Shared Memory and GPU Parallelization of an Operational Atmospheric Transport and Dispersion Application

Fan Yu*, Peter E. Strazdins*, Joerg Henrichs†, Tim F. Pugh†

*:Computer Systems Group,
Research School of Computer Science,
The Australian National University
†: Bureau of Meteorology, Melbourne, Australia

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

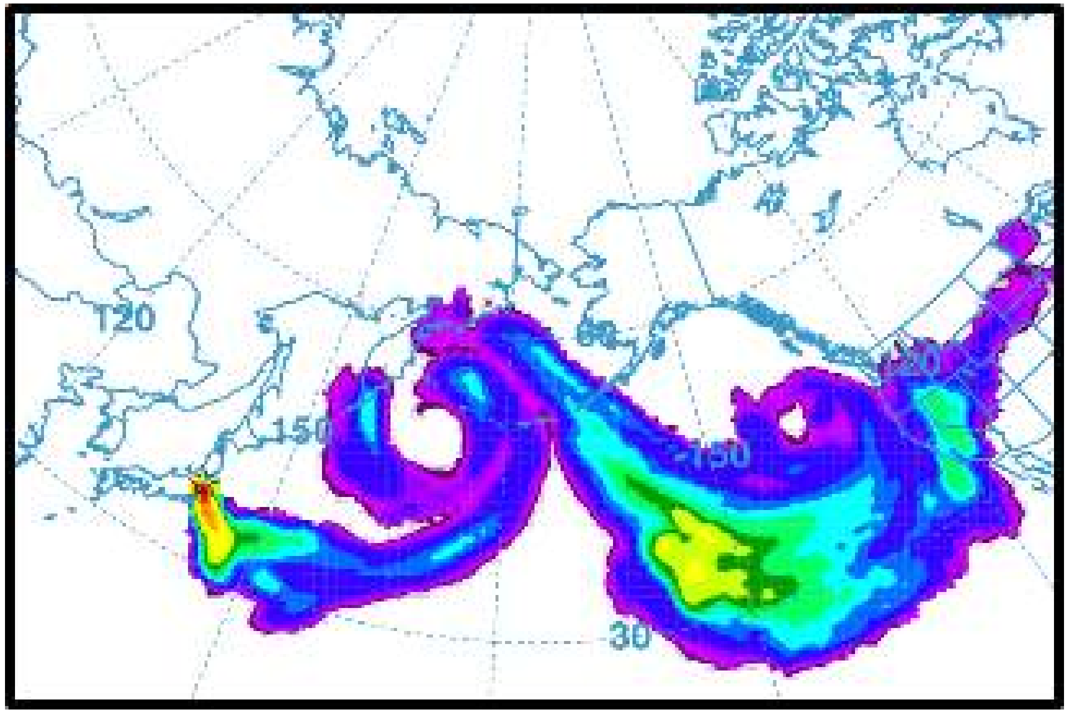
The 20th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing,
Rio de Janeiro,
24 May 2019

THE AUSTRALIAN NATIONAL UNIVERSITY

1 Talk Overview

- 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

2 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

3 The HYSPLIT Application (II)

- has two primary models: air trajectories and air concentration;

  the latter is more computationally intensive – we concentrate on this

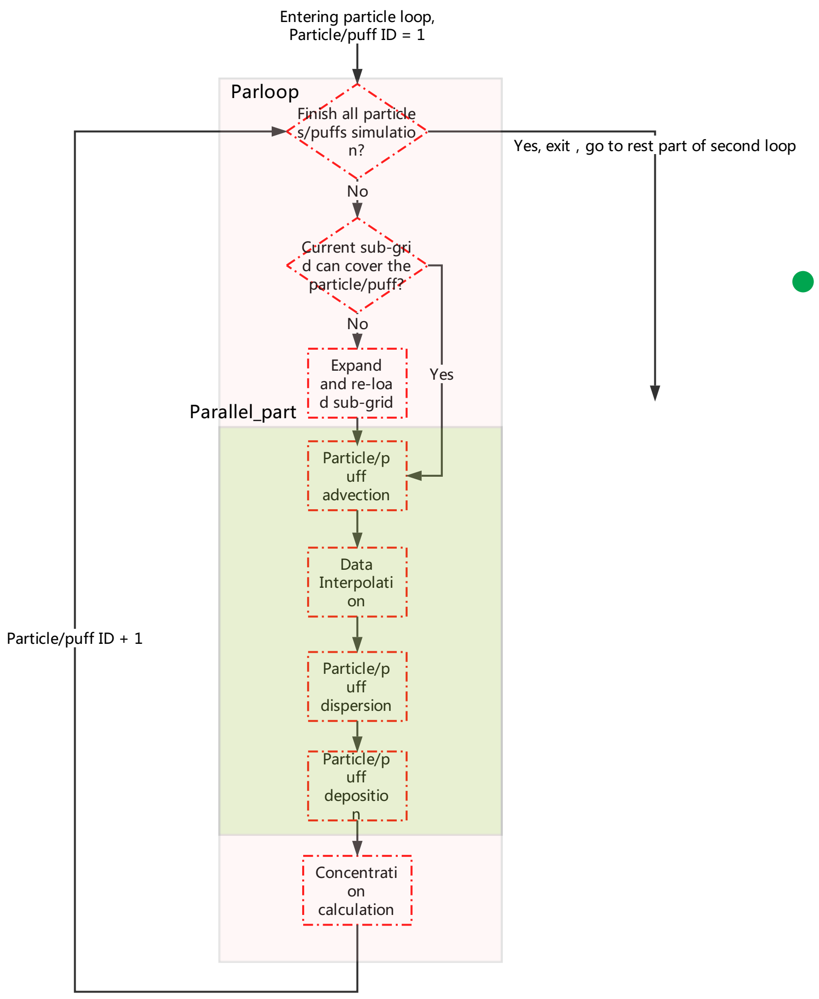
  - pollutants released at certain positions and times are modelled by particles or puffs
  - on each time step, it iterates over all the particles or puffs and performs 5 steps to each:

    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

4 Shared Memory Parallelization: Particle Loop Refactoring


Entering particle loop,
Particle/puff ID = 1
Parloop
Finish all particle s/puffs simulatio n?
Yes, exit , go to rest part of second loop
No
Current sub-gri d can cover the particle/puff?
No
Yes
Expand and re-loa d sub-grid
Parallel_part
Particle/p uff advection
Data Interpolati on
Particle/puff ID + 1
Particle/p uff dispersion
Particle/p uff depositio n
Concentrati on calculation

- 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

5 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

6 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

7 Retaining Bit Reproducibility

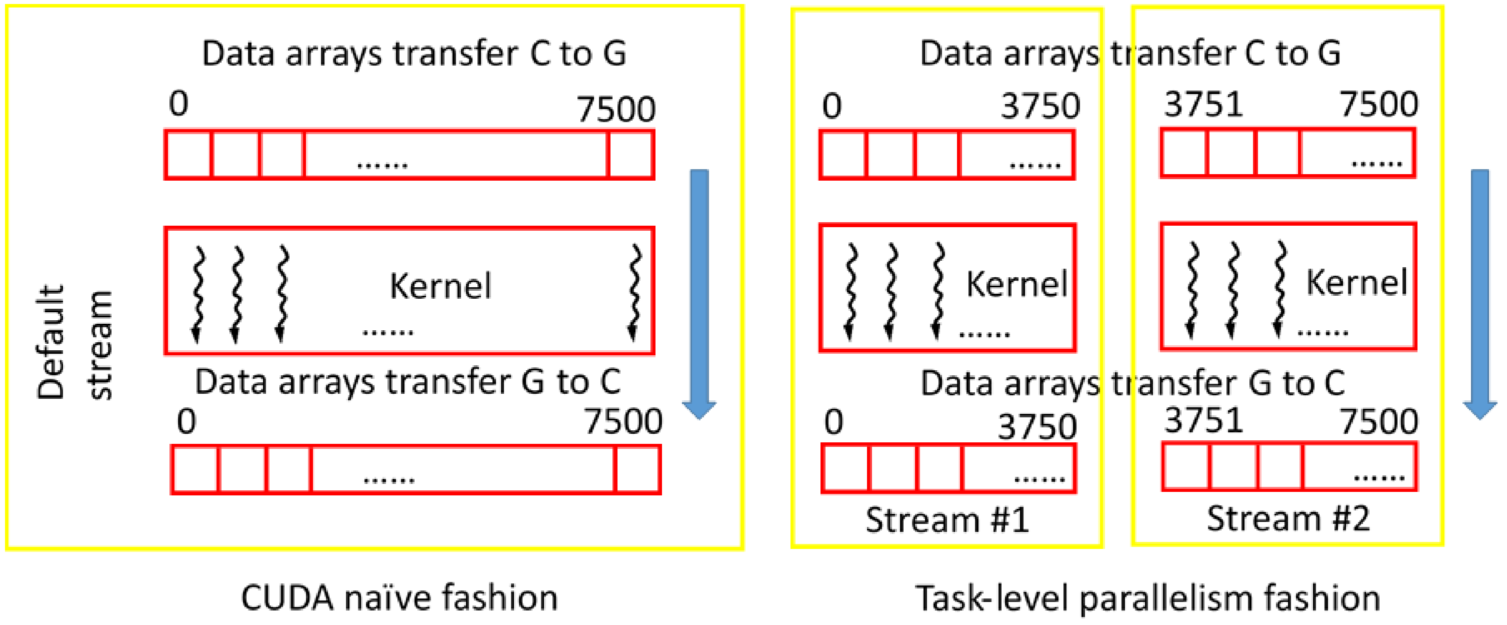
- this is extremely important in practice for operational codes!
- the dispersion step requires one Gaussian random number generated per particle

  These are similarly generated by the first loop and be saved in an per-particle array
- tor the CUDA implementation, we compile with the `-emu` flag to ensure IEEE-standard results for `exp()` and `sqrt()`

8 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

9 Coarse Grain GPU Parallelization Approach


Data arrays transfer C to G
0
7500
……
Kernel
……
Data arrays transfer G to C
0
7500
……
Default stream
CUDA naïve fashion
Data arrays transfer C to G
0
3750
3751
7500
Kernel
Data arrays transfer G to C
Stream #1
Stream #2
Task-level parallelism fashion

- 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

10 Coarse Grain GPU Parallelization Approach (II)

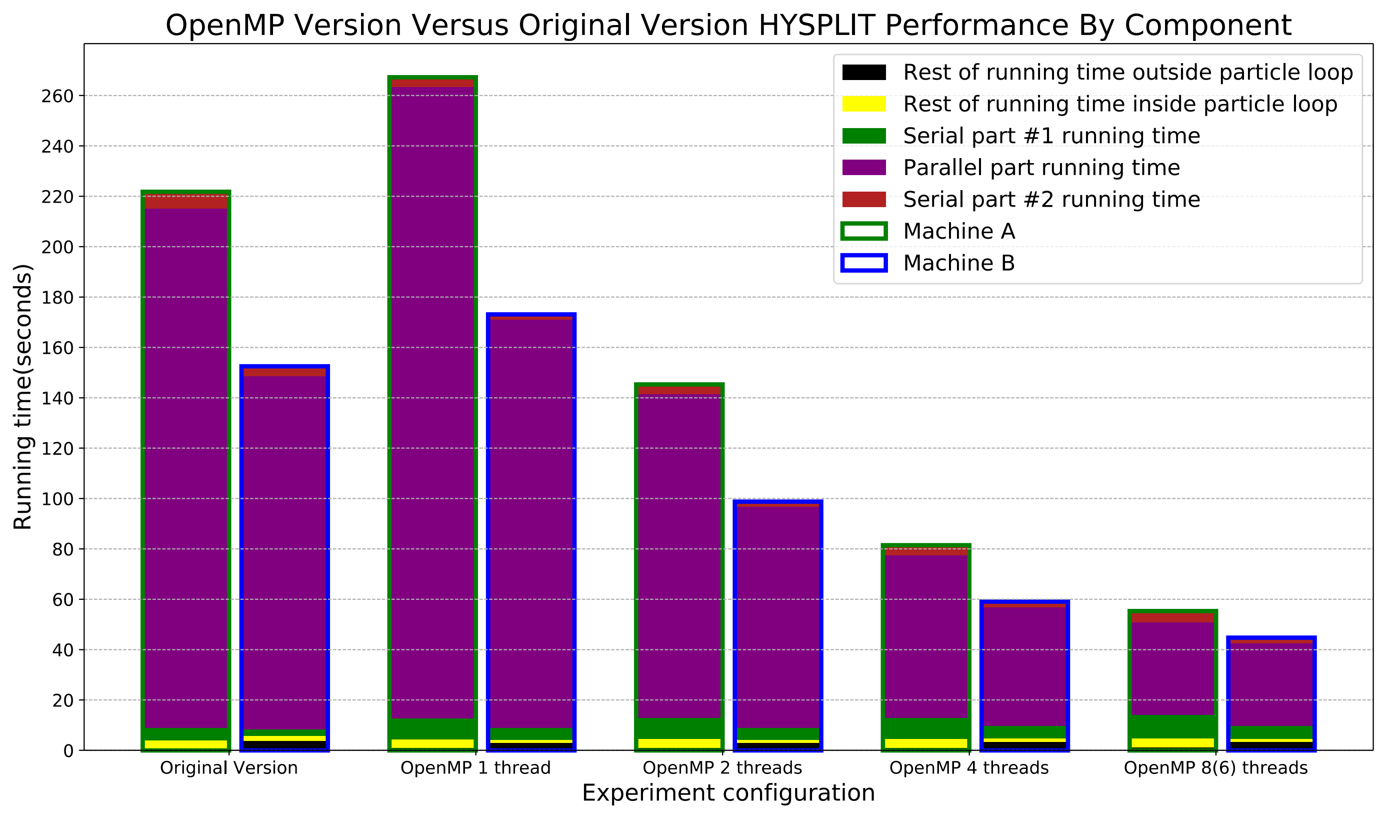
- 3 approaches are used
  - single-thread approach: a single CPU thread assigns tasks to different CUDA streams in a round-robin fashion
  - multi-thread approach: each CPU thread assigns tasks to its own stream
  - multi-process approach: each MPI process assigns tasks to its own stream.

    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

11 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

12 Performance Results: OpenMP Version



(left bar on machine A, right on B)

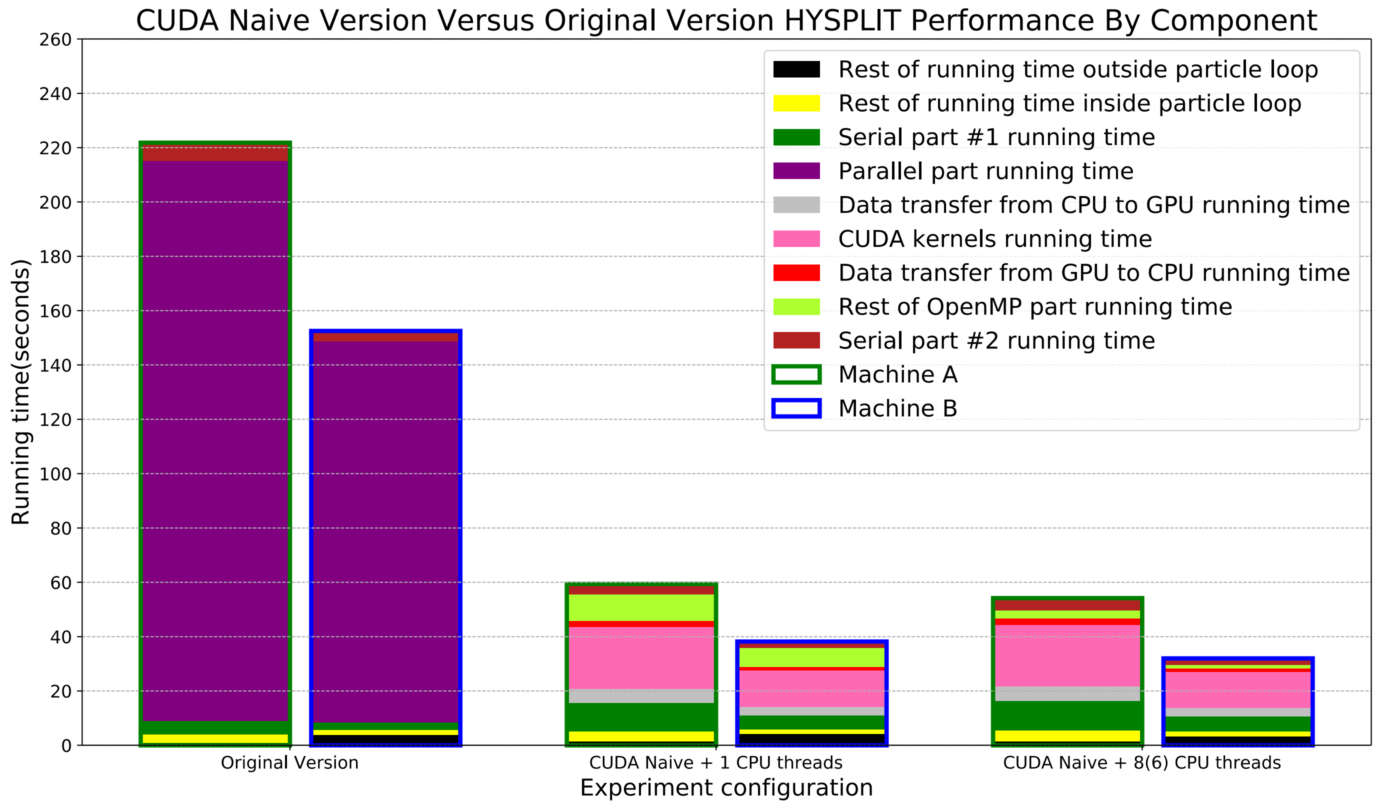
13 Performance Results: OpenMP Version (II)

	Machine A				Machine B			
	whole program		parallel part		whole program		parallel part	
Speedups:	actual	theor.	actual	theor.	actual	theor.	actual	theor.
Original	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×
OMP-1-T	0.83×	1.00×	0.82×	1.00×	0.88×	1.00×	0.87×	1.00×
OMP-2-T	1.53×	1.82×	1.60×	2.00×	1.54×	1.82×	1.60×	2.00×
OMP-4-T	2.77×	3.08×	3.19×	4.00×	2.58×	3.08×	2.98×	4.00×
OMP-8(6)-T	4.01×	4.71×	5.62×	8.00×	3.41×	4.00×	4.29×	6.00×

- 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

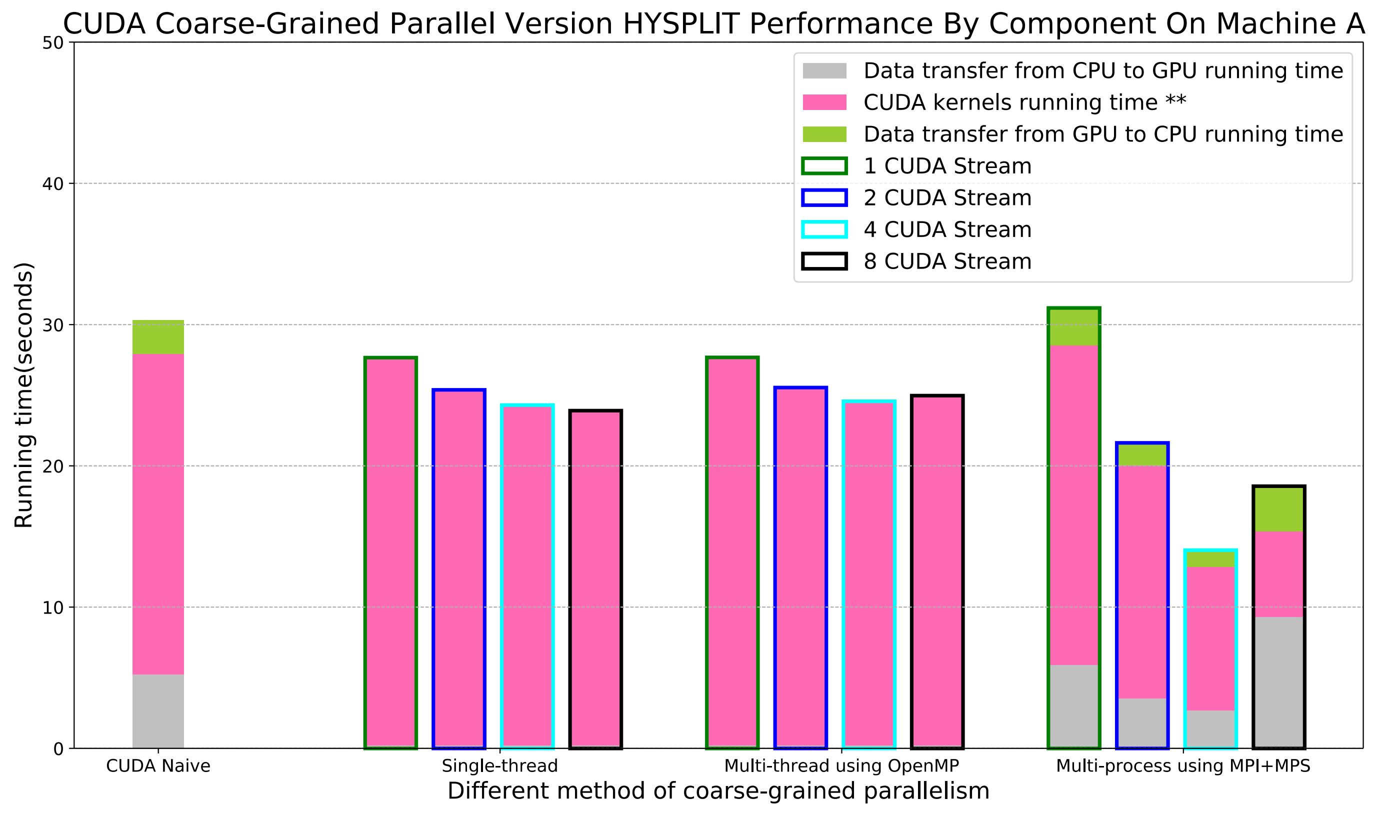
THE AUSTRALIAN NATIONAL UNIVERSITY

14 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

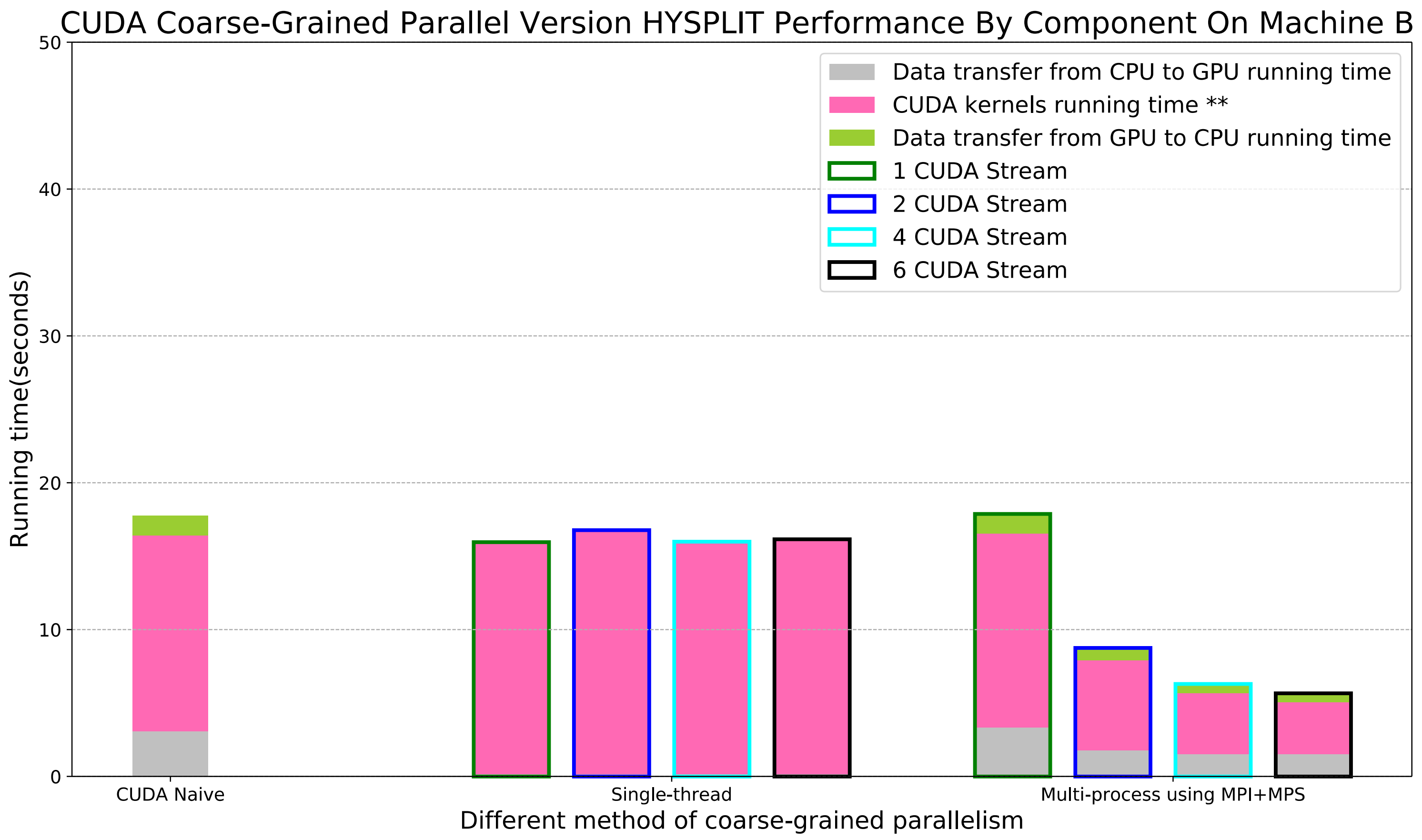
15 Results: Coarse-grain CUDA Version, Machine A



**For single-thead and OpenMP multi-thread version, this component is the aggregated time for concurrent kernels and data transfer

- note: for the single- and multi-thread approaches, times for data transfer and kernel execution cannot be distinguished

16 Results: Coarse-grain CUDA Version, Machine B



**For single-thead version, this component is the aggregated time for concurrent kernels and data transfer

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

17 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

18 Results: Coding Effort

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

code category	OpenMP	CUDA	difficulty
main program	706	719	medium
Parloop & its isolation	815	942	high
parallelization barrier removal	457	490	high
interfaces (for parallel programs)	706	811	low
device data (allocate, transfer)	–	480	low
device kernel	–	292	medium
device subprogram	–	1127	mostly low

- 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

19 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$\times$ speedup (best with multiple CPUs on deposition)

20 Conclusions (II) and Future Work

- coarse-grained GPU parallelization with MPI processes gained 2–3$\times$ 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`)