Programming GPUs using Directives

Alan Gray

EPCC

The University of Edinburgh

Overview

- Introduction to GPU Directives
- OpenACC
- New OpenMP 4 accelerator support

Accelerator Directives

- Language extensions, e.g. *Cuda* or *OpenCL*, allow programmers to interface with the GPU
  - This gives control to the programmer, but is often tricky and time consuming, and results in complex/non-portable code
- An alternative approach is to allow the compiler to automatically accelerate code sections on the GPU (including decomposition, data transfer, etc).
- There must be a mechanism to provide the compiler with hints regarding which sections to be accelerated, parallelism, data usage, etc
- *Directives* provide this mechanism
  - Special syntax which is understood by accelerator compilers and ignored (treated as code comments) by non-accelerator compilers.
  - Same source code can be compiled for CPU/GPU combo or CPU only
  - c.f. OpenMP for multi-core programming.

- OpenACC standard was announced in Nov 2011
  - By CAPS, CRAY, NVIDIA and PGI
  - Standardisation based on several pre-existing models
  - Examples later

OpenACC

- We will now illustrate accelerator directives using OpenACC
- For definitive guide, full list of available directives, clauses, options etc see

http://www.openacc-standard.org/

OpenACC Directives

- With directives inserted, the compiler will attempt to compile the key kernels for execution on the GPU, and will manage the necessary data transfer automatically.
- Directive format:
  - C: `#pragma acc ....`
  - Fortran: `!$acc ....`
- These are ignored by non-accelerator compilers

Accelerator Parallel Construct

- The programmer specifies which regions of code should be offloaded to the accelerator with the `parallel` construct

C:

```
#pragma acc parallel
{
…code region…
}
```

Fortran:

```
!$acc parallel
…code region…
!$acc end parallel
```

- Note: this directive is usually not sufficient on it’s own – it needs (at least) to be combined with the `loop` directive (next slide)

Accelerator Loop Construct

- The `loop` construct is applied immediately before a loop (or nest of loops), specifying that it should be parallelised on the accelerator.

C:

```
#pragma acc loop
for(…){
…loop body…
}
```

Fortran:

```
!$acc loop
do …
…loop body…
end do
```

Accelerator Loop Construct

- The `loop` construct must be used inside a `parallel` construct.

C:

```
#pragma acc parallel
{
…
#pragma acc loop
 for(…){
 …loop body…
 }
…
}
```

Fortran:

```
!$acc parallel
…
!$acc loop …
do …
…loop body…
end do
!$acc end loop
…
!$acc end parallel
```

- Multiple `loop` constructs may be used within a single `parallel` construct.

Accelerator Loop Construct

- The `parallel loop` construct is shorthand for the combination of a `parallel` and (single) `loop` construct

C:

```
#pragma acc parallel loop
for(…){
…loop body…
}
```

Fortran:

```
!$acc parallel loop
do …
…loop body…
end do
!$acc end parallel loop
```

Parallel Loop Example

```
!$acc parallel loop
do i=1, N
   output(i)=2.*input(i)
end do
!$acc end parallel loop
```

- Compiler automatically offloads loop to GPU (using default values for the parallel decomposition), and performs necessary data transfers.
- Use of `parallel loop` may be sufficient, on its own, to get code running on the GPU, but further directives and clauses exist to give more control to programmer
    - to improve performance and enable more complex cases

Tuning Clauses for loop construct

Clauses can be added to `loop` (or `parallel loop`) directives

- `gang, vector`
  - Targets specific loop at specific level of hardware
    - gang↔ CUDA block of threads
    - vector ↔ CUDA threads in block
  - You can specify both together

    `!$acc loop gang vector` schedules loop over all hardware

Tuning Clauses for parallel construct

To be added to `parallel` (or `parallel loop`) directives

- `num_gangs, vector_length`
  - Tunes the amount of parallelism used
  - equivalent to setting the number of threads per block, number of blocks in CUDA
- Set the number of threads per block by specifying the `vector_length(NTHREADS)` clause
  - NTHREADS must be one of: 1, 64, 128 (default), 256, 512, 1024
- E.g. to specify 128 threads per block

```
#pragma acc parallel vector_length(128)
```

Other clauses

Further clauses to `loop` (or `parallel loop`) directives

- `seq`: loop executed sequentially
- `independent`: compiler hint
- `if(logical)` Executes on GPU if .TRUE. at runtime, otherwise on CPU
- `reduction`: as in OpenMP
- `cache`: specified data held in software-managed data cache e.g. explicit blocking to shared memory on NVIDIA GPUs

Data Management

- Consider case with 2 loops.

```
!$acc parallel loop
do i=1, N
   output(i)=2.*input(i)
end do
!$acc end parallel loop

write(*,*) "finished 1st region"

!$acc parallel loop
do i=1, N
   output(i)=i*output(i)
end do
!$acc end parallel loop
```

- The `output` array will be unnecessarily copied from/to device between regions
    - Host/device data copies are very expensive

Accelerator Data Construct

- Allows more efficient memory management

C:

```
#pragma acc data
{
…code region…
}
```

Fortran:

```
!$acc data
…code region…
!$acc end data
```

Accelerator Data Construct

```
!$acc data copyin(input) copyout(output)

!$acc parallel loop
do i=1, N
   output(i)=2.*input(i)
end do
!$acc end parallel loop

write(*,*) "finished first region"

!$acc parallel loop
do i=1, N
   output(i)=i*output(i)
end do
!$acc end parallel loop

!$acc end data
```

- the `output` array is no longer unnecessarily transferred between host and device between kernel calls

Data clauses

Applied to `data, parallel [loop]` regions

- `copy, copyin, copyout`
  - copy data "in" to GPU at start of region and/or "out" to CPU at end
  - `copy` means `copyin` and `copyout`
  - Supply list of arrays or array sections (using ":” notation)
    - N.B.Fortran uses start:end; C/C++uses start:length
    - e.g. first N elements of array: Fortran `1:N`; C/C++ `0:N`
- `create`
  - Do not copy at all – useful for temporary arrays
  - Host copy still exists
- `private, firstprivate`
  - as per OpenMP
  - Scalars private by default
- `present`
  - Specify that data is already present on the GPU, so copying should be avoided (example later)

Sharing GPU data between subroutines

```
PROGRAM main
   INTEGER :: a(N)
   …
!$acc data copy(a)
!$acc parallel loop
   DO i = 1,N
        a(i) = i
   ENDDO
!$acc end parallel loop
   CALL double_array(a)
!$acc end data
   …
END PROGRAM main
```

```
SUBROUTINE double_array(b)
   INTEGER :: b(N)
!$acc parallel loop present(b)
   DO i = 1,N
        b(i) = 2*b(i)
   ENDDO
!$acc end parallel loop
END SUBROUTINE double_array
```

- The `present` data clause allows the programmer to specify that the data is already on the device, so should not be copied again

- See the documentation for full list of directives and clauses.
- **Runtime Library Routines** are available to, e.g.
  - Retrieve information about the GPU hardware environment
  - Specify which device to use
  - Explicitly initialize the accelerator (helps with benchmarking)
- **Environment Variables**
  - e.g. can be set to specify which device to use
- There are still a number of limitations with the model and current implementations
  - Meaning that the feasibility of use depends on the complexity of code

OpenMP Accelerator Directives

OpenMP®

- All OpenACC partners plus Intel, AMD (and several other organisations including EPCC) formed a subcommittee of the OpenMP committee, looking at extending the OpenMP directive standard to support accelerators.
- More comprehensive and with wider remit than OpenACC
- Accelerator support now included in OpenMP 4
  - But implementations are not yet widely available

OpenMP 4 Accelerator support

- Similar to, but not the same as, OpenACC directives.
- Support for more than just loops
- Less reliance on compiler to parallelise and map code to threads
- Advantage of being portable: not GPU specific
  - suitable for Xeon Phi or DSPs, for example
  - This has the disadvantage that the syntax is more complex than OpenACC when used for GPUs
- Fully integrated into the rest of OpenMP

Summary

- A directives based approach to programming GPUs is potentially much more simplistic and productive than direct use of language extensions
  - More portable
  - Higher level: compiler automates much of the work
  - Less flexible and possibly poorer performance
  - Limitations may offer challenges for complex codes
- The OpenACC standard emerged in 2011
  - We went through the main concepts with examples
- OpenMP 4 now incorporates accelerators
  - More comprehensive and with wider participation than OpenACC