
27 GPU Parallelization: Naive Approach

- the fundamental parallelization techniques used for shared memory are also the basis for this approach
  - allocate GPU threads in the place of OpenMP threads
  - a single kernel for advection and interpolation steps, another for dispersion
  - the deposition step on the GPU showed a low (40%) warp efficiency and a branch divergence of 18%, with a relative slowdown of 1.85

- thus the approach has the following steps:
  - transfer input data (including the set of meteorological sub-grids) to the GPU
  - invoke the two kernels to the GPU
  - transfer output data from the GPU
  - call the deposition routine with an optionally parallel (OpenMP) loop
Coarse Grain GPU Parallelization Approach

- idea: reduce the host-device memory overhead and poor GPU utilization by overlapping different kernel invocations
- can be easily adopted into the MPI and OpenMP parallelizations of HYSPLIT
- need only change the kernel invocation, to incorporate streaming
29 Coarse Grain GPU Parallelization Approach (II)

- 3 approaches are used
  - single-thread approach: a single CPU thread assigns tasks to different CUDA streams in a round-robin fashion
  - multi-thread approach: each CPU thread assigns tasks to its own stream
  - multi-process approach: each MPI process assigns tasks to its own stream.
    This uses NVIDIA’s Multi-Process Service (MPS) to share a single CUDA context between the MPI processes
- a parallel OpenMP loop is used for deposition


30 **Performance Results: Configuration**

- used a standard test case (called ‘jexanple’, with 30K particles) provided by the Bureau of Meteorology
- test each of the OpenMP, CUDA naive, and CUDA coarse-grained versions
  - CUDA versions used 32 threads per block
- run on two different machines:
  - A 8-core 3.3GHz AMD FX-8300 with one NVIDIA GeForce GTX960 GPU (Maxwell, 1024 cores)
  - B 6-core 2.6GHz Intel Broadwell E5-2650v4 with one NVIDIA Pascal P100 (3584 cores)
- each CUDA course-grain approach divides the single kernel and its corresponding data transfer into 1,2,4,8(6) CUDA streams
### Performance Results: OpenMP Version

![Bar chart](chart.png)

- **Legend**:
  - Black: Rest of running time outside particle loop
  - Yellow: Rest of running time inside particle loop
  - Green: Serial part #1 running time
  - Red: Serial part #2 running time
  - Purple: Parallel part running time
  - Machine A
  - Machine B

- **Experiment configuration**:
  - Original Version
  - OpenMP 1 thread
  - OpenMP 2 threads
  - OpenMP 4 threads
  - OpenMP 8(6) threads

(left bar on machine A, right on B)
### Performance Results: OpenMP Version (II)

<table>
<thead>
<tr>
<th></th>
<th>Machine A</th>
<th>Machine B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>whole program</td>
<td>parallel part</td>
</tr>
<tr>
<td>Speedups:</td>
<td>actual</td>
<td>theor.</td>
</tr>
<tr>
<td>Original</td>
<td>1.00×</td>
<td>1.00×</td>
</tr>
<tr>
<td>OMP-1-T</td>
<td>0.83×</td>
<td>1.00×</td>
</tr>
<tr>
<td>OMP-2-T</td>
<td>1.53×</td>
<td>1.82×</td>
</tr>
<tr>
<td>OMP-4-T</td>
<td>2.77×</td>
<td>3.08×</td>
</tr>
<tr>
<td>OMP-8(6)-T</td>
<td>4.01×</td>
<td>4.71×</td>
</tr>
</tbody>
</table>

- on a single thread, 20% (A) and 13% (B) slower than original program
  - due to extra overhead in 1st serial loop (maintain data dependencies)
- on B, speed-up on 6 threads vs 1 thread is only 4.93× for the parallel part
  - possibly due to the Intel Xeon’s turbo boost
Performance Results: Naive CUDA Version

center: 1 CPU thread (3.73× (A), 4.00× (B) speedup),
right: 8(6) CPU threads (4.09× (A), 4.77×(B) speedup)
for the deposition step
34 Results: Coarse-grain CUDA Version, Machine A

- note: for the single- and multi-thread approaches, times for data transfer and kernel execution cannot be distinguished
35 Results: Coarse-grain CUDA Version, Machine B

![CUDA Coarse-Grained Parallel Version HYSPLIT Performance By Component On Machine B](image)

- note: the OpenMP multi-thread approach failed to run here

**For single-thread version, this component is the aggregated time for concurrent kernels and data transfer**
Performance Results: Coarse-grain CUDA Version

- single- and multi-thread approaches show a small improvement, probably due to hiding of data transfer time
- the multi-process approach also shows a drop in kernel execution time
  - may be due to MPS’s context funnelling, which can merge kernels from independent processes
  - note also MPI version of HYSPLIT allows the kernel execution to overlap with the CPU deposition computation in another process
- on machine A (B), we see a best speedup of $2.16 \times (2.70 \times)$ over the naive CUDA
37 Results: Coding Effort

- number of lines of source code changed or created for the two versions:

<table>
<thead>
<tr>
<th>code category</th>
<th>OpenMP</th>
<th>CUDA</th>
<th>difficulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>main program</td>
<td>706</td>
<td>719</td>
<td>medium</td>
</tr>
<tr>
<td>Parloop &amp; its isolation</td>
<td>815</td>
<td>942</td>
<td>high</td>
</tr>
<tr>
<td>parallelization barrier removal</td>
<td>457</td>
<td>490</td>
<td>high</td>
</tr>
<tr>
<td>interfaces (for parallel programs)</td>
<td>706</td>
<td>811</td>
<td>low</td>
</tr>
<tr>
<td>device data (allocate, transfer)</td>
<td>−</td>
<td>480</td>
<td>low</td>
</tr>
<tr>
<td>device kernel</td>
<td>−</td>
<td>292</td>
<td>medium</td>
</tr>
<tr>
<td>device subprogram</td>
<td>−</td>
<td>1127</td>
<td>mostly low</td>
</tr>
</tbody>
</table>

- the main non-trivial work is in the removal of dependencies. Complexity & subtlety of the original code makes this a substantial effort!

- the CUDA version requires significantly more changes, due to:
  - GPU memory management
  - incompatibilities with CUDA Fortran
38 Part 2: Conclusions

- HYSPLIT’s particle loop was the principal target for parallelization
  - barriers included particle-dependent I/O and variables
  - also concentration calculation, due to a high degree on irregular dependencies
- significant refactoring required; introduced a 10-15% serial overhead
  - once done, the OpenMP parallelization was trivial and showed good parallel speedup
  - to retain bit reproducibility, similar refactoring was required
- GPU implementation was similarly based on the refactored code
  - the deposition step created significant divergence and was left on the CPU
  - yielded 4–5× speedup (best with multiple CPUs on deposition)
Part 2: Conclusions (II) and Future Work

• coarse-grained GPU parallelization with MPI processes gained $2\times$–$3\times$ further speedup

• coding effort analysis showed extensive non-trivial changes required
  • CUDA version nearly doubles the number of changes, although these are mostly less trivial

• possible directions for future work
  • OpenACC or device-aware OpenMP version (performance vs code-base impact)
  • an extension to multiple GPUs (particularly useful with MPI+CUDA)
Thank You!!

...Questions???

(email peter at cs.anu.edu.au)