Applied Parallel Computing LLC
http://parallel-computing.pro

# OpenACC Interoperation

Dr. Aleksei Ivakhnenko

March 11, 2018

## Outline

- OpenACC interoperation with CUDA C, CUDA Fortran and GPU-enabled libraries
  - *host_data*
  - *deviceptr*
- External dependencies in OpenACC kernels, functions inlining
  - OpenACC routine directive, separate compilation and procedure calls
- Accessing global variables

## The host_data directive

This construct is used to make the device address of data available in host code.
The only valid clause is:

```
use_device (var-list)
```

The use_device tells the compiler to use the device address of any variable or array in the var-list in code within the construct. In particular, this may be used to pass the device address of variables or arrays to optimized procedures written in a lower-level API.

- Fortran:

```
!$acc host_data clause-list
      ...
!$acc end host_data clause-list
```

- C:

```
#pragma acc host_data clause-list
      for loop
```

## The `deviceptr` clause

- The deviceptr clause is used to declare that the pointers in the var-list are device pointers:
  - the data need not be allocated or moved between the host and device for this pointer.
- In C and C++, the variables in var-list must be pointer variables.
- In Fortran, the variable in var-list must be dummy arguments (arrays or scalars), and may not have the Fortran pointer, allocatable or value attributes.

## `host_data` vs `deviceptr`

`host_data` (directive):

- When you have allocated memory (e.g. by `#pragma acc declare device_resident`) on the device.
- You may pass this pointer to CUDA C/Fortran code or to GPU enabled library.

`deviceptr` (data clause):

- When you have allocated memory in CUDA C/Fortran code (e.g. by `cudaMalloc()`) or by GPU enabled library on the device.
- You may pass this pointer to OpenACC.

## The routine directive

The routine directive is used to tell the compiler to compile a given procedure for an accelerator as well as the host. In a file or routine with a procedure call, the routine directive tells the implementation the attributes of the procedure when called on the accelerator.

- Fortran:

```
!$acc routine clause-list
!$acc routine ( name ) clause-list
```

- C:

```
#pragma acc routine clause-list
#pragma acc routine ( name ) clause-list
```

## The `routine` directive

- In C and C++, the routine directive without a name may appear immediately before a function definition or just before a function prototype and applies to that immediately following function or prototype.
  - The routine directive with a name may appear anywhere that a function prototype is allowed and applies to the function in that scope with that name, but must appear before any definition or use of that function.
- In Fortran, the routine directive without a name may appear within the specification part of a subroutine or function definition, or within an interface body for a subroutine or function in an interface block, and applies to the containing subroutine or function.
  - The routine directive with a name may appear in the specification part of a subroutine, function or module, and applies to the named subroutine or function.

## The `routine` directive

Clauses:

- `gang`
- `worker`
- `vector`
- `seq`
- `bind`( name )
- `bind`( string )
- `device_type`( device-type-list )
- `nohost`

Restrictions:

- Only the `gang`, `worker`, `vector`, `seq` and `bind` clauses may follow a `device_type` clause.
- In C and C++, function `static` variables are not supported in functions to which a `routine` directive applies.
- In Fortran, variables with the `save` attribute, either explicitly or implicitly, are not supported in subprograms to which a `routine` directive applies.

## The routine directive: example

Listing 1: routine.cpp

```
int n = 1000;
float* a = (float*)malloc(n*sizeof(float));
float* b = (float*)malloc(n*sizeof(float));
float* c = (float*)malloc(n*sizeof(float));
for (int i = 0; i<n; i++)
{
    a[i] = (float)rand() / RAND_MAX;
    b[i] = (float)rand() / RAND_MAX;
}

#pragma acc parallel copyout (c[0:1000]), \
 copyin(a[0:1000], b[0:1000])
vecadd(a, b, c, n);

free(a);
free(b);
free(c);
```

Listing 2: routine_vecadd.hpp

```
void vecadd(
    float *a, float *b, float *c, int n);
```

Listing 3: routine_vecadd.cpp

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

## The routine directive: failure

```
$ pgc++ no_routine_vecadd.cpp -c -acc -Minfo=accel -ta=nvidia,time -o
no_routine_vecadd.o
no_routine_vecadd.cpp:
pgc++ no_routine.cpp -acc -Minfo=accel -ta=nvidia,time -o no_routine
./no_routine_vecadd.o
no_routine.cpp:
main:
     20, Generating copyin(a[:1000],b[:1000])
         Generating copyout(c[:1000])
         Accelerator kernel generated
         Generating Tesla code
nvlink error: Undefined reference to '_Z6vecaddPfS_S_i' in 'no_routine.o'
pgacclnk: child process exit status 2: /opt/pgi/linux86-64/15.7/bin/pgnvd
make: *** [no_routine] Error 2
```

# The routine directive: solution

Listing 4: routine.cpp

```
int n = 1000;
float* a = (float*)malloc(n*sizeof(float));
float* b = (float*)malloc(n*sizeof(float));
float* c = (float*)malloc(n*sizeof(float));
for (int i = 0; i<n; i++)
{
    a[i] = (float)rand() / RAND_MAX;
    b[i] = (float)rand() / RAND_MAX;
}

#pragma acc parallel copyout (c[0:1000]), \
 copyin(a[0:1000], b[0:1000])
vecadd(a, b, c, n);

free(a);
free(b);
free(c);
```

Listing 5: routine_vecadd.hpp

```
#pragma acc routine gang
void vecadd(
    float *a, float *b, float *c, int n);
```

Listing 6: routine_vecadd.cpp

```
#pragma acc routine gang
void vecadd(
    float *a, float *b, float *c, int n)
{
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

## The routine directive: solution

```
$ pgc++ routine_vecadd.cpp -c -acc -Minfo=accel -ta=nvidia,time -o routine_vecadd.o
routine_vecadd.cpp:
vecadd(float *, float *, float *, int):
      3, Generating acc routine gang
         Generating Tesla code
      5, #pragma acc loop gang, vector /* blockIdx.x threadIdx.x */
         Loop is parallelizable
pgc++ routine.cpp -acc -Minfo=accel -ta=nvidia,time -o routine ./routine_vecadd.o
routine.cpp:
main:
     20, Generating copyin(a[:1000],b[:1000])
         Generating copyout(c[:1000])
         Accelerator kernel generated
         Generating Tesla code
```

## Global variables

The `declare` directive must appear in any file that refers to the variable in device code; it must also appear in the file that actually declares the variable without the `extern` keyword:

```
extern float coef;

#pragma acc declare copyin(coef)
...
#pragma acc parallel
{
    float y = sin(coef);

    #pragma acc loop
    for (i = 0; i < n; ++i)
        x[i] *= y;
}
```