GPU Computing with OpenACC Directives
Presented by Robert Strzodka

Authored by Mark Harris
NVIDIA Corporation
GPUs Reaching Broader Set of Developers

1,000,000’s

100,000’s

Early Adopters

- Universities
- Supercomputing Centers
- Oil & Gas
- Research

CAE
CFD
Finance
Rendering
Data Analytics
Life Sciences
Defense
Weather
Climate
Plasma Physics

2004

Present
3 Ways to Accelerate Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
OpenACC Directives

Program myscience
... serial code ...

!$acc kernels
  do k = 1, n1
  do i = 1, n2
    ... parallel code ...
  enddo
  enddo

!$acc end kernels
... 
End Program myscience

Your original Fortran or C code

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

CPU

GPU

OpenACC Compiler Hint
Familiar to OpenMP Programmers

OpenMP

```c
main() {
    double pi = 0.0; long i;
    #pragma omp parallel for reduction(+:pi)
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n", pi/N);
}
```

OpenACC

```c
main() {
    double pi = 0.0; long i;
    #pragma acc kernels
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n", pi/N);
}
```
OpenACC
Open Programming Standard for Parallel Computing

“OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan.”

--Buddy Bland, Titan Project Director, Oak Ridge National Lab

“OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP.”

--Michael Wong, CEO OpenMP Directives Board
OpenACC
The Standard for GPU Directives

**Easy:** Directives are the easy path to accelerate compute intensive applications

**Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

**Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU
High-level, with low-level access

- Compiler directives to specify parallel regions in C, C++, Fortran
  - OpenACC compilers offload parallel regions from host to accelerator
  - Portable across OSes, host CPUs, accelerators, and compilers
- Create high-level heterogeneous programs
  - Without explicit accelerator initialization,
  - Without explicit data or program transfers between host and accelerator
- Programming model allows programmers to start simple
  - Enhance with additional guidance for compiler on loop mappings, data location, and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.
Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications.

-- Developer at the Global Manufacturer of Navigation Systems
Small Effort. Real Impact.

Large Oil Company
3x in 7 days
Solving billions of equations iteratively for oil production at world’s largest petroleum reservoirs

Univ. of Houston
Prof. M.A. Kayali
20x in 2 days
Studying magnetic systems for innovations in magnetic storage media and memory, field sensors, and biomagnetism

Uni. Of Melbourne
Prof. Kerry Black
65x in 2 days
Better understand complex reasons by lifecycles of snapper fish in Port Phillip Bay

Ufa State Aviation
Prof. Arthur Yuldashev
7x in 4 Weeks
Generating stochastic geological models of oilfield reservoirs with borehole data

GAMESS-UK
Dr. Wilkinson, Prof. Naidoo
10x
Used for various fields such as investigating biofuel production and molecular sensors.

* Achieved using the PGI Accelerator Compiler
Focus on Exposing Parallelism

With Directives, tuning work focuses on exposing parallelism, which makes codes inherently better.

Example: Application tuning work using directives for new Titan system at ORNL

**S3D**
Research more efficient combustion with next-generation fuels

- Tuning top 3 kernels (90% of runtime)
- 3 to 6x faster on CPU+GPU vs. CPU+CPU
- But also improved all-CPU version by 50%

**CAM-SE**
Answer questions about specific climate change adaptation and mitigation scenarios

- Tuning top key kernel (50% of runtime)
- 6.5x faster on CPU+GPU vs. CPU+CPU
- Improved performance of CPU version by 100%
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online
  
  http://www.openacc-standard.org

- Quick reference card also available

- Beta implementations available now from PGI, Cray, and CAPS
Start Now with OpenACC Directives

Sign up for a free trial of the directives compiler now!

Free trial license to PGI Accelerator
Tools for quick ramp

www.nvidia.com/gpudirectives

Accelerate Your Scientific Code with OpenACC
The Open Standard for GPU Accelerator Directives

Thousands of cores working for you.

Based on the OpenACC standard, GPU directives are the easy, proven way to accelerate your scientific or industrial code. With GPU directives, you can accelerate your code by simply inserting compiler hints into your code and the compiler will automatically map compute-intensive portions of your code to the GPU. Here’s an example of how easy a single directive hint can accelerate the calculation of pi. With GPU directives, you can get started and see results in the same afternoon.

```c
#include <stdio.h>

#define PI 3.14159

int main() { 
    double pi = PI; 
    int i;
    for (i = 0; i < 100000000; i++) { 
        pi = pi/2 + PI/2; 
    }
    printf("%f", pi/PI); 
    return 0;
}
```

By starting with a free, 30-day trial of PGI Directives today, you are working on the technology that is the foundation of the OpenACC directives standard. OpenACC is:
A Very Simple Exercise: SAXPY

SAXPY in C

```c
void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...```

SAXPY in Fortran

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    !$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$acc end kernels
end subroutine saxpy
...
$ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...```
Directive Syntax

- Fortran
  
  ```fortran
  !$acc directive [clause [,] clause] …
  Often paired with a matching end directive surrounding a structured code block
  !$acc end directive
  ```

- C
  
  ```c
  #pragma acc directive [clause [,] clause] …
  Often followed by a structured code block
  ```
Kernels: Your first OpenACC Directive

Each loop executed as a separate *kernel* on the GPU.

```!
$acc kernels
   do i=1,n
      a(i) = 0.0
      b(i) = 1.0
      c(i) = 2.0
   end do

   do i=1,n
      a(i) = b(i) + c(i)
   end do
$acc end kernels!
```

**Kernel:**
A parallel function that runs on the GPU
Kernels Construct

**Fortran**

```fortran
!$acc kernels [clause ...]
  structured block
!$acc end kernels
```

**C**

```c
#pragma acc kernels [clause ...]
  { structured block }
```

**Clauses**

- `if( condition )`
- `async( expression )`

Also, any data clause (more later)
C tip: the restrict keyword

- Declaration of intent given by the programmer to the compiler
  Applied to a pointer, e.g.
  ```c
  float *restrict ptr
  ```
  Meaning: “for the lifetime of ptr, only it or a value directly derived from it (such as ptr + 1) will be used to access the object to which it points”*

- Limits the effects of pointer aliasing
- OpenACC compilers often require restrict to determine independence
  - Otherwise the compiler can’t parallelize loops that access ptr
  - Note: if programmer violates the declaration, behavior is undefined

Trivial first example
- Apply a loop directive
- Learn compiler commands

```c
#include <stdlib.h>

void saxpy(int n, float a, float *x, float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}

int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats
    if (argc > 1)
        N = atoi(argv[1]);

    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));

    for (int i = 0; i < N; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(N, 3.0f, x, y);
    return 0;
}
```

*restrict:*
“\textit{I promise y does not alias x}”
Compile and run

C:

```c
pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c
```

Fortran:

```fortran
pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90
```

Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
  8, Generating copyin(x[:n-1])
  Generating copy(y[:n-1])
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary
  9, Loop is parallelizable
  Accelerator kernel generated
  9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
  CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
  CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```
Example: Jacobi Iteration

Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.

- Common, useful algorithm
- Example: Solve Laplace equation in 2D: \( \nabla^2 f(x, y) = 0 \)

\[
A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}
\]
while ( error > tol && iter < iter_max ) {
    error=0.0;
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
while (error > tol && iter < iter_max) {
    error = 0.0;

#pragma omp parallel for shared(m, n, Anew, A)
    for(int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for shared(m, n, Anew, A)
    for(int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels
  for( int j = 1; j < n-1; j++ ) {
    for( int i = 1; i < m-1; i++ ) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc kernels
  for( int j = 1; j < n-1; j++ ) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
Jacobi Iteration Fortran Code

do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

do j=1,m
  do i=1,n
    Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                        A(i   , j-1) + A(i   , j+1))
    err = max(err, Anew(i,j) - A(i,j))
  end do
end do

do j=1,m-2
  do i=1,n-2
    A(i,j) = Anew(i,j)
  end do
end do

iter = iter +1
end do
OpenMP Fortran Code

```fortran
!$omp parallel do shared(m,n,Anew,A) reduction(max:err)
  do j=1,m
    do i=1,n
      Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                 A(i   , j-1) + A(i   , j+1))
      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do

!$omp parallel do shared(m,n,Anew,A)
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do

iter = iter +1
end do
```
OpenACC Fortran

do while ( err > tol .and. iter < iter_max )
  err = 0._fp_kind

!$acc kernels
  do j=1,m
    do i=1,n
      Anew(i,j) = .25_fp_kind * (A(i+1, j) + A(i-1, j) + &
                                A(i, j-1) + A(i, j+1))
      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$acc end kernels

!$acc kernels
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end kernels

iter = iter + 1
## Performance

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz

GPU: NVIDIA Tesla M2070

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>162.16</td>
<td>0.24x FAIL</td>
</tr>
</tbody>
</table>
What went wrong?

Add `-ta=nvidia,time` to compiler command line

Accelerator Kernel Timing data
/usr/users/6/harrism/openacc-workshop/solutions/001-laplace2D-kernels/laplace2d.c

69: region entered 1000 times
   time(us): total=77524918 init=240 region=77524678
   kernels=4422961 data=66464916
   w/o init: total=77524678 max=83398 min=72025 avg=77524
   72: kernel launched 1000 times
      grid: [256x256] block: [16x16]
      time(us): total=4422961 max=4543 min=4345 avg=4422

60: kernel launched 1000 times
   grid: [256x256] block: [16x16]
   time(us): total=8201000 max=8297 min=8187 avg=8201
   64: kernel launched 1000 times
      grid: [1] block: [256]
      time(us): total=145306 max=242 min=143 avg=145

Huge Data Transfer Bottleneck!
Computation: 12.7 seconds
Data movement: 133.3 seconds
For efficiency, decouple data movement and compute off-load
Excessive Data Transfers

while ( error > tol && iter < iter_max ) {
    error=0.0;
    #pragma acc kernels
    for (int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
}

A, Anew resident on host

These copies happen every iteration of the outer while loop!*

A, Anew resident on host

A, Anew resident on accelerator

A, Anew resident on accelerator

A, Anew resident on host

Copy

Copy

...
DATA MANAGEMENT
Data Construct

Fortran

```fortran
!$acc data [clause ...]
   structured block
!$acc end data
```

C

```c
#pragma acc data [clause ...]
   { structured block }
```

General Clauses

```c
if( condition )
async( expression )
```

Manage data movement. Data regions may be nested.
Data Clauses

**copy ( list )**  Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

**copyin ( list )**  Allocates memory on GPU and copies data from host to GPU when entering region.

**copyout ( list )**  Allocates memory on GPU and copies data to the host when exiting region.

**create ( list )**  Allocates memory on GPU but does not copy.

**present ( list )**  Data is already present on GPU from another containing data region.

and **present_or_copy[in|out]**, **present_or_create**, **deviceptr**.
Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”

- C
  ```
  #pragma acc data copyin(a[0:size-1]), copyout(b[s/4:3*s/4])
  ```

- Fortran
  ```
  !$pragma acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
  ```

- Note: data clauses can be used on data, kernels or parallel
Update Construct

Fortran

!$acc update [clause ...]

Clauses

  host( list )
  device( list )

C

#pragma acc update [clause ...]

if( expression )
async( expression )

Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

Move data from GPU to host, or host to GPU. Data movement can be conditional, and asynchronous.
while ( error > tol && iter < iter_max ) {
    error=0.0;

    #pragma acc kernels
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {


            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc kernels
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
OpenACC C with Data Handling

```c
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.0;

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i]);
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator
OpenACC Fortran without Data Handling

do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels
  do j=1,m
    do i=1,n

      Anew(i,j) = 0.25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
          A(i , j-1) + A(i , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$acc end kernels

!$acc kernels
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end kernels

iter = iter +1
end do
OpenACC Fortran with Data Handling

```fortran
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
    err=0._fp_kind

!$acc kernels
    do j=1,m
        do i=1,n

            Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                      A(i , j-1) + A(i , j+1))

            err = max(err, Anew(i,j) - A(i,j))
        end do
    end do
!$acc end kernels

...  

iter = iter +1
end do
!$acc end data
```
# Performance with Data Handling

**CPU:** Intel Xeon X5680 6 Cores @ 3.33GHz

**GPU:** NVIDIA Tesla M2070

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>13.65</td>
<td>2.9x</td>
</tr>
</tbody>
</table>

Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU
Further speedups

- OpenACC gives us more detailed control over parallelization
  - Via gang, worker, and vector clauses

- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code

- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance

- This is covered in advanced OpenACC courses
Finding Parallelism in your code

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable.
Tips and Tricks

- (PGI) Use time option to learn where time is being spent
  - `ta=nvidia,time`

- Eliminate pointer arithmetic

- Inline function calls in directives regions
  - (PGI): `-inline` or `-inline,levels(<N>)`

- Use contiguous memory for multi-dimensional arrays

- Use data regions to avoid excessive memory transfers

- Conditional compilation with `_OPENACC` macro
OpenACC Learning Resources

- OpenACC info, specification, FAQ, samples, and more
  - [http://openacc.org](http://openacc.org)
- PGI OpenACC resources
  - [http://www.pgroup.com/resources/accel.htm](http://www.pgroup.com/resources/accel.htm)
COMPLETE OpenACC API
Directive Syntax

- **Fortran**
  ```fortran
  !$acc directive [clause [,] clause] ...]
  Often paired with a matching end directive surrounding a structured code block
  !$acc end directive
  ```

- **C**
  ```c
  #pragma acc directive [clause [,] clause] ...]
  Often followed by a structured code block
  ```
Kernels Construct

Fortran

!$acc kernels [clause ...]
   structured block
!$acc end kernels

C

#pragma acc kernels [clause ...]
   { structured block }

Clauses

if( condition )
async( expression )

Also any data clause
Kernels Construct

Each loop executed as a separate kernel on the GPU.

```plaintext
 !$acc kernels 
    do i=1,n
        a(i) = 0.0
        b(i) = 1.0
        c(i) = 2.0
    end do 

    do i=1,n
        a(i) = b(i) + c(i)
    end do 

!$acc end kernels
```

kernel 1

kernel 2
Parallel Construct

Fortran

```fortran
!$acc parallel [clause ...]
  structured block
!$acc end parallel
```

Clauses

- `if(condition)`
- `async(expression)`
- `num_gangs(expression)`
- `num_workers(expression)`
- `vector_length(expression)`

C

```c
#pragma acc parallel [clause ...]
  { structured block }
```

```c
private(list)
firstprivate(list)
reduction(operator:list)
```

Also any data clause
## Parallel Clauses

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_gangs (expression)</td>
<td>Controls how many parallel gangs are created (CUDA gridDim).</td>
</tr>
<tr>
<td>num_workers (expression)</td>
<td>Controls how many workers are created in each gang (CUDA blockDim).</td>
</tr>
<tr>
<td>vector_length (list)</td>
<td>Controls vector length of each worker (SIMD execution).</td>
</tr>
<tr>
<td>private (list)</td>
<td>A copy of each variable in list is allocated to each gang.</td>
</tr>
<tr>
<td>firstprivate (list)</td>
<td>private variables initialized from host.</td>
</tr>
<tr>
<td>reduction (operator:list)</td>
<td>private variables combined across gangs.</td>
</tr>
</tbody>
</table>
Loop Construct

Fortran

```fortran
!$acc loop [clause ...]
    loop
!$acc end loop
```

C

```c
#pragma acc loop [clause ...]
    { loop }
```

Combined directives

```fortran
!$acc parallel loop [clause ...]
!$acc kernels loop [clause ...]
```

Detailed control of the parallel execution of the following loop.
<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>collapse(n)</code></td>
<td>Applies directive to the following <code>n</code> nested loops.</td>
</tr>
<tr>
<td><code>seq</code></td>
<td>Executes the loop sequentially on the GPU.</td>
</tr>
<tr>
<td><code>private(list)</code></td>
<td>A copy of each variable in list is created for each iteration of the loop.</td>
</tr>
<tr>
<td><code>reduction(operator:list)</code></td>
<td><em>private</em> variables combined across iterations.</td>
</tr>
</tbody>
</table>
Loop Clauses Inside parallel Region

**gang**
Shares iterations across the gangs of the parallel region.

**worker**
Shares iterations across the workers of the gang.

**vector**
Execute the iterations in SIMD mode.
Loop Clauses Inside kernels Region

**gang [(num_gangs)]**
Shares iterations across across at most `num_gangs` gangs.

**worker [(num_workers)]**
Shares iterations across at most `num_workers` of a single gang.

**vector [(vector_length)]**
Execute the iterations in SIMD mode with maximum `vector_length`.

**independent**
Specify that the loop iterations are independent.
OTHER SYNTAX
### Other Directives

<table>
<thead>
<tr>
<th>Construct/Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cache</code> construct</td>
<td>Cache data in software managed data cache (CUDA shared memory).</td>
</tr>
<tr>
<td><code>host_data</code> construct</td>
<td>Makes the address of device data available on the host.</td>
</tr>
<tr>
<td><code>wait</code> directive</td>
<td>Waits for asynchronous GPU activity to complete.</td>
</tr>
<tr>
<td><code>declare</code> directive</td>
<td>Specify that data is to allocated in device memory for the duration of an implicit data region created during the execution of a subprogram.</td>
</tr>
</tbody>
</table>
Runtime Library Routines

**Fortran**

```fortran
use openacc
#include "openacc_lib.h"

acc_get_num_devices
acc_set_device_type
acc_get_device_type
acc_set_device_num
acc_get_device_num
acc_async_test
acc_async_test_all
```

**C**

```c
#include "openacc.h"

acc_async_wait
acc_async_wait_all
acc_shutdown
acc_on_device
acc_malloc
acc_free
```
Environment and Conditional Compilation

**ACC DEVICE device**

Specifies which device type to connect to.

**ACC DEVICE NUM num**

Specifies which device number to connect to.

**_OPENACC**

Preprocessor directive for conditional compilation. Set to OpenACC version.
Thank you