7th International Workshop on OpenMP
Chicago, Illinois, USA

James C. Beyer, Eric J. Stotzer, Alistair Hart, and Bronis R. de Supinski

OPENMP FOR ACCELERATORS
Accelerator programming

- Why a new model? There are already many ways to program:
  - CUDA, PGI CUDA Fortran, OpenCL...
  - All are quite low-level and closely coupled to the GPU
- User needs to write specialist kernels:
  - Hard to write and debug
  - Hard to optimise for specific GPU
  - Hard to update (porting/functionality)
- Directives provide high-level approach
  - Based on original source code
    - Easier to maintain/port/extend code (especially if original developer has moved on)
    - Users with OpenMP experience find it a familiar programming model
    - You run the same code on multi-core CPU
  - Possible performance sacrifice
    - A small performance gap is acceptable (do you still hand-code in assembler?)
    - Goal is within 10% of CUDA (already seeing this in many cases, more tuning ongoing)
Proposed changes
Changes to Memory and Execution Models

- Add the concept of an accelerator region to the execution model
- Accelerator task tied to accelerator it starts on
- Complete at known locations, barriers, “acc_sync”, program exit
- Data motion directives provide hints to the compiler on where to place data that accelerator regions access
Directives

- acc_region – execution
- acc_data – data motion
- acc_loop – nestible
- acc_region_loop
- acc_call – make a call
  - acc_call_declaration – change all call sites
  - acc_call_definition – change target architecture
- acc_update – move data inside data region
- Array shaping
Directive clauses

- **Data clauses:**
  - acc_copy, acc_copyin, acc_copyout, acc_shared
    - e.g. copy moves data "in" to GPU at start of region and "out" to CPU at end
    - supply list of arrays or array sections (using Fortran ":" notation)
  - present: share GPU-resident data between kernels

- **Tuning clauses:**
  - num_pes, cache, collapse...
    - optimise GPU occupancy, register and shared memory usage, loop scheduling...

- **Some other important clauses/directives:**
  - async: Launch accelerator region asynchronously
    - allows overlap of GPU computation/PCI transfers with CPU computation/network
  - acc_call: Call external libraries or CUDA kernels
    - optionally using data already resident on the GPU
  - hetero: split loop iterations between CPU and GPU
Accelerator Region Construct

Syntax:

**C/C++**
#pragma omp acc region [clause[[], clause]...] newline
  Structured-block

**Fortran**
!$omp acc region [clause[], clause]...
  Structured-block
!$omp end acc region

Clauses:

device(integer-expression [,integer-expression])
if (scalar-expression) or if (scalar-logical-expression)
num pes(depth:number [, depth:number] )
acc shared(list)
acc copy(list), acc copyin(list), acc copyout(list)
host shared(list)
firstprivate(list)
private(list)
present(list*)
default(acc shared|acc copy|firstprivate|private|none|ignore)
Accelerator Data Region Construct

Syntax:

**C/C++**

```c
#pragma omp acc data [clause [, clause]...] newline
   Structured-block
```

**Fortran**

```fortran
!$omp acc data [clause[[, clause]...] ]
   Structured-block
!$omp end acc data
```

**Clauses:**

- `device( integer-expression [, integer-expression])`
- `acc copy(list), acc copyin(list), acc copyout(list)`
- `host shared(list)`
- `acc shared(list)`
- `present(list|*)`
- `default(acc shared|host shared|acc copy|none|ignore)`
Accelerator Loop Construct

Syntax:

**C/C++**
```
#pragma omp acc loop [clause[, clause]]... new-line
```

**Fortran**
```
!$omp acc loop [clause[, clause]]...
```

Do-construct
```
!$omp end acc loop
```

**Clauses**

- `cache(obj[:depth], obj[:depth]...)`
- `collapse(n)`
- `hetero( 1 expr, width ) *`
- `host( expr ) *`
- `kernel or innermost`
- `level(dimension )`
- `max_par_level( expr )`
- `num_pes(depth:num [, depth:num] )`
- `reduction(operator:list)`
- `schedule( schedule_name )`
- `seq[(width)]`
- `stripmine( width, list )`
- `vector`
Accelerator Region Loop Construct

Syntax:

C/C++
#pragma omp acc region loop [clause[, clause]...] new-line
   For-loop-nest

Fortran
!$omp acc region loop [clause[, clause]]
   Do-loop-nest
!$omp end acc region loop

Clauses

See component constructs.
Accelerator Call Construct

**Syntax:**

**C/C++**

```c
#pragma omp acc call [clause[[,]clause]...] newline
  Function call expression
```

**Fortran**

```fortran
!$omp acc call [clause[[,] clause]...] ...
  Call statement
!$omp end acc call
```

**Clauses**

- `device(integer-expression)`
- `if (scalar-expression)`
- `implements(device:name [, device:name])`
- `num pes(depth:num [, depth:num] )`
- `acc copy(list), acc copyin(list), acc copyout(list)`
- `present(list[]*)`
Accelerator Update Directive

Syntax:

\textit{C/C++}

\#pragma omp acc update clause[, clause]... new-line

\textit{Fortran}

!$omp acc update clause[, clause]...

Clauses

\texttt{host(obj1[:obj2] [,obj1[:obj2]])}
\texttt{acc(obj1[:obj2] [,obj1[:obj2]])}
Array shaping

- **Fortran**
  - Fortran array syntax can be used to define the array section.
  - The placement of Explicit, Assumed and Deferred shape array types may be modified with the array section construct.
  - CRI pointers inherit the shape of the pointee

- **C/C++:**
  - We provide an extended array shaping syntax for C and C++.
  - Shaping operator ::= [<lower bound> : <length> : <stride>]
    - where <lower bound>, <length>, and <stride> are integer expressions that represent the integer values:
      - <lower bound>, ..., <lower bound> + (<length> - 1) * <stride>
  - Successive operators designate a sub-array of a multidimensional array object.
  - Based on Intel shaping syntax
  - <stride> defaults to 1
  - If the <length> is less than 1, the array section is undefined
  - use [:] as a short hand for a whole array dimension if the size of the dimension is known

- \( \text{int **A;} \)
- \( \text{A[:n-1][:n-1]} \)
- \( \text{A[0:n-1:2][0:n-1:2]} \)
Examples and results
Matrix multiply

Version 1
!$omp acc_region_loop
do j = 1,L
do i = 1,N
do k = 1,M
   C(i,j) = C(i,j) + A(i,k)*B(k,j)
enddo
enddo
enddo
!$omp end acc_region_loop

Version 2
!$omp acc_region_loop acc copyin(a,b) acc copy(c)
do j = 1,L
do i = 1,N
do k = 1,M
   C(i,j) = C(i,j) + A(i,k)*B(k,j)
enddo
enddo
enddo
!$omp end acc_region_loop

Version 3
!$omp acc_region_loop acc copyin(a,b) acc copy(c) present(a, b, c)
do j = 1,L
do i = 1,N
do k = 1,M
   C(i,j) = C(i,j) + A(i,k)*B(k,j)
enddo
enddo
enddo
!$omp end acc_region_loop
Data regions to hold data on GPU

- data region spans two accelerator regions
  - The acc_region checks at runtime if b is already on GPU:
    - yes: it uses this without copies; no: it follows the acc_copy(b) clause
  - Can also call double_me() from outside a data region
- Do not need to inline the subroutine (manually or by compiler)
  - Can even be in different source file
Scalar examples: benchmarks

- NAS Parallel Benchmarks and SPEC suite

- MG (multigrid) solves Laplacian on 3D grid
  - Hotspots: resid (50% of runtime); psinv (25%); rprj3 (9%)
    - Data arrays passed to/from subroutines at every iteration
  - Whole application ported (25 directive pairs for 1500 lines)
    - present clause essential to eliminate data movement costs
  - GPU 50% faster than 12-core AMD Magny-Cours CPU
    - Even before compiler starts to use GPU shared memory

- CG (conjugate gradient)
  - whole application ported (19 directive pairs for 1200 lines)
    - less than 1 hour's work (from first sight of code)
  - GPU 15% faster than 12-core AMD Magny-Cours CPU
    - more tuning is possible
Real kernel example: S3D turbulent combustion

- 3d simulation of HCCI combustion
- detailed chemical kinetics (60 species)
- very important for low emission engines burning second-generation biofuels

![Graph showing wallclock time vs. cores on host for different acceleration methods.]

- **OpenMP**
- **Accelerator using CCE**
- **Cuda Fortran**
Tuning the directive performance: comp_heat

- Original code: 3d loop nest over sites plus loop over 3 directions
  - CPU time (12 cores of MC12): 0.050s
  - hand-coded CUDA Fortran kernel time: 0.00843s
- First add 2 directives: acc_region and acc_loop
  - GPU kernel time: 0.046s
- Add collapse(2) clause to acc_loop
  - GPU kernel time: 0.017s
- Add num_pes(2:512) clause to acc_region and collapse(3)
  - GPU kernel time: 0.010s
- Some final loop restructuring (mostly automatic)
  - (unroll direction loop, interchange and fuse loops)
  - GPU kernel time: 0.00826s
Conclusions

- Motivation
- Model
- Directives
- examples