OpenACC programming

PRACE Summer of HPC 2022 - Day 4

Leon Bogdanović

University of Ljubljana, FME, LECAD lab

OpenACC
Directives for Accelerators

OpenACC: history and current state

https://www.openacc.org/

History:

- developed by Cray, CAPS, Nvidia and PGI (2011)
- latest version specification: 3.2 (2021)
- implementations available from Cray, PGI (now NVIDIA HPC SDK) and GNU (partial support)

A de facto standard for GPU directives:

- open standard, codes portable across parallel and multi-core processors
- directives allow easy acceleration of compute intensive applications
- offers complete access to the GPU massive parallel power
- OpenACC is **not GPU programming** per se, but rather **expressing parallelism** in the code

OpenACC: basic features

- **high-level compiler directives for specifying parallel regions in Fortran and C/C++**:
  - parallel regions offloading
  - portability across operating systems, accelerators, host CPUs and compilers
- **high-level heterogenous programming**:
  - without explicit accelerator initialization
  - without explicit data or program transfers between host and accelerator
- **low-level features**:
  - compiler can give additional hints (loop mappings, data location and other performance details)
  - compatibility with other GPU languages and libraries (e.g., CUDA C and CUDA Fortran, GPU libraries: cuFFT, cuBLAS, cuSPARSE…)
- **OpenMP comparison**:
  - modeled after OpenMP - but specific for accelerators, OpenMP 4.0/4.5 also supports offloading
  - OpenACC more descriptive, OpenMP more prescriptive
  - same execution model : fork/join model

OpenACC: porting codes to GPU

- **OpenACC execution model**:
  - host-directed execution with an attached accelerator
  - part of the program is usually executed by the host
  - computationally intensive parts are offloaded to the accelerator that executes parallel regions
- **OpenACC data model**:
  - host manages memory of the device
  - host copies data to/from the device
- **compilers that support OpenACC usually require an option that enables the feature**:
  - PGI (now NVIDIA HPC SDK): `-acc`
  - Cray: `-hacc`
  - GNU (partial support): `-fopenacc`

OpenACC: directives syntax

	sentinel	construct	clauses	
C/C++	`#pragma acc`	`data`	`copy(data)`	data model
		`update`	`host(data)`	data model
		`kernels`		execution model
		`parallel`		execution model
Fortran	`!$acc`	`data`	`copy(data)`	data model
		`update`	`host(data)`	data model
		`kernels`		execution model
		`parallel`		execution model

- OpenACC uses **compiler directives** for defining compute regions (and data transfers) that are to be executed on a GPU
- **important constructs**:
  - `parallel, kernels, data, loop, update, host_data, wait`
- **often used clauses**:
  - `if (condition), async(handle)`

OpenACC: basic programming structure

C/C++:

```
#include "openacc.h"
#pragma acc <directive> [clauses [[,] clause] . . .]
<code>
```

Fortran:

```
use openacc
!$acc <directive> [clauses [[,] clause] . . .]
<code>
!$acc end <directive>
```

EXAMPLE 1

OpenACC Hello world

Based on material by E. Krishnasamy: https://ulhpc-tutorials.readthedocs.io/en/latest/gpu/openacc/basics/

See also (Week 4 & 5): https://www.futurelearn.com/courses/gpu-programming-scientific-computing

Hello world example in C

Code:

`Hello_World.c, Hello_World_OpenACC.c`

Notebook:

`17_Hello_World_OpenACC.ipynb`

C:

```
void Print_Hello_World()
{
  for(int i = 0; i < 5; i++)
    {
      printf("Hello World!\n");
    }
}
```

C OpenACC:

```
void Print_Hello_World()
{
#pragma acc kernels
  for(int i = 0; i < 5; i++)
    {
      printf("Hello World!\n");
    }
}
```

- compile the C code:

```
pgcc -fast -Minfo=all -ta=tesla -acc Hello_World.c -o Hello_World_C
```

- what information is given by the compiler?
- add either `kernels` or `parallel` directives to vectorize/parallelize the loop and compile the code:

```
pgcc -fast -Minfo=all -ta=tesla -acc Hello_World_OpenACC.c -o Hello_World_OpenACC_C
```

- what information is now given by the compiler?

Hello world example in C (cont.)

C code compiler info:

```
Print_Hello_World:
      5, Loop not vectorized/parallelized: contains call
main:
     13, Print_Hello_World inlined, size=4 (inline) file Hello_World.c (4)
           5, Loop not vectorized/parallelized: contains call
```

- the loop **is not parallelized**
- can be parallelized with OpenACC

C OpenACC code compiler info:

```
Print_Hello_World:
      8, Loop is parallelizable
         Generating Tesla code
          8, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

- the loop **is now parallelized**
- the loop was offloaded to an NVIDIA GPU:

`/* blockIdx.x threadIdx.x */`

Hello world example in Fortran

Code:

`Hello_World.f90, Hello_World_OpenACC.f90`

Notebook:

`17_Hello_World_OpenACC.ipynb`

Fortran:

```
subroutine Print_Hello_World()
  integer ::i
  do i=1,5
     print *, "hello world"
  end do
end subroutine Print_Hello_World
```

Fortran OpenACC:

```
subroutine Print_Hello_World()
  integer ::i
  !$acc kernels
  do i=1,5
     print *, "hello world"
  end do
  !$acc end kernels
end subroutine Print_Hello_World
```

- compile the Fortran code:

```
pgfortran -fast -Minfo=all -ta=tesla -acc Hello_World.f90 -o Hello_World_F
```

- what information is given by the compiler?
- add either `kernels` or `parallel` directives to vectorize/parallelize the loop and compile the code:

```
pgfortran -fast -Minfo=all -ta=tesla -acc Hello_World_OpenACC.f90 -o
Hello_World_OpenACC_F
```

- what information is now given by the compiler?

Hello world example in Fortran (cont.)

Fortran code compiler info:

```
print_hello_world:
      3, Loop not vectorized/parallelized: contains call
```

- the loop **is not parallelized**
- can be parallelized with OpenACC

Fortran OpenACC code compiler info:

```
print_hello_world:
      4, Loop is parallelizable
         Generating Tesla code
          4, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

- the loop **is now parallelized**
- the loop was offloaded to an NVIDIA GPU:

```
! blockidx%x threadidx%x
```

EXAMPLE 2

OpenACC vector addition

Based on material by E. Krishnasamy: https://ulhpc-tutorials.readthedocs.io/en/latest/gpu/openacc/basics/

See also (Week 4 & 5): https://www.futurelearn.com/courses/gpu-programming-scientific-computing

Vector addition example in C

Code:

`Vector_Addition.c, Vector_Addition_OpenACC.c`

Notebook:

`18_Vector_addition_OpenACC.ipynb`

C:

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

C OpenACC:

```c
void Vector_Addition(float *a, float *b, float *restrict c, int n)
{
#pragma acc kernels loop copyin(a[:n], b[0:n]) copyout(c[0:n])
  for(int i = 0; i < n; i ++)
    {
      c[i] = a[i] + b[i];
    }
}
```

- the `loop` construct will parallelize the `for` loop and also accommodate other OpenACC clauses, e.g., `copyin` and `copyput`
- two input vectors `a` and `b` are transferred to the GPU and one vector `c` (sum of input vectors) is transferred back to the CPU
- `copyin` will allocate memory on the GPU and transfer the data from CPU to GPU
- `copyout` will allocate memory on the GPU and transfer the data from GPU to CPU

Vector addition example in Fortran

Code:

`Vector_Addition.f90, Vector_Addition_OpenACC.f90`

Notebook:

`18_Vector_addition_OpenACC.ipynb`

Fortran:

```
module Vector_Addition_Mod
  implicit none
contains
  subroutine Vector_Addition(a, b, c, n)
    ! Input vectors
    real(8), intent(in), dimension(:) :: a
    real(8), intent(in), dimension(:) :: b
    real(8), intent(out), dimension(:) :: c
    integer :: i, n
    do i = 1, n
       c(i) = a(i) + b(i)
    end do
  end subroutine Vector_Addition
end module Vector_Addition_Mod
```

Fortran OpenACC:

```
module Vector_Addition_Mod
  implicit none
contains
  subroutine Vector_Addition(a, b, c, n)
    ! Input vectors
    real(8), intent(in), dimension(:) :: a
    real(8), intent(in), dimension(:) :: b
    real(8), intent(out), dimension(:) :: c
    integer :: i, n
    !$acc kernels loop copyin(a(1:n), b(1:n)) copyout(c(1:n))
    do i = 1, n
       c(i) = a(i) + b(i)
    end do
    !$acc end kernels
  end subroutine Vector_Addition
end module Vector_Addition_Mod
```

EXERCISE 1

OpenACC profiling

Profiling vector addition OpenACC codes

- compile all vector addition codes in C and Fortran and execute them
- what information is given by the compiler and what are the outputs of the programs?
- execute OpenACC programs for different vector sizes and use **automatic instrumentation** at runtime with:

```
PGI_ACC_TIME=1 ./Vector_Addition_OpenACC_C
```

- how does execution times, kernel launch times and memory transfer times increase with vector sizes?
- is there any difference in execution between C and Fortran codes?

EXAMPLE 3

OpenACC vector addition with reduction

Based on material by E. Krishnasamy: https://ulhpc-tutorials.readthedocs.io/en/latest/gpu/openacc/basics/

See also (Week 4 & 5): https://www.futurelearn.com/courses/gpu-programming-scientific-computing

Vector addition with reduction

Code:

`Vector_Addition_Reduction_OpenACC.c`

Notebook:

`19_Vector_addition_reduction_OpenACC.ipynb`

C OpenACC:

```c
void Vector_Addition(float *a, float *b, float *restrict c, int n)
{
#pragma acc kernels loop reduction(+:sum) copyin(a[:n], b[0:n]) copyout(c[0:n])
  for(int i = 0; i < n; i ++)
    {
      c[i] = a[i] + b[i];
      sum += c[i];
    }
}
```

- the `reduction` clause is used for summing arrays or any counting inside the parallel region
- using reduction generally results in increase of performance and minimization of the error in the total sum

Code:

`Vector_Addition_Reduction_OpenACC.f90`

Fortran OpenACC:

```fortran
module Vector_Addition_Mod
  implicit none
contains
  subroutine Vector_Addition(a, b, c, n, sum)
    ! Input vectors
    real(8), intent(in), dimension(:) :: a
    real(8), intent(in), dimension(:) :: b
    real(8), intent(out), dimension(:) :: c
    real(8) :: sum
    integer :: i, n
    !$acc kernels loop reduction(+:sum) copyin(a(1:n), b(1:n)) copyout(c(1:n))
    do i = 1, n
       c(i) = a(i) + b(i)
       sum = sum + c(i)
    end do
    !$acc end kernels
  end subroutine Vector_Addition
end module Vector_Addition_Mod
```

EXERCISE 2

Riemann sum with OpenACC

Exercise 2: Riemann sum with OpenACC

Code:

`riemann_openacc_gpu_exercise.c`

Notebook:

`16_Riemann_sum_OpenACC_GPU.ipynb`

Complete the following tasks:

- use OpenACC to parallelize the for loop in the Riemann sum C (or Fortran) code
- execute the code and compare the performance with the CUDA code
- use the tool `nsys` in command line to profile the code, then use OpenACC automatic instrumentation and compare the results of the profilers