NVIDIA® HPC Compilers

OPENACC MPI TUTORIAL

DU-09865-001-V20.9 | September 2020

TABLE OF CONTENTS

5× in 5 Hours: Porting a 3D Elastic Wave Simulator to GPUs Using OpenACC.....iii
Chapter 1. Step 0: Evaluation..... 1
Chapter 2. Step 1: Adding Setup Code.....3
Chapter 3. Step 2: Adding Compute Regions.....5
Chapter 4. Step 3: Adding Data Regions.....8
Chapter 5. Step 4: Optimizing Data Transfers..... 10
Chapter 6. Step 5: Loop Schedule Tuning.....13
Chapter 7. Conclusion..... 14

5× IN 5 HOURS: PORTING A 3D ELASTIC WAVE SIMULATOR TO GPUS USING OPENACC

Scientists in the oil and gas industry often investigate large yet highly parallelizable problems that can adapt well to accelerators. SEISMIC_CPML is an example case. Developed by Dimitri Komatitsch and Roland Martin from University of Pau, France. From their website: "SEISMIC_CPML is a set of ten open-source Fortran 90 programs to solve the two-dimensional or three-dimensional isotropic or anisotropic elastic, viscoelastic or poroelastic wave equation using a finite-difference method with Convolutional or Auxiliary Perfectly Matched Layer (C-PML or ADE-PML) conditions."

In particular, we decided to accelerate the 3D Isotropic application which is "a 3D elastic finite-difference code in velocity and stress formulation with Convolutional-PML (C-PML) absorbing conditions." In addition to being highly compute intensive, the code uses MPI and OpenMP to perform the domain decomposition, giving us an opportunity to showcase the use of multi-GPU programming.

This tutorial will give you an understanding of the steps involved in porting applications to GPUs using OpenACC, some optimization tips, and ways to identify several potential pitfalls. It should be useful as a guide for how you can apply OpenACC directives to your own code. All source code for this tutorial can be downloaded from NVIDIA Tutorials webpage, https://www.pgroup.com/resources/tutorials.htm?tab=tutorials. After downloading, unpack and untar the file which will create a directory where you can work along with the tutorial if you like.

Chapter 1. STEP 0: EVALUATION

The most difficult part of accelerator programing begins before the first line of code is written. Having the right algorithm is essential for success. An accelerator is like an army of ants. Individually the cores are weak, but taken together they can do great things. If your program is not highly parallel, an accelerator won't be of much use. Hence, the first step is to ensure that your algorithm is parallel. If it's not, you should determine if there are alternate algorithms you might use or if your algorithm could be reworked to be more parallel.

In evaluating the SEISMIC_CPML 3-D Isotropic code, we see the main time-step loop contains eight parallel loops. However, being parallel may not be enough if the loops are not computationally intensive. While computational intensity shouldn't be your only metric, it is a good indicator of potential success. Using the compiler flag -Minfo=intensity, we can see that the compute intensity, the ratio of computation to data movement, of the various loops is between 2.5 and 2.64. These are average intensities. As a rule, anything below 1.0 is generally not worth accelerating unless it is part of a larger program. Ideally we would like the average intensity to be above 4, but 2.5 is sufficient to move forward.

SEISMIC_CPML uses MPI to decompose the problem space across the Z dimension. This will allow us to utilize more than one GPU, but it also adds extra data movement as the program needs to pass halos (regions of the domain that overlap across processes). We could use OpenMP threads as well, but doing so would add more programming effort and complexity to our example. As a result, we chose to remove the OpenMP code from the GPU version. We may revisit that decision in a future article.

As an aside, with the MPI portion of the code the programmer manually decomposes the domain. With OpenMP, the compiler does that automatically if the program is running in a shared memory environment. Currently, OpenMP compilers aren't able to automatically decompose a problem across multiple discrete memory spaces as is the case when using multiple devices. As a result, the programmer must manually decompose the problem just like they would using MPI. Because this would basically require us to duplicate our effort, we decided to forgo using OpenMP here.

To build and run the original MPI/OpenMP code on your system do the following (make sure that the compiler (`pgfortran`) and MPI wrappers (`mpif90`) are in your path before building and running):

```
cd step0
make build
make run
make verify
```

Chapter 2. STEP 1: ADDING SETUP CODE

Because this is an MPI code where each process will use its own GPU, we need to add some utility code to ensure that happens. The **`setDevice`** routine first determines which node the process is on (via a call to **`hostid`**) and then gathers the hostids from all other processes. It then determines how many GPUs are available on the node and assigns the devices to each process.

Note that in order to maintain portability with the CPU version, this section of code is guarded by the preprocessor macro **`_OPENACC`**, which is defined when the OpenACC directives are enabled in the HPC Fortran compiler through the use of the -acc command-line compiler option.

```
#ifdef _OPENACC
#
function setDevice(nprocs,myrank)

  use iso_c_binding
  use openacc
  implicit none
  include 'mpif.h'

  interface
    function gethostid() BIND(C)
      use iso_c_binding
      integer (C_INT) :: gethostid
    end function gethostid
  end interface

  integer :: nprocs, myrank
  integer, dimension(nprocs) :: hostids, localprocs
  integer :: hostid, ierr, numdev, mydev, i, numlocal
  integer :: setDevice

! get the hostids so we can determine what other processes are on this node
  hostid = gethostid()
  CALL mpi_allgather(hostid,1,MPI_INTEGER,hostids,1,MPI_INTEGER, &
                     MPI_COMM_WORLD,ierr)

! determine which processors are on this node
  numlocal=0
  localprocs=0
  do i=1,nprocs
    if (hostid .eq. hostids(i)) then
      localprocs(i)=numlocal
      numlocal = numlocal+1
    endif
  enddo

! get the number of devices on this node
  numdev = acc_get_num_devices(ACC_DEVICE_NVIDIA)

  if (numdev .lt. 1) then
    print *, 'ERROR: There are no devices available on this host.  &
              ABORTING.', myrank
    stop
  endif

! print a warning if the number of devices is less then the number
! of processes on this node.  Having multiple processes share devices is not
! recommended.
  if (numdev .lt. numlocal) then
   if (localprocs(myrank+1).eq.1) then
     ! print the message only once per node
   print *, 'WARNING: The number of process is greater then the number  &
             of GPUs.', myrank
   endif
   mydev = mod(localprocs(myrank+1),numdev)
  else
   mydev = localprocs(myrank+1)
  endif

 call acc_set_device_num(mydev,ACC_DEVICE_NVIDIA)
 call acc_init(ACC_DEVICE_NVIDIA)
 setDevice = mydev

end function setDevice
#endif
```

To build and run the `step1` code on your system do the following (make sure that the compiler (`pgfortran`) and MPI wrappers (`mpif90`) are in your path before building and running):

```
cd step1
make build
make run
make verify
```

Chapter 3. STEP 2: ADDING COMPUTE REGIONS

Next, we spent a few minutes adding six compute regions around the eight parallel loops. For example, here's the final reduction loop.

```
!$acc kernels
  do k = kmin,kmax
    do j = NPOINTS_PML+1, NY-NPOINTS_PML
      do i = NPOINTS_PML+1, NX-NPOINTS_PML

! compute kinetic energy first, defined as 1/2 rho ||v||^2
! in principle we should use rho_half_x_half_y instead of rho for vy
! in order to interpolate density at the right location in the staggered grid
! cell but in a homogeneous medium we can safely ignore it

      total_energy_kinetic = total_energy_kinetic + 0.5d0 * rho*( &
              vx(i,j,k)**2 + vy(i,j,k)**2 + vz(i,j,k)**2)

! add potential energy, defined as 1/2 epsilon_ij sigma_ij
! in principle we should interpolate the medium parameters at the right location
! in the staggered grid cell but in a homogeneous medium we can safely ignore it

! compute total field from split components
      epsilon_xx = ((lambda + 2.d0*mu) * sigmaxx(i,j,k) - lambda *  &
      sigmayy(i,j,k) - lambda*sigmazz(i,j,k)) / (4.d0 * mu * (lambda + mu))
      epsilon_yy = ((lambda + 2.d0*mu) * sigmayy(i,j,k) - lambda *  &
          sigmaxx(i,j,k) - lambda*sigmazz(i,j,k)) / (4.d0 * mu * (lambda + mu))
      epsilon_zz = ((lambda + 2.d0*mu) * sigmazz(i,j,k) - lambda *  &
          sigmaxx(i,j,k) - lambda*sigmayy(i,j,k)) / (4.d0 * mu * (lambda + mu))
      epsilon_xy = sigmaxy(i,j,k) / (2.d0 * mu)
      epsilon_xz = sigmaxz(i,j,k) / (2.d0 * mu)
      epsilon_yz = sigmayz(i,j,k) / (2.d0 * mu)

      total_energy_potential = total_energy_potential + &
        0.5d0 * (epsilon_xx * sigmaxx(i,j,k) + epsilon_yy * sigmayy(i,j,k) + &
        epsilon_yy * sigmayy(i,j,k)+ 2.d0 * epsilon_xy * sigmaxy(i,j,k) + &
        2.d0*epsilon_xz * sigmaxz(i,j,k)+2.d0*epsilon_yz * sigmayz(i,j,k))

      enddo
    enddo
  enddo
!$acc end kernels
```

The -acc command line option to the HPC Accelerator Fortran compiler enables OpenACC directives. Note that OpenACC is meant to model a generic class of devices. While NVIDIA is the current market leader in HPC accelerators and default target for NVIDIA's OpenACC implementation, the model can, and will in the future, target other devices.

Another compiler option you'll want to use during development is -Minfo, which causes the compiler to output feedback on optimizations and transformations performed on your code. For accelerator-specific information, use the -Minfo=accel sub-option. Examples of feedback messages produced when compiling SEISMIC_CPML include:

```
1113, Generating copyin(vz(11:91,11:631,kmin:kmax)) Generating
copyin(vy(11:91,11:631,kmin:kmax)) Generating copyin(vx(11:91,11:631,kmin:kmax))
Generating copyin(sigmaxx(11:91,11:631,kmin:kmax))
Generating copyin(sigmayy(11:91,11:631,kmin:kmax))
Generating copyin(sigmazz(11:91,11:631,kmin:kmax)) Generating
copyin(sigmaxy(11:91,11:631,kmin:kmax)) Generating
copyin(sigmaxz(11:91,11:631,kmin:kmax)) Generating
copyin(sigmayz(11:91,11:631,kmin:kmax))
```

To compute on a GPU, the first step is to move data from host memory to GPU memory. In the example above, the compiler tells you that it is copying over nine arrays. Note the copyin statements. These mean that the compiler will only copy the data to the GPU but not copy it back to the host. This is because line 1113 corresponds to the start of the reduction loop compute region, where these arrays are used but never modified. Other data movement clauses you may see include copy where the data is copied to the device at the beginning of the region and copied back at the end of the region, and copyout where the data is only copied back to the host.

Notice that the compiler is only copying an interior subsection of the arrays. By default, the compiler is conservative and only copies the data that's actually required to perform the necessary computations. Unfortunately, because the interior sub-arrays are not contiguous in host memory, the compiler needs to generate multiple data transfers for each array. Overall GPU performance is determined largely by how well we can optimize memory transfers. That means not just how much data gets transferred, but how many transfers occur. Transferring multiple sub-arrays is very costly. For now we will just note it. Later, we'll look at improving performance by overriding the compiler defaults and copying entire arrays in one large contiguous block.

```
1114, Loop is parallelizable 1115, Loop is parallelizable 1116, Loop is parallelizable
Accelerator kernel generated
```

Here the compiler has performed dependence analysis on the loops at lines 1114, 1115, and 1116 (the reduction loop shown earlier). It finds that all three loops are parallelizable so it generates an accelerator kernel. When the compiler encounters loops that can not be parallelized, it generally reports a reason why so that you can adapt the code accordingly. If inner loops are not parallelizable, a kernel may still be generated for outer loops; in those cases the inner loop(s) will run sequentially on the GPU cores.

The compiler may attempt to work around dependences that prevent parallelization by interchanging loops (i.e changing the order) where it's safe to do so. At least one outer or interchanged loop must be parallel for an accelerator kernel to be generated. In some cases you may be able to use the loop directive independent clause to work around potential dependences, or the private clause to eliminate a dependence entirely. In other cases, code may need to be restructured significantly to enable parallelization.

The generated accelerator kernel is just a serial bit of code that gets executed on the GPU by many threads simultaneously. Every thread will be executing the same code but operating on different data. How the threads are organized is called the loop schedule. Below we can see the loop schedule for our reduction loop. The do loops have been replaced with a three-dimensional gang, which in turn is composed of a two-dimensional vector section.

1114, !$acc loop gang ! blockidx%y 1115, !$acc loop gang, vector(4) ! blockidx%z threadidx%y 1116, !$acc loop gang, vector(32) ! blockidx%x threadidx%x

In CUDA terminology, the gang clause corresponds to a grid dimension and the vector clause corresponds to a thread block dimension. For new or non-CUDA programmers, we highly recommend reading Michael Wolfe's PGInsider article *Understanding the Data Parallel Threading Model for GPUs*.

Don't feel too overwhelmed by loop schedules. It's just a way to organize how the GPU threads act on data elements of an array. So here we have a 3-D array that's being grouped into blocks of 32×4 elements where a single thread is working on a specific element. Because the number of gangs is not specified in the loop schedule, it will be determined dynamically when the kernel is launched. If the **`gang`** clause had a fixed width, such as **`gang(16)`**, then each kernel would be written to loop over multiple elements.

With CUDA, programming reductions and managing shared memory can be a fairly difficult task. In the example below, the compiler has automatically generated optimal code using these features. By the way, the compiler is always looking for opportunities to optimize your code.

1122, Sum reduction generated for total_energy_kinetic 1140, Sum reduction generated for total_energy_potential

OK, so how are we doing?

To build and run the `step2` code on your system do the following (as mention above, make sure that the compiler (`pgfortran`) and MPI wrappers (`mpif90`) are in your path before building and running):

```
cd step2
make build
make run
make verify
```

After ten minutes of programming time, we've managed to make the program really really slow! Remember this is an article about speeding-up SEISMIC_CPML by 5× in 5 hours not slowing it down 3× in ten minutes. We'll soon overcome this slowdown but it's not uncommon to see this type of one step back experience when programming GPUs.

Don't get discouraged if this happens to you. So why the slowdown and how can we fix it?

Chapter 4. STEP 3: ADDING DATA REGIONS

You may have guessed that the slowdown is caused by excessive data movement between host memory and GPU memory. In looking at a section of our CUDA profile information, we see that each compute region is spending a lot of time copying data back and forth between the host and device. Because each compute region is executed 2500 times, this makes for a lot of data movement. Adding up all the data transfer times in the profile output for Step 2 shows that the vast majority, over 99%, of the time was spent copying data while only a small fraction was spent in the compute kernels. The remaining time is spent either in host code or blocked waiting on data transfers (both MPI processes must use the same PCIe bus to transfer data).

Note that the exact amout of time spent copying data or computing on the device will vary between systems. To see the time for your system, set the environment variable **`PGI_ACC_TIME=1`** and run your executable. This option prints basic profile information such as the kernel execution time, data transfer time, initialization time, the actual launch configuration, and total time spent in a compute region. Note that the total time is measured from the host and includes time spent executing host code within a region.

To improve performance, we need to find a way to minimize the amount of time transferring data. Enter the data directive. You can use a data region to specify exact points in your program where data should be copied from host memory to GPU memory, and back again. Any compute region enclosed within a data region will use the previously copied data, without the need to copy at the boundaries of the compute region. A data region can span across host code and multiple compute regions, and even across subroutine boundaries.

In looking at the arrays in SEISMIC_CMPL, there are 18 arrays with constant values. Another 21 are used only within compute regions so are never needed on the host. Let's start by adding a data region around the outer time step loop. The final three arrays do need to be copied back to the host to pass their halos. For those cases, we use the update directive.

```
!---
!---  beginning of time loop
!---
!$acc data &
!$acc copyin(a_x_half,b_x_half,k_x_half,                        &
!$acc        a_y_half,b_y_half,k_y_half,                        &
!$acc        a_z_half,b_z_half,k_z_half,                        &
!$acc        a_x,a_y,a_z,b_x,b_y,b_z,k_x,k_y,k_z,               &
```

```
!$acc        sigmaxx,sigmaxz,sigmaxy,sigmayy,sigmayz,sigmazz,  &
!$acc        memory_dvx_dx,memory_dvy_dx,memory_dvz_dx,        &
!$acc        memory_dvx_dy,memory_dvy_dy,memory_dvz_dy,        &
!$acc        memory_dvx_dz,memory_dvy_dz,memory_dvz_dz,        &
!$acc        memory_dsigmaxx_dx, memory_dsigmaxy_dy,           &
!$acc        memory_dsigmaxz_dz, memory_dsigmaxy_dx,           &
!$acc        memory_dsigmaxz_dx, memory_dsigmayz_dy,           &
!$acc        memory_dsigmayy_dy, memory_dsigmayz_dz,           &
!$acc        memory_dsigmazz_dz)

  do it = 1,NSTEP

...

!$acc update host(sigmazz,sigmayz,sigmaxz)
! sigmazz(k+1), left shift
  call MPI_SENDRECV(sigmazz(:,:,1),number_of_values,MPI_DOUBLE_PRECISION, &
         receiver_left_shift,message_tag,sigmazz(:,:,NZ_LOCAL+1), &
         number_of_values,

...

!$acc update device(sigmazz,sigmayz,sigmaxz)

...

  ! --- end of time loop
  enddo
!$acc end data
```

Data regions can be nested, and in fact we used this feature in the time loop body for the arrays **vx**, **vy** and **vz** as shown below. While these arrays are copied back and forth at the inner data region boundary, and so are moved more often than the arrays moved in the outer data region, they are used across multiple compute regions instead of being copied at each compute region boundary. Note that we do not specify any array dimensions in the copy clause. This instructs the compiler to copy each array in its entirety as a contiguous block, and eliminates the inefficiency we noted earlier when interior sub-arrays were being copied in multiple blocks.

```
!$acc data copy(vx,vy,vz)

... data region spans over 5 compute regions and host code

!$acc kernels

...

!$acc end kernels

!$acc end data
```

How are we doing on our timings?

To build and run the `step3` code on your system do the following:

```
cd step3
make build
make run
make verify
```

This step took just about an hour of coding time and reduced our execution time significantly. We're making good progress, but we can improve the performance even further.

Chapter 5. STEP 4: OPTIMIZING DATA TRANSFERS

For our next step we'll work to optimize the data transfers even further by migrating as much of the computation as we can over to the GPU and moving only the absolute minimum amount of data required. The first step is to move the start of the outer data region up so that it occurs earlier in the code, and to put the data initialization loops into compute kernels. This includes the **vx**, **vy** and **vz** arrays. Using this approach enables us to remove the inner data region used in our previous optimization step.

In the following example code, notice the use of the create clause. This instructs the compiler to allocate space for variables in GPU memory for local use but to perform no data movement on those variables. Essentially they are used as scratch variables in GPU memory.

```
!$acc data                                                          &
!$acc copyin(a_x_half,b_x_half,k_x_half,                            &
!$acc        a_y_half,b_y_half,k_y_half,                            &
!$acc        a_z_half,b_z_half,k_z_half,                            &
!$acc        ix_rec,iy_rec,                                         &
!$acc        a_x,a_y,a_z,b_x,b_y,b_z,k_x,k_y,k_z),                  &
!$acc copyout(sisvx,sisvy),                                         &
!$acc create(memory_dvx_dx,memory_dvy_dx,memory_dvz_dx,             &
!$acc        memory_dvx_dy,memory_dvy_dy,memory_dvz_dy,             &
!$acc        memory_dvx_dz,memory_dvy_dz,memory_dvz_dz,             &
!$acc        memory_dsigmaxx_dx, memory_dsigmaxy_dy,                &
!$acc        memory_dsigmaxz_dz, memory_dsigmaxy_dx,                &
!$acc        memory_dsigmaxz_dx, memory_dsigmayz_dy,                &
!$acc        memory_dsigmayy_dy, memory_dsigmayz_dz,                &
!$acc        memory_dsigmazz_dz,                                    &
!$acc        vx,vy,vz,vx1,vy1,vz1,vx2,vy2,vz2,                      &
!$acc        sigmazz1,sigmaxz1,sigmayz1,                            &
!$acc        sigmazz2,sigmaxz2,sigmayz2)                            &
!$acc copyin(sigmaxx,sigmaxz,sigmaxy,sigmayy,sigmayz,sigmazz)

...

! Initialize vx, vy and vz arrays on the device
!$acc kernels
  vx(:,:,:) = ZERO
  vy(:,:,:) = ZERO
  vz(:,:,:) = ZERO
!$acc end kernels

...
```

One caveat to using data regions is that you must be aware of which copy (host or device) of the data you are actually using in a given loop or computation. The host and device copies of the data are not automatically kept coherent. That is the responsibility of the programmer when using data regions. For example, any update to the copy of a variable in device memory won't be reflected in the host copy until you specify that it should be updated using either an update directive or a copy clause at a data or compute region boundary.

Unintentional loss of coherence between the host and device copy of a variable is one of the most common causes of validation errors in OpenACC programs. After making the above change to SEISMIC_CPML, the code generated incorrect results. After nearly a half hour of debugging, we determined that the section of the time step loop that initializes boundary conditions was omitted from an OpenACC compute region. As a result we were initializing the host copy of the data, rather than the device copy as intended, which resulted in uninitialized variables in device memory.

The next challenge in optimizing the data transfers related to the handling of the halo regions. SEISMIC_CPML passes halos from six 3-D arrays between MPI processes during the course of the computations. Ideally we would simply copy back the 2-D halo sub-arrays using update directives or copy clauses, but as we saw earlier copying non-contiguous array sections between host and device memory is very inefficient. As a first step, we tried copying the entire arrays from device memory back to the host before passing the halos. This was also very inefficient, given that only a small amount of the data moved between host and device memory was needed in the eventual MPI transfers.

After some experimentation, we settled on an approach whereby we added six new temporary 2-D arrays to hold the halo data. Within a compute region we gathered the 2-D halos from the main 3-D arrays into the new temp arrays, copied the temporaries back to the host in one contiguous block, passed the halos between MPI processes, and finally copied the exchanged values back to device memory and scattered the halos back into the 3-D arrays. While this approach does add to the kernel execution time, it saves a considerable amount of data transfer time.

In the example code below, note that the source code added to support the halo gathers and transfers is guarded by the preprocessor **`_OPENACC`** macro and will only be executed if the code is compiled by an OpenACC-enabled compiler.

```
#ifdef _OPENACC
#
! Gather the sigma 3D arrays to a 2D slice to allow for faster
! copy from the device to host
!$acc kernels
   do i=1,NX
    do j=1,NY
      vx1(i,j)=vx(i,j,1)
      vy1(i,j)=vy(i,j,1)
      vz1(i,j)=vz(i,j,NZ_LOCAL)
    enddo
  enddo
!$acc end kernels
!$acc update host(vxl,vyl,vzl)

! vx(k+1), left shift
  call MPI_SENDRECV(vx1(:,:), number_of_values, MPI_DOUBLE_PRECISION, &
       receiver_left_shift, message_tag, vx2(:,:), number_of_values, &
       MPI_DOUBLE_PRECISION, sender_left_shift, message_tag, MPI_COMM_WORLD,&
       message_status, code)
```

```
! vy(k+1), left shift
  call MPI_SENDRECV(vy1(:,:), number_of_values, MPI_DOUBLE_PRECISION, &
       receiver_left_shift,message_tag, vy2(:,:),number_of_values,   &
       MPI_DOUBLE_PRECISION, sender_left_shift, message_tag, MPI_COMM_WORLD,&
       message_status, code)

! vz(k-1), right shift
  call MPI_SENDRECV(vz1(:,:), number_of_values, MPI_DOUBLE_PRECISION, &
       receiver_right_shift, message_tag, vz2(:,:), number_of_values, &
       MPI_DOUBLE_PRECISION, sender_right_shift, message_tag, MPI_COMM_WORLD, &
       message_status, code)

!$acc update device(vx2,vy2,vz2)
!$acc kernels
  do i=1,NX
    do j=1,NY
      vx(i,j,NZ_LOCAL+1)=vx2(i,j)
      vy(i,j,NZ_LOCAL+1)=vy2(i,j)
      vz(i,j,0)=vz2(i,j)
    enddo
  enddo
!$acc end kernels

#else
```

The above modifications required about two hours of coding time, but the total execution time has been reduced significantly with only a small fraction of time spent copying data!

To build and run the `step4` code on your system do the following:

```
cd step4
make build
make run
make verify
```

Chapter 6. STEP 5: LOOP SCHEDULE TUNING

The final step in our tuning process was to tune the OpenACC compute region loop schedules using the gang, worker and vector clauses. In many cases, including this code, the default kernel schedules chosen by the NVIDIA OpenACC compiler are quite good. Manual tuning efforts often don't improve timings significantly. However, in some cases the compiler doesn't do as well. It's always worthwhile to spend a little time examining whether you can do better by overriding compiler-generated loop schedules using explicit loop scheduling clauses. You can usually tell fairly quickly if the clauses are having an effect.

Unfortunately, there is no well-defined method for finding an optimal kernel schedule (short of trying all possible schedules). The best advice is to start with the compiler's default schedule and try small adjustments to see if and how they affect execution time. The kernel schedule you choose will affect whether and how shared memory is used, global array accesses, and various types of optimizations. Typically, it's better to perform gang scheduling of loops with large iteration counts.

```
!$acc loop gang
  do k = k2begin,NZ_LOCAL
    kglobal = k + offset_k
!$acc loop worker vector collapse(2)
    do j = 2,NY
      do i = 2,NX
```

To build and run the `step5` code on your system do the following:

```
cd step5
make build
make run
make verify
```

Chapter 7. CONCLUSION

In a little over five hours of programming time we achieved about a 7× speed-up over the original MPI/OpenMP version running on our test system. On a much larger cluster running a much larger dataset speed-up is not quite as good. There is additional overhead in the form of inter-node communication, and the CPUs on the system have more cores and run at a higher clock rate. Nonetheless, the speed-up is still nearly 5×.