OpenACC pgfortran: substantial speedups and beyond for the O(3) Condensation algorithm for determinants and estimation

By

Drs. Damien Mather and Chris Scott

University of Otago and NeSI

damien.mather@otago.ac.nz chris.scott@nesi.org.nz

background

- this process was the bottleneck to scaling a wider project to extract D-efficient training, validation and test samples for big data predictive analytics
  - Example application areas include:
    - Supermarket discount card member data
    - Mobile app user data
    - Web user data
  - May also be useful in:
    - Better insights from agricultural field trial design and analysis (welcome comments)

The problem with current split samples

- The predictors, or inputs, are often highly correlated
- This can be shown bias the effect estimates unless you use mixed model estimators
  - Current estimators can be very slow and/or unreliable when estimating large (1000 vars) mixed models on large (50,000 obs) data samples
  - Not all business (and other domain) problem models of interest benefit from variable clustering
    - Currently best practice for dealing with the curse of dimensionality
    - E.g. cross-availability effects in logit supermarket category brand models

A solution: find D-efficient arrangements

- Uses design of experiments tool:
  - Modified Federov designs found using OR techniques to maximise the D-efficiency of a candidate design sample
  - Good design samples can be found quickly from big data up to a bottleneck limit of about 144 availability and cross-availability candidate effects
  - Independent samples, for optimal model complexity pruning for generalisation, can be found after filtering out all cases (customers) from the training sample

The bottleneck

- The modified federov algorithm needs to evaluate the (log) determinant of each candidate design.
- The log determinant function needed, being 0(3), quickly slows down the sample generation beyond usefulness on current desktop applications
- The big data will generate ill-conditioned candidate designs
  - Often enough to fail the main algorithm
- Main research ‘pushed to stack’ whilst solutions investigated…

I found an algorithm for calculating determinants that can be immunized against ill-conditioned data

Dodgson, C. L. (1866). Condensation of Determinants, Being a New and Brief Method for Computing their Arithmetical Values. *Proceedings of the Royal Society of London, 15,* 150-155.

..by reordering each condensation argument to preserve as much good condition as possible

Dong, X., Barnett, E. N., & Dhall, S. K. (2018). Parallel Matrix Condensation for Calculating Log-Determinant of Large Matrix.

Initial GPU implementation

- Starting with an MPI parallel version of the code
- Add in offloading to GPU
- Different options for running the code
  - MPI only
  - GPU only
  - MPI+GPU hybrid
- OpenACC for GPU offloading
  - Simple/not invasive - often just adding pragmas without other code changes
  - Support for Fortran and C/C++
  - Supported by PGI and Cray compilers (and others?)
  - Managed memory with PGI compilers - compilers handles data transfers between host and device (good starting point)

Algorithm

1. Distribute columns among MPI processes
2. Main loop over columns to compute determinant
    a. One MPI process only finds pivot row - max value in column - and normalises column
    b. Broadcast normalised column and pivot row index to other MPI processes
    c. All MPI processes switch pivot row with last effective row
    d. All MPI processes condensation calculation (nested loop over rows and columns)

- Most expensive operation - most of the time is spent here
- Compile with PGI and managed memory
    - mpipgf90 -ta=tesla:managed ...
- Add “kernels” block around these sections and enable managed memory
    - Couple of additional lines of code: code link

```
182     ! This part is matrix condensation calculation
183     pivot_row = INT(column_array(N_row))
184     N_row = N_row-1    ! Number of row will be one less
185     !$acc data
186     !$acc kernels
187     row_array(i+col_shift:N_col) = local_A(pivot_row,i+col_shift:N_col)  !Get the row_array
188     !Switch last effective row and pivot row for local_A
189     local_A(pivot_row,i+col_shift:N_col) = local_A(N_row+1,i+col_shift:N_col)
190     do col = i+col_shift,N_col
191      do row = 1,N_row
192       local_A(row,col)= local_A(row,col) - column_array(row)*row_array(col)
193      end do
194     end do
195     !$acc end kernels
196     !$acc end data
```

OpenACC notes

- “kernels” region - compiler will try and figure out what can be parallelised
  - The compiler might not always be able to determine whether a loop is safe to parallelise
- Compile with “-ta=tesla:managed” - managed means compiler will figure out when to copy data
  - Might end up with more copies than necessary
- Alternatives
  - “loop” pragma - tell the compiler to parallelise a loop
    - Compiler assumes you know what you are doing
  - “copy”, “copyin”, “copyout”, “update”, etc. clauses to manually copy data
    - Compiler assumes you know what you are doing
- To use OpenACC with PGI compilers on NeSI
  - “module load PGI CUDA”
  - “module load impi/2019.6.166-PGI-19.10-GCC-9.2.0-2.32 CUDA” (if you need MPI too)

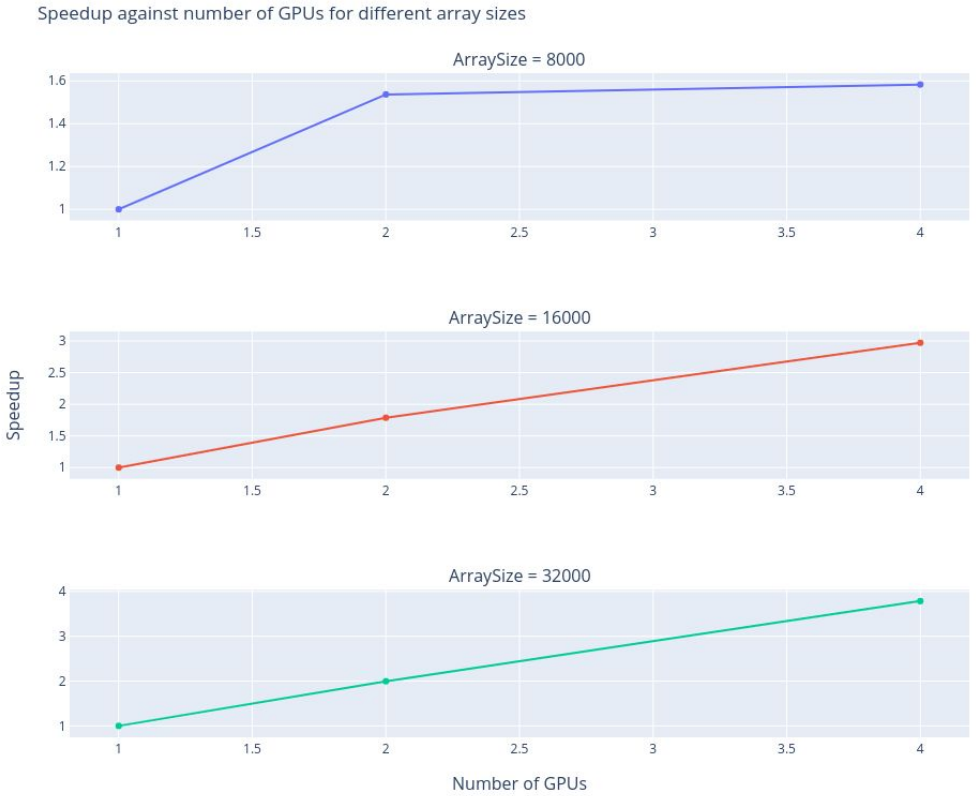
Initial GPU implementation - Mahuika timings

Num procs	Time without GPUs (s)	Time with GPUs (s)	Speedup from GPUs
1	184.7	14.0	13.2
2	95.2	9.1	10.5
4	47.2	8.9	5.3

- 8000x8000 matrix
- 1 Tesla P100 GPU per MPI process
- 13x speedup by adding two lines of code

Initial GPU implementation - Mahuika

- Good scaling for larger array sizes - ~4x speedup with 4 GPUs for 32,000 row matrix
- Speedup drops off at smaller sizes - not as much work



Optimised version - data locality

1. Distribute columns among MPI processes
2. Main loop over columns to compute determinant
   a. One MPI process only finds pivot row - max value in column - and normalises column
   b. Broadcast normalised column and pivot row index to other MPI processes
   c. All MPI processes switch pivot row with last effective row
   d. All MPI processes condensation calculation (nested loop over rows and columns)

- Currently the full data gets copied too and from the device every iteration
- Instead copy data onto device once at beginning of main loop and off once at the end
- Do all calculations on the GPU and just update host arrays as needed

Optimised version - data locality

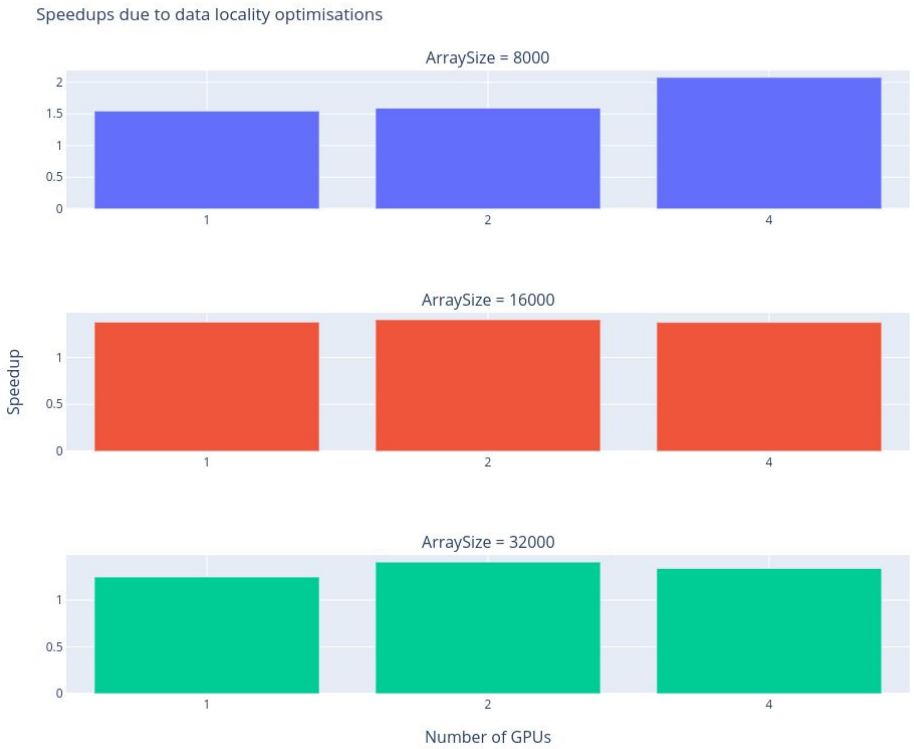
- Requires more changes to source code
  - For example, instead of two lines to find location of maximum element in a column and its value, required ~30 lines and two loops (code)
- All operations are run on the GPU except the MPI broadcast, this requires some transfer between host and device
  - The process that is finding the pivot row updates its host column array before broadcasting to other processes, then all process update their device column array

```
!    pivot_row = maxloc(abs(local_A(1:N_row,i)),DIM=1)
!    pivot_value = local_A(pivot_row,i)
    ! find pivot row and value (first pass to find pivot value)
    pivot_value = 0.d0
    !$acc parallel loop reduction(max:pivot_value) present(local_A)
    do j=1,N_row
     val = abs(local_A(j,i))
     if (val > pivot_value) pivot_value = val
    enddo
    !$acc end parallel loop

    ! find pivot row and value on device (second pass to find pivot row index)
    matches = 0
    !$acc parallel loop present(local_A) reduction(+:matches) copyout(pivot_row)
    do j=1,N_row
     val = abs(abs(local_A(j,i)) - pivot_value)
     if (val .lt. tol) then
      matches = matches + 1
      !$acc atomic write
      pivot_row = j
     endif
    enddo
    !$acc end parallel loop
    !$acc serial present(local_A) copyout(pivot_value)
    pivot_value = local_A(pivot_row,i)
    !$acc end serial
```

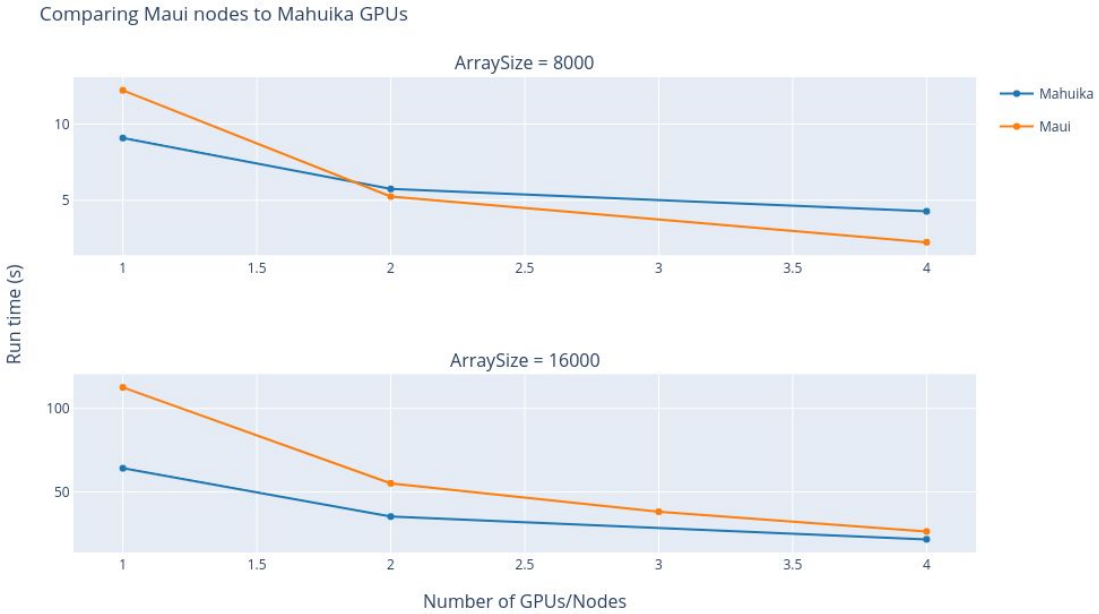
Optimised version - performance (Mahuika)

- 1.5x - 2x speedup for 8000 row matrix
- 1.2-1.4x speedups for larger size matrices
- Probably a big enough boost to make adding the extra complexity to the code worthwhile



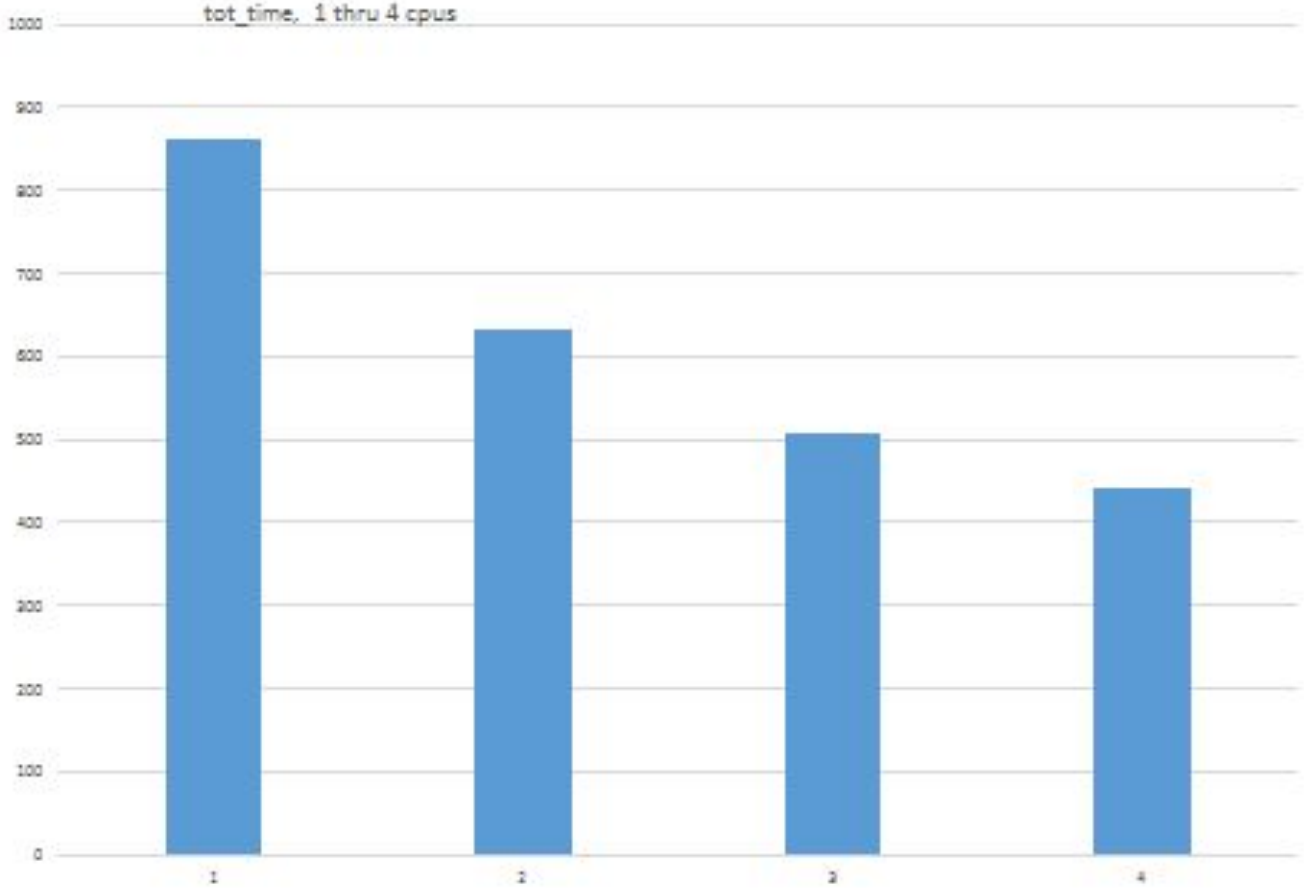
MPI-only vs MPI+OpenACC

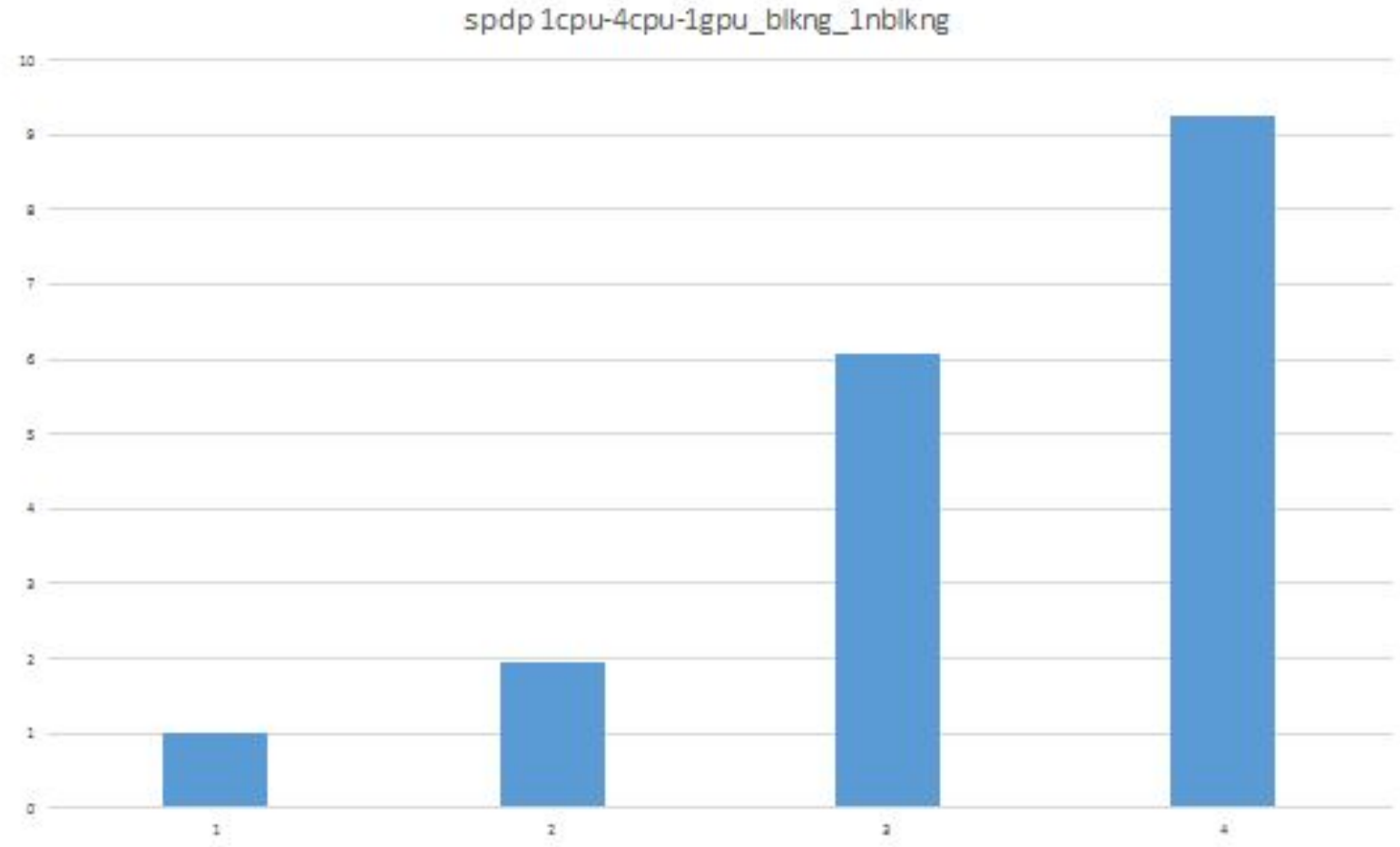
- Maui Cray XC with 40 cores per node
- Good scaling as increase number of nodes
- Mahuika better with 1 GPU vs 1 node on Maui
- Scales up better on Maui

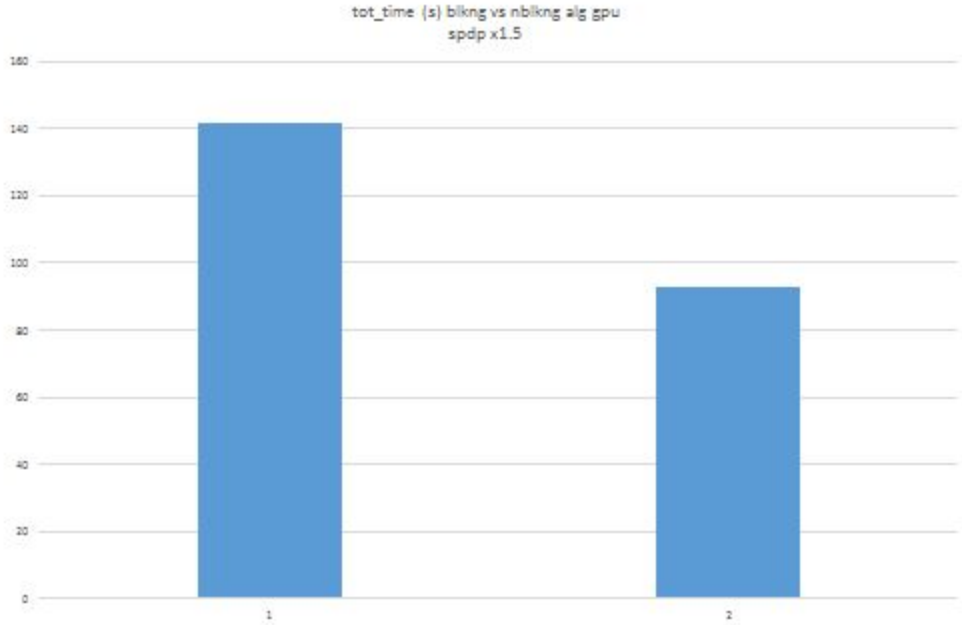


Checking for similar performance gains at other end of the hardware spectrum

OpenACC accelerator = Nvidia GT 730 GDDR5
2 multiprocessors
128 double precision cores
warp size 32
unified addressing
managed memory
pgi pgfortran CE
openmpi 3.3
linux 18.04LTS intel Core 2 Quad Q9750@3GHz


tot_time, 1 thru 4 cpus
1000
900
800
700
600
500
400
300
200
100
0
1
2
3
4


spdp 1cpu-4cpu-1gpu_blkng_1nblkng
10
9
8
7
6
5
4
3
2
1
0
1
2
3
4


tot_time (s) blkng vs nblkng alg gpu
spdp x1.5
160
140
120
100
80
60
40
20
0
1
2

Summary

- OpenACC with PGI compilers and managed memory
  - Big speedup achieved with minimal code changes
- More speedup available at the cost of slightly more complex code
- NeSI Consultancy Service
  - https://www.nesi.org.nz/services/consultancy
  - Team of Research Software Engineers and Data Science Engineers
  - Contact us to find out more