

Introduction to OpenACC and OpenMP GPU



1 Introduction

Definitions

Short history

Architecture

Execution models

Porting strategy

PGI compiler

Profiling the code

2 Parallelism

Offloaded Execution

Add directives

Compute constructs

Serial

Kernels/Teams

Parallel

Work sharing

Parallel loop

Reductions

Coalescing

Routines

Routines

Data Management

Asynchronism

3 GPU Debugging

4 Annex: Optimization example

5 Annex: OpenACC 2.7/OpenMP 5.0 translation table

6 Contacts

- 1 Introduction
 - Definitions
 - Short history
 - Architecture
 - Execution models
 - Porting strategy
 - PGI compiler
 - Profiling the code

- 2 Parallelism

- 3 GPU Debugging

- 4 Annex: Optimization example

- 5 Annex: OpenACC 2.7/OpenMP 5.0 translation table

- 6 Contacts


Introduction - Useful definitions

- Gang(OpenACC)/Teams(OpenMP) : Coarse-grain parallelism
- Worker(OpenACC) : Fine-grain parallelism
- Vector : Group of threads executing the same instruction (SIMT)
- Thread : Execution **entity**
- SIMT : Single Instruction Multiple Threads
- Device : Accelerator on which execution can be offloaded (ex : GPU)
- Host : Machine hosting 1 or more accelerators and in charge of execution control
- Kernel : Piece of code that runs on an accelerator
- Execution thread : Sequence of kernels to be executed on an accelerator

Introduction - Comparison of CPU/GPU cores

The number of compute cores on machines with GPUs is much greater than on a classical machine.


IDRIS' Jean-Zay has 2 partitions:

- Non-accelerated : $2 \times 20 = 40$ cores 
- Accelerated with 4 Nvidia V100 =

Introduction - Comparison of CPU/GPU cores

The number of compute cores on machines with GPUs is much greater than on a classical machine.

IDRIS' Jean-Zay has 2 partitions:


- Non-accelerated : $2 \times 20 = 40$ cores 
- Accelerated with 4 Nvidia V100 = 32

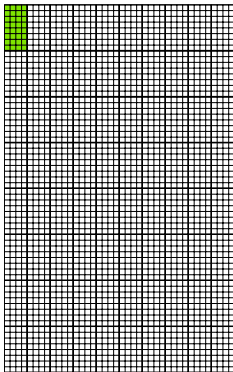


Introduction - Comparison of CPU/GPU cores

The number of compute cores on machines with GPUs is much greater than on a classical machine.

IDRIS' Jean-Zay has 2 partitions:


- Non-accelerated : $2 \times 20 = 40$ cores 
- Accelerated with 4 Nvidia V100 = 32×80

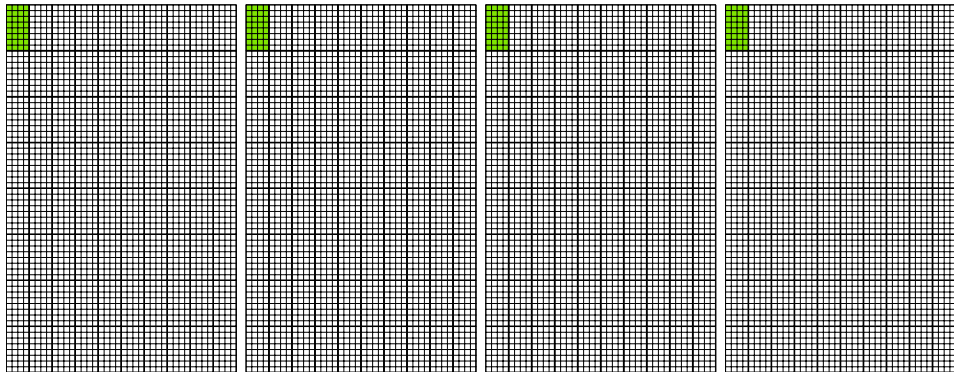


Introduction - Comparison of CPU/GPU cores

The number of compute cores on machines with GPUs is much greater than on a classical machine.

IDRIS' Jean-Zay has 2 partitions:

- Non-accelerated : $2 \times 20 = 40$ cores 
- Accelerated with 4 Nvidia V100 = $32 \times 80 \times 4 = 10240$ cores



Applications

Programming effort and technical expertise

Libraries

- cuBLAS
- cuSPARSE
- cuRAND
- AmgX
- MAGMA

- Minimum change in the code
- Maximum performance

Directives

- OpenACC
- OpenMP 5.0

- Portable, simple, low intrusiveness
- Still efficient

Programming languages

- CUDA
- OpenCL

- Complete rewriting, complex
- Non-portable
- Optimal performance

OpenACC

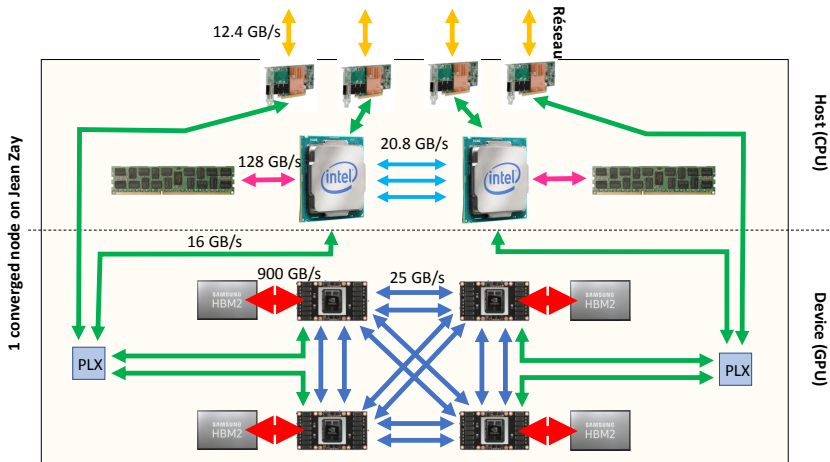
- <http://www.openacc.org/>
- Cray, NVidia, PGI, CAPS
- First standard 1.0 (11/2011)
- Latest standard 3.0 (11/2019)
- Main compilers :
 - PGI
 - Cray (for Cray hardware)
 - GCC (since 5.7)

OpenMP target

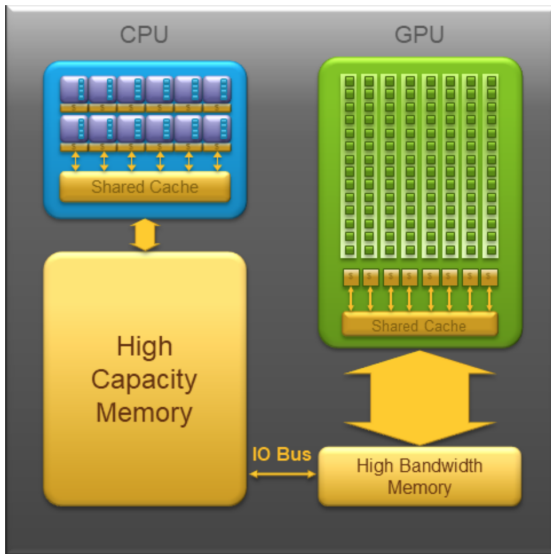
- <http://www.openmp.org/>
- First standard OpenMP 4.5 (11/2015)
- Latest standard OpenMP 5.0 (11/2018)
- Main compilers :
 - Cray (for Cray hardware)
 - GCC (since 7)
 - CLANG
 - IBM XL
 - PGI support announced

Introduction - Architecture of a converged node on Jean Zay

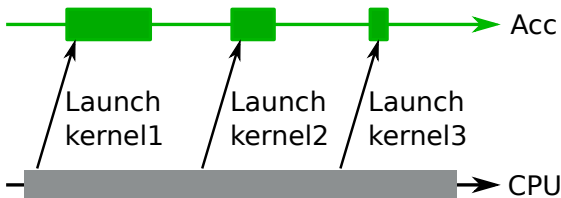
Architecture of a converged node on Jean Zay (40 cores, 192 GB of memory, 4 GPU V100)



Introduction - Architecture CPU-GPU



Execution controlled by the host



In OpenACC, the execution is controlled by the host (CPU). It means that kernels, data transfers and memory allocation are managed by the host.

Introduction - Execution model: OpenACC

OpenACC has 4 levels of parallelism for offloaded execution:

- Coarse grain: Gang
- Fine grain : worker
- Vectorization : vector
- Sequential : seq

By specifying clauses within OpenACC directives, your code will be run with a combination of the following modes :

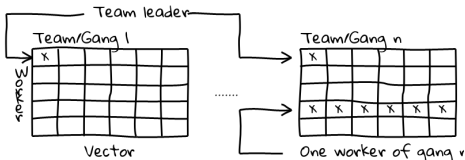
- Gang-Redundant (GR) : All gangs run the same instructions redundantly
- Gang-Partitioned (GP) : Work is shared between the gangs (ex : !\$ACC loop gang)
- Worker-Single (WS) : One worker is active in GP or GR mode
- Worker-Partitioned (WP) : Work is shared between the workers of a gang
- Vector-Single (VS) : One vector channel is active
- Vector-Partitioned (VP) : Several vector channels are active

These modes must be combined in order to get the best performance from the accelerator.

Introduction - Execution model

The execution of a kernel uses a set of threads that are mapped on the hardware resources of the accelerator. These threads are grouped within teams of the same size, with one master thread per team (this defines a gang). Each team is spread on a 2D thread-grid (worker x vector). One worker is actually a vector of $vector_length$ threads. So the total number of threads is

$$nb_{threads} = nb_{gangs} * nb_{workers} * vector_length$$



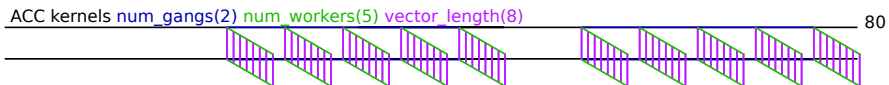
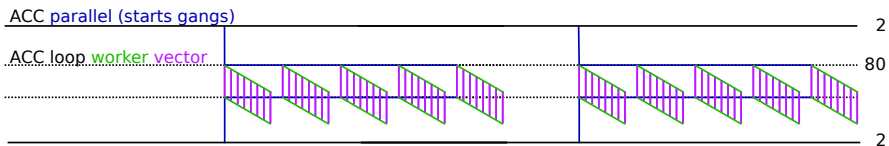
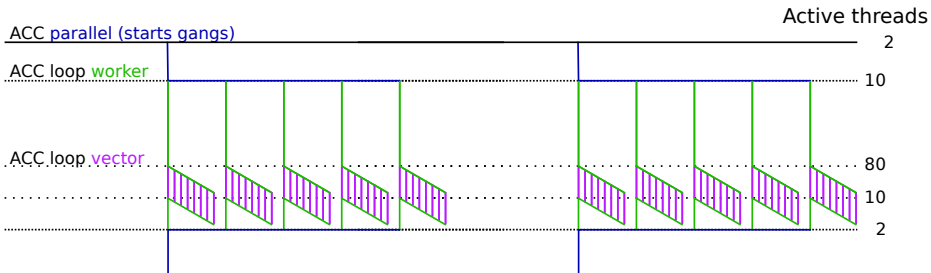
Important notes

- No synchronization is possible between gangs.
- The compiler can decide to synchronize the threads of a gang (all or part of them).
- The threads of a worker run in SIMT mode (all threads run the same instruction at the same time, for example on NVidia GPUs, groups of 32 threads are formed).

NVidia P100 restrictions on Gang/Worker/Vector

- The number of gangs is limited to $2^{31} - 1$.
- The thread-grid size (i.e. $nb_{workers} \times vector_{length}$) is limited to 1024.
- Due to register limitations the size of the grid should be less than or equal to 256 if the programmer wants to be sure that a kernel can be launched.
- The size of a worker ($vector_{length}$) should be a multiple of 32.
- PGI limitation: In a kernel that contains calls to external subroutines (not seq), the size of a worker is set at to 32 .

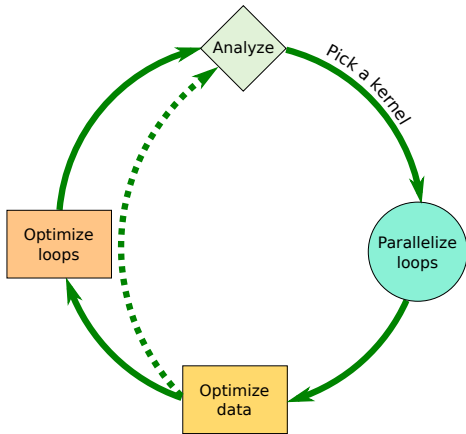
Introduction - Execution model



Introduction - Porting strategy

When using OpenACC, it is recommended to follow this strategy:

1. Identify the compute intensive loops (Analyze)
2. Add OpenACC directives (Parallelization)
3. Optimize data transfers and loops (Optimizations)
4. Repeat steps 1 to 3 until everything is on GPU



PGI compiler

During this training course, we will use PGI for the pieces of code containing OpenACC directives.

Information

- Company founded in 1989
- Acquired by NVidia in 2013
- Develops compilers, debuggers and profilers

Compilers

- Latest version: 19.10
- Hands-on version : 19.10
- C : pgcc
- C++ : pgc++
- Fortran : pgf90, pgfortran

Activate OpenACC

- -acc : Activates OpenACC support
- -ta=<options> : OpenACC options
- -Minfo=accel : **USE IT!** Displays information about compilation. The compiler will do implicit operations that you want to be aware of!

Tools

- nvprof : CPU/GPU profiler
- nvvp : nvprof GUI

Important options for *-ta*.

Compute capability

Each GPU generation has increased capabilities. For NVidia hardware, this is reflected by a number:

- K80 : cc35
- P100 : cc60
- V100 : cc70

For example, if you want to compile for V100 :

`-ta=tesla:cc70`

Complete documentation:

<https://www.pgroup.com/resources/docs/19.10/x86/pvf-user-guide/index.htm#ta>

Memory management

- pinned: The memory location on the host is pinned. It might improve data transfers.
- managed: The memory of both the host and the device(s) is unified.

PGI Compiler - Compiler information

Information available with *-Minfo=accel*.

```
$ pgfortran -O0 -acc -ta=tesla -Minfo=accel loop.f90
loop:
  4, Accelerator kernel generated
  Generating Tesla code
    5, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
  4, Generating implicit copyout(a(:))
```

```
1 program loop
  integer :: a(10000)
  integer :: i
  !$ACC parallel loop
  do i=1,10000
6     a(i) = i
  enddo
end program loop
```

../exemples/loop.f90

```
$ pgfortran -acc -ta=tesla -Minfo=accel reduction.f90
reduction:
  5, Accelerator kernel generated
  Generating Tesla code
    6, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
  5, Generating implicit copyout(a(:))
  9, Accelerator kernel generated
  Generating Tesla code
    9, Generating reduction(+:sum)
   10, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
  9, Generating implicit copy(sum)
  Generating implicit copyin(a(:))
```

```
program reduction
2  integer :: a(10000)
  integer :: i
  integer :: sum=0
  !$ACC parallel loop
  do i=1,10000
7     a(i) = i
  enddo
  !$ACC parallel loop reduction(+:sum)
  do i=1,10000
    sum = sum + a(i)
12  enddo
end program reduction
```

../exemples/reduction.f90

PGI Compiler - Compiler information

Information available with `-Minfo=accel`.

```
$ pgfortran -acc -ta=tesla -Minfo=accel reduction.f90
reduction:
  5, Accelerator kernel generated
  Generating Tesla code
  6, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
  5, Generating implicit copyout(a(:))
  9, Accelerator kernel generated
  Generating Tesla code
  9, Generating reduction(+:sum)
  10, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
  9, Generating implicit copy(sum)
  Generating implicit copyin(a(:))
```

```
2      !$ACC parallel loop
      do i=1,10000
          a(i) = i
      enddo
      !$ACC parallel loop reduction(+:sum)
      do i=1,10000
          sum = sum + a(i)
      enddo
7
```

`../exemples/reduction.f90`

To avoid CPU-GPU communication, a data region is added.

```
$ pgfortran -acc -ta=tesla -Minfo=accel reduction_data_region.f90
reduction_data:
  5, Generating create(a(:))
  6, Accelerator kernel generated
  Generating Tesla code
  7, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
  10, Accelerator kernel generated
  Generating Tesla code
  10, Generating reduction(+:sum)
  11, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
  10, Generating implicit copy(sum)
```

```
3      !$ACC data create(a(1:10000))
      !$ACC parallel loop
      do i=1,10000
          a(i) = i
      enddo
      !$ACC parallel loop reduction(+:sum)
      do i=1,10000
          sum = sum + a(i)
      enddo
8      !$ACC end data
```

`../exemples/reduction_data_region.f90`

Analysis - Code profiling

Here are 3 tools available to profile your code:

PGI_ACC_TIME

- Command line tool
- Gives basic information about:
 - Time spent in kernels
 - Time spent in data transfers
 - How many times a kernel is executed
 - The number of gangs, workers and vector size mapped to hardware

nvprof

- Command line tool
- Options which can give you a fine view of your code

nvvp/pgprof/NSight Graphics Profiler

- Graphical interface for nvprof

Analysis - Code profiling : PGI_ACC_TIME

For codes generated with PGI compilers, it is possible to have an automatic profiling of the code by setting the following environment variable: PGI_ACC_TIME=1
The *grid* is the number of gangs. The *block* is the size of one gang ($[vector_length \times nb_workers]$).

```
$ export PGI_ACC_TIME=1
$ ./a.out
a(1,1):          0 a(1,150)          42 a(1,200):          0

Accelerator Kernel Timing data
/gpfs16l/pwrwork/idris/sos/ssos013/gpu-pragma/exemples/arrayshape.f90
arrayshape  NVIDIA  devicenum=0
time(us): 1,432
 7: compute region reached 1 time
 7: kernel launched 1 time
    grid: [1000] block: [128]
    device time(us): total=175 max=175 min=175 avg=175
    elapsed time(us): total=1,208 max=1,208 min=1,208 avg=1,208
 7: data region reached 2 times
 7: data copyin transfers: 1
    device time(us): total=542 max=542 min=542 avg=542
 14: data copyout transfers: 1
    device time(us): total=543 max=543 min=543 avg=543
 14: compute region reached 1 time
 14: kernel launched 1 time
    grid: [1000] block: [128]
    device time(us): total=25 max=25 min=25 avg=25
    elapsed time(us): total=907 max=907 min=907 avg=907
 14: data region reached 2 times
 14: data copyin transfers: 1
    device time(us): total=84 max=84 min=84 avg=84
 20: data copyout transfers: 1
    device time(us): total=63 max=63 min=63 avg=63

$
```


Analysis - Code profiling: nvprof

Nvidia provides a command line profiler: nvprof

It gives the time spent in the kernels and data transfers for all GPU regions.

Caution : PGI_ACC_TIME and nvprof are incompatible. Ensure that PGI_ACC_TIME=0 before using the tool.

```
$ nvprof <options> executable_file <arguments of executable file>
```

	Type	Time(%)	Time	Calls	Avg	Min	Max	Name
1	GPU activities :	53.51%	2.70662s	500	5.4132ms	448ns	6.8519ms	[CUDA memcpy HtoD]
		36.72%	1.85742s	300	6.1914ms	384ns	9.2907ms	[CUDA memcpy DtoH]
		6.34%	320.52ms	100	3.2052ms	3.1372ms	3.6247ms	gol_39_gpu
		2.62%	132.39ms	100	1.3239ms	1.2727ms	1.3659ms	gol_33_gpu
6		0.81%	40.968ms	100	409.68us	401.22us	417.25us	gol_52_gpu
		0.01%	359.27us	100	3.5920us	3.3920us	4.0640us	gol_55_gpu__red

```
../exemples/profils/gol_opti2.prof
```

Important notes

- Only the GPU is profiled by default
- Option “--cpu-profiling on” activates CPU profiling
- Options “--metrics flop_count_dp --metrics dram_read_throughput --metrics dram_write_throughput” give respectively, the number of operations, memory read throughput and memory write throughput for each kernel running on the CPU.
- To get a file readable by nvvp you have to specify “-o filename” (add -f in case you want to overwrite a file)

Analysis - Graphical profiler: nvvp

Nvidia also provides a GUI for nvprof: nvvp.

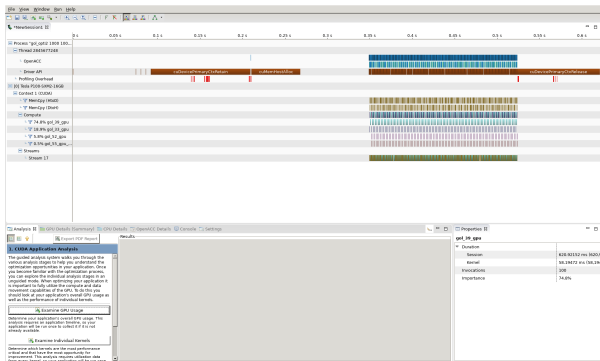
It gives the time spent in the kernels and data transfers for all GPU regions.

Caution : PGI_ACC_TIME and nvprof are incompatible. Ensure that PGI_ACC_TIME=0 before using the tool.

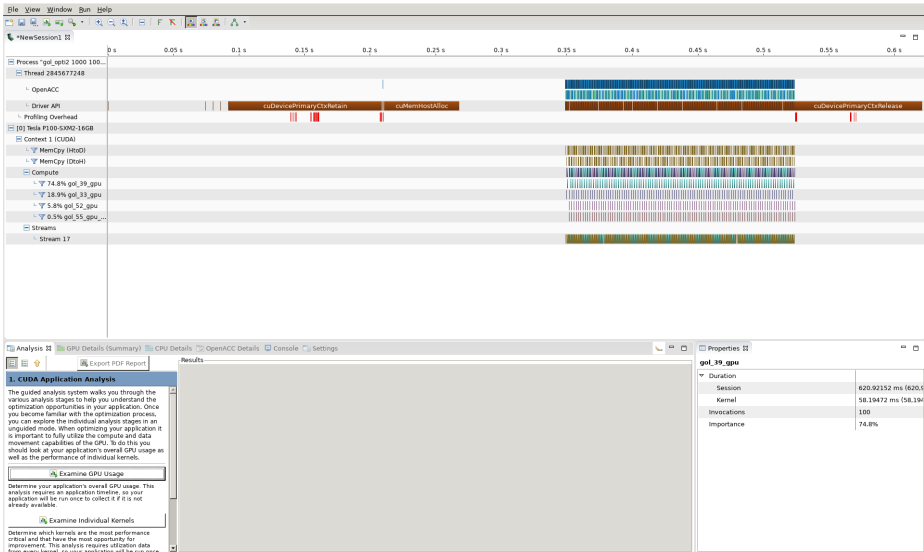
```
$ nvvp <options> executable <--args arguments of executable>
```

or if you want to open a profile generated with nvprof:

```
$ nvvp <options> prog.profile
```



Analysis - Graphical profiler : NVVP



1 Introduction

2 Parallelism

Offloaded Execution

- Add directives
- Compute constructs
- Serial
- Kernels/Teams
- Parallel

Work sharing

- Parallel loop
- Reductions
- Coalescing

Routines

- Routines

Data Management

Asynchronism

3 GPU Debugging

4 Annex: Optimization example

5 Annex: OpenACC 2.7/OpenMP 5.0 translation table

6 Contacts

Add directives

To activate OpenACC or OpenMP features, you need to add directives. If the right compiler options are not set, the directives are treated as comments. The syntax is different for a Fortran or a C/C++ code.

Fortran OpenACC

```
!$ACC directive <clauses>  
..  
!$ACC end directive
```

C/C++ OpenACC

```
#pragma acc directive <clauses>  
{  
..  
}
```

Some examples

kernels, loop, parallel, data, enter data, exit data
In this presentation, the examples are in Fortran.

Execution offloading

To run on the device you need to specify one of the 3 following directives (called compute construct):

serial

The code runs one single thread (1 gang with 1 worker of size 1). Only one kernel is generated.

kernels

The compiler analyzes the code and decides where parallelism is generated.

One kernel is generated for each parallel loop enclosed in the region.

parallel

The programmer creates a parallel region that runs on the device. Only one kernel is generated. The execution is redundant by default.

- Gang-redundant
- Worker-Single
- Vector-Single

The programmer has to share the work manually.

Offloaded execution - Directive *serial*

The directive *serial* tells the compiler that the enclosed region of code have to be offloaded to the GPU.

Instructions are executed only by one thread. This means that one gang with one worker of size one is generated.

This is equivalent to a parallel region with the following parameters: `num_gangs(1)`, `num_workers(1)`, `vector_length(1)`.

Data : Default behavior

- Arrays present in the *serial* region not specified in a data clause (*present*, *copyin*, *copyout*, etc) or a *declare* directive are assigned to a *copy*. They are *SHARED*.
- Scalar variables are implicitly assigned to a *firstprivate* clause. They are *PRIVATE*.

Fortran

```
1 !$ACC serial
do i=1, n
  do j=1, n
    ...
  enddo
6
  do j=1, n
    ...
  enddo
enddo
1 !$ACC end serial
```

Offloaded execution - Directive *serial*

```
!$ACC serial
do g=1,generations
do r=1, rows
do c=1, cols
35   oworld(r,c) = world(r,c)
enddo
enddo
do r=1, rows
do c=1, cols
40   neigh = oworld(r-1,c-1)+oworld(r,c-1)+oworld(r+1,c-1)+&
           oworld(r-1,c) +oworld(r+1,c)&
           oworld(r-1,c+1)+oworld(r,c+1)+oworld(r+1,c+1)
   if (oworld(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
45   else if (neigh == 3) then
       world(r,c) = 1
   endif
enddo
enddo
50 cells = 0
do r=1, rows
do c=1, cols
       cells = cells + world(r,c)
enddo
enddo
55 print *, "Cells alive at generation ", g, ": ", cells
enddo
!$ACC end serial
```

../exemples/gol_serial.f90

```
gol:
31, Accelerator serial kernel generated
Generating Tesla code
32, !$acc do seq
33, !$acc do seq
34, !$acc do seq
38, !$acc do seq
39, !$acc do seq
51, !$acc do seq
52, !$acc do seq
31, Generating implicit copy(world(1:rows,1:cols))
Generating implicit copyin(old_world(0:rows+1,0:cols+1))
Generating implicit copyout(old_world(1:rows,1:cols))
```

Test

- Size: 1000x1000
- Generations: 100
- Elapsed time: 123.640 s

Execution offloading - Kernels : In the compiler we trust

The *kernels* directive tells the compiler that the region contains instructions to be offloaded on the device.

Each loop nest will be treated as an independant kernel with its own parameters (number of gangs, workers and vector size).

Data : Default behavior

- Data arrays present inside the *kernels* region and not specified inside a data clause (*present*, *copyin*, *copyout*, etc) or inside a *declare* are assigned to a *copy* clause. They are *SHARED*.
- Scalar variables are implicitly assigned to a *copy* clause. They are *SHARED*.

Fortran

```
1 !$ACC kernels
2 do i=1, n
3   do j=1, n
4     ...
5   enddo
6
7   do j=1, n
8     ...
9   enddo
10 enddo
11 !$ACC end kernels
12
```

Important notes

- The parameters of the parallel regions are independent (gangs, workers, vector size).
- Loop nests are executed consecutively.

Execution offloading - Kernels

```
!$ACC kernels
do g=1,generations
do r=1, rows
do c=1, cols
oworld(r,c) = world(r,c)
enddo
enddo
do r=1, rows
do c=1, cols
neigh = oworld(r-1,c-1)+oworld(r,c-1)+oworld(r+1,c-1)+&
oworld(r-1,c) +oworld(r+1,c)+&
oworld(r-1,c+1)+oworld(r,c+1)+oworld(r+1,c+1)
if (oworld(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
world(r,c) = 0
else if (neigh == 3) then
world(r,c) = 1
endif
enddo
enddo
cells = 0
do r=1, rows
do c=1, cols
cells = cells + world(r,c)
enddo
enddo
print *, "Cells alive at generation ", g, ": ", cells
enddo
!$ACC end kernels
```

../examples/gol_kernels.f90

```
gol:
31, Generating implicit copy(world(1:rows,1:cols))
Generating implicit copyin(old_world(0:rows+1,0:cols+1))
Generating implicit copyout(old_world(1:rows,1:cols))
32, Loop carried dependence due to exposed use of old_world(0:
rows+1,0:cols+1),world(1:rows,1:cols) prevents parallelization
Accelerator kernel generated
Generating Tesla code
32, !$acc loop seq
33, !$acc loop vector(128) ! threadidx%x
34, !$acc loop seq
38, !$acc loop vector(128) ! threadidx%x
39, !$acc loop seq
51, !$acc loop vector(128) ! threadidx%x
52, !$acc loop seq
53, Generating implicit reduction(+:cells)
33, Loop is parallelizable
34, Loop is parallelizable
38, Loop is parallelizable
39, Loop is parallelizable
51, Loop is parallelizable
52, Loop is parallelizable
```

Test

- Size: 1000x1000
- Generations: 100
- Execution time: 1.951 s

Execution offloading - *parallel* directive

The *parallel* directive opens a parallel region on the device and generates one or more gangs. All gangs execute redundantly the instructions met in the region (gang-redundant mode). Only parallel loop nests with a *loop* directive might have their iterations spread among gangs.

Data: default behavior

- Data arrays present inside the region and not specified in a data clause (*present*, *copyin*, *copyout*, *etc*) or a *declare* directive are assigned to a *copy* clause. They are *SHARED*.
- Scalar variables are implicitly assigned to a *firstprivate* clause. They are *PRIVATE*.

```
!$ACC parallel
2a = 2 !!!! Gang-redundant
!$ACC loop !!!! Work sharing
do i = 1, n
...
enddo
7!$ACC end parallel
```

Important notes

- The number of gangs, workers and the vector size are constant inside the region.

Execution offloading - *parallel* directive

```
!$ACC parallel
do g=1,generations
do r=1, rows
do c=1, cols
35 oworld(r,c) = world(r,c)
enddo
enddo
do r=1, rows
do c=1, cols
40 neigh = oworld(r-1,c-1)+oworld(r,c-1)+oworld(r+1,c-1)+&
oworld(r-1,c) +oworld(r+1,c)&
oworld(r-1,c+1)+oworld(r,c+1)+oworld(r+1,c+1)
if (oworld(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
45 else if (neigh == 3) then
world(r,c) = 0
world(r,c) = 1
endif
enddo
enddo
50 cells = 0
do r=1, rows
do c=1, cols
cells = cells + world(r,c)
enddo
enddo
55 print *, "Cells alive at generation ", g, " : ", cells
enddo
!$ACC end parallel
```

../exemples/gol_parallel.f90

gol:

```
31, Accelerator kernel generated
Generating Tesla code
32, !$acc loop seq
33, !$acc loop seq
34, !$acc loop seq
38, !$acc loop seq
39, !$acc loop seq
51, !$acc loop seq
52, !$acc loop seq
31, Generating implicit copy(world(1:rows,1:cols))
Generating implicit copyin(old_world(0:rows+1,0:cols+1))
Generating implicit copyout(old_world(1:rows,1:cols))
32, Loop carried dependence due to exposed use of world(1:rows
,1:cols),old_world(0:rows+1,0:cols+1) prevents parallelization
33, Loop is parallelizable
34, Loop is parallelizable
38, Loop is parallelizable
39, Loop is parallelizable
51, Loop is parallelizable
52, Loop is parallelizable
```

acc=noautopar

Test

- Size: 1000x1000
- Generations : 100
- Execution time : 120.467 s (noautopar)
- Execution time : 5.067 s (autopar)

The sequential code is executed redundantly by all gangs.

The compiler option `acc=noautopar` is activated to reproduce the expected behavior of the OpenACC specification.

Execution offloading - *parallel* directive

```
gol:
  31, Accelerator kernel generated
  Generating Tesla code
  32, !$acc loop seq
  33, !$acc loop seq
  34, !$acc loop vector(128) ! threadidx%x
  38, !$acc loop seq
  39, !$acc loop vector(128) ! threadidx%x
  51, !$acc loop seq
  52, !$acc loop vector(128) ! threadidx%x
  53, Generating implicit reduction(+:cells)
  31, Generating implicit copy(world(1:rows,1:cols))
  Generating implicit copyin(old_world(0:rows+1,0:cols+1))
  Generating implicit copyout(old_world(1:rows,1:cols))
  32, Loop carried dependence due to exposed use of world(1:rows
,1:cols),old_world(0:rows+1,0:cols+1) prevents parallelization
  33, Loop is parallelizable
  34, Loop is parallelizable
  38, Loop is parallelizable
  39, Loop is parallelizable
  51, Loop is parallelizable
  52, Loop is parallelizable
```

acc=autopar activated

```
gol:
  31, Accelerator kernel generated
  Generating Tesla code
  32, !$acc loop seq
  33, !$acc loop seq
  34, !$acc loop seq
  38, !$acc loop seq
  39, !$acc loop seq
  51, !$acc loop seq
  52, !$acc loop seq
  31, Generating implicit copy(world(1:rows,1:cols))
  Generating implicit copyin(old_world(0:rows+1,0:cols+1))
  Generating implicit copyout(old_world(1:rows,1:cols))
  32, Loop carried dependence due to exposed use of world(1:rows
,1:cols),old_world(0:rows+1,0:cols+1) prevents parallelization
  33, Loop is parallelizable
  34, Loop is parallelizable
  38, Loop is parallelizable
  39, Loop is parallelizable
  51, Loop is parallelizable
  52, Loop is parallelizable
```

acc=noautopar

Test

- Size: 1000x1000
- Generations : 100
- Execution time : 120.467 s (noautopar)
- Execution time : 5.067 s (autopar)

The sequential code is executed redundantly by all gangs.

The compiler option `acc=noautopar` is activated to reproduce the expected behavior of the OpenACC specification.

Parallelization control

The default behavior is to let the compiler decide how many workers are generated and their vector size. The number of gangs is set at execution time by the runtime (memory is usually the limiting criterion).

Nonetheless the programmer might set those parameters inside *kernels* and *parallel* directives with clauses:

- *num_gangs* : The number of gangs
- *num_workers* : The number of workers
- *vector_length* : The vector size

```
program param
2   integer :: a(1000)
   integer :: i

   !$ACC parallel num_gangs(10) num_workers(1) &
   !$ACC& vector_length(128)
7   print *, "Hello I am a gang"
   do i=1,1000
       a(i) = i
   enddo
   !$ACC end parallel
12end program
```

../examples/parametres_paral.f90

Important notes

- These clauses are mainly useful if the code uses a data structure which is difficult for the compiler to analyze.
- The optimal number of gangs is highly dependent on the architecture. Use *num_gangs* with care.

Work sharing - Loops

Loops are at the heart of OpenACC parallelism. The *loop* directive, is responsible for sharing the work (i.e. the iterations of the associated loop).

It could also activate another level of parallelism. This point (automatic nesting) is a critical difference between OpenACC and OpenMP-GPU for which the creation of threads relies on the programmer.

Clauses

- Level of parallelism :
 - *gang*, the iterations of the subsequent loop are distributed block-wise among gangs (going from GR to GP)
 - *worker*, within gangs, the worker's threads are activated (going from WS to WP) and the iterations are shared between those threads.
 - *vector*, each worker activates its SIMT threads (going from VS to VP) and the work is shared between those threads.
 - *seq*, the iterations are executed sequentially on the accelerator
 - *auto*, The compiler analyses the subsequent loop and decides which mode is the most suitable to respect dependancies.

	Work sharing (loop iterations)	Activation of new threads?
<i>loop gang</i>	X	
<i>loop worker</i>	X	X
<i>loop vector</i>	X	X

Clauses

- Merge tightly nested loops: *collapse(#loops)*
- Tell the compiler that iterations are independent: *independent* (useful for *kernels*)
- Privatize variables : *private(variable-list)*
- Reduction : *reduction(operation:variable-list)*

Notes

- You might see the directive *do* for compatibility.

Work sharing - Loops

```
!$ACC parallel
do g=1,generations
!$ACC loop
do r=1, rows
do c=1, cols
35   oworld(r,c) = world(r,c)
enddo
enddo
do r=1, rows
40 do c=1, cols
    neigh = oworld(r-1,c-1)+oworld(r,c-1)+oworld(r+1,c-1)&
            oworld(r-1,c) +oworld(r+1,c)&
            oworld(r-1,c+1)+oworld(r,c+1)+oworld(r+1,c+1)
    if (oworld(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
45       world(r,c) = 0
    else if (neigh == 3) then
        world(r,c) = 1
    endif
enddo
enddo
50 cells = 0
do r=1, rows
do c=1, cols
    cells = cells + world(r,c)
55 enddo
enddo
print *, "Cells alive at generation ", g, ":", cells
enddo
!$ACC end parallel
```

../exemples/gol_parallel_loop.f90

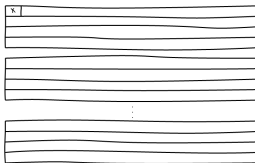
```
gol:
31, Accelerator kernel generated
Generating Tesla code
32, !$acc loop seq
34, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
35, !$acc loop seq
39, !$acc loop seq
40, !$acc loop seq
52, !$acc loop seq
53, !$acc loop seq
31, Generating implicit copy(world(1:rows,1:cols))
Generating implicit copyin(old_world(0:rows+1,0:cols+1))
Generating implicit copyout(old_world(1:rows,1:cols))
32, Loop carried dependence due to exposed use of world(1:rows
,1:cols),old_world(0:rows+1,0:cols+1) prevents parallelization
35, Loop is parallelizable
39, Loop is parallelizable
40, Loop is parallelizable
52, Loop is parallelizable
53, Loop is parallelizable
```

Work sharing - Loops

serial

```
!$acc serial
do i=1,nx
  A(i)=B(i)*i
end do
!$acc end serial
```

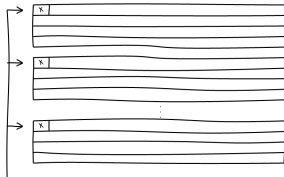
../exemples/loop_serial.f90



- Active threads : 1
- Number of operations : nx

Execution GRWSVS

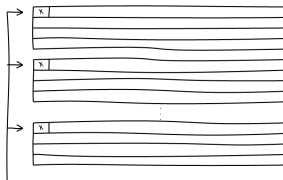
```
!$acc parallel num_gangs(10)
do i=1,nx
  A(i)=B(i)*i
end do
!$acc end parallel
../exemples/loop_GRWSVS.f90
```



- Redundant execution by gang leaders
- Active threads : 10
- Number of operations : $10 * nx$

Execution GPWSVS

```
!$acc parallel num_gangs(10)
!$acc loop gang
do i=1,nx
  A(i)=B(i)*i
end do
!$acc end parallel
../exemples/loop_GPWSVS.f90
```

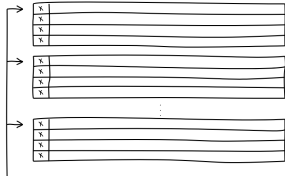


- Each gang executes a different block of iterations.
- Active threads: 10
- Number of operations : nx

Execution GPWPVS

```

0 !$acc parallel num_gangs(10)
  !$acc loop gang worker
  do i=1,nx
    A(i)=B(i)*i
  end do
15 !$acc end parallel
  ..examples/loop_GPWPVS.f90
  
```

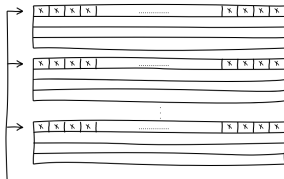


- Iterations are shared among the active workers of each gang.
- Active threads: $10 * \text{\#workers}$
- Number of operations : nx

Execution GPWSVP

```

0 !$acc parallel num_gangs(10)
  !$acc loop gang vector
  do i=1,nx
    A(i)=B(i)*i
  end do
15 !$acc end parallel
  ..examples/loop_GPWSVP.f90
  
```

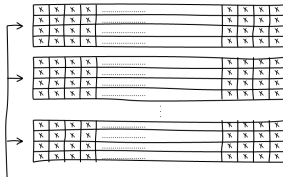


- Active threads: $10 * \text{\#workers}$
- Iterations are shared among the threads of the worker of all gangs
- Active threads: $10 * \text{vector_length}$
- Number of operations : nx

Execution GPWPVP

```

0 !$acc parallel num_gangs(10)
  !$acc loop gang worker vector
  do i=1,nx
    A(i)=B(i)*i
  end do
15 !$acc end parallel
  ..examples/loop_GPWPVP.f90
  
```



- Iterations are shared on the grid of threads of all gangs.
- Active threads: $10 * \text{\#workers} * \text{vector_length}$
- Number of operations : nx

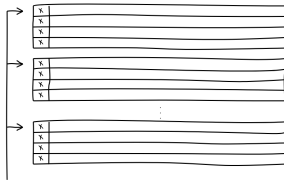
Execution GRWPVS

```

0  !$acc parallel num_gangs(10)
   !$acc loop worker
   do i=1,nx
     A(i)=B(i)*i
   end do
15 !$acc end parallel

```

../exemples/loop_GRWPVS.f90



- Each gang is assigned all the iterations which are shared among the workers.
- Active threads: $10 * \text{\#workers}$
- Number of operations : $10 * nx$

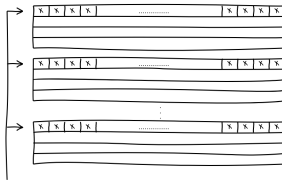
Execution GRWSVP

```

0  !$acc parallel num_gangs(10)
   !$acc loop vector
   do i=1,nx
     A(i)=B(i)*i
   end do
15 !$acc end parallel

```

../exemples/loop_GRWSVP.f90



- Each gang is assigned all the iterations which are shared on the threads of the active worker.
- Active threads: $10 * \text{vector_length}$
- Number of operations : $10 * nx$

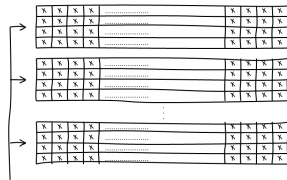
Execution GRWPVP

```

0  !$acc parallel num_gangs(10)
   !$acc loop worker vector
   do i=1,nx
     A(i)=B(i)*i
   end do
15 !$acc end parallel

```

../exemples/loop_GRWPVP.f90



- Each gang is assigned all the iterations and the grid of threads distributes the work.
- Active threads: $10 * \text{\#workers} * \text{vector_length}$
- Number of operations : $10 * nx$

Work sharing - Loops

Reminder: There is no thread synchronization at gang level, especially at the end of a *loop* directive. There is a risk of *race condition*. Such risk does not exist for *loop* with *worker* and/or *vector* parallelism since the threads of a gang wait until the end of the iterations they execute to start a new portion of the code after the loop.

Inside a parallel region, if several parallel loops are present and there are some dependencies between them, then you must not use *gang* parallelism.

```
10 !$acc parallel
   !$acc loop gang
   do i=1,nx
     A(i)=1.0_8
   end do
15 !$acc loop gang reduction(+:somme)
   do i=nx,1,-1
     somme=somme+A(i)
   end do
   !$acc end parallel
```

../exemples/loop_pb_sync.f90

- Result : sum = 97845213 (Wrong value)

```
10 !$acc parallel
   !$acc loop worker vector
   do i=1,nx
     A(i)=1.0_8
   end do
15 !$acc loop worker vector reduction(+:somme)
   do i=nx,1,-1
     somme=somme+A(i)
   end do
   !$acc end parallel
```

../exemples/loop_pb_sync_corr.f90

- Result : sum = 100000000 (Right value)

Parallelism - Merging *kernels/parallel* directives and *loop*

It is quite common to open a parallel region just before a *loop*. In this case it is possible to merge both directives into a *kernels loop* or *parallel loop*. The clauses available for this construct are those of both constructs.

For example, with *kernels* directive:

```
program fused
  integer :: a(1000), b(1000), i
  !$ACC kernels <clauses>
  !$ACC loop <clauses>
5  do i=1,1000
    a(i) = b(i)*2
  enddo
  !$ACC end kernels
! The construct above is equivalent to
10 !$ACC kernels loop <kernels and loop clauses>
  do i=1,1000
    a(i) = b(i)*2
  enddo
end program fused
```

../exemples/fused.f90

Parallelism - Reductions

A variable assigned to a *reduction* is privatised for each element of the parallelism level of the loop. At the end of the region, an operation is executed among those described in table 1 to get the final value of the variable.

Operations

Operation	Effect	Language(s)	Operation	Effect	Language(s)
+	Sum	Fortran/C(++)	iand	Bitwise and	Fortran
*	Product	Fortran/C(++)	ior	Bitwise or	Fortran
max	Maximum	Fortran/C(++)	ieor	Bitwise xor	Fortran
min	Maximum	Fortran/C(++)	.and.	Logical and	Fortran
&	Bitwise and	C(++)	.or.	Logical or	Fortran
	Bitwise or	C(++)			
&&	Logical and	C(++)			
	Logical or	C(++)			

Reduction operators

Restrictions

The variable must be a scalar with numerical value C(char, int, float, double, _Complex), C++(char, wchar_t, int, float, double), Fortran(integer, real, double precision, complex).

In the OpenACC specification 2.7, reductions are possible on arrays but the implementation is lacking in PGI (for the moment).

Parallelism - Reductions : example

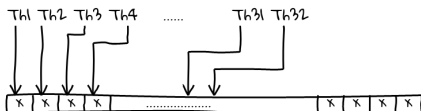
```
!$ACC parallel
do g=1,generations
!$ACC loop
do r=1, rows
35 do c=1, cols
    oworld(r,c) = world(r,c)
    enddo
enddo
do r=1, rows
40 do c=1, cols
    neigh = oworld(r-1,c-1)+oworld(r,c-1)+oworld(r+1,c-1)+&
            oworld(r-1,c) +oworld(r+1,c)&
            oworld(r-1,c+1)+oworld(r,c+1)+oworld(r+1,c+1)
    if (oworld(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
45 world(r,c) = 0
    else if (neigh == 3) then
        world(r,c) = 1
    endif
enddo
enddo
50 cells = 0
!$ACC loop reduction(+:cells)
do r=1, rows
do c=1, cols
55 cells = cells + world(r,c)
enddo
enddo
print *, "Cells alive at generation ", g, " : ", cells
enddo
60 !$ACC end parallel
```

../exemples/gol_parallel_loop_reduction.f90

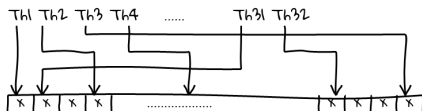
Parallelism - Memory accesses and coalescing

Principles

- Contiguous access to memory by the threads of a worker can be merged. This optimizes the use of memory bandwidth.
- This happens if thread i reaches memory location n , and thread $i + 1$ reaches memory location $n + 1$ and so on.
- For loop nests, the loop which has *vector* parallelism should have contiguous access to memory.



Example with «Coalescing»



Example with «noCoalescing»

Parallelism - Memory accesses and coalescing

```
10 !$acc parallel  
11 !$acc loop gang  
12 do i=1,nx  
13     !$acc loop vector  
14     do j=1, nx  
15         A(i, j)=1.14_8  
16     end do  
17 end do  
18 !$acc end parallel
```

../exemples/loop_nocoalescing.f90

```
20 !$acc parallel  
21 !$acc loop gang  
22 do j=1,nx  
23     !$acc loop vector  
24     do i=1, nx  
25         A(i, j)=1.14_8  
26     end do  
27 end do  
28 !$acc end parallel
```

../exemples/loop_coalescing.f90

```
30 !$acc parallel  
31 !$acc loop gang vector  
32 do i=1,nx  
33     !$acc loop seq  
34     do j=1, nx  
35         A(i, j)=1.14_8  
36     end do  
37 end do  
38 !$acc end parallel
```

../exemples/loop_coalescing.f90

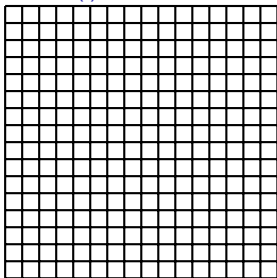
	Without Coalescing	With Coalescing
Tps (ms)	439	16

The memory coalescing version is 27 times faster !

Parallelism - Memory accesses and coalescing

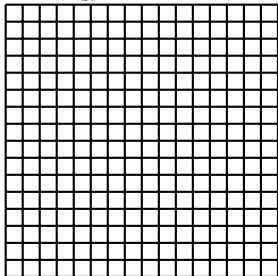
With coalescing :

gang(j)/vector(i)

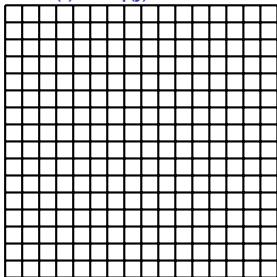


Without coalescing :

gang(i)/vector(j)



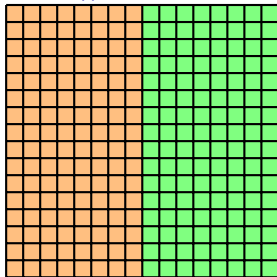
gang-vector(i)/seq(j)



Parallelism - Memory accesses and coalescing

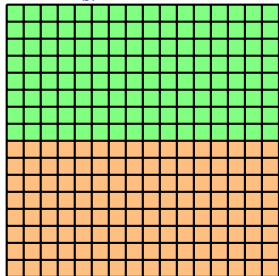
With coalescing :

gang(j)/vector(i)

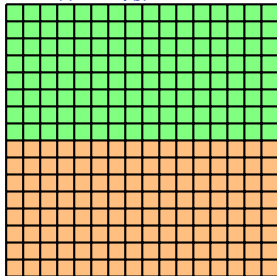


Without coalescing :

gang(i)/vector(j)



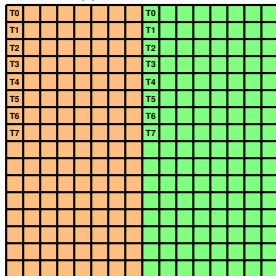
gang-vector(i)/seq(j)



Parallelism - Memory accesses and coalescing

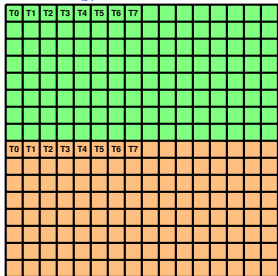
With coalescing :

gang(j)/vector(i)

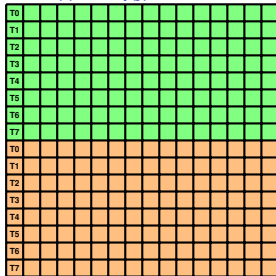


Without coalescing :

gang(i)/vector(j)



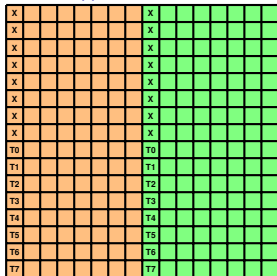
gang-vector(i)/seq(j)



Parallelism - Memory accesses and coalescing

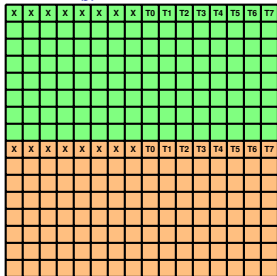
With coalescing :

gang(j)/vector(i)

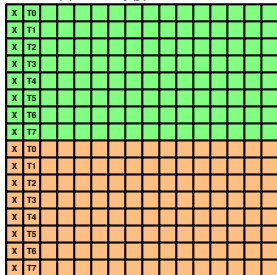


Without coalescing :

gang(i)/vector(j)



gang-vector(i)/seq(j)



Parallelism - Routines

If an offloaded region contains a call to a function or a subroutine, it is necessary to declare it as *routine*. It gives the information to the compiler that a device version of the function/subroutine has to be generated. It is mandatory to set the parallelism level inside the function (*seq*, *gang*, *worker*, *vector*).

Without ACC *routine*:

```
program routine
  integer :: s = 10000
  integer, allocatable :: array(:, :)
  allocate(array(s,s))
  !$ACC parallel
  !$ACC loop
  do i=1, s
  call fill(array(:, :), s, i)
  enddo
  !$ACC end parallel
  print *, array(1,10)
  contains
  subroutine fill(array, s, i)
    integer, intent(out) :: array(:, :)
    integer, intent(in) :: s, i
    integer :: j
    do j=1, s
      array(i, j) = 2
    enddo
  end subroutine fill
end program routine
```

../exemples/routine_wrong.f90

```
$ pgf90 -ta=tesla routine.f90 -Minfo=accel
PGF90-S-0155-Procedures called in a compute region must have acc routine information: fill (routine.f90: 8)
PGF90-S-0155-Accelerator region ignored; see -Minfo messages (routine.f90: 5)
routine:
  5, Accelerator region ignored
  8, Accelerator restriction: call to 'fill' with no acc routine information
  0 inform, 0 warnings, 2 severe, 0 fatal for routine
```


Parallelism - Routines

If an offloaded region contains a call to a function or a subroutine, it is necessary to declare it as *routine*. It gives the information to the compiler that a device version of the function/subroutine has to be generated. It is mandatory to set the parallelism level inside the function (*seq*, *gang*, *worker*, *vector*).

Correct version :

```
program routine
  integer :: s = 10000
  integer, allocatable :: array(:, :)
  allocate(array(s,s))
  !$ACC parallel copyout(array)
  !$ACC loop
  do i=1, s
    call fill(array(:, :), s, i)
  enddo
  !$ACC end parallel
  print *, array(1,10)
  contains
  subroutine fill(array, s, i)
    !$ACC routine seq
    integer, intent(out) :: array(:, :)
    integer, intent(in) :: s, i
    integer :: j
    do j=1, s
      array(i, j) = 2
    enddo
  end subroutine fill
end program routine
```

../exemples/routine.f90

```
$ pgf90 -ta=tesla routine.f90 -Minfo=accel
routine:
  5, Generating copyout(array(:, :))
  Accelerator kernel generated
  Generating Tesla code
    7, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
  5, Generating implicit copy(.50000)
fill:
  13, Generating acc routine seq
  Generating Tesla code
```

Data on device - Opening a data region

There are several ways of making data visible on devices by opening different kinds of data regions.

Computation offloading

Offloading regions are associated with a local data region:

- *serial*
- *parallel*
- *kernels*

Global region

An implicit data region is opened during the lifetime of the program. The management of this region is done with *enter data* and *exit data* directives.

Notes

The actions taken for the data inside these regions depend on the clause in which they appear.

Local regions

To open a data region inside a programming unit (function, subroutine) use *data* directive inside a code block.

Data region associated to programming unit lifetime

A data region is created when a procedure is called (function or subroutine). It is available during the lifetime of the procedure. To make data visible use *declare* directive.

Data on device - Data clauses

H : Host ; *D* : Device ; variable : scalar or array

Data movement

- copyin: The variable is copied H→D, the memory is allocated when entering the region.
- copyout: The variable is copied D→H, the memory is allocated when entering the region.
- copy: copyin + copyout

No data movement

- create: The memory is allocated when entering the region.
- present: The variable is already on the device.
- delete: Frees the memory allocated on the device for this variable.

By default, the clauses check if the variable is already on the device. If so, no action is taken. It is possible to see clauses prefixed with *present_or_* or *p* for OpenACC 2.0 compatibility.

Other clauses

- no_create
- attach
- deviceptr
- detach

Data on device - Shape of arrays

To transfer an array it might be necessary to specify its shape. The syntax differs between Fortran and C/C++:

Fortran

The array shape is to be specified in parentheses. You must specify the first and last index.

```
5!We copy a 2d array on the GPU for matrix a
!In fortran we can omit the shape: copy(a)
!$ACC parallel loop copy(a(1:1000,1:1000))
do i=1, 1000
  do j=1, 1000
10    a(i,j) = 0
  enddo
enddo
!We copyout columns 100 to 199 included to the host
!$ACC parallel loop copy(a(:,100:199))
5do i=1, 1000
  do j=100,199
    a(i,j) = 42
  enddo
enddo
```

../exemples/arrayshape.f90

C/C++

The array shape is to be specified in square brackets. You must specify the first index and the number of elements.

```
5// We copy the array a by giving first element
// and the size of the array
#pragma acc parallel loop copy(a[0:1000][0:1000])
for(int i=0; i< 1000; ++i)
  for (int j=0; j<1000; ++j)
10    a[i][j] = 0;
// Copy copy columns 99 to 198
#pragma acc parallel loop copy(a[:1000][99:100])
for (int i=0; i<1000; ++i)
  for (int j=99;j<199; ++j)
15    a[i][j] = 42;
```

../exemples/arrayshape.c

Data on device - Shape of arrays

Restrictions

- In Fortran, the last index of an assumed-size dummy array must be specified.
- In C/C++, the number of elements of a dynamically allocated array must be specified.

Notes

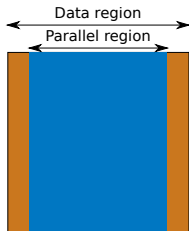
- The shape must be specified when using a slice
- If the first index is omitted, it is considered as the default of the language (C/C++: 0 ; Fortran:1).

Data on device - Parallel regions

Compute constructs *serial*, *parallel*, *kernels* have a data region associated with variables necessary to execution.

```
program para
  integer :: a(1000)
  integer :: i, l
5
  !$ACC parallel copyout(a(:1000))
  !$ACC loop
  do i=1, 1000
    a(i) = i
10
  enddo
  !$ACC end parallel
end program para
```

../exemples/parallel_data.f90



Data on device - Parallel regions

Compute constructs *serial*, *parallel*, *kernels* have a data region associated with variables necessary to execution.

```
program para
  integer :: a(1000)
  integer :: i,l

  !$ACC parallel copyout(a(:1000))
  !$ACC loop
  do i=1, 1000
    a(i) = i
  enddo
  !$ACC end parallel

  do l=1,100
    !$ACC parallel copy(a(:1000))
    !$ACC loop
    do i=1, 1000
      a(i) = a(i) + 1
    enddo
    !$ACC end parallel
  enddo
end program para
```

../exemples/parallel_data_multi.f90

```
Accelerator Kernel Timing data
/gpfs161/pmrwork/idris/sos/ssos013/gpu-pragme/exemples/parallel_data_multi.f90
para NVIDIA devicenum=0
time(us): 2,656
5: compute region reached 1 time
5: kernel launched 1 time
   grid: [8] block: [128]
   device time(us): total=7 max=7 min=7 avg=7
   elapsed time(us): total=146 max=146 min=146 avg=146
5: data region reached 2 times
10: data copyout transfers: 1
   device time(us): total=38 max=38 min=38 avg=38
13: compute region reached 100 times
13: kernel launched 100 times
   grid: [8] block: [128]
   device time(us): total=501 max=6 min=5 avg=5
   elapsed time(us): total=3,662 max=57 min=35 avg=36
13: data region reached 200 times
13: data copyin transfers: 100
   device time(us): total=1,908 max=19 min=9 avg=10
18: data copyout transfers: 100
   device time(us): total=1,102 max=13 min=10 avg=11
```

Notes

- Compute region reached 100 times
- Data region reached 100 times (enter and exit so potentially 200 data transfers)

Not optimal

Data on device - Local data regions

It is possible to open a data region inside a procedure. By doing this you make the variables inside the clauses visible on the device.

You have to use the *data* directive.

```
5  !$ACC parallel copyout(a(:1000))
   !$ACC loop
   do i=1, 1000
     a(i) = i
   enddo
   !$ACC end parallel

10 !$ACC data copy(a(:1000))
   do l=1,100
     !$ACC parallel
     !$ACC loop
     do i=1, 1000
       a(i) = a(i) + 1
     enddo
     !$ACC end parallel
   enddo
   !$ACC end data
```

../exemples/parallel_data_single.f90

```
Accelerator Kernel Timing data
/gpfs161/pwrwork/ldris/sos/ssos013/gpu-pragma/exemples/parallel_data_single.f90
para NVIDIA devicenum=0
time(us): 575
5: compute region reached 1 time
   5: kernel launched 1 time
     grid: [8] block: [128]
     device time(us): total=7 max=7 min=7 avg=7
     elapsed time(us): total=147 max=147 min=147 avg=147
5: data region reached 2 times
   10: data copyout transfers: 1
     device time(us): total=36 max=36 min=36 avg=36
   12: data region reached 2 times
     data copyin transfers: 1
     device time(us): total=19 max=19 min=19 avg=19
   21: data copyout transfers: 1
     device time(us): total=12 max=12 min=12 avg=12
14: compute region reached 100 times
   14: kernel launched 100 times
     grid: [8] block: [128]
     device time(us): total=501 max=6 min=5 avg=5
     elapsed time(us): total=3,488 max=57 min=32 avg=34
```

Notes

- Compute region reached 100 times
- Data region reached 1 time (enter and exit so potentially 2 data transfers)

Optimal?

Data on device - Local data regions

It is possible to open a data region inside a procedure. By doing this you make the variables inside the clauses visible on the device.

You have to use the *data* directive.

```
5  !$ACC parallel copyout(a(:1000))
   !$ACC loop
   do i=1, 1000
     a(i) = i
   enddo
   !$ACC end parallel

10  !$ACC data copy(a(:1000))
   do l=1,100
     !$ACC parallel
     !$ACC loop
     do i=1, 1000
       a(i) = a(i) + 1
     enddo
     !$ACC end parallel
   enddo
   !$ACC end data
```

../exemples/parallel_data_single.f90

```
Accelerator Kernel Timing data
/gpfs161/pwrwork/ldris/sos/ssos013/gpu-pragma/exemples/parallel_data_single.f90
para NVIDIA devicenum=0
time(us): 575
5: compute region reached 1 time
5: kernel launched 1 time
   grid: [8] block: [128]
   device time(us): total=7 max=7 min=7 avg=7
   elapsed time(us): total=147 max=147 min=147 avg=147
5: data region reached 2 times
10: data copyout transfers: 1
   device time(us): total=36 max=36 min=36 avg=36
12: data region reached 2 times
12: data copyin transfers: 1
   device time(us): total=19 max=19 min=19 avg=19
21: data copyout transfers: 1
   device time(us): total=12 max=12 min=12 avg=12
14: compute region reached 100 times
14: kernel launched 100 times
   grid: [8] block: [128]
   device time(us): total=501 max=6 min=5 avg=5
   elapsed time(us): total=3,468 max=57 min=32 avg=34
```

Notes

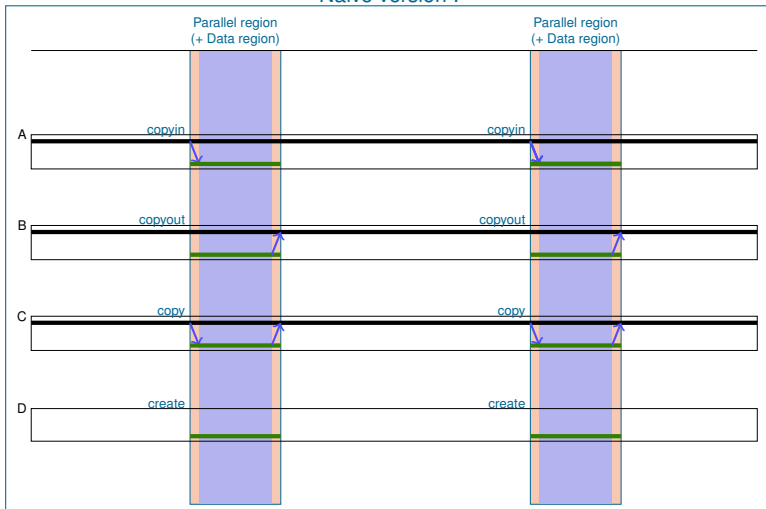
- Compute region reached 100 times
- Data region reached 1 time (enter and exit so potentially 2 data transfers)

Optimal?

No! You have to move the beginning of the data region before the first loop!

Data on device

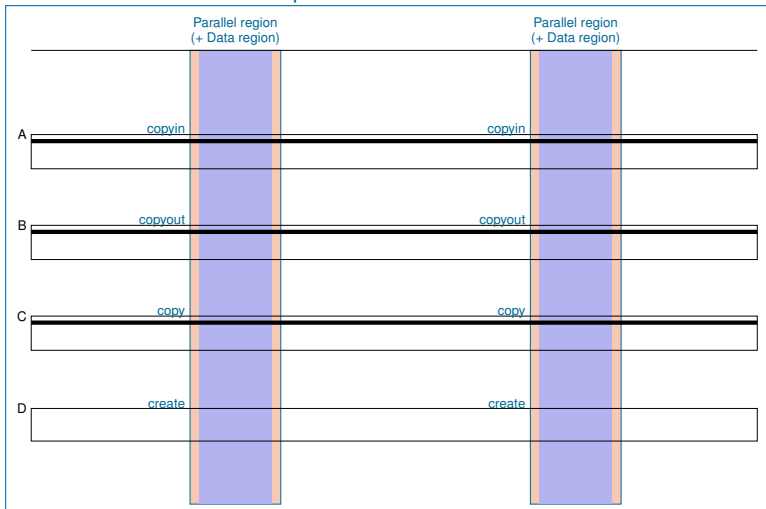
Naive version :



- Transfers : 8
- Allocations : 2

Data on device

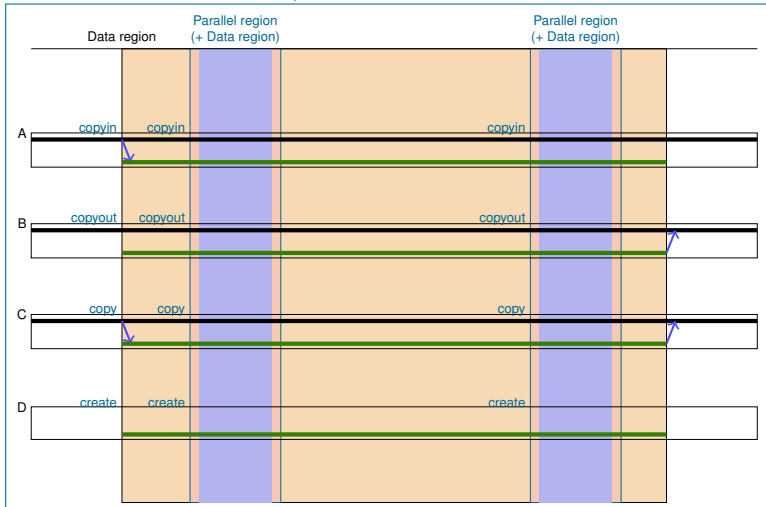
Optimize data transfers:



Lets add a data region

Data on device

Optimize data transfers:

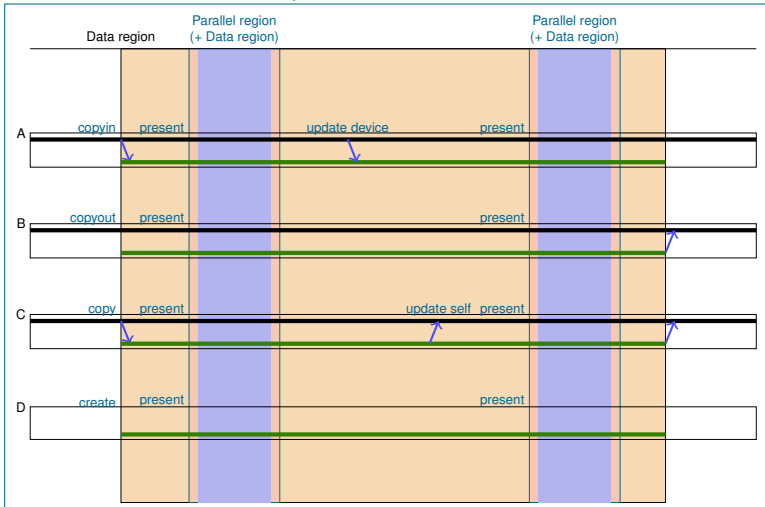


A,B and C are now transferred at entry and exit of the data region.

- Transfers : 4
- Allocation : 1

Data on device

Optimize data transfers:



The clauses check for data presence so to make the code clearer you can use the *present* clause. To make sure the updates are done, use the *update* directive.

- Transfers : 6
- Allocation : 1

Data on device - *update self*

Inside a data region you have to use *update* directive to transfer data. *It is however impossible to use update inside a parallel region.*

self (or *host*)

Variables included within *self* are updated D→H.

```
!$ACC data copyout(a)
!$ACC parallel loop
do i=1,1000
  a(i) = 0
enddo

do j=1, 42
  call random_number(test)
  rng = floor(test*100)
  !$ACC parallel loop copyin(rng) &
  !$ACC& copyout(a)
  do i=1,1000
    a(i) = a(i) + rng
  enddo
enddo
!print *, "before update self", a(42)
!$ACC update self(a(42:42))
!print *, "after update self", a(42)
!$ACC serial
a(42) = 42
!$ACC end serial
print *, "before end data", a(42)
!$ACC end data
print *, "after end data", a(42)
```

../examples/update_dir_comment.f90

Without *update*

```
before end data      0
after end data       42

Accelerator Kernel Timing data
/gpfs16l/purwork/idris/sos/ssos013/gpu-pragma/exemples/./update_dir_comment.f90
update NVIDIA devicenum=0
time(us): 733
6: data region reached 2 times
  28: data copyout transfers: 1
      device time(us): total=25 max=25 min=25 avg=25
7: compute region reached 1 time
  7: kernel launched 1 time
     grid: [8] block: [128]
     device time(us): total=7 max=7 min=7 avg=7
     elapsed time(us): total=2,610 max=2,610 min=2,610 avg=2,610
15: compute region reached 42 times
  15: kernel launched 42 times
     grid: [8] block: [128]
     device time(us): total=274 max=8 min=6 avg=6
     elapsed time(us): total=128,932 max=5,948 min=2,331 avg=3,069
15: data region reached 84 times
  15: data copyin transfers: 42
     device time(us): total=422 max=15 min=5 avg=10
24: compute region reached 1 time
  24: kernel launched 1 time
     grid: [1] block: [1]
     device time(us): total=5 max=5 min=5 avg=5
     elapsed time(us): total=5,259 max=5,259 min=5,259 avg=5,259
```

The *a* array is not initialized on the host before the end of the data region.

Data on device - *update self*

Inside a data region you have to use *update* directive to transfer data. *It is however impossible to use update inside a parallel region.*

self (or *host*)

Variables included within *self* are updated D→H.

```
!$ACC data copyout(a)
!$ACC parallel loop
do i=1,1000
  a(i) = 0
enddo

do j=1, 42
  call random_number(test)
  rng = floor(test*100)
  !$ACC parallel loop copyin(rng) &
  !$ACC& copyout(a)
  do i=1,1000
    a(i) = a(i) + rng
  enddo
enddo
print *, "before update self", a(42)
!$ACC update self(a(42:42))
print *, "after update self", a(42)
!$ACC serial
a(42) = 42
!$ACC end serial
print *, "before end data", a(42)
!$ACC end data
print *, "after end data", a(42)
```

../exemples/update_dir.f90

With *update*

```
before update self      0
after update self      2259
before end data        2259
after end data         42
```

```
Accelerator Kernel Timing data
/gpfs16l/pwrwork/idris/sos/ssos013/gpu-pragma/exemples/./update_dir.f90
update NVIDIA devicenum=0
time(us): 1,946
6: data region reached 2 times
  28: data copyout transfers: 1
     device time(us): total=15 max=15 min=15 avg=15
7: compute region reached 1 time
  7: kernel launched 1 time
     grid: [8] block: [128]
     device time(us): total=7 max=7 min=7 avg=7
     elapsed time(us): total=783 max=783 min=783 avg=783
15: compute region reached 42 times
  15: kernel launched 42 times
     grid: [8] block: [128]
     device time(us): total=253 max=7 min=5 avg=6
     elapsed time(us): total=46,901 max=2,213 min=144 avg=1,116
15: data region reached 84 times
  15: data copyin transfers: 42
     device time(us): total=1,646 max=95 min=6 avg=39
22: update directive reached 1 time
  22: data copyout transfers: 1
     device time(us): total=19 max=19 min=19 avg=19
24: compute region reached 1 time
  24: kernel launched 1 time
     grid: [1] block: [1]
     device time(us): total=6 max=6 min=6 avg=6
     elapsed time(us): total=1,340 max=1,340 min=1,340 avg=1,340
```

The *a* array is initialized on the host after the *update* directive.

Data on device - *update device*

Inside a data region you have to use the *update* directive to transfer data. *It is however impossible to use *update* in a parallel region.*

device

The variables included inside a *device* clause are updated H→D.

Important notes

One benefit of using the *enter data* and *exit data* directives is to manage the data transfer on the device at another place than where they are used. It is especially useful when using C++ constructors and destructors.

```
1 program enterdata
  integer :: s=10000
  real*8, allocatable, dimension(:) :: vec
  allocate (vec(s))
  !$ACC enter data create(vec(1:s))
6  call compute
  !$ACC exit data delete(vec(1:s))
  contains
  subroutine compute
    integer :: i
11    !$ACC parallel loop
    do i=1, s
      vec(i) = 12
    enddo
  end subroutine
16 end program enterdata
```

../exemples/enter_data.f90

Data on device - Global data regions : *declare*

Important notes

In the case of the *declare* directive, the lifetime of the data equals the scope of the code region where it is used.

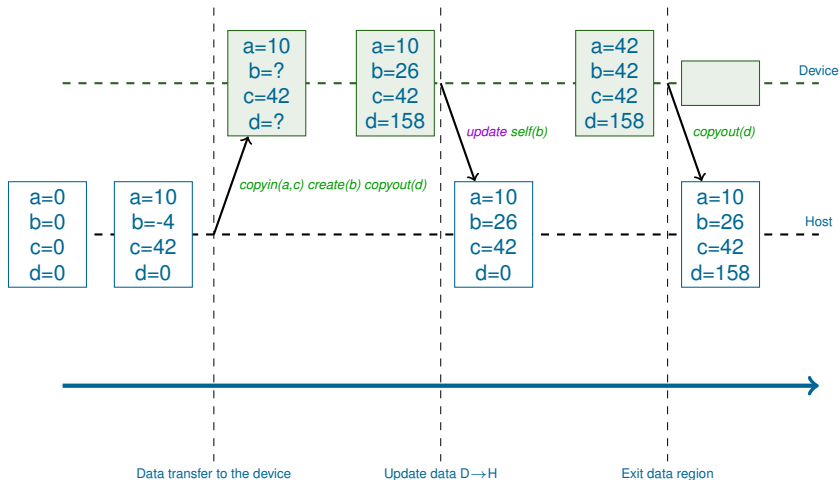
For example :

Zone	Scope
Module	the program
function	the function
subroutine	the subroutine

```
module declare
  integer :: rows = 1000
  integer :: cols = 1000
  integer :: generations = 100
  !$ACC declare copyin(rows, cols, generations)
end module
```

../exemples/declare.f90

Data on device - Time line



Data on device - Important notes

- Data transfers between the host and the device are costly. It is mandatory to minimize these transfers to achieve good performance.
- Since it is possible to use data clauses within *kernels* and *parallel*, if a data region is opened with *data* and it encloses the parallel regions, you should use *update* to avoid unexpected behaviors.
This is equally applicable for a data region opened with *enter data*.

Directive *update*

The *update* directive is used to update data either on the host or on the device.

```
! Update the value of a located on host with the value on device
!$ACC update self(a)
```

4

```
! Update the value of a located on device with the value on host
!$ACC update device(b)
```

../exemples/update.f90

Parallelism - Asynchronism

By default, only one execution thread is created. The kernels are executed **synchronously**, i.e. one after the other. The accelerator is able to manage several execution threads, running concurrently.

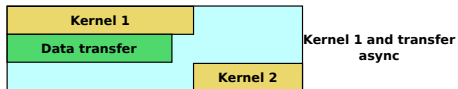
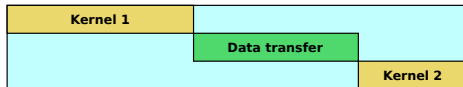
To get better performance it is recommended to maximize the overlaps between:

- Computation and data transfers
- kernel/kernel if they are independant

Asynchronism is activated by adding the *async(execution thread number)* clause to one of these directives: *parallel*, *kernels*, *serial*, *enter data*, *exit data*, *update* and *wait*.

In all cases *async* is optional.

It is possible to specify a number inside the clause to create several execution threads.



```
1 | b is initialized on host
2 | do i=1, s
3 |   b(i) = i
4 | enddo
5 | !$ACC parallel loop async(1)
6 |   do i=1, s
7 |     a(i) = 42
8 |   enddo
9 | !$ACC update device(b) async(2)
10 |
11 | !$ACC parallel loop async(3)
12 |   do i=1, s
13 |     c(i) = 1
14 |   enddo
```

../exemples/async_slides.f90

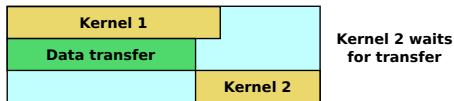
Parallelism - Asynchronism - *wait*

To have correct results, it is likely that some asynchronous kernels need the completion of other. Then you have to add a *wait(execution thread number)* to the directive that needs to wait.

A *wait* is also available.

Important notes

- The argument of *wait* or *wait* is a list of integers representing the execution thread number. For example, *wait(1,2)* says to wait until completion of execution threads 1 and 2.



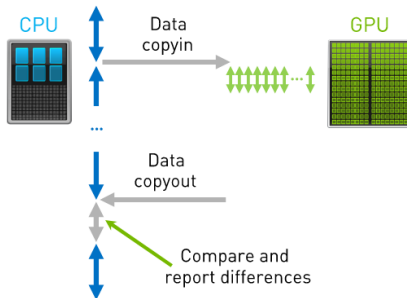
```
! b is initialized on host
do i=1, s
  b(i) = i
4 enddo
!$ACC parallel loop async(1)
do i=1, s
  a(i) = 42
9 enddo
!$ACC update device(b) async(2)
!$ACC parallel loop wait(2)
do i=1, s
  b(i) = b(i) + 1
14 enddo
```

../exemples/async_wait_slides.f90

- 1 Introduction
- 2 Parallelism
- 3 GPU Debugging**
- 4 Annex: Optimization example
- 5 Annex: OpenACC 2.7/OpenMP 5.0 translation table
- 6 Contacts

Debugging - PGI autocompare

- Automatic detection of result differences when comparing GPU and CPU execution.
- How it works: The kernels are run on GPU and CPU; if the results are different, the execution is stopped.



Debugging - PGI autocompare

- To activate this feature, just add autocompare to the compilation (-ta=tesla:cc60,autocompare)
- Example of code that has a *race condition*

```
[ssos89@ouessantm01: Test_OpenACC]$ ./loop_pb_sync.exe
PCAST loop_pb_sync:21 Double
  idx: 0 FAIL ABS act: 5.055585600000000000e+07 exp: 1.000000000000000000e+08 tol: 0.000000000000000000e+00
Somme = 100000000.00000000
compared 2 blocks, 100000001 elements, 800000008 bytes
1 errors found in 1 blocks
absolute tolerance = 0.000000000000000000e+00, abs=0
```

- 1 Introduction
- 2 Parallelism
- 3 GPU Debugging
- 4 Annex: Optimization example**
- 5 Annex: OpenACC 2.7/OpenMP 5.0 translation table
- 6 Contacts

Optimization - Game of Life

From now on, we are going to optimize Conway's Game of Life.
Our starting point is the version using the *parallel* directive.

```
do g=1,generations
  cells = 0
  !$ACC parallel
  do r=1, rows
  do c=1, cols
35     oworld(r,c) = world(r,c)
  enddo
enddo
!$ACC end parallel
40 !$ACC parallel
do r=1, rows
  do c=1, cols
    neigh = oworld(r-1,c-1)+oworld(r,c-1)+oworld(r+1,c-1)+&
            oworld(r-1,c) +oworld(r+1,c)+&
            oworld(r-1,c+1)+oworld(r,c+1)+oworld(r+1,c+1)
45     if (oworld(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
        world(r,c) = 0
    else if (neigh == 3) then
        world(r,c) = 1
50     endif
  enddo
enddo
!$ACC end parallel
!$ACC parallel
55 do r=1, rows
  do c=1, cols
```

../exemples/opti/gol_opti1.f90

```
gol:
33, Accelerator kernel generated
   Generating Tesla code
34, !$acc loop seq
35, !$acc loop seq
33, Generating implicit copyin(world(1:rows,1:cols))
   Generating implicit copyout(oworld(1:rows,1:cols))
34, Loop is parallelizable
35, Loop is parallelizable
40, Accelerator kernel generated
   Generating Tesla code
41, !$acc loop seq
42, !$acc loop seq
40, Generating implicit copy(world(1:rows,1:cols))
   Generating implicit copyin(oworld(0:rows+1,0:cols+1))
41, Loop is parallelizable
42, Loop is parallelizable
54, Accelerator kernel generated
   Generating Tesla code
55, !$acc loop seq
56, !$acc loop seq
57, Generating implicit reduction(+:cells)
54, Generating implicit copyin(world(1:rows,1:cols))
55, Loop is parallelizable
56, Loop is parallelizable
```

Info

- Size : 10000x5000
- Generations : 100
- Serial Time : 26 s
- Time : 34m27 s

The time to solution is increased since the code is executed sequentially and redundantly by *all gangs*.

Optimization - Game of Life

Lets share the work among the active gangs!

```
do g=1,generations
cells = 0
!$ACC parallel loop
do r=1, rows
do c=1, cols
35   oworld(r,c) = world(r,c)
enddo
enddo
!$ACC parallel loop
40do r=1, rows
do c=1, cols
neigh = oworld(r-1,c-1)+oworld(r,c-1)+oworld(r+1,c-1)&
oworld(r-1,c) +oworld(r+1,c)+&
oworld(r-1,c+1)+oworld(r,c+1)+oworld(r+1,c+1)
45 if (oworld(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
world(r,c) = 0
else if (neigh == 3) then
world(r,c) = 1
endif
enddo
50 enddo
enddo
!$ACC parallel loop reduction(+:cells)
do r=1, rows
do c=1, cols
55   cells = cells + world(r,c)
enddo
enddo
!$ACC end parallel
print *, "Cells alive at generation ", g, " : ", cells
60 enddo
```

../exemples/opt/gol_opti2.f90

gol:

```
33, Accelerator kernel generated
Generating Tesla code
34, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
35, !$acc loop seq
33, Generating implicit copyin(world(1:rows,1:cols))
Generating implicit copyout(oworld(1:rows,1:cols))
35, Loop is parallelizable
39, Accelerator kernel generated
Generating Tesla code
40, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
41, !$acc loop seq
39, Generating implicit copy(world(1:rows,1:cols))
Generating implicit copyin(oworld(0:rows+1,0:cols+1))
41, Loop is parallelizable
52, Accelerator kernel generated
Generating Tesla code
52, Generating reduction(+:cells)
53, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
54, !$acc loop seq
52, Generating implicit copyin(world(1:rows,1:cols))
Generating implicit copy(cells)
54, Loop is parallelizable
```

Info

- Size : 10000x5000
- Generations : 100
- Serial Time : 26 s
- Time : 9.5 s

Loops are distributed among gangs.

Optimization - Game of Life

Lets optimize data transfers!

```
cells = 0
!$ACC data copy(oworld,world)
do g=1,generations
cells=0
5!$ACC parallel loop
  do c=1, cols
  do r=1, rows
    oworld(r,c) = world(r,c)
  enddo
10enddo
!$ACC parallel loop
  do c=1, cols
  do r=1, rows
    neigh = oworld(r-1,c-1)+oworld(r,c-1)+oworld(r+1,c-1)+&
15      oworld(r-1,c) +oworld(r+1,c)+&
      oworld(r-1,c+1)+oworld(r,c+1)+oworld(r+1,c+1)
    if (oworld(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
      world(r,c) = 0
    else if (neigh == 3) then
20      world(r,c) = 1
    endif
  enddo
enddo
!$ACC parallel loop reduction(+:cells)
25 do c=1, cols
  do r=1, rows
    cells = cells + world(r,c)
  enddo
enddo
30 print *, "Cells alive at generation ", g, ": ", cells
enddo
!$ACC end data
```

../exemples/opti/gol_opti3.f90

```
gol:
32, Generating copy(world(:,:),oworld(:,:))
35, Accelerator kernel generated
  Generating Tesla code
36, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
37, !$acc loop seq
37, Loop is parallelizable
41, Accelerator kernel generated
  Generating Tesla code
42, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
43, !$acc loop seq
43, Loop is parallelizable
54, Accelerator kernel generated
  Generating Tesla code
54, Generating reduction(+:cells)
55, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
56, !$acc loop seq
54, Generating implicit copy(cells)
56, Loop is parallelizable
```

Info

- Size : 10000x5000
- Generations : 100
- Serial Time : 26 s
- Time : 5 s

Transfers H→D and D→H have been optimized.

- 1 Introduction
- 2 Parallelism
- 3 GPU Debugging
- 4 Annex: Optimization example
- 5 Annex: OpenACC 2.7/OpenMP 5.0 translation table**
- 6 Contacts

Features	OpenACC 2.7		OpenMP 5.0	
	Directives/Clauses	Notes	Directives/Clauses	Notes
GPU offloading	PARALLEL	GRWSVS mode - each team executes instructions redundantly	TEAMS	create teams of threads, execute redundantly
	SERIAL	Only 1 thread executes the code	TARGET	GPU offloading, initial thread only
	KERNELS	Offloading + automatic parallelization	X	Does not exist in OpenMP
Implicit transfer H→D or D→H		scalar variables , non-scalar variables		scalar variables, non-scalar variables
Explicit transfer H→D od D→H inside explicit parallel regions	Clauses PARALLEL/SERIAL/KERNEL copyin(...) copyout(...) copy(...) create(...) present(...) delete(...)	copy H→D when entering construct copy D→H when leaving construct copy H→D when entering and D→H when leaving created on device ; no transfers	Clause TARGET map(to:...) map(from:...) map(tofrom:...) map(alloc:...)	Data movement defined from Host perspective

Features	OpenACC 2.7		OpenMP 5.0	
	Directives/Clauses	Notes	Directives/Clauses	Notes
Data region	declare(...) data/end data enter data/exit data update self (...) update device (...)	Add data to the global data region inside the same programming unit opening and closing anywhere in the code update D→H update H→D	TARGET DATA MAP(to/from)/END TARGET DATA TARGET ENTER DATA/TARGET EXIT DATA UPDATE FROM UPDATE TO	inside the same programming unit opening and closing anywhere in the code Copy D→H inside a data region Copy H→D inside a data region
Loop parallelization	kernel loop gang/work- er/vector	offloading + automatic parallelization of suitable loops, sync at the end of loops Share iterations among gangs, workers and vectors	X distribute	Does not exist in OpenMP Share iterations among TEAMS

Features	OpenACC 2.7		OpenMP 5.0	
	Directives/Clauses	Notes	Directives/Clauses	Notes
routines	routine [seq/vector/worker/gang]	compile a device routine with specified level of parallelism		
asynchronism	async()/wait	execute kernel/transfers asynchronously and explicit sync	nowait/depend	manages asynchronism et task dependencies

Some examples of loop nests parallelized with OpenACC and OpenMP

OpenACC

```
10 !$acc parallel
   !$acc loop gang worker vector
   do i=1,nx
     A(i)=1.14_8*i
   end do
   !$acc end parallel
```

../exemples/loop1D.f90

```
10 !$acc parallel
   !$acc loop gang worker
   do j=1,nx
     !$acc loop vector
     do i=1, nx
       A(i,j)=1.14_8
     end do
   end do
   !$acc end parallel
```

../exemples/loop2D.f90

OpenMP

```
10 !$OMP TARGET
   !$OMP TEAMS DISTRIBUTE PARALLEL FOR SIMD
   do i=1,nx
     A(i)=1.14_8*i
   end do
15 !$OMP END TEAMS DISTRIBUTE PARALLEL FOR
   !$OMP END TARGET
```

../exemples/loop1DOMP.f90

```
10 !$OMP TARGET TEAMS DISTRIBUTE
   do j=1,nx
     !$OMP PARALLEL DO SIMD
     do i=1, nx
       A(i,j)=1.14_8
     end do
15   !$OMP END PARALLEL DO SIMD
   end do
   !$OMP END TARGET TEAMS DISTRIBUTE
```

../exemples/loop2DOMP.f90

OpenACC

```
10 !$acc parallel
11 !$acc loop gang
12 do k=1,nx
13     !$acc loop worker
14     do j=1,nx
15         !$acc loop vector
16         do i=1, nx
17             A(i, j)=1.14_8
18         end do
19     end do
20 end do
21 !$acc end parallel
```

../exemples/loop3D.f90

OpenMP

```
10 !$OMP TARGET TEAMS DISTRIBUTE
11 do k=1,nx
12     !$OMP PARALLEL DO
13     do j=1,nx
14         !$OMP SIMD
15         do i=1, nx
16             A(i, j, k)=1.14_8
17         end do
18     end do
19     !$OMP END SIMD
20 end do
21 !$OMP END PARALLEL DO
22 !$OMP END TARGET TEAMS DISTRIBUTE
```

../exemples/loop3DOMP.f90

- 1 Introduction
- 2 Parallelism
- 3 GPU Debugging
- 4 Annex: Optimization example
- 5 Annex: OpenACC 2.7/OpenMP 5.0 translation table
- 6 Contacts**

Contacts

- Pierre-François Lavallée : pierre-francois.lavallee@idris.fr
- Thibaut Véry : thibaut.very@idris.fr
- Rémi Lacroix : remi.lacroix@idris.fr