Introduction to OpenACC programming

Dr. Ezhilmathi Krishnasamy

University of Luxembourg, ULHPC & PCOG, FSTM

June 7, 2023

Outline

1. Introduction to OpenACC programming
2. Parallel programming
3. Parallel computer architecture
4. GPU architecture
5. GPU computing
6. Basics of OpenACC
7. Compute and loop constructs
8. Data construct
9. Controlling threads
10. Profiling
11. Other clauses
12. Practical session

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, show you how to run these examples using Iris cluster (ULHPC)
  - both interactively and using a batch job script.

Prerequiste

- C/C++ and/or FORTRAN languages.
- Knowing OpenMP or some basic parallel programming concept is an advantage but not necessary.

Note

This session is limited to 30-45 min. lecture and 30-45 min. practicals; therefore, it only covers a basic tutorial about OpenACC.

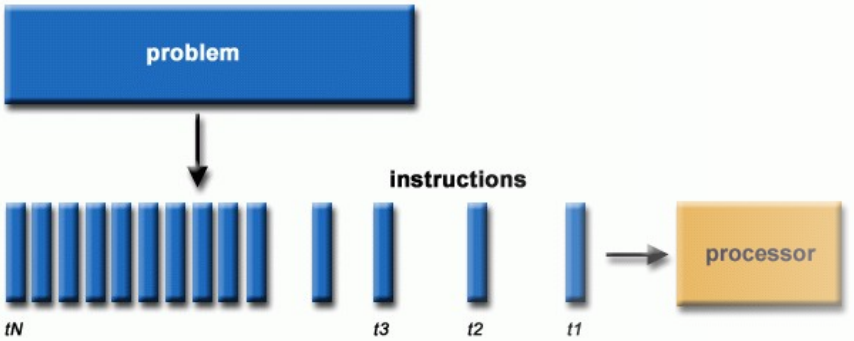
MOOC GPU programming - running from September- 2023

To know more about (from basic to advanced) CUDA programming and the OpenACC programming model, please refer to **PRACE MOOC GPU Programming for Scientific Computing and Beyond** - *Dr. Ezhilmathi Krishnasamy* and *Prof. Pascal Bouvry*

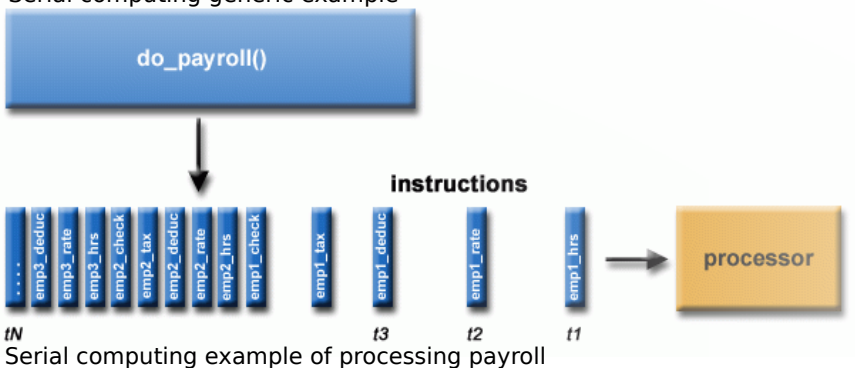
Serial programming vs. parallel programming

- Serial programming
  - An entire problem can be divided into discrete series of instructions.
  - All the instructions are executed one by one.
  - Executed by single thread or processor.
  - Only one instruction can be executed at the same time.
- Parallel programming
  - An entire problem can be divided into discrete parts in such a way that it can be solved concurrently.
  - Each part may have a set of instructions.
  - Each part's instructions are executed on a different thread/processor.
  - Since it is parallel execution, a target problem needs to be controlled/coordinated.
- CPU, GPU, and other parallel processors can perform the parallel computing

Serial programming vs. parallel programming

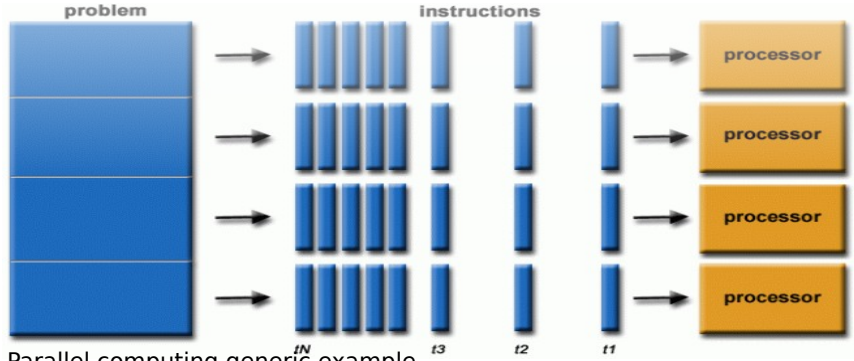


Serial computing generic example

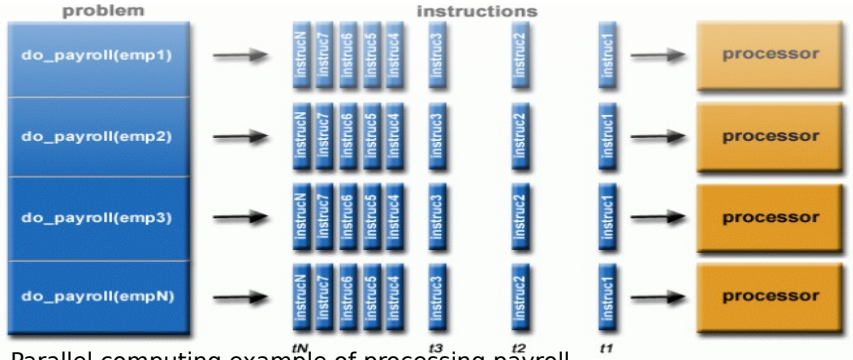


Serial computing example of processing payroll

Source:HPC LLNL



Parallel computing generic example



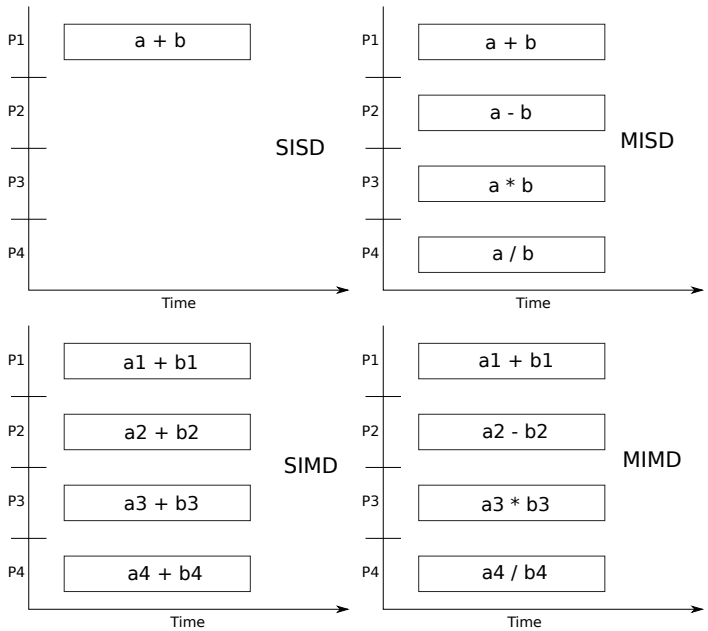
Parallel computing example of processing payroll

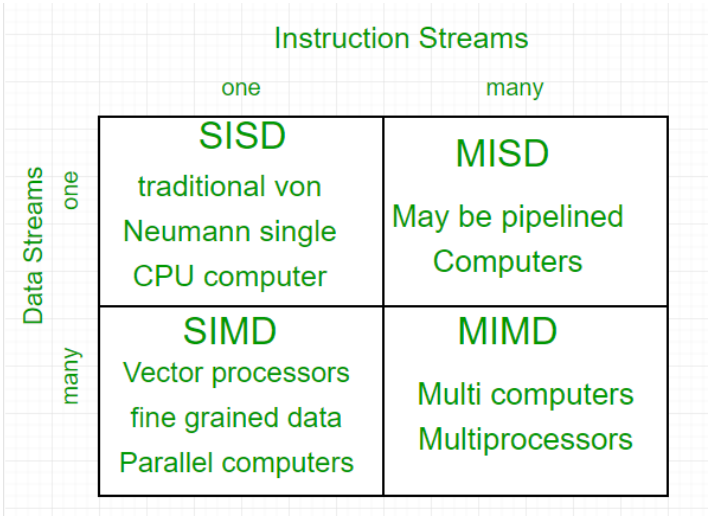
Classification of parallel computer architecture

- Control structure: based on the instructions and data streams.
  - SISD - single instruction stream, single data stream
  - SIMD - single instruction streams, multiple data streams
  - MISD - multiple instruction streams, single data stream
  - MIMD - multiple instruction streams, multiple data streams
- Memory organisation: shared memory and distributed memory
- Network topology (connectivity): for example, 3D grid and tree

Control structure - Flynn's taxonomy

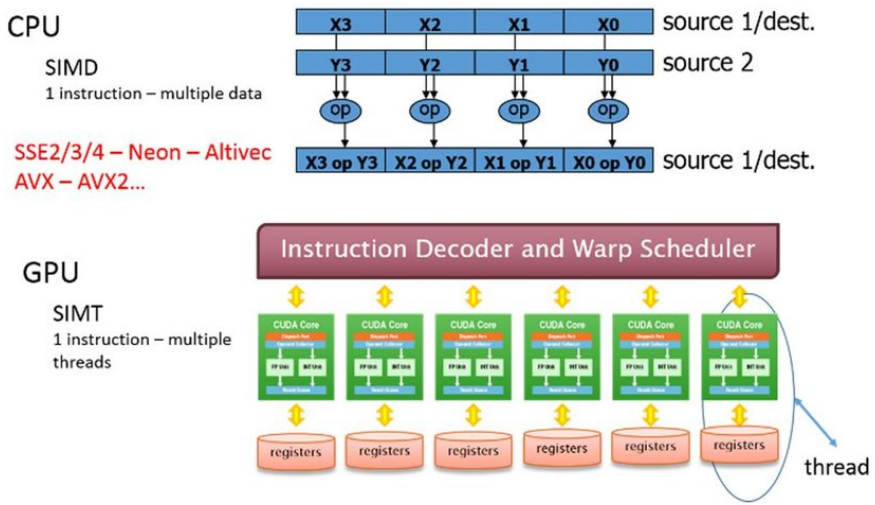
- SISD - an example of sequential execution (conventional single processor - von Neumann model).
- SIMD - an example of array computers and traditional vector processors; instructions can be executed in a pipeline or in parallel.
- MISD - it is not used commonly.
- MIMD - it works with both shared and distributed memory models.





GPU architecture

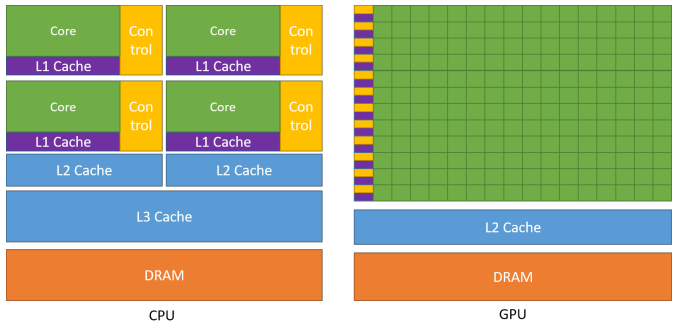
GPUs are based on Single Instruction Multiple Threads (SIMT)



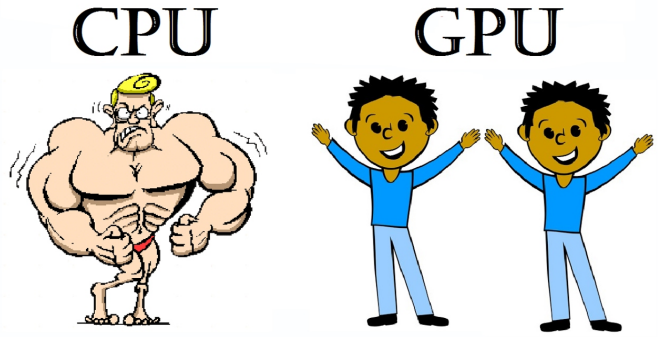
Source:Daniel E. 45 year CPU evolution

CPU vs. GPU

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

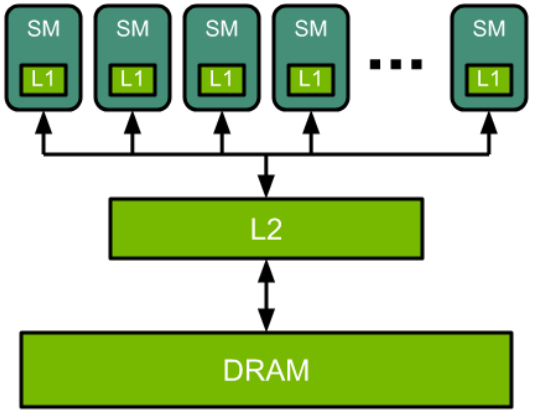


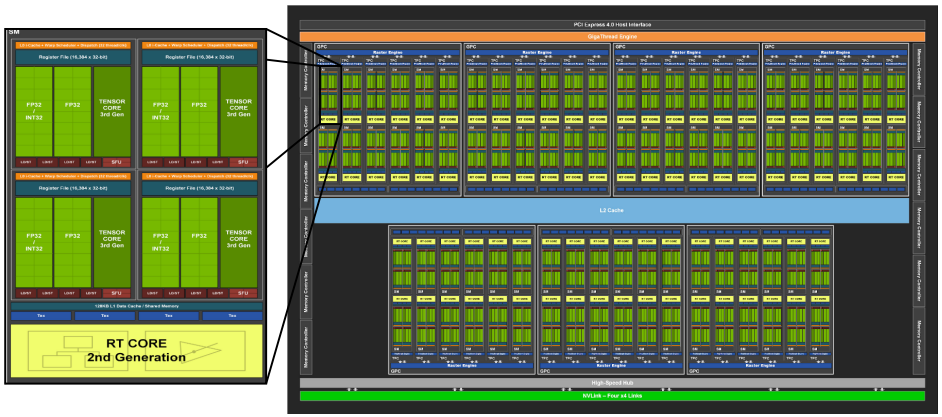
Source:Nvidia: CUDA programming



GPU architecture

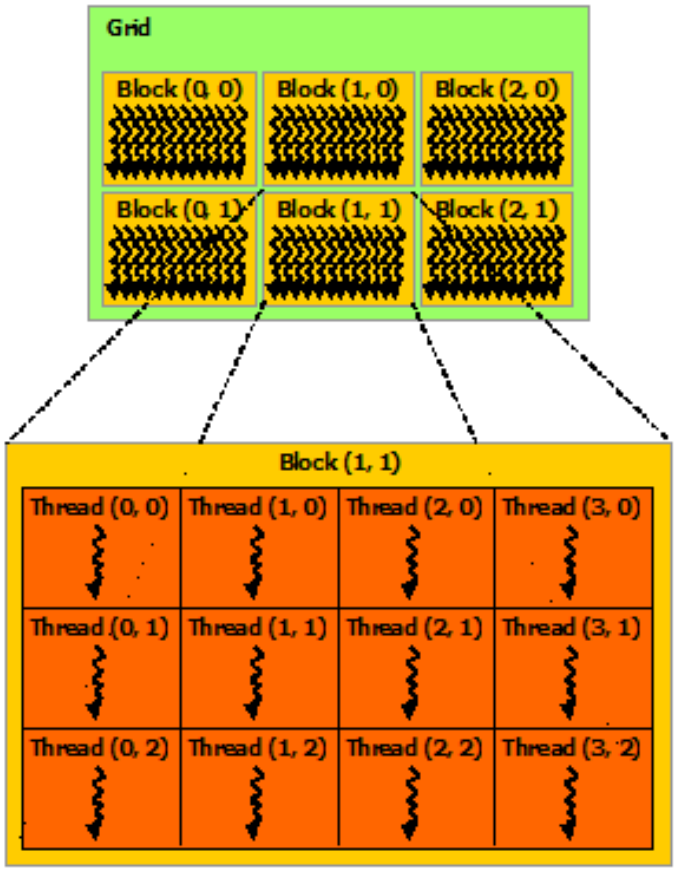
- Ampere GPU has seven Graphics Processing Clusters (GPCs), 42 TPCs, and 84 SMs.
- Volta GPU has six GPCs; each GPC has seven TPCs (each including two SMs), and 14 SMs.
- Each SMs has L1 cache (up to 128 KB), and L2 (up to 6144 KB) cache is shared between the GPCs.
- RT (Ray Tracing) cores dedicated to doing the ray-tracing rendering math computation.
- Tensor Cores: provides the speedups for AI neural network training computation.
- Programmable Shading Cores, which have CUDA cores.

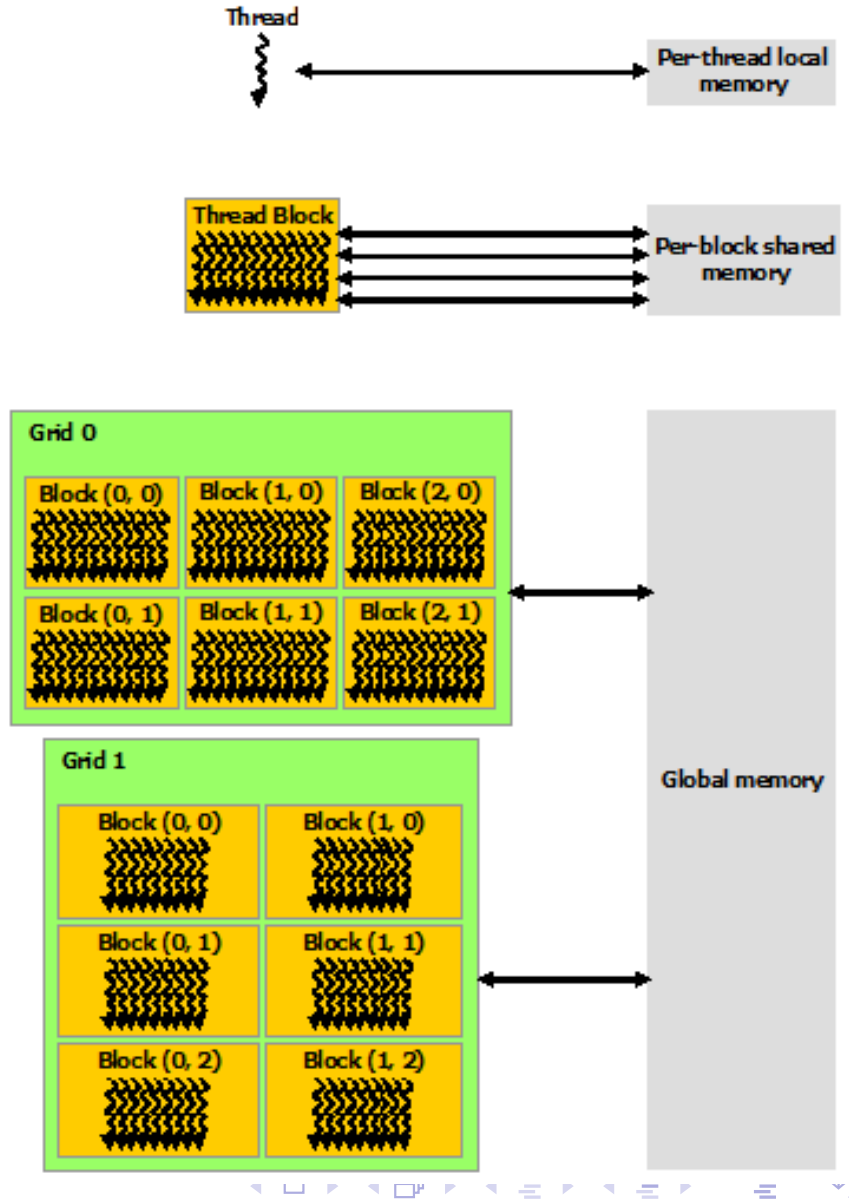




Source:Nvidia: deep learning

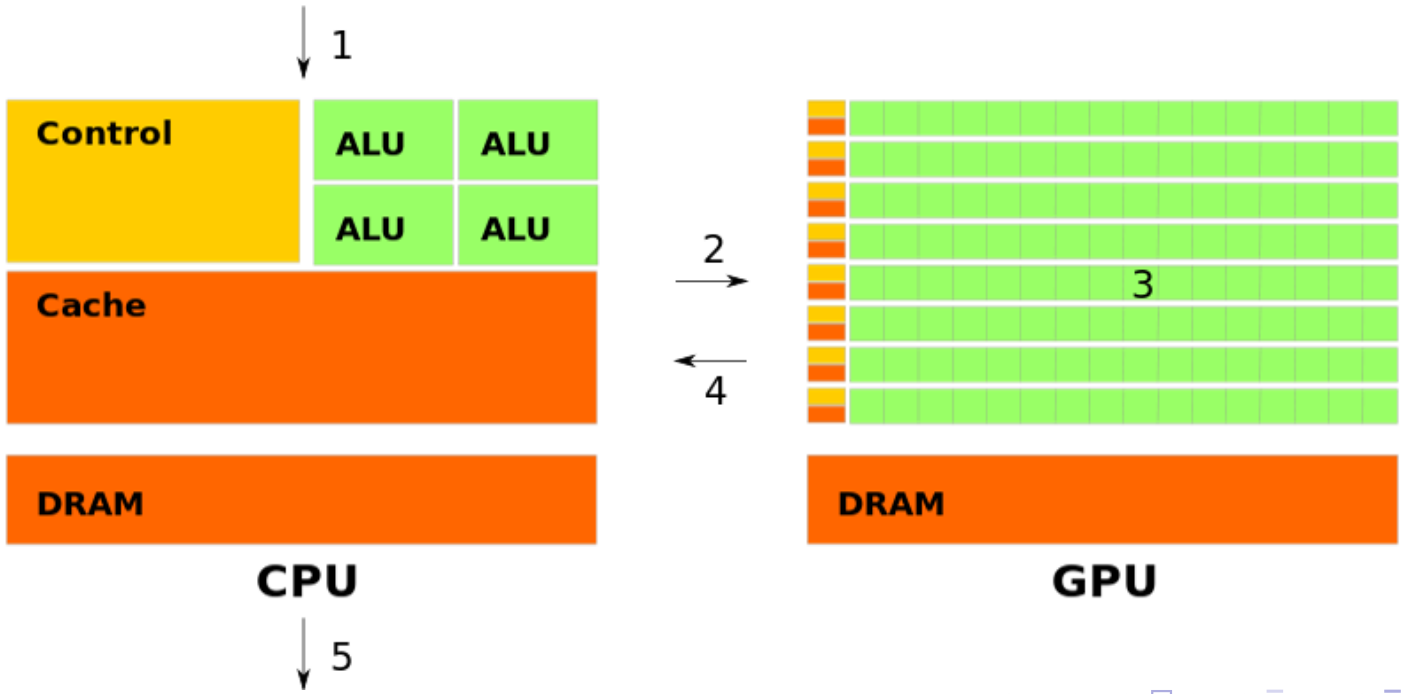
Thread hierarchy





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 the CPU.
- Step 5: finalize the application and delete the memories on both CPU and GPU.

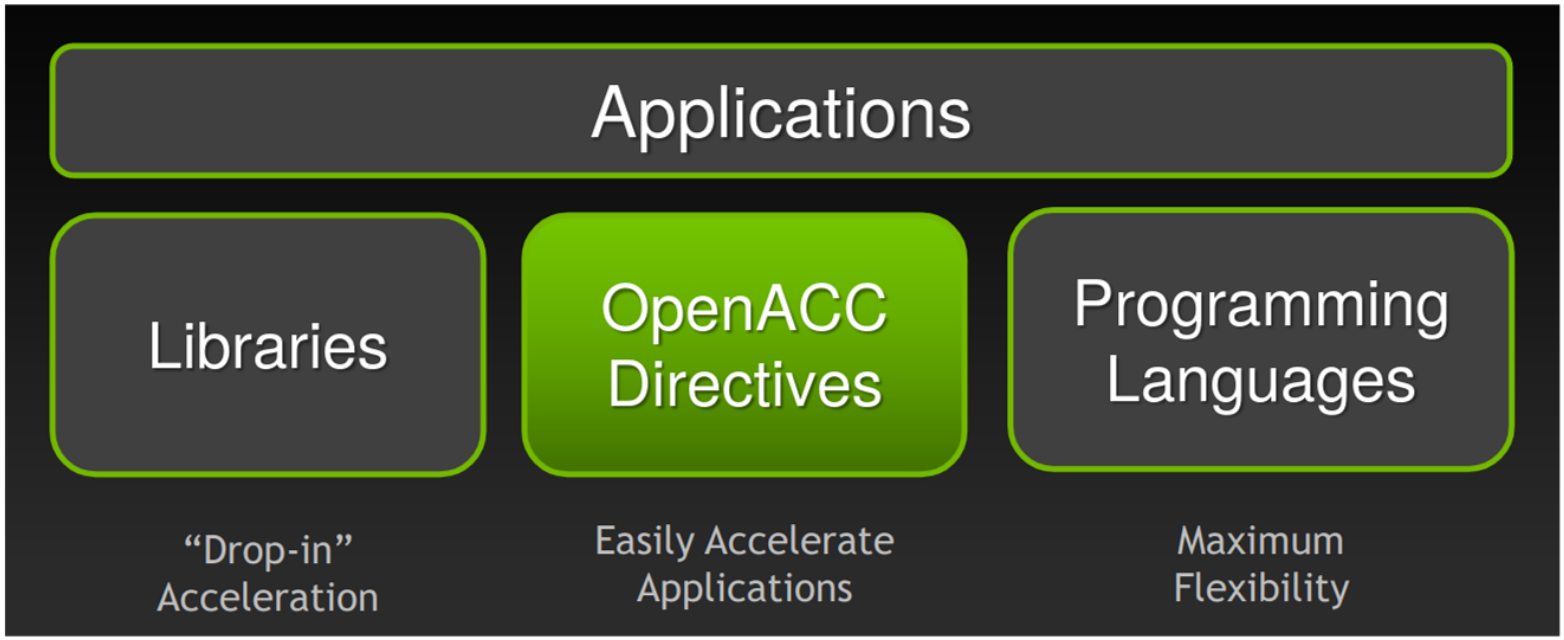


Few points about OpenACC

- OpenACC is not GPU programming language
  - compiler and directive-based.
- OpenACC is expressing the parallelism in your code
  - for example, in C/C++ and FORTRAN.
- OpenACC can be used in both Nvidia and AMD GPUs
  - single source code, no need to change a code for another GPU vendor.
- Latest version of OpenACC API contains definitions of directives and clauses.

"OpenACC will enable programmers to develop portable applications that maximize easily the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan." - *Buddy Bland, Titan Project Director, Oak Ridge National Lab.*

Ways to accelerate applications on the GPU



"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 of OpenMP Directives Board.*

Ways to accelerate applications on the GPU

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

Compilers

- OpenACC is supported by many commercial and open-source compilers; however, the following are widely used.
  - PGI
  - GCC
  - HPE Gray (only for FORTRAN) compilers.

- Now PGI is part of Nvidia, and it is available through Nvidia HPC SDK.

- Today, we will use the Nvidia HPC SDK compiler for our practical session.

Compiler options

- `-acc:` the compiler will recognize the OpenACC directives OpenACC is also able to generate code for multicore CPUs (close to OpenMP). Some interesting options are:
  - `-acc=gpu` to build for GPU
  - `-acc=multicore` to build for CPU (multithreaded)
  - `-acc=host` to build for CPU (sequential)
  - `-acc=noautopar` disable the automatic parallelization inside parallel regions (the default is -acc=autopar)
- `-gpu:` GPU-specific options to be passed to the compiler Some interesting options are:
  - `-gpu=ccXX` specify the compute capability for which the code has to be built, for example, cc70 Compile for compute capability 7.0.
  - `-gpu=managed` activate NVIDIA Unified Memory (with it, you can ignore data transfers, but it might fail sometime)
  - `-gpu=pinned` activate pinned memory. It can help to improve the performance of data transfers
  - `-gpu=lineinfo` generate debugging line information; less overhead than -g

Profiling information during compilation

- `-Minfo:` the compiler prints information about the optimizations it uses
- `-Minfo=accel` information about OpenACC (Mandatory in this training course!)
- `-Minfo=all` all optimizations are printed (OpenACC, vectorization, FMA, ...), recommended.

C: nvc -fast -Minfo=accel -acc=gpu -gpu=cc70 Test.c

C++: nvc++ -fast -Minfo=accel -acc=gpu -gpu=cc70 Test.cc

FORTRAN: nvfortran -fast -Minfo=accel -acc=gpu -gpu=cc70 Test.f90

Device information

$ nvaccelinfo

```
CUDA Driver Version:           12000
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  525.85.12
                               Sat Jan 28 02:10:06 UTC 2023

Device Number:                 0
Device Name:                   Tesla V100-SXM2-32GB
Device Revision Number:        7.0
Global Memory Size:            34079637504
Number of Multiprocessors:     80
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
```

Directives

- Compute constructs:
  - parallel and kernel - creating a parallel region.
- Loop constructs:
  - loop, collapse, gang, worker, vector, etc. - efficiently use the threads for work-sharing constructs.
- Data management clauses:
  - copy, create, copyin, copyout, delete and present - data handling between host and device.
- Others:
  - reduction, atomic, cache, etc. - for some special operations not to slow down the parallel computation.
- More information about the OpenACC directives can be found here.

Basic programming structure

Example for C/C++

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

Example for FORTRAN

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

Compute constructs

- `parallel`
  - Very similar to OpenMP directive.
  - It's up to the programmer's responsibility to control the parallel region.
  - Beginners need to be very careful to avoid data race or incorrect computation.

- `kernels`
  - Compilers choose the best options to parallelize within `kernels` scope.
  - Beginners can simply take advantage of this.
  - However, a data movement must be done carefully; otherwise, no parallelization would occur, for example, pointer aliasing with `restrict`.

kernels in C/C++

```c
// Hello_World.c
void Print_Hello_World()
{

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

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

compilation

nvc -fast -Minfo=all -acc=gpu -gpu=cc70 Hello_World.c - the compiler will already give much info; what do you see?

kernels in FORTRAN

```
1  !! Hello_World.f90
2  subroutine Print_Hello_World()
3    integer :: i
4
5    do i = 1, 5
6       print *, "hello world"
7    end do
8
9  end subroutine Print_Hello_World
```

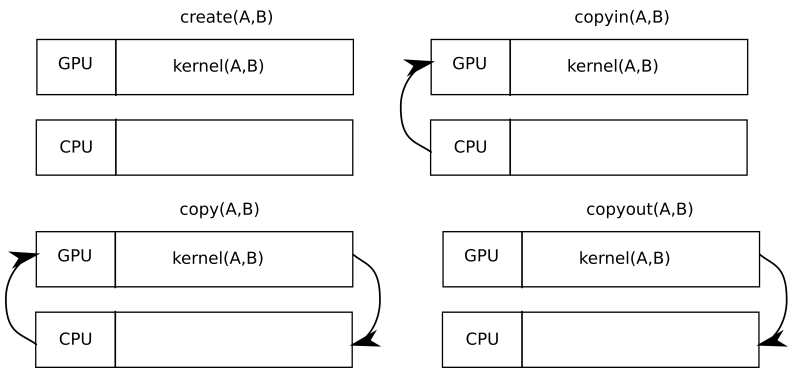
```
1  !! Hello_World_OpenACC.f90
2  subroutine Print_Hello_World()
3    integer :: i
4    !$acc kernels loop
5    do i = 1, 5
6      print *, "hello world"
7    end do
8   !$acc end kernels
9  end subroutine Print_Hello_World
```

compilation

nvfrotran -fast -Minfo=all -acc=gpu Hello_World_OpenACC.f90; here -acc refers to target-specific switches; and -gpu=cc70 refers to compute capability.

Data management

- copyin(list) - Allocates memory on GPU and copies data from CPU(host) to GPU when entering a region
- copyout(list) - Allocates memory on GPU and copies data to the CPU(host)host when exiting a region
- copy(list) - Allocates memory on GPU and copies data from CPU(host) to GPU when entering region and copies data to the CPU(host) when exiting a region
- create(list) - Allocates memory on GPU but does not copy
- delete(list) - Deallocate memory on the GPU without copying
- present(list) - Data is already present on GPU from another containing data a region



Loop and data clauses in vector addition in C/C++

```c
// Vector_Addition.c
float * Vector_Addition
(float *restrict a, float *restrict b,
 float *restrict c, int n)
{


  for(int i = 0; i < n; i ++)
    {
      c[i] = a[i] + b[i];
    }
  return c;
}
```

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

}
```

```
$ nvc -fast -Minfo=accel -acc=gpu -gpu=cc70 Vector_Addition_OpenACC.c
Vector_Addition:
     14, Generating copyin(a[:n]) [if not already present]
         Generating copyout(c[:n]) [if not already present]
         Generating copyin(b[:n]) [if not already present]
     16, Loop is parallelizable
         Generating Tesla code
         16, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Loop and data clauses in vector addition in FORTRAN

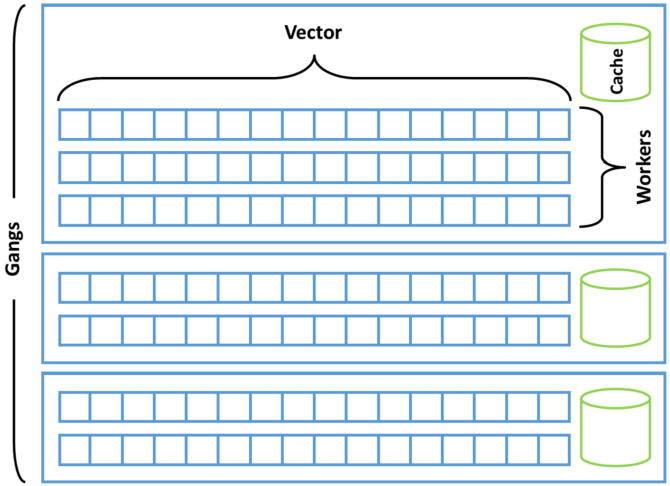
```
1   !! Vector_Addition.f90
2   module Vector_Addition_Mod
3     implicit none
4   contains
5     subroutine Vector_Addition(a, b, c, n)
6       !! Input vectors
7       real(8), intent(in), dimension(:) :: a,b
8       real(8), intent(out), dimension(:) :: c
9       integer :: i, n
10
11
12      do i = 1, n
13         c(i) = a(i) + b(i)
14      end do
15
16    end subroutine Vector_Addition
17  end module Vector_Addition_Mod
```

```
1   | !! Vector_Addition_OpenACC.f90
2   | module Vector_Addition_Mod
3   |   implicit none
4   | contains
5   |   subroutine Vector_Addition(a, b, c, n)
6   |     !! Input vectors
7   |     real(8), intent(in), dimension(:) :: a,b
8   |     real(8), intent(out), dimension(:) :: c
9   |     integer :: i, n
10  |     !$acc kernels loop copyin(a(1:n), b(1:n))
11  |     copyout(c(1:n))
12  |     do i = 1, n
13  |        c(i) = a(i) + b(i)
14  |     end do
15  |     !$acc end kernels
16  |   end subroutine Vector_Addition
17  | end module Vector_Addition_Mod
```

```
$ nvfortran -fast -Minfo=accel -acc=gpu -gpu=cc70 Vector_Addition_OpenACC.f90
vector_addition:
     12, Generating copyin(a(:n)) [if not already present]
         Generating copyout(c(:n)) [if not already present]
         Generating copyin(b(:n)) [if not already present]
     13, Loop is parallelizable
         Generating Tesla code
         13, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

Loop optimization

CUDA	OpenACC (2.7 API)
thread blocks	`num_gangs()`
warps	`num_workers()`
threads	`vector_length()`



- OpenACC provides a similar way of mapping the threads to SIMT architecture.
- As a programmer, we have the freedom to create a number of threads (using OpenACC APIs).
- A compiler chooses the good threads based on the given architecture and also depends on the problem.
- However, it is good to set it manually to optimize the code even further.

Example - Loop optimization

FORTRAN

```
!$acc data copyin(a(1:n), b(1:n)) copyout(c(1:n))
!$acc parallel loop num_gangs(128) vector_length(128)
    do i = 1, n
       c(i) = a(i) + b(i)
    end do
!$acc end parallel
!$acc end data
```

C/C++

```
#pragma acc data copyin(a[0:n], b[0:n]) copyout(c[0:n])
  {
#pragma acc parallel loop num_gangs(128) vector_length(128)
    for(int i = 0; i < n; i ++)
      {
        c[i] = a[i] + b[i];
      }
  }
```

Example - Loop optimization

FORTRAN

```
nvfortran -fast -Minfo=accel -acc=gpu -gpu=cc70 Vector_Addition_OpenACC_Optimized.f
vector_addition:
     12, Generating copyin(a(:n)) [if not already present]
         Generating copyout(c(:n)) [if not already present]
         Generating copyin(b(:n)) [if not already present]
     13, Generating Tesla code
         14, !$acc loop gang(128), vector(128) ! blockidx%x threadidx%x
```

C/C++

```
nvc -fast -Minfo=accel -acc=gpu -gpu=cc70 Vector_Addition_OpenACC_Optimized.c
Vector_Addition:
     16, Generating copyin(a[:n]) [if not already present]
         Generating copyout(c[:n]) [if not already present]
         Generating copyin(b[:n]) [if not already present]
         Generating Tesla code
         18, #pragma acc loop gang(128), vector(128) /* blockIdx.x threadIdx.x */
```

Profiling (without GUI)

- `export NVCOMPILER_ACC_TIME=1`
  - if you plan to use another profiling tool, please set `NVCOMPILER_ACC_TIME=0`
- `export NVCOMPILER_ACC_NOTIFY=3`
  - 1: kernel launches
  - 2: data transfers
  - 4: region entry/exit
  - 8: wait for operations or synchronizations
  - 16: device memory allocates and deallocates

Keeping `NVCOMPILER_ACC_NOTIFY=3` provides kernel executions and data transfer information.

```
/mnt/irisgpfs/users/ekrishnasamy/ULHPC-School/Vector_Addition_OpenACC.f90
  vector_addition  NVIDIA  devicenum=0
    time(us): 66
    12: compute region reached 1 time
        12: kernel launched 1 time
            grid: [128]  block: [128]
            elapsed time(us): total=45 max=45 min=45 avg=45
    12: data region reached 2 times
        12: data copyin transfers: 2
             device time(us): total=38 max=23 min=15 avg=19
        16: data copyout transfers: 1
             device time(us): total=28 max=28 min=28 avg=28
```

Reduction clause

- Useful for incrementing or summation of an array into a shared numerical variable.
- `reduction(operators: variable)`
  - arithmetic reductions: `+,*,max,min`
- For example, useful in matrix-matrix multiplication.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

reduction clause in C/C++

```c
// Matrix_Multiplication.c
for(int row = 0; row < width ; ++row)
    {
      for(int col = 0; col < width ; ++col)
        {
          sum=0;
          for(int i = 0; i < width ; ++i)
            {
              sum += a[row*width+i]
                   * b[i*width+col];
            }
          c[row*width+col] = sum;
        }
    }
```

```c
// Matrix_Multiplication_OpenACC.c
| pragma acc kernels loop collapse(2)
|                         reduction(+:sum)
|   for(int row = 0; row < width ; ++row)
|     {
|       for(int col = 0; col < width ; ++col)
|         {
|           sum=0;
|           for(int i = 0; i < width ; ++i)
|             {
|               sum += a[row*width+i]
|                    * b[i*width+col];
|             }
|           c[row*width+col] = sum;
|         }
|     }
```

```
nvc -fast -Minfo=accel -acc=gpu -gpu=cc70 Matrix_Multiplication_OpenACC.c
Matrix_Multiplication:
      9, Generating copyin(a[:width*width],b[:width*width]) [if not already present]
         Generating copyout(c[:width*width]) [if not already present]
         Generating create(sum) [if not already present]
     11, Loop carried dependence of c-> prevents parallelization
         Loop carried backward dependence of c-> prevents vectorization
     13, Loop is parallelizable
         Generating Tesla code
         11, #pragma acc loop seq collapse(2)
         13,   collapsed */
             Generating reduction(+:sum)
         16, #pragma acc loop vector(128) /* threadIdx.x */
             Generating implicit reduction(+:sum)
     16, Loop is parallelizable
```

reduction clause in FORTRAN

```
 1    !! Matrix_Multiplication.f90
 2        do row = 0, width-1
 3           do col = 0, width-1
 4              sum=0
 5              do i = 0, width-1
 6                 sum = sum + (a((row*width)+i+1)
 7                           * b((i*width)+col+1))
 8              enddo
 9              c(row*width+col+1) = sum
10           enddo
11        enddo
```

```
 1    !! Matrix_Multiplication_OpenACC.f90
 2    |   !$acc loop collapse(2) reduction(+:sum)
 3    |   do row = 0, width-1
 4    |      do col = 0, width-1
 5    |         sum=0
 6    |         do i = 0, width-1
 7    |            sum = sum + (a((row*width)+i+1)
 8    |                         * b((i*width)+col+1))
 9    |         enddo
10    |         c(row*width+col+1) = sum
11    |      enddo
12    |   enddo
13    |   !$acc end loop
```

```
nvfortran -fast -Minfo=accel -acc=gpu -gpu=cc70 Matrix_Multiplication_OpenACC.f90
matrix_multiplication:
     14, Generating copyin(a(:width*width),b(:width*width)) [if not already present]
         Generating copyout(c(:width*width)) [if not already present]
         Generating create(sum) [if not already present]
         Generating Tesla code
         16, !$acc loop gang collapse(2) ! blockidx%x
             Generating reduction(+:sum)
         17,   ! blockidx%x collapsed
         19, !$acc loop vector(128) ! threadidx%x
             Generating implicit reduction(+:sum)
     19, Loop is parallelizable
```

Practical session

- Try simple `Hello_World_OpenACC.c` and `Hello_World_OpenACC.f90` with OpenACC parallel/kernels constructs; and try to understand what compiler producing.
- Do simple `Vector_Addition_OpenACC.c` and `Vector_Addition_OpenACC.f90` with OpenACC parallel/kernels constructs and use data clauses for data management.
- Similarly, try `Vector_Addition_OpenACC.c` and `Vector_Addition_OpenACC.f90` with OpenACC parallel/kernels constructs and use data clauses for data management. And include thread clauses for creating threads.
- Finally, do `Matrix_Multiplication_OpenACC.c` and `Matrix_Multiplication_OpenACC.f90` and use reduction clause along with parallel/kernels constructs and data management clauses.
- Similarly include the thread blocks for `Matrix_Multiplication_OpenACC.c` and `Matrix_Multiplication_OpenACC.f90`.

Questions

Thank you!

Please contact me if there any questions:

ezhilmathi.krishnasamy@uni.lu