

# Introduction to OpenACC directives

Carlos Junqueira-Junior  
Science Computing Research Engineer



# Summary

## 1 Introduction

## 2 Parallelism

- Offloaded Execution
  - Parallel Directives
- Work Sharing
  - Parallel Loop
  - Merging Directives
  - Reductions
  - Memory Access
- Routines
- Data Management
  - Definition
  - Data Clauses
  - Shape of Arrays
  - Data Region
- Asynchronism

## 3 GPU Debugging

- PGI Auto-compare

## 4 Optimization Example

- Game of Life

# Summary

## 1 Introduction

## 2 Parallelism

- Offloaded Execution
  - Parallel Directives
- Work Sharing
  - Parallel Loop
  - Merging Directives
  - Reductions
  - Memory Access
- Routines
- Data Management
  - Definition
  - Data Clauses
  - Shape of Arrays
  - Data Region
- Asynchronism

## 3 GPU Debugging

- PGI Auto-compare

## 4 Optimization Example

- Game of Life

# Summary

## 1 Introduction

## 2 Parallelism

- Offloaded Execution
  - Parallel Directives
- Work Sharing
  - Parallel Loop
  - Merging Directives
  - Reductions
  - Memory Access
- Routines
- Data Management
  - Definition
  - Data Clauses
  - Shape of Arrays
  - Data Region
- Asynchronism

## 3 GPU Debugging

- PGI Auto-compare

## 4 Optimization Example

- Game of Life

# Summary

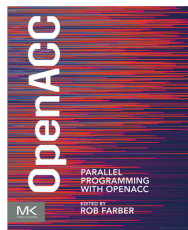
- 1 Introduction
- 2 Parallelism
  - Offloaded Execution
    - Parallel Directives
  - Work Sharing
    - Parallel Loop
    - Merging Directives
    - Reductions
    - Memory Access
  - Routines
  - Data Management
    - Definition
    - Data Clauses
    - Shape of Arrays
    - Data Region
  - Asynchronism
- 3 GPU Debugging
  - PGI Auto-compare
- 4 Optimization Example
  - Game of Life

# References

## IDRIS/CNRS - FRANCE

- Material provided by the Institute for *Development and Resources in Intensive Scientific Computing* (IDRIS/CNRS);
- One can find more information on the [IDRIS website](#).

## Parallel Programming with OpenACC



- Farber, R. (2016). *Parallel programming with OpenACC*. Elsevier.

# Introduction


## Source files

- The source files can be downloaded at the given GitHub repository:
  - ▶ [junqjr's repository](#)

# Introduction

## The Ways to GPU

### Programming Effort and Technical Expertise



#### Libraries

- cuBLAS
- cuSPARSE
- cuRAND
- AmgX
- MAGMA

- Min. change in the code
- Max. performance

#### Directives

- OpenACC
- OpenMP 5.0

- Portable, simple, low intrusiveness
- Efficient

#### Programming Languages

- CUDA
- OpenCL

- Complete rewriting, complex
- Non portable
- Excellent performance



# Introduction

## Brief History - Directives for GPUs

### OpenACC

- <https://www.openacc.org>
- AMD, CRAY, NVIDIA, Oak Ridge
- First standard 1.0 (11/2011)
- Last standard 3.0 (11/2019)
- Main compilers:
  - PGI;
  - CRAY/HPE (only CRAY/HPE hardware);
  - GNU ( $\geq 5.7$ ).

### OpenMP target

- <https://www.openmp.org>
- First standard 4.5 (11/2015)
- Last standard 5.0 (11/2018)
- Main compilers:
  - PGI;
  - CRAY/HPE (only CRAY/HPE hardware);
  - CLANG;
  - IBM XL;
  - GNU ( $\geq 7.0$ ).

# Introduction

## Definitions

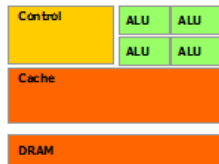
- Thread : Execution entity, serial thread of execution that runs any valid C/C++ or Fortran code;
- SIMT : Single Instruction Multiple Threads;
- SIMD : Single Instruction Multiple Data;
- Device : Accelerator on which execution can be offloaded (ex: GPU);
- Host : Machine hosting 1 or more accelerators 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;
- Vector: Group of threads executing the same instruction (SIMT). Vectors cause worker threads to work in lockstep when running vector or SIMD instruction;
- Worker(OpenACC) : Fine-grain parallelism. Group of threads that can operate together in a SIMD or vector fashion;
- Gang(OpenACC)/Teams(OpenMP) : Coarse-grain parallelism. Groups of workers which operate independently of each other.

# Introduction

## CPU and GPU Architectures

### CPU

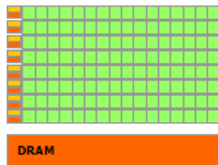
- Several Arithmetic Logic Units (ALU);
- Control unit to control the ALUs;
- Fast cache memory;
- Dynamic Random Access Memory;
- High capacity memory;
- $\approx 1/8$  of the area is dedicated to the computing.



CPU

### GPU

- Hundreds of (ALU), grouped in several multiprocessors;
- Several control units;
- Several cache memories;
- One Dynamic Random Access Memory;
- High bandwidth memory;
- $\approx 3/4$  of the area is dedicated to the computing.



GPU

GPUs are designed such that more transistors are devoted to data processing rather than data caching and flow control as CPUs.

# Introduction

## CPU and GPU Architectures

- OpenACC uses vectorization as an parallelism approach.
- Vector instructions use hardware to effectively (from a software point of view) perform a number of operation at the same time.

### Modern x86 processor

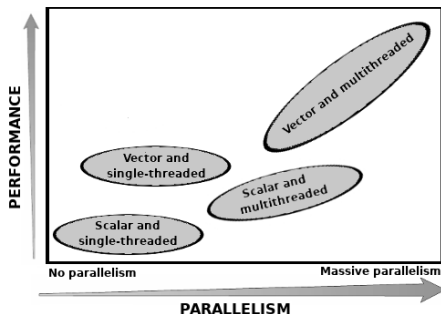
- Can issue vector instructions to hardware vector units (or multiple per-core vector units) to perform multiple data-parallel operations at the same time;
- The AVX-512 vector instruction set is the current longest vector instruction set on x86 architectures that can perform up to 16 concurrent single-precision, 32-bit floating point operations, per instruction call.
- The approach can result in a large overall performance gain when all the vector units on all the cores in a high-end processor are fully utilized.

### GPU architecture

- GPUs utilize SIMD instructions to achieve similar performance multiplier effect, except that vectorization naturally occurs in hardware across a group of threads referred to as a *warp* (CUDA terminology for a set of 32 threads) rather than through the explicit insuance of vector instructions by the programmer or compiler.

# Introduction

## Performance Benefits of Vector and Parallel Programming on Multicore Processors



# Introduction

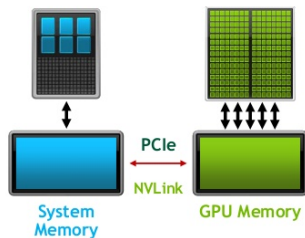
## Execution Model

Kernels, data transfers and memory allocation are managed by the host (CPU).

# Programming GPU-Accelerated Systems

Separate CPU System and GPU Memories

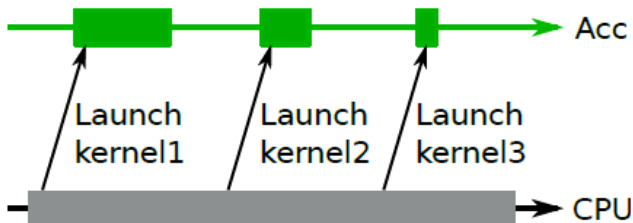
## GPU Developer View



# Introduction

## Execution Model

Kernels, data transfers and memory allocation are managed by the host (CPU).

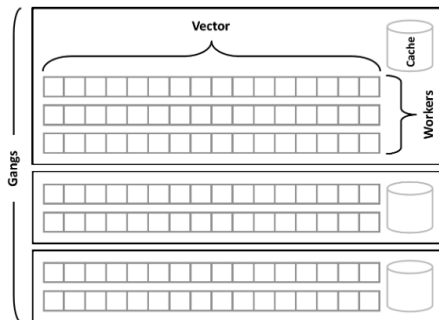


# Introduction

## Execution Model

Three levels of parallelism:

- Coarse grain: gang;
- Fine grain: worker;
- Vectorization: vector;
- Sequential: seq.



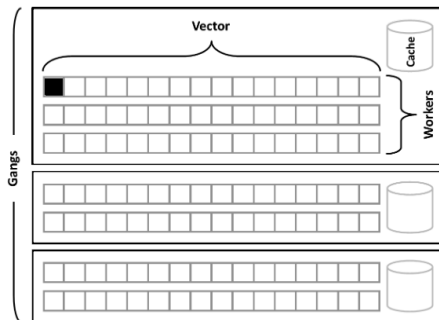


# Introduction

## Execution Model

Three levels of parallelism:

- Coarse grain: gang;
- Fine grain: worker;
- Vectorization: vector;
- **Sequential: seq.**

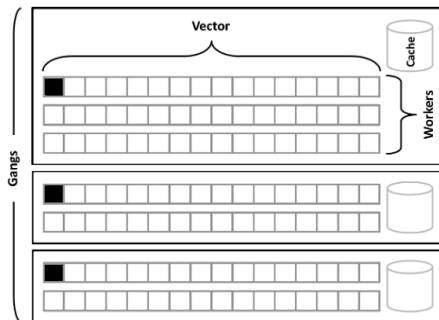


# Introduction

## Execution Model

Three levels of parallelism:

- **Coarse grain: gang;**
- Fine grain: worker;
- Vectorization: vector;
- Sequential: seq.

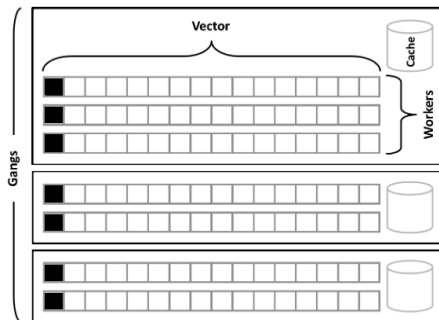


# Introduction

## Execution Model

Three levels of parallelism:

- **Coarse grain: gang;**
- **Fine grain: worker;**
- Vectorization: vector;
- Sequential: seq.

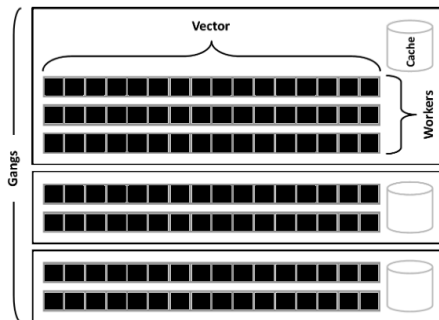


# Introduction

## Execution Model

Three levels of parallelism:

- **Coarse grain: gang;**
- **Fine grain: worker;**
- **Vectorization: vector;**
- **Sequential: seq.**



# Introduction

## Parallelism Models

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

These models must be combined in order to get the best performance from the calculator.

# Introduction

## Parallelism Models

- The execution of a kernel uses a set of threads that are mapped on the hardware resources of the accelerator;
- Threads are grouped within team of the same size, with one master thread per team (gang definition);
- Each team is spread on a 2-D thread-grid (worker-vector);
- One worker is actually a vector of  $vector_{length}$  threads;
- The total number of threads is:

$$nb_{threads} = nb_{gangs} * nb_{workers} * vector_{length} .$$

## Important Notes

- There is no synchronization among gangs;
- The compiler can decide to synchronize the threads of a gang (all or part of them);
- Threads of a worker works in SIMD\* (single instruction multiple data) fashion.

All threads run the same instruction at the same time, for example on NVidia GPUs, groups of 32 threads are formed.

# Introduction

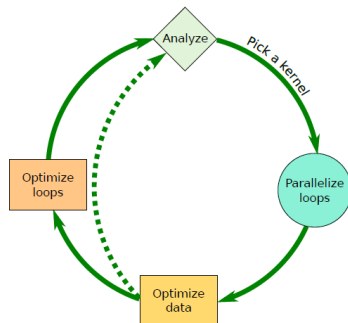
## NVIDIA P100 Restrictions

- Number of gangs is restricted to  $2^{31} - 1$ ;
- The 2-D thread-grid size ( $nb_{workers} * vector_{length}$ ) 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 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

## Porting Strategy

- 1 Identify the compute intensive loops;
- 2 Add OpenACC directives;
- 3 Optimize data transfers and loops;
- 4 Repeats 1 to 3 until everything is on the devices.





# Introduction

## PGI Compiler

### Information

- 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** : Display informations about compilation. The compiler will do implicit operations that are important to the developers.  
**Highly recommended to use!**

### Tools

- nvprof: CPU/GPU profilers;
- nvvp: nvprof GUI.

# Introduction

## "-ta" Options

Each GPU generation presents different capabilities.

For NVIDIA hardware, it is represented by a number:

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

A compilation using V100 features is given by:

- **-ta=tesla:cc70**

## Memory Management

- Pinned: The memory location on the host is pinned. It might improve data transfers;
- Managed: The memory of both, host and device, are united.

Complete documentation:

► [OpenACC User-Guide](#)

# Introduction

## Information available with -Minfo=accel

```

4  program loop
5      integer :: a(10000)
6      integer :: i
7
8      !$ACC parallel loop
9      do i=1,10000
10         a(i) = i
11      enddo
12
13  end program loop

```

**pgfortran -O0 -acc -ta=tesla -Minfo=accel loop.f90**

loop:

```

8, Generating Tesla code
9, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
8, Generating implicit copyout(a(:)) [if not already present]

```

# Introduction

## Information available with -Minfo=accel

```

15  !$acc parallel loop
16  do i=1,10000
17    a(i) = i
18  enddo
19  !$acc parallel loop reduction(+:summ)
20  do i=1,10000
21    summ = summ + a(i)
22  enddo
23  end program reduction

```

**pgfortran -O0 -acc -ta=tesla -Minfo=accel reduction.f90**

reduction:

```

15, Generating Tesla code
16, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
15, Generating implicit copyout(a(:)) [if not already present]
19, Generating Tesla code
19, Generating reduction(+:summ)
20, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
19, Generating implicit copyin(a(:)) [if not already present]
    Generating implicit copy(summ) [if not already present]

```

# Introduction

## Information available with -Minfo=accel

```

15  !$acc parallel loop
16  do i=1,10000
17    a(i) = i
18  enddo
19  !$acc parallel loop reduction(+:summ)
20  do i=1,10000
21    summ = summ + a(i)
22  enddo
23  end program reduction

```

**pgfortran -O0 -acc -ta=tesla -Minfo=accel reduction.f90**

reduction:

```

15, Generating Tesla code
16, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
15, Generating implicit copyout(a(:)) [if not already present]
19, Generating Tesla code
19, Generating reduction(+:summ)
20, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
19, Generating implicit copyin(a(:)) [if not already present]
    Generating implicit copy(summ) [if not already present]

```

# Introduction

## Data region in order to avoid CPU-GPU communication

```

15  !$acc data create (a(1:10000))
16  !$acc parallel loop
17  do i=1,10000
18      a(i) = i
19  enddo
20  !$acc parallel loop reduction(+:summ)
21  do i=1,10000
22      summ = summ + a(i)
23  enddo
24  !$acc end data

```

**pgfortran -acc -ta=tesla -Minfo=accel reduction\_data\_region.f90**

reduction:

```

15, Generating create(a(:)) [if not already present]
16, Generating Tesla code
    17, !$acc loop gang, vector(128) ! blockid%x threadid%x
20, Generating Tesla code
    20, Generating reduction(+:summ)
    21, !$acc loop gang, vector(128) ! blockid%x threadid%x
20, Generating implicit copy(summ) [if not already present]

```

# Introduction

## Code Profiling

### **PGI\_ACC\_TIME**

- Command line tool;
- Environment variable for PGI compilers (PGI\_ACC\_TIME=1);
- Provides basic information such as:
  - Time spent in kernels;
  - Time spent in data transfers;
  - How many times a given kernel is executed;
  - The number of gangs, workers and vector size mapped to hardware.

### **nvprof**

- Command line tools;
- Options that can give you a fine view of the code.

### **nvvp, pgprof and NSight Graphics Profiler**

- Graphical interface for nvprof.

# Introduction

## PGI\_ACC\_TIME=1 — parallel-data-multi.f90

```

29  !$ACC parallel copyout( a(:s) )
30  !$ACC loop
31  do i=1,s
32      a(i) = i
33  enddo
34  !$ACC end parallel

35
36  do j=1,p
37      !$ACC parallel copy( a(:s) )
38      !$ACC loop
39      do i=1,s
40          a(i) = a(i) + 1
41      enddo
42      !$ACC end parallel
43  enddo

```

```

para NVIDIA devicenum=0
time(us): 29,740
29: compute region reached 1 time
    29: kernel launched 1 time
        grid: [79] block: [128]
            device time(us): total=4 max=4 min=4 avg=4
            elapsed time(us): total=368 max=368 min=368 avg=368
29: data region reached 2 times
    34: data copyout transfers: 1
        device time(us): total=22 max=22 min=22 avg=22
37: compute region reached 1000 times
    37: kernel launched 1000 times
        grid: [79] block: [128]
            device time(us): total=2,006 max=4 min=2 avg=2
            elapsed time(us): total=17,550 max=31 min=15 avg=17
37: data region reached 2000 times
    37: data copyin transfers: 1000
        device time(us): total=13,797 max=17 min=5 avg=13
42: data copyout transfers: 1000
    device time(us): total=13,911 max=20 min=5 avg=13

```

The grid is the number of gangs. The block is the size of one gang ( $[vector_{length} \times nb_{workers}]$ ).



# Introduction

## PGI\_ACC\_TIME=1 — parallel-data-single.f90

```

28  !$ACC parallel copyout( a(:s) )
29  !$ACC loop
30  do i=1,s
31      a(i) = i
32  enddo
33  !$ACC end parallel
34
35  !$ACC data copy( a(:s) )
36  do j=1,p
37      !$ACC parallel
38      !$ACC loop
39      do i=1,s
40          a(i) = a(i) + 1
41      enddo
42      !$ACC end parallel
43  enddo
44  !$ACC end data

```

```

para NVIDIA devicenum=0
time(us): 2,069
28: compute region reached 1 time
    28: kernel launched 1 time
        grid: [79] block: [128]
        device time(us): total=4 max=4 min=4 avg=4
        elapsed time(us): total=378 max=378 min=378 avg=378
28: data region reached 2 times
    33: data copyout transfers: 1
        device time(us): total=22 max=22 min=22 avg=22
35: data region reached 2 times
    35: data copyin transfers: 1
        device time(us): total=14 max=14 min=14 avg=14
    44: data copyout transfers: 1
        device time(us): total=9 max=9 min=9 avg=9
37: compute region reached 1000 times
    37: kernel launched 1000 times
        grid: [79] block: [128]
        device time(us): total=2,020 max=5 min=2 avg=2
        elapsed time(us): total=16,670 max=33 min=12 avg=16

```

The grid is the number of gangs. The block is the size of one gang ( $[vector_{length} \times nb_{workers}]$ ).

# Introduction

## nvprof or pgprof

- Nvidia Toolkit provides the command line profiler: nvprof;
- **PGI\_ACC\_TIME=0**, the env. var. is incompatible with nvprof;
- Options:
  - **--cpu-profiling on** : Activates CPU profiling;
  - **--metrics flop\_count\_dp** : Number of operations;
  - **--metrics dram\_read\_throughput** : memory read throughput for each kernel running on the CPU;
  - **--metrics dram\_write\_throughput** : memory write throughput for each kernel running on the CPU;

# Introduction

## nvprof or pgprof - parallel-data-multi.f90

	Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	47.59%	6.0764ms	1000	6.0760us	4.8320us	24.352us	[CUDA memcpy HtoD]	
	40.21%	5.1337ms	1001	5.1280us	3.8080us	19.935us	[CUDA memcpy DtoH]	
	12.18%	1.5552ms	1000	1.5550us	1.4070us	13.503us	para_37_gpu	
API calls:	48.50%	115.20ms	1	115.20ms	115.20ms	115.20ms	cuDevicePrimaryCtxRetain	
	21.14%	50.208ms	1	50.208ms	50.208ms	50.208ms	cuDevicePrimaryCtxRelease	
	13.10%	31.106ms	3002	10.361us	859ns	61.105us	cuStreamSynchronize	
	5.98%	14.198ms	1	14.198ms	14.198ms	14.198ms	cuMemHostAlloc	
	4.48%	10.639ms	1	10.639ms	10.639ms	10.639ms	cuMemFreeHost	
	2.53%	6.0090ms	1001	6.0030us	4.8850us	28.241us	cuLaunchKernel	
	1.56%	3.6943ms	1000	3.6940us	3.0770us	28.117us	cuMemcpyHtoDAsync	
	1.50%	3.5689ms	1001	3.5650us	2.9210us	23.222us	cuMemcpyDtoHAsync	
OpenACC (excl):	27.17%	21.357ms	2000	10.678us	1.4670us	62.106us	acc_wait@parallel-data-multi.f90:37	
	19.31%	15.178ms	1000	15.177us	4.5620us	40.211us	acc_wait@parallel-data-multi.f90:42	
	18.10%	14.231ms	1	14.231ms	14.231ms	14.231ms	acc_exit_data@parallel-data-multi.f90:29	
	9.42%	7.4051ms	1000	7.4050us	5.7670us	390.34us	acc_enqueue_launch@parallel-data-multi.f90:37 (para_37_gpu)	
	7.67%	6.0306ms	1000	6.0300us	4.6600us	350.63us	acc_enqueue_download@parallel-data-multi.f90:42	
	6.70%	5.2692ms	1000	5.2690us	4.0230us	28.372us	acc_enter_data@parallel-data-multi.f90:37	
	5.71%	4.4901ms	1000	4.4900us	3.7610us	28.943us	acc_enqueue_upload@parallel-data-multi.f90:37	
	3.76%	2.9595ms	1000	2.9590us	2.3790us	21.317us	acc_exit_data@parallel-data-multi.f90:37	
	1.80%	1.4138ms	1000	1.4130us	1.1650us	23.892us	acc_compute_construct@parallel-data-multi.f90:37	

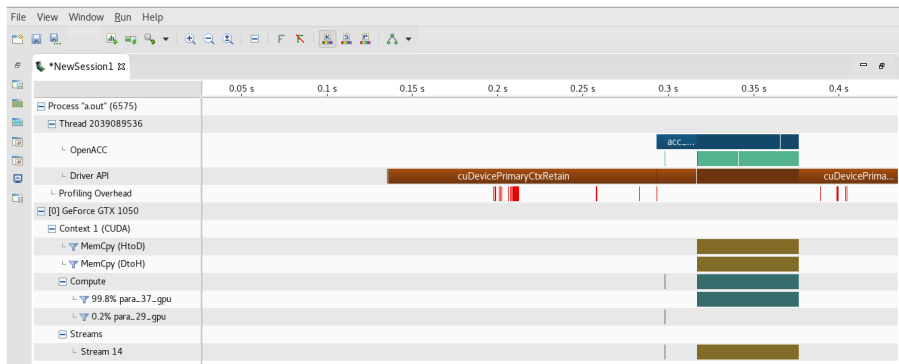
# Introduction

## nvprof or pgprof - parallel-data-single.f90

	Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	98.72%	1.4593ms	1000	1.4590us	1.4070us	12.447us	para_37_gpu	
	0.69%	10.272us	2	5.1360us	4.8960us	5.3760us	[CUDA memcpy DtoH]	
	0.37%	5.5040us	1	5.5040us	5.5040us	5.5040us	[CUDA memcpy HtoD]	
	0.21%	3.0720us	1	3.0720us	3.0720us	3.0720us	para_28_gpu	
API calls:	64.76%	143.60ms	1	143.60ms	143.60ms	143.60ms	cuDevicePrimaryCtxRetain	
	19.97%	44.288ms	1	44.288ms	44.288ms	44.288ms	cuDevicePrimaryCtxRelease	
	6.45%	14.306ms	1	14.306ms	14.306ms	14.306ms	cuMemHostAlloc	
	3.97%	8.8095ms	1	8.8095ms	8.8095ms	8.8095ms	cuMemFreeHost	
	2.94%	6.5292ms	1004	6.5030us	884ns	27.722us	cuStreamSynchronize	
	1.54%	3.4067ms	1001	3.4030us	3.0620us	23.182us	cuLaunchKernel	
	53.03%	14.340ms	1	14.340ms	14.340ms	14.340ms	acc_exit_data@parallel-data-single.f90:28	
OpenACC (excl):	26.16%	7.0745ms	1000	7.0740us	1.4780us	28.318us	acc_wait@parallel-data-single.f90:37	
	15.58%	4.2143ms	1000	4.2140us	3.7990us	21.628us	acc_enqueue_launch@parallel-data-single.f90:37 (para_37_gpu)	
	3.97%	1.0743ms	1000	1.0740us	1.0140us	16.360us	acc_compute_construct@parallel-data-single.f90:37	

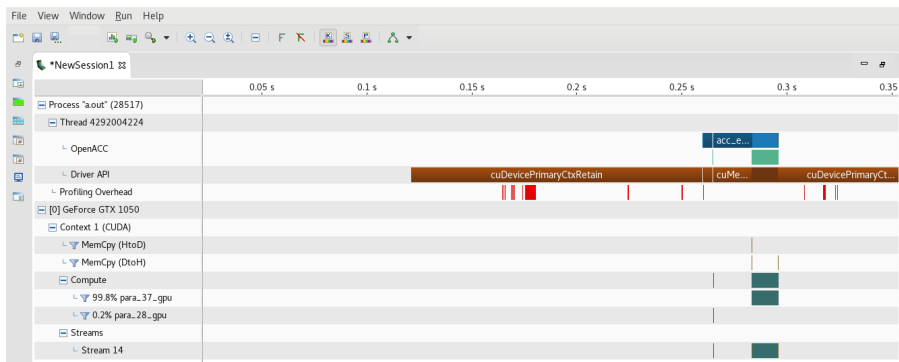
# Introduction

## pgprof GUI - parallel-data-multi.f90



# Introduction

## pgprof GUI - parallel-data-single.f90



# Parallel Directives

- OpenACC features are activated through directives.
- Directives are treated as comments if the right compiler options are not set.
- The syntax is different for a Fortran or a C/C++ code.

## Fortran

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

## C/C++

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

## Directive examples:

kernels, loop, parallel, data, enter data, exit data

# Compute Construct

## Serial

- The code runs on a single thread;
- 1 gang with 1 worker of size 1.

## Kernels

- The compiler analyzes the code and decides the parallelism level of the kernel;
- One kernel is generated for each parallel loop enclosed in the region.

## Parallel

- Parallel region to be run on the device;
- Only one kernel is generated;
- The programmer is responsible for sharing the work manually;
- Execution is redundant by default:
  - Gang-redundant;
  - Worker-single;
  - Vector-single.



# Serial Directive

## Characteristics

- Inform the compiler that the enclosed region has to be offloaded to the GPU;
- Instructions are executed only by one thread, *i.e.* one gang with a worker of size one;
- **Serial** directive is equivalent to the following parameters: **num\_gang(1)**, **num\_worker(1)**, **vector\_length(1)**.

## 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**.

# Serial Directive

## Game of Life - Serial – gol.f90

```

23  !$ACC serial
24  do g=1,generations
25    do r=1, rows
26      do c=1, cols
27        old_world(r,c) = world(r,c)
28      enddo
29    enddo
30    do r=1, rows
31      do c=1, cols
32        neigh = old.world(r-1,c-1)+old.world(r,c-1)+&
33        old.world(r+1,c-1)+old.world(r-1,c)+old.world(r+1,c)+&
34        old.world(r-1,c+1)+old.world(r,c+1)+old.world(r+1,c+1)
35        if (old.world(r,c) == 1 .and. (neigh < 2 or neigh > 3)) then
36          world(r,c) = 0
37        else if (neigh == 3) then
38          world(r,c) = 1
39        endif
40      enddo
41    enddo
42    cells = 0
43    do r=1, rows
44      do c=1, cols
45        cells = cells + world(r,c)
46      enddo
47    enddo
48    print *, "Cells_alive_at_generation-", g, ":", cells
49  enddo
50  !$ACC end serial

```

```

gol:
23: Accelerator serial kernel generated
Generating Tesla code
24: !$acc do seq
25: !$acc do seq
26: !$acc do seq
30: !$acc do seq
31: !$acc do seq
43: !$acc do seq
44: !$acc do seq
23: Generating implicit copy(world(:, :), old_world(:, :)) [if not already present]

```

### Test

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

# Kernels Directive

## Characteristics

- Inform the compiler that the enclosed region contains instructions to be offloaded on the device;
- Each loop nest is treated as an independent kernel with its own parameters *i.e.* number of gangs, workers and vector size.

## Default Behavior

- Data arrays within **kernels** region that are 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 **copy** clause. They are **SHARED**.

## Important Notes

- Parameters of parallel regions, such as gangs, workers, and vector length, are independent.
- Loop nests are executed consecutively.

# Kernels Directive

## Game of Life - Kernels – gol.f90

```

23  !$ACC kernels
24  do g=1,generations
25    do r=1, rows
26      do c=1, cols
27        old_world(r,c) = world(r,c)
28      enddo
29    enddo
30    do r=1, rows
31      do c=1, cols
32        neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
33        old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
34        old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
35        if (old_world(r,c) == 1 .and. (neigh < 2 or neigh > 3)) then
36          world(r,c) = 0
37        else if (neigh == 3) then
38          world(r,c) = 1
39        endif
40      enddo
41    enddo
42    cells = 0
43    do r=1, rows
44      do c=1, cols
45        cells = cells + world(r,c)
46      enddo
47    enddo
48    print *, "Cells_alive_at_generation-", g, ":", cells
49  enddo
50  !$ACC end kernels

```

```

gol:
23: Generating implicit copy(world(:,:),old_world(:,:)) [if not already present]
24: Loop carried dependence due to exposed use of world(:,:) prevents parallelization
    Parallelization would require privatization of array old_world(i2+1,:)
    Generating Tesla code
    24, !$acc loop seq
    25, !$acc loop vector(128) ! threadid%x%
    26, !$acc loop seq
    30, !$acc loop vector(128) ! threadid%x%
    31, !$acc loop seq
    43, !$acc loop vector(128) ! threadid%x%
    44, !$acc loop seq
    45, Generating implicit reduction(+:cells)
25: Loop is parallelizable
26: Loop is parallelizable
30: Loop is parallelizable
31: Loop is parallelizable
43: Loop is parallelizable
44: Loop is parallelizable

```

### Test

- Size : 1000x1000;
- Generations: 100;
- Elapsed time: 1.951s.

# Parallel Directive

## Characteristics

- It creates a parallel region on the device and generates one or more gangs;
- All gangs execute redundantly the instructions within the parallel region, *i.e.* gang-redundant mode;
- Only parallel loop nests with a loop directive are eligible to have their iterations spread among gangs.

## loop.f90

```
8      !$ACC parallel
9      a = 0 !!! Gang redundant
10     !$ACC loop !!! Work sharing
11     do i=1,10000
12         a(i) = i
13     enddo
14     !$ACC end parallel
```

```
loop:
8, Generating Tesla code
9, !$acc loop vector(128) ! threadidx%x
11, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
8, Generating implicit copyout(a(:)) [if not already present]
9, Loop is parallelizable
```

# Parallel Directive

## Default Behavior

- Data arrays within a **parallel** region that are 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**.

## Important Notes

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

# Parallel Directive

## Game of Life - Paralel -acc=noautopar – gol.f90

```

23  !$ACC parallel
24  do g=1,generations
25    do r=1, rows
26      do c=1, cols
27        old_world(r,c) = world(r,c)
28      enddo
29    enddo
30    do r=1, rows
31      do c=1, cols
32        neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
33        old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
34        old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
35        if (old_world(r,c) == 1 .and. (neigh < 2 or neigh > 3)) then
36          world(r,c) = 0
37        else if (neigh == 3) then
38          world(r,c) = 1
39        endif
40      enddo
41    enddo
42    cells = 0
43    do r=1, rows
44      do c=1, cols
45        cells = cells + world(r,c)
46      enddo
47    enddo
48    print *, " Cells_alive_at_generation_", g, ":", cells
49  enddo
50  !$ACC end parallel

```

```

gol:
23: Generating Tesla code
24: !$acc loop seq
25: !$acc loop seq
26: !$acc loop seq
30: !$acc loop seq
31: !$acc loop seq
43: !$acc loop seq
44: !$acc loop seq
23: Generating implicit copy(old_world(:,:),world(:,:)) [if not already present]
24: Loop carried dependence due to exposed use of world(:,:) prevents parallelization
    Parallelization would require privatization of array old_world(i2+1,:).
25: Loop is parallelizable
26: Loop is parallelizable
30: Loop is parallelizable
31: Loop is parallelizable
43: Loop is parallelizable
44: Loop is parallelizable

```

### Test

- Size : 1000x1000;
- Generations: 100;
- Elapsed time: 120.167s

- 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.

# Parallel Directive

## Game of Life - Parallel -acc=autopar - gol.f90

```

23  !$ACC parallel
24  do g=1,generations
25    do r=1, rows
26      do c=1, cols
27        old_world(r,c) = world(r,c)
28      enddo
29    enddo
30    do r=1, rows
31      do c=1, cols
32        neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
33        old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
34        old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
35        if (old_world(r,c) == 1 .and. (neigh < 2 or neigh > 3)) then
36          world(r,c) = 0
37        else if (neigh == 3) then
38          world(r,c) = 1
39        endif
40      enddo
41    enddo
42    cells = 0
43    do r=1, rows
44      do c=1, cols
45        cells = cells + world(r,c)
46      enddo
47    enddo
48    print *, "Cells_alive_at_generation-", g, ":", cells
49  enddo
50  !$ACC end parallel

```

```

gol:
23, Generating Tesla code
24, !$acc loop seq
25, !$acc loop seq
26, !$acc loop vector(128) ! threadid%x%
30, !$acc loop seq
31, !$acc loop vector(128) ! threadid%x%
43, !$acc loop seq
44, !$acc loop vector(128) ! threadid%x%
45, Generating implicit reduction(+:cells)
23, Generating implicit copy(old_world(:,:),world(:,:)) [if not already present]
24, Loop carried dependence due to exposed use of world(:,:) prevents parallelization
Parallelization would require privatization of array old_world(i2+1,:.)
25, Loop is parallelizable
26, Loop is parallelizable
30, Loop is parallelizable
31, Loop is parallelizable
43, Loop is parallelizable
44, Loop is parallelizable

```

### Test

- Size : 1000x1000;
- Generations: 100;
- Elapsed time: 5.067s



# Parallel Directive

## Default Behavior

- It is up to the compiler to 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.

## Control Clauses

The programmer can set control parameters for **kernels** and **parallel** clauses:

- **num\_gangs**: provide number of gangs;
- **num\_workers**: provide number of workers;
- **vector\_length**: provide vector length.

## 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.

# Parallel Directive

## parametres\_parallel.f90

```

14  !$acc parallel num_gangs(10)&
15  !$acc& num_workers(1)      &
16  !$acc& vector_length(128)
17  print *, "Hello_I_am_a_gang"
18  do i=1,1000
19      a(i) = i
20  enddo
21  !$acc end parallel

```

```

1  Hello I am a gang
2  Hello I am a gang
3  Hello I am a gang
4  Hello I am a gang
5  Hello I am a gang
6  Hello I am a gang
7  Hello I am a gang
8  Hello I am a gang
9  Hello I am a gang
10 Hello I am a gang

```

```

param:
14, Generating Tesla code
    18, !$acc loop gang(10), vector(128) ! blockid%x% threadid%x%
14, Generating implicit copyout(a(:)) [if not already present]
18, Loop is parallelizable

```

# Parallel Directive

## parametres\_mod.f90

```

14  !$acc parallel num_gangs(10)&
15  !$acc& num_workers(1)      &
16  !$acc& vector_length(128)
17  print *, "Hello_I_am_a_gang"
18  !$acc loop
19  do i=1,1000
20    a(i) = i
21  enddo
22  !$acc end parallel

```

```

1  Hello I am a gang
2  Hello I am a gang
3  Hello I am a gang
4  Hello I am a gang
5  Hello I am a gang
6  Hello I am a gang
7  Hello I am a gang
8  Hello I am a gang
9  Hello I am a gang
10 Hello I am a gang

```

```

param:
14, Generating Tesla code
    19, !$acc loop gang(10), vector(128) ! blockidx%x threadidx%x
14, Generating implicit copyout(a(:)) [if not already present]

```

# Work Sharing – Loop

## Loop

- 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.
- Automatic nesting is a critical difference between **OpenACC** and **OpenMP-GPU** for which the creation of threads relies on the programmer.

# Work Sharing – Loop

## Clauses

The use of clauses can indicate different parallelism levels:

- **gang**: The iterations of the subsequent loop are distributed block-wise among gangs. It can be gang-redundant or gang-parallel;
- **worker**: Combined with **gangs**, threads of workers are activated, as worker-single or worker-parallel, and the task is shared among those threads.
- **vector**: Workers activate vectors in a single-instruction multiple-thread (SIMT) to share the task in vector single (VS) or vector parallel (VP) modes;
- **seq**: Iterations are sequentially executed on the device;
- **auto**: The compiler analyses the loop region and decides which options are more suitable to respect dependencies;
- **collapse(#loops)**: Merge tightly nested loops;
- **independent**: Tell the compilers that iterations are independent. Useful for the **kernel** directive;
- **private(variable-list)**: Privatize variables;
- **reduction(operation:variable-list)**: Reduction operation.

# Work Sharing – Loop

## Game of Life – acc=noautopar – Parallel loop – gol.f90

```

23  !$ACC parallel
24  do g=1,generations
25    !$acc loop
26    do r=1, rows
27      do c=1, cols
28        old_world(r,c) = world(r,c)
29      enddo
30    enddo
31    do r=1, rows
32      do c=1, cols
33        neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
34        old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
35        old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
36        if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3)) then
37          world(r,c) = 0
38        else if (neigh == 3) then
39          world(r,c) = 1
40        endif
41      enddo
42    enddo
43    cells = 0
44    do r=1, rows
45      do c=1, cols
46        cells = cells + world(r,c)
47      enddo
48    enddo
49    print *, " Cells=alive=at-generation=", g, ":", cells
50  enddo
51  !$ACC end parallel

```

```

gol:
23, Generating Tesla code
24, !$acc loop seq
26, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
27, !$acc loop seq
31, !$acc loop seq
32, !$acc loop seq
44, !$acc loop seq
45, !$acc loop seq
23, Generating implicit copy(old_world(:,:),world(:,:)) [if not already present]
24, Loop carried dependence due to exposed use of world(:,:) prevents parallelization
    Parallelization would require privatization of array old_world(i2+1,:.)
27, Loop is parallelizable
31, Loop is parallelizable
32, Loop is parallelizable
44, Loop is parallelizable
45, Loop is parallelizable

```

# Work Sharing – Loop

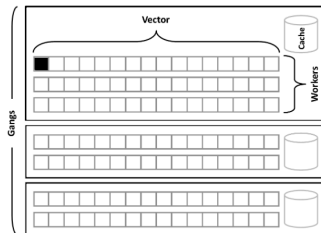
## Sequential – `loop_serial.f90`

```

38  !$ACC serial
39  do i=1,10000
40    a(i) = i
41  enddo
42  !$ACC end serial

```

- Active threads: 1;
- Number of operations:  $nx$ .



```

loop:
38, Accelerator serial kernel generated
   Generating Tesla code
   39, !$acc do seq
38, Generating implicit copyout(a(:)) [if not already present]

```

# Work Sharing – Loop

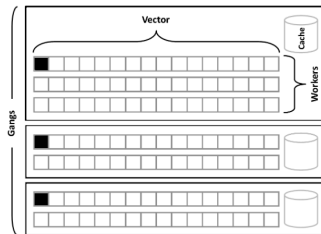
## Gang-Redundant Worker-Serial Vector-Single (GRWSVS) –loop\_grwsvs.f90

```

39  !$ACC parallel num-gangs(10)
40  do i=1,10000
41    a(i) = i
42  enddo
43  !$ACC end parallel

```

- Redundant execution by gang leaders;
- Active threads: 10;
- Number of operations:  $10 \times nx$ .



```

loop:
39, Generating Tesla code
40, !$acc loop gang(10), vector(128) ! blockidx%x threadidx%x
39, Generating implicit copyout(a(:)) [if not already present]
40, Loop is parallelizable

```



# Work Sharing – Loop

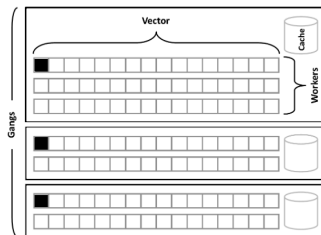
## Gang-Parallel Worker-Single Vector-Single (GPWSVS) – `loop_gpwsvs.f90`

```

40  !$ACC parallel num-gangs(10)
41  !$ACC loop gang
42  do i=1,10000
43    a(i) = i
44  enddo
45  !$ACC end parallel

```

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



```

loop:
40, Generating Tesla code
    42, !$acc loop gang(10), vector(128) ! blockidx%x threadidx%x
40, Generating implicit copyout(a(:)) [if not already present]

```

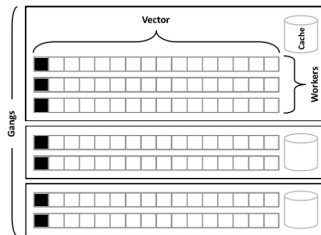
# Work Sharing – Loop

## Gang-Parallel Worker-Parallel Vector-Single (GPWPVS) – `loop_gpwpvs.f90`

```

40  !$ACC parallel num-gangs(10)
41  !$ACC loop gang worker
42  do i=1,10000
43    a(i) = i
44  enddo
45  !$ACC end parallel
  
```

- Iterations are shared among the active workers of each gang;
- Active threads:  $10 \times nb_{workers}$ ;
- Number of operations:  $nx$ .



```

loop:
40, Generating Tesla code
    42, !$acc loop gang(10) ! blockidx%x threadidx%y
40, Generating implicit copyout(a(:)) [if not already present]
  
```

# Work Sharing – Loop

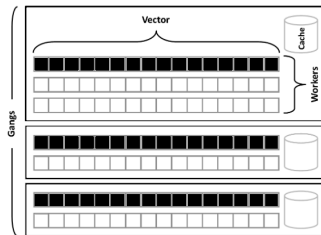
## Gang-Parallel Worker-Single Vector-Parallel (GPWSVP) – `loop_gpwsvp.f90`

```

40  !$ACC parallel num-gangs(10)
41  !$ACC loop gang vector
42  do i=1,10000
43    a(i) = i
44  enddo
45  !$ACC end parallel

```

- Iterations are shared among the threads of the worker of all gangs;
- Active threads:  $10 \times \text{vector\_length}$ ;
- Number of operations:  $nx$ .



```

loop:
40, Generating Tesla code
   42, !$acc loop gang(10), vector(128) ! blockid%x% threadid%x%x
40, Generating implicit copyout(a(:)) [if not already present]

```

# Work Sharing – Loop

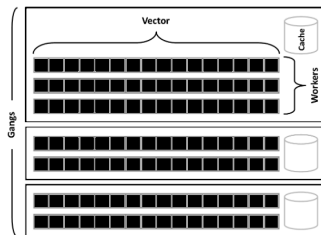
## Gang-Parallel Worker-Parallel Vector-Parallel (GPWPVP) – `loop_gpwpvp.f90`

```

42  !$ACC parallel num-gangs(10)
43  !$ACC loop gang worker vector
44  do i=1,10000
45    a(i) = i
46  enddo
47  !$ACC end parallel

```

- Iterations are shared among the threads of the worker of all gangs;
- Active threads:  $10 \times nb_{workers} \times vector_{length}$ ;
- Number of operations:  $nx$ .



```

loop:
42, Generating Tesla code
   44, !$acc loop gang(10), worker(4), vector(32) ! blockid%x% threadidx%y threadidx%x
42, Generating implicit copyout(a(:)) [if not already present]

```

# Work Sharing – Loop

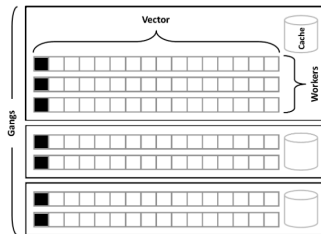
## Gang-Redundant Worker-Parallel Vector-Single (GRWPVS) – `loop_grwpvs.f90`

```

44  !$ACC parallel num-gangs(10)
45  !$ACC loop worker
46  do i=1,10000
47    a(i) = i
48  enddo
49  !$ACC end parallel

```

- All iterations are assigned to each active gang which share the task among workers;
- Active threads:  $10 \times nb_{workers}$ ;
- Number of operations:  $10 \times nx$ .



```

loop:
44, Generating Tesla code
46, !$acc loop gang(10) ! blockidx%x threadidx%y
44, Generating implicit copyout(a(:)) [if not already present]
46, Loop is parallelizable

```

# Work Sharing – Loop

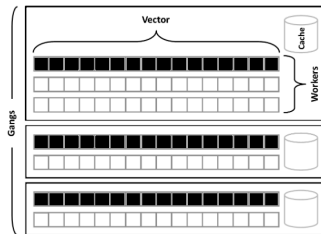
## Gang-Redundant Worker-Single Vector-Parallel (GRWSVP) – `loop_grwsvp.f90`

```

41  !$ACC parallel num-gangs(10)
42  !$ACC loop vector
43  do i=1,10000
44    a(i) = i
45  enddo
46  !$ACC end parallel

```

- All iterations are assigned to each active gang which share the task among vectors of one worker;
- Active threads:  $10 \times \text{vector\_length}$ ;
- Number of operations:  $10 \times nx$ .



```

loop:
41, Generating Tesla code
43, !$acc loop vector(128) ! threadidx%x
41, Generating implicit copyout(a(:)) [if not already present]
43, Loop is parallelizable

```

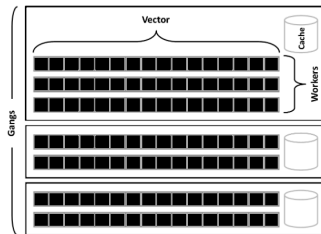
# Work Sharing – Loop

## Gang-Redundant Worker-Parallel Vector-Parallel (GRWPVP) – `loop_grwpvp.f90`

```

49  !$ACC parallel num-gangs(10)
50  !$ACC loop worker vector
51  do i=1,10000
52    a(i) = i
53  enddo
54  !$ACC end parallel
  
```

- All iterations are assigned to each active gang which share the task among the whole thread grid;
- Active threads:  $10 \times nb_{workers} \times vector_{length}$ ;
- Number of operations:  $10 \times nx$ .



```

loop:
49, Generating Tesla code
51, !$acc loop gang(10), worker(4), vector(32) ! blockidx%x threadidx%y threadidx%x
49, Generating implicit copyout(a(:)) [if not already present]
51, Loop is parallelizable
  
```

# Work Sharing – Loop

## Race Conditions

- There is no thread synchronization at gang level (specially at the end of a loop directive). **Risk of race condition**;
- Race condition**: Occurs when multiple threads race to perform some operations on a shared data item. The final result can be undefined;
- OpenACC does not provide any locking mechanism aside from atomic operations to protect against race conditions;
- Synchronization is present when using the *loop* directive 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;
- Gang** parallelism should be avoided inside parallel regions in the presence of dependencies.

pb-sync.f90

```
!$acc parallel
!$acc loop gang
do i=1,nx
  a(i) = 1.0_8
enddo
!$acc loop gang reduction(+:summ)
do i=nx,1,-1
  summ = summ + a(i)
enddo
!$acc end parallel
```

- $Summ = 5.0066080E + 07$  (Race condition);

corr\_sync.f90

```
!$acc parallel
!$acc loop worker vector
do i=1,nx
  a(i) = 1.0_8
enddo
!$acc loop worker vector reduction(+:summ)
do i=nx,1,-1
  summ = summ + a(i)
enddo
!$acc end parallel
```

- $Summ = 1.0000000E + 08$



# Work Sharing – Loop

## Race Conditions - Atomic Operations

- **Atomic Operations:** An operation, such as a write or read-modify-write operation, which starts and runs to completion before another thread can access the shared data item. **(There is no race condition on read-only data items.)**
- **acc atomic** directive example: `accCounter.cpp`.

```

19  high_resolution_clock::time_point t1 = high_resolution_clock::now();
20  // Here is where we define and increment the counter.
21  int counter=0;
22  #pragma acc parallel loop
23  for( int i=0; i < nCount; i++) {
24  #ifdef USE_ATOMIC
25      #pragma acc atomic update
26  #endif
27      counter++;
28  }
29  high_resolution_clock::time_point t2 = high_resolution_clock::now();
30  duration<double> time_span = duration_cast<duration<double>>(t2 - t1);

```

```
pgc++ -std=c++11 -acc -ta=multicore,nvidia:cc35 -Minfo=accel accCounter.cpp -o accCounter.no-atomic
```

```

21, Generating Tesla code
23, #pragma acc loop seq
21, Generating Single core code
27, Accelerator restriction: induction variable live-out from loop: counter

```

# Work Sharing – Loop

## Race Conditions - Atomic Operations

- **Atomic Operations:** An operation, such as a write or read-modify-write operation, which starts and runs to completion before another thread can access the shared data item. **(There is no race condition on read-only data items.)**
- **acc atomic** directive example: `accCounter.cpp`.

```

19  high_resolution_clock::time_point t1 = high_resolution_clock::now();
20  // Here is where we define and increment the counter.
21  int counter=0;
22  #pragma acc parallel loop
23  for( int i=0; i < nCount; i++) {
24  #ifdef USE_ATOMIC
25      #pragma acc atomic update
26  #endif
27      counter++;
28  }
29  high_resolution_clock::time_point t2 = high_resolution_clock::now();
30  duration<double> time_span = duration_cast<duration<double>>(t2 - t1);

```

`pgc++ -DUSE_ATOMIC -std=c++11 -acc -ta=multicore,nvidia:cc35 -Minfo=accel accCounter.cpp -o accCounter.w.atomic`

```

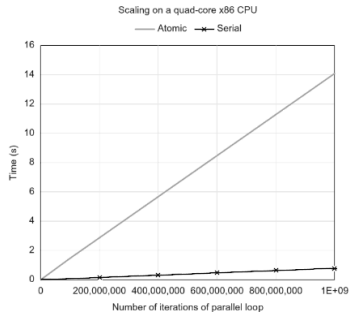
21, Generating Tesla code
23, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
21, Generating Multicore code
23, #pragma acc loop gang
21, Generating implicit copy(counter) [if not already present]

```

# Work Sharing – Loop

## Atomic Operations - x86 Architecture

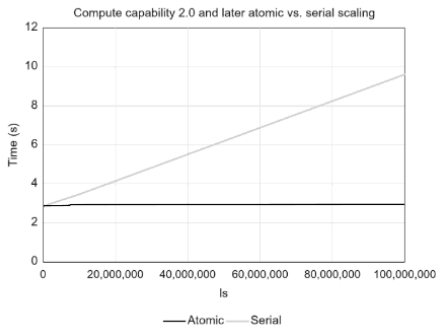
- When using **x86** architecture, the parallel runtime can be significantly worse than the serial code due to the thread and atomic operations overheads.



# Work Sharing – Loop

## Atomic Operations - Nvidia K40 GPU Architecture

- NVIDIA K40 presents Compute Capability 3.5;
- Serial version still scales linearly;
- Parallel version presents constant runtime;
- NVIDIA has made optimizations in hardware that allow some atomic operations to scale in a massively parallel computing environment.



# Work Sharing – Loop

## Lock-Free Programming

- There is no locks in OpenACC;
- *Lock-free* method guarantees progress of at least one thread executing the method. There is no *dead-locks*;
- *Dead-locks* can occur when two threads get stuck waiting for the other thread to release a lock on a shared resource;
- *Lock-based* programs cannot provide any of the performance guarantees of a *lock-free* method. Locks are common culprit that limits or prevents scaling.

### accParaCounter.cpp

```
22  high_resolution_clock::time_point t1 = high_resolution_clock::now();
23  // Here is where we define and increment the counter.
24  int counter=0;
25  #pragma acc parallel loop
26  for( int i=0; i < nCount; i+=nPartial) {
27      int partialSum=0;
28      int n = (i+nPartial < nCount)? i+nPartial:nCount;
29      #pragma acc loop worker reduction(+:partialSum)
30      for( int j=i; j < n; j++){
31          partialSum += 1;
32      }
33      #pragma acc atomic update
34      counter += partialSum;
35  }
36  high_resolution_clock::time_point t2 = high_resolution_clock::now();
37  duration<double> time_span = duration.cast<duration<double>>(t2 - t1);
```

# Work Sharing – Loop

## Lock-Free Programming

- There is no locks in OpenACC;
- Lock-free* method guarantees progress of at least one thread executing the method. There is no *dead-locks*;
- Dead-locks* can occur when two threads get stuck waiting for the other thread to release a lock on a shared resource;
- Lock-based* programs cannot provide any of the performance guarantees of a *lock-free* method. Locks are common culprit that limits or prevents scaling.

**accParaCounter.cpp**

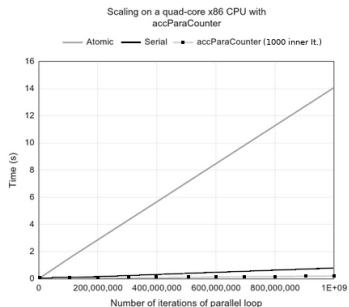
**pgc++ -std=c++11 -acc -ta=multicore,tesla:cc35,cc60 -Minfo=accel accParaCounter.cpp -o accParaCounter**

```
main:
24, Generating Tesla code
    26, #pragma acc loop gang /* blockIdx.x */
    30, #pragma acc loop seq /* threadIdx.y */
        Generating reduction(+:partialSum)
24, Generating implicit copy(counter) [if not already present]
30, Loop is parallelizable
```

# Work Sharing – Loop

## Lock-Free Programming

- There is no locks in OpenACC;
- Lock-free* method guarantees progress of at least one thread executing the method. There is no *dead-locks*;
- Dead-locks* can occur when two threads get stuck waiting for the other thread to release a lock on a shared resource;
- Lock-based* programs cannot provide any of the performance guarantees of a *lock-free* method. Locks are common culprit that limits or prevents scaling.



# Merging Directives

## Kernels, Parallel and Loop

- One can merge read **Kernels/ Parallel** regions with the **loop** directive;
- The clauses available for this construct are those of both constructs.

fused.f90

```

18      !$acc kernels
19      do i=1,10000
20          a1(i) = i
21          a2(i) = i
22          b(i) = i+i*i
23      enddo
24      !$acc end kernels
25      !$acc kernels
26      !$acc loop
27      do i=1,10000
28          a1(i) = b(i)*2
29          summ1 = summ1 + a1(i)
30      enddo
31      !$acc end kernels
32      !$acc kernels loop worker vector
33      do i=1,10000
34          a2(i) = b(i)*2
35          summ2 = summ2 + a2(i)
36      enddo

```

```

fused:
18, Generating implicit copyout(a1(:),a2(:),b(:)) [if not already present]
19, Loop is parallelizable
   Generating Tesla code
19, !$acc loop gang, vector(128) ! blockid%x%w threadid%y%w
25, Generating implicit copyin(b(:)) [if not already present]
   Generating implicit copyout(a1(:)) [if not already present]
27, Loop is parallelizable
   Generating Tesla code
27, !$acc loop gang, vector(128) ! blockid%x%w threadid%y%w
29, Generating implicit reduction(+:summ1)
32, Generating implicit copyin(b(:)) [if not already present]
   Generating implicit copyout(a2(:)) [if not already present]
33, Loop is parallelizable
   Generating Tesla code
33, !$acc loop gang, worker(4), vector(32) ! blockid%x%w threadid%y%w threadid%z%w
35, Generating implicit reduction(+:summ2)

```

- $summ1 = 1146749120$
- $summ2 = 1146749120$



# Reductions

## Reduction directive

- A variable assigned to a **reduction** clause is privatised for each element of the parallelism level of the loop;
- At the end of the region, an operation is executed using options provided below:

### Reduction operations in Fortran.

Operation	Effect
+	Summ
*	Product
max	Maximum
min	Minimum
land	Bitwise and
lor	Bitwise or
lexor	Bitwise xor
.and.	Logical and
.or.	Logical or

### Reduction operations in C/C++.

Operation	Effect
+	Summ
*	Product
max	Maximum
min	Minimum
&	Bitwise and
	Bitwise or
&&	Logical and
	Logical or

## Restrictions

The variable have to be a scalar with a numerical value:

- Fortran: **integer,real,double precision,complex;**
- C: **char,int,float,double, \_Complex;**
- C++: **char,wchar\_t,int,float,double;**

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

# Reductions

## Example - corr\_sync.f90

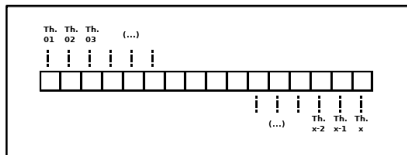
```
18  program reduction
19      real    :: a(100000000)
20      integer :: i
21      integer :: nx=100000000
22      real    :: summ=0
23      !$acc parallel
24      !$acc loop worker vector
25      do i=1,nx
26          a(i) = 1.0_8
27      enddo
28      !$acc loop worker vector reduction(+:summ)
29      do i=nx,1,-1
30          summ = summ + a(i)
31      enddo
32      !$acc end parallel
33      write (*,*) summ
34  end program reduction
```

# Memory Access and Coalescing

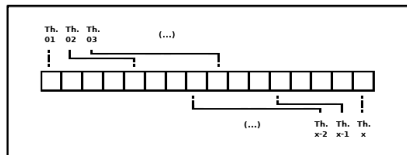
## Principles

- 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.

Contiguous data access.



Non-contiguous data access.



# Memory Access and Coalescing

## Sources

### loop\_nocoalescing.f90

```

22  !$ACC parallel
23  !$ACC loop gang
24  do i=1,nx
25    !$ACC loop vector
26    do j=1,nx
27      a(i,j) = 1.14d-8
28    enddo
29  enddo
30  !$ACC end parallel

```

---


$$nx = 10000$$


---

### loop\_coalescing-seq.f90

```

22  !$ACC parallel
23  !$ACC loop gang vector
24  do i=1,nx
25    !$ACC loop seq
26    do j=1,nx
27      a(i,j) = 1.14d-8
28    enddo
29  enddo
30  !$ACC end parallel

```

### loop\_coalescing-vec.f90

```

22  !$ACC parallel
23  !$ACC loop gang
24  do j=1,nx
25    !$ACC loop vector
26    do i=1,nx
27      a(i,j) = 1.14d-8
28    enddo
29  enddo
30  !$ACC end parallel

```

# Memory Access and Coalescing

## Compiling

### loop\_nocoalescing.f90

```
loop:
22, Generating Tesla code
24, !$acc loop gang ! blockidx%x
26, !$acc loop vector(128) ! threadidx%x
22, Generating implicit copyout(a(1:nx,1:nx)) [If not already present]
26, Loop is parallelizable
```

### loop\_coalescing-seq.f90

```
loop:
22, Generating Tesla code
24, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
26, !$acc loop seq
22, Generating implicit copyout(a(1:nx,1:nx)) [If not already present]
```

### loop\_coalescing-vec.f90

```
loop:
22, Generating Tesla code
24, !$acc loop gang ! blockidx%x
26, !$acc loop vector(128) ! threadidx%x
22, Generating implicit copyout(a(1:nx,1:nx)) [If not already present]
26, Loop is parallelizable
```

# Memory Access and Coalescing

## Computing

### loop\_nocoalescing.f90

```

loop NVIDIA devicenum=0
time(us): 126,451
22: compute region reached 1 time
22: kernel launched 1 time
   grid: [10000] block: [128]
       device time(us): total=94,509 max=94,509 min=94,509 avg=94,509
       elapsed time(us): total=94,553 max=94,553 min=94,553 avg=94,553
22: data region reached 2 times
30: data copyout transfers: 24
   device time(us): total=31,942 max=1,394 min=1,211 avg=1,330

```

### loop\_coalescing-seq.f90

```

loop NVIDIA devicenum=0
time(us): 37,550
22: compute region reached 1 time
22: kernel launched 1 time
   grid: [79] block: [128]
       device time(us): total=5,634 max=5,634 min=5,634 avg=5,634
       elapsed time(us): total=5,677 max=5,677 min=5,677 avg=5,677
22: data region reached 2 times
30: data copyout transfers: 24
   device time(us): total=31,916 max=1,505 min=1,174 avg=1,329

```

### loop\_coalescing-vec.f90

```

loop NVIDIA devicenum=0
time(us): 35,804
22: compute region reached 1 time
22: kernel launched 1 time
   grid: [10000] block: [128]
       device time(us): total=4,145 max=4,145 min=4,145 avg=4,145
       elapsed time(us): total=4,188 max=4,188 min=4,188 avg=4,188
22: data region reached 2 times
30: data copyout transfers: 24
   device time(us): total=31,659 max=1,337 min=1,126 avg=1,319

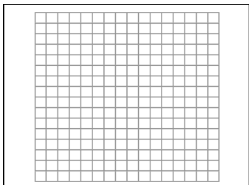
```

	W/O. Coal.	With Coal.
Time ( $\mu$ s)	$\approx 126$	$\approx 36$

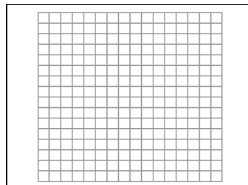
# Memory Access and Coalescing

## Computing

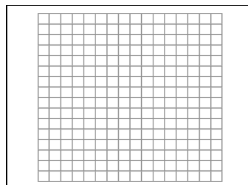
`loop_nocoalescing.f90` – `gang(i)/vector(j)`



`loop_coalescing-seq.f90` – `gang-vector(i)/seq(j)`



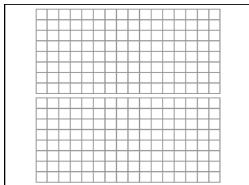
`loop_coalescing-vec.f90` – `gang(j)/vector(i)`



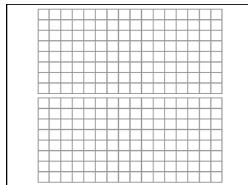
# Memory Access and Coalescing

## Computing

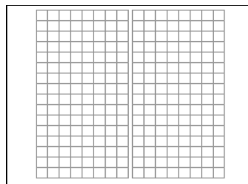
`loop_nocoalescing.f90` – `gang(i)/vector(j)`



`loop_coalescing-seq.f90` – `gang-vector(i)/seq(j)`



`loop_coalescing-vec.f90` – `gang(j)/vector(i)`

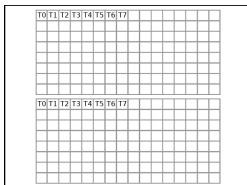




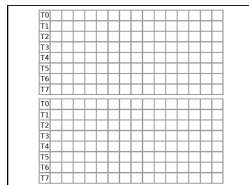
# Memory Access and Coalescing

## Computing

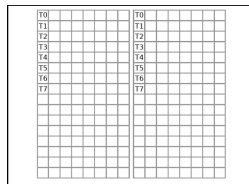
`loop_nocoalescing.f90` – `gang(i)/vector(j)`



`loop_coalescing-seq.f90` – `gang-vector(i)/seq(j)`



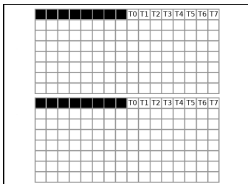
`loop_coalescing-vec.f90` – `gang(j)/vector(i)`



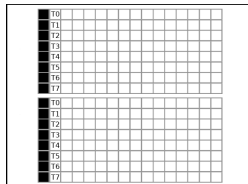
# Memory Access and Coalescing

## Computing

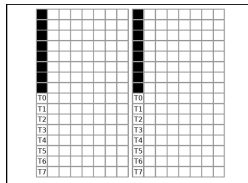
`loop_nocoalescing.f90` – `gang(i)/vector(j)`



`loop_coalescing-seq.f90` – `gang-vector(i)/seq(j)`



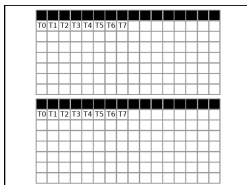
`loop_coalescing-vec.f90` – `gang(j)/vector(i)`



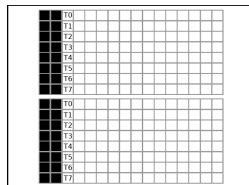
# Memory Access and Coalescing

## Computing

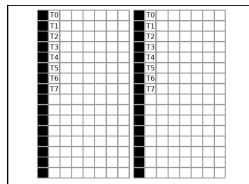
`loop_nocoalescing.f90` – `gang(i)/vector(j)`



`loop_coalescing-seq.f90` – `gang-vector(i)/seq(j)`



`loop_coalescing-vec.f90` – `gang(j)/vector(i)`



# Routines

- Routines and functions need to be declared using the **routine** directive;
- The **routine** directive provides 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**).

## **routine-wrong.f90**

```

22  !$ACC parallel
23  !$ACC loop
24  do i=1,s
25      call fill( a(:,i), s, i )
26  enddo
27  !$ACC end parallel
28  write(*,*) a(1,10)
29  contains
30  subroutine fill( arr, j, k )
31      integer, intent(out) :: arr(:,i)
32      integer, intent(in)  :: j, k
33      integer :: l
34      do l=1,j
35          arr(k,l) = 2
36      enddo
37  end subroutine

```

```

PGF90-S-005-Procedures called in a compute region must have &
          acc routine information: fill (routine-wrong.f90: 25)
PGF90-S-005-Accelerator region ignored;                               &
          see-Minfo messages (routine-wrong.f90: 22)
routine:
22, Accelerator region ignored
25, Accelerator restriction: call to 'fill' with no acc routine information
0 inform, 0 warnings, 2 severs, 0 fatal for routine

```

# Routines

- Routines and functions need to be declared using the **routine** directive;
- The **routine** directive provides 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**).

## `routine-corr.f90`

```

22  !$ACC parallel copyout(a)
23  !$ACC loop
24  do i=1,s
25      call fill( a(:,i), s, i )
26  enddo
27  !$ACC end parallel
28  write(*,*) a(1,10)
29  contains
30  subroutine fill( arr, j, k )
31      !$ACC routine seq
32      integer, intent(out) :: arr(:,i)
33      integer, intent(in)  :: j, k
34      integer :: l
35      do l=1,j
36          arr(k,l) = 2
37      enddo
38  end subroutine

```

```

routine:
22. Generating copyout(a(:,i)) [if not already present]
    Generating implicit copy(.50000) [if not already present]
    Generating Tesla code
24. !$acc loop gang, vector(128) ! blockidx%x threadidx%x
fill:
30. 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

Routines and functions need to be declared using the

- `serial`;
- `parallel`;
- `kernel`.

### Global region

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

### Local regions

To open a data region inside a programming unit (function, subroutine) it is necessary to use the `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 the `declare` directive.

## Notes

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

# Data on Device

## Data Clauses

### Abbreviations and definitions:

- $H$  : Host;
- $D$  : Device;

### Data Movement

- **copyin** : The variable is copied  $H \rightarrow D$ . Memory is allocated when entering the the region;
- **copyout** : The variable is copied  $D \rightarrow H$ . Memory is allocated when entering the the region;
- **copy** : **copyin** + **copyout**.

### Note

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 from the OpenACC 2.7 Standard.

- **deviceptr** (Sec. 2.7.3, pg. 41, OpenACC-2.7 Std.);
- **no\_create** (Sec. 2.7.9, pg. 44, OpenACC-2.7 Std.);
- **attach** (Sec. 2.7.11, pg. 45, OpenACC-2.7 Std.);
- **dettach** (Sec. 2.7.12, pg. 45, OpenACC-2.7 Std.).

### No Data Movement

- **Variable**: The variable can be a scalar or an array as well.
- **create** : The memory is allocated when entering the region;
- **present** : The variable is already in the device;
- **delete** : It dellocates memory on the device used by the variable.

# Data on Device

## Shape of Arrays

- It is necessary to specify the shape of an array when transferring data;
- Fortran and C++ do not use the same syntax when transferring arrays.

### Fortran — array-shape.f90

```
22      !Copy a 2-D array on the GPU — matrix "a"  
23      !$ACC parallel loop gang copy(a(1:s,1:s))  
24      do j=1,s  
25          !$ACC loop worker vector  
26          do i=1,s  
27              a(i,j) = 0  
28          enddo  
29      enddo  
30      !Copyout columns 100 to 199 included  
31      !to the host  
32      !$ACC parallel loop gang copy(a(1:s,100:199))  
33      do j=100,199  
34          !$ACC loop worker vector  
35          do i=1,s  
36              a(i,j) = 42  
37          enddo  
38      enddo
```

- The array shape have to be provided in parentheses;
- It is necessary to provide the first and last indices.



# Data on Device

## Shape of Arrays

- It is necessary to specify the shape of an array when transferring data;
- Fortran and C++ do not use the same syntax when transferring arrays.

### Fortran — array-shape.f90

```
array_shape :
23, Generating copy(a(1:s,1:s)) [if not already present]
   Generating Tesla code
   24, !$acc loop gang ! blockidx%x
   26, !$acc loop worker(4), vector(32) ! threadidx%y threadidx%x
26, Loop is parallelizable
32, Generating copy(a(1:s,100:199)) [if not already present]
   Generating Tesla code
   33, !$acc loop gang ! blockidx%x
   35, !$acc loop worker(4), vector(32) ! threadidx%y threadidx%x
35, Loop is parallelizable
```

- The array shape have to be provided in parentheses;
- It is necessary to provide the first and last indices.

# Data on Device

## Shape of Arrays

- It is necessary to specify the shape of an array when transferring data;
- Fortran and C++ do not use the same syntax when transferring arrays.

### C/C++ – array-shape.cpp

```
28 // Copy the array "a" by giving first element
29 // and the size of the array
30 #pragma acc parallel loop gang copy (a[0:s][0:s])
31 for (int i=0; i<s; ++i )
32     #pragma acc loop worker vector
33     for ( int j=0; j<s; ++j )
34         a[i][j]=0;
35 // Copy copy columns 99 to 198
36 #pragma acc parallel loop gang copy (a[0:s][99:100])
37 for (int i=0; i<s; ++i)
38     #pragma acc loop worker vector
39     for (int j=99; j<199; ++j)
40         a[i][j]=42;
```

- The array shape have to be provided in square brackets;
- It is necessary to provide the first index and the number of elements.

# Data on Device

## Shape of Arrays

- It is necessary to specify the shape of an array when transferring data;
- Fortran and C++ do not use the same syntax when transferring arrays.

### C/C++ – array-shape.cpp

```
main:
26, Generating copy(a[:s][:s]) [if not already present]
   Generating Tesla code
   31, #pragma acc loop gang /* blockIdx.x */
   33, #pragma acc loop worker(4), vector(32) /* threadIdx.y threadIdx.
33, Loop is parallelizable
34, Generating copy(a[:s][99:100]) [if not already present]
   Generating Tesla code
   37, #pragma acc loop gang /* blockIdx.x */
   39, #pragma acc loop worker(4), vector(32) /* threadIdx.y threadIdx.
39, Loop is parallelizable
```

- The array shape have to be provided in square brackets;
- It is necessary to provide the first index and the number of elements.

# Data on Device

## Restrictions

- In Fortran, the last index of an assumed-size dummy array must be specified;  
The dummy argument is a deferred-shape array with (:) bounds.
- 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;  
A slice is subset of elements from an array which is rearranged into another array.
- If the first index is omitted, it is considered as the default of the language
  - **Fortran:** 1;
  - **C/C++:** 0.

# Data Region

## Parallel Regions

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

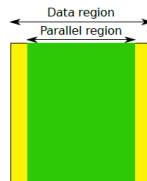
**parallel-data.f90**

20  
21  
22  
23  
24  
25

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

```
parallel.data:
20: Generating copyout(a(:)) [if not already present]
Generating Tesla code
22: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

```
parallel.data NVIDIA devicenum=0
time(us): 30
20: compute region reached 1 time
20: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=4 max=4 min=4 avg=4
elapsed time(us): total=373 max=373 min=373 avg=373
20: data region reached 2 times
25: data copyout transfers: 1
device time(us): total=26 max=26 min=26 avg=26
```



Type	Time(%)	Name
GPU activities:	67.24%	[CUDA memcpy DtoH]
	32.76%	parallel.data.20.gpu
API calls:	65.14%	cuDevicePrimaryCtxRetain
	21.76%	cuDevicePrimaryCtxRelease
	7.61%	cuMemHostAlloc
	5.08%	cuMemFreeHost

# Data Region

## Amdahl's Law and Scaling

- Parallel programming can ideally deliver a factor of  $N$  speedup when running on a parallel computer that has  $N$  processing elements;
- Amdahl's law is an approximation that models the ideal speedup,  $S$ , considering an strong scalability, than can happen when serial programs are modified to run in parallel.
- Strong scalability considers a constant problem size when the problem is parallelized.

# Data Region

## Amdahl's Law and Scaling

- The expected speedup,  $S(N)$  of a parallel code over the serial code when using  $N$  processors is dictated by the proportion of a program that can be made parallel,  $P$ , and the portion that cannot be parallelized,  $(1 - P)$ :

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

## OpenACC developers have two goals:

- 1 Express the parallel sections of code so they run as fast as possible. Ideally they should run  $N$  times faster when using  $N$  processors.
- 2 Utilize whatever techniques or inventiveness they have to minimize the  $(1 - P)$  serial time.

Data transfers are included in the  $(1 - P)$  portion of the code!

# Data Region

## parallel-data-multi.f90 – p=1000 and s=10000

```

28  !$ACC parallel copyout( a(:s) )
29  !$ACC loop
30  do i=1,s
31      a(i) = i
32  enddo
33  !$ACC end parallel
34  do j=1,p
35      !$ACC parallel copy( a(:s) )
36      !$ACC loop
37      do i=1,s
38          a(i) = a(i) + 1
39      enddo
40      !$ACC end parallel
41  enddo

```

```

parallel_data:
28, Generating copyout(a(:s)) [if not already present]
   Generating Tesla code
   30, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
35, Generating copy(a(:s)) [if not already present]
   Generating Tesla code
   37, !$acc loop gang, vector(128) ! blockidx%x threadidx%x

```

```

time(us): 26,006
28: compute region reached 1 time
   28: kernel launched 1 time
       grid: [79] block: [128]
       device time(us): total=3 max=3 min=3 avg=3
       elapsed time(us): total=372 max=372 min=372 avg=372
28: data region reached 2 times
   33: data copyout transfers: 1
       device time(us): total=21 max=21 min=21 avg=21
35: compute region reached 1000 times
   35: kernel launched 1000 times
       grid: [79] block: [128]
       device time(us): total=2,032 max=12 min=2 avg=2
       elapsed time(us): total=19,385 max=56 min=15 avg=19
35: data region reached 2000 times
   35: data copyin transfers: 1000
       device time(us): total=11,589 max=36 min=7 avg=11
   40: data copyout transfers: 1000
       device time(us): total=12,361 max=40 min=7 avg=12

```

### Notes

- Compute region reached 1000 times;
- Data region reached 2000 times (copyin+copyout).



# Data Region

## parallel-data-single.f90 – p=1000 and s=10000

```

27  !$ACC parallel copyout( a(:s) )
28  !$ACC loop
29  do i=1,s
30    a(i) = i
31  enddo
32  !$ACC end parallel
33  !$ACC data copy( a(:s) )
34  do j=1,p
35    !$ACC parallel
36    !$ACC loop
37    do i=1,s
38      a(i) = a(i) + 1
39    enddo
40    !$ACC end parallel
41  enddo
42  !$ACC end data

```

```

parallel_data:
27, Generating copyout(a(:s)) [if not already present]
   Generating Tesla code
   29, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
33, Generating copy(a(:s)) [if not already present]
35, Generating Tesla code
   37, !$acc loop gang, vector(128) ! blockidx%x threadidx%x

```

```

time(us): 2,076
27: compute region reached 1 time
27: kernel launched 1 time
   grid: [79] block: [128]
       device time(us): total=4 max=4 min=4 avg=4
elapsed time(us): total=357 max=357 min=357 avg=357
27: data region reached 2 times
32: data copyout transfers: 1
   device time(us): total=20 max=20 min=20 avg=20
33: data region reached 2 times
33: data copyin transfers: 1
   device time(us): total=15 max=15 min=15 avg=15
42: data copyout transfers: 1
   device time(us): total=9 max=9 min=9 avg=9
35: compute region reached 1000 times
35: kernel launched 1000 times
   grid: [79] block: [128]
       device time(us): total=2,028 max=12 min=2 avg=2
elapsed time(us): total=17,143 max=47 min=13 avg=17

```

### Notes

- Compute region reached 1000 times;
- Data region reached 3 times.

# Data Region

**optm.f90 – p=1000 and s=10000**

```

27  !$ACC data copyout( a(:s) )
28  !$ACC parallel loop
29  do i=1,s
30      a(i) = i
31  enddo
32  do j=1,p
33      !$ACC parallel loop
34      do i=1,s
35          a(i) = a(i) + 1
36      enddo
37  enddo
38  !$ACC end data
  
```

```

parallel.data:
28, Generating copyout(a(:s)) [if not already present]
29, Generating Tesla code
30, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
34, Generating Tesla code
35, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
  
```

```

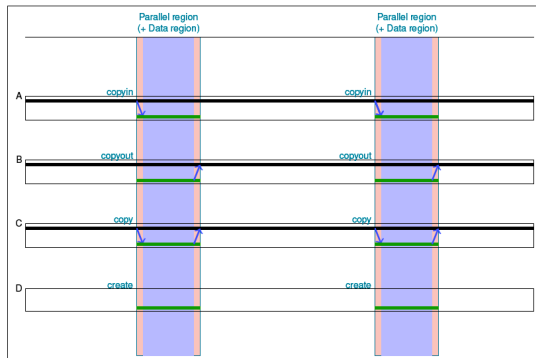
time(us): 2,040
28: data region reached 2 times
39: data copyout transfers: 1
   device time(us): total=23 max=23 min=23 avg=23
29: compute region reached 1 time
29: kernel launched 1 time
   grid: [79] block: [128]
   device time(us): total=4 max=4 min=4 avg=4
elapsed time(us): total=379 max=379 min=379 avg=379
34: compute region reached 1000 times
34: kernel launched 1000 times
   grid: [79] block: [128]
   device time(us): total=2,013 max=5 min=2 avg=2
elapsed time(us): total=16,992 max=29 min=14 avg=16
  
```

## Notes

- Compute region reached 1000 times;
- Data region reached 2 times.

# Data Region

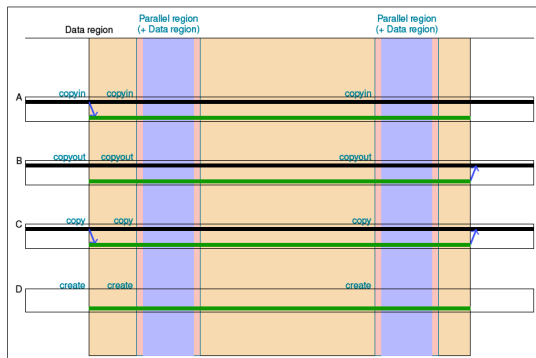
## Visual Example – Multiple data and parallel regions



- 8 transfers;
- 2 allocations.

# Data Region

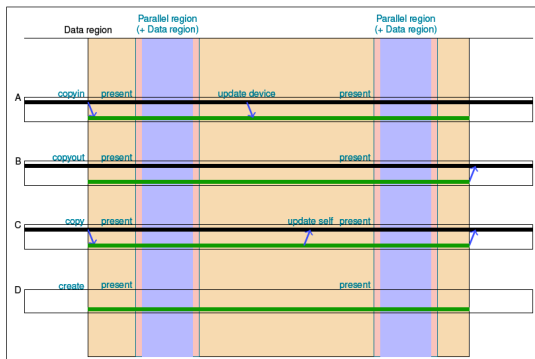
## Visual Example – Multiple parallel regions within a single data region



- 4 transfers;
- 1 allocations;
- A, B and C are now transferred at entry and exit of the data region.

# Data Region

## Visual Example – Using **update** in parallel and data regions



- 6 transfers and 1 allocation;
- Clauses check for data presence. However, it is a good practice to use the **present** clause in order to make the code clear;
- The **update** clause can be used to make sure data is up to date in the host or device.
  - The **update** clause is used inside a data region;
  - The **update** clause cannot be used inside a parallel region.

# Data Region

## Update – self or host – update-err.f90 – p=42, s=1000

```

29  !$ACC data copyout( a(:s) )
30  !$ACC parallel loop
31  do i=1,s
32    a(i) = 0
33  enddo
34  do j=1,p
35    call random_number(test)
36    rng = floor(test*100)
37    !$ACC parallel loop copyin(rng) &
38    !$ACC& copyout(a)
39    do i=1,s
40      a(i) = a(i) + rng
41    enddo
42  enddo
43  ! write(*,*) "before update self", a(p)
44  !$ACC update self(a(p:p))
45  ! write(*,*) "after update self", a(p)
46  !$ACC serial
47  a(p) = p
48  !$ACC end serial
49  write(*,*) "before_end_data", a(p)
50  !$ACC end data
51  write(*,*) "after_end_data", a(p)

```

The **self** and **host** clauses update the variable in the  $H \rightarrow D$  direction.

before end data	0
after end data	42

<pre> para: 29, Generating copyout(a(:s)) [if not already present] 30, Generating Tesla code 31, !\$acc loop gang, vector(128) ! blockidx%x threadidx%x 37, Generating copyout(a(:)) [if not already present]    Generating copyin(rng) [if not already present]    Generating Tesla code 39, !\$acc loop gang, vector(128) ! blockidx%x threadidx%x 46, Accelerator serial kernel generated    Generating Tesla code </pre>
--

The "a" array is not initialized on the host before the end of the data region in update-err.f90.

# Data Region

## Update – self or host – update-err.f90 – p=42, s=1000

```

29  !$ACC data copyout( a(:s) )
30  !$ACC parallel loop
31  do i=1,s
32      a(i) = 0
33  enddo
34  do j=1,p
35      call random_number(test)
36      rng = floor(test*100)
37      !$ACC parallel loop copyin(rng) &
38      !$ACC& copyout(a)
39      do i=1,s
40          a(i) = a(i) + rng
41      enddo
42  enddo
43  ! write(*,*) "before update self", a(p)
44  !$ACC update self(a(p:p))
45  ! write(*,*) "after update self", a(p)
46  !$ACC serial
47  a(p) = p
48  !$ACC end serial
49  write(*,*) "before_end_data", a(p)
50  !$ACC end data
51  write(*,*) "after_end_data", a(p)

```

The **self** and **host** clauses update the variable in the  $H \rightarrow D$  direction.

before end data	0
after end data	42

```

para NVIDIA devicenum=0
time(us): 497
29: data region reached 2 times
50: data copyout transfers: 1
   device time(us): total=22 max=22 min=22 avg=22
30: compute region reached 1 time
30: kernel launched 1 time
   grid: [79] block: [128]
   device time(us): total=4 max=4 min=4 avg=4
   elapsed time(us): total=362 max=362 min=362 avg=362
37: compute region reached 42 times
37: kernel launched 42 times
   grid: [79] block: [128]
   device time(us): total=86 max=3 min=2 avg=2
   elapsed time(us): total=738 max=27 min=15 avg=17
37: data region reached 84 times
37: data copyin transfers: 42
   device time(us): total=383 max=10 min=8 avg=9
46: compute region reached 1 time
46: kernel launched 1 time
   grid: [1] block: [1]
   device time(us): total=2 max=2 min=2 avg=2
   elapsed time(us): total=17 max=17 min=17 avg=17

```

The **"a"** array is not initialized on the host before the end of the data region in update-err.f90.

# Data Region

## Update – self or host – update-corr.f90 – p=42, s=1000

```

28  !$ACC data copyout( a(:s) )
29  !$ACC parallel loop
30  do i=1,s
31      a(i) = 0
32  enddo
33  do j=1,p
34      call random_number(test)
35      rng = floor(test*100)
36      !$ACC parallel loop copyin(rng) &
37      !$ACC& copyout(a)
38      do i=1,s
39          a(i) = a(i) + rng
40      enddo
41  enddo
42  write(*,*) "before_update_self", a(p)
43  !$ACC update self(a(p:p))
44  write(*,*) "after_update_self", a(p)
45  !$ACC serial
46  a(p) = p
47  !$ACC end serial
48  write(*,*) "before_end_data", a(p)
49  !$ACC update host(a(p:p))
50  write(*,*) "second_update_host", a(p)
51  !$ACC end data
52  write(*,*) "after_end_data", a(p)

```

The **self** and **host** clauses update the variable in the  $H \rightarrow D$  direction.

before update self	0
after update self	2259
before end data	2259
second update host	42
after end data	42

```

para:
28, Generating copyout(a(:s)) [if not already present]
29, Generating Tesla code
30, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
36, Generating copyout(a(:)) [if not already present]
   Generating copyin(rng) [if not already present]
   Generating Tesla code
38, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
43, Generating update self(a(p))
45, Accelerator serial kernel generated
   Generating Tesla code
49, Generating update self(a(p))

```

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



# Data Region

## Update – self or host – update-corr.f90 – p=42, s=1000

```

28  !$ACC data copyout( a(:s) )
29  !$ACC parallel loop
30  do i=1,s
31      a(i) = 0
32  enddo
33  do j=1,p
34      call random_number(test)
35      rng = floor(test*100)
36      !$ACC parallel loop copyin(rng) &
37      !$ACC& copyout(a)
38      do i=1,s
39          a(i) = a(i) + rng
40      enddo
41  enddo
42  write(*,*) "before_update_self", a(p)
43  !$ACC update self(a(p:p))
44  write(*,*) "after_update_self", a(p)
45  !$ACC serial
46  a(p) = p
47  !$ACC end serial
48  write(*,*) "before_end_data", a(p)
49  !$ACC update host(a(p:p))
50  write(*,*) "second_update_host", a(p)
51  !$ACC end data
52  write(*,*) "after_end_data", a(p)

```

The **self** and **host** clauses update the variable in the  $H \rightarrow D$  direction.

```

before update self      0
after update self       2259
before end data         2259
second update host      42
after end data          42

```

```

para NVIDIA devicenum=0
time(us): 517
28: data region reached 2 times
51: data copyout transfers: 1
   device time(us): total=9 max=9 min=9 avg=9
29: compute region reached 1 time
29: kernel launched 1 time
   grid: [79] block: [128]
   device time(us): total=3 max=3 min=3 avg=3
   elapsed time(us): total=407 max=407 min=407 avg=407
36: compute region reached 42 times
36: kernel launched 42 times
   grid: [79] block: [128]
   device time(us): total=86 max=3 min=2 avg=2
   elapsed time(us): total=745 max=38 min=16 avg=17
36: data region reached 84 times
36: data copyin transfers: 42
   device time(us): total=391 max=18 min=8 avg=9
43: update directive reached 1 time
43: data copyout transfers: 1
   device time(us): total=21 max=21 min=21 avg=21
45: compute region reached 1 time
45: kernel launched 1 time
   grid: [1] block: [1]
   device time(us): total=3 max=3 min=3 avg=3
   elapsed time(us): total=32 max=32 min=32 avg=32
49: update directive reached 1 time
49: data copyout transfers: 1
   device time(us): total=4 max=4 min=4 avg=4

```

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

# Data Region

## Update – **device**

The **device** clause updates the variable in the  $D \rightarrow H$  direction.

# Data Region

## Global Data Regions – declare

**Important Notes** The lifetime of data inside the **declare** directive is the same lifetime of the scope of the code region where it is used. Example:

Zone	Scope
Module	Summ
Function	Product
Subroutine	Maximum

```
loop:
13: Generating Tesla code
14: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
13: Generating implicit copyout{a(:)} [if not already present]
```

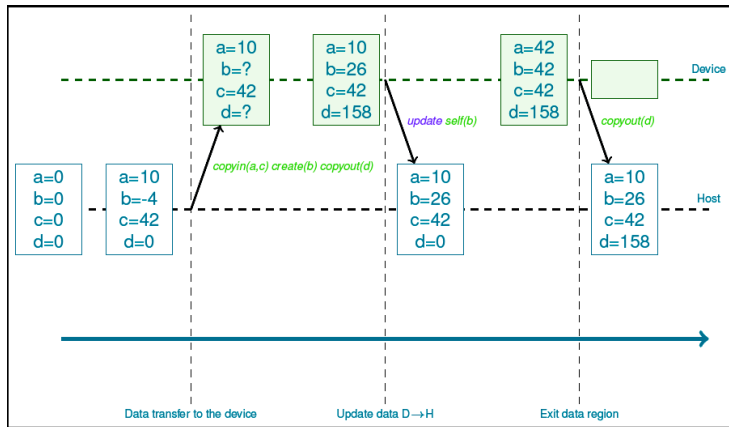
```
Accelerator Kernel Timing data
loop NVIDIA devicenum=0
time(us): 15
13: compute region reached 1 time
13: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=4 max=4 min=4 avg=4
elapsed time(us): total=360 max=360 min=360 avg=360
13: data region reached 2 times
17: data copyout transfers: 1
device time(us): total=11 max=11 min=11 avg=11
```

**module.f90**

```
4 module var
5   integer :: i
6   integer, parameter :: maxi=10000
7   integer :: a(maxi)
8   !$ACC declare copyout( i, maxi, a(:) )
9 end module var
10
11 program loop
12   use var
13   !$ACC parallel loop
14   do i=1,maxi
15     a(i) = i
16   enddo
17   write(*,*) a(maxi)
18 end program loop
```

# Data Region

## Time line



# Data Region

## Important Notes

- Data transfers between the host and the device are costly;
- It is mandatory to minimize these transfers to achieve good performance;
- It is possible to use data clauses within **kernels/parallel** and/or **data** regions;
  - The **update** directive can be used to avoid unexpected behaviors.

# Asynchronism

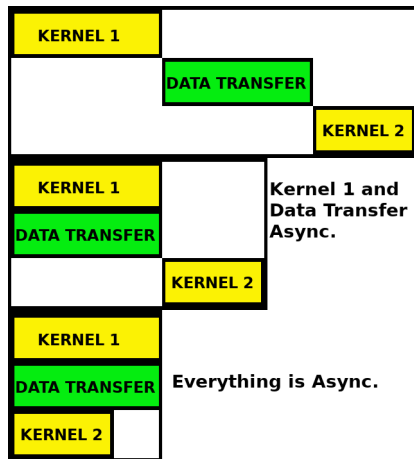
## Introduction

- By default kernels are executed synchronously;
- The accelerator is able to manage several execution threads, running concurrently;
- In order to achieve better performance it is recommended to maximize overlaps between:
  - Computations and data transfers;
  - kernel/kernel if they are independent.

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.



# Asynchronism

sync.f90 – s=10000

```

25  !$ACC enter data create( a(1:s), b(1:s), &
26  !$ACC& c(1:s) )
27  ! b is initialized on host
28  do i=1,s
29      b(i) = i
30  enddo
31  !$ACC parallel loop
32  do i=1,s
33      a(i) = 42
34  enddo
35  ! Update vector b located on device
36  ! with data from the host
37  !$ACC update device(b)
38  !$ACC parallel loop
39  do i=1,s
40      c(i) = 1
41  enddo
42  !$ACC exit data delete( a(1:s), b(1:s), &
43  !$ACC& c(1:s) )

```

```

async:
25: Generating enter data create(c(1:s),b(1:s),a(1:s))
31: Generating Tesla code
32: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
31: Generating implicit copyout(a(1:10000)) [if not already present]
37: Generating update device(b(:))
38: Generating Tesla code
39: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
38: Generating implicit copyout(c(1:10000)) [if not already present]
42: Generating exit data delete(c(1:s),b(1:s),a(1:s))

```

```

async: NVIDIA devicenum=0
time(us): 28
25: data region reached 1 time
31: compute region reached 1 time
31: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=4 max=4 min=4 avg=4
elapsed time(us): total=357 max=357 min=357 avg=357
31: data region reached 2 times
37: update directive reached 1 time
37: data copyin transfers: 1
device time(us): total=19 max=19 min=19 avg=19
38: compute region reached 1 time
38: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=3 max=3 min=3 avg=3
elapsed time(us): total=30 max=30 min=30 avg=30
38: data region reached 2 times
42: data region reached 1 time

```

# Asynchronism

async-1.f90 – s=10000

```

25  !$ACC enter data create( a(1:s), b(1:s), &
26  !$ACC& c(1:s) )
27  ! b is initialized on host
28  do i=1,s
29      b(i) = i
30  enddo
31  !$ACC parallel loop async(1)
32  do i=1,s
33      a(i) = 42
34  enddo
35  ! Update vector b located on device
36  ! with data from the host
37  !$ACC update device(b)
38  !$ACC parallel loop
39  do i=1,s
40      c(i) = 1
41  enddo
42  !$ACC exit data delete( a(1:s), b(1:s), &
43  !$ACC& c(1:s) )

```

```

async:
25: Generating enter data create(c(1:s),b(1:s),a(1:s))
31: Generating Tesla code
32: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
31: Generating implicit copyout(a(1:10000)) [if not already present]
37: Generating update device(b(:))
38: Generating Tesla code
39: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
38: Generating implicit copyout(c(1:10000)) [if not already present]
42: Generating exit data delete(c(1:s),b(1:s),a(1:s))

```

```

async: NVIDIA devicenum=0
time(us): 23
25: data region reached 1 time
31: compute region reached 1 time
31: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=4 max=4 min=4 avg=4
elapsed time(us): total=355 max=355 min=355 avg=355
31: data region reached 2 times
37: update directive reached 1 time
37: data copyin transfers: 1
device time(us): total=16 max=16 min=16 avg=16
38: compute region reached 1 time
38: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=3 max=3 min=3 avg=3
elapsed time(us): total=33 max=33 min=33 avg=33
38: data region reached 2 times
42: data region reached 1 time

```



# Asynchronism

async-2.f90 – s=10000

```

25  !$ACC enter data create( a(1:s), b(1:s), &
26  !$ACC& c(1:s) )
27  ! b is initialized on host
28  do i=1,s
29      b(i) = i
30  enddo
31  !$ACC parallel loop async(1)
32  do i=1,s
33      a(i) = 42
34  enddo
35  ! Update vector b located on device
36  ! with data from the host
37  !$ACC update device(b) async(2)
38  !$ACC parallel loop
39  do i=1,s
40      c(i) = 1
41  enddo
42  !$ACC exit data delete( a(1:s), b(1:s), &
43  !$ACC& c(1:s) )

```

```

async:
25: Generating enter data create(c(1:s),b(1:s),a(1:s))
31: Generating Tesla code
32: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
31: Generating implicit copyout(a(1:10000)) [if not already present]
37: Generating update device(b(:))
38: Generating Tesla code
39: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
38: Generating implicit copyout(c(1:10000)) [if not already present]
42: Generating exit data delete(c(1:s),b(1:s),a(1:s))

```

```

Timing may be affected by asynchronous behavior
set PGIACC_SYNCHRONOUS to 1 to disable async() clauses
async: NVIDIA devicenum=0
time(us): 25
25: data region reached 1 time
31: compute region reached 1 time
31: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=4 max=4 min=4 avg=4
elapsed time(us): total=378 max=378 min=378 avg=378
31: data region reached 2 times
37: update directive reached 1 time
37: data copyin transfers: 1
device time(us): total=17 max=17 min=17 avg=17
38: compute region reached 1 time
38: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=4 max=4 min=4 avg=4
elapsed time(us): total=35 max=35 min=35 avg=35
38: data region reached 2 times
42: data region reached 1 time

```

# Asynchronism

async-3.f90 – s=10000

```

25  !$ACC enter data create( a(1:s), b(1:s), &
26  !$ACC& c(1:s) )
27  ! b is initialized on host
28  do i=1,s
29      b(i) = i
30  enddo
31  !$ACC parallel loop async(1)
32  do i=1,s
33      a(i) = 42
34  enddo
35  ! Update vector b located on device
36  ! with data from the host
37  !$ACC update device(b) async(2)
38  !$ACC parallel loop async(3)
39  do i=1,s
40      c(i) = 1
41  enddo
42  !$ACC exit data delete( a(1:s), b(1:s), &
43  !$ACC& c(1:s) )

```

```

async:
25: Generating enter data create(c(1:s),b(1:s),a(1:s))
31: Generating Tesla code
32: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
31: Generating implicit copyout(a(1:10000)) [if not already present]
37: Generating update device(b(:))
38: Generating Tesla code
39: !$acc loop gang, vector(128) ! blockidx%x threadidx%x
38: Generating implicit copyout(c(1:10000)) [if not already present]
42: Generating exit data delete(c(1:s),b(1:s),a(1:s))

```

```

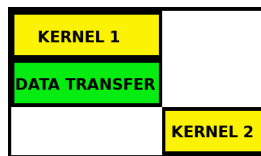
Timing may be affected by asynchronous behavior
set PGIACC_SYNCHRONOUS to 1 to disable async() clauses
async NVIDIA devicenum=0
time(us): 22
25: data region reached 1 time
31: compute region reached 1 time
31: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=4 max=4 min=4 avg=4
elapsed time(us): total=369 max=369 min=369 avg=369
31: data region reached 2 times
37: update directive reached 1 time
37: data copyin transfers: 1
device time(us): total=15 max=15 min=15 avg=15
38: compute region reached 1 time
38: kernel launched 1 time
grid: [79] block: [128]
device time(us): total=3 max=3 min=3 avg=3
elapsed time(us): total=33 max=33 min=33 avg=33
38: data region reached 2 times
42: data region reached 1 time

```

# Asynchronism

## Wait Clause

- The **wait** clause can be used in of dependent kernels;
- **Wait** can be used alone or with a list of **kernels** numbers, *i.e.* **wait(1,2)**.



# Asynchronism

wait.f90 – s=10000

```

25  !$ACC enter data create( a(1:s), b(1:s), &
26  !$ACC& c(1:s) )
27  ! b is initialized on host
28  do i=1,s
29      b(i) = i
30  enddo
31  !$ACC parallel loop async(1)
32  do i=1,s
33      a(i) = 42
34  enddo
35  ! Update vector b located on device
36  ! with data from the host
37  !$ACC update device(b) async(2)
38  !$ACC parallel loop async(3)
39  do i=1,s
40      c(i) = 1
41  enddo
42  !$ACC parallel loop wait(2)
43  do i=1,s
44      b(i) = b(i)*i
45  enddo
46  !$ACC exit data delete( a(1:s), b(1:s), &
47  !$ACC& c(1:s) )

```

```

async:
25, Generating enter data create(b(1:s),c(1:s),a(1:s))
31, Generating Tesla code
32, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
31, Generating implicit copyout(a(1:10000)) [if not already present]
37, Generating update device(b(:))
38, Generating Tesla code
39, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
38, Generating implicit copyout(c(1:10000)) [if not already present]
42, Generating Tesla code
43, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
42, Generating implicit copy(b(1:10000)) [if not already present]
46, Generating exit data delete(c(1:s),b(1:s),a(1:s))

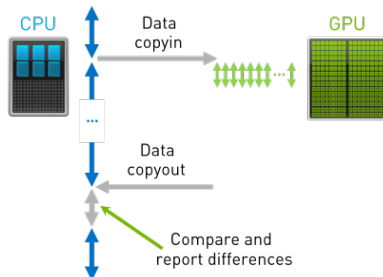
```

Timing may be affected by asynchronous behavior  
 set PGIACC\_SYNCHRONOUS to 1 to disable async() clauses  
 async NVIDIA devicenum=0  
 time(us): 28  
 25: data region reached 1 time  
 31: compute region reached 1 time  
 31: kernel launched 1 time  
 grid: [79] block: [128]  
 device time(us): total=4 max=4 min=4 avg=4  
 elapsed time(us): total=382 max=382 min=382 avg=382  
 31: data region reached 2 times  
 37: update directive reached 1 time  
 37: data copyin transfers: 1  
 device time(us): total=16 max=16 min=16 avg=16  
 38: compute region reached 1 time  
 38: kernel launched 1 time  
 grid: [79] block: [128]  
 device time(us): total=3 max=3 min=3 avg=3  
 elapsed time(us): total=34 max=34 min=34 avg=34  
 38: data region reached 2 times  
 42: compute region reached 1 time  
 42: kernel launched 1 time  
 grid: [79] block: [128]  
 device time(us): total=5 max=5 min=5 avg=5  
 elapsed time(us): total=21 max=21 min=21 avg=21  
 42: data region reached 2 times  
 46: data region reached 1 time

# GPU Debugging

## PGI Auto-compare for OpenACC

- Results can diverge between programs running on a CPU versus a GPU due to programming errors, precision of numerical intrinsics, or variations in compiler optimizations.
- OpenACC auto-compare runs compute regions redundantly on both the CPU and GPU.
- When data is copied from the GPU back to the CPU, GPU results are compared with those computed on the CPU.
- Auto-compare works on both structured and unstructured data regions, with difference reports controlled by environment variables to quickly pinpoint where results start to diverge and adapt the program or compiler options as needed.



# GPU Debugging

## PGI Auto-compare for OpenACC

The auto-compare is activated during compilation as the race condition example, **pb-sync.f90**:

● **pgf90 -acc -ta=tesla:cc35,cc60,autocompare -Minfo=accel pb-sync.f90**

```

23  !$acc parallel
24  !$acc loop gang
25  do i=1,nx
26      a(i) = 1.0_8
27  enddo
28  !$acc loop gang reduction(+:summ)
29  do i=nx,1,-1
30      summ = summ + a(i)
31  enddo
32  !$acc end parallel
  
```

```

reduction:
23: Generating Tesla code
25: !$acc loop gang, vector(128) ! blockid%x% threadid%x%
29: !$acc loop gang, vector(128) ! blockid%x% threadid%x%
Generating reduction(+:summ)
23: Generating implicit copy(summ) [if not already present]
Generating implicit copyout(a(:nx)) [if not already present]
  
```

```

PCAST Float summ in function reduction, pb-sync.f90:32
idx: 0 FAIL ABS act: 5.00267200e+07 exp: 1.67772160e+07 dif: 3.32495040e+07
compared 2 blocks, 100000001 elements, 400000004 bytes
1 errors found in 1 blocks
absolute tolerance = 0.000000000000000000e+00, abs=0
  
```

```

Accelerator Kernel Timing data
pb-sync.f90
reduction NVIDIA devicenum=0
time(us): 41,027
23: compute region reached 1 time
23: kernel launched 1 time
grid: [65536] block: [128]
device time(us): total=9,036 max=9,036 min=9,036 avg=9,036
elapsed time(us): total=9,077 max=9,077 min=9,077 avg=9,077
23: reduction kernel launched 1 time
grid: [1] block: [256]
device time(us): total=83 max=83 min=83 avg=83
elapsed time(us): total=111 max=111 min=111 avg=111
23: data region reached 2 times
23: data copyin transfers: 1
device time(us): total=7 max=7 min=7 avg=7
32: data copyout transfers: 25
device time(us): total=31,901 max=1,433 min=12 avg=1,276
  
```

# Optimization Example - Game of Life

4000x2000 - 100 gen.

## Serial

```

22 CALL CPU.TIME(TIME.INI)
23 allocate(world(0:rows+1,0:cols+1), old.world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill.world
26 do g=1,generations
27   cells = 0
28   do r=1, rows
29     do c=1, cols
30       old_world(r,c) = world(r,c)
31     enddo
32   enddo
33   do r=1, rows
34     do c=1, cols
35       neigh = old.world(r-1,c-1)+old.world(r,c-1)+&
36         old.world(r+1,c-1)+old.world(r-1,c)+old.world(r+1,c)+&
37         old.world(r-1,c+1)+old.world(r,c+1)+old.world(r+1,c+1)
38       if (old.world(r,c) == 1 .and. (neigh<2or.neigh>3)) then
39         world(r,c) = 0
40       else if (neigh == 3) then
41         world(r,c) = 1
42       endif
43     enddo
44   enddo
45   do r=1, rows
46     do c=1, cols
47       cells = cells + world(r,c)
48     enddo
49   enddo
50   print *, "Cells_alive_at-generation=", g, ":", cells
51 enddo
52 deallocate(world, old.world)
53 CALL CPU.TIME(TIME.END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	10.29104113578796	

make.out

```

pgf90 -O0 -acc=noautopar -ta=tesla:cc35,cc60 -Minfo=all -c gol.f90 -o gol.o
pgf90 -O0 -acc=noautopar -ta=tesla:cc35,cc60 -Minfo=all gol.o -o gol

```

- Gardener, M. (1970). MATHEMATICAL GAMES: The fantastic combinations of John Conway's new solitaire game "life," *Scientific American*, 223, 120-123.
- Wainwright, R. (2010). Conway's Game of Life: Early Personal Recollections. In: Adamatzky A. (eds) *Game of Life Cellular Automata*. Springer, London.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Serial

```

22 CALL CPU.TIME(TIME,INI)
23 allocate (world (0:rows+1,0:cols+1), old_world (0:rows+1,0:cols+1))
24 old_world (:,:) = 0
25 call fill_world
26 do g=1,generations
27   cells = 0
28   do r=1, rows
29     do c=1, cols
30       old_world(r,c) = world(r,c)
31     enddo
32   enddo
33   do r=1, rows
34     do c=1, cols
35       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
36         old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
37         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
38       if (old_world(r,c) == 1 .and. (neigh < 2 or neigh > 3)) then
39         world(r,c) = 0
40       else if (neigh == 3) then
41         world(r,c) = 1
42       endif
43     enddo
44   enddo
45   do r=1, rows
46     do c=1, cols
47       cells = cells + world(r,c)
48     enddo
49   enddo
50   print *, "Cells alive at generation=", g, ":", cells
51 enddo
52 deallocate(world, old_world)
53 CALL CPU.TIME(TIME,END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	10.29104113578796	

make.out

```

pgf90 -O0 -acc=noautopar -ta=tesla:cc35,cc60 -Minfo=all -c gol.f90 -o gol.o
pgf90 -O0 -acc=noautopar -ta=tesla:cc35,cc60 -Minfo=all gol.o -o gol

```

Intel Core i7-8700 3.2GHz 12Mo Smart Cache  
(up to 4.60 GHz)

Prix: 380,90 €





# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #1

```

22 CALL CPU.TIME(TIME.INI)
23 allocate(world(0:rows+1,0:cols+1), old.world(0:rows+1,0:cols+1))
24 old.world(:,:) = 0
25 call fill.world
26 do g=1,generations
27   cells = 0
28   !$ACC parallel
29   do r=1, rows
30     do c=1, cols
31       old.world(r,c) = world(r,c)
32     enddo
33   enddo
34   !$ACC end parallel
35   do r=1, rows
36     do c=1, cols
37       neigh = old.world(r-1,c-1)+old.world(r,c-1)+&
38         old.world(r+1,c-1)+old.world(r-1,c)+&
39         old.world(r-1,c+1)+old.world(r,c+1)+old.world(r+1,c+1)
40       if (old.world(r,c) == 1 .and. (neigh<2or.neigh>3)) then
41         world(r,c) = 0
42       else if (neigh == 3) then
43         world(r,c) = 1
44       endif
45     enddo
46   enddo
47   do r=1, rows
48     do c=1, cols
49       cells = cells + world(r,c)
50     enddo
51   enddo
52   print *, "Cells_alive_at_generation=", g, ":", cells
53 enddo
54 deallocate(world, old.world)
55 CALL CPU.TIME(TIME.END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	34.91349506378174	

make.err

```

gol:
28, Generating Tesla code
29, !$acc loop seq
30, !$acc loop seq
28, Generating implicit copyin(world{1:rows,1:cols})[if not already present]
Generating implicit copyout(old.world{1:rows,1:cols})[if not already present]
29, Loop is parallelizable
30, Loop is parallelizable

```

Nvidia GeForce GTX 1050 2Go

Price: 161,98 €



# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #1

```

22 CALL CPU.TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 do g=1,generations
27   cells = 0
28   !$ACC parallel
29   do r=1, rows
30     do c=1, cols
31       old_world(r,c) = world(r,c)
32     enddo
33   enddo
34   !$ACC end parallel
35   do r=1, rows
36     do c=1, cols
37       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
38       old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
39       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
40       if (old_world(r,c) == 1 .and. (neigh < 2 or neigh > 3)) then
41         world(r,c) = 0
42       else if (neigh == 3) then
43         world(r,c) = 1
44       endif
45     enddo
46   enddo
47   do r=1, rows
48     do c=1, cols
49       cells = cells + world(r,c)
50     enddo
51   enddo
52   print *, "Cells alive at generation=", g, ":", cells
53 enddo
54 deallocate(world, old_world)
55 CALL CPU.TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=		34.91349506378174

make.err

```

gol:
28, Generating Tesla code
29, !$acc loop seq
30, !$acc loop seq
28, Generating implicit copyin(world(1:rows,1:cols))[if not already present]
Generating implicit copyout(old_world(1:rows,1:cols))[if not already present]
29, Loop is parallelizable
30, Loop is parallelizable

```

- fill\_world routine cannot be parallelized with OpenACC. Random number routine is not yet available to the PGI 19.10 compiler;
- noautopar is used in order to be in control of the optimization procedure;
- The generation loop presents a dependency and cannot be parallelized;
- Time is increasing due to gang redundancy from **\$ACC parallel** directive.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #1

```

22 CALL CPU.TIME(TIME.INI)
23 allocate(world(0:rows+1,0:cols+1), old.world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 do g=1, generations
27   cells = 0
28   !$ACC parallel
29   do r=1, rows
30     do c=1, cols
31       old.world(r,c) = world(r,c)
32     enddo
33   enddo
34   !$ACC end parallel
35   do r=1, rows
36     do c=1, cols
37       neigh = old.world(r-1,c-1)+old.world(r,c-1)+&
38       old.world(r+1,c-1)+old.world(r-1,c)+&
39       old.world(r-1,c+1)+old.world(r,c+1)+old.world(r+1,c+1)
40       if (old.world(r,c) == 1 .and. (neigh < 2 or neigh > 3)) then
41         world(r,c) = 0
42       else if (neigh == 3) then
43         world(r,c) = 1
44       endif
45     enddo
46   enddo
47   do r=1, rows
48     do c=1, cols
49       cells = cells + world(r,c)
50     enddo
51   enddo
52   print *, "Cells alive at generation=", g, ":", cells
53 enddo
54 deallocate(world, old.world)
55 CALL CPU.TIME(TIME.END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	34.91349506378174	

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 24,307,904
28: compute region reached 50 times
28: kernel launched 50 times
grid: [1] block: [1]
device time(us): total=24,056,960 max=545,493 min=475,765 avg=481,139
elapsed time(us): total=24,059,713 max=545,542 min=475,800 avg=481,194
28: data region reached 100 times
28: data copyin transfers: 100
device time(us): total=123,399 max=1,432 min=1,166 avg=1,233
34: data copyout transfers: 100
device time(us): total=127,545 max=1,492 min=1,200 avg=1,275

```

- fill\_world routine cannot be parallelized with OpenACC. Random number routine is not yet available to the PGI 19.10 compiler;
- noautopar is used in order to be in control of the optimization procedure;
- The generation loop presents a dependency and cannot be parallelized;
- Time is increasing due to gang redundancy from **\$ACC parallel** directive.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #2

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 do g=1,generations
27   cells = 0
28   !$ACC parallel
29   do r=1, rows
30     do c=1, cols
31       old_world(r,c) = world(r,c)
32     enddo
33   enddo
34   !$ACC end parallel
35   !$ACC parallel
36   do r=1, rows
37     do c=1, cols
38       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
39       old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
40       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
41       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3)) then
42         world(r,c) = 0
43       else if (neigh == 3) then
44         world(r,c) = 1
45       endif
46     enddo
47   enddo
48   !$ACC end parallel
49   do r=1, rows
50     do c=1, cols
51       cells = cells + world(r,c)
52     enddo
53   enddo
54   print *, "Cells_alive_at_generation_", g, ":", cells
55 enddo
56 deallocate(world, old_world)
57 CALL CPU_TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	80.00242805480957	

make.err

```

gol:
28, Generating Tesla code
29, !$acc loop seq
30, !$acc loop seq
28, Generating implicit copyout(old_world(1:rows,1:cols)) [if not already present]
Generating implicit copyin(world(1:rows,1:cols)) [if not already present]
29, Loop is parallelizable
30, Loop is parallelizable
35, Generating Tesla code
36, !$acc loop seq
37, !$acc loop seq
35, Generating implicit copy(world(1:rows,1:cols)) [if not already present]
Generating implicit copyin(old_world(0:rows+1,0:cols+1)) [if not already present]
36, Loop is parallelizable
37, Loop is parallelizable

```

- 2nd loop is parallelized;
- Time is increasing due to gang redundancy from **\$ACC parallel** directive.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #2

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 do g=1,generations
27   cells = 0
28   !$ACC parallel
29   do r=1, rows
30     do c=1, cols
31       old_world(r,c) = world(r,c)
32     enddo
33   enddo
34   !$ACC end parallel
35   !$ACC parallel
36   do r=1, rows
37     do c=1, cols
38       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
39         old_world(r+1,c-1)+old_world(r-1,c)+&
40         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
41       if (old_world(r,c) == 1 .and. (neigh < 2 or neigh > 3)) then
42         world(r,c) = 0
43       else if (neigh == 3) then
44         world(r,c) = 1
45       endif
46     enddo
47   enddo
48   !$ACC end parallel
49   do r=1, rows
50     do c=1, cols
51       cells = cells + world(r,c)
52     enddo
53   enddo
54   print *, "Cells_alive_at_generation_", g, ":", cells
55 enddo
56 deallocate(world, old_world)
57 CALL CPU_TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	80.00242805480957	

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA device=0
time(us): 68,714,075
28: compute region reached 50 times
28: kernel launched 50 times
grid: [1] block: [1]
device time(us): total=24,131,149 max=546,650 min=478,350 avg=482,622
elapsed time(us): total=24,133,536 max=546,708 min=478,388 avg=482,670
28: data region reached 100 times
28: data copyin transfers: 100
device time(us): total=123,208 max=1,554 min=1,166 avg=1,232
34: data copyout transfers: 100
device time(us): total=127,980 max=1,515 min=1,204 avg=1,279
35: compute region reached 50 times
35: kernel launched 50 times
grid: [1] block: [1]
device time(us): total=43,957,428 max=884,821 min=868,052 avg=879,148
elapsed time(us): total=43,959,444 max=884,855 min=868,102 avg=879,188
35: data region reached 100 times
35: data copyin transfers: 200
device time(us): total=246,842 max=1,531 min=1,162 avg=1,234
48: data copyout transfers: 100
device time(us): total=127,468 max=1,394 min=1,204 avg=1,274

```

- 2nd loop is parallelized;
- Time is increasing due to gang redundancy from **\$ACC parallel** directive.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #3

```

22 CALL CPU.TIME(TIME_INIT)
23 allocate (world(0:rows+1,0:cols+1), old.world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 do g=1,generations
27   !$ACC parallel
28   do r=1, rows
29     do c=1, cols
30       old_world(r,c) = world(r,c)
31     enddo
32   enddo
33   !$ACC end parallel
34   !$ACC parallel
35   do r=1, rows
36     do c=1, cols
37       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
38       old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
39       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
40       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
41         world(r,c) = 0
42       else if (neigh == 3) then
43         world(r,c) = 1
44       endif
45     enddo
46   enddo
47   !$ACC end parallel
48   cells = 0
49   !$ACC parallel
50   do r=1, rows
51     do c=1, cols
52       cells = cells + world(r,c)
53     enddo
54   enddo
55   !$ACC end parallel
56   print *, "Cells_alive_at_generation=", g, ":", cells
57 enddo
58 deallocate(world, old.world)
59 CALL CPU.TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=		87.45015597343445

make.err

```

gol:
27, Generating Tesla code
28, !$acc loop seq
29, !$acc loop seq
27, Generating implicit copyout(old.world(1:rows,1:cols)) [if not already present]
Generating implicit copyin(world(1:rows,1:cols)) [if not already present]
28, Loop is parallelizable
29, Loop is parallelizable
34, Generating Tesla code
35, !$acc loop seq
36, !$acc loop seq
34, Generating implicit copy(world(1:rows,1:cols)) [if not already present]
Generating implicit copyin(old.world(0:rows+1,0:cols+1)) [if not already present]
35, Loop is parallelizable
36, Loop is parallelizable
49, Generating Tesla code
50, !$acc loop seq
51, !$acc loop seq
52, Generating implicit reduction(+:cells)
49, Generating implicit copyin(world(1:rows,1:cols)) [if not already present]
50, Loop is parallelizable
51, Loop is parallelizable

```

- 3rd loop is parallelized;
- A **reduction** clause is proposed by the compiler;
- Time is increasing due to gang redundancy from **\$ACC parallel** directive.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #3

```

22 CALL CPU_TIME(TIME_INI)
23 allocate (world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 do g=1,generations
27   !$ACC parallel
28   do r=1, rows
29     do c=1, cols
30       old_world(r,c) = world(r,c)
31     enddo
32   enddo
33   !$ACC end parallel
34   !$ACC parallel
35   do r=1, rows
36     do c=1, cols
37       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
38         old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
39         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
40       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3)) then
41         world(r,c) = 0
42       else if (neigh == 3) then
43         world(r,c) = 1
44       endif
45     enddo
46   enddo
47   !$ACC end parallel
48   cells = 0
49   !$ACC parallel
50   do r=1, rows
51     do c=1, cols
52       cells = cells + world(r,c)
53     enddo
54   enddo
55   !$ACC end parallel
56   print *, "Cells_alive_at_generation_", g, ":", cells
57 enddo
58 deallocate(world, old_world)
59 CALL CPU_TIME(TIME_END)

```

gol.er

```

Accelerator Kernel Timing data
gol NVIDIA deviceum=0
time(us): 76,131,242
27: compute region reached 50 times
27: kernel launched 50 times
grid: [1] block: [1]
device time(us): total=24,040,550 max=554,250 min=475,706 avg=480,811
elapsed time(us): total=24,042,408 max=554,314 min=475,740 avg=480,848
27: data region reached 100 times
27: data copyin transfers: 100
device time(us): total=123,469 max=1,388 min=1,162 avg=1,234
33: data copyout transfers: 100
device time(us): total=127,652 max=1,465 min=1,202 avg=1,276
34: compute region reached 50 times
34: kernel launched 50 times
grid: [1] block: [1]
device time(us): total=43,810,816 max=881,837 min=858,809 avg=876,216
elapsed time(us): total=43,812,794 max=881,876 min=858,845 avg=876,255
34: data region reached 100 times
34: data copyin transfers: 200
device time(us): total=246,924 max=1,497 min=1,162 avg=1,234
47: data copyout transfers: 100
device time(us): total=128,736 max=1,759 min=1,204 avg=1,287
49: compute region reached 50 times
49: data copyin transfers: 50
device time(us): total=396 max=35 min=7 avg=7
49: kernel launched 50 times
grid: [1] block: [1]
device time(us): total=7,528,219 max=152,493 min=149,245 avg=150,564
elapsed time(us): total=7,529,896 max=152,531 min=149,279 avg=150,597
49: reduction kernel launched 50 times
grid: [1] block: [256]
device time(us): total=148 max=3 min=2 avg=2
elapsed time(us): total=2,029 max=75 min=23 avg=40
49: data region reached 100 times
49: data copyin transfers: 100
device time(us): total=123,516 max=1,471 min=1,164 avg=1,235
55: download reached 50 times
55: data copyout transfers: 50
device time(us): total=816 max=23 min=8 avg=16

```

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #4

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 do g=1,generations
27   cells = 0
28   !$ACC parallel copy(cells)
29   !$ACC loop
30   do r=1, rows
31     do c=1, cols
32       old_world(r,c) = world(r,c)
33     enddo
34   enddo
35   !$ACC loop
36   do r=1, rows
37     do c=1, cols
38       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
39       old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
40       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
41       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3)) then
42         world(r,c) = 0
43       else if (neigh == 3) then
44         world(r,c) = 1
45       endif
46     enddo
47   enddo
48   !$ACC loop reduction(+:cells)
49   do r=1, rows
50     do c=1, cols
51       cells = cells + world(r,c)
52     enddo
53   enddo
54   !$ACC end parallel
55   print *, "Cells=alive=at=generation=", g, ":", cells
56 enddo
57 deallocate(world, old_world)
58 CALL CPU_TIME(TIME_END)

```

gol.out

```

Cells alive at generation      45 :      997185
Cells alive at generation      46 :      988790
Cells alive at generation      47 :      981380
Cells alive at generation      48 :      974253
Cells alive at generation      49 :      967943
Cells alive at generation      50 :      959634
Total time=      11.68190693855286

```

make.err

```

gol:
28, Generating Tesla code
30, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
31, !$acc loop seq
36, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
37, !$acc loop seq
49, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
Generating reduction(+:cells)
50, !$acc loop seq
28, Generating copy(cells) [if not already present]
Generating implicit copyin(old_world(0:rows+1,0:cols+1)) [if not already present]
Generating implicit copyout(old_world(1:rows,1:cols)) [if not already present]
Generating implicit copy(world(1:rows,1:cols)) [if not already present]
31, Loop is parallelizable
37, Loop is parallelizable
50, Loop is parallelizable

```

- One **parallel** region and multiple **loop** directives;
- A **reduction** clause is used on the last parallel loop;
- GPWPVP level;
- Still more expensive than serial solution.



# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #4

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 do g=1,generations
27   cells = 0
28   !$ACC parallel copy(cells)
29   !$ACC loop
30   do r=1, rows
31     do c=1, cols
32       old_world(r,c) = world(r,c)
33     enddo
34   enddo
35   !$ACC loop
36   do r=1, rows
37     do c=1, cols
38       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
39       old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
40       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
41       if (old_world(r,c) == 1 .and. (neigh < 2or.neigh > 3)) then
42         world(r,c) = 0
43       else if (neigh == 3) then
44         world(r,c) = 1
45       endif
46     enddo
47   enddo
48   !$ACC loop reduction(+:cells)
49   do r=1, rows
50     do c=1, cols
51       cells = cells + world(r,c)
52     enddo
53   enddo
54   !$ACC end parallel
55   print *, "Cells=alive=at=generation=", g, ":", cells
56 enddo
57 deallocate(world, old_world)
58 CALL CPU_TIME(TIME_END)

```

gol.out

```

Cells alive at generation      45 :      997185
Cells alive at generation      46 :      988790
Cells alive at generation      47 :      981380
Cells alive at generation      48 :      974253
Cells alive at generation      49 :      967943
Cells alive at generation      50 :      959634
Total time=      11.68190693855286

```

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 623,314
28: compute region reached 50 times
28: kernel launched 50 times
   grid: [32] block: [128]
       device time(us): total=121,508 max=2,603 min=2,378 avg=2,430
       elapsed time(us): total=123,326 max=2,633 min=2,407 avg=2,466
28: reduction kernel launched 50 times
   grid: [1] block: [256]
       device time(us): total=193 max=4 min=3 avg=3
       elapsed time(us): total=987 max=31 min=17 avg=19
28: data region reached 200 times
28: data copyin transfers: 250
       device time(us): total=246,221 max=1,448 min=5 avg=984
54: data copyout transfers: 250
       device time(us): total=255,392 max=1,576 min=5 avg=1,021

```

- One **parallel** region and multiple **loop** directives;
- A **reduction** clause is used on the last parallel loop;
- GPWPVP level;
- Still more expensive than serial solution.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #5

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old_world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop
32   do r=1, rows
33     do c=1, cols
34       old_world(r,c) = world(r,c)
35     enddo
36   enddo
37   !$ACC loop
38   do r=1, rows
39     do c=1, cols
40       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
41         old_world(r+1,c-1)+old_world(r-1,c)+&
42         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
43       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
44         world(r,c) = 0
45       else if (neigh == 3) then
46         world(r,c) = 1
47       endif
48     enddo
49   enddo
50   !$ACC loop reduction(+:cells)
51   do r=1, rows
52     do c=1, cols
53       cells = cells + world(r,c)
54     enddo
55   enddo
56   !$ACC end parallel
57   print *, "Cells_alive_at_generation_", g, ":", cells
58 enddo
59 !$ACC end data
60 deallocate(world, old_world)
61 CALL CPU_TIME(TIME_END)

```

gol.out

```

Cells alive at generation      45 :      997185
Cells alive at generation      46 :      988790
Cells alive at generation      47 :      981380
Cells alive at generation      48 :      974253
Cells alive at generation      49 :      967943
Cells alive at generation      50 :      959634
Total time=      10.53407311439514

```

make.err

```

gol:
26, Generating create(old_world(0:rows+1,0:cols+1)) [if not already present]
Generating copy(world(0:rows+1,0:cols+1)) [if not already present]
30, Generating copy(cells) [if not already present]
Generating Tesla code
32, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
33, !$acc loop seq
38, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
39, !$acc loop seq
51, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
Generating reduction(+:cells)
52, !$acc loop seq
33, Loop is parallelizable
39, Loop is parallelizable
52, Loop is parallelizable

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- A **reduction** clause is used on the last parallel loop;
- GPWPVP level;
- Same cost of serial solution;
- Non-contiguous nested loops.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #5

```

22 CALL CPU.TIME(TIME.INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC& create (old_world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop
32   do r=1, rows
33     do c=1, cols
34       old_world(r,c) = world(r,c)
35     enddo
36   enddo
37   !$ACC loop
38   do r=1, rows
39     do c=1, cols
40       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
41         old_world(r+1,c-1)+old_world(r,c)+old_world(r+1,c)+&
42         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
43       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
44         world(r,c) = 0
45       else if (neigh == 3) then
46         world(r,c) = 1
47       endif
48     enddo
49   enddo
50   !$ACC loop reduction(+:cells)
51   do r=1, rows
52     do c=1, cols
53       cells = cells + world(r,c)
54     enddo
55   enddo
56   !$ACC end parallel
57   print *, "Cells_alive_at_generation-", g, ":", cells
58 enddo
59 !$ACC end data
60 deallocate(world, old_world)
61 CALL CPU.TIME(TIME.END)

```

gol.out

```

Cells alive at generation      45 :      997185
Cells alive at generation      46 :      988790
Cells alive at generation      47 :      981380
Cells alive at generation      48 :      974253
Cells alive at generation      49 :      967943
Cells alive at generation      50 :      959634
Total time=      10.53407311439514

```

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 127,424
26: data region reached 2 times
26: data copyin transfers: 4
   device time(us): total=2,474 max=1,282 min=4 avg=618
59: data copyout transfers: 2
   device time(us): total=2,536 max=1,327 min=1,209 avg=1,268
30: compute region reached 50 times
30: kernel launched 50 times
   grid: [32] block: [128]
       device time(us): total=121,389 max=2,525 min=2,366 avg=2,427
       elapsed time(us): total=122,869 max=2,569 min=2,394 avg=2,457
30: reduction kernel launched 50 times
   grid: [1] block: [256]
       device time(us): total=186 max=5 min=3 avg=3
       elapsed time(us): total=1,110 max=80 min=18 avg=22
30: data region reached 100 times
30: data copyin transfers: 50
   device time(us): total=310 max=10 min=4 avg=6
56: data copyout transfers: 50
   device time(us): total=529 max=13 min=4 avg=10

```

- Single **parallel** and **data** regions and multiple loop directives;
- A **reduction** clause is used on the last parallel loop;
- GPWPVP level;
- Same cost of serial solution;
- Non-contiguous nested loops.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #6

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:, :) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old_world(0:rows+1,0:cols+1))
28 do g=1, generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang vector
32   do r=1, rows
33     !$ACC loop seq
34     do c=1, cols
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop
39   do c=1, cols
40     do r=1, rows
41       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
42         old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
43         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
44       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
45         world(r,c) = 0
46       else if (neigh == 3) then
47         world(r,c) = 1
48       endif
49     enddo
50   enddo
51   !$ACC loop reduction(+:cells)
52   do r=1, rows
53     do c=1, cols
54       cells = cells + world(r,c)
55     enddo
56   enddo
57   !$ACC end parallel
58   print *, "Cells=alive=at=generation=", g, ":", cells
59 enddo
60 !$ACC end data
61 deallocate(world, old_world)
62 CALL CPU_TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=		9.730012893676758

make.err

```

gol:
26, Generating create(old_world(0:rows+1,0:cols+1)) [if not already present]
Generating copy(world(0:rows+1,0:cols+1)) [if not already present]
30, Generating copy(cells) [if not already present]
Generating Tesla code
32, !$acc loop gang, vector(128) ! blockid%x threadid%x%x
34, !$acc loop seq
39, !$acc loop gang ! blockid%x threadid%x%x
40, !$acc loop seq
52, !$acc loop gang ! blockid%x threadid%x%x
Generating reduction(+:cells)
53, !$acc loop seq
40, Loop is parallelizable
53, Loop is parallelizable

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- First loop rewritten in contiguous fashion (gang-vector[i]/seq[j]);
- Cheaper than serial.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #6

```

22 CALL CPU.TIME(TIME.INI)
23 allocate (world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old_world(0:rows+1,0:cols+1))
28 do g=1, generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang vector
32   do r=1, rows
33     !$ACC loop seq
34     do c=1, cols
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop
39   do c=1, cols
40     do r=1, rows
41       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
42       old_world(r+1,c-1)+old_world(r-1,c)+&
43       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
44       if (old_world(r,c) == 1 .and. (neigh < 2 or neigh > 3) ) then
45         world(r,c) = 0
46       else if (neigh == 3) then
47         world(r,c) = 1
48       endif
49     enddo
50   enddo
51   !$ACC loop reduction(+: cells)
52   do r=1, rows
53     do c=1, cols
54       cells = cells + world(r,c)
55     enddo
56   enddo
57   !$ACC end parallel
58   print *, "Cells alive at generation-", g, ":", cells
59 enddo
60 !$ACC end data
61 deallocate(world, old_world)
62 CALL CPU.TIME(TIME.END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	9.730012893676758	

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 960,564
26: data region reached 2 times
26: data copyin transfers: 4
   device time(us): total=2,477 max=1,284 min=5 avg=619
60: data copyout transfers: 2
   device time(us): total=2,531 max=1,326 min=1,205 avg=1,265
30: compute region reached 50 times
30: kernel launched 50 times
   grid: [4000] block: [1]
   device time(us): total=954,268 max=19,975 min=19,022 avg=19,085
   elapsed time(us): total=955,917 max=20,007 min=19,053 avg=19,118
30: reduction kernel launched 50 times
   grid: [1] block: [256]
   device time(us): total=400 max=8 min=8 avg=8
   elapsed time(us): total=1,696 max=95 min=23 avg=33
30: data region reached 100 times
30: data copyin transfers: 50
   device time(us): total=311 max=8 min=6 avg=6
57: data copyout transfers: 50
   device time(us): total=577 max=19 min=10 avg=11

```

- Single parallel and data regions and multiple loop directives;
- GPWPVP level;
- First loop rewritten in contiguous fashion (gang-vector[i]/seq[j]);
- Cheaper than serial.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #7

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC& create (old_world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop
39   do c=1, cols
40     do r=1, rows
41       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
42         old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
43         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
44       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
45         world(r,c) = 0
46       else if (neigh == 3) then
47         world(r,c) = 1
48       endif
49     enddo
50   enddo
51   !$ACC loop reduction(+:cells)
52   do r=1, rows
53     do c=1, cols
54       cells = cells + world(r,c)
55     enddo
56   enddo
57   !$ACC end parallel
58   print *, "Cells=alive=at=generation=", g, ":", cells
59 enddo
60 !$ACC end data
61 deallocate(world, old_world)
62 CALL CPU_TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=		6.911005973815918

make.err

```

gol:
26, Generating create(old_world(0:rows+1,0:cols+1)) [if not already present]
Generating copy(world(0:rows+1,0:cols+1)) [if not already present]
30, Generating copy(cells) [if not already present]
Generating Tesla code
32, !$acc loop gang ! blockidx%x
34, !$acc loop vector(128) ! threadidx%x
39, !$acc loop gang ! blockidx%x threadidx%x
40, !$acc loop seq
52, !$acc loop gang ! blockidx%x threadidx%x
Generating reduction(+:cells)
53, !$acc loop seq
34, Loop is parallelizable
40, Loop is parallelizable
53, Loop is parallelizable

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- First loop rewritten in contiguous fashion (gang[j]/vector[i]).

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #7

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC& create (old_world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop
39   do c=1, cols
40     do r=1, rows
41       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
42       old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
43       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
44       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
45         world(r,c) = 0
46       else if (neigh == 3) then
47         world(r,c) = 1
48       endif
49     enddo
50   enddo
51   !$ACC loop reduction(+:cells)
52   do r=1, rows
53     do c=1, cols
54       cells = cells + world(r,c)
55     enddo
56   enddo
57   !$ACC end parallel
58   print *, "Cells_alive_at_generation=", g, "-", cells
59 enddo
60 !$ACC end data
61 deallocate(world, old_world)
62 CALL CPU_TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	6.911005973815918	

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 959,563
26: data region reached 2 times
26: data copyin transfers: 4
   device time(us): total=2,497 max=1,281 min=4 avg=624
60: data copyout transfers: 2
   device time(us): total=2,587 max=1,329 min=1,258 avg=1,293
30: compute region reached 50 times
30: kernel launched 50 times
   grid: [4000] block: [1]
   device time(us): total=953,343 max=19,882 min=18,935 avg=19,066
   elapsed time(us): total=954,949 max=19,917 min=18,963 avg=19,098
30: reduction kernel launched 50 times
   grid: [1] block: [256]
   device time(us): total=399 max=8 min=7 avg=7
   elapsed time(us): total=1,685 max=95 min=22 avg=33
30: data region reached 100 times
30: data copyin transfers: 50
   device time(us): total=321 max=12 min=3 avg=6
57: data copyout transfers: 50
   device time(us): total=416 max=14 min=4 avg=8

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- First loop rewritten in contiguous fashion (gang[j]/vector[i]).

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #8

```

22 CALL CPU.TIME(TIME.INI)
23 allocate (world(0:rows+1,0:cols+1), old.world(0:rows+1,0:cols+1))
24 old.world(:,:) = 0
25 call fill.world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old.world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old.world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop gang vector
39   do r=1, rows
40     !$ACC loop seq
41     do c=1, cols
42       neigh = old.world(r-1,c-1)+old.world(r,c-1)+&
43       old.world(r+1,c-1)+old.world(r-1,c)+old.world(r+1,c)+&
44       old.world(r-1,c+1)+old.world(r,c+1)+old.world(r+1,c+1)
45       if (old.world(r,c) == 1 .and. (neigh < 2or.neigh > 3) ) then
46         world(r,c) = 0
47       else if (neigh == 3) then
48         world(r,c) = 1
49       endif
50     enddo
51   enddo
52   !$ACC loop reduction(+:cells)
53   do r=1, rows
54     do c=1, cols
55       cells = cells + world(r,c)
56     enddo
57   enddo
58   !$ACC end parallel
59   print *, "Cells_alive_at_generation=", g, ":", cells
60 enddo
61 !$ACC end data
62 deallocate(world, old.world)
63 CALL CPU.TIME(TIME.END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	7.931918859481812	

make.err

```

gol:
26, Generating create(old.world(0:rows+1,0:cols+1)) [if not already present]
Generating copy(world(0:rows+1,0:cols+1)) [if not already present]
30, Generating copy(cells) [if not already present]
Generating Tesla code
32, !$acc loop gang ! blockidx%x
34, !$acc loop vector(128) ! threadidx%x
39, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
41, !$acc loop seq
53, !$acc loop gang ! blockidx%x threadidx%x
Generating reduction(+:cells)
54, !$acc loop seq
34, Loop is parallelizable
54, Loop is parallelizable

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- Second loop rewritten in contiguous fashion (gang-vector[i]/seq[j]);
- The modification increased the cost.



# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #8

```

22 CALL CPU_TIME(TIME,INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:, :) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC& create (old_world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop gang vector
39   do r=1, rows
40     !$ACC loop seq
41     do c=1, cols
42       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
43         old_world(r+1,c-1)+old_world(r-1,c)+&
44         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
45       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
46         world(r,c) = 0
47       else if (neigh == 3) then
48         world(r,c) = 1
49       endif
50     enddo
51   enddo
52   !$ACC loop reduction(+:cells)
53   do r=1, rows
54     do c=1, cols
55       cells = cells + world(r,c)
56     enddo
57   enddo
58   !$ACC end parallel
59   print *, "Cells_alive_at-generation=", g, ":", cells
60 enddo
61 !$ACC end data
62 deallocate(world, old_world)
63 CALL CPU_TIME(TIME,END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	7.931918859481812	

gol.err

Accelerator Kernel Timing data

```

gol NVIDIA devicenum=0
time(us): 176,785
26: data region reached 2 times
26: data copyin transfers: 4
   device time(us): total=2,467 max=1,277 min=4 avg=616
61: data copyout transfers: 2
   device time(us): total=2,550 max=1,327 min=1,223 avg=1,275
30: compute region reached 50 times
30: kernel launched 50 times
   grid: [4000] block: [1]
   device time(us): total=170,560 max=3,427 min=3,396 avg=3,411
   elapsed time(us): total=172,076 max=3,467 min=3,423 avg=3,441
30: reduction kernel launched 50 times
   grid: [1] block: [256]
   device time(us): total=402 max=12 min=7 avg=8
   elapsed time(us): total=1,269 max=34 min=21 avg=25
30: data region reached 100 times
30: data copyin transfers: 50
   device time(us): total=319 max=9 min=5 avg=6
58: data copyout transfers: 50
   device time(us): total=487 max=13 min=4 avg=9

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- Second loop rewritten in contiguous fashion (gang-vector[i]/seq[j]);
- The modification increased the cost.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #9

```

22 CALL CPU.TIME(TIME.INI)
23 allocate(world(0:rows+1,0:cols+1), old.world(0:rows+1,0:cols+1))
24 old.world(:,:) = 0
25 call fill.world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create( old.world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old.world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop gang
39   do c=1, cols
40     !$ACC loop vector
41     do r=1, rows
42       neigh = old.world(r-1,c-1)+old.world(r,c-1)+&
43       old.world(r+1,c-1)+old.world(r-1,c)+old.world(r+1,c)+&
44       old.world(r-1,c+1)+old.world(r,c+1)+old.world(r+1,c+1)
45       if (old.world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
46         world(r,c) = 0
47       else if (neigh == 3) then
48         world(r,c) = 1
49       endif
50     enddo
51   enddo
52   !$ACC loop reduction(+:cells)
53   do r=1, rows
54     do c=1, cols
55       cells = cells + world(r,c)
56     enddo
57   enddo
58   !$ACC end parallel
59   print *, "Cells_alive_at_generation=", g, ":", cells
60 enddo
61 !$ACC end data
62 deallocate(world, old.world)
63 CALL CPU.TIME(TIME.END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=		6.136991024017334

make.err

```

gol:
26, Generating create(old.world(0:rows+1,0:cols+1)) [if not already present]
Generating copy(world(0:rows+1,0:cols+1)) [if not already present]
30, Generating copy(cells) [if not already present]
Generating Tesla code
32, !$acc loop gang ! blockidx%x
34, !$acc loop vector(128) ! threadidx%x
39, !$acc loop gang ! blockidx%x
41, !$acc loop vector(128) ! threadidx%x
53, !$acc loop gang ! blockidx%x threadidx%x
Generating reduction(+:cells)
54, !$acc loop seq
34, Loop is parallelizable
41, Loop is parallelizable
54, Loop is parallelizable

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- Second loop rewritten in contiguous fashion (gang[j]-vector[i]).

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #9

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old_world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop gang
39   do c=1, cols
40     !$ACC loop vector
41     do r=1, rows
42       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
43       old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
44       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
45       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
46         world(r,c) = 0
47       else if (neigh == 3) then
48         world(r,c) = 1
49       endif
50     enddo
51   enddo
52   !$ACC loop reduction(+:cells)
53   do r=1, rows
54     do c=1, cols
55       cells = cells + world(r,c)
56     enddo
57   enddo
58   !$ACC end parallel
59   print *, "Cells=alive=at=generation=", g, ":", cells
60 enddo
61 !$ACC end data
62 deallocate(world, old_world)
63 CALL CPU_TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=		6.136991024017334

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 182,817
26: data region reached 2 times
   26: data copyin transfers: 4
      device time(us): total=2,472 max=1,281 min=4 avg=618
61: data copyout transfers: 2
   61: device time(us): total=2,581 max=1,374 min=1,207 avg=1,290
30: compute region reached 50 times
   30: kernel launched 50 times
      grid: [4000] block: [1]
         device time(us): total=176,547 max=6,297 min=3,464 avg=3,530
         elapsed time(us): total=178,119 max=6,336 min=3,492 avg=3,562
30: reduction kernel launched 50 times
      grid: [1] block: [256]
         device time(us): total=400 max=8 min=8 avg=8
         elapsed time(us): total=1,286 max=37 min=22 avg=25
30: data region reached 100 times
   30: data copyin transfers: 50
      device time(us): total=307 max=9 min=3 avg=6
58: data copyout transfers: 50
   58: device time(us): total=510 max=13 min=4 avg=10

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- Second loop rewritten in contiguous fashion (gang[j]-vector[i]).

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #10

```

22 CALL CPU_TIME(TIME.INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:, :) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old_world(0:rows+1,0:cols+1))
28 do g=1, generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop gang
39   do c=1, cols
40     !$ACC loop vector
41     do r=1, rows
42       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
43         old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
44         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
45       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
46         world(r,c) = 0
47       else if (neigh == 3) then
48         world(r,c) = 1
49       endif
50     enddo
51   enddo
52   !$ACC loop gang vector
53   do r=1, rows
54     !$ACC loop seq reduction(+:cells)
55     do c=1, cols
56       cells = cells + world(r,c)
57     enddo
58   enddo
59   !$ACC end parallel
60   print *, "Cells=alive=at=generation=", g, ":", cells
61 enddo
62 !$ACC end data
63 deallocate(world, old_world)
64 CALL CPU_TIME(TIME.END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	6.449497938156128	

make.err

```

gol:
26, Generating create(old_world(0:rows+1,0:cols+1)) [if not already present]
Generating copy(world(0:rows+1,0:cols+1)) [if not already present]
30, Generating copy(cells) [if not already present]
Generating Tesla code
32, !$acc loop gang ! blockidx%x
34, !$acc loop vector(128) ! threadidx%x
39, !$acc loop gang ! blockidx%x
41, !$acc loop vector(128) ! threadidx%x
53, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
55, !$acc loop seq
34, Loop is parallelizable
41, Loop is parallelizable

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- Third loop rewritten in contiguous fashion (gang-vector[i]/seq[j]);
- The modification increased the cost.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #10

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:, :) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old_world(0:rows+1,0:cols+1))
28 do g=1, generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop gang
39   do c=1, cols
40     !$ACC loop vector
41     do r=1, rows
42       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
43         old_world(r+1,c-1)+old_world(r-1,c)+&
44         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
45       if (old_world(r,c) == 1 .and. (neigh < 2 or neigh > 3)) then
46         world(r,c) = 0
47       else if (neigh == 3) then
48         world(r,c) = 1
49       endif
50     enddo
51   enddo
52   !$ACC loop gang vector
53   do r=1, rows
54     !$ACC loop seq reduction(+:cells)
55     do c=1, cols
56       cells = cells + world(r,c)
57     enddo
58   enddo
59   !$ACC end parallel
60   print *, "Cells=alive=at=generation=", g, ":", cells
61 enddo
62 !$ACC end data
63 deallocate(world, old_world)
64 CALL CPU_TIME(TIME_END)

```

gol.out

```

Cells alive at generation      45 :      997185
Cells alive at generation      46 :      988790
Cells alive at generation      47 :      981380
Cells alive at generation      48 :      974253
Cells alive at generation      49 :      967943
Cells alive at generation      50 :      959634
Total time=      6.449497938156128

```

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 183,847
26: data region reached 2 times
26: data copyin transfers: 4
   device time(us): total=2,475 max=1,280 min=5 avg=618
62: data copyout transfers: 2
   device time(us): total=2,525 max=1,322 min=1,203 avg=1,262
30: compute region reached 50 times
30: kernel launched 50 times
   grid: [32] block: [128]
   device time(us): total=178,057 max=3,644 min=3,493 avg=3,561
   elapsed time(us): total=179,522 max=3,672 min=3,521 avg=3,590
30: data region reached 100 times
30: data copyin transfers: 50
   device time(us): total=315 max=9 min=5 avg=6
59: data copyout transfers: 50

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- Third loop rewritten in contiguous fashion (gang-vector[i]/seq[j]);
- The modification increased the cost.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #11

```

22 CALL CPU_TIME(TIME_INIT)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:, :) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old_world(0:rows+1,0:cols+1))
28 do g=1, generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop gang
39   do c=1, cols
40     !$ACC loop vector
41     do r=1, rows
42       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
43         old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
44         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
45       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
46         world(r,c) = 0
47       else if (neigh == 3) then
48         world(r,c) = 1
49       endif
50     enddo
51   enddo
52   !$ACC loop gang
53   do c=1, cols
54     !$ACC loop vector reduction(+:cells)
55     do r=1, rows
56       cells = cells + world(r,c)
57     enddo
58   enddo
59   !$ACC end parallel
60   print *, "Cells=alive=at=generation=", g, ":", cells
61 enddo
62 !$ACC end data
63 deallocate(world, old_world)
64 CALL CPU_TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	4.712203025817871	

make.err

```

gol:
26, Generating create(old_world(0:rows+1,0:cols+1)) [if not already present]
   Generating copy(world(0:rows+1,0:cols+1)) [if not already present]
30, Generating copy(cells) [if not already present]
   Generating Tesla code
32, !$acc loop gang ! blockidx%x
34, !$acc loop vector(128) ! threadidx%x
39, !$acc loop gang ! blockidx%x
41, !$acc loop vector(128) ! threadidx%x
53, !$acc loop gang ! blockidx%x
55, !$acc loop vector(128) ! threadidx%x
   Generating reduction(+:cells)
34, Loop is parallelizable
41, Loop is parallelizable
55, Loop is parallelizable

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- Third loop rewritten in contiguous fashion (gang[j]/vector[i]).

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #11

```

22 CALL CPU.TIME(TIME.INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:, :) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC& create (old_world(0:rows+1,0:cols+1))
28 do g=1, generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop gang
39   do c=1, cols
40     !$ACC loop vector
41     do r=1, rows
42       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
43         old_world(r+1,c-1)+old_world(r-1,c)+&
44         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
45       if (old_world(r,c) == 1 .and. (neigh < 2*nr.neigh > 3)) then
46         world(r,c) = 0
47       else if (neigh == 3) then
48         world(r,c) = 1
49       endif
50     enddo
51   enddo
52   !$ACC loop gang
53   do c=1, cols
54     !$ACC loop vector reduction(+:cells)
55     do r=1, rows
56       cells = cells + world(r,c)
57     enddo
58   enddo
59   !$ACC end parallel
60   print *, "Cells=alive=at=generation=", g, ":", cells
61 enddo
62 !$ACC end data
63 deallocate(world, old_world)
64 CALL CPU.TIME(TIME.END)

```

gol.out

```

Cells alive at generation      45 :      997185
Cells alive at generation      46 :      988790
Cells alive at generation      47 :      981380
Cells alive at generation      48 :      974253
Cells alive at generation      49 :      967943
Cells alive at generation      50 :      959634
Total time=      4.712203025817871

```

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 121,749
26: data region reached 2 times
26: data copyin transfers: 4
   device time(us): total=2,471 max=1,278 min=4 avg=617
62: data copyout transfers: 2
   device time(us): total=2,525 max=1,321 min=1,204 avg=1,262
30: compute region reached 50 times
30: kernel launched 50 times
   grid: [2000] block: [128]
   device time(us): total=115,851 max=2,398 min=2,261 avg=2,317
   elapsed time(us): total=117,247 max=2,439 min=2,289 avg=2,344
30: data region reached 100 times
30: data copyin transfers: 50
   device time(us): total=351 max=26 min=5 avg=7
59: data copyout transfers: 50
   device time(us): total=551 max=19 min=4 avg=11

```

- Single **parallel** and **data** regions and multiple **loop** directives;
- GPWPVP level;
- Third loop rewritten in contiguous fashion (gang[j]/vector[i]).

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #11

```

22 CALL CPU.TIME(TIME.INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:, :) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC& create (old_world(0:rows+1,0:cols+1))
28 do g=1, generations
29   cells = 0
30   !$ACC parallel copy(cells)
31   !$ACC loop gang
32   do c=1, cols
33     !$ACC loop vector
34     do r=1, rows
35       old_world(r,c) = world(r,c)
36     enddo
37   enddo
38   !$ACC loop gang
39   do c=1, cols
40     !$ACC loop vector
41     do r=1, rows
42       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
43         old_world(r+1,c-1)+old_world(r-1,c)+old_world(r+1,c)+&
44         old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
45       if (old_world(r,c) == 1 .and. (neigh < 2*nr.neigh > 3)) then
46         world(r,c) = 0
47       else if (neigh == 3) then
48         world(r,c) = 1
49       endif
50     enddo
51   enddo
52   !$ACC loop gang
53   do c=1, cols
54     !$ACC loop vector reduction(+:cells)
55     do r=1, rows
56       cells = cells + world(r,c)
57     enddo
58   enddo
59   !$ACC end parallel
60   print *, "Cells=alive=at=generation=", g, ":", cells
61 enddo
62 !$ACC end data
63 deallocate(world, old_world)
64 CALL CPU.TIME(TIME.END)

```

gol.out

```

Cells alive at generation      45 :      997185
Cells alive at generation      46 :      988790
Cells alive at generation      47 :      981380
Cells alive at generation      48 :      974253
Cells alive at generation      49 :      967943
Cells alive at generation      50 :      959634
Total time=      4.712203025817871

```

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 121,749
26: data region reached 2 times
26: data copyin transfers: 4
   device time(us): total=2,471 max=1,278 min=4 avg=617
62: data copyout transfers: 2
   device time(us): total=2,525 max=1,321 min=1,204 avg=1,262
30: compute region reached 50 times
30: kernel launched 50 times
   grid: [2000] block: [128]
   device time(us): total=115,851 max=2,398 min=2,261 avg=2,317
   elapsed time(us): total=117,247 max=2,439 min=2,289 avg=2,344
30: data region reached 100 times
30: data copyin transfers: 50
   device time(us): total=351 max=26 min=5 avg=7
59: data copyout transfers: 50
   device time(us): total=551 max=19 min=4 avg=11

```

### Final Ratio

$$\begin{aligned}
 & \frac{T_{\text{serial}}[\text{sec}]}{T_{\text{GTX}} - 1050[\text{sec}]} \times \frac{P_{\text{serial}}[\text{€}]}{P_{\text{GTX}} - 1050[\text{€}]} \\
 & \frac{10.29[\text{sec}]}{4.71[\text{sec}]} \times \frac{380.9[\text{€}]}{161.98[\text{€}]} \\
 & \approx 5.14
 \end{aligned}$$



# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #11 - Kernels with contiguous data

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:, :) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old_world(0:rows+1,0:cols+1))
28 do g=1, generations
29   cells = 0
30   !$ACC kernels copy(cells)
31   !$ACC loop
32   do c=1, cols
33     do r=1, rows
34       old_world(r,c) = world(r,c)
35     enddo
36   enddo
37   !$ACC loop
38   do c=1, cols
39     do r=1, rows
40       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
41       old_world(r+1,c-1)+old_world(r-1,c)+&
42       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
43       if (old_world(r,c) == 1 .and. (neigh < 2or.neigh > 3)) then
44         world(r,c) = 0
45       else if (neigh == 3) then
46         world(r,c) = 1
47       endif
48     enddo
49   enddo
50   !$ACC loop
51   do c=1, cols
52     do r=1, rows
53       cells = cells + world(r,c)
54     enddo
55   enddo
56   !$ACC end kernels
57   print *, "Cells alive at generation=", g, ":", cells
58 enddo
59 !$ACC end data
60 deallocate(world, old_world)
61 CALL CPU_TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=	4.718765020370483	

make.err

```

gol:
26, Generating create(old_world(0:rows+1,0:cols+1)) [if not already present]
    Generating copy(world(0:rows+1,0:cols+1)) [if not already present]
30, Generating copy(cells) [if not already present]
32, Loop is parallelizable
33, Loop is parallelizable
    Generating Tesla code
32, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
33, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
38, Loop is parallelizable
39, Loop is parallelizable
    Generating Tesla code
38, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
39, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
51, Loop is parallelizable
52, Loop is parallelizable
    Generating Tesla code
51, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
52, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
53, Generating implicit reduction(+:cells)

```

- Parallel directives are replaced by kernels ones;
- The compiler manages the parallelism;
- Memory access and data contiguity is up to the developer;
- The new version presents the same performance with less effort.

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #11 - Kernels with contiguous data

```

22 CALL CPU.TIME(TIME.INI)
23 allocate(world(0:rows+1,0:cols+1), old.world(0:rows+1,0:cols+1))
24 old.world(:, :) = 0
25 call fill.world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC& create (old.world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC kernels copy(cells)
31   !$ACC loop
32   do c=1, cols
33     do r=1, rows
34       old.world(r,c) = world(r,c)
35     enddo
36   enddo
37   !$ACC loop
38   do c=1, cols
39     do r=1, rows
40       neigh = old.world(r-1,c-1)+old.world(r,c-1)+&
41       old.world(r+1,c-1)+old.world(r,c-1)+old.world(r+1,c)+&
42       old.world(r-1,c+1)+old.world(r,c+1)+old.world(r+1,c+1)
43       if (old.world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
44         world(r,c) = 0
45       else if (neigh == 3) then
46         world(r,c) = 1
47       endif
48     enddo
49   enddo
50   !$ACC loop
51   do c=1, cols
52     do r=1, rows
53       cells = cells + world(r,c)
54     enddo
55   enddo
56   !$ACC end kernels
57   print *, "Cells alive at generation-", g, ":", cells
58 enddo
59 !$ACC end data
60 deallocate(world, old.world)
61 CALL CPU.TIME(TIME.END)

```

gol.out

```

Cells alive at generation      45 :      997185
Cells alive at generation      46 :      988790
Cells alive at generation      47 :      981380
Cells alive at generation      48 :      974253
Cells alive at generation      49 :      967943
Cells alive at generation      50 :      959634
Total time=      4.718765020370483

```

gol.err

```

Accelerator Kernel Timing data
gol NVIDIA devicenum=0
time(us): 166,310
26: data region reached 2 times
26: data copyin transfers: 4
   device time(us): total=2,483 max=1,285 min=4 avg=620
59: data copyout transfers: 2
   device time(us): total=2,542 max=1,331 min=1,211 avg=1,271
30: compute region reached 50 times
33: kernel launched 50 times
   grid: [125x500] block: [32x4]
   device time(us): total=39,926 max=816 min=794 avg=798
   elapsed time(us): total=41,321 max=844 min=821 avg=826
39: kernel launched 50 times
   grid: [125x500] block: [32x4]
   device time(us): total=63,119 max=1,265 min=1,261 avg=1,262
   elapsed time(us): total=63,934 max=1,283 min=1,275 avg=1,278
52: kernel launched 50 times
   grid: [125x500] block: [32x4]
   device time(us): total=53,498 max=1,073 min=1,068 avg=1,069
   elapsed time(us): total=54,268 max=1,090 min=1,082 avg=1,085
52: reduction kernel launched 50 times
   grid: [1] block: [256]
   device time(us): total=3,872 max=78 min=77 avg=77
   elapsed time(us): total=4,783 max=162 min=91 avg=95
30: data region reached 100 times
30: data copyin transfers: 50
   device time(us): total=301 max=7 min=5 avg=6
56: data copyout transfers: 50
   device time(us): total=569 max=37 min=4 avg=11

```

# Optimization Example - Game of Life (4000x2000 - 100 gen.)

## Step #11 - Multicore (fair comparison) - kernel2multi-core

```

22 CALL CPU_TIME(TIME_INI)
23 allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
24 old_world(:,:) = 0
25 call fill_world
26 !$ACC data copy( world(0:rows+1,0:cols+1) ) &
27 !$ACC create (old_world(0:rows+1,0:cols+1))
28 do g=1,generations
29   cells = 0
30   !$ACC kernels copy(cells)
31   !$ACC loop
32   do c=1, cols
33     do r=1, rows
34       old_world(r,c) = world(r,c)
35     enddo
36   enddo
37   !$ACC loop
38   do c=1, cols
39     do r=1, rows
40       neigh = old_world(r-1,c-1)+old_world(r,c-1)+&
41       old_world(r+1,c-1)+old_world(r-1,c)+&
42       old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
43       if (old_world(r,c) == 1 .and. (neigh<2or.neigh>3) ) then
44         world(r,c) = 0
45       else if (neigh == 3) then
46         world(r,c) = 1
47       endif
48     enddo
49   enddo
50   !$ACC loop
51   do c=1, cols
52     do r=1, rows
53       cells = cells + world(r,c)
54     enddo
55   enddo
56   !$ACC end kernels
57   print *, "Cells alive at generation_", g, ":", cells
58 enddo
59 !$ACC end data
60 deallocate(world, old_world)
61 CALL CPU_TIME(TIME_END)

```

gol.out

Cells alive at generation	45 :	997185
Cells alive at generation	46 :	988790
Cells alive at generation	47 :	981380
Cells alive at generation	48 :	974253
Cells alive at generation	49 :	967943
Cells alive at generation	50 :	959634
Total time=		1.720873832702637

make.err

```

pgf90 -O0 -acc=noautopar -ta=multicore -Minfo=all -c gol.f90 -o gol.o
pgf90 -O0 -acc=noautopar -ta=multicore -Minfo=all gol.o -o gol
gol:
32, Loop is parallelizable
Generating Multicore code
32, 1$acc loop gang
33, Loop is parallelizable
38, Loop is parallelizable
Generating Multicore code
38, 1$acc loop gang
39, Loop is parallelizable
51, Loop is parallelizable
Generating Multicore code
51, 1$acc loop gang
52, Loop is parallelizable
53, Generating implicit reduction(+:cells)

```

### Final Fair Ratio

$$\frac{T_{multi-core}[sec]}{T_{GTX-1050}[sec]} \times \frac{P_{multi-core}[€]}{P_{GTX-1050}[€]}$$

$$\frac{1.72[sec]}{4.71[sec]} \times \frac{380.9[€]}{161.98[€]}$$

$$\approx 0.86$$