
# Uni.lu HPC School 2021

## PS10a: Introduction to GPU programming with OpenACC

**Uni.lu High Performance Computing (HPC) Team**

**Dr. E. Krishnasamy**

University of Luxembourg (UL), Luxembourg

http://hpc.uni.lu

**Latest versions available on Github:**

UL HPC tutorials: https://github.com/ULHPC/tutorials

UL HPC School: hpc.uni.lu/education/hpcschool

PS10a tutorial sources: ulhpc-tutorials.rtfd.io/en/latest/openacc/

## Summary

# Objectives

- Understanding the OpenACC programming model
- How to use some of the directives from OpenACC to parallelize the code
  - ↪ compute constructs, loop constructs, data clauses
- Implementing OpenACC parallel strategy in C/C++ and FORTRAN programming languages
- Simple mathematical examples to support and understand the OpenACC programming model
- Finally to show how to run these examples using Iris cluster (ULHPC) - both interactively and using a batch job script

# Prerequisite

- C/C++ and/or FORTRAN languages
- OpenMP or some basic parallel programming concept (advantage but not necessary)

***NOTE****: this lecture is limited to just 45 min, it only covers very basic tutorial about OpenACC. To know more about (from basic to advanced) CUDA programming and OpenACC programming model, please refer to* ***PRACE MOOC*** ***GPU Programming for Scientific Computing and Beyond - Dr. Ezhilmathi Krishnasamy***

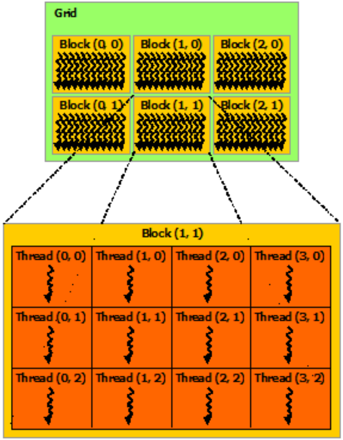
# Summary

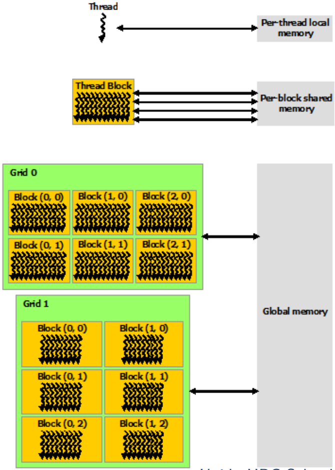
# CPU vs GPU

- CPU frequency is higher compared to GPU
- But GPU can run many threads in parallel compared to CPU
- On the GPU, the cores are grouped and called “Streaming Multiprocessor - SM”
- Even on the Nvidia GPU, it has a “Tensor Process Unit - TPU” to handle the AI/ML computations in an optimized way
- GPU are based on the “Single Instruction Multiple Threads”
- Threads are executed in a group on the GPU, typically they have 32 threads. This is called “warps” on the Nividia GPU and “wavefronts” on the AMD GPU.

# CPU vs GPU

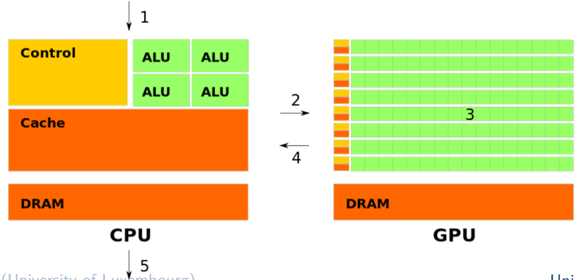
# Thread Hierarchy



# How GPUs are used for computations

- Step 1: application preparation, initialize the memories on both CPU and GPU
- Step 2: transfer the data to GPU
- Step 3: do the computation on the GPU
- Step 4: transfer the data back to CPU
- Step 5: finalize the application and delete the memories on both CPU and GPU

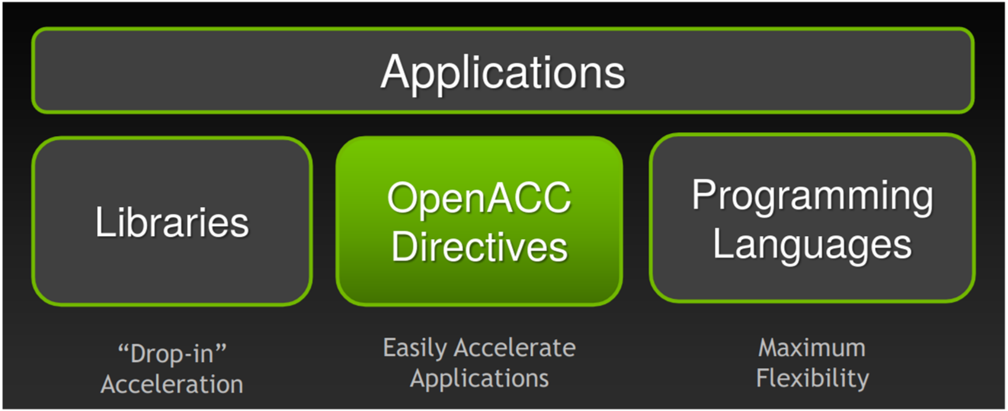


# Summary

# Few points about OpenACC

- OpenACC is not GPU programming
- OpenACC is expressing the parallelism in your code
- OpenACC can be used in both Nvidia and AMD GPUs
- "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

## Ways to accelerate applications on the GPU



# Ways to accelerate applications on the GPU

- Libraries: easy to use with very limited knowledge with GPU programming
  - ↪ cuBLAS, cuFFT, CUDA Math Library, etc,.
- Directive based programming model: will accelerate the application with using directives in the existing code
  - ↪ OpenACC and OpenMP (might be applicable in the future)
- Programming languages: low level programming languages that will further optimize the application on the accelerator
  - ↪ CUDA, OpenCL, etc,.

# Compilers and directives (only few of them listed in here)

- OpenACC is supported by the Nvidia, PGI, GCC, and HPE Gray (only for FORTRAN) compilers
  - ↪ Now PGI is part of Nvidia, and it is available through Nvidia HPC SDK
- Compute directives:
  - ↪ parallel and kernel
- Loop directive:
  - ↪ loop
- Data management directives:
  - ↪ copyin and copyout
- Others:
  - ↪ reduction

# Basic programming structure

```
// C/C++
#include "openacc.h"
#pragma acc <directive> [clauses [[,] clause] . . .] new-line
<code>
```

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

# Summary

# kernels in C/C++

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

- compilation: `pgcc -fast -Minfo=all -ta=tesla -acc Hello_World.c`
  - ↪ Compiler will give already much info, what do you see?

# kernels in FORTRAN

```
!! Hello_World.f90                   | !! Hello_World_OpenACC.f90
subroutine Print_Hello_World()       | subroutine Print_Hello_World()
  integer :: i                       |   integer :: i
                                     |   !$acc kernels
  do i = 1, 5                        |   do i = 1, 5
     print *, "hello world"          |     print *, "hello world"
  end do                             |   end do
                                     |  !$acc end kernels
end subroutine Print_Hello_World     | end subroutine Print_Hello_World
```

- compilation: `pgfortran -fast -Minfo=all -ta=tesla -acc Hello_World_OpenACC.f90`
  - ↪ -ta refers to target architecture, here is it Nvidia Tesla and -acc compiler flag instructing to target accelerators

# Summary

# loop and data directive in vector addition in C/C++

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

# loop and data directive in vector addition in FORTRAN

```
!! Vector_Addition.f90                          | !! Vector_Addition_OpenACC.f90
module Vector_Addition_Mod                      | module Vector_Addition_Mod
  implicit none                                 |   implicit none
contains                                        | contains
  subroutine Vector_Addition(a, b, c, n)        |   subroutine Vector_Addition(a, b, c, n)
    !! Input vectors                            |     !! Input vectors
    real(8), intent(in), dimension(:) :: a      |     real(8), intent(in), dimension(:) :: a
    real(8), intent(in), dimension(:) :: b      |     real(8), intent(in), dimension(:) :: b
    real(8), intent(out), dimension(:) :: c     |     real(8), intent(out), dimension(:) :: c
    integer :: i, n                             |     integer :: i, n
                                                |     !$acc kernels loop copyin(a(1:n), b(1:n))
                                                |      copyout(c(1:n))
    do i = 1, n                                 |     do i = 1, n
       c(i) = a(i) + b(i)                       |        c(i) = a(i) + b(i)
    end do                                      |     end do
                                                |     !$acc end kernels
  end subroutine Vector_Addition                |   end subroutine Vector_Addition
end module Vector_Addition_Mod                  | end module Vector_Addition_Mod
```

# Summary

# reduction directive in vector addition in C/C++

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

# reduction directive in vector addition in FORTRAN

```
!! Vector_Addition.f90                          | !! Vector_Addition_OpenACC.f90
module Vector_Addition_Mod                      | module Vector_Addition_Mod
  implicit none                                 |   implicit none
contains                                        | contains
  subroutine Vector_Addition(a, b, c, n)        |   subroutine Vector_Addition(a, b, c, n)
    !! Input vectors                            |     !! Input vectors
    real(8), intent(in), dimension(:) :: a      |     real(8), intent(in), dimension(:) :: a, b
    real(8), intent(in), dimension(:) :: b      |     real(8):: sum
    real(8), intent(out), dimension(:) :: c     |     real(8), intent(out), dimension(:) :: c
    integer :: i, n                             |     integer :: i, n
                                                |     !$acc kernels loop reduction(+:sum)
                                                |      copyin(a(1:n), b(1:n)) copyout(c(1:n))
    do i = 1, n                                 |     do i = 1, n
       c(i) = a(i) + b(i)                       |        c(i) = a(i) + b(i)
    end do                                      |        sum = c(i)
                                                |     end do
  end subroutine Vector_Addition                |     !$acc end kernels
end module Vector_Addition_Mod                  |   end subroutine Vector_Addition
                                                | end module Vector_Addition_Mod
```

Thank you for your attention...

# Questions?

ulhpc-tutorials.rtfd.io/en/latest/openacc/


*High Performance Computing @ Uni.lu*

University of Luxembourg, Belval Campus
Maison du Nombre, 4th floor
2, avenue de l'Université
L-4365 Esch-sur-Alzette
*mail:* hpc@uni.lu




## Uni.lu HPC School 2021 Contributors

- **Dr. Xavier Besseron** – Research Scientist
- **Hyacinthe Cartiaux** – Infra. & HPC Arch. Engineer
- **Dr. Aurelien Ginohac** – Research Scientist
- **Dr. Emmanuel Kieffer** – Research Scientist
- **Dr. Loizos Koutsantonis** – Postdoctoral Researcher
- **Dr. Ezhilmathi Krishnasamy** – Postdoctoral Researcher
- **Abatcha Olloh** – Infra. & HPC Arch. Engineer
- **Dr. Tiago C. Pessoa** – Postdoctoral Researcher
- **Sarah Peter** – Infra. & Arch. Engineer
- **Teddy Valette** – Infra. & HPC Arch. Engineer
- **Dr. Sebastien Varrette** – Research Scientist

... and additional help (Survey, session tests)

- **Arlyne Vandeventer** – Project Manager

hpc.uni.lu