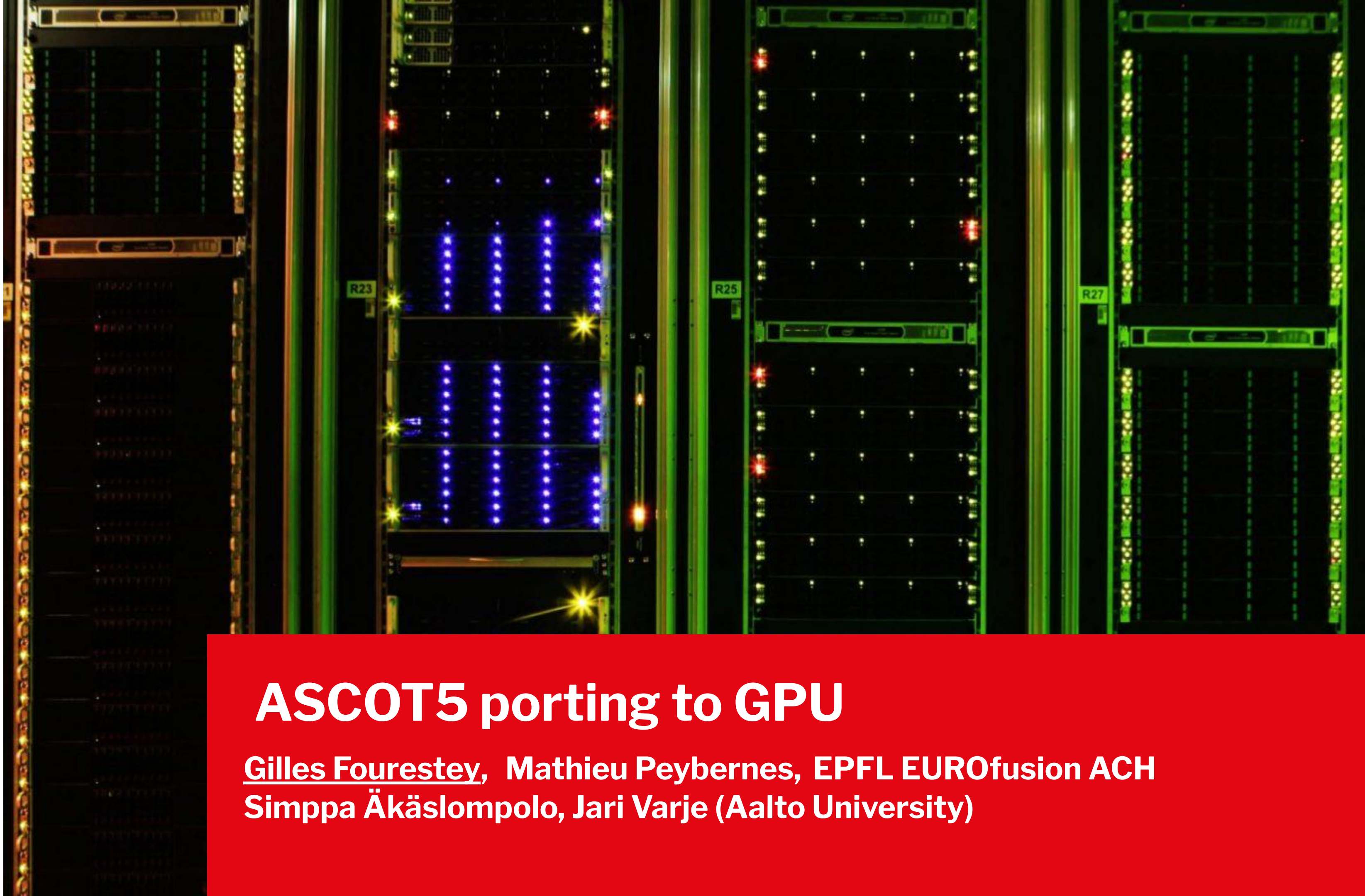


ASCOT5 porting to GPU

Gilles Fourestey, Mathieu Peybernes, EPFL EUROfusion ACH
Simppa Äkäslompolo, Jari Varje (Aalto University)

ASCOT5

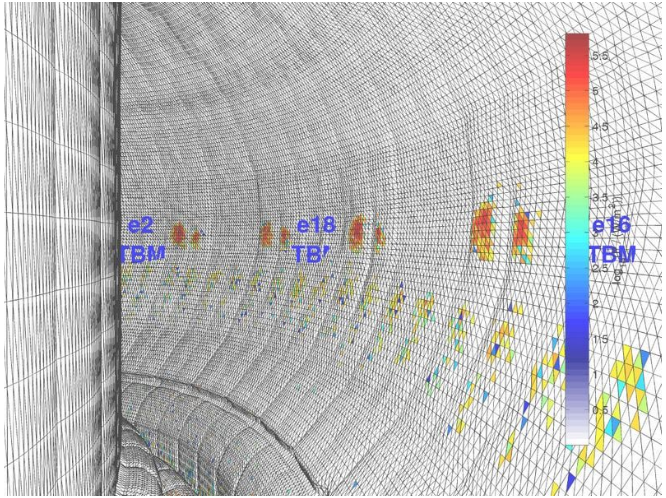
- ASCOT5 is a test **particle orbit-following** code for toroidal magnetically confined fusion devices
- The code uses the **Monte Carlo method** to solve the distribution of particles by following their trajectories.
  - The **evolution of the distribution function** for a test particle species *a* is described by the **Fokker-Planck equation**

$$\frac{\partial f_a}{\partial t} + \mathbf{v} \cdot \nabla f_a + \frac{q_a}{m_a}(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \nabla_{\mathbf{v}} f_a = \sum_b -\nabla_{\mathbf{v}} \cdot [\mathbf{a}_{ab} f_a - \nabla_{\mathbf{v}} \cdot (\mathbf{D}_{ab} f_a)]$$

and **approximated by the Langevin equation** for a large number of markers that represent the distributed function:

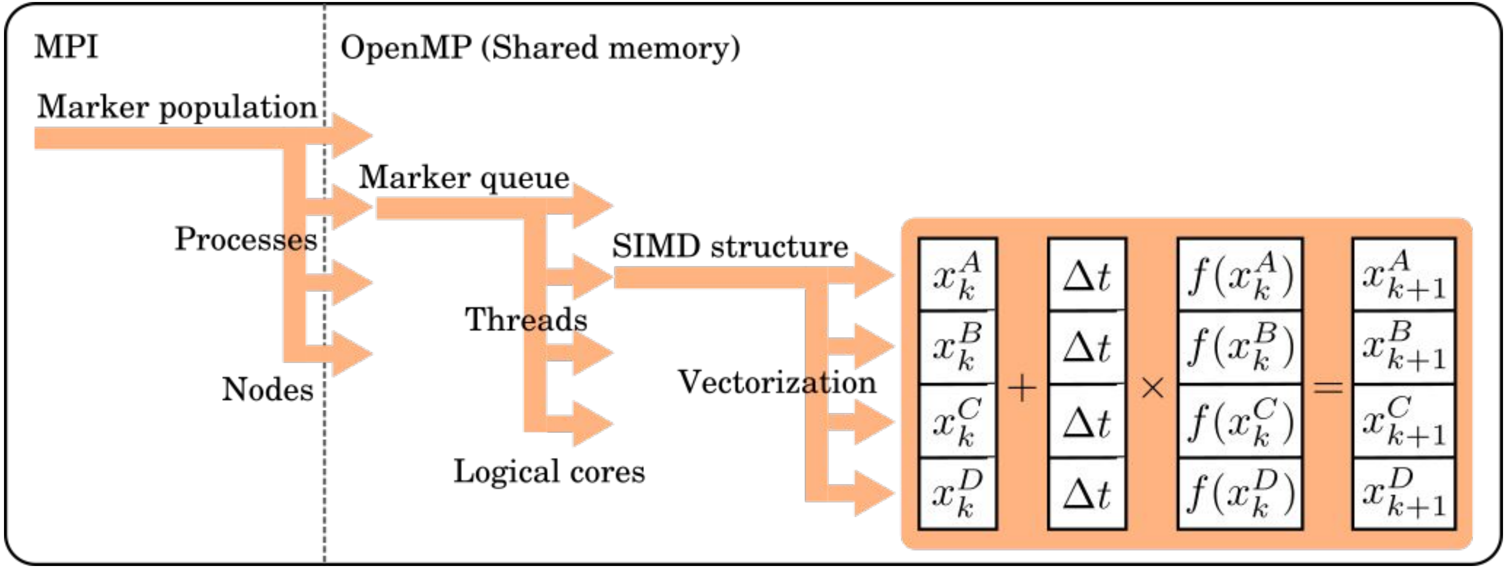
$$d\mathbf{z} = [\dot{\mathbf{z}} + \mathbf{a}(\mathbf{z}, t)]\, dt + \boldsymbol{\sigma}(\mathbf{z}, t) \cdot d\mathcal{W}$$

- The particles undergo **collisions with a static Maxwellian background plasma**
- The detailed magnetic fields and the first wall can be **fully three-dimensional**

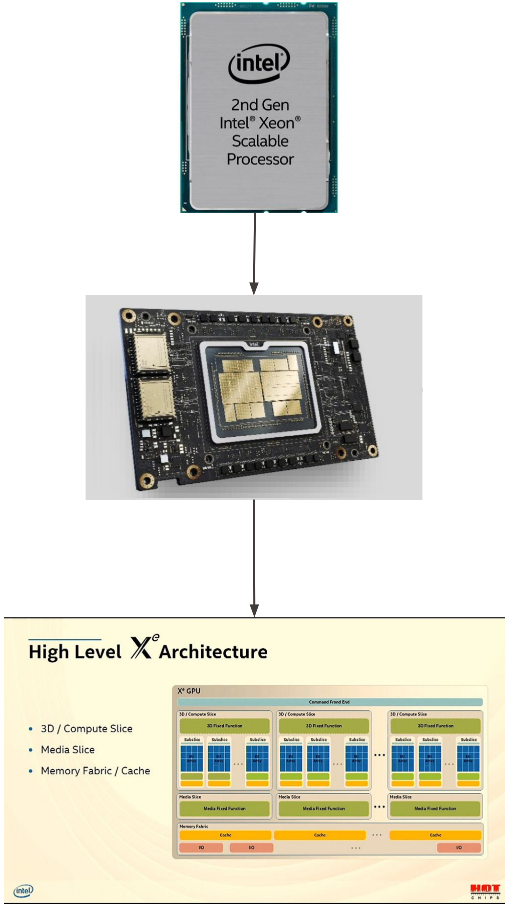


ASCOT5 CPU version

- The time evolutions of each particle are **independent** from each other
- **One + two levels of parallelism**:
  - **MPI**: Particles distributed among tasks, fields replicated
  - **OpenMP:** queue based approach
  - highly vectorized using the **SIMD**, originally developed for KNL manycore systems as target
- **Very good performance on CPU**

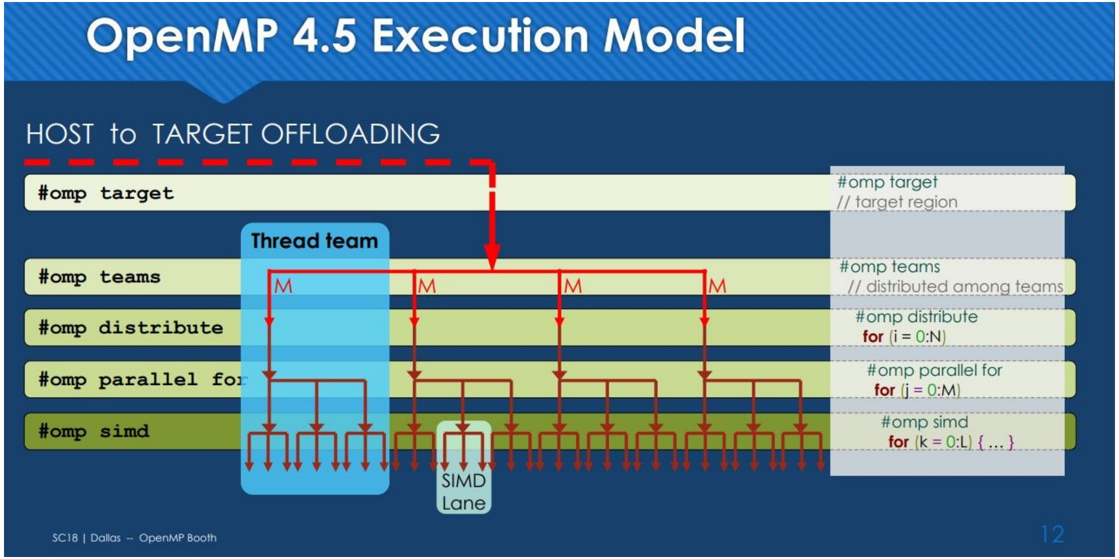

MPI
OpenMP (Shared memory)
Marker population
Marker queue
Processes
SIMD structure
Threads
Nodes
Vectorization
Logical cores
x_k^A x_k^B x_k^C x_k^D
+
Δt Δt Δt Δt
×
$f(x_k^A)$ $f(x_k^B)$ $f(x_k^C)$ $f(x_k^D)$
=
x_{k+1}^A x_{k+1}^B x_{k+1}^C x_{k+1}^D

Moving to GPUs first attempt: OMP-Offload hardware mapping



High Level Xe Architecture

- 3D / Compute Slice
- Media Slice
- Memory Fabric / Cache



OpenMP 4.5 Execution Model

HOST to TARGET OFFLOADING

```
#omp target
#omp teams
#omp distribute
#omp parallel for
#omp simd
```

```
#omp target
// target region
#omp teams
// distributed among teams
#omp distribute
for (i = 0:N)
#omp parallel for
for (j = 0:M)
#omp simd
for (k = 0:L) { ... }
```

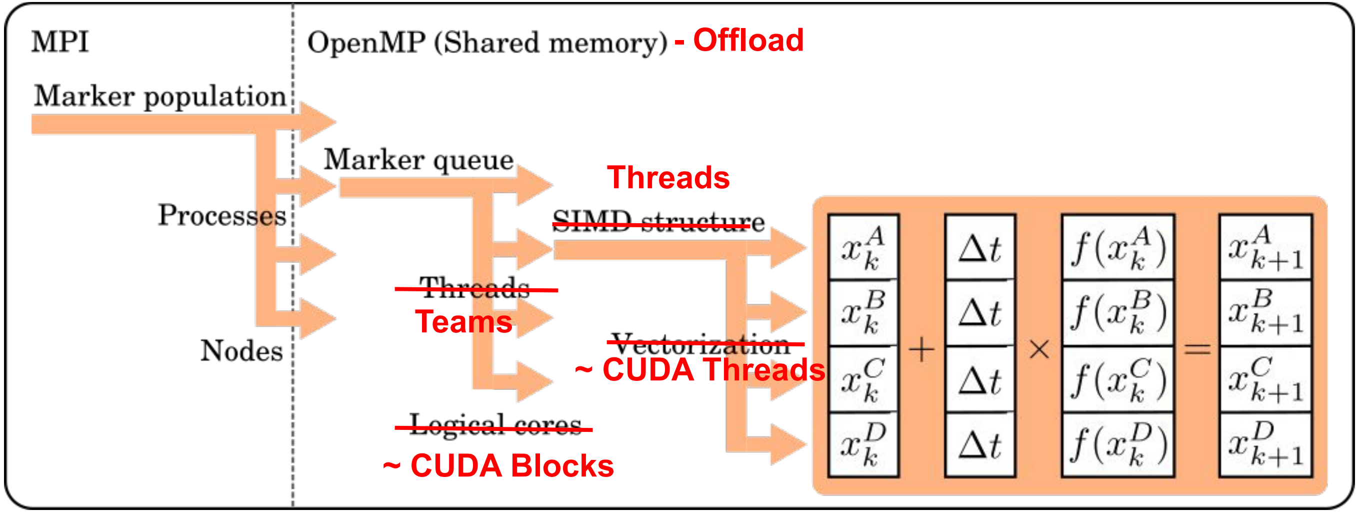
SC18 | Dallas -- OpenMP Booth

12

#pragma omp teams	Master thread of each team will execute the team region
#pragma omp distribute	split the iteration space among all the league of thread teams
#pragma omp parallel for	split the iteration space between all the threads within a team
#pragma omp simd	split an iteration space into SIMD lanes

Moving to GPUs: OpenMP-Offload hardware mapping

- **MPI+GPU** levels of parallelism:
  - Message Passing: particles distributed among MPI tasks, fields replicated: One GPU per MPI process
  - GPU OpenMP-offload based – **2 levels of parallelism** map to:
    - Marker queues distributed over **OpenMP teams**
    - Each marker is distributed over **OpenMP team threads**



ASCOT5 – GPU version

- **MPI+GPU** levels of parallelism:
  - Message Passing: particles distributed among MPI tasks, fields replicated:

    One GPU per MPI process
  - GPU OpenMP-offload based – **2 levels of parallelism** map to:
    - Marker queues distributed over **OpenMP teams**
    - Each marker is distributed over **OpenMP team threads**

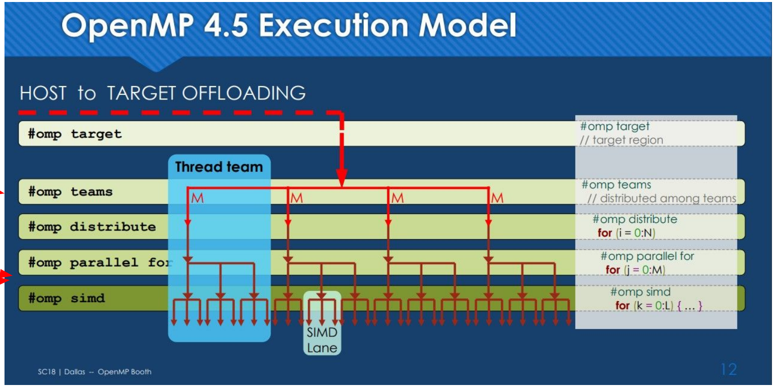
L1

L2

L2

GPU

```
#pragma omp target teams distribute
for(int iprt = 0; iprt < NbParticules ; iprt += NSIMD) {
          …some work…
          particle_simd_fo p; //new set of NSIMD particles
          #pragma omp parallel for simd
          for(int i=0; i< NSIMD; i++) {
                    p.running[i] = 0;
                    …some work…
          #pragma omp parallel for simd
          for(int i=0; i< NSIMD; i++) {
                    …some work…
```



OpenMP 4.5 Execution Model

HOST to TARGET OFFLOADING

SC18 | Dallas -- OpenMP Booth

12

ASCOT5 Benchmarks - Mapping

May2022 Benchmark, comparison with different compilers/platforms

- **gcc11** on **x86 + v100 (Phoenix@EPFL)**
- **XL compilers** + v100 (m100@Cineca)
- intel compilers on skylake and icelake (Jed@EPFL, ASCOT5 cpu-only)
- **gcc11 with OpenACC** on **x86 + v100 (Phoenix@EPFL)**

ASCOT5	TTS [s]	may2022_2dwall_go_analyticB			
	markers:	10000	100000		
				Platform	Compiler
m100@CINECA	OMP Offload	**46**	**473**	Power9 + v100	XL compilers
Phoenix@EPFL	**OMP Offload**	**232**	**2143**	**6138 gold + v100**	**gcc 11**
Helvetios@EPFL	**OMP (cpu-only)**	87	860	2x Gold 6140	intel compilers
Jed@EPFL	**OMP (cpu-only)**	31	318	2x Platinum 8360Y	intel compilers

- OMP Offload with GCC is very slow on x86
- OMP Offload is barely on-par with CPU-only code on P9

?

ASCOT5 Benchmarks

Reason? Probably multi-factor.

Reason #1: It seems gcc cannot take advantage of the last level of parallelism although it spawns a threadblock to do so (that's the standard).

From nsight:
gcc: CUDA kernel launched: dim={#teams,1,1}, blocks=**{#threads, 32, 1}**
xl : CUDA kernel launched: dim={#teams,1,1}, blocks=**{#threads*32, 1, 1}**

number of teams = 160

GCC 11	L1 = omp distribute, No L2			L1 = omp distribute, L2 = omp simd		
particles	**10^3**	**10^4**	**10^5**	**10^3**	**10^4**	**10^5**
NSIMD = 1	107	565	4314	110	577	4429
NSIMD = 2	110	558	4726	109	558	4750
NSIMD = 4	113	620	4651	114	623	4676
NSIMD = 8	108	608	4576	110	621	4665
NSIMD = 16	181	617	4653	176	601	4549
NSIMD = 32	307	616	4965	298	599	4844
NSIMD = 64	534	610	4902	520	596	4807

Consequence: performance is divided by 32

Reason #2: the CPU approach generates one huge kernel to big for GPUs: looking at the nvptx code, it seems each kernel uses ~1500 registers per thread:

- Reduced number of total threads, therefore reduced number of in-flight warps
- Register spilling to local/main memory

Consequence: Nsight gives 3% occupancy

ASCOT5-GPU with OpenACC/OpenMP offload interop

Solution #1: moving to OpenACC

- OpenACC is more mature than OpenMP offload
- gcc supports it along OpenMP offload (-fopenacc or -fopenmp)
- OpenACC and OpenMP offload are very similar

```
OMP_L1
for(int iprt = 0; iprt < NbParticules ; iprt += NSIMD) {
        …some work…
        particle_simd_fo p; //new set of NSIMD particles
        OMP_L2
        for(int i=0; i< NSIMD; i++) {
                p.running[i] = 0;
                …some work…
        OMP_L2
        for(int i=0; i< NSIMD; i++) {
                …some work…
```

GPU

```
#pragma once
#define STRINGIFY(X) #X
#define MY_PRAGMA(X) _Pragma(STRINGIFY(X))
#if !defined(_OPENMP) && !defined(_OPENACC)
#warning "No Openmp or OpenACC"
#define OMP_L1
#define OMP_L2
#define DECLARE_TARGET
#define DECLARE_TARGET_END
#endif
#ifdef _OPENMP
#warning "OpenMP"
#define OMP_L1 MY_PRAGMA(omp distribute parallel for)
#define OMP_L2 MY_PRAGMA(omp simd)
#define DECLARE_TARGET     MY_PRAGMA(omp declare target)
#define DECLARE_TARGET_END MY_PRAGMA(omp end declare target
#endif
#ifdef _OPENACC
#warning "OpenACC"
#define OMP_L1 MY_PRAGMA(acc loop gang worker)
#define OMP_L2 MY_PRAGMA(acc vector)
#define DECLARE_TARGET     MY_PRAGMA(acc routine vector)
#define DECLARE_TARGET_END MY_PRAGMA(warning "ACC")
#endif
```

ASCOT5 Benchmarks w/ OpenACC

May2022 Benchmark, comparison with different compilers/platforms

- **gcc11** on **x86 + v100 (Phoenix@EPFL)**
- **XL compilers** + v100 (m100@Cineca)
- intel compilers on skylake and icelake (Jed@EPFL, ASCOT5 cpu-only)
- **gcc11 with OpenACC** on **x86 + v100 (Phoenix@EPFL)**

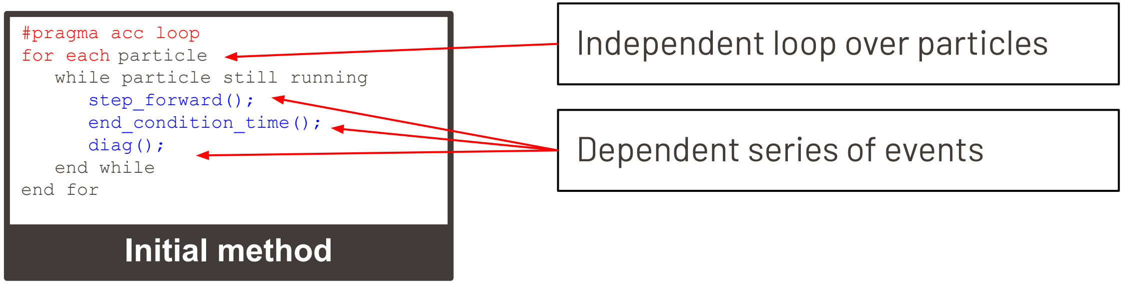
ASCOT5	TTS [s]	may2022_2dwall_go_analyticB			
	markers:	10000	100000		
				Platform	Compiler
m100@CINECA	OMP Offload	**46**	**473**	Power9 + v100	XL compilers
Phoenix@EPFL	**OMP Offload**	**232**	**2143**	**6138 gold + v100**	**gcc 11**
NEW Phoenix@EPFL	**OpenACC**	**48**	**261**	6138 gold + v100	gcc 11
Helvetios@EPFL	**OMP (cpu-only)**	87	860	2x Gold 6140	intel compilers
Jed@EPFL	**OMP (cpu-only)**	31	318	2x Platinum 8360Y	intel compilers

- OpenACC gives CPU-like performance

New algorithmic approach

- **Solution #2:** **Implement a new version by splitting the initial whole kernel**

Current situation:



```
#pragma acc loop
for each particle
   while particle still running
      step_forward();
      end_condition_time();
      diag();
   end while
end for
```

Initial method

Independent loop over particles

Dependent series of events

This leads to a huge kernel that works well for CPUs , but unfit for GPUs:

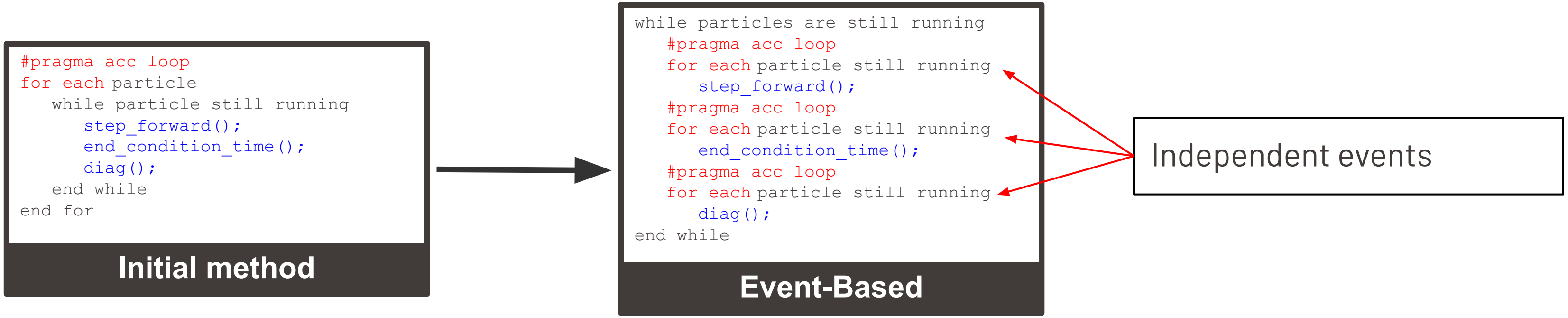
- Not enough registers
- Thread divergence
- Un-coalesced memory accesses

Solution: algorithmic modification with **event-based approach**

(https://www.openmp.org/wp-content/uploads/OpenMP_Telecon_Talk_final.pdf)

New Event-based approach

- Implement a new version by **splitting the initial kernel**:
  - **Parallelize over events** instead of execute all particles
  - Could lead to hybrid CPU/GPU execution



```
#pragma acc loop
for each particle
   while particle still running
      step_forward();
      end_condition_time();
      diag();
   end while
end for
```

Initial method

```
while particles are still running
   #pragma acc loop
   for each particle still running
      step_forward();
   #pragma acc loop
   for each particle still running
      end_condition_time();
   #pragma acc loop
   for each particle still running
      diag();
end while
```

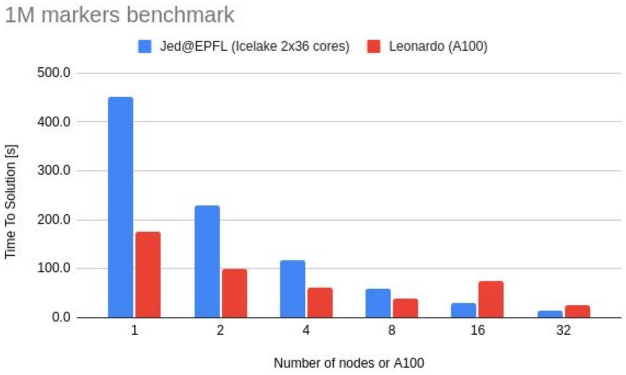
Event-Based

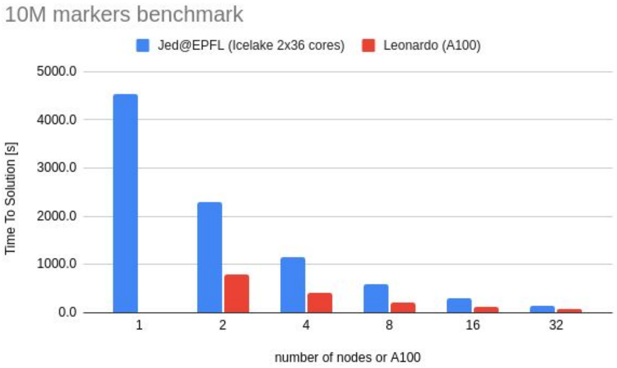
Independent events

SCITAS

Benchmarks

- "Sept2023" Benchmark:
  - **Jed**: 2x Platinum 8360Y, intel/2021.6.0, -Ofast -qopt-zmm-usage=high -march=native
  - **Leonardo**: A100, nvhpc/23.1, -O3 -acc -Minfo=accel -gpu=managed
  - Time-to-Solution, lower is better





Next steps and Conclusions

Next steps:

- Implement packing and sorting of particles on GPUs
- Insert event-based approach in all the code
- Investigate the possibility to unify CPU and GPU version
- Share work between CPU and GPU? New hardware (Grace/Hopper) might facilitate this

Conclusions:

- ASCOT5.5 is working with OpenMP-offload and OpenACC (for simulate_fo_fixed as a POC)
- OpenMP-offload version is extremely slow with gcc, reasonably fast with XL compilers
- OpenACC version fast and we only exploited one level of parallelism
- Will be interesting to test on newer GPU hardware (ongoing test on H100 and PVC)

On a more general note

- OpenMP offload/OpenACC (i.e. directive based) is probably the way to go as opposed to CUDA/Hip because of the high level of optimization of ASCOT5, but ...
- ... (performance) portability is a function of hardware, compiler version and language,
- and algorithmic modifications are necessary to get to next level of performance, but doesn't that contradict portability?