OpenACC Updates

Committee Meeting
Feb 19-21

Charlene Yang
Application Performance Specialist
cjyang@lbl.gov
OpenACC vs. OpenMP

• Aims to build a ‘leaner’ set of directives
  • targeting scalable parallelism, not general parallelism
  • e.g. no tasking, less synchronization primitives

• Descriptive vs. Prescriptive
  • lets compilers figure out how to move data/parallelize compute
  • less directed by the programmer
  • hence more performance portable

• More mature for accelerators whereas OpenMP more mature for multi-cores
  • can work together though
  • e.g. OpenACC inside OpenMP

• At the end of the day, the method of parallelizing is the most valuable!
## OpenACC vs. OpenMP

<table>
<thead>
<tr>
<th>OpenACC</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Focused on accelerated computing</td>
<td>• General purpose parallelism</td>
</tr>
<tr>
<td>• More agile</td>
<td>• More measured</td>
</tr>
<tr>
<td>• Performance portability</td>
<td>• Performance portability a challenge</td>
</tr>
<tr>
<td>• Descriptive</td>
<td>• Prescriptive</td>
</tr>
<tr>
<td>• Extensive interoperability</td>
<td>• Limited interoperability</td>
</tr>
<tr>
<td>• More mature for accelerators</td>
<td>• More mature for multi-core</td>
</tr>
</tbody>
</table>

* Michael Wolfe, Duncan Poole  
  https://www.nextplatform.com/2015/11/30/is-openacc-the-best-thing-to-happen-to-openmp/*
Face-to-Face Meeting

- Feedback from previous hackathons
  - OLCF GPU Hackathons
  - OpenACC Hackathons

- Issues from previous discussions or GitHub OpenACC/openacc-spec/Issues
  - Deep copy
  - Multiple devices
  - Task graphs
  - Optimization directives
  - C++ Lambdas
  - Aliasing on data clauses, #14
  - Reductions, #148, #157
  - requires directive
  - Cleaning up C/C++/Fortran pointers
  - Error handler
  - Memory Allocation
  - New C/C++/Fortran language features

- Prioritizing/Assigning open issues
Deep Copy

- Nested dynamic data structures
- e.g. ICON, climate code from CSCS, Fortran, four levels of derived structured arrays

```fortran
 type t_nh_state

    !array of prognostic states at different timelevels
 type(t_nh_prog), allocatable :: prog(:)  !< shape: (timelevels)
 type(t_var_list), allocatable :: prog_list(:)  !< shape: (timelevels)

 type(t_nh_diag) :: diag
 type(t_var_list) :: diag_list

 type(t_nh_ref) :: ref
 type(t_var_list) :: ref_list

 type(t_nh_metrics) :: metrics
 type(t_var_list) :: metrics_list

 type(t_var_list), allocatable :: tracer_list(:)  !< shape: (timelevels)

end type t_nh_state

 type(t_nh_state), allocatable :: p_nh_state(:)
```

diag and metrics both have 80 allocatable:pointer array members
Deep Copy

A motivating example:

```c
struct deep_type {
    int n;
    float* a;
    float* b;
    float* c;
};
depth_type X;

// Performs shallow copy of X
#pragma acc data copy(X)
```

(a) Shallow copy

(b) Deep copy
Manual deep copy:
- attach/detach pointers, multi-level pointers

```c
struct deep_type {
    int n;
    float* a;
    float* b;
    float* c;
};
dee_type X;

// Performs copy of X, X.a, X.b, X.c and attach a, b, c to parent pointer X (top-down copy)
#pragma acc data copy(X)
#pragma acc data copy(X.a[0:n],X.b[0:n],X.c[0:n])
```
Deep Copy

True deep copy:

- **shape** allows defining the size of global deep-copy behavior
- **policy** enables defining selective direction behavior of deep-copy

```c
struct deep_type {
    int n;
    float* a;
    float* b;
    float* c;

    // This default shape includes deep copy of members a, b, and c, and
    // it ensures member n is always initialized
    #pragma acc shape init_needed(n) include(a[0:n],b[0:n],c[0:n])
};

deep_type X;
// Performs deep copy of X
#pragma acc data copy(X)
```
Deep Copy

True deep copy: shape syntax

```c
struct deep_type {
    int n;
    float* a;
    float* b;
    float* c;

    // This default shape includes deep copy of members a, b, and c, and
    // it ensures member n is always initialized
    #pragma acc shape init_needed(n) include(a[0:n],b[0:n],c[0:n])
};

depth_type* Y;
int size;

// Performs a deep copy of Y; note that member n can be different for each element of Y
#pragma acc data copy(Y[0:size])
```
Deep Copy

True deep copy: two layers

template <Type T>
class vector {
    T* base;
    T* end;
    #pragma acc shape include(base[0:size()], end[@base])
};

class Data {
    vector <float> d1;
    vector <float> d2;
};

Data d;

// This directive performs full deep-copy, since shape is default(include) and each member has a default shape
#pragma data copy(d)
Deep Copy

True deep copy: policy syntax

```c
struct deep_type {
    int n;
    float* a;
    float* b;
    float* c;

    #pragma acc shape init_needed(n) include(a[0:n],b[0:n],c[0:n])
    // Policy to copyin members b and c and copyout member a (which might be used for a computation like a = b + c)
    #pragma acc policy(calc_a) default(copyin) copyout(a)
};

deep_type X;

// Performs selective directional deep copy of X
#pragma acc data invoke<calc_a>(X)
```
• Syntax is still in discussion

• Details are at

• May make it to OpenACC 3.0, releasing in Nov 2019.
Multiple Devices

- Currently, the OpenACC execution model is one device at a time
- To support multiple devices, we need to think about expanding the execution model
  - today, OMP/MPI outer, then single device programming within OMP/MPI thread/rank
- One growth area is multiple-device fat workstations/nodes
  - want to be able to control multiple GPUs all within OpenACC
- Two bits of low-hanging fruit when there’s only one host thread/rank
  - copying directly between different devices
  - synchronization across device queues
Multiple Devices

- Copying directly between different devices
  - how to specify source and/or target device
  - do we want to support broadcast to multiple devices
  - do we want to support host as a device

```c
acc update device(a[0:n]) dstdev(1) srcdev(0)
acc update device(a[0:n]) device_num(0,1) // destination, src
acc update device(a[0:n]) device_num(from:0,to:1)
acc update device(a[0:n]) device_num(1) // no 'from' implies self
acc update device(a[0:n]) device_num(from:1) // no 'to' implies current device
acc update device(a[0:n]) device_num(0,:) // colon implies current device
acc update device(from:a[0:n],to:b[0:n]) device_num(from:0,to:1)
acc update (from:a[0:n],to:b[0:n]) device_num(from:0,toself)
acc memcpy (from:a[0:n],to:b[0:n]) device_num(from:0,toself)
acc set (from:a[0:n],to:b[0:n]) device_num(from:0,toself)
acc update (from:a[0:n],to:b[0:n]) device_num(from:0,to:1)
```
Multiple Devices

- Synchronization across device queues
  - the host waits for each device individually
  - do we want to allow waiting on more than one device

```
acc wait(1,2) device_num(0,1)
acc wait(0:1,1:2)
acc wait(0:1) async(1:2) // device_num:queuenum
acc wait(dev=0:1,dev=1:2) async(dev=2:2)
acc wait([device_num:1,queue:1], device_num:1,queue:2) async([device_num:2,queue:2])
acc wait([d:1,q:1], d:1,q:2) async([d:2,q:2])
```
Multiple Devices

• All of this is probably not a functionality issue but more of a syntax issue

• In the future,
  – support ‘any’ integer levels of parallelism
  – how to map parallelism to the fixed levels of parallelism on the device
Task Graphs

- Stephen Jones, Asynchronous Task Graphs in CUDA
- CUDA operations are submitted in streams, FIFO queues with dependences between operations
- Executional dependences and data dependences
- Easy to translate CUDA streams with dependences into a task DAG

- Graph nodes are kernels, data movement, CPU callbacks, subgraphs
- Define the CUDA graph, and launch (and relaunch) the graph very cheaply [instantiate + execute]
  - graph sequence and configurations must be invariant

- A simple example with a sequence of short OpenACC parallel loops launched many times
  - 10 iterations
  - CUDA graph took .014us, and the regular version took .410us -- 30x improvement!
Optimization Directives

• An **unroll** directive for loops?

• An IWOMP paper proposed a plethora of loop transformations for OpenMP
  - unroll
  - tile
  - interchange
  - cache-tiling / strip-mining
  - unroll-and-jam
  - fusion
  - distribute / fission
  - vectorization / simd
  - interleave
  - software pipelining
  - loop invariant code motion
  - if conversion
  - collapsing
C++ Lambdas

- Compiler generates an anonymous struct with an operator() containing the lambda body, and a struct member for each captured item, either by value or by reference (address).

- Problems
  - unnamed struct does not get copied to the device as there is no named symbol for it
  - operator() function has no 'acc routine' information
  - how to attach pointer members

- Solutions
  - for named lambdas, let user specify 'acc routine' above the lambda declaration
  - for unnamed lambdas, let compiler inject 'acc routine seq'?
  - deep copy lambda members
    - copyin(lambda_struct), copyin(reference members), no_create/attach(pointer_members)
Reference

• All notes are available here
  – https://github.com/OpenACC/openacc-spec/wiki/Notes

• Kyle Friedline (Udel)’s links for compiler comparisons
  – OpenACC stuff:
    – https://crpl.cis.udel.edu/blog/2018/07/15/openaccvv/
    – https://www.researchgate.net/publication/318445660_OpenACC_25_Validation_Testsuite_Targeting_Multiple_Architectures
  – OpenMP stuff:
    – https://crpl.cis.udel.edu/ompvvsollve/results/
Thank You