|epcc|

Accelerator weather forecasting

Angus Lepper

August 21, 2015

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2015

Abstract

Advection is the transport of a quantity due to fluid flow, and is an important, computationally intensive part of any fluid simulation. OpenACC GPU acceleration of the advection components of MONC, an atmospheric LES, was pursued. Although this yielded no speedup, the reasons for this are examined, and the conditions under which it may become achievable are considered. OpenACC's usability is also evaluated.

Contents

List of Tables

List of Figures

List of Listings

Acronyms

CFL Courant-Friedrichs-Lewy.

CSCS the Swiss National Supercomputing Centre.

DNS Direct Numerical Simulation.

FFT Fast Fourier Transform.

FP Floating Point.

GCM General Circulation Model.

GPU Graphics Processing Unit.

ILP Instruction-Level Parallelism.

LEM Large Eddy Model.

LES Large Eddy Simulation.

MONC the Met Office NERC Cloud model.

MPI Message Passing Interface.

PCI Peripheral Component Interconnect.

PCIe Peripheral Component Interconnect (PCI) Express.

PGI The Portland Group, Incorporated.

POD Plain Old Data.

PW Piacsek-Williams.

QUICKEST Quadratic Upstream Interpolation for Convective Kinematics with Estimated Streaming Terms.

SM Streaming Multiprocessor.

TDP Thermal Design Power.

TVD Total Variation Diminishing.

ULTIMATE Universal Limiter for Transient Interpolation Modeling of the Advective Transport Equations.

Acknowledgements

I would like to express my most sincere gratitude for the support, enthusiasm, and robust feedback provided by Nick Brown in supervision of this project. This project may not have been possible except for the *Piz Daint* facility kindly provided by CSCS. Although not directly involved in this project, I would like also to thank Adam Carter for his effort in ensuring I could proceed with the year which got me here. Finally, I would like to thank my family and friends for their summer of boundless patience.

Chapter 1

Introduction

Weather forecasts and the models which inform them are a very visible application of HPC. The layperson may be most familiar with the daily forecast from television, but they also play a vital role in global logistics and transport. Airports which see dozens of flights per hour may divert incoming aircraft based not on the clear weather when they leave but on the fog expected when they arrive, with cost to the carriers measured in millions of dollars per year. They also provide a means by which atmospheric scientists can provoke and study weather phenomena in the controlled environment they would otherwise lack.

As the previously-reliable steady upward tick in single-threaded performance falters, and traditional CPUs provide only around a dozen cores per socket, alternative massively-parallel 'accelerator' architectures are being explored. The most common of these are Graphics Processing Units (GPUs), by now well established as much more general-purpose processors than their graphical origin might suggest.

Although the hardware is relatively mature, a dominant software model or technology is yet to assert itself. This project pursued GPU acceleration of weather model components using OpenACC, a directive-based technology with which both CPU and GPU may be targeted by a single source code. This did not yield an improvement in performance for the component evaluated, but the circumstances under which a benefit might be seen are investigated, and the process and infrastructure developed in this project could equally be applied there.

A description of the structure of the rest of this document follows. Chapter 2 reviews weather simulation in general, and the particular weather model and the components which relate to this project. Chapter 3 describes the design, implementation, and development process of the accelerated components produced by this project. Performance, design and technology are evaluated in Chapter 4. The document draws to a close and set out a path for further work in Chapter 5.

Chapter 2

Background and literature review

Modern weather forecasting is driven by numerical simulation of the atmosphere. These very large-scale simulations are decomposed geometrically, and distributed over highly-parallel computer clusters or supercomputers. Different simulations target a variety of use-cases, such that the UK Met Office operate more than a dozen distinct weather and climate models [23]. For many use-cases, the time in which a forecast simulation must complete is fixed e.g. a 3-day forecast should be available in hours. Faster simulation permits improved fidelity within the same real-time period via e.g. increased resolution.

Fluid dynamic simulations such as these are categorised according to their treatment of the fine-scale structure. Turbulent mediums such as the atmosphere exhibit significant structure at a wide range of scales [14]. To capture fine-scale behaviour directly requires a very high resolution, with correspondingly lengthy run time. This Direct Numerical Simulation (DNS) is impractical for the large volumes typically simulated in weather forecasting. More common is Large Eddy Simulation (LES), whereby fine-scale ('sub-grid') turbulence is filtered out, then approximated by a simplified model ('parameterisation') with direct simulation of dynamics retained above this scale [35].

2.1 Met Office LEM and MONC

The Met Office Large Eddy Model (LEM) [22] is one such LES, commonly used to model the behaviour of boundary layers, convection, precipitation and clouds [4, 23]. Although the model's scientific development continues to keep pace with the state of the art, the 30-year-old code does not scale to the thousands or more cores available in modern machines [4, 9, 39]. EPCC are part of a collaboration to address this by developing a replacement, integrating modern software engineering and scalability with the proven scientific basis. This replacement is the Met Office NERC Cloud model (MONC).

Brown et al. [4] examine MONC's performance and scalability for a dry, neutral boundary layer scenario. This is similar to an existing LEM test case [13, 31]. For a problem

size of 536 million grid points, MONC exhibits strong scaling to over 16 thousand cores, equivalent to 32 thousand grid points per core. Good weak scaling is achieved until 16 thousand cores, at which run time begins to quickly increase. These results suggest the code can scale up to a maximum of around 32 thousand cores and 2.1 billion grid points. Poisson solver choice can affect scalability: run time at large numbers of cores (around 16 thousand or more) can be reduced by using the iterative rather than Fast Fourier Transform (FFT) solver, trading computation for communication.

2.1.1 Architecture

Most functionality in MONC is provided by components. A component is one or more Fortran modules organised within a directory structure. Using a known naming scheme, one of these modules exposes a component descriptor. As illustrated by Figure 2.1, this descriptor defines a 'friendly' name and version number for the component, a list of names for any data fields the component may publish, and a set of pointers to subroutines (the targets of which are termed callbacks) which provide the entry points to the component's functionality. These allow a generic treatment of the subroutines at runtime e.g. calling in turn each subroutine in an array.

A small amount of core functionality is not provided by a component. This core includes the program entry point, a configuration database, and component registration and dispatch. As there may be no compile-time dependency between components, these may directly rely only upon the functionality provided by this core infrastructure. This restriction allows new functionality e.g. this project to be developed wihout damage to existing code, enabling incremental changes under an iterative development model. At build time, a sequence of calls are automatically generated to register each component with the core at runtime. The registry module may thereafter be queried for such attributes of a component as whether it is enabled.

MONC's configuration may be split across a pair of plain text files, with a local configuration extending a more general global configuration. Parsed into a configuration database by the core at runtime, these select which components to enable and specify their order within a sequence of component groups. Whereas components are defined at compile time, all group configuration takes place at runtime.

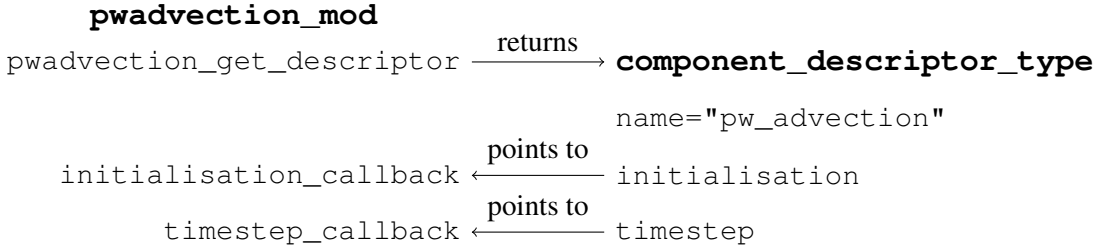


Figure 2.1: Example component: `pwadvection/src/pwadvection.F90`

```
halo_swapper_enabled=.true.
⋮

group_names=start, subgrid, dynamics, solver, pressure-
   terms, last

start_group_type=entire
start_group_contents=halo_swapper, ...
⋮

initialisation_stage_ordering=decomposition, ...
finalisation_stage_ordering=checkpointer, ...
```

Listing 2.1: Example group configuration.

Many components have relatively weak constraints on their order e.g. as they update an accumulator using a commutative and associative operation such as addition, in which case all orders produce the same result modulo the imprecision of Floating Point (FP) arithmetic. Meanwhile e.g. integration of the variable represented by this sum can only be performed after all terms are accounted for. These runtime dependencies, which reflect those encountered within e.g. the `dynamics` group, can be expressed in the configuration. The order of execution of timestep callbacks is defined by the order of appearance of each component within the group configuration, and of the groups themselves. Every component in a group is executed before any subsequent group's component. A similar arrangement independently defines the orders of component initialisation and finalisation.

A component may commonly only give special treatment to the z axis e.g. because of the variation in pressure with altitude, with the same action performed for each column in the grid. A number of such components may be organised into a `column`-type group. Effectively hoisting into the core and fusing the outer loops over x and y, the components execute in the same order as before but now for each column in turn. This may increase temporal locality between components and shrink temporary or intermediate results, yielding better cache utilisation and less demand for memory capacity. The alternative where components execute once per timestep is `entire`-type. Listing 2.1 demonstrates the features discussed here in an extract of the configuration at the beginning of this project.

MONC strictly eschews global state. Every component callback's first argument is of type `model_state_type`. This is a Fortran derived type (a composite data type similar to a Plain Old Data (POD) `struct` in C or C++) which contains the entire current state of the model: more than a hundred components, many of which are themselves derived types. This includes basic and derived properties of the local and global grid and parallel decomposition, the prognostic fields themselves and whether they are active in the current simulation, and both the configuration options database and a large

number of flags by which the physical elements of the simulation are configured.

2.1.2 Advection

During the preparatory phase of this project, computationally-intensive portions of MONC which represented good candidates for acceleration were identified through profiling on ARCHER, the UK National Supercomputing Service. A number of configurations of two scenarios were investigated, varying both scale and enabled simulation components. The first scenario begins with a bubble of cold, dry air at altitude. As it falls, winds blow in from the surrounding area to equalise the region of low pressure left behind. The second scenario is a simulation of the development of a dry, neutral boundary layer without backscatter[1] similar to that used by Brown et al. [4]. As a fluid flows near a surface, the effect of viscosity becomes more prominent, and so a different flow regime may be exhibited. Figure 2.2 illustrates this using a simulation of the scenario described here.

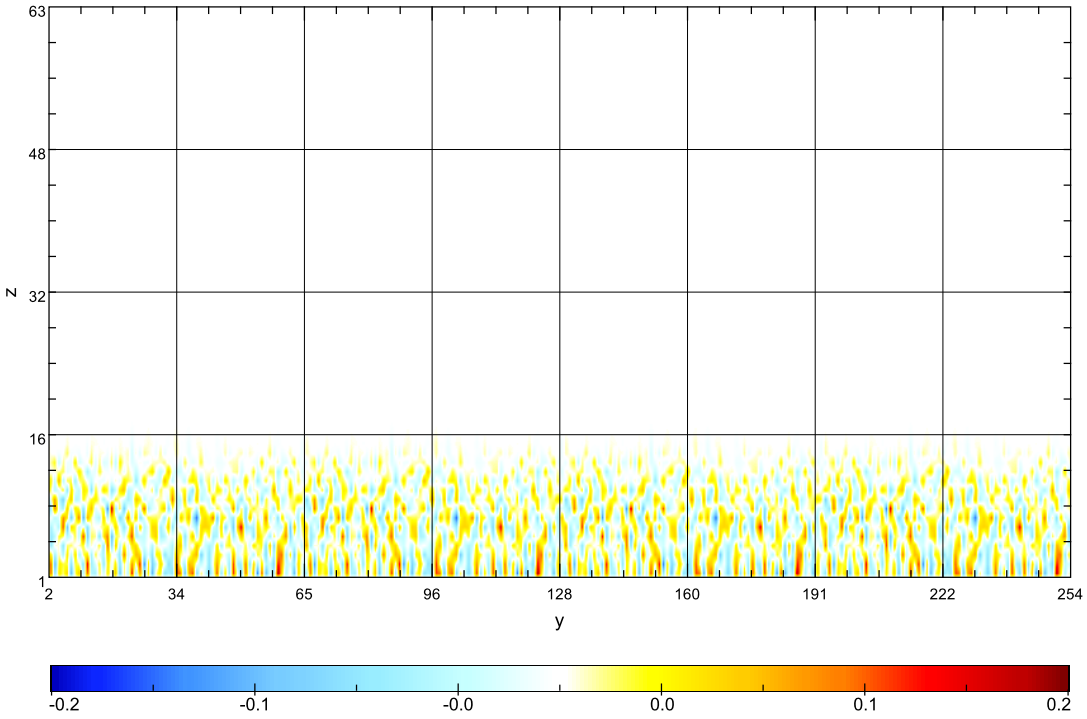
This investigation found the advection components to account for a significant proportion of `USER`[2] time in both scenarios. Advection is the transport of a quantity through fluid flow. The importance to fluid dynamics of this process is reflected in the existence of numerous algorithms for its numerical computation [37]. There is no known best algorithm [37], and so the choice is influenced by the properties most desirable for a particular simulation. MONC implements a pair of advection schemes, either of which may be selected for each individual field in a simulation, allowing the end user to make this choice.

These are implemented in components `pw_advection` and `tvd_advection`. The 'source terms' they produce are accumulated alongside those from other components such as `buoyancy` and `coriolis` before a subsequent component, `stepfields` performs Euler integration over time. All of these components which contribute source terms, and `stepfields` are members of the `dynamics` group.

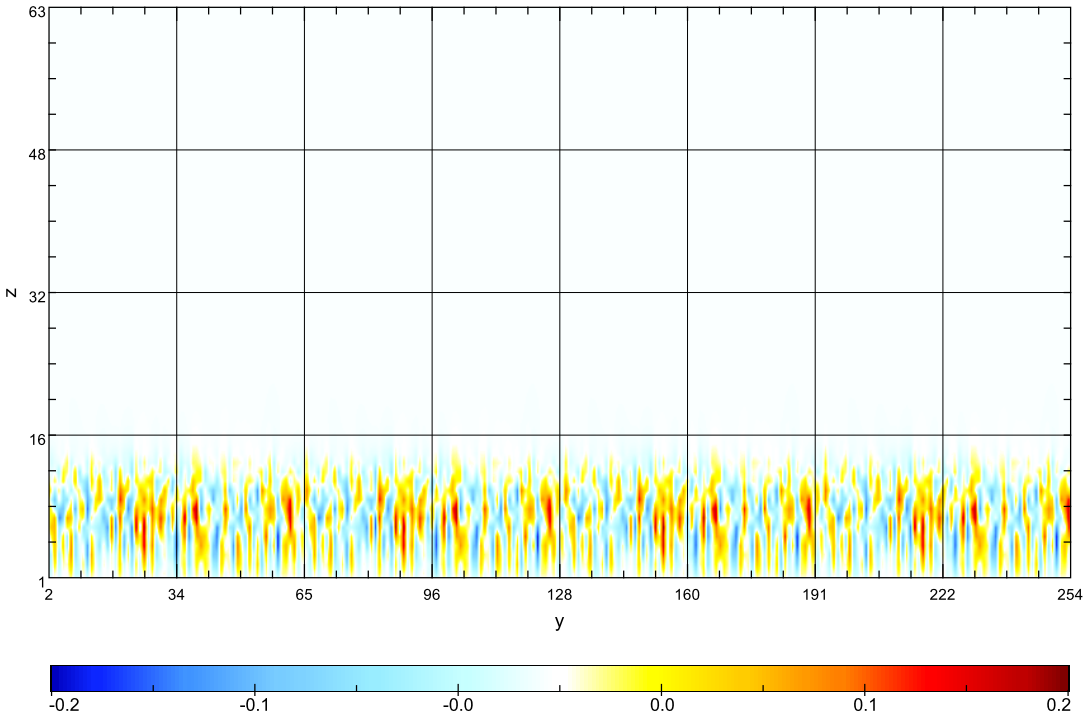
`pw_advection` may reduce to a single loop nest per field, but `tvd_advection` has significantly more complex control flow, to clear effect in the profile report. Table 2.1 shows a profile in which `pw_advection` accounts for around 9% of `USER` time. In contrast, Table 2.2 shows `tvd_advection` to account for more than 60% under an otherwise identical configuration. As expected, the profile of a simulation which combines the two is dominated by the latter. Table 2.3 indicates `tvd_advection` consumes just under 60% of time, while `pw_advection` of temperature does not enter the top 10 functions. All three were obtained at a scale of $512 \times 512 \times 128 = 33.6 \times 10^6$ grid points across 264 processes.

That different advection schemes may exhibit different or orthogonal properties was briefly discussed above. One such useful property is linear conservation: without sinks

[1] Energy transfer from small-scale sub-grid parameterisation to the larger, grid-scale direct simulation.
[2] 40–60% of core time was lost to Message Passing Interface (MPI) communication.



(a) v



(b) w

Figure 2.2: Rate of flow (ms^{-1}) in v and w after a single step of the dry boundary layer scenario. An constant 10 ms^{-1} flow in u is not presented here for space reasons. This plot interpolates between grid points. $x = 128$ of 256.

Time (%)	Function or subroutine
29.1	rearrange_data_for_sending
16.9	contiguise_data
6.1	swap_and_smooth_robert_filter
6.1	*advect_flow_fields*
5.9	tridiagonal_solver
4.9	copy_field_to_buffer
3.1	step_field
2.8	*advect_th_field*
2.8	step_pressure_field
2.6	copy_buffer_to_field

Table 2.1: Top 10 USER functions by percentage of coldbubble time. Functions within pw_advection are highlighted.

Time (%)	Function or subroutine
23.4	*handle_x_direction_fluxes*
18.2	*handle_y_direction_fluxes*
16.8	*handle_vertical_fluxes_middleofcolumn*
10.3	rearrange_data_for_sending
5.8	contiguise_data
3.9	*interpolate_to_dual*
2.2	swap_and_smooth_robert_filter
2.2	tridiagonal_solver
1.7	copy_field_to_buffer
1.6	perform_galilean_transformation

Table 2.2: Top 10 USER functions by percentage of coldbubble time. Functions within, or directly called by tvd_advection are highlighted.

or sources, the total quantity should remain constant. Another is quadratic conservation: the scheme should not artificially introduce or dissipate higher-order quantities such as kinetic energy. The following history reviews development of numerical advection schemes which could provide these properties.

Phillips [32] describes an early General Circulation Model (GCM), accurate for up to around 24 days of simulated time, shortly followed by a catastrophic increase in energy. This instability is due to spatial aliasing, as interference produces wavelengths shorter than 2 grid intervals, and is unaffected by the use of a shorter timestep [33, 36]. Arakawa [1] shows conservation of either kinetic energy or enstrophy[3] is sufficient to eliminate the instability, and derives a suitable scheme for the vorticity equation. Lilly [21] derives a similar scheme for the general 'primitive equations' of atmospheric flow,

[3] $\int_S \omega^2 dS = \int_S (\nabla \times u)\, dS$ for vorticity ω and flow velocity u where $\nabla \cdot u = 0$.

Time (%)	Function or subroutine
21.9	*handle_x_direction_fluxes*
16.9	*handle_y_direction_fluxes*
16.0	*handle_vertical_fluxes_middleofcolumn*
10.9	rearrange_data_for_sending
6.4	contiguise_data
4.1	*interpolate_to_dual*
2.3	swap_and_smooth_robert_filter
2.1	tridiagonal_solver
2.0	copy_field_to_buffer
1.6	perform_galilean_transformation

Table 2.3: Top 10 USER functions by percentage of coldbubble time. Functions within, or directly called by tvd_advection are highlighted. No functions within pw_advection appear.

generalised to the advection equation for incompressible flow on an irregular grid by Bryan [5]. These schemes are conservative, conditional on the flow's divergence being everywhere zero, which is not certain in numerical computation. Piacsek and Williams [34] derive an unconditionally conservative scheme, arranging to combine two similar central differences such that their errors cancel. Piacsek-Williams (PW) advection is implemented by pw_advection as shown in Figure 2.3.

$$s_x = \frac{1}{4\Delta x}\left(u_{x-1}(u_x + u_{x-1}) - u_{x+1}(u_x + u_{x+1})\right)$$

Figure 2.3: One-dimensional advection after Piacsek and Williams [34], per MONC.

Rood [37] links linear interpolation and advection via monotonicity. Leonard [18] introduces the QUICKEST advection scheme, which avoids the previously-discussed instability in central differences by instead using a quadratic interpolation of neighbouring grid points. Arakawa and Lamb [2] describe five distinct grid configurations for the discretisation of variables, identified as A through E as shown in Figure 2.4. These are referenced in subsequent literature as e.g. the Arakawa B grid. Their influence on the simulation is analysed, and grid C found to produce the most general, accurate solutions.

A third property to consider in the choice of advection scheme is the response to large gradients such as might be encountered at a cloud's boundary. Ringing, or artificial oscillation, may be produced in their vicinity. A Total Variation Diminishing (TVD) scheme supresses this response, ensuring that e.g. quantities such as humidity remain non-negative. Leonard [19, 20] introduces ULTIMATE, a 'limiter' to suppress oscillation introduced by an advection scheme itself. tvd_advection implements the ULTIMATE-QUICKEST scheme.

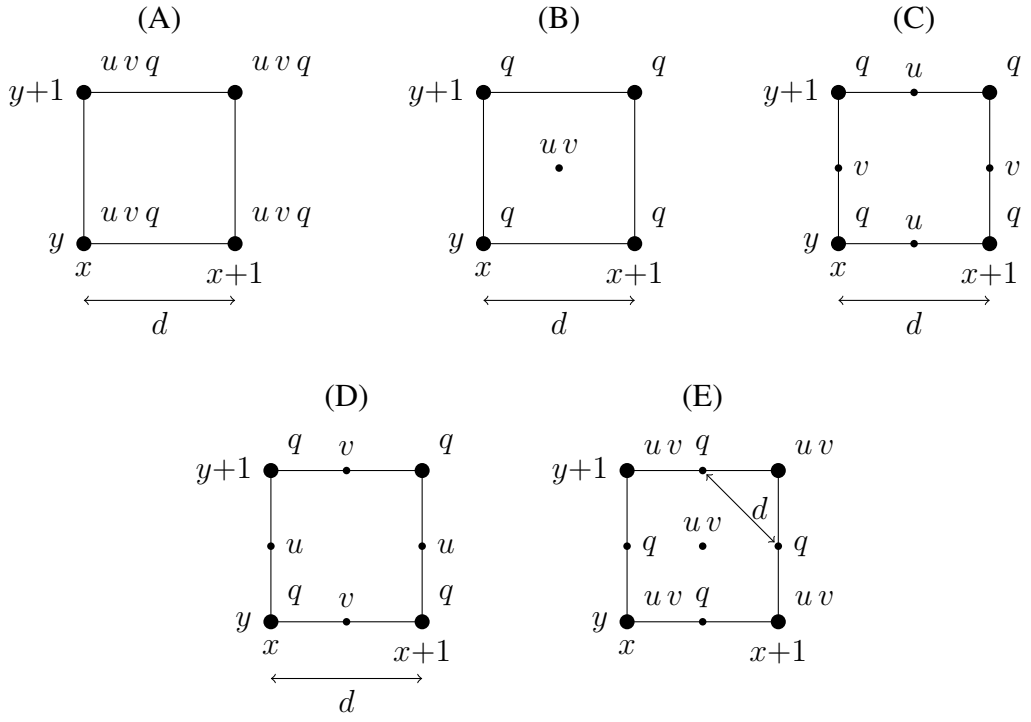


Figure 2.4: Arakawa and Lamb [2] grid configurations.

The Courant-Friedrichs-Lewy (CFL) condition bounds timestep length from above at a given rate of advection: $\frac{v\Delta t}{d} \leq 1$ for speed of sound v, timestep length Δt and grid point separation d [8]. MONC's `cfltest` component maintains this condition by adjusting the timestep at regular intervals. The simulation is ended if the timestep becomes too small i.e. if the rate of advection becomes too large. This has implications for the methodology used in evaluation of performance and correctness.

2.2 Graphics Processing Units (GPUs)

Originally supplying the computer games and scientific/industrial visualisation markets, GPU demand for general purpose computation has increased amongst both HPC and more traditional software developers. Although several vendors and similarities between their products exist, the discussion which follows focuses on the NVIDIA hardware available to this project. GPU performance is contingent upon sufficient available parallelism and appropriate memory usage.

An NVIDIA K20X exposes a Kepler-generation GK110 GPU [29] and its 15 Streaming Multiprocessors (SMs) [27]. Each SM contains 192 single- and 64 double-precision FP cores in addition to other functional units e.g. to provide fast hardware implementations of trigonometric functions [27]. These oversubscribe the available schedulers, which operate instead on 'warps' of 32 threads [26, 27] executing together: if condi-

tional branches cause execution to diverge, the n partitions are executed sequentially, producing an n-fold reduction in performance. Distinct warp and instruction schedulers enable Instruction-Level Parallelism (ILP) of more than one instruction per warp per cycle [27].

The GPU memory hierarchy is more diverse, and subject to greater software control (i.e. more explicit) than the traditional CPU equivalent. A GK110 SM contains three SRAMs: 64 KB which may be split 48/16, 32/32 or 16/48 between an explicit 'shared' region and a transparent L1 cache, 48 KB of explicit read-only data cache, and 65,536 32-bit compiler-managed registers – in increasing order of speed [27, 41, 42]. Above the SMs but still on-die is a single transparent 1,536 KB shared L2 cache [27]. Finally, the NVIDIA K20X provides 6 GB of GDDR5 DRAM 'global' memory which also houses 'local' thread-private spilled registers and large or dynamic arrays [29, 42].

Access to this large off-die memory incurs a penalty of 400–800 cycles [42]. This may be hidden behind further computation if no data dependency exists in the current warp, or otherwise if another warp is available for scheduling [41]. Maximum throughput requires suitably-aligned, sequential memory access within a warp, allowing these to be coalesced into a single access of up to 384 bits [28, 42].

Although significantly faster than global memory, performance of shared memory also depends on the access pattern. Multiple requests for different locations within the same bank must be serviced serially [42]. This scenario can be avoided by a small adjustment of e.g. array stride.

2.2.1 OpenACC

Fortran remains common in scientific computing, and the current MONC code is around 35 kLOC of Fortran 2003. Although historically less-well supported for GPU software development, a number of technologies are now available and were evaluated in preparation for this project. This led to the selection of OpenACC as the technology of choice to accelerate certain MONC components. Also considered were CUDA Fortran, OpenMP 4 and OpenCL.

OpenACC is a standardised extension of Fortran, C and C++ with directives for an accelerator programming model independent of any particular hardware [30]. A sentinel prefix is used to allow a non-supporting compiler to produce a correct if slower executable from the same source. For example, Fortran's sentinel `!$acc` indicates to such compilers that the rest of the line contains a comment. A directive's syntax may include one or more clauses, comparable to procedure arguments. A clause may itself take an argument, which may be a comma-separated list.

A region of the code may be identified for acceleration by the `kernels` or `parallel` directives. A pair of directives may enclose a contiguous block of source lines, or a single directive may identify a loop or loop nest in combination with the `!$acc loop` directive as shown in Listing 2.2. These directives may be used to similar effect, but

```
!$acc kernels
do i = ...
  ⋮
end do
⋮
!$acc end kernels

!$acc parallel loop
do i = ...
  ⋮
end do
```

Listing 2.2: OpenACC `kernel` and `parallel` regions.

their behaviour is very different. The `kernels` directive instructs the compiler to automatically extract a sequence of kernels e.g. loop nests from the region. A `parallel` region instead behaves similarly to the OpenMP directive of the same name, each parallel thread redundantly executing the region. A loop must carry the `loop` directive for its iterations to be divided amongst the threads. Although both regions are specified to end with an implicit barrier, a default Cray compiler extension attempts to identify when these may be elided.

An OpenACC compiler will manage data between host and device, but the default behaviour is found to be overly conservative in Section 3.1.3, the unnecessary data movement causing performance degradation. A number of directives control data movement, summarised in Table 2.4. Each of these takes a set of clauses to describe the movement necessary, some of which are highlighted in Table 2.5. These clauses may also be directly applied to `kernels` or `parallel` directives. The `update` directive may also be used at any point when both device and host copies of a variable are in scope to perform a one-way copy the direction specified by the given `host` or `device` clause.

As OpenACC targets a general accelerator programming model, it does not require

Directive	**Scope**	**Restrictions**
`data`	Region	None.
`declare`	Module or procedure	Only `create`, `copyin` and `device_resident` at module scope.
`enter/exit data`	Dynamic	Only `copyin`, `create` and their `present_or_` variants on `enter`. Only `copyout` and `delete` on exit.

Table 2.4: OpenACC directives affecting data movement between host and device.

Clause	Meaning
`copy`, `copyin`, `copyout`	Two- or one-way copy into and out of device.
`create`, `delete`	Allocate or release device memory without copy.
`present`	Data already present on device. May combine with any copy or the `create` directive using the `present_or_` or `p` prefix.
`device_resident`	Allocate on device only.

Table 2.5: OpenACC data clauses.

any particular correspondence between its three-tiered (gang, worker and vector) parallelism hierarchy and the GPU architecture described in Section 2.2. The Cray compiler distributes gang parallelism amongst thread blocks (a block executes on one SM, but each SM may have several active blocks [26]) and worker and vector amongst warps and threads, respectively [7]. A procedure may carry the `routine` directive to specify which tier of parallelism it provides when compiled for the accelerator, with the optional `nohost` clause to suppress generation of a host copy. The procedure may then be called within a loop to provide the encompassing tiers.

2.3 GPU-accelerated weather models

Other weather or climate atmospheric models comparable to MONC have investigated GPU acceleration, with varying success. This section reviews the output of three such efforts, examining their approach, technologies used, and applicable findings. Table 2.6 summarises these in the same order. Following the preparatory work described in Section 2.2.1, particular attention is given to a pair of OpenACC case studies.

Griffith et al. [14] and Schalkwijk et al. [38] describe a GPU acceleration of an atmospheric LES, rewriting the existing Fortran code in CUDA C++ to achieve a $50\times$ improvement in throughput. Kernels within the simulation are categorised as 'regular' or 'reduction' by whether they produce a new value for each cell in their domain, or a single value for an entire horizontal slab of the domain. The decomposition into threads and thread blocks of each class of kernel is designed to maximise coalescing of memory access. GPU single-precision FP throughput remains higher than double-precision,

Model	Relevance	References
GALES	GPU-only	Griffith et al. [14], Schalkwijk et al. [38]
NIM	Dynamical core, directives	Govett et al. [10, 11, 12]
CAM	Advection, OpenACC	Carpenter et al. [6], Norman et al. [24, 25]

Table 2.6: Comparison of GPU-accelerated weather models.

although this disparity continues to narrow [27]. A numerical analysis found only the FFT-based Poisson solver required double-precision, with single-precision optionally retained elsewhere. To minimise the impact of data transfer between CPU and GPU, almost all computation is carried out on the GPU.

Govett et al. [11] describe the acceleration of a new weather model's dynamical core through a combination of an incomplete directive-based compiler developed for this purpose, and subsequent manual optimisation of the generated code. A number of optimisation strategies are set out. Firstly, fine-grain parallelism is reserved for the vertical dimension, which carries fewer data dependencies. Data transfer is to be minimised or eliminated, with data layout to enable coalescing of what remains. Finally, to maximise utilisation, per-block usage of shared memory and registers is minimised e.g. by partioning kernels. Govett et al. [12] find for devices of the same generation, the addition of 1 or 2 GPUs per node yielded an improvement of 1.75 or $2.45\times$, respectively. Further additions degraded performance due to PCI Express (PCIe) bus contention. Essential to both performance and favourable cost-benefit analysis of accelerator purchase was simultaneous or 'symmetric' use of both CPU and GPU.

Carpenter et al. [6] detail the GPU acceleration of advection kernels within a mature atmospheric model, the source of 25% of runtime in the configuration evaluated. The intervening release of CUDA Fortran soon after publication of [14] eliminated this pressure to rewrite in CUDA C or C++. Norman et al. [24] emphasise the need for agility in response to new scientific demands. GPU parallelism is provided by pushing loops lower in the call tree, but at the cost of decreased performance on the CPU. Elimination of redundant transfers and the use of shared memory improve performance, as do kernel fusion and overlap of asynchronous data transfers with kernel execution. MPI buffers are packed on the GPU to further reduce device-to-host communication. Validation is provided by showing the deviation of the GPU implementation from the CPU is strictly less than that produced by a rounding-level perturbation to the initial conditions on the CPU alone.

Norman et al. [25] use a port of this same code from CUDA Fortran to OpenACC to compare their performance, maturity and ease of use. A similar effort was undertaken by Govett et al. [10] with reference to OpenACC's performance relative to their own directive-based compiler described above. Norman et al. [25] identify the expected benefits of OpenACC: portability, automatic management of decomposition and transfers between device and host memory, and less exposure of the user to architectural details of device memory.

Norman et al. [25] observe that OpenACC optimisation often reduces performance on traditional architectures. As a result, they suggest that a single source code may not be acceptably portable, especially as new scientific development often takes place on these platforms. Although Govett et al. [10] describe a literal single source code, this is achieved in part through conditional compilation. Both Norman et al. [25] and Govett et al. [10] further claim portability between available implementations is not yet practical, hindered by incompatible interpretations of the specification, and by numerous compiler bugs. Norman et al. [25] consider the Cray compiler "closer to a usable

state" than the The Portland Group, Incorporated (PGI) alternative. In particular, Norman et al. [25] claim the current state of user-defined type support makes broad use of OpenACC in modern code impractical.

Both Norman et al. [25] and Govett et al. [10] highlight performance limitations arising from lack of programmer control over the type of device memory used for allocations. Although Norman et al. [25] identify the abstraction leading to this loss of control as a potential benefit, the compiler must make acceptable decisions for performance. Available compilers made little [10] to no [25] use of the high-speed shared memory region. Norman et al. [25] praise the implementations for nonetheless approaching 65% of CUDA performance. Govett et al. [10] investigate other currently-missed opportunities for compiler optimisation. The most significant of these is 'variable demotion' whereby one or two array dimensions is replaced with thread-local copies, for improved cache and register usage. Also important was loop manipulation to maximise available thread-level parallelism and ensure memory access coalescing.

2.3.1 Project-oriented summary

GALES achieves excellent performance at the expense of a complete rewrite in CUDA C++, which would not typically be a palatable option for a heavily-used model [14]. This project's aim is to investigate whether OpenACC provides suitable ease-of-use and performance to justify directive-based acceleration of established weather models at the Met Office and elsewhere.

More relevant to this goal are the NIM and CAM, as they accelerate the same or similar parts of the model as this work does: both focus on dynamics, and the CAM on advection in particular [6, 11]. This 'offload' style produces much more modest performance improvements in exchange for greatly reduced implementation cost [6, 10–12, 24, 25]. These projects also produced evaluations of OpenACC's current state, finding it difficult to use due to key deficiencies in the specification, but more commonly due to poor implementation quality [12, 24]. Norman et al. [25] believe performance is acceptable despite these limitations, which suggests promise if these may be alleviated. They also believe, as also suggested by this project's preparatory work, the Cray compiler to be the only practically usable implementation at this point.

Although many broadly-applicable optimisation tactics were discussed, some of these are especially relevant to MONC. MONC uses double-precision FP arithmetic by default, but this is easily reconfigured in the source. An numerical analysis for GALES found that single-precision was acceptable throughout, with the sole exception (the solver) outside this project's scope. This suggests an evaluation of the influence of FP precision on performance. Govett et al. [11] advocate for fine-grained (i.e. vector) parallelism in the z axis, which is also appropriate given MONC's array index order.

This project will convert `column`-type components to `entire`-type to expose parallelism for the GPU, as in the CAM. As these GPU-targeted changes were found to reduce performance on the CPU, neither the NIM nor CAM maintain a true single source

code [11, 24, 25]. Maintainability concerns remain, but MONC's component system allows these to be easily toggled at runtime.

Norman et al. [25] identify abstraction of accelerator architecture as a key benefit of OpenACC. This is relevant to MONC in particular as the design considerations highlighted by Brown et al. [4] include usability and extensibility of the model by atmospheric rather than computer scientists.

Chapter 3

Design and implementation

Chapter 2 establishes that advection is an important, and computationally expensive component of MONC, an atmospheric LES with a modular design. With reference to work on similar numerical weather models, the potential performance benefits of GPU acceleration are set out alongside the challenges it poses for development. OpenACC was chosen from a range of technologies to allow a Fortran code such as MONC to target accelerator devices, and a number of project-relevant features are described. Motivated by this background, this chapter describes the GPU acceleration of two advection components using OpenACC, including coverage of design considerations and implementation practicalities.

3.1 Piacsek-Williams (PW) advection

Section 2.1.2 found MONC's TVD advection implementation especially complicated. To better understand accelerator-specific elements of the project before encountering increased domain complexity, and cognizant that infrastructure developed for one scheme might also be used for the other, the acceleration of PW advection was undertaken first.

The importance of the treatment of data, its movement and access pattern, to performance in GPU acceleration is known, and Sections 2.2 and 2.3 respectively detail technical considerations and heuristic 'best practices' to address this. Analysis of the existing `pw_advection` component thus began with an investigation of which data were used and where, and an estimate of their total size.

`pw_advection` contains three subroutines to advect each of the vector flow (with components u, v and w), and scalar θ (temperature) and q (tracer[1]) fields. These will become the GPU kernels. As discussed in Section 2.1.2, each field may use a distinct advection scheme, and so any of these three may not be used under a given configuration. Each subroutine takes a collection of small scalar and array constants e.g. grid

[1] e.g. water vapour or particulate pollution.

```
group_names=..., dynamics-fork, dynamics, ...
dynamics-fork_group_type=entire
dynamics-fork_group_contents=pw_advection_acc
```

Listing 3.1: `dynamics-fork` configuration.

size and interval, and at least one pair of much larger input and output fields. All three subroutines require the relevant field's current and source fields as input and output, respectively. θ and q also depend upon the current flow field. These constant, input and output data are contained within the current model state provided by the single `model_state_type` argument provided to each component callback as described in Section 2.1.1.

`pw_advection` and `tvd_advection` are both members of the `dynamics` component group, contributing source terms for integration by `stepfields` as described in Section 2.1.2. Configured as `column`-type, components in this group are invoked for each column in the local grid, effectively hoisting the outer loops over x and y into MONC's core as discussed in Section 2.1.1. To expose parallelism for the GPU, it is convenient to tightly nest these loops lower in the call tree where they may be prefixed with a single directive. Although the core sets flags to indicate e.g. whether a column is within the local halo, these are used by `pw_advection` only to skip halo columns. An equivalent outcome is obtained by reducing the range of the reintroduced loops.

To make these changes without compromising the ability to easily evaluate correctness or performance, a new component was created: `pw_advection_acc`. As the implementation was self-contained, with few dependencies on MONC's infrastructure, the new component was created *ex nihilo* rather than by copying the existing code and modifying it in place. Ensuring that the component was not then redundantly invoked for each column, Listing 3.1 shows the configuration of a new `dynamics-fork` component group. Each field could then be ported individually to the new component, with those remaining advected by the existing component. No modification of `dynamics` itself is necessary. This maintained the ability to run and (ideally) receive an identical result through development. As explained in Section 2.1.1, configuration of even a newly-created group is done entirely at runtime.

3.1.1 Derived types

The first difficulty encountered was that OpenACC does not meaningfully support derived types in Fortran, nor their equivalent in C and C++. Although documented in the case study[2] by Norman et al. [25], this was not highlighted in introductory material available, nor in the OpenACC specification. Further examination reveals the equivalence between identifiers in OpenACC clauses is never exactly specified. An implementation might feasibly conform whether or not support for derived types is provided.

[2]Published after the literature review undertaken in preparation for this project.

```
!$acc parallel loop &
!$acc copyout(current_state%su%data(k, y, x))
do x = ...
  ⋮
end do
```

Listing 3.2: Derived type usage leading to internal compiler error.

```
!$acc parallel loop copyout(su%data, sv%data)
do x = ...
  ⋮
end do
```

Listing 3.3: Derived type usage leading to incorrect diagnostic.

```
!$acc parallel loop present(current_state)
do x = ...
  ⋮
end do
!$acc update device(current_state%u%data)
```

Listing 3.4: Derived type usage leading invalid code to be emitted.

Perhaps the main problem with support for user-defined types is the presence of weak references or pointers to other objects in host memory. Either these must be absent from the device copy of the type, or the structure recursively copied. The Cray compiler does provide limited deep copy support, but research exploring the semantics of user-defined type transfer continues [3].

Unfortunately, lacking external documentation is compounded by erratic compiler behaviour in the presence of derived types. This was a cause for confusion until the underlying cause was discovered. For instance, given appropriate context and under the Cray Compiler Environment 8.3.10, Listing 3.2 produces an internal compiler error, Listing 3.3 generates a diagnostic warning `data` is repeated in the `copyout` construct, and Listing 3.4 compiles without error, but results in a segmentation fault after a null pointer is passed to CUDA library subroutine `cuMemcpyHtoD` at runtime.

An attempt was made to use Cray's proprietary deep copy support[3] but compiler quality was again problematic. Having already classified data as input, output, or constant, the existence of a single variable of derived type for each would reduce the code necessary to copy these to the device or host together at the appropriate time. Each class was addressed in turn, for low code velocity. First, it was found that the compiler diagnostic error "unsupported deep-copy type (function returning void)" was generated unless a

[3]Enabled by adding `-h acc_model=deep_copy` to the compiler flags.

compiler directive was used to disable automatic inlining as shown in Listing 3.5. This workaround allowed a derived type to be used for the simulation constants as described.

```
subroutine advect_q_fields_accelerated(...)
  ⋮
  !dir$ !dir$ inlinenever advect_q_field_accelerated
  do n = 1, current_state%number_q_fields
    call advect_q_field_accelerated(...)
  end do
end subroutine advect_q_fields_accelerated

subroutine advect_q_field_accelerated(...)
  ⋮
  !acc parallel loop ...
  ⋮
end subroutine advect_q_field_accelerated
```

Listing 3.5: Automatic inlining of `advect_q_field_accelerated` leads to internal compiler error.

As the input variables were provided a similar treatment, very similar code produced the internal compiler error "unexpected missing symbol␣␣in stack layout map replace"[4], followed by the GNU C Library detecting heap corruption within the compiler and an immediate exit. A workaround was not identified, ending development of this 'derived types' branch. Further, it was discovered that only synchronous deep copy transfers are supported. Verbose runtime diagnostic output[5] indicates asynchronous deep copy transfers fall back to synchronous execution.

The conclusion drawn from this investigation was that any component of a derived type used within an OpenACC region or directive must first be unpacked i.e. copied to a separate variable of non-derived type. For a modern code with such pervasive usage of derived types as MONC, this can be arduous. To reduce the boilerplate of a sequence of assignment statements, variables were at least initially unpacked into subroutine arguments as shown in Figure 3.6. This is acceptable for a few arguments, but no good solution was found for a larger number e.g. as in `tvd_advection`. Alterations discussed in Section 3.1.2 to support asynchronous kernels partially mitigate this boilerplate.

3.1.2 Asynchronous execution

A planned feature of this project was asynchronous execution of the accelerated components, overlapping computation with data transfer between host and device to reduce

[4] Seemingly empty symbol emphasised here.

[5] `CRAY_ACC_DEBUG=$n` for $0 < \$n < 4$.

```
call advect_q_field_accelerated(
  current_state%u%data, current_state%v%data,&
  current_state%w%data, current_state%q(n)%data,&
  current_state%sq(n)%data,&
  current_state%local_grid%size,&
  current_state%local_grid%halo_size,&
  current_state%global_grid%configuration%horizontal%cx,&
  current_state%global_grid%configuration%horizontal%cy,&
  current_state%global_grid%configuration%vertical%tzc1,&
  current_state%global_grid%configuration%vertical%tzc2)
```

Listing 3.6: Derived type unpacking into subroutine arguments. Were it possible, it would be preferable to pass just `current_state`.

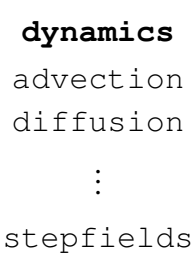
apparent latency. Maximum efficiency is achieved by balancing the two such that each result becomes available at the same time, and neither device idly waits for the other to finish. Literature reviewed in Section 2.3 found concurrent CPU and GPU execution key to realisation of the latter's potential benefits.

As implied by the discussion of component order in Section 2.1.1, the window for asynchronous execution of advection begins each timestep with entry to `dynamics-fork` and ends immediately prior to the last component of `dynamics`, `stepfields`. Every other component in this pair of groups sums the source terms it produces for each field with the other accumulated source terms before `stepfields`, the final component steps these forwards in time. Any `dynamics-join` counterpart must then be of `entire`-type to match `dynamics-fork`, and must take place before `stepfields`. Figure 3.1 illustrates the introduction of first `fork`, and then `join` as seen here.

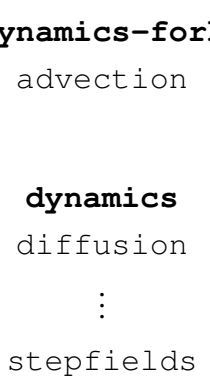
To minimise the risk of misconfiguration which might arise from unnecessary proliferation of components, it was chosen to reconfigure `stepfields` as the only member of the `dynamics-join` group. The component was modified in-place to operate in `entire` rather than `column` fashion. This again entailed reintroducing the x and y outer loops, ensuring that code which should run exactly once per timestep fell outside these loops, and mechanically replacing references to the column index components of the current state. This change was made before the addition of directives for asynchronous execution.

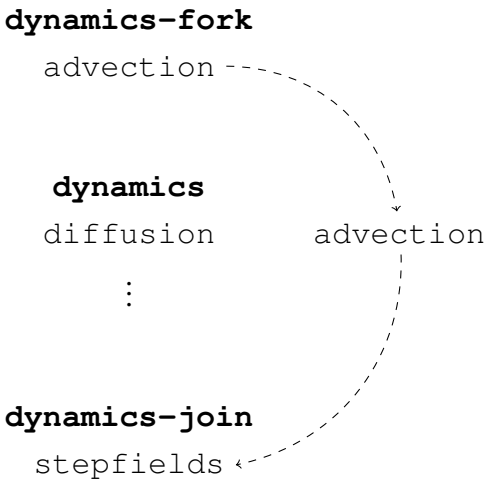
Although hardware and software might be devised to permit e.g. the host's source terms to be incremented in-place as the device data is copied back, this is not yet available. The device data must instead first be received into a buffer in host memory, and the host processor itself must combine the two. This may increase host memory load by up to 100% of the baseline per device. For applications already bound by memory capacity per node, it may be better to find a decomposition which avoids device duplication of host data. The potential for this problem is noted, but does not in practice affect this project as MONC is not so constrained.

(a) Original



(b) Synchronous GPU advection





(c) Asynchronous GPU advection

Figure 3.1: `fork`, then `join` split from `dynamics`.

```
!$acc declare create(su)
real, dimension(:, :, :), allocatable :: su
```

Listing 3.7: OpenACC `declare` directive.

Where in MONC to allocate buffers was not readily apparent. Per Section 2.1.1, components are built independently so e.g. `stepfields` cannot access even public components of `pw_advection_acc`. OpenACC and derived types were found to be incompatible by Section 3.1.1, precluding extension of the `current_state` argument passed to each component.

The first approach investigated was to create a MONC core module to store accelerator data. As part of the core, this would be visible to all components. An OpenACC `declare` directive was used to allocate the device arrays, as seen in Listing 3.7. However, the corresponding device symbol does not appear to be emitted with correct linkage, as use from another module provokes a diagnostic error[6] from the GPU linker. Whether this indicates user or compiler error is unclear: per Section 3.1.1, the equivalence between identifiers e.g. with and without a module name prefix is unspecified.

The next approach was to use MONC's built-in support for component field publication. The infrastructure requires implementing a pair of callbacks to provide the field information e.g. type and size, as well as the value itself. A list of published field names must also be added to the component's descriptor. These names may be used by other components to indirectly invoke the two callbacks. Within the component, these fields are just conformable array components of a module. This is the mitigation for the unpacking of derived types alluded to in Section 3.1.1.

Another series of challenges were overcome to place these published fields on the GPU using OpenACC. To what extent their provenance lies with the user or the toolchain is unclear, as discussed below. Firstly, it is not possible to use a `declare` directive with the `create` clause as shown in Listing 3.7 in combination with the `copyout` clause for an OpenACC region to update the host copy. Attempting to do so produces 'INTERNAL COMPILER ERROR: "Duplicate or non-conforming acc data contexts in construct 1"'.

The OpenACC specification suggests this may be justified. The `create` clause does not delete the host instance of the variable, but does imply "any values computed and assigned on the accelerator are not needed back in local memory". This may be read as either a specification of the default behaviour, or a strict limitation on the variable's usage. If the latter is the correct understanding, it is inconsistently applied. No diagnostic or internal error is produced when the `copyout` clause is replaced with a separate `update` directive as shown in Listing 3.8.

Whether a variable allocated on the GPU with a `create` clause may be updated on the host using any directive colours the interpretation of another another pair of bugs: either

[6] "nvlink error: undefined reference to su__cray_acc"

```
!$acc update host(su)
```

Listing 3.8: OpenACC `update` directive.

user error which the compiler fails to identify, or incorrect code emission. The combination of `declare` and `update` directives appears to build successfully, but produces an error at runtime. The `update` directive does not find the relevant arrays registered present on the GPU, despite successful execution of the GPU kernel itself. The verbose runtime diagnostic output yields no indication that the array is ever unregistered.

An alternative to the `declare` directive is the `enter data` and `exit data` directive pair. The difference between the two is lexical vs. dynamic lifetime: the `enter` and `exit` directives may exist e.g. in different procedures. When the combination is `enter` and `exit data` with `update`, a different error is produced. The `update`'s 'client', `stepfields` encounters a segmentation fault when attempting to use the updated host data. The verbose runtime diagnostic output indicates that the transfer failed: "ACC: ERROR: unknown transfer type 32767". This error was encountered several times during the course of this project, but with no obvious similarity in circumstances which might illuminate the underlying cause. In particular, the same combination of directives in a later revision appears to execute without error.

Listing 3.9 provides a simplified 'sketch' of the final asynchronous code, with published fields, multiple fields, and individually toggleable fields omittted for brevity. Although not used in `pw_advection` itself, later components require a zero halo. As this initialisation is done on the host, the `enter data` directive is used with the `copyin` clause instead of `create`. Both host and device allocations are released in the component finalisation callback, and where the real code uses multiple `enter data` directives, their `exit data` counterparts are executed in reverse order, perfectly nesting their pairings.

The `present` clause is used with the `parallel` region[7] to suppress any redundant automatic data movement. The `async` clause requests asynchronous execution, with a variant which allows a non-negative integer handle for the operation to be specified. This allows the `parallel` region and `update` to be sequenced, and the published field retrieval callback to `wait` for their completion before returning the value. As soon as the asynchronous operations are enqueued, we can proceed to the unaccelerated `dynamics` components.

`stepfields` checks whether the `pw_advection_acc` component is enabled and in turn whether each published field is enabled. Enabled accelerated fields are added into the host-only fields attached to the model current state type before these are integrated forwards in time.

[7] Here combined with the `loop` directive, but the clause is applicable to `parallel` only.

```
real, dimension(:, :, :), allocatable :: su

subroutine initialisation(...)
  allocate(su(...))
  su = 0
  !$acc enter data copyin(su)
end subroutine initialisation

subroutine advect_flow()
  integer :: x, y, z

  !$acc parallel loop async(10) present(su)
  do x = ...
    do y = ...
      do z = ...
        su(z, y, x) = ...
      end do
    end do
  end do

  !$acc update host(su) wait(10) async(11)
end subroutine

subroutine finalisation(...)
  !$acc exit data
  deallocate(su)
end subroutine finalisation
```

Listing 3.9: OpenACC asynchronous execution.

3.1.3 Data management

As noted in Section 2.2.1, OpenACC has enough information for inefficient-but-correct automatic movement of data between host and device. Typically, this takes the form of an automatic copy in upon entry to and copy out upon exit from an OpenACC region. For kernels with distinct inputs and outputs, `copyin` and `copyout` clauses allow either of these to be omitted. By separating the computation from the data management, improvements are obtained by avoiding repeated device memory allocations and data-sharing between kernels. The same separation was already necessary to support asynchronous execution of GPU kernels, so the work of this section is to describe the optimisation of the infrastructure developed in Section 3.1.2. Table 3.1 shows these changes produce a significant ($2\times$) reduction in data transfer from the original management which went no further than to apply `copyin` clauses to loop directives.

	Field	Host-to-device (GB)	Device-to-host (GB)
Original	u, v, w	5.6	2.8
	θ	4.6	0.9
	q	9.3	1.9
	Total	19.5	5.6
Improved	u, v, w	0.0	2.8
	θ	0.9	0.9
	q	1.9	1.9
	Common	2.8	0.0
	Total	5.6	5.6

Table 3.1: Data transfer for five model minutes at $256 \times 256 \times 64$ with two q fields.

Earlier, the variables upon which the kernels depend were categorised as input, output, or constant. Little can easily be done to reduce output size, or the frequency of input transfer. Lossless compression might improve transfer rate, but was not practical to implement within the time available for this project. As a result, the focus is on transferring inputs shared between multiple kernels (only u, v and w, per Section 3.1) only once per timestep, and on transferring constants (the values of which depend on the configuration and so are not available to the compiler) only once per simulation. These two categories of movement are represented in the 'common' row of Table 3.1.

The source term array for each field is transferred into the array at most once, if that field is enabled for advection by `pw_advection_acc`, during component initialisation. All transfers during initialisation are synchronous, as there was no performance case to be made for the incremental complexity of asynchrony. The newly-computed source term array is transferred back exactly once per timestep for each enabled field. All three fields depend on the current state of the flow field, the GPU copy of which is updated at the start of each timestep. The other input fields are also transferred once per timestep, with the multiple three-dimensional q fields packed into a single four-dimensional array for convenience and to reduce kernel launch overhead.

All simulation constants are transferred to the device during initialisation if any field is to be advected. Although some constants might be omitted in the absence of certain fields, the improved maintainability of a single source location for transfer of constants is preferable to what would be, given their very small size and the once-only nature of initialisation, a premature optimisation. There is no host memory allocation except during initialisation, and the only device allocation each timestep is for the θ field input, as this is neither shared between kernels as in u, v and w, nor packed as in q.

3.2 ULTIMATE-QUICKEST (TVD) advection

As mentioned in Section 2.1.2, `tvd_advection` has a more complex implementation than `pw_advection`. Consequently, although development begins with the same step of creating a `tvd_advection_acc` component within `dynamics-fork`, the existing implementation was copied-and-pasted rather than beginning from an empty component as before. This avoided time wasted retyping boilerplate e.g. to generate stencils for interpolated fields.

Again, three top-level subroutines provide entry points for the advection of each field. These compute flux through three faces of the cell cube surrounding each grid point in a column: below in z, and to the left in both x and y directions as shown in Figure 3.3. A sum of their finite differences is used to update the source field. Figure 2.4 shows how the Arakawa C grid computes flow and scalar fields on offset grids. To advect the flow field, the current u, v and w components are interpolated between these 'primal' and 'dual' grids. When calculating flux for a single component of the source field, the three interpolated components and one corresponding uninterpolated component e.g. for `su`, `u` itself is required.

MONC uses a two-dimensional MPI cartesian topology, with a geometric decomposition in the x and y dimension. y-direction flux is sent from the second i.e. innermost halo row of the first process in y to the last non-halo row of the last process in y as shown in Figure 3.2. As `tvd_advection` is a member of `dynamics column`-type component group, this communication takes the form of a non-blocking, unbuffered point-to-point MPI send and receive for each column in these rows.

The `tvd_advection` base for the new component contained four similar but not

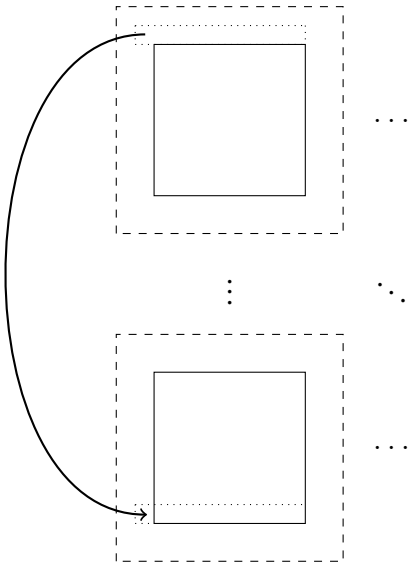


Figure 3.2: Wraparound in the y dimension of flux through the left (in y) grid cell face.

identical advection subroutines: one for each flow field component, and one common between the θ and q scalar fields. To avoid duplication of effort, these implementations were merged to produce a single subroutine. This would then be provided different arguments for each field. Although this proved useful to identify which parts of the implementation might vary, the single routine was again split into scalar and flow field versions to maintain tightly-nested loops[8].

Table 2.2 shows 94% of `tvd_advection` `USER` time is spent in the flux calculation procedures, justifying a focus on these. As OpenACC regions may not contain IO, the goal was then to rewrite the advection subroutines as a series of separate tightly-nested loops over x and y, each containing a call to one or more subroutines which provide the innermost loop over z. These loops would then form the basis for a series of GPU kernels. This process was complicated by the translation of conditions on a column's coordinates into ranges in those dimensions, and by identification and elimination of the loop-carried dependencies which were implicit in the original `column`-type formulation.

3.2.1 Loop-carried dependency

As mentioned above, the advection subroutines contribute source terms using a finite difference of the flux through the grid cell face to the left in x and y, and below in z. A column provides the necessary information in the z direction, but x and y require further treatment. To calculate a forward difference in the x direction, the x face flux from the next column in x must be available. `ultflx` achieves this by calculating x face flux one column ahead in x as shown in Figure 3.3. The previous column's 'next' flux is copied to the current flux before being updated. Any confusion between `flux_previous_x` and `y` may be dispelled by recalling that whilst both are calculated at the previous column, only the latter is measured there.

As described in Section 2.1.1, MONC's core timestep procedure invokes `column`-type components for each column in the local domain, including halo columns. Where `pw_advection` advects only the interior or non-halo columns, `tvd_advection` begins work from the last non-halo column in both x and y as Figure 3.2 suggests. This allows the backward difference in y to be calculated without neighbour communication, and initialises `flux_previous_x` which will become the flux at the first non-halo column.

These loop-carried dependencies were eliminated by loop fission: flux, then differences calculated over the entire grid at once, rather than both for each column in turn. `flux_previous_x` and `y` are obtained from the entire-grid `flux_x` and `y` at `y-1` and `x+1`, respectively. The copy in `ultflx` of the next flux to the current column must also be removed, otherwise it will overwrite the current column's previously-calculated flux with the not-yet-calculated i.e. invalid flux from the next column. Additionally,

[8]The specific difficulty was fixing `ultflx`'s `x_` and `y_flow_index` arguments for the flow fields, but incrementing them in step with their `scalar_index` counterparts for the scalar fields.

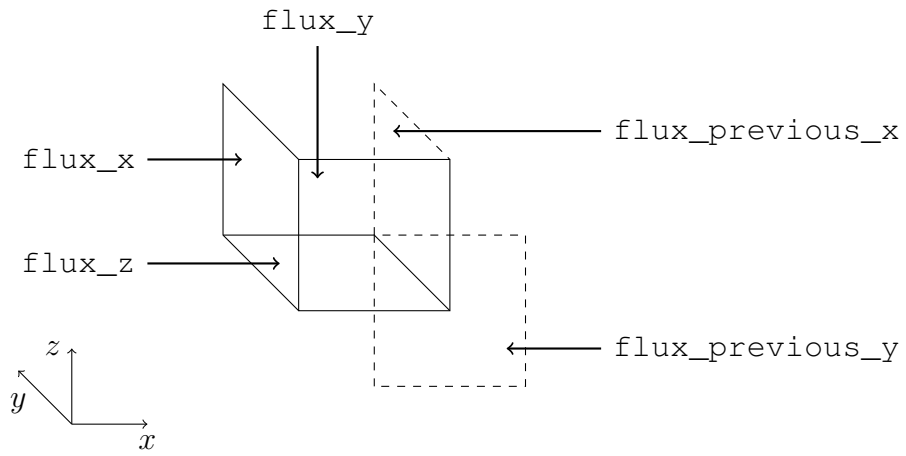


Figure 3.3: Flux through, and orthogonal to, grid cell faces in `tvd_advection`.

MPI communication of flux in y could then use a single slice in x and z, rather than a sequence of columns in z.

3.2.2 Attempted directive use

More derived types, and more derived type components are used by `tvd_advection` than `pw_advection`. These were replaced with an argument for each component used in `ultflx`, and a subroutine was used as described in Section 3.1.1 to unpack these in the scalar and flux advection subroutines. Having removed the impediments previously identified in Section 3.1, OpenACC directives were added to enable GPU execution of these kernels. As the flux calculation operates on columns, these procedures were annotated with the `routine vector` directive and clause described in Section 2.2.1 to indicate that these would provide fine-grain i.e. thread parallelism within the kernel. A `parallel loop` directive was also added, with clauses to indicate the coarser tiers of OpenACC's parallelism hierarchy were to be found in the combined iteration space of the outer loops over x and y: `collapse(2) gang worker`.

Development was forced to halt at this stage as no way was found to launch the GPU kernel successfully. As described, the NVIDIA CUDA procedure `cuLaunchKernel` fails, indicating an invalid device memory access. Executed under the Allinea Forge debugger, a segmentation fault is encountered. Below `cuLaunchKernel`, this appears to occur within `cudbgGetAPIVersion`. However, repetition of this procedure within the call tree displayed suggests the stack may be corrupted. If the `kernels` directive described in Section 2.2.1 replaces `parallel loop`, the Cray OpenACC runtime reports "invalid dope vector[9] version" which may again suggest memory corruption. These errors are encountered regardless of whether the directives carry data clauses i.e. whether manual data management is used or not. Compiled with OpenACC

[9] A structure representing a Fortran array, containing e.g. bounds and a pointer to the array contents.

disabled, the same code runs to completion on the CPU even with Valgrind's 'thorough' validation of memory access.

3.3 Development process and tools

As discussed above, efforts were made to ensure work in progress was always able to run to completion. This iterative development process made it very easy to validate the new code by reference to the existing code. MONC can write a checkpoint at a regular, configurable interval e.g. every hundred timesteps. A Python script was used to compare the absolute and relative difference between a pair of checkpoints, using the `Scientific.IO.NetCDF` library to read the NetCDF file format.

As Section 2.1.2 mentions, MONC contains a component which adjusts the timestep to maintain the CFL condition limiting timestep length for a given rate of advection. As a result, the number of timesteps necessary to simulate a fixed period of model time under a given configuration should also vary little. This provided a means to detect numerical or correctness errors which were hidden by a reduced timestep.

The synchronous PW advection was within 0.05% of the original. Asynchronous kernel execution introduced an error of up to 3.0% in v and w after five model minutes or 200 to 250 iterations. As the kernel itself was unchanged, this is believed to arise from the reordering of FP operations by effectively moving the addition of advection source terms from the first to last operation performed before `stepfields`.

Difficulty encountered in the development of TVD advection forced a halt to development, leaving features such as asynchronous execution and efficient data management unimplemented for this component. A number of workarounds were attempted, but to no avail. This is believed to expose a memory bug e.g. a buffer overrun within the OpenACC runtime. Although the relevant directives could be included in the code, they would remain untested due to the inability to launch the GPU kernel. Despite their omission in `tvd_advection_acc`, the necessary process and infrastructure are described in Section 3.1 with reference to the `pw_advection_acc` implementation. Mitigation for this truncated implementation's effect on this project is in place.

Section 3.2.2 mentions that the code has been built with OpenACC disabled and executed on the CPU to check the correctness of the transformations made to expose parallelism, and that it then runs to completion with no memory errors indicated. The physics within e.g. `ultflx` has not been modified, suggesting many classes of error may not be present.

`tvd_advection_acc` results are observed to initially adhere closely to the original, but an error at $z = 64$ is propagated by the advection itself as time progresses. This is believed to arise from a combination of `ultflx`'s existing operation over only a subrange within a column, and this project's removal of the loop-carried dependencies described in Section 3.2.1. Manual data management may be the only 'unsafe' feature

of the OpenACC specification, in the sense that an error may produce an incorrect result rather than inefficiency, or a compiler or runtime diagnostic. As this feature is absent, some confidence may be had that the directives provided are correct and demonstrative for any future development effort.

Chapter 4 also investigates the effect of increased computational load within the original and OpenACC PW advection implementations, and uses this to assess by proxy the likelihood that a complete `tvd_advection_acc` implementation would produce performance benefits to justify continued development effort were a workaround found or fix released for the difficulty encountered.

In preparation for this project, a work plan was produced which divided time for development equally between PW and TVD advection. This was driven by an evaluation of OpenACC which, in demonstrating a $3.5\times$-accelerated Gauss-Seidel solver, encountered only limited difficulty when using the PGI compiler, and none with the Cray alternative. This was felt to have sufficiently derisked the OpenACC aspects of the project. As discussed in Section 3.1, it was also felt prudent to begin with PW advection.

This plan was appropriate given what was then known, but it became clear in the course of the project's main effort that the implementation challenges had been underestimated, especially surrounding the use of derived types. The difficulty of estimating software development schedule, particularly when working with unfamiliar technology, is well-known. The decision was made to reschedule time originally planned for TVD, using it instead for PW. This allowed at least one component's implementation to be completed and evaluated. As much of the infrastructure developed e.g. for asynchronous execution as in Section 3.1.2 is applicable to any accelerated component, and given the mitigation described above, this change to the work plan does not significantly impair this project's aims.

Version control was used for the changes to MONC, and for configuration files and utilities[10] to support the development effort. Feature-level commits were made to MONC's NERC-hosted Subversion repository, but Git was also used to manage changes with a finer granularity, and to manage resources e.g. for evaluation which were not externally relevant. Risk of data loss was mitigated by NERC's backup policy and the use of a GitHub private repository.

[10]For checkpoint comparison, job submission, and CSV formatting of evaluation results.

Chapter 4

Results and evaluation

This chapter evaluates several aspects of the OpenACC PW advection component described in Section 3.1, including a broad examination of performance, a comparison of design against existing GPU-accelerated weather models described in Section 2.3, and a discussion of the current state of OpenACC specification and implementation.

4.1 Performance

Development and subsequent performance evaluation was carried out on the Swiss National Supercomputing Centre (CSCS) resource *Piz Daint*, the fastest computer in Europe and the sixth-fastest in the world [40]. A node contains an 8-core Intel Xeon E5-2670 CPU, 32 GB of RAM, and an NVIDIA Tesla K20X with 6 GB of device memory. Detailed specification of this GPU is available in Section 2.2. Except where noted, performance measurements are given as the mean of three samples to minimise the influence of variance on interpretation.

When NVIDIA CUDA or Cray Performance Analysis Tool profilers were enabled in either sample or trace mode, they produced slightly different results and so different (40% shorter) run times due to imposition of the CFL condition discussed in Section 3.3. The NVIDIA CUDA profiler in particular does not require compile-time instrumentation. Time constraints prohibit detailed investigation of this effect, but a possible reason would be a change to the shell environment which causes different dynamic library versions to be loaded at runtime.

As any user's interest is in the time taken to simulate a fixed period of time, rather than a number of time steps of unknown size, as the effect was similar across evaluation configurations, and to permit profile-based investigation of results, the NVIDIA CUDA profiler was enabled[1] at runtime throughout the evaluation. Were the profiler's presence to have such a significant effect on the result, it would not otherwise be possible to use

[1] `COMPUTE_PROFILE=1`

this instrumentation to investigate the cause of a result observed without instrumentation.

4.1.1 Single node

Exploratory single-node performance evaluation of `pw_advection_acc` was carried out using the configurations detailed in Table 4.1. Two q fields were used in each configuration. Performance variables were measured over five simulated minutes of the dry boundary layer, except for 'Ratio of work to data' where three hundred timesteps of the same are used instead. Where a performance was evaluated at a single problem size, as in 'Ratio of work to data' and 'Features and structure', this was $256 \times 256 \times 64$. Evaluation within a single node permits comparison of a single CPU and GPU socket of the same generation, as suggested by Govett et al. [12]. MPI communication typically carries much lower overheads within a node than between nodes, allowing a relatively direct comparison of the computation.

Horizontal scale

For scientific and forecasting purposes, it is more commonly useful to vary the area simulated than the height. This permits e.g. larger weather systems to be analysed, or more population centres to receive coverage. At each of five scale points, three variables were measured: wall clock run time, GPU compute time, and system energy consumption. Although less immediately relevant to the user, the second provides insight into load balance between CPU and GPU. Each successive scale multiplies size in x and y by $\sqrt{2}$, and so the total number of grid points by 2. As this grid volume is linearly related work done and run time, rather than quadratically as x and y are, this is presented as the dependent variable.

Figure 4.1 shows both GPU-only and Mixed CPU/GPU configurations performing more slowly than CPU-only over the $16\times$ range examined. As problem size increases, the relative difference from CPU-only to GPU-only falls from 18% to under 5%, and similarly to Mixed CPU/GPU. However, these larger problem sizes are not representative of expectations for a single node: they would more likely tbe distributed across multiple nodes to reduce absolute run time. Unless other components within `dynamics` contain

		Advected fields	
Configuration	**Advection component(s)**	**CPU**	**GPU**
CPU-only	`pw_advection`	u, v, w, θ, q	—
GPU-only	`pw_advection_acc`	—	u, v, w, θ, q
Mixed CPU/GPU	`pw_advection`, `..._acc`	θ, q	u, v, w

Table 4.1: `pw_advection` and `pw_advection_acc` evaluation configurations.

substantial communication and so perform differently with an increased number of processes, an inferior performance on a single node suggests there may be no configuration in which the OpenACC implementation is faster.

A benefit of the management of common data described in Section 3.1.3 can be seen in Figure 4.2. Although the GPU-only configuration performs around twice as much computation as the Mixed CPU/GPU, it incurs only around a 75% increase in GPU time.

Each node's Intel Xeon E5-2670 has a Thermal Design Power (TDP) of 115 W [17], while the NVIDIA K20X has a TDP of 235 W [28]. Despite the larger headline figure, the GPU offers just under $4\times$ the CPU's double-precision FLOPS/W [16, 29]. Although the CPU may intelligently manage energy consumption, the GPU energy consumption is near-constant [15]. GPU utilisation must approach peak throughput or sacrifice energy efficiency. Figure 4.3 shows up to 43% higher energy consumption by GPU configurations, as both system power and simulation run time increase: energy is used more quickly, and for longer.

Ratio of work to data

A contributing factor to the relatively poor performance seen above might be that PW advection simply offers too low a ratio of computation to data which must be transferred to the GPU. The importance of data movement's detrimental effect on GPU performance has been emphasised throughout this dissertation. The ULTIMATE-QUICKEST TVD scheme offers a higher ratio, but the implementation difficulties described in Section 3.2.2 preclude a direct examination of the OpenACC implementation's performance on the GPU. An investigation into the effect of increased computation per column in the available PW advection components may hint at the performance of a completed `tvd_advection_acc` component relative to the CPU implementation.

To enable this investigation, both PW advection components were extended to require an additional runtime configuration parameter: a 'work factor'. A new inner loop within a column repeats each point this many times before advancing to the next. These modified procedures will produce unphysical output, violating the CFL condition. As stated in Section 2.1.2, `cfltest` would detec this violation and terminate the simulation. The component was disabled to avoid this.

Varying this work factor at run- rather than compile-time not only made it easier to evaluate performance at different values, but also discouraged compiler optimisation. Additionally, the OpenACC version of this new loop was prefixed with a `loop` directive using the `seq` clause, to avoid unduly favouring the GPU implementation by introducing additional parallelism. A comparison of the 'loopmark' source output by the compiler with annotations to indicate loop optimisation, vectorisation and accelerator use suggests that the only variable between the original and work factor executable is this deliberate repetition.

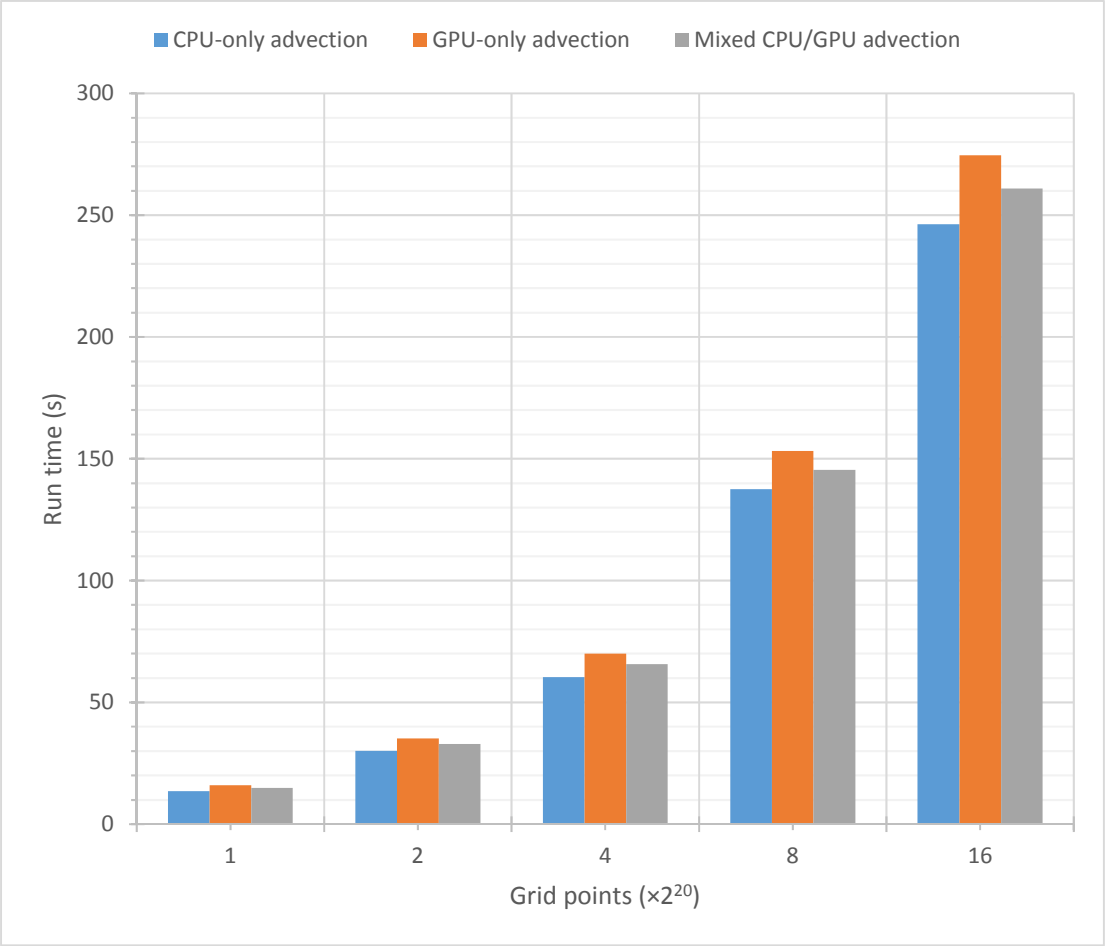


Figure 4.1: Single-node run time by horizontal (x, y) scale.

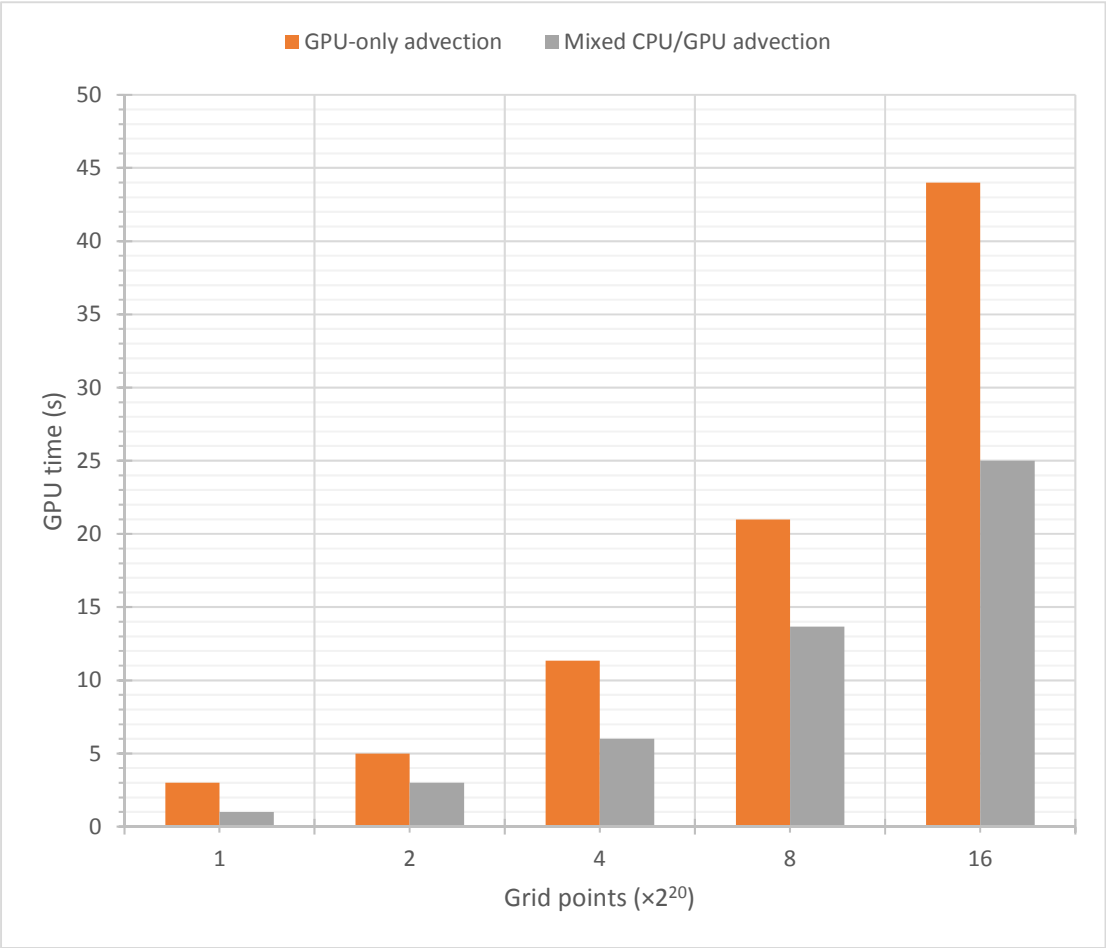


Figure 4.2: Single-node GPU time by horizontal (x, y) scale.

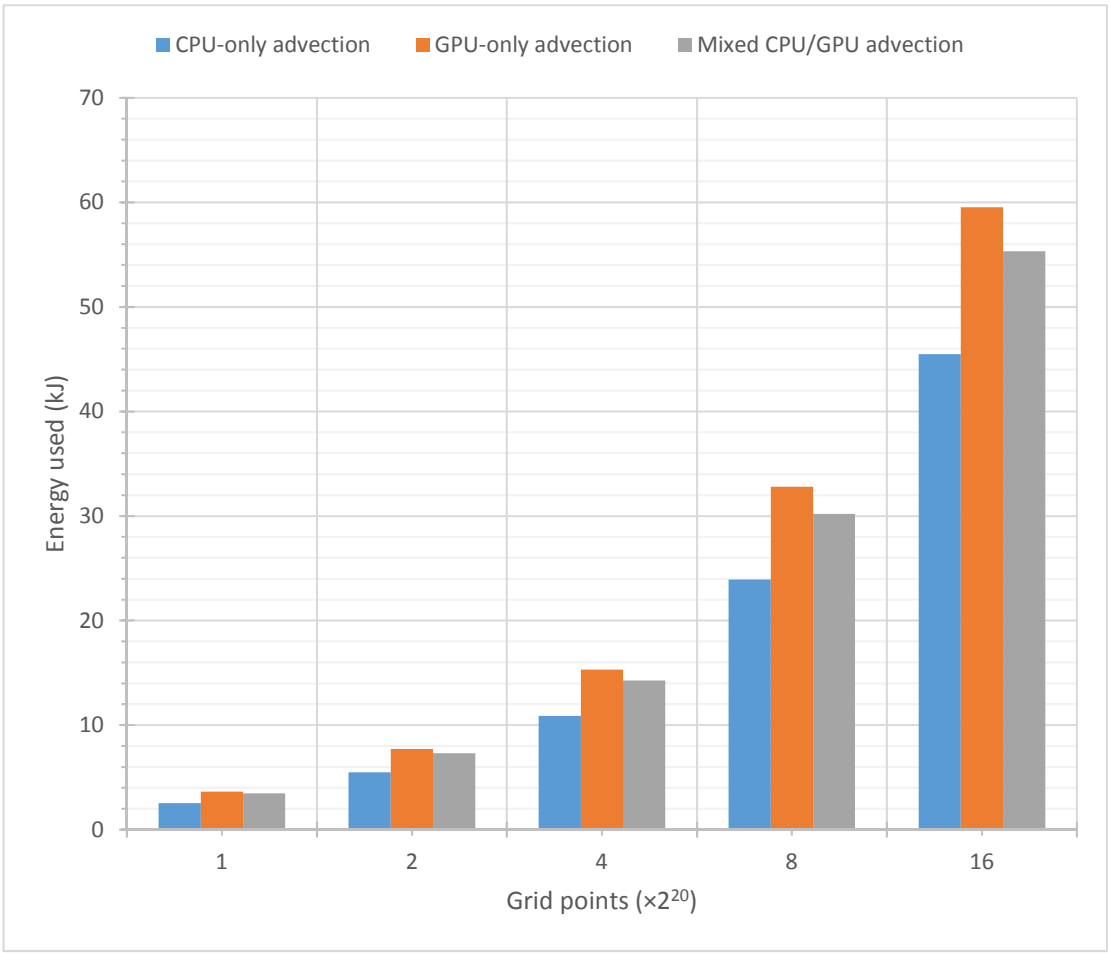


Figure 4.3: Single-node energy usage by horizontal (x, y) scale.

	Run time		
Advection component	**Total (s)**	**Advection**	**Advection (s)**
`pw_advection`	61	7.5%	5
`tvd_advection`	234	48%	112

Table 4.2: Comparison of `pw_advection` and `tvd_advection` run time.

A direct comparison of each advection scheme's ratio of computation to data, measured perhaps in units of FP operations per unit of data, would be difficult. For example, each operation's performance impact in a pipeline may depend on their context in the instruction stream. A comparison of this sort is not necessary, as only the difference between (ratio of) their ratios is relevant. As the two advection schemes use very similar data, a first approximation of this difference may be found by comparing their run time.

Figure 4.4 suggests the GPU becomes competitive at or around a ratio of work to data $128\times$ that seen in PW advection. Table 4.2 indicates ULTIMATE-QUICKEST's ratio of work to data is $25\times$ that of PW. Although $5\times$ less than appears necessary, this comparison should not be made with too much precision. An actual implementation may vary in more factors than just the ratio of work to data investigated here, some of which are explored in 'Features and structure'.

No increase in GPU run time is shown by Figure 4.5 until a work factor of 256, at which scale the GPU-only configuration is 12% faster than the CPU. This may indicate a combination of a reduction in the proportion of time spent on data transfer, and an increase in the time spent synchronising with the host. The earlier statement that GPU energy consumption is almost constant is strongly supported by Figure 4.6: very little variation is seen over the entire $256\times$ range.

Features and structure

`pw_advection_acc`'s development included 'infrastructure' features which did not centre on the addition of OpenACC directives, but which were necessary to support the GPU implementation described in Section 3.1. These may themselves reduce performance, even before any GPU kernel is launched. Despite the benefits of `column`-type component execution described in Section 2.1.1, `entire`-type is necessary to expose sufficient parallelism. Additional work must be done by `stepfields` to collect the device source terms if necessary. OpenACC can be disabled at compile-time to allow a direct comparison between the new implementation and the original, without any of this project's changes. Table 4.3 shows the results of this comparison.

A numerical analysis by Griffith et al. [14] leads Section 2.3.1 to suggest an investigation of the benefit to performance of switching to single-precision FP. As seen in Section 2.2, GPUs provide many more functional units for single- than double-precision FP: the NVIDIA K20X's $3\times$ is an improvement over previous generations [27]. How-

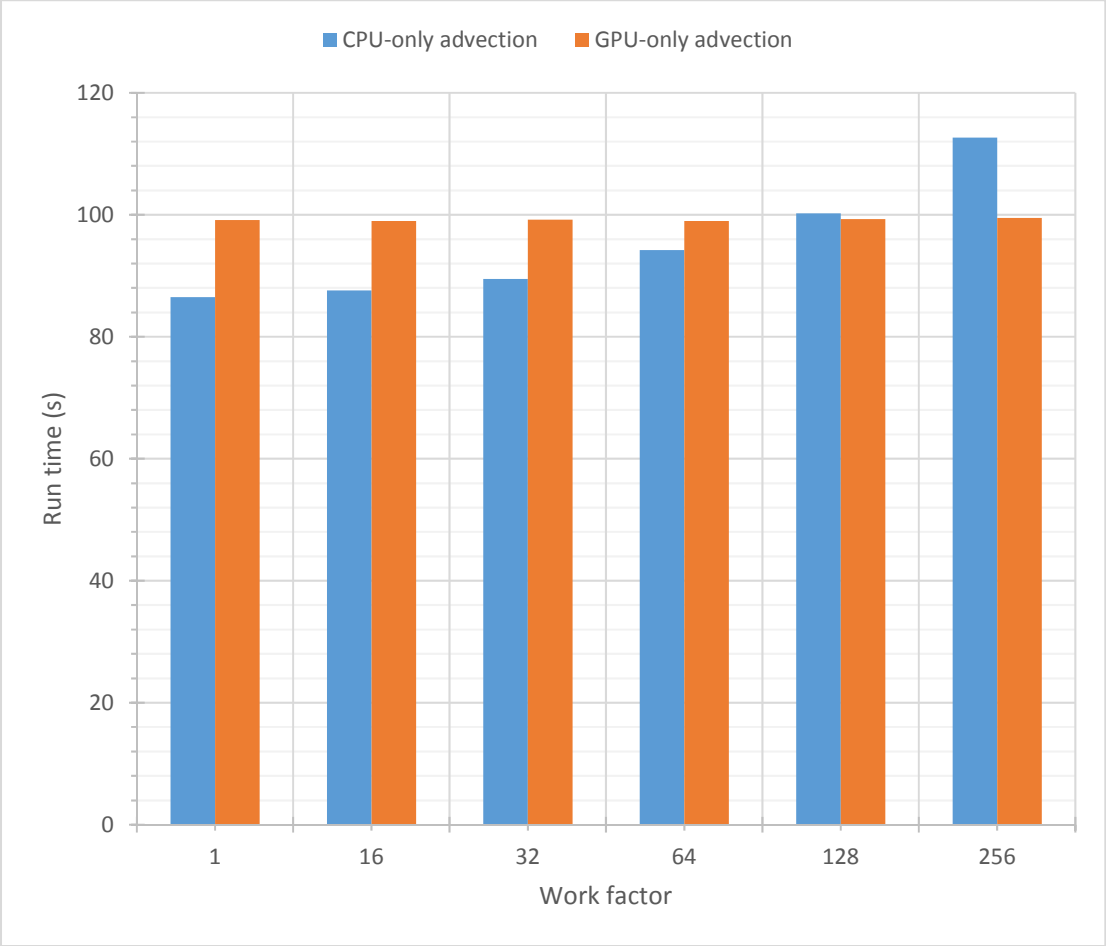


Figure 4.4: Single-node run time by ‘work factor’, or artificial additional computation.

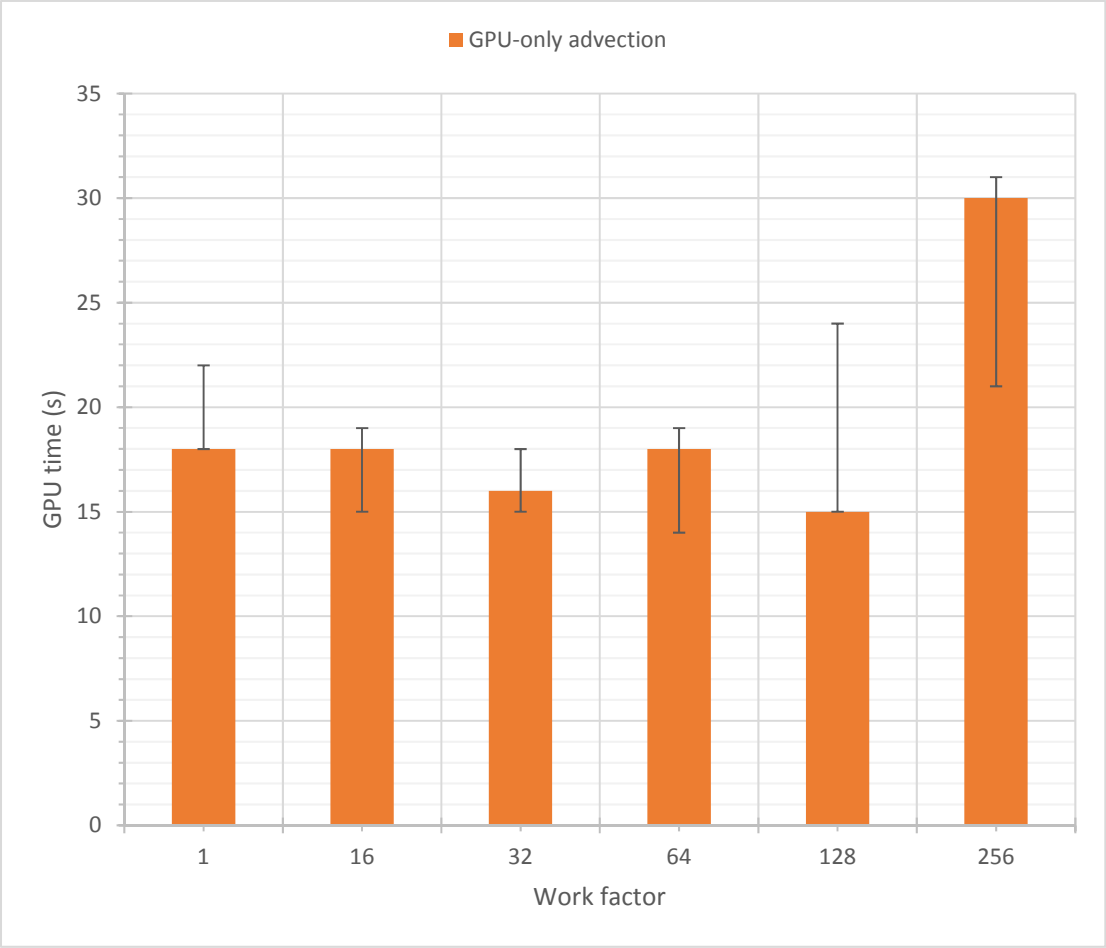


Figure 4.5: Single-node GPU time by ‘work factor’, or artificial additional computation. Median and range of three measurements presented due to high variance.

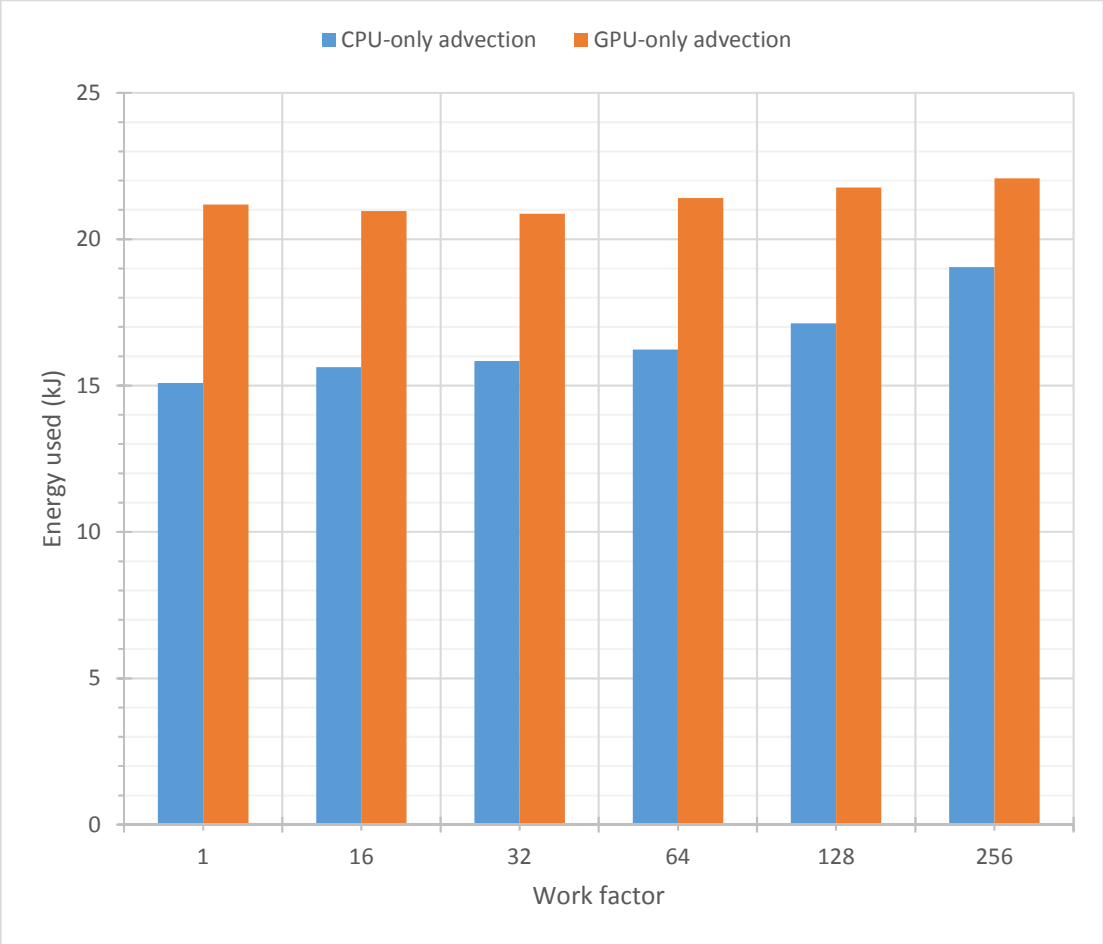


Figure 4.6: Single-node run time by ‘work factor’, or artificial additional computation.

Description	Run time (s)	Change
Original	60.4	—
GPU asynchronous	70.0	+16%
GPU code on CPU	74.1	+23%
GPU synchronous	79.2	+31%

Table 4.3: Absolute and relative performance of `pw_advection_acc` revisions.

ever, the largest benefit to `pw_advection_acc` may be in the 50% reduction in data transfer, as array elements shrink from 8 bytes to 4. MONC's default FP precision is specified by a compile-time parameter, but some mechanical changes must also be made to the FFTW library's use[2] by the FFT solver. Table 4.4 shows the lower precision narrowing the CPU's lead over the GPU by 4 percentage points.

		Change from ...	
Configuration	**Run time (s)**	CPU-only	Double-precision
CPU-only	60.4	—	—
GPU-only	70.1	+16%	—
Mixed CPU/GPU	65.8	+9%	—
CPU-only	57.9	—	−4%
GPU-only	63.6	+10%	−9%
Mixed CPU/GPU	61.3	+6%	−7%

Table 4.4: Absolute and relative performance of `pw_advection_acc` configurations with single- and double-precision FP.

Attempted optimisation

A number of optimisations motivated by technical considerations described in Section 2.2 were attempted on `pw_advection_acc`, but had no measurable effect on performance, likely because they do not reduce data transfer and hence the critical path. This section describes these unsuccessful attempts.

Although Section 2.3.1 supports this implementation's decision to arrange fine-grained parallelism in z, the balance within the parallelism hierarchy was explored by reducing the argument to the `loop` directive's `collapse(`*n*`)` clause and explicitly requesting each loop's iteration space be divided amongst gangs, workers, or vector lanes by using the corresponding `loop` directive clause. The thread decomposition was varied using the `num_workers` and `gangs` clauses, and an attempt to manually calculate a number of gangs based on the number of SMs available and each loop's workload was made.

[2] `fftw` to `fftwf`, `C_DOUBLE_COMPLEX` to `C_FLOAT_COMPLEX`, and `-lfftw3` to `-lfftw3f`.

OpenACC specifies a `cache` directive to fetch array elements or subarrays into the highest level of cache for the body of the loop. There is no provision for scalar values, and the only permitted subarray indexes are constants, the array index, or the array index ± 1. The Cray implementation imposes a further restriction that the directive may appear only within an innermost loop, regardless of any `collapse` clause. As MONC's prognostic fields are necessarily `allocatable` arrays, no benefit from the these restrictions eliminate any expected utility: we can cache only the innermost loop's data, and then only immediately prior to using it. Attempting to use the directive in this way resulted in a segmentation fault.

4.1.2 Multiple nodes

Section 4.1.1 explains that the performance trend on multiple nodes is not expected to differ qualitatively from the performance on a single node, as there is very little interprocess communication within `dynamics` i.e. the factors which influence this project's effect on performance are contained within a node. This hypothesis should still be tested, however.

A segmentation fault was encountered at certain combinations of process count and grid size which meant the FFT solver could not be used to examine weak scalability and was replaced with the iterative solver. Although the timestep was found very consistent with the FFT solver, much more variation was observed both within and between configurations. To avoid this affecting run time, the number of timesteps was fixed at three hundred, rather than the period of simulated time. The wall clock time taken to simulate each timestep was also less consistent than when the FFT solver was used, so a minimum (not an average) of three measurements was taken.

Time taken to initialise the simulation occasionally rose from tens of milliseconds to seconds or even minutes, with no obvious pattern. This would exhause the otherwise adequate five minute time limit for a run, sometimes without any output. All configurations (both CPU and GPU) were affected. Job submission was scripted, and identical[3] configurations and executables were used by affected and unaffected submissions. Available time did not permit an investigation of the cause, so Figure 4.7 presents timestep-only time, which excludes the time taken for initialisation. The effect of random variation is more apparent, but there is no evidence seen to contradict the hypothesis above.

4.2 Comparison with similar models

This project adheres to the common 'accelerator' or 'offload' programming and execution model favoured by the NIM and CAM models discussed in Section 2.3. Although

[3]Confirmed by `diff(1)`.

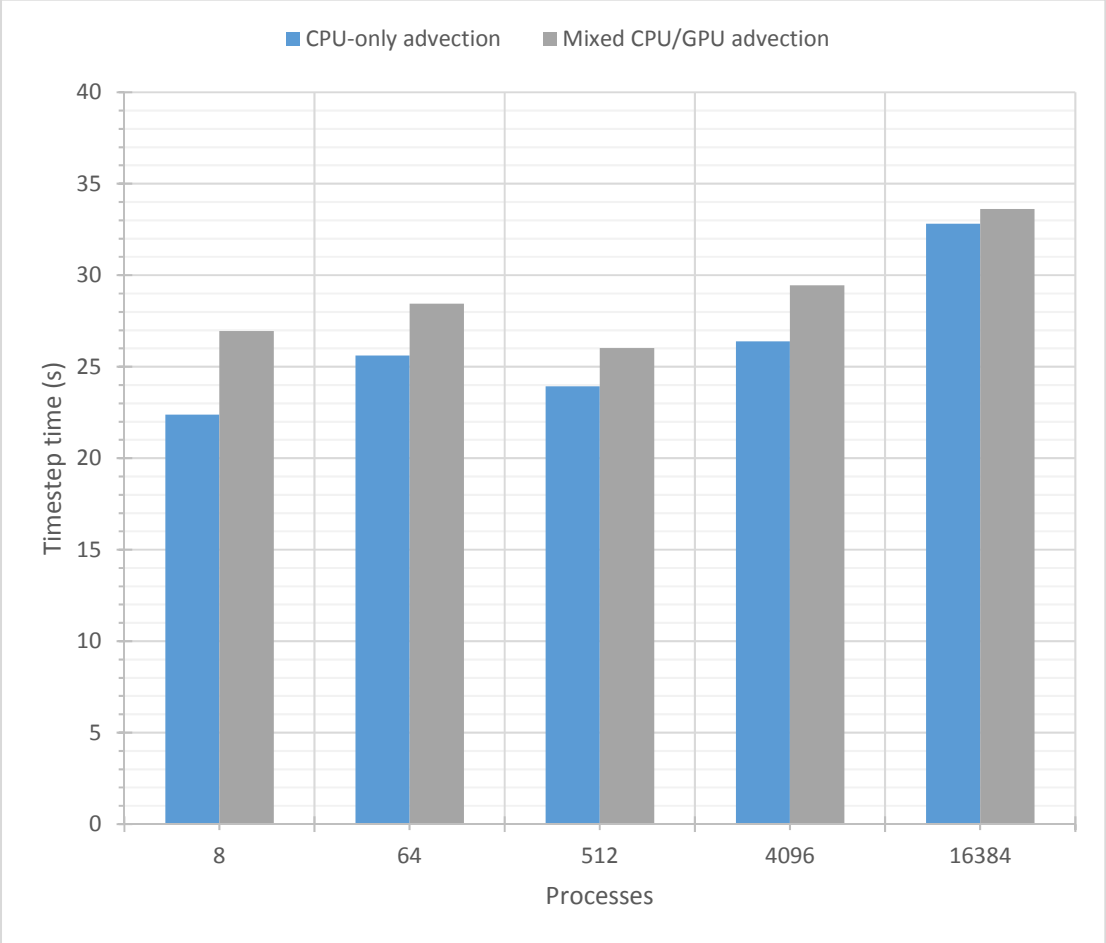


Figure 4.7: Timestep run time by number of processes, one process per core. For reasons discussed in the text, the figures presented are the minimum of between one and three measurements.

these projects eventually found modest performance benefits, they also encountered implementation difficulties which were overcome over a period of several years. The first NIM paper indicates a slowdown from CPU to GPU unless data transfer time is ignored [11], although this was improved by later publications. The work described in this dissertation is also limited by data transfer, exacerbated by implementation difficulties blocking acceleration of the advection scheme which was anticipated to provide a more favourable ratio of computation to data transfer.

This is the well-known problem of this approach, but while the 'GPU first' approach of GALES avoids data transfer between device and host except for export and/or visualisation, it is extremely costly except when integrated into the development of a new application, and may limit accessibility by members of the scientific community without easy access to GPU resources. The latter would be a disadvantage for MONC, which aims to remain extensible and usable by the wider community [4].

Govett et al. [10] and Norman et al. [25] both mention that it may be difficult to maintain a single source, as the performance impact of infrastructure and code restructuring to support the GPU may have an unacceptable effect on CPU performance. Norman et al. [25] in particular recognise that much scientific development continues to take place on the CPU. MONC's component system makes developing and evaluating two independent implementations relatively easy, with no need for e.g. pre-processor directives interspersed throughout. Enforced independence of components does not allow the redundancy between the two implementations to be eliminated, which is a concern for maintainability, but this could in principle be achieved by extracting the `column`-type functionality to a module, annotating it with the OpenACC `routine` directive as seen in Section 3.2.2, and using this from both CPU and GPU.

OpenACC seems to encourage a distinctive style, and the code structure, directives and their clauses seen in [25] are noticeably similar to those independently developed during this project.

4.3 OpenACC

OpenACC was chosen following a preparatory evaluation of technologies available to program GPUs from Fortran, and although some literature[4] reviewed in Section 2.3 may have given pause, the evaluation's outcome would have been no different. OpenMP 4 was not yet available in a released product. OpenCL would effectively introduce another language to which kernels must first be ported and then maintained indefinitely. CUDA Fortran appears competitive on technical merit but a proprietary solution is unsuitable for policy reasons. For this reason, it is important that OpenACC's promise and present limitations are understood.

A pair of high-level problems presented friction to OpenACC's use: lacking or absent

[4] Published after this evaluation.

support for derived types, and compiler and runtime quality. These were exacerbated by a secondary issue: the specification is neither sufficiently formal nor complete.

The concept of user-defined composite types in high-level languages has existed for fifty years, and their implementation in derived types have been a part of standard Fortran for more than twenty. Although the semantics of references held within derived types on the CPU and GPU remain an open question, the situation described in Section 3.1.1 is that derived types are so inconsistently and unsafely handled by OpenACC regions and directives, even in the absence of such references and in the presence of vendor-specific extensions which in any case propose to augment these with 'deep copy' semantics, that they must be systematically eliminated from code to be accelerated. As a result of their long history, modern software such as MONC make pervasive use of these types. Their elimination is then at best clumsy, and at worst impractical. Norman et al. [25] declare this "not worth the effort" in a similar numerical weather simulation.

As recognised by Norman et al. [25] and Govett et al. [10], the available OpenACC implementations are yet to mature. A number of compiler bugs were encountered in the PGI compiler during the preparatory technology evaluation, and although the previous authors found the Cray implementation more mature, many defects were also uncovered in this compiler and runtime during this project. A particular challenge is the implementation of user-visible language features in terms of an unknown set of compiler or runtime features, with the result that seemingly unrelated changes to the code can introduce or eliminate an error. An example of this behaviour is the need to disable with a compiler directive the inlining of a particular function as described in Section 3.1.1. At times the effect is to make a rational approach to directive choice and code design very difficult at times, instead encouraging trial-and-error.

When these defects are encountered, the diagnostic messages produced are often difficult to understand, seemingly (and sometimes explicitly) written for vendor developers rather than end users. Although a truly formal specification may neither practical nor necessary, Norman et al. [25] find that the language used may permit multiple implementations as seen in Section 3.1.2. These factors do not contribute to a code accessible to experts in e.g. atmospheric science rather than HPC.

An unfortunate but necessary outcome of efficient data management and 'unpacking' of derived types is code which closely resembles what might be expected of a CUDA Fortran implementation: a number of arrays explicitly duplicated and marshalled between host and device, manipulated by a kernel of tightly-nested loops. Much could be done to improve the treatment of derived types e.g. by allowing individual components to be identified in data clauses and transferred alone, or entire types to be transferred correctly in the absence of problematic references. There do not appear to be such easy solutions for data management. The compiler already avoids unnecessary transfer in or out of the device in simple cases e.g. with reference to procedure argument intents.

OpenACC promises a considerable improvement in development experience over alternatives such as CUDA Fortran if these challenges are addressed. This seems likely: although this project was not long enough to experience any improvement, both Nor-

man et al. [25] and Govett et al. [10] highlight active development of usability and performance improvements in the available implementations. Beyond these improvements under the current specification, a further two 'wish list' features would have been useful to this project and perhaps generally.

This project found asynchronous execution and data transfer important to performance, with Govett et al. [12] emphasising the importance of CPU-GPU load balance to performance and overall cost. OpenACC could in principle provide a clause applicable to the `loop` directive specify 'symmetric' execution, the iterations split between host and devices. Although this may require some hard engineering problems be solved, they do not appear intractable given that there exist codes which solve their particular case.

The second proposed feature solves a problem discussed in Section 3.1.2: if a dataset split between host and device is to be recombined, then the host must allocate a buffer to receive the device data before the two are combined. This may represent a large reduction in useful memory capacity. OpenACC could add an argument to the `update` directive and `copy` and `copyout` clauses to specify an operation with which to recombine the two in-place. Additional hardware support could make this more time-efficient, but Kepler GPUs have the necessary access to host memory for this to be implemented today [27].

Chapter 5

Conclusion

Exploration of this project's background in weather simulation, techniques for the numerical simulation of atmospheric flow, and the results and practices established by similarly GPU-accelerated weather models was used to motivate the design of two accelerated advection components. The solutions to specific implementation challenges encountered in their development were set out, including a more generally-applicable set of suggestions for OpenACC's use and the infrastructure necessary to support asynchronous execution and data transfer.

An evaluation of scalability within a single node and across multiple nodes established no immediate performance benefit for the complete PW advection component, but a comparison-based method was given by which the benefit of GPU acceleration might be estimated for other components. The performance impact of GPU infrastructure, asynchronous execution and FP precision were investigated. A detailed survey was made of OpenACC's current attractions and deficiencies, and specific areas for improvement identified.

5.1 Further work

OpenACC may not yet be practical for mainstream use, but holds promise for the future even if the specification does not radically change but implementations improve considerably. Section 4.3 highlights features which may be improved to increase usability of the existing specification, and suggests two realistic extensions to the specification which would enable improved application performance.

Section 4.2 suggests a means by which the significant duplication between CPU and GPU components might be eliminated. This would first require MONC's component system be adapted to allow components to share a dependency on functionality not provided directly by the core, but either by a component as they exist today, or by a new, similar aggregation of functionality.

As the number of accelerated components increases, so will the number of published fields and integer identifiers for asynchronous operations. A collision or error in these will not be detected at compile time, and may produce sporadic or obscure errors or incorrect results at runtime. Central management by a broker or at least a generally-accessible repository would make their use much less fragile.

Bibliography

[1] Akio Arakawa. Computational design for long-term numerical integration of the equations of fluid motion: Two dimensional incompressible flow. Part I. *Journal of Computational Physics*, 1(1):119–143, August 1966. URL http://www.sciencedirect.com/science/article/pii/0021999166900155.

[2] Akio Arakawa and Vivian R. Lamb. Computational design of the basic dynamical processes of the ucla general circulation model. *Methods in Computational Physics: Advances in Research and Applications*, 17:173–265, 1977. URL http://www.sciencedirect.com/science/article/pii/B9780124608177500094.

[3] James Beyer, David Oehmke, and Jeff Sandoval. Transferring user-defined types in OpenACC. In *CUG2014 Final Proceedings*, CUG2014, Lugano, May 2014. Cray User Group, Inc. URL https://cug.org/proceedings/cug2014_proceedings/includes/files/pap122.pdf.

[4] Nick Brown, Michèle Weiland, Adrian Hill, Ben Shipway, and Chris Maynard. A highly scalable Met Office NERC Cloud model. In *Third International Conference on Exascale Applications and Software, EASC 2015*, Edinburgh, May 2015. Association for Computing Machinery. URL http://www.easc2015.ed.ac.uk/proceedings.

[5] Kirk Bryan. A scheme for numerical integration of the equations of motion on an irregular grid free of nonlinear instability. *Monthly Weather Review*, 94(1):11–25, January 1966. URL http://journals.ametsoc.org/doi/abs/10.1175/1520-0493(1966)094%3C0039%3AASFNIO%3E2.3.CO%3B2.

[6] I. Carpenter, R. K. Archibald, K. J. Evans, J. Larkin, P. Micikevicius, M. Norman, J. Rosinski, J. Schwarzmeier, and M. A. Taylor. Progress towards accelerating HOMME on hybrid multi-core systems. *International Journal of High Performance Computing Applications*, 27(3):335–347, August 2013. URL http://hpc.sagepub.com/content/27/3/335.abstract.

[7] Cray. intro_OpenACC - Summarize OpenACC support for accelerators. Retrieved August 14, 2015 from `man 7 intro_OpenACC`, 2014.

[8] Dale R. Durran. *Numerical Methods for Fluid Dynamics*, volume 32 of *Texts in Applied Mathematics*. Springer-Verlag, 2010. ISBN 978-1-4419-6412-0.

[9] Alan Gadian. Use of high performance computing (HPC) in weather modelling.

In *The 5th HPCx Annual Seminar*, Daresbury, 2007. HPCx. URL http://www.hpcx.ac.uk/about/events/annual2007/Alan%20Gadian.pdf.

[10] Mark Govett, Jacques Middlecoff, and Tom Henderson. Directive-based parallelization of the NIM weather model for GPUs. In *Proceedings of the First Workshop on Accelerator Programming using Directives*, WACCPD '14, pages 55–61, New Orleans, 2014. IEEE Computer Society, Association for Computing Machinery. URL http://dl.acm.org/citation.cfm?id=2691165.

[11] Mark W. Govett, Jacques Middlecoff, and Tom Henderson. Running the NIM next-generation weather model on GPUs. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 792–796, Washington, D.C., 2010. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=1845128.

[12] Mark W. Govett, Tom Henderson, Jim Rosinski, Jacques Middlecoff, and Paul Madden. Parallelization and performance of the NIM for CPU, GPU and MIC. In *95th Annual Meeting of the American Meteorological Society*, Phoenix, 2015. American Meteorological Society. URL https://ams.confex.com/ams/95Annual/webprogram/Paper262515.html.

[13] M. E. B. Gray and J. Petch. Version 2.3 of the Met Office Large Eddy Model: Part I. User documentation. Technical Report 1, Met Office, 2001. URL http://www.met.rdg.ac.uk/~lem/large_models/lem/documentation/DOCUMENTATION/lemdoc.ps.

[14] Eric J. Griffith, Frits H. Post, Thijs Heus, and Harm J. J. Jonker. Interactive simulation and visualisation of atmospheric large-eddy simulations. Technical Report 2, TuDelft Data Visualization Group, 2009. URL http://graphics.tudelft.nl/images/atmospheric/vis2009-02.pdf.

[15] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *International Symposium on Parallel Distributed Processing*, Rome, May 2009. Institute of Electrical and Electronics Engineers. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5160980.

[16] Intel. Intel xeon processor e5-2600 series. Retrieved August 19, 2014 from Intel: http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf, 2012.

[17] Intel. Intel xeon processor e5-2670 (20m cache, 2.60 ghz, 8.00 gt/s intel qpi) specifications. Retrieved August 19, 2014 from Intel: http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI, 2012.

[18] B. P. Leonard. A stable and accurate convective modelling procedure based on quadratic upstream interpolation. *Computer Methods in Applied Mechanics and*

Engineering, 19(1):59–98, June 1979. URL http://www.sciencedirect.com/science/article/pii/0045782579900343.

[19] B. P. Leonard. Universal limiter for transient interpolation modeling of the advective transport equations: the ULTIMATE conservative difference scheme. Technical Report NASA-TM-100916, National Aeronautics and Space Administration, 1988. URL http://www.researchgate.net/profile/Brian_Leonard4/publication/24325214_Universal_limiter_for_transient_interpolation_modeling_of_the_advective_transport_equations_The_ULTIMATE_conservative_difference_scheme/links/00b4953454d4ebe1c1000000.pdf.

[20] B. P. Leonard. The ULTIMATE conservative difference scheme applied to unsteady one-dimensional advection. *Computer Methods in Applied Mechanics and Engineering*, 88(1):17–74, June 1991. URL http://www.hadian.ir/teaching/CompHydr/3.pdf.

[21] Douglas K. Lilly. On the computational stability of numerical simulations of time-dependent non-linear geophysical fluid dynamics problems. *Monthly Weather Review*, 93(1):11–25, January 1965. URL http://journals.ametsoc.org/doi/abs/10.1175/1520-0493(1965)093%3C0011%3AOTCSON%3E2.3.CO%3B2.

[22] Met Office. LEM homepage. Retrieved March 15, 2015, from the Met Office: http://appconv.metoffice.com/LEM/, 2006.

[23] Met Office. Modelling systems used in the Met Office. Retrieved March 15, 2015, from the Met Office: http://www.metoffice.gov.uk/research/modelling-systems, 2014.

[24] Matthew Norman, Valentine Anantharaj, Richard Archibald, Ilene Carpenter, Katherine Evans, Jeffrey Larkin, Paulius Micikevicius, and Aaron Vose. Porting CAM-SE to use Titan's GPUs. In *Heterogeneous Multi-core 4 Workshop*, Boulder, 2014. National Center for Atmospheric Research. URL https://www2.cisl.ucar.edu/sites/default/files/norman_6.pdf.

[25] Matthew Norman, Jeffrey Larkin, Aaron Vose, and Katherine Evans. A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel. *Journal of Computational Science*, 9(0):1–6, April 2015. URL http://www.sciencedirect.com/science/article/pii/S1877750315000605.

[26] NVIDIA. CUDA toolkit documentation v7.0. Retrieved March 16, 2015 from NVIDIA: http://docs.nvidia.com/cuda/index.html, 2015.

[27] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110. Retrieved March 16, 2015 from NVIDIA: http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[28] NVIDIA. Tesla K20X GPU accelerator board specification. Retrieved July 12, 2015 from NVIDIA: http:

//www.nvidia.co.uk/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf, 2012.

[29] NVIDIA. NVIDIA Tesla GPU accelerators. Retrieved July 12, 2015 from NVIDIA: http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf, 2013.

[30] OpenACC.org. The OpenACC application programming interface. Retrieved August 14, 2015, from OpenACC.org: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, June 2013.

[31] J. Petch. Test case 1: Dry, neutral boundary layer with backscatter. Retrieved March 18, 2015, from the University of Reading: http://www.met.rdg.ac.uk/~lem/large_models/lem/test_cases/test_case1/index.html, 2001.

[32] Norman A. Phillips. The general circulation of the atmosphere: a numerical experiment. *Quarterly Journal of the Royal Meteorological Society*, 82(352): 123–164, April 1956. URL http://www.phy.pku.edu.cn/climate/class/cm2010/Phillips_QJRMS_1956.pdf.

[33] Norman A. Phillips. An example of non-linear computational instability. In Bert Bolin, editor, *The Atmosphere and the Sea in Motion*. Rockefeller Institute Press in association with Oxford University Press, New York, 1959. URL https://www.ualberta.ca/~eec/Phillips_NLInstablity.pdf.

[34] Steve A. Piacsek and Gareth P. Williams. Conservation properties of convection difference schemes. *Journal of Computational Physics*, 6(3):392–405, December 1970. URL http://www.sciencedirect.com/science/article/pii/0021999170900380.

[35] Ugo Piomelli. Large-eddy simulation: achievements and challenges. *Progress in Aerospace Sciences*, 35(4):335–362, May 1999. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.5036&rep=rep1&type=pdf.

[36] Robert D. Richtmyer. A survey of difference methods for non-steady fluid dynamics. Technical Report 63-2, National Center for Atmospheric Research, August 1962. URL http://nldr.library.ucar.edu/repository/assets/technotes/TECH-NOTE-000-000-000-050.pdf.

[37] Richard B. Rood. Numerical advection algorithms and their role in atmospheric transport and chemistry models. *Review of Geophysics*, 25(1):71–100, February 1987. URL http://climateknowledge.org/figures/Rood_Library/1987_Rood_Rev_Geophys_Advection.PDF.

[38] Jerôme Schalkwijk, Eric J. Griffith, Frits H. Post, and Harm J. J. Jonker. High-performance simulations of turbulent clouds on a desktop PC: Exploiting the GPU. *Bulletin of the American Meteorological Society*, 93(3):307–314, March 2012. URL http://journals.ametsoc.org/doi/abs/10.1175/BAMS-D-11-00059.1.

[39] Marc Stringer. Machine efficiency. Retrieved July 26, 2015, from the University of Reading: http://www.met.rdg.ac.uk/~lem/large_models/lem/efficiency/, 2004.

[40] TOP500.org. June 2015 | top supercomputer sites. Retrieved August 16, 2015, from TOP500.org: http://www.top500.org/list/2015/06/, 2015.

[41] Vasily Volkov. Better performance at lower occupancy. In *GPU Technology Conference*, San Jose, September 2010. NVIDIA. URL http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[42] Cliff Woolley. GPU optimization fundamentals. Retrieved March 18, 2015 from Oak Ridge National Laboratory: https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf, 2013.