CUG Talk

Porting the microphysics model CASIM to GPU and KNL Cray machines

Nick Brown, EPCC
nick.brown@ed.ac.uk

Contents

- Background
  - What is CASIM
  - Existing work on our porting of atmospheric codes to GPUs
- CASIM on the GPU
  - Approach adopted
  - Performance comparison between K20X and P100 with CASIM
  - Challenges encountered
- CASIM on the KNL
  - Performance against GPU and CPU versions

Background

- Cloud AeroSol Interactions Microphysics (CASIM) model is a bulk microphysics scheme
  - Concerned with modelling the interaction of water droplets at millimetre scale
  - Microphysics is a critical aspect of many weather and climate models. CASIM is used as a sub-model by the UM, MONC, LEM and KiD models
- Computationally intensive
  - Can double or even treble the overall runtime
- CASIM is interesting because it explicitly carries aerosol mass in the cloud which allows one to study the evaporation of aerosol and how moisture is returned to the system

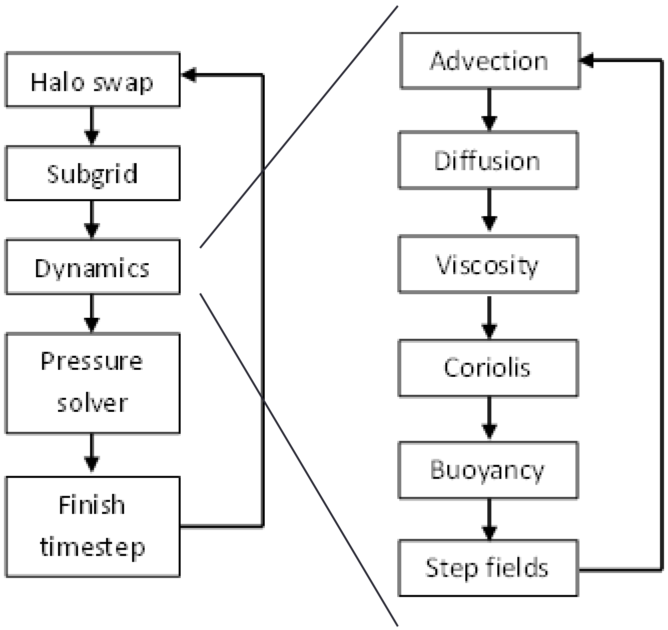
Met Office NERC Cloud (MONC) model

- MONC is a new model we have developed for simulating clouds and atmospheric flows
  - Written in Fortran 2003 and oriented around the concept of plug-ins.
  - A model core is provided which contains general utility functionality but all science and parallelism is provided by independent, separate components

- Runs on much larger domains (billions of grid points) than previous generations of models
  - A parent model for CASIM and a coupler has been developed as a component

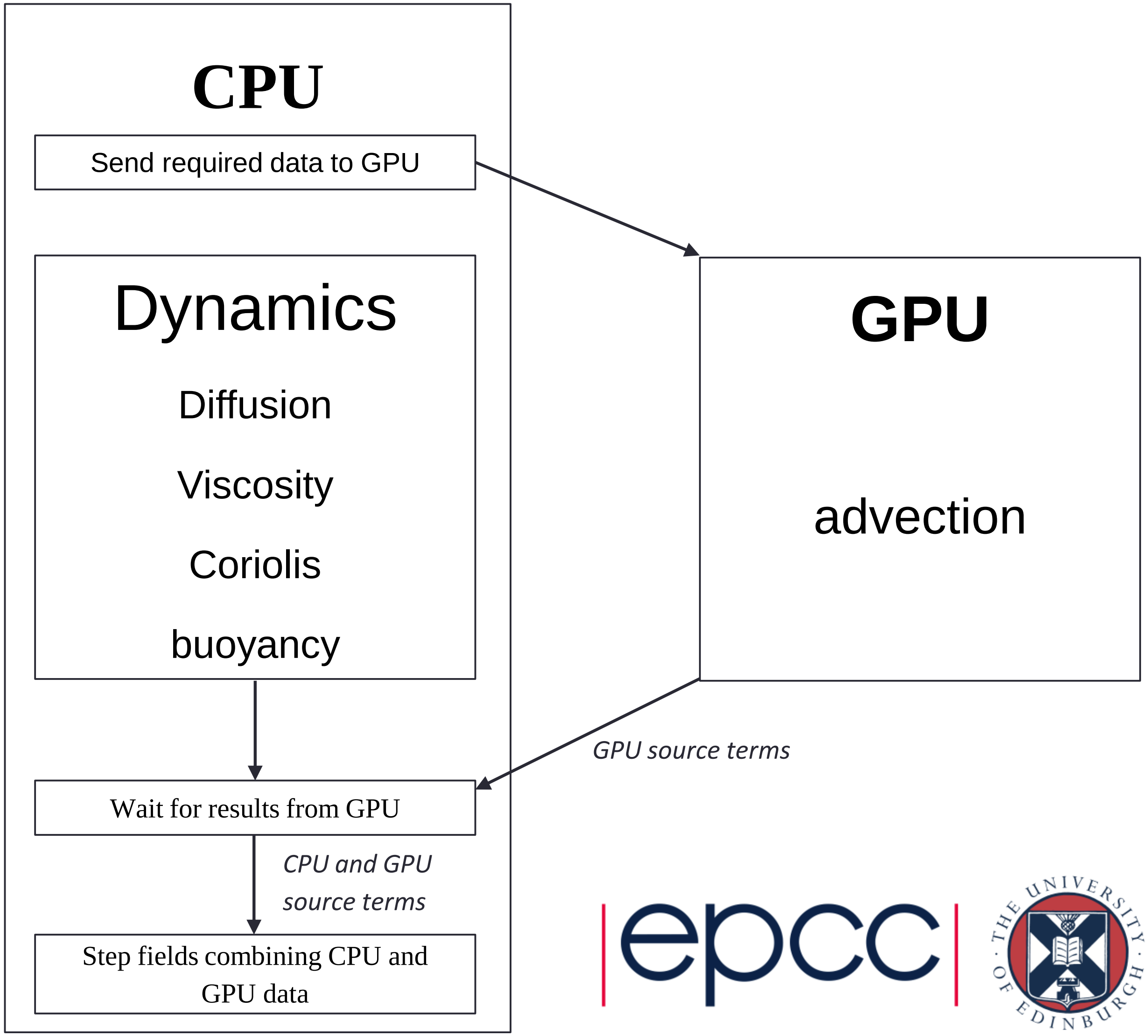
MONC acceleration on GPUs

- In previous work, before CASIM was integrated, we identified that advection was the most computationally intensive part of the code
  - Part of the dynamics group of components
- Components in the dynamics group contribute their calculations to source terms
  - This operator (addition) is commutative and associative
  - Can execute in any order as long as step fields (integration of source terms) is done last
  - So lets offload it!



The hybrid approach

- Data transfer is asynchronous
- Constants copied across only once on model initialisation
- Share data between GPU kernels
  - Wind in x,y,z is common to all
- OpenACC
  - Cray compiler



However…….

- The performance was not particularly good
  - There was not enough computation in advection to amortise the cost of data transfer and-so we did not get a speed up
- But CASIM is many times more computationally intensive than the advection kernel we offloaded
  - So can we use this same hybrid approach, infrastructure and OpenACC to offload this and get a performance benefit?
- This previous work was done with OpenACC (Cray compiler)
  - Learnt about a number of challenges and their workarounds

OpenACC CASIM: A choice

- The challenge is that CASIM has a number of computationally intensive kernels
  - These are called frequently in the code but with lots of non-computational work done before and after each *hotspot*
  - Such as conditionals, loops etc
- The scheme is operating on many Q (moisture) fields in vertical columns
  - Tightly coupled in the vertical but not in other two dimensions
  - Per timestep columns are independent

```
subroutine CASIM()
   do i = i_start, i_end
      do j = j_start, j_end
         ...
         call hotspot1()
         ...
         call hotspot2()
         ...
         call hotspot3()
         ...
      end do
   end do
end subroutine CASIM
```

OpenACC CASIM: A choice

```
do i = i_start, i_end
   do j = j_start, j_end
      call before()
      call hotspot()
      call after()
   end do
end do
```

```
do i = i_start, i_end
   do j = j_start, j_end
      call before()
   end do
end do
do i = i_start, i_end
   do j = j_start, j_end
      call hotspot()
   end do
end do
do i = i_start, i_end
   do j = j_start, j_end
      call after()
   end do
end do
```

- So we can either refactor the code like so
  - But this will result in lots of data movement and the CPU will still be busy (negate our hybrid approach)
- Or we can offload the entirety of CASIM to the GPU

Offloading the entirety of CASIM

```
subroutine CASIM()
   !$acc parallel
   !$acc loop collapse(2) gang worker vector
   do i = is, ie
      do j = js, je
          ...
          call microphysics_common(i,j)
          ...
      end do
   end do
   !$acc end loop
   !$acc end parallel
end subroutine CASIM

subroutine microphysics_common(i,j)
   !$acc routine seq
   ...
end subroutine microphysics_common
```

- OpenACC support offloading subroutines
  - We allocate a thread per column in the domain
  - Hence the *seq* in each subroutine
- In total 50 Fortran modules and 123 subroutines offloaded
- CPU code contains lots of intermediate temporary variables
  - Which have to be duplicated for each thread

OpenACC implementation challenges

- Passing arrays of derived types to OpenACC subroutines
  - We had to wrap these arrays in a wrapper derived type or else the PTX code would report an error during assembly

```
type :: process_name
    integer :: id
    ...
end type process_name

subroutine sum_procs(..., iprocs, ...)
    !acc routine seq
    type(process_name), intent(in) ::
         iprocs(:)
    ...
end subroutine sum_procs
```

```
type :: cray_workaround_iprocs_wrapper
   type(process_name) :: iprocs(22)
   integer :: iprocs_count
end type cray_workaround_iprocs_wrapper

subroutine sum_procs(..., iprocs, ...)
   !acc routine seq
   type(cray_workaround_iprocs_wrapper),
         intent(in) :: iprocs
   ...
end subroutine sum_procs
```

OpenACC CASIM challenges

- Above a certain threshold CUDA supports arguments by copying them into a buffer and passing the pointer
- Host side of large arguments is not supported by the Cray compiler
  - Empirically found at 532 arguments are packed into memory rather than passed separately which is not supported
  - Arrays can take up to 10 CUDA kernel arguments, we merged many of the Q fields together from multiple 3D arrays to a single 4D array

```
subroutine process(q1, q2, q3, ...)
    !acc routine seq
    double, dimension(:,:,:),
         intent(in) :: q1, q2, q3
    ...
end subroutine process
```

```
subroutine process(q, ...)
    !acc routine seq
    double, dimension(:,:,:, :),
         intent(in) :: q
    ...
end subroutine process
```

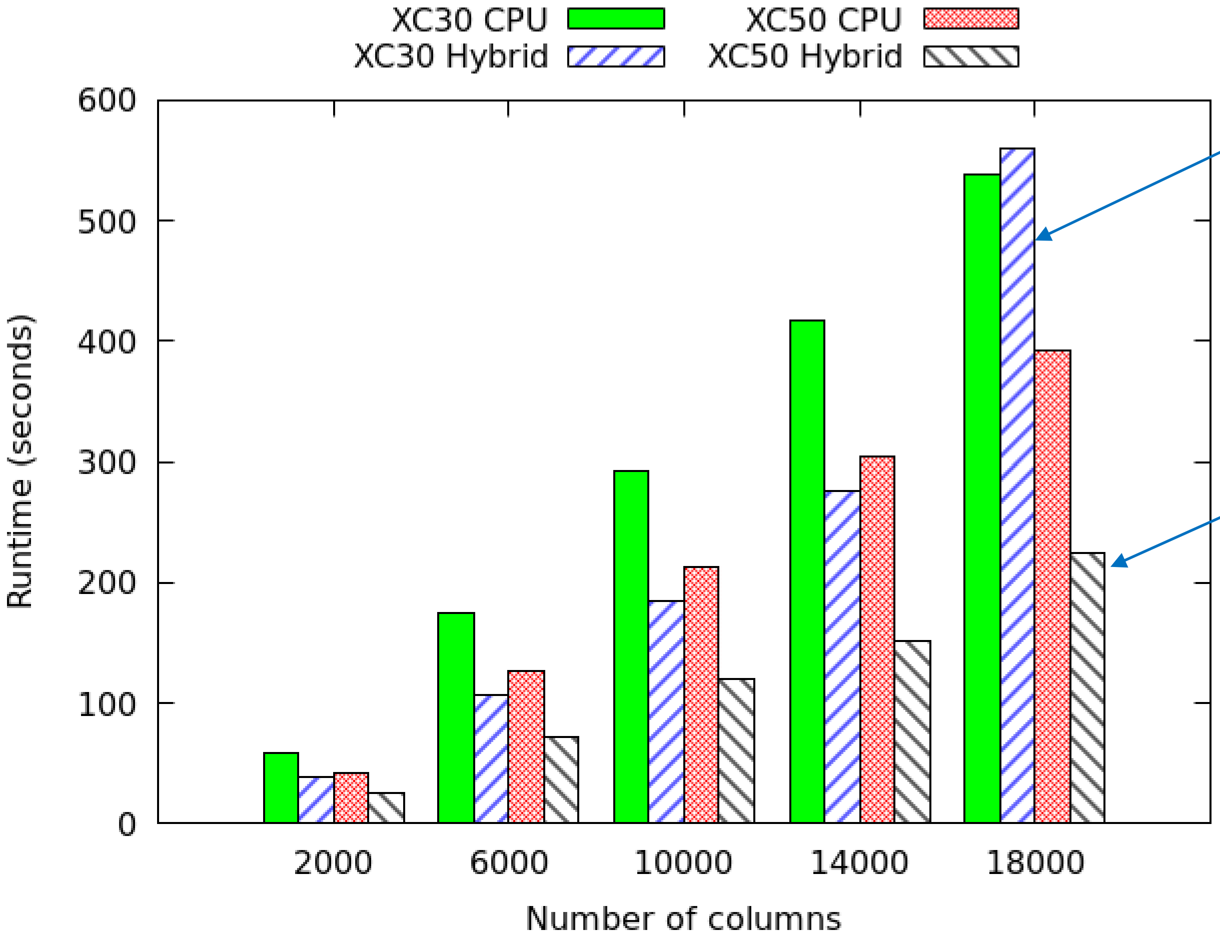
OpenACC CASIM challenges

- At higher levels of optimisation, variables that are provided a value only under some logical condition generate an error during assembly
  - So need to ensure that all variables are assigned a value irrespective
  - Arguably this is good style anyway, but the existing code did not do this & was still legal Fortran

```
...
if (some_condition) dm_3=5.435
...
if (some_condition) othervariable=dm_3
```

```
...
dm_3=0.0
othervariable=0.0
if (some_condition) dm_3=5.435
...
if (some_condition) othervariable=dm_3
```

OpenACC CASIM performance

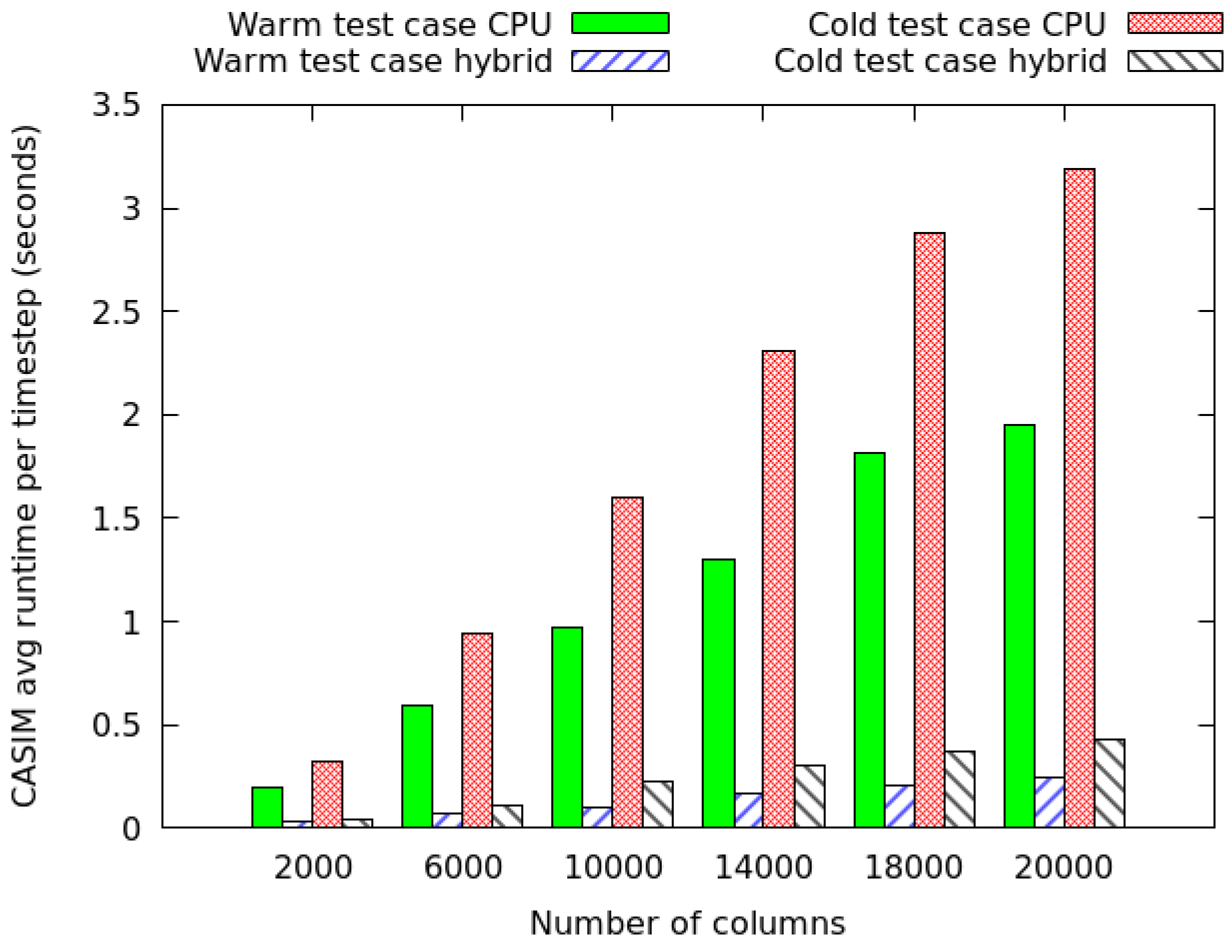


Large runtime jump due to exhaustion of registers (64 per thread)

With 56 SMs we have many more registers so use 128 per thread on the P100

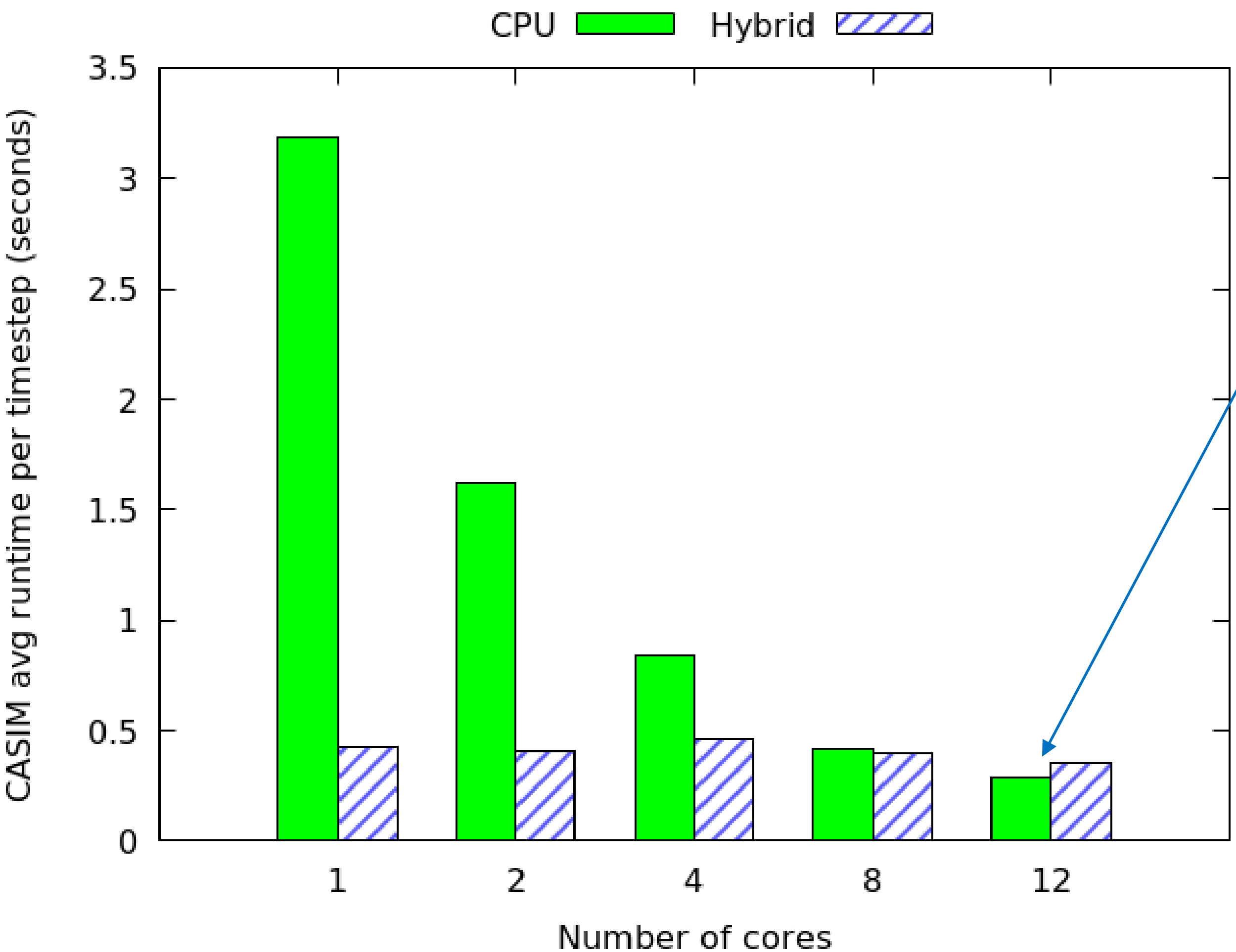
- Run on Piz Daint, warm stratus test case for 2000 model seconds. 60 vertical levels. Cray compiler (CCE 8.5.5), O3

CASIM OpenACC Performance



- XC50 runs (P100)
- Warm (5 Q fields) and cold (18 Q fields) stratus test cases
- Time is average CASIM runtime per model timestep
- Cray compiler

CASIM OpenACC Performance

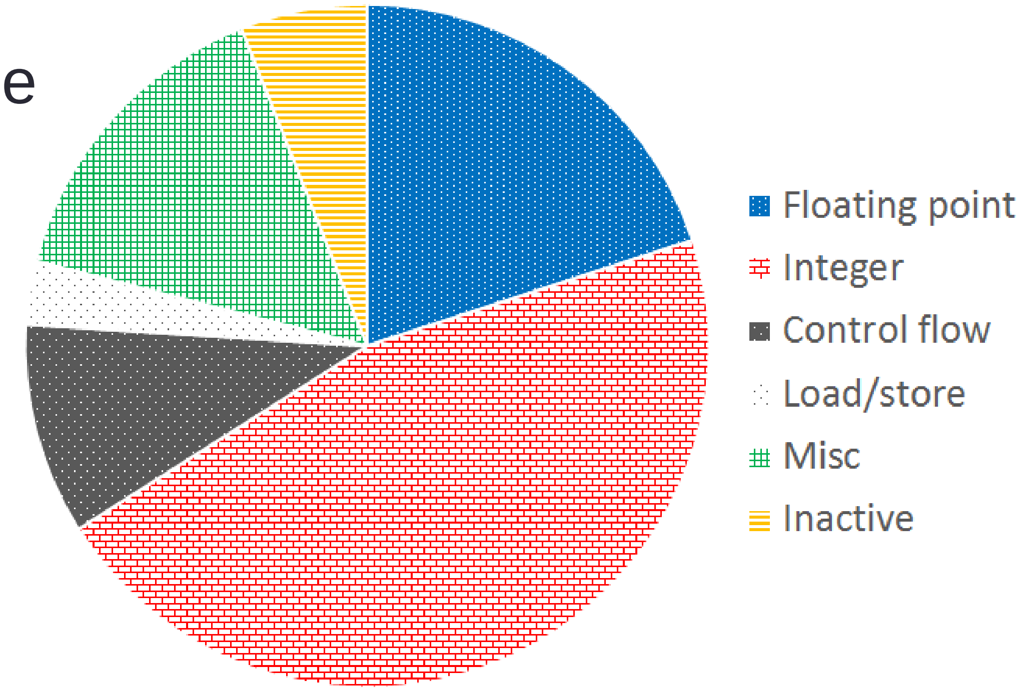


With 12 Haswell cores the CPU is outperforming the GPU

But remember that this is running in a concurrent fashion to the rest of MONC, so the overall parent model runtime would still be less

CASIM Performance reasons

- The vast majority of GPU time is spent executing CASIM rather than data transfers
- But the option for offloading the entirety of the model means we are dominated by integer operations



Column size	Config	To GPU	Kernel	From GPU
2000	Warm	1.5ms 5%	29ms 93%	0.8ms 3%
2000	Cold	1.82ms 4%	39ms 94%	0.74ms 4%
10000	Warm	7ms 7%	88ms 88%	4.6ms 5%
10000	Cold	11.2ms 5%	214ms 93%	4.6ms 2%
20000	Warm	18ms 7%	224ms 90%	8ms 3%
20000	Cold	22.56ms 5%	395ms 93%	8.1ms 2%

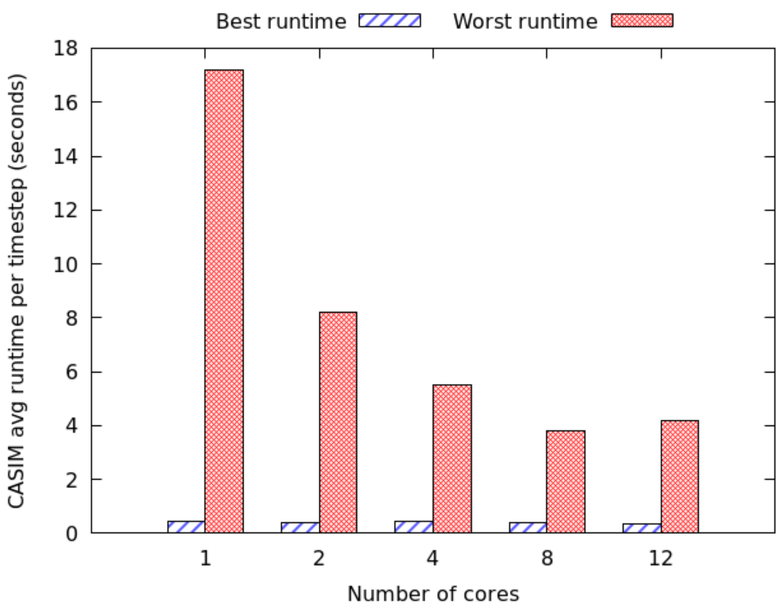
The pie chart is based on figures from nvprof on 4000 columns, but due to the size of the kernel this ran out of memory (collecting metrics and events) with larger numbers of columns

epcc

THE UNIVERSITY OF EDINBURGH

Configuration choices

- With OpenACC it is possible to explicitly set the number of gangs (number of thread blocks) and vector length (threads per block.)
  - Getting the “wrong” value resulted in very poor performance. As did the default settings



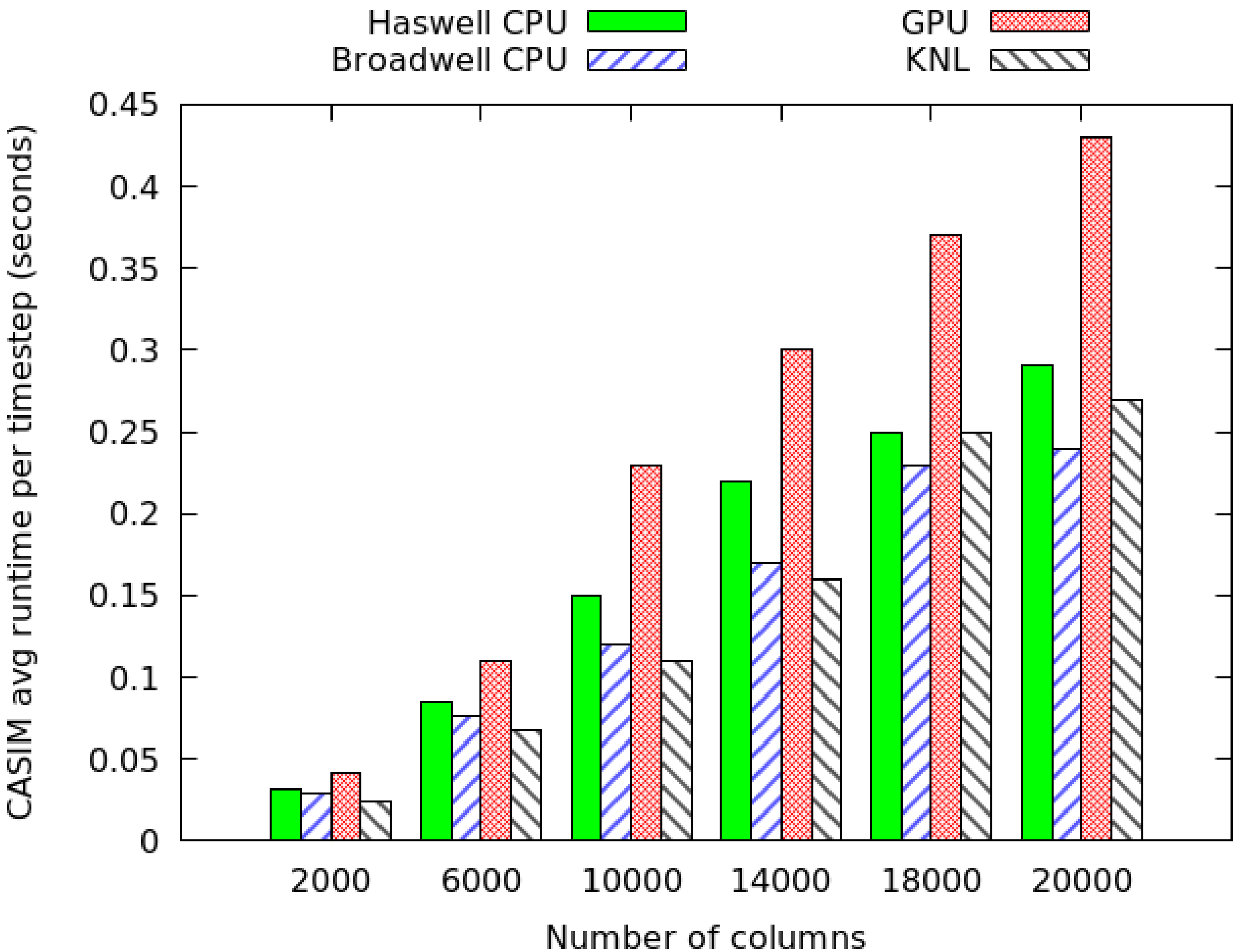
- At smaller numbers of vertical columns (threads) use more, smaller, thread blocks
- At greater numbers of vertical columns (threads), use fewer larger thread blocks
- It is a shame we can not dynamically control this at runtime

CASIM on the KNL

- An implementation of CASIM for the KNL using OpenMP has been produced
  - So we have a choice between processes and threads on the KNL
  - Distributed columns amongst threads
- Deals with the global data via the *threadprivate* clause by the declaration and *copyin* to copy the values in (and allocate space) for them on entry to CASIM.

- Applied the SIMD OpenMP directive to inner computationally intensive loops working up or down the column

Performance of CASIM on KNL

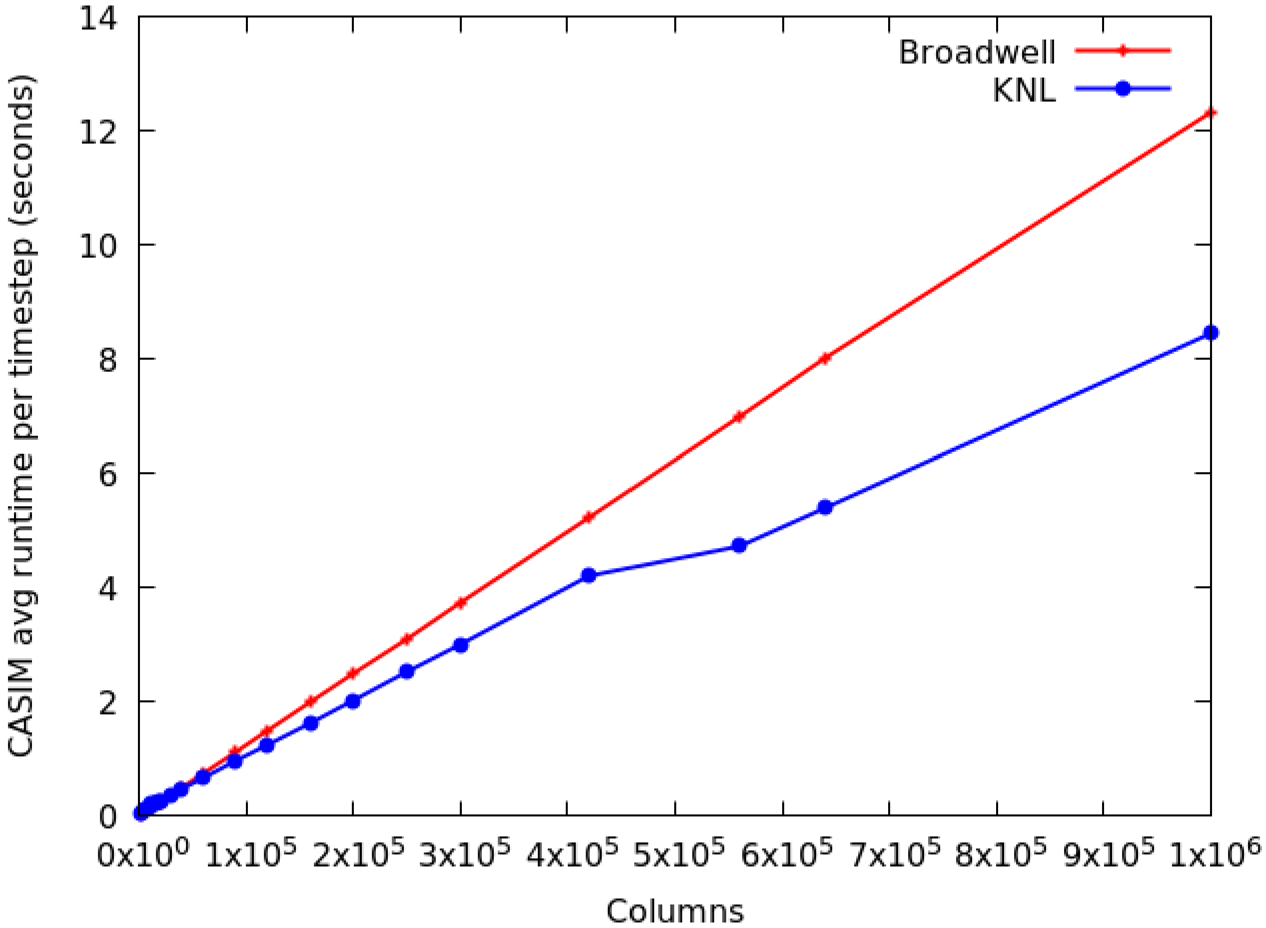


- 12 core Haswell
- 18 core Broadwell

- Stratus cold cloud test case, with 60 vertical levels
- Average runtime for CASIM per timestep
- Cray compiler, O3

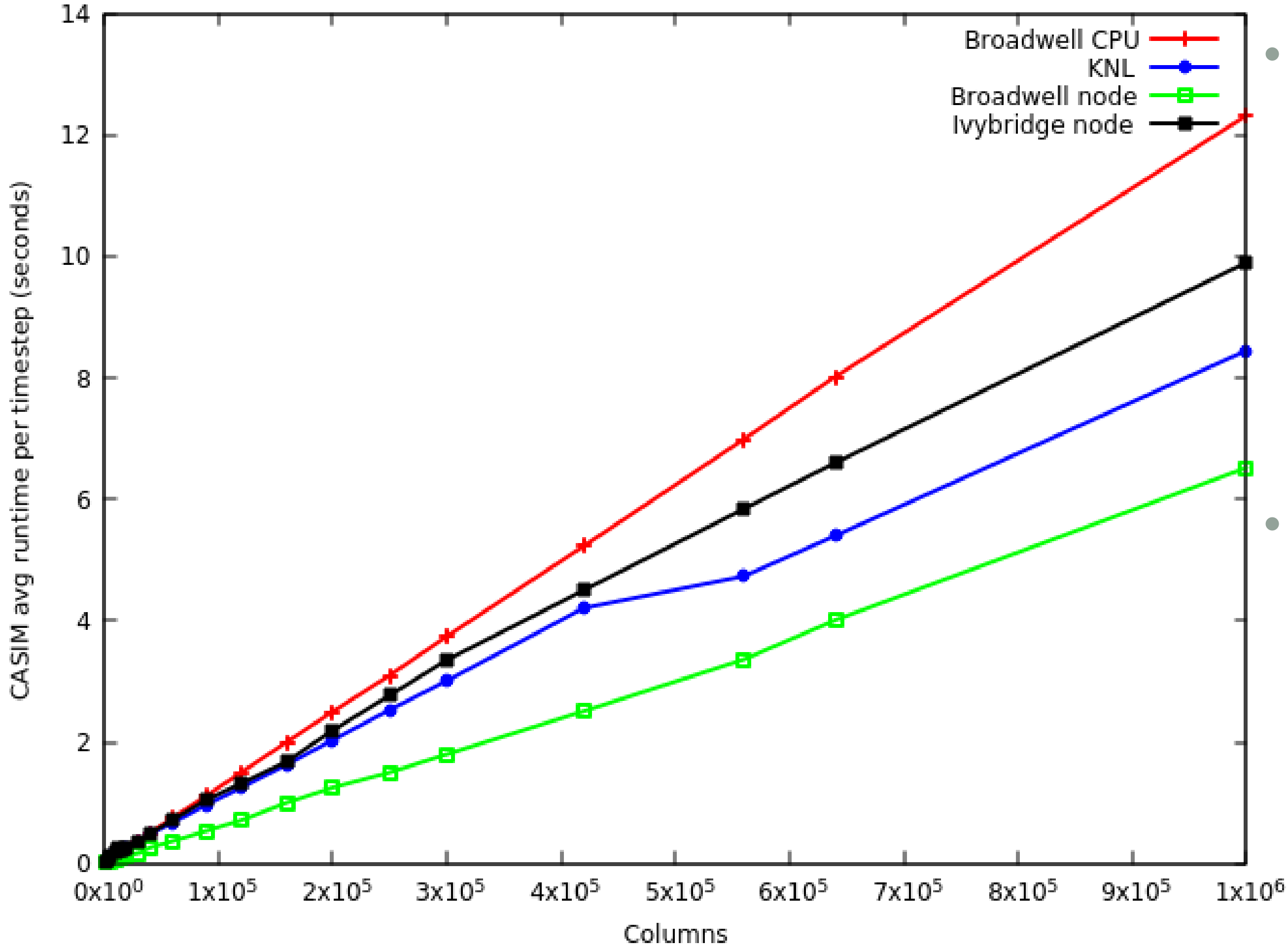
ARCHER XC40, 64 core KNL (7210), MCDRAM as cache and running in quadrant mode

But that's not quite the end of the story….



- For smaller amounts of data there isn't much difference between the Broadwell and KNL
- But for larger domain sizes the KNL performs significantly better

But that's not quite the end of the story….



- For smaller amounts of data there isn't much difference between the Broadwell and KNL
- But for larger domain sizes the KNL performs significantly better

Placement choices for the KNL

- We found that for small numbers of columns it was best to enable hyper-threading and run one CASIM process per hyper-thread (4 per physical core)
  - There was a significant performance impact to running a process per physical core with four threads
- At around 12000-18000 columns, it became optimal to run one process per physical core threading over the hyper-threads

- *The GPU ran out of memory after 20000 columns, as on the GPU there was far more replication of temporary data for each column running as threads run concurrently*

Conclusions and further work

- Offloaded the entirety of CASIM which was necessary to avoid excessive data transfer but this did have a memory, performance and development time impact
  - The P100 is a significant improvement over the K20X
- KNL looks promising as long as one runs with large system sizes.
  - Much quicker to utilise and experiment with.
  - Hard to beat a node with 36 Broadwell cores

- Many thanks to CSCS for access to Piz Daint for this work and previous GPU acceleration of MONC we also did

This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk)