ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High Performance Computing for Weather and Climate

Report for the Final Project

Parallelisation of stencil computations on GPUs using OpenACC

Authors: Max Frei, Ronan McCormack, Gabriel Vollenweider

Supervisor: Oliver Fuhrer

16.09.2022

1 Introduction

Weather and climate models are based on physical principles like conservation of momentum, conservation of mass, and conservation of energy. These imply a set of governing equations for the variables: wind velocity $\boldsymbol{u}$, pressure p, temperature T, density ρ, and specific humidity q. The governing equations are non-linear partial differential equations (PDEs) which must be solved numerically. Typically, this is done by dividing the atmosphere into a collection of grid cells. Each grid cell is represented by a single grid point. More grid points imply a higher resolution of the atmosphere and more accurate results. Current weather and climate models typically involve about 10^8 to 10^9 grid points, each storing the variables $\boldsymbol{u}$, p, T, ρ, q. Given an initial state of the atmosphere on such a grid (about 10^9 to 10^{10} degrees of freedom), the variables are then numerically integrated forward in time. This needs to be done faster than real time, and current models are about 100 times faster than real time.

To achieve such a speed, computer hardware needs to be aggregated to maximise the computations for a given input of electrical power. Such high performance computing systems are typically designed in a modular way, where different components may work on different parts of a given numerical problem. If a problem can be divided into separate chunks, where each chunk can be solved independently, then the solution can be computed in parallel without requiring much communication between the chunks. One example of such a problem are computations that only rely on a compact set of neighbouring grid points, with the same pattern everywhere on the numerical grid. These so-called stencil computations typically arise from the discretisation of PDEs and they are thus a common theme in weather and climate models.

Global weather and climate models are starting to reach resolutions of about 10 km in the horizontal. However, this resolution is not enough to resolve convection, which is a major source of uncertainty for the models. Regional models with an increased horizontal resolution of about 2 km predict cloud formation and precipitation much more accurately. It is thus desirable to increase the resolution of global weather and climate models. However, decreasing the horizontal grid spacing by a factor of 5 increases computations by at least a factor of 5^3 (two horizontal dimensions, and one time dimension for numerical stability). To still have the computations run faster than real time, ideally with minimal losses in speed, it is then necessary to further optimise the required time per computation.

There are several options to speed up a numerical computation. Concerning hardware, better performance can be achieved by designing an architecture geared towards optimal speed for a given problem. This may involve optimised caches, increased memory bandwidth or processing in or close to memory. Without such a change in design, it has become difficult to speed up hardware, as transistors are reaching their minimum physical size, indicating a slow down of Moore's law (Shalf, 2019). Another possibility to gain speed is specific code optimisation for a given hardware and problem. This might include the minimisation of data transfers (by using caches optimally, or processing where the memory is stored) or increasing the distribution of tasks to different parallel components. In our report, we employ both ideas by using GPUs (graphical processing units, hardware that excels at parallelisable stencil computations) and then writing code that aims to use the GPU system as efficiently as possible.

2 Mathematical model

This report considers numerical diffusion, which is often implemented to control small-scale noise arising from numerical dispersion, non-linear instability, or external forcing (Xue, 2000). In particular, we focus on 4th order diffusion in the horizontal for a scalar quanity $\varphi(x, y, z, t)$.

This process is described by the equation

$$\partial_t \varphi = -\alpha_4 \, \Delta_\mathrm{h} \left(\Delta_\mathrm{h} \varphi \right), \tag{1}$$

where $\alpha_4 > 0$ and $\Delta_\mathrm{h} = \nabla_\mathrm{h}^2 = \partial_x^2 + \partial_y^2$. Using finite differences with $\Delta x = \Delta y$ and the notation $\varphi_{i,j}^n = \varphi(i\,\Delta x, j\,\Delta x, z, n\,\Delta t)$, the discretisations of the operators ∂_t and Δ_h of equation (1) are then given by

$$\partial_t \varphi_{i,j}^n \approx \frac{1}{\Delta t} \left(\varphi_{i,j}^{n+1} - \varphi_{i,j}^n \right), \tag{2a}$$

$$\begin{aligned} \Delta_\mathrm{h} \varphi_{i,j}^n &\approx \frac{1}{(\Delta x)^2} \Big(\left(\varphi_{i+1,j}^n - 2\varphi_{i,j}^n + \varphi_{i-1,j}^n \right) + \left(\varphi_{i,j+1}^n - 2\varphi_{i,j}^n + \varphi_{i,j-1}^n \right) \Big) \\ &= \frac{1}{(\Delta x)^2} \left(\varphi_{i+1,j}^n + \varphi_{i-1,j}^n + \varphi_{i,j+1}^n + \varphi_{i,j-1}^n - 4\varphi_{i,j}^n \right). \end{aligned} \tag{2b}$$

Given the discretisations in equation (2), the quantity φ is then integrated forwards in time according to the following procedure:

1. update the boundary conditions of $\varphi_{i,j}^n =: (\mathrm{in})_{i,j}^n$ (in our case: periodic),
2. compute $(\mathrm{tmp1})_{i,j}^n = \Delta_\mathrm{h} (\mathrm{in})_{i,j}^n$ from equation (2b),
3. compute $(\mathrm{tmp2})_{i,j}^n = \Delta_\mathrm{h} (\mathrm{tmp1})_{i,j}^n$ from equation (2b),
4. compute $\varphi_{i,j}^{n+1} =: (\mathrm{out})_{i,j}^n = (\mathrm{in})_{i,j}^n - \Delta t \, \alpha_4 \, (\mathrm{tmp2})_{i,j}^n$ from equation (2a),
5. update $(\mathrm{in})_{i,j}^n$ with the new result $(\mathrm{out})_{i,j}^n$.

This procedure is a stencil computation in two dimensions, since the same neighbouring grid points are required for each location on the (horizontal) grid $(x, y) = (i\,\Delta x, j\,\Delta x)$. In addition, all the vertical levels are completely independent of each other, since there is no reference to the z-coordinate. It is thus possible to parallelise the computations in all three spatial dimensions, helping to achieve better performance. Such parallelisations are defined in section 3.2. Note that the implementation of this procedure is shown in appendix A.

3 Parallelisation on GPUs

GPUs contain thousands of cores and are designed to have increased peak performance. A GPU can perform about 7 times as many $\mathrm{FLOP\,s^{-1}}$ as a typical multi-core CPU. Additionally, the internal memory bandwidth of a GPU is roughly 5 times larger than the memory bandwidth of a multi-core CPU. These properties are particularly useful for computation intensive parallelisable programs (like stencil motives). However, parallel components also introduce communication overheads, which results in a trade-off between speedup through parallelisation and slowdown through communication. Some GPU systems also require transfers between the GPU memory and the CPU memory, and this communication is limited by a low bandwidth (about half the CPU bandwidth). It is thus crucial to optimise the usage of GPUs by finding the best kernel sizes for parallelisation, and reducing the amount of memory transferred between the GPU and the CPU. Programming interfaces like CUDA, OpenCL, or OpenACC allow manual handling of these issues.

3.1 OpenACC overview

As a solution relying on compiler directives, OpenACC is particularly versatile. CPU codes can be extended to include GPU functionality. If the system supports GPUs, the compiler will translate the directives to parallel code for GPUs, but if no GPUs are available, the directives will be ignored and the original CPU code will be run. In this sense, OpenACC allows hardware agnostic programming. However, the exact translation of directives to GPU code is compiler dependent, and if optimal performance is desired, the directives are typically designed to circumvent the hardware limitations of a given system architecture. There is a trade-off between practical portability and optimal performance.

In this report, we are interested in optimal performance. We aim to reach a low runtime for the stencil computation described in section 2. The computation is executed on Piz Daint, a supercomputer of CSCS. Our code is run on an XC50 Compute Node which includes a 12 core Intel® Xeon® E5-2690 v3 CPU, and an NVIDIA® Tesla® P100 GPU.

The initial implementation (shown in appendix A) is run on a single core of the CPU, and it follows precisely the structure in section 2. This version of the program is referred to as the *initial CPU code*. The OpenACC directives are then gradually added to parallelise the loops involved in this procedure. However, note that only the spatial loops can be parallelised, as the different timesteps are not independent of each other. The different versions of the GPU code are then run on a unit consisting of a single CPU core and an NVIDIA® GPU. By including more and more specific directives, it is possible to optimise the use of the GPU for our program.

One aspect is the level of parallelism. Depending on the data structure, different loops might perform best with either vector, worker (set of vectors), or gang (set of workers) parallelism. These levels control how the hardware is used for parallel regions. The vector parallelism has the finest granularity (comes closest to a SIMD architecture in Flynn's taxonomy) and is best suited for loops over an index that corresponds to close locations in memory (small stride). In our case, this is the i-index for the x-coordinate. The loop over the next larger stride (j-index) is then best parallelised with workers. The worker parallelism allows for grouping of vectors. And finally the loop over the largest stride (k-index) should be assigned to different gangs, which can run independently from other gangs.

Another important aspect is whether data is stored in the GPU memory or the CPU memory. OpenACC directives allow manual data management to minimise data transfers between the GPU and the CPU. Typically, fewer data transfers improve performance because the bandwidth of the communication bus between the GPU and the CPU is small. However, fewer data transfers also imply that more data needs to be stored in GPU memory. Eventually, there might be a point where the data stored on the GPU memory blocks the efficient usage of resources for parallel computations. It is thus good practice to start with small regions of GPU-only memory usage, and then gradually increase the regions while checking for performance.

3.2 OpenACC implementation

After the general considerations above, we now specify a series of steps which add more and more parallelisation to the initial CPU code. The increments of parallelisation are given below.

(acc01) Add parallel regions and specify parallel loops

Refering to appendix A, the computation intensive loops are the two horizontal Laplacians, and the forward timestep (all other loops only involve data copying). A first step towards parallelising these three loops is to add the OpenACC directives `!$acc parallel`, `!$acc end parallel`, and

`!$acc loop`. As an example, we show the OpenACC code for the computation of the first Laplacian. The other loops are parallelised accordingly.

```
!$acc parallel
!$acc loop
do k = 1, nz
!$acc loop
do j = 1 + num_halo - extend, ny + num_halo + extend
!$acc loop
do i = 1 + num_halo - extend, nx + num_halo + extend
    tmp1_field(i, j, k) = -4._wp * in_field(i, j, k)      &
        + in_field(i - 1, j, k) + in_field(i + 1, j, k)   &
        + in_field(i, j - 1, k) + in_field(i, j + 1, k)
end do
end do
end do
!$acc end parallel
```

(acc02) Add gangs, workers, and vectors to parallel loops

The parallelisation can be made more specific by adding clauses to the `!$acc loop` directive. As mentioned in section 3.1, we add the `gang` clause to the loop with the largest stride (k-index), the `worker` clause to the loop with the next smaller stride (j-index), and the `vector` clause to the loop with the smallest stride (i-index). The code for the first Laplacian is shown below.

```
!$acc parallel
!$acc loop gang
do k = 1, nz
!$acc loop worker
do j = 1 + num_halo - extend, ny + num_halo + extend
!$acc loop vector
do i = 1 + num_halo - extend, nx + num_halo + extend
    tmp1_field(i, j, k) = -4._wp * in_field(i, j, k)      &
        + in_field(i - 1, j, k) + in_field(i + 1, j, k)   &
        + in_field(i, j - 1, k) + in_field(i, j + 1, k)
end do
end do
end do
!$acc end parallel
```

(acc03) Add collapse clauses to parallel loops

While loops with large strides should be mapped to `gang` parallelism, multiple loops with small strides may be collapsed into a single loop with `vector` parallelism. This modifies the code for the first Laplacian as follows.

```
!$acc parallel
!$acc loop gang collapse( 1 )
do k = 1, nz
!$acc loop vector collapse( 2 )
do j = 1 + num_halo - extend, ny + num_halo + extend
do i = 1 + num_halo - extend, nx + num_halo + extend
```

```
    tmp1_field(i, j, k) = -4._wp * in_field(i, j, k)      &
        + in_field(i - 1, j, k) + in_field(i + 1, j, k)   &
        + in_field(i, j - 1, k) + in_field(i, j + 1, k)
end do
end do
end do
!$acc end parallel
```

(acc04) Add data management within the integration loop

As mentioned in section 3.1, the use of GPU resources for parallel computation is further optimised by minimising data transfers between the GPU memory and the CPU memory. As a first step, we specify a region of GPU-only memory usage for the three computation intensive loops (two Laplacians, one forward timestep). This is done by specifying a GPU data region with the `!$acc data` and `!$acc end data` directives. Further specification is added with the clauses `copyin`, `create`, and `copyout`. The `copyin` clause is used to specify which data should be copied in from the CPU to the GPU at the start of the `!$acc data` region. But this data is not copied back out to the CPU at the end of the region. The `create` clause indicates data that is only used on the GPU (first allocated, then modified, finally deleted). Finally, the `copyout` clause specified data which is copied out to the CPU at the end of the `!$acc data` region. This data is not copied in from the CPU at the start of the region. In our case, the `!$acc data` region is implemented as follows.

```
!$acc data copyin( in_field ) create( tmp1_field, tmp2_field ) copyout( out_field )

! horizontal laplacians
extend=1
!$acc parallel
...
!$acc end parallel

extend=0
!$acc parallel
...
!$acc end parallel

! forward timestep
!$acc parallel
...
!$acc end parallel

!$acc end data
```

(acc05) Move data management outside the integration loop

The previous version dealt with a region of three computation intensive loops. Originally this region included six automatic data transfers between the GPU and the CPU (2 per loop). These data transfers were reduced to two with one at the start of the region and one at the end. Even more data transfers can be saved making the GPU data region larger and larger. In our case, it is beneficial to specify the data region outside of the integration loop. This ensures that data in

only transferred before the integration starts, and after the integration has finished. Note that every computation inside a GPU data region need to be contained within an `!$acc parallel` region. This is because the compiler translates code outside of `!$acc parallel` regions as CPU code, regardless if it is within a GPU data region or not. We thus parallelise *every* loop according to the structure presented in the third version above. The overall form of the code is shown below (the full implementation is found in appendix B).

```
!$acc data copyin( in_field ) create( tmp1_field, tmp2_field ) copyout( out_field )

do iter = 1, num_iter
    ! periodic boundary conditions
    !$acc parallel
    ...
    !$acc end parallel



    ! horizontal laplacians
    !$acc parallel
    ...
    !$acc end parallel

    ! forward timestep
    !$acc parallel
    ...

    ! update in_field with out_field (except for last iteration)
    ...
    !$acc end parallel
end do



! periodic boundary conditions
!$acc parallel
...
!$acc end parallel

!$acc end data
```

4 Performance analysis

In this section, we show how the performance depends on both the degree of parallelisation and the size of the horizontal domain. For the parallelisation, we compare the initial CPU code to more and more efficient versions of the GPU code, according to section 3.2. This illustrates the performance improvements that are possible with OpenACC directives. For the domain size, we compare the performance between the initial CPU code and the best GPU code. This allows us to investigate the trade-off between speedup through parallelisation and slowdown through communication overheads, as mentioned in section 3.

4.1 Effect of parallelisation

To investigate the effect of parallelisation, we define a reference domain with $N = 128$ grid points in both horizontal directions ($N_x = N_y = N$), and 64 levels in the vertical direction (N_z). In this domain, we then run six different versions of the diffusion operator from section 2. One version is the original CPU code, and each version after that includes more and more detailed OpenACC directives for improved parallelism.

Table 1 shows the runtimes of the different versions in the reference domain. It is striking that the CPU version (orig) is the second fastest and is only clearly outperformed by one GPU version (acc05). Three versions that are run on the GPU are significantly slower (acc01, acc02, acc03), and one is comparable to the CPU version (acc04). These differences are mainly caused by data bottlenecks between the GPU and the CPU. In the following, we give possible explanations as to how the specific runtimes could have come about.

In version acc01, only the parallel construct was used. This approach tells the compiler to parallelise the specific loop or nest of loops, that is enclosed by the directives, across OpenACC gangs. The compiler does this as efficiently as possible, but it is rather difficult for the compiler to detect parallelisable regions, which leaves the parallelisation on a lower level than the optimum. On the other hand, this method is less error-prone. Version acc02 tries to exploit the OpenACC's three levels of parallelism (gangs, workers, vectors) to make use of the different granularity of parallelism in our code. Version acc03 uses the collapse clause, which turns the two innermost loops into one loop with an overall count of $N_{2D} = N_x N_y$. These three versions all show very similar runtimes. This can definitely be attributed to the small domain size resulting in a relatively small computational load. Overall, these three versions were heavily limited by memory bandwidth, meaning that the data transfer between the GPU and the CPU was far too large compared to the computational load.

To circumvent the excessive data transfer, OpenACC's data directives have been introduced to the code in versions acc04 and acc05. These directives allow for manual allocation of memory on the GPU and specification of transfers between the GPU and the CPU. In version acc04, the data management was done for each timestep separately. The data was allocated on the GPU, and transferred from the host to the device and back again. This, still rather excessive data transfer, already caused a reduction in runtime by about a factor of 3 (compared to acc03, that uses the same OpenACC parallelisation directives, but no data directives). Inserting data directives around the integration loop results in a single data transfer to the GPU at the beginning and a single copy back to the CPU at the end. This yields a remarkable speedup of 15 times compared to the version acc03 with no data directives, and a 5 times speedup compared to the original version that ran purely on CPUs.

Table 1: Runtimes for the different versions of the code on the reference domain.

version	orig	acc01	acc02	acc03	acc04	acc05
runtime	1.422 s	5.733 s	5.766 s	5.707 s	1.784 s	0.386 s

4.2 Effect of domain size

In the previous section, it was established that the version acc05 clearly outperforms the other versions on the reference domain with $N = 128$ and $N_z = 64$. In this section, the performance of this version is tested on larger domains and is compared to the version with the initial CPU code. For the comparison, we define the speedup S as the runtime of the CPU version divided by

the runtime of the GPU version (acc05). The horizontal domain size $N = N_x = N_y$ is increased in powers of 2, while the vertical domain size remains fixed at $N_z = 64$ levels. Table 2 shows the speedup for different domain sizes and one can clearly see that the larger the domain gets, the more beneficial it is to run the code on a GPU.

The larger domains make better use of the GPU's strong computing power. CPUs generally execute instructions sequentially. This requires consistently more time for larger domains, whereas the computation time on GPUs hardly increases when the domain is increased. There is thus an increased performance gap between the CPU and the GPU for larger domains, resulting in higher speedups. This is a clear indication that the GPUs were memory bound and that the potential computing power can be better harnessed with larger domains. As soon as the GPU runtime also increases significantly with an increase in domain size, it can be assumed that they have become compute bound as well. However, this did not occur for our domain sizes.

Table 2: Speedup of the acc05 against the orig version on different domain sizes.

domain size (N)	64	128	256	512	1024	2048
speedup (S)	3.427	3.685	7.988	8.602	8.797	9.067

5 Conclusions

We have presented an example of how to port stencil computations to GPUs using OpenACC compiler directives. We tested different ways to parallelise the code with OpenACC. First, by only implementing parallel regions, and then, by giving the compiler more information on how to parallelise the code exactly. The version with the fastest parallel code on our reference domain ($N = 128$) was then further improved by adding OpenACC data directives. These allow the manual handling of the communication between the host (CPU) and the device (GPU). In this way, we achieved a speedup of up to $S_{\text{max}} = 3.685$ between the best parallelised version and the initial CPU code on the reference domain.

In a second step, we scaled up the domain size to investigate changes in performance of both the original CPU code and the best-performing OpenACC version (including data directives). For domain sizes of N between 64 and 2048, a speedup of $S_{64} = 3.427$ up to $S_{2048} = 9.067$ was observed. Note, however, that more than a doubling in speedup was registered for a doubling of the domain size between $N = 128$ and $N = 256$. Such a drastic increase in speedup only occurred on this step. This sudden increase could likely be attributed to a combination of optimal GPU utilisation and an associated exponentially growing CPU time. However, further investigation is required as to why subsequent doublings of the domain size are accompanied by a smaller speedup ratio.

Compiler directives have proven to be a good tool to utilise parallelisation from a module like the stencil operations. Given our problem of 4th order horizontal diffusion, the most important aspect to improve performance was to run all parallelisable code sequences on the GPU and to minimise data transfers between the GPU and the CPU. It was also shown that the compiler can extract some part of parallelism on its own (version acc01), but needs guiding directives to handle parallelisation more efficiently (versions after acc01). In any case, the main aspect to achieve good performance is proper data management between the host and the device, which was clearly shown when comparing versions acc01 through acc03 with the versions acc04 and acc05. For future implementations, it might be beneficial if weather and climate models are run entirely on GPUs, to avoid data transfer costs altogether.

A Initial CPU code

In this section, we show the relevant sections of the initial CPU code. This is the implementation of the procedure defined in section 2, and forms the basis for the parallelisation with OpenACC directives described in section 3. The loop over the `num_iter` timesteps forms the overall scope, where the last iteration is treated differently (`in_field` not updated with `out_field`). After the integration loop, a final update of the boundary conditions is necessary (halo update for `out_field`). The initial CPU code for this procedure is given below.

```
do iter = 1, num_iter

    ! periodic boundary conditions
    ! bottom edge (without corners)
    do k = 1, nz
    do j = 1, num_halo
    do i = 1 + num_halo, nx + num_halo
        in_field(i, j, k) = in_field(i, j + ny, k)
    end do
    end do
    end do

    ! top edge (without corners)
    do k = 1, nz
    do j = ny + num_halo + 1, ny + 2 * num_halo
    do i = 1 + num_halo, nx + num_halo
        in_field(i, j, k) = in_field(i, j - ny, k)
    end do
    end do
    end do

    ! left edge (including corners)
    do k = 1, nz
    do j = 1, ny + 2 * num_halo
    do i = 1, num_halo
        in_field(i, j, k) = in_field(i + nx, j, k)
    end do
    end do
    end do

    ! right edge (including corners)
    do k = 1, nz
    do j = 1, ny + 2 * num_halo
    do i = nx + num_halo + 1, nx + 2 * num_halo
        in_field(i, j, k) = in_field(i - nx, j, k)
    end do
    end do
    end do
```

```
45      ! horizontal laplacians
46      extend=1
47      do k = 1, nz
48      do j = 1 + num_halo - extend, ny + num_halo + extend
49      do i = 1 + num_halo - extend, nx + num_halo + extend
50          tmp1_field(i, j, k) = -4._wp * in_field(i, j, k)      &
51              + in_field(i - 1, j, k) + in_field(i + 1, j, k)   &
52              + in_field(i, j - 1, k) + in_field(i, j + 1, k)
53      end do
54      end do
55      end do
56
57      extend=0
58      do k = 1, nz
59      do j = 1 + num_halo - extend, ny + num_halo + extend
60      do i = 1 + num_halo - extend, nx + num_halo + extend
61          tmp2_field(i, j, k) = -4._wp * tmp1_field(i, j, k)       &
62              + tmp1_field(i - 1, j, k) + tmp1_field(i + 1, j, k)  &
63              + tmp1_field(i, j - 1, k) + tmp1_field(i, j + 1, k)
64      end do
65      end do
66      end do
67
68
69
70
71      ! forward timestep
72      do k = 1, nz
73      do j = 1 + num_halo, ny + num_halo
74      do i = 1 + num_halo, nx + num_halo
75          out_field(i, j, k) = in_field(i, j, k) - alpha * tmp2_field(i, j, k)
76      end do
77      end do
78      end do
79
80
81
82
83      ! update in_field with out_field (except for last iteration)
84      if ( iter /= num_iter ) then
85          do k = 1, nz
86          do j = 1 + num_halo, ny + num_halo
87          do i = 1 + num_halo, nx + num_halo
88              in_field(i, j, k) = out_field(i, j, k)
89          end do
90          end do
91          end do
92      end if
93
94  end do
95
96
97
```

```
98  ! periodic boundary conditions
99  ! bottom edge (without corners)
100 do k = 1, nz
101 do j = 1, num_halo
102 do i = 1 + num_halo, nx + num_halo
103     out_field(i, j, k) = out_field(i, j + ny, k)
104 end do
105 end do
106 end do
107
108 ! top edge (without corners)
109 do k = 1, nz
110 do j = ny + num_halo + 1, ny + 2 * num_halo
111 do i = 1 + num_halo, nx + num_halo
112     out_field(i, j, k) = out_field(i, j - ny, k)
113 end do
114 end do
115 end do
116
117 ! left edge (including corners)
118 do k = 1, nz
119 do j = 1, ny + 2 * num_halo
120 do i = 1, num_halo
121     out_field(i, j, k) = out_field(i + nx, j, k)
122 end do
123 end do
124 end do
125
126 ! right edge (including corners)
127 do k = 1, nz
128 do j = 1, ny + 2 * num_halo
129 do i = nx + num_halo + 1, nx + 2 * num_halo
130     out_field(i, j, k) = out_field(i - nx, j, k)
131 end do
132 end do
133 end do
```

Note that there are multiple sections with essentially the same code. It is thus possible to define two subroutines: `halo_update` for the boundary conditions, and `laplacian` for the horizontal Laplacians. This would improve readability, but it also complicates the implementation of parallel code. Therefore we stick to the inlined version here.

B Best GPU code

As mentioned in section 3.2, we show here the full implementation of the best GPU code.

```
!$acc data copyin( in_field ) create( tmp1_field, tmp2_field ) copyout( out_field )
do iter = 1, num_iter
    ! periodic boundary conditions
    !$acc parallel
    ! bottom edge (without corners)
    !$acc loop gang collapse( 1 )
    do k = 1, nz
    !$acc loop vector collapse( 1 )
    do j = 1, num_halo
    do i = 1 + num_halo, nx + num_halo
        in_field(i, j, k) = in_field(i, j + ny, k)
    end do
    end do
    end do

    ! top edge (without corners)
    !$acc loop gang collapse( 1 )
    do k = 1, nz
    !$acc loop vector collapse( 1 )
    do j = ny + num_halo + 1, ny + 2 * num_halo
    do i = 1 + num_halo, nx + num_halo
        in_field(i, j, k) = in_field(i, j - ny, k)
    end do
    end do
    end do

    ! left edge (including corners)
    !$acc loop gang collapse( 1 )
    do k = 1, nz
    !$acc loop vector collapse( 1 )
    do j = 1, ny + 2 * num_halo
    do i = 1, num_halo
        in_field(i, j, k) = in_field(i + nx, j, k)
    end do
    end do
    end do

    ! right edge (including corners)
    !$acc loop gang collapse( 1 )
    do k = 1, nz
    !$acc loop vector collapse( 1 )
    do j = 1, ny + 2 * num_halo
    do i = nx + num_halo + 1, nx + 2 * num_halo
        in_field(i, j, k) = in_field(i - nx, j, k)
    end do
    end do
    end do
    !$acc end parallel

```

```
50      ! horizontal laplacians
51      !$acc parallel
52      extend=1
53      !$acc loop gang collapse( 1 )
54      do k = 1, nz
55      !$acc loop vector collapse( 2 )
56      do j = 1 + num_halo - extend, ny + num_halo + extend
57      do i = 1 + num_halo - extend, nx + num_halo + extend
58          tmp1_field(i, j, k) = -4._wp * in_field(i, j, k)  &
59              + in_field(i - 1, j, k) + in_field(i + 1, j, k) &
60              + in_field(i, j - 1, k) + in_field(i, j + 1, k)
61      end do
62      end do
63      end do
64
65      extend=0
66      !$acc loop gang collapse( 1 )
67      do k = 1, nz
68      !$acc loop vector collapse( 2 )
69      do j = 1 + num_halo - extend, ny + num_halo + extend
70      do i = 1 + num_halo - extend, nx + num_halo + extend
71          tmp2_field(i, j, k) = -4._wp * tmp1_field(i, j, k)    &
72              + tmp1_field(i - 1, j, k) + tmp1_field(i + 1, j, k) &
73              + tmp1_field(i, j - 1, k) + tmp1_field(i, j + 1, k)
74      end do
75      end do
76      end do
77      !$acc end parallel
78
79      ! forward timestep
80      !$acc parallel
81      !$acc loop gang collapse( 1 )
82      do k = 1, nz
83      !$acc loop vector collapse( 2 )
84      do j = 1 + num_halo, ny + num_halo
85      do i = 1 + num_halo, nx + num_halo
86          out_field(i, j, k) = in_field(i, j, k) - alpha * tmp2_field(i, j, k)
87      end do
88      end do
89      end do
90
91      ! update in_field with out_field (except for last iteration)
92      if ( iter /= num_iter ) then
93          !$acc loop gang collapse( 1 )
94          do k = 1, nz
95          !$acc loop vector collapse( 2 )
96          do j = 1 + num_halo, ny + num_halo
97          do i = 1 + num_halo, nx + num_halo
98              in_field(i, j, k) = out_field(i, j, k)
99          end do
100         end do
101         end do
102     end if
```

```
103     !$acc end parallel
104
105 end do
106
107 ! periodic boundary conditions
108 !$acc parallel
109 ! bottom edge (without corners)
110 !$acc loop gang collapse( 1 )
111 do k = 1, nz
112 !$acc loop vector collapse( 2 )
113 do j = 1, num_halo
114 do i = 1 + num_halo, nx + num_halo
115     out_field(i, j, k) = out_field(i, j + ny, k)
116 end do
117 end do
118 end do
119
120 ! top edge (without corners)
121 !$acc loop gang collapse( 1 )
122 do k = 1, nz
123 !$acc loop vector collapse( 2 )
124 do j = ny + num_halo + 1, ny + 2 * num_halo
125 do i = 1 + num_halo, nx + num_halo
126     out_field(i, j, k) = out_field(i, j - ny, k)
127 end do
128 end do
129 end do
130
131 ! left edge (including corners)
132 !$acc loop gang collapse( 1 )
133 do k = 1, nz
134 !$acc loop vector collapse( 2 )
135 do j = 1, ny + 2 * num_halo
136 do i = 1, num_halo
137     out_field(i, j, k) = out_field(i + nx, j, k)
138 end do
139 end do
140 end do
141
142 ! right edge (including corners)
143 !$acc loop gang collapse( 1 )
144 do k = 1, nz
145 !$acc loop vector collapse( 2 )
146 do j = 1, ny + 2 * num_halo
147 do i = nx + num_halo + 1, nx + 2 * num_halo
148     out_field(i, j, k) = out_field(i - nx, j, k)
149 end do
150 end do
151 end do
152 !$acc end parallel
153
154 !$acc end data
```

References

Shalf, John (2019). 'Faster code: get more "bang for your buck": Robert Roe interviews John Shalf on the development of digital computing in the post Moore's law era'. In: *Scientific Computing World* 166, pp. 4–6.

Xue, M. (2000). 'High-Order Monotonic Numerical Diffusion and Smoothing'. In: *Monthly Weather Review* 128.8, pp. 2853–2864. DOI: 10.1175/1520-0493(2000)128<2853:HOMNDA>2.0.CO;2.