Programming GPUs using Directives

Alan Gray
EPCC
The University of Edinburgh
• Alistair Hart (Cray) and Mark Bull (EPCC) contributed material to this lecture.
• Introduction to GPU Directives

• OpenACC

• New OpenMP 4.0 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*
Accelerator Directives

• Several variants emerged, including
  – **PGI** Accelerator Compiler for C/Fortran
    – Using directives, translates to CUDA (future releases to PTX)
    – NVIDIA GPUs only
  – **CAPS** HMPP compiler
    – Using directives, translates to CUDA or OpenCL
      – NVIDIA or AMD GPUs
    – **Pathscale** compiler supports NVIDIA GPUs through HMPP programming model
  – **Cray** compiler
    – Cray systems only

• All of above are commercial products

• Directives different but similar across products
• OpenACC standard announced in November 2011
  – By CAPS, CRAY, NVIDIA and PGI
• Only applied to NVIDIA GPUs only (so far)
• Examples later
• 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 4.0
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/
• 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 its own – it needs (at least) to be combined with the `loop` directive (next slide).
The *loop* construct is applied immediately before a loop, specifying that it should be parallelised on the accelerator.

C:
```c
#pragma acc loop
for(...) {
    ...loop body...
}
```

Fortran:
```fortran
!$acc loop
do ...
    ...loop body...
end do
```
• 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.
The parallel loop construct is shorthand for the combination of a parallel and (single) loop construct.

**C:**
```c
#pragma acc parallel loop
for (...) {
...loop body...
}
```

**Fortran:**
```fortran
!$acc parallel loop
do ...
...loop body...
end do
!$acc end parallel loop
```
Parallel Loop Example

```c
!$acc parallel loop
do i=1, N
   output(i)=2.*input(i)
end do
!$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, worker, vector
  – Targets specific loop at specific level of hardware
    – gang↔CUDA threadblock (scheduled on a single SM)
    – worker ↔ CUDA warp of threads (32 on Fermi)
    – Vector ↔ CUDA threads in warp executing in SIMT lockstep
  – You can specify more than one
    !$acc loop gang worker vector schedules loop over all hardware
Tuning Clauses for parallel construct

To be added to parallel (or parallel loop) directives

- **num_gangs, num_workers, vector_length**
  - Tunes the amount of parallelism used
  - equivalent to setting the number of threads per block, number of blocks in CUDA

- **Easiest way to set the number of threads per block is to specify vector_length(NTHREADS) clause**
  - NTHREADS must be one of: 1, 64, 128 (default), 256, 512, 1024
  - NTHREADS > 32 automatically decomposed into warps of length 32
  - Don't need to specify number of threadblocks

- **E.g. to specify 128 threads per block**

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

Further clauses to \texttt{loop} (or \texttt{parallel loop}) directives

\texttt{seq}: loop executed sequentially

\texttt{independent}: compiler hint

\texttt{if(logical)} \quad \text{Executes on GPU if .TRUE. at runtime, otherwise on CPU}

\texttt{reduction}: as in OpenMP

\texttt{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.

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

```c
#pragma acc data
{
...code region...
}
```

Fortran:

```fortran
!$acc data
...code region...
!$acc end data
```

Accelerator Data Construct

```fortran
!$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

```fortran
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
```

```fortran
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.
• !$acc update [host|device]  
  – Copy specified arrays (slices) within data region  
    – Useful if you only need to send a small subset of data to/from GPU  
    – e.g. halo exchange for domain-decomposed parallel code  
    – or sending a few array elements to the CPU for printing/debugging
Kernels directive

• The `kernels` directive exists as an alternative to the `parallel` directive.

• `kernels` has emerged from the PGI model, while `parallel` has emerged from the Cray model
  – Both in standard as a compromise

• `parallel` and `kernels` regions look very similar
  – Both define a region to be accelerated
  – Different levels of obligation for the compiler
Kernels directive

- **parallel**
  - Prescriptive (like OpenMP programming model)
  - Uses a single accelerator kernel to accelerate region
  - Compiler will accelerate region even if this leads to incorrect results

- **kernels**
  - Descriptive (like PGI Accelerator programming model)
  - Uses one or more accelerator kernels to accelerate region
  - Compiler may accelerate region
  - May not if not sure that loop iterations are independent

- **Within a kernels region**, the compiler will automatically try to offload all it can to the GPU (even if there are no loop directives).
• 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 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
- Not GPU specific
  - suitable for Xeon Phi or DSPs, for example
- Fully integrated into the rest of OpenMP
• Host-centric model with one host device and multiple target devices of the same type.

• **device**: a logical execution engine with local storage.

• **device data environment**: a data environment associated with a target data or target region.

• **target** constructs control how data and code is offloaded to a device.

• Data is mapped from a host data environment to a device data environment.
**target regions**

- Code inside **target** region is executed on the device.
- Executes sequentially by default.
- Can include other OpenMP directives to make it run in parallel.
- Clauses to control data movement.
- Can specify which device to use.

```c
#pragma omp target map(to:B,C), map(tofrom:sum)
#pragma omp parallel for reduction(+:sum)
for (int i=0; i<N; i++){
    sum += B[i] + C[i];
}
```
• **target data** construct just moves data and does not execute code (c.f. `#pragma acc data` in OpenACC).
  – can have multiple target regions inside a target data region
  – allows data to persist on device between target regions

• **target update** construct updates data during a target data region.

• **declare target** compiles a version of function/subroutine that can be called on the device.

• Target regions are blocking: the encountering thread waits for them to complete.
  – Asynchronous behaviour can be achieved by using target regions inside tasks (with dependencies if required).
What about GPUs?

• Executing a target region on a GPU can only use one multiprocessor
  – synchronisation required for OpenMP not possible between multiprocessors
  – not much use!

• **teams** construct creates multiple master threads which can execute in parallel, and spawn parallel regions, but cannot synchronise or communicate with each other.

• **distribute** construct spreads the iterations of a parallel loop across teams.
  – only schedule option is static (with optional chunksize).
#pragma omp target teams distribute parallel for\  
map(to:B,C), map(tofrom:sum) reduction(+:sum)

for (int i=0; i<N; i++){
    sum += B[i] + C[i];
}

- Executes the loop on the device and distributes iterations across multiprocessors and across threads within each multiprocessor.
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