Comparing OpenACC 2.5 and OpenMP 4.1

James C Beyer PhD, Sept 29th 2015
Abstract

As both an OpenMP and OpenACC insider I will present my opinion of the current status of these two directive sets for programming “accelerators”. Insights into why some decisions were made in initial releases and why they were changed later will be used to explain the trade-offs required to achieve agreement on these complex directive sets.
Why a talk about OpenMP vs OpenACC?

These are the two dominate directive based methods for programming “accelerators”

They are relatives!

Why not, it seems like a nice non-controversial topic
Agenda

OpenMP status
OpenACC status
Major differences
Loops
Status - OpenMP 4.1

New or changed functionality

- use-device-ptr
- is-device-ptr
- nowait
- depend
- (first)private
- defaultmap
- declare target link
- enter|exit target data
- scalar implicit data-sharing
  - firstprivate
- deeper-shallow copy
- device memory routines

Existing OpenACC 2.0 functionality

- host_data
- deviceptr
- async
- async()/wait()
- (first)private
- declare link
- enter|exit data
- scalar implicit data-sharing
- device memory routines
Status - OpenACC 2.5

New or changed functionality

- declare create on allocatable
- init, shutdown, set directives
- if_present on update
- routine bind
- openacc_lib.h deprecated
- API to get/set default async queue
- switch default present_or behavior
- default(present)
- async version of memory APIs
- acc_memcpy_device
- profiling support

OpenMP 4.1 functionality

- Update no-op if not present
- omp_target_memcpy[_rect]
- TR
Major differences

What is the fundamental difference between OpenMP and OpenACC?

1) Prescriptive vs Descriptive (or is it?)

2) Synchronization

3) Independent loop iterations
   This is an interesting and important topic

4) Expressiveness vs Performance and Portability
   Is this real or just perceived?
“**Scope**: The OpenMP API covers only user-directed parallelization, wherein the programmer **explicitly specifies the actions to be taken** by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations are **not** required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs. In addition, compliant implementations are not required to check for code sequences that cause a program to be classified as non-conforming. Application developers are responsible for correctly using the OpenMP API to produce a conforming program. The OpenMP API **does not cover compiler-generated automatic parallelization** and directives to the compiler to assist such parallelization.”

— OpenMP 4.1 Comment Draft, Page 1
“Introduction: The programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator, guidance on mapping of loops onto an accelerator, and similar performance-related details.”

“Scope: This OpenACC API document covers only user-directed accelerator programming, where the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The remainder of the program will be executed on the host. This document does not describe features or limitations of the host programming environment as a whole; it is limited to specification of loops and regions of code to be offloaded to an accelerator.”

— OpenACC 2.5 Comment Draft, page 7
So what's different between those two statements?

wherein the programmer explicitly specifies the actions to be taken by the compiler -- OpenMP

EXPLICIT

guidance on mapping of loops

IMPLICIT

user specifies the regions of a host program to be targeted for offloading -- OpenACC

EXPLICIT

So is this difference significant?

Clearly we thought it was when we formed the OpenACC technical committee
Major differences
Synchronization

- OpenMP
  - master
  - critical
  - barrier
  - taskwait
  - taskgroup
  - atomic - restrictions?
  - flush
  - ordered
  - depend

- OpenACC
  - atomic - with restrictions

  async/wait - if you squint
Present or not

OpenMP learns from OpenACC and vice versa

OpenACC 1.0 had two forms of data motion clauses, one form for testing presence and one for “skipping” the presence test.

OpenMP 4.0 had one form of data motion clause which always checks for presence

OpenACC 2.5 eliminates the form that skips the presence test - actually, it makes the “fast” path do a present test
Update directives

Present or not (take 2)

OpenACC 1.0 and OpenMP 4.0 both made it an “error” to do an “update” on an object that was not present on the device.

OpenACC 2.0 and OpenMP 4.1 both relaxed this hindrance to programmers.

OpenACC 2.0 added the if_present clause.

OpenMP 4.1 just makes the update a no-op if the object is not present.
Routines and/or calls

OpenACC 1.0 did not support calls, either they had to flatten out or they could not be present, with a few minor exceptions provided by implementations.

OpenMP 4.0 provided “target declare” which places objects and routines on the device.

The OpenMP compiler does not look for automatic parallelism thus it does routines do not need to reserve parallelism for themselves.

OpenACC 2.0 provided “routine” which caused the compiler to generate one type of subroutine for the device, either gang, worker, vector or sequential.
Default firstprivate scalars

OpenACC 1.0 made all scalars firstprivate unless overridden by the programmer.

OpenMP 4.0 made all scalars map(inout) on target constructs.

What happens if the object is already on the device?

Either the system goes to the device to determine important kernel sizing details or it has to make safe assumptions.

User: why is my code slow? Implementer: because the kernel is mis-sized.
User: How do I fix it? Implementer: you have to rewrite your code to ...
User: That’s ridiculous! Implementer: Agreed.

OpenMP 4.1 makes all scalars firstprivate by default.
Biggest mistake made in both directive sets

OpenACC 1.0 and OpenMP 4.0 both contained structured data constructs.

C++ has constructor and destructors (surprise!)

Structured data constructs do not mix well in this environment.

C and Fortran developers create their own initializers, (constructors) and finalizers, (destructors)

Both OpenACC and OpenMP learned from this mistake.
Host or device compilation

OpenMP 4.1

- Host
  - omp parallel for/do

- Device
  - omp target parallel do
  - or
  - omp target teams distribute parallel for
  - or
  - omp target teams distribute parallel for simd
  - or
  - omp target teams distribute simd
  - or
  - omp target parallel for simd
  - or
  - omp target simd

OpenACC 2.5

- Host
  - acc parallel loop

- Device
  - acc parallel loop*
OpenACC loops vs OpenMP loops
Independent loop example(s)

In C

```c
foo( int *a, int n ) {
    for ( int i = 0; i<n; i++ ) {
        a[i] = i;
    }
}
```

```c
foo2( int * a, int * b, int n ) {
    for( int i = 0; i<n; i++ ) {
        a[i] = a[i] * b[i];
    }
}
```
Independent loop example(s)

In C

```c
foo( int *a, int n ) {
    for ( int i = 0; i<n; i++) {
        a[i] = i;
    }
}
```

```c
foo2( int * restrict a, int * restrict b, int n ) {
    for( int i = 0; i<n; i++ ) {
        a[i] = a[i] * b[i];
    }
}
```
Independent loop example(s)

In Fortran

```fortran
subroutine foo( a, n )
  integer :: a(n)
  integer n,i
  do i = 1, n
    a(i) = i;
  enddo
end subroutine foo
```

```fortran
subroutine foo2( a, b, n )
  integer :: a(n), b(n)
  integer n,i
  do i = 1, n
    a(i) = a(i) * b(i);
  enddo
end subroutine foo2
```
Parallelizing Independent loops
Using OpenMP and OpenACC

```c
foo2( int * a, int * b, int n ) {
    #pragma omp parallel for
    for( int i = 0; i < n; i++ ) {
        a[i] = a[i] * b[i];
    }
}
```

```c
foo2( int * a, int * b, int n ) {
    #pragma acc parallel loop
    for( int i = 0; i < n; i++ ) {
        a[i] = a[i] * b[i];
    }
}
```
Parallelizing Independent loops

OpenMP and OpenACC (the other way)

```c
foo2( int * a, int * b, int n ) {
    #pragma omp parallel for
    for( int i = 0; i<n; i++ ) {
        a[i] = a[i] * b[i];
    }
}
```

```c
foo2( int * restrict a, int * restrict b, int n ) {
    #pragma acc kernels
    for( int i = 0; i<n; i++ ) {
        a[i] = a[i] * b[i];
    }
}
```
Parallelizing Independent loops
Nearest equivalents to OpenACC versions

```c
foo2( int * a, int * b, int n ) {
    #pragma omp target teams distribute [parallel for simd]
    for( int i = 0; i<n; i++ ) {
        a[i] = a[i] * b[i];
    }
}
```
Parallelizing Independent loops
Nearest equivalents to OpenACC versions

```c
foo2( int * a, int * b, int n ) {
#pragma omp target teams distribute [simd]
    for( int i = 0; i<n; i++ ) {
        a[i] = a[i] * b[i];
    }
}
```
Parallelizing dependent loop example

OpenMP and OpenACC

```c
foo2( int * a, int * b, int n ) {
    #pragma omp parallel for ordered
    for( int i = 0; i<n; i++ ) {
        #pragma omp ordered
        a[i] = a[i-1] * b[i];
    }
}
```

```c
foo2( int * a, int * b, int n ) {
    #pragma acc kernels
    for( int i = 0; i<n; i++ ) {
        a[i] = a[i-1] * b[i];
    }
}
```
1-D Stencil loop examined

What is the “best” solution

- !$acc parallel loop
  do k = 2, n3-1
  !$acc loop worker
    do j = 2, n2-1
      do i = 2, n1-1
        !$acc cache( u(i-1:i+1, j, k) )
        r(i,j,k) = <full stencil on u>
      enddo
    enddo
  enddo
- !$omp target teams distribute
  do k = 2, n3-1
    !$omp parallel for [simd?]
      do j = 2, n2-1
        do i = 2, n1-1
          r(i,j,k) = <full stencil on u>
        enddo
      enddo
  enddo
1-D Stencil loop examined

!$omp target teams distribute parallel for [simd] collapse(3)
  do k = 2, n3-1
    do j = 2, n2-1
      do i = 2, n1-1
        r(i,j,k) = <full stencil on u>
        enddo
      enddo
    enddo
  enddo
Sparse matrix vector multiply

```c
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
    for(int j=row_start; j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```
Sparse matrix vector multiply

OpenMP: 1

```c
#pragma omp parallel for
for(int i=0; i<num_rows; i++) {
    double sum = 0;
    int row_start = row_offsets[i];
    int row_end = row_offsets[i+1];
    for(int j=row_start; j<row_end; j++) {
        unsigned int Acol = cols[j];
        double Acoef = Acoefs[j];
        double xcoef = xcoefs[Acol];
        sum += Acoef * xcoef;
    }
    ycoefs[i] = sum;
}
```
Sparse matrix vector multiply

OpenMP: 2 (when autovectorization fails)

Which is better?

Answer depends on hardware and inputs
Sparse matrix vector multiply

OpenMP: 3 (on an “accelerator”)

```c
#pragma omp target teams distribute
for(int i=0; i<num_rows; i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
    for(int j=row_start; j<row_end; j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```
Sparse matrix vector multiply

OpenMP: 3.1 (what about the threads?)

```c
#pragma omp target teams distribute [parallel for|simd]
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
    [#pragma omp [parallel for][simd]]
    for(int j=row_start; j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```
Sparse matrix vector multiply

OpenMP: 3.2 (what about running on the host?)

```c
#pragma omp target teams distribute [parallel for | simd] if( num_rows<min_rows )
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
    [#pragma omp [parallel for][simd]]
    for(int j=row_start; j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```

Do we really want a new team?

Do we really get a new team?

What if we want nested parallelism at different levels for different targets?

Things just start looking strange.

The language committee is working on fixing some of these “issues”.
Sparse matrix vector multiply

OpenACC: 1

```c
#pragma acc parallel loop
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum) // this is usually discovered by the compiler
    for(int j=row_start; j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```
Sparse matrix vector multiply

OpenACC: 2 (when auto-vectorization fails)

```c
#pragma acc parallel loop [gang vector]
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
    [#pragma acc loop vector]
    for(int j=row_start; j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```

Which is better?

Answer depends on hardware and inputs
Sparse matrix vector multiply

OpenACC: 2.1 (when autovectorization fails)

```c
#pragma acc parallel loop gang device_type( CrayV ) vector
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
    #pragma acc loop device_type(*) vector \
    device_type( CrayV ) seq
    for(int j=row_start; j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```
Sparse matrix vector multiply

OpenACC: 2.2 (what about the host?)

```c
#pragma acc parallel loop gang if(num_rows< min_rows)
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop device_type(*) vector
for(int j=row_start; j<row_end; j++) {
    unsigned int Acol=cols[j];
    double Acoef=Acoefs[j];
    double xcoef=xcoefs[Acol];
    sum+=Acoef*xcoef;
}
} ycoefs[i]=sum;
}'''

What does this do?
Just says run the construct on the host

What does this do?
Not well defined, spec says nothing about whether or not this affects the host
“First as a member of the Cray technical staff and now as a member of the NVIDIA technical staff, I am working to ensure that OpenMP and OpenACC move towards parity whenever possible!”

James Beyer, Co-chair OpenMP accelerator sub-committee and OpenACC technical committee
• Common questions people ask about this topic
  • Why did OpenACC split off from OpenMP?
    • The final decision came down to time to release
  • When will the two specs merge?
    • 4.1 shows significant effort went into merging the specs
  • Which is better?
    • I cannot answer this question for you, it depends on what you are trying to do,
    • I will say that OpenMP has more compilers that support it at this time
  • When or will <insert company name here> support OpenACC X or OpenMP 4.1?
    • I have no idea, ...
SIMD versus Vector

Why does OpenMP say SIMD while OpenACC says Vector?

1) SIMD has been coopted to mean one type of implementation
2) Vector is a synonym for SIMD without the implying an implementation

Can vector loops contain dependencies within the vector length?

- Only if the hardware supports forwarding results to “future” elements
- SIMT machines can do this, relatively easily
- !$omp simd on a dependent loop may give “interesting” results
- !$acc vector on a dependent loop should give the “correct” result
Independent loop iterations

Independent loop iterations can be run in any order

Dependent loops require some form a synchronization and potentially order guarantees

Question:

Can all algorithms be written to use independent loop iterations?

Can NVIDIA gpus handle dependent loop iterations? To a limited extent yes

Can Intel Xeon PHI processors handle dependent loop iterations? Yes

Can processor X handle dependent loop iterations?