Smilei Workshop

Prague,November 2023

Achieving Performance

Charles Prouveur
CNRS | MDLS

What is performance in HPC?

- Two ways to look at it: fastest time to result and least energy to result
- In both case you will need:

  - Node-level efficiency

  - Efficient weak scaling (staying >75%)

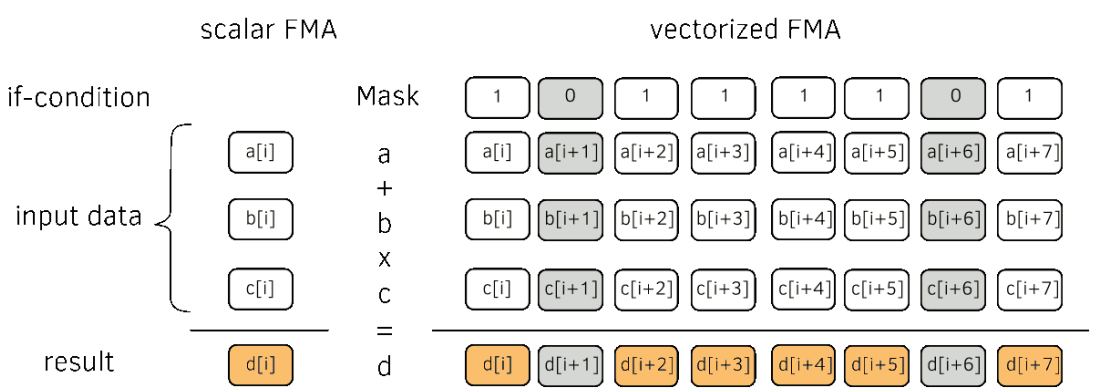
Configuration

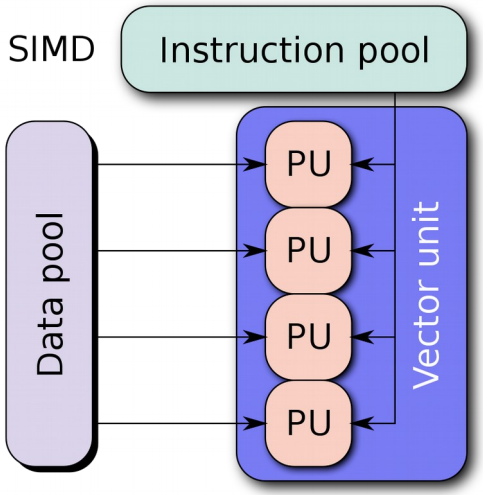
- Specifying the best compilation options is the first step to optimal runs
- -O2, -O3, -Ofast but also -arch / -gpu and -maxrregcount for nvcc (see for reference in the compilation scripts in scripts/compile/machine/)
- Optimize vectorization at compile time
- GPU acceleration

Vectorization

Single Instruction Multiple Data (SIMD) vectorization consists in performing on a contiguous set of data, usually called vector, the same operation(s) in a single instruction

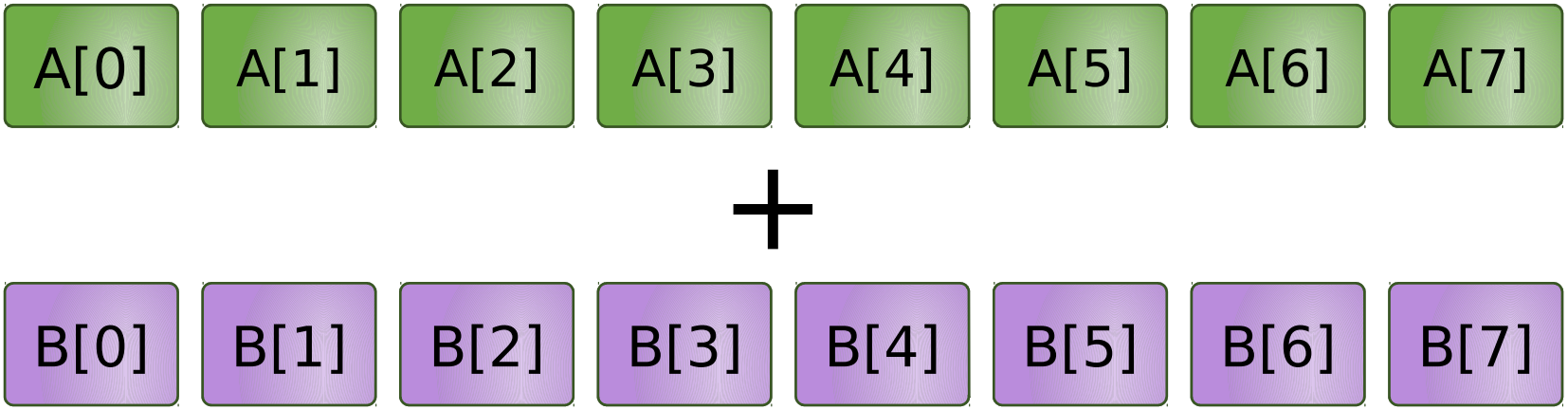
```
for(int i = 0 ; i < n ; i++) {
  if (condition dependent of i) {
    d[i] = a[i] + b[i]*c[i]
  }
}
```





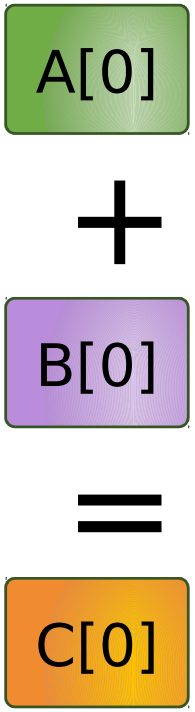
Understand the vectorized treatment of data

To sum vector A with vector B:



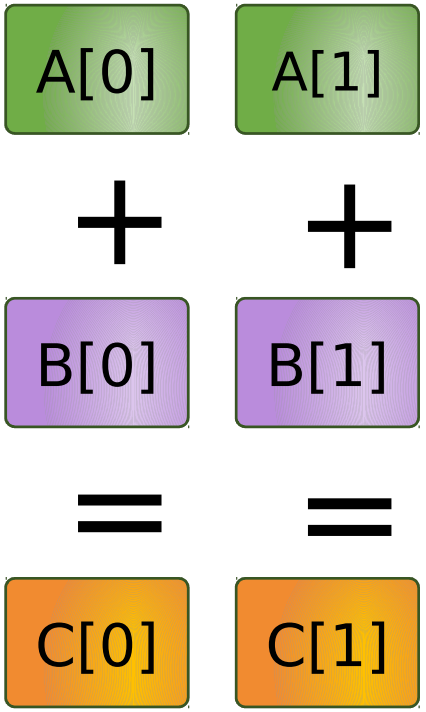
Understand the vectorized treatment of data

In a scalar loop, the core will perform each sum one by one...



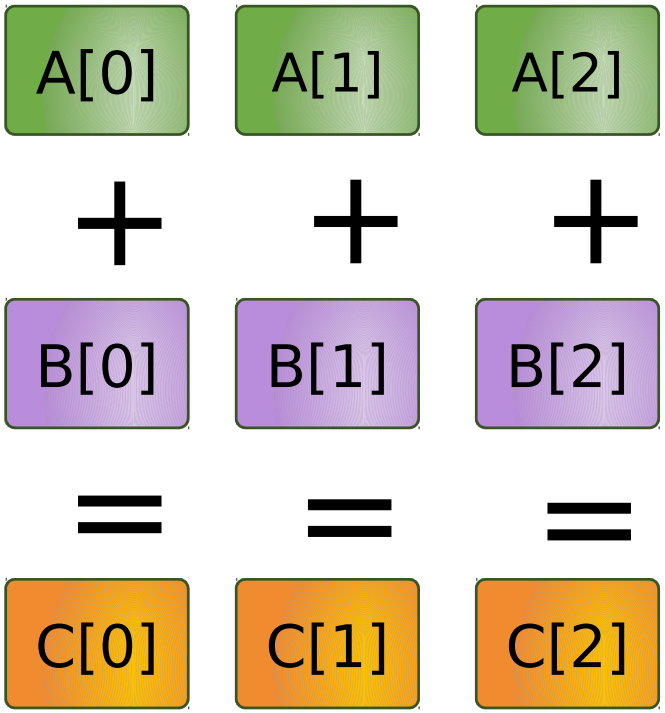
Understand the vectorized treatment of data

In a scalar loop, the core will perform each sum one by one...



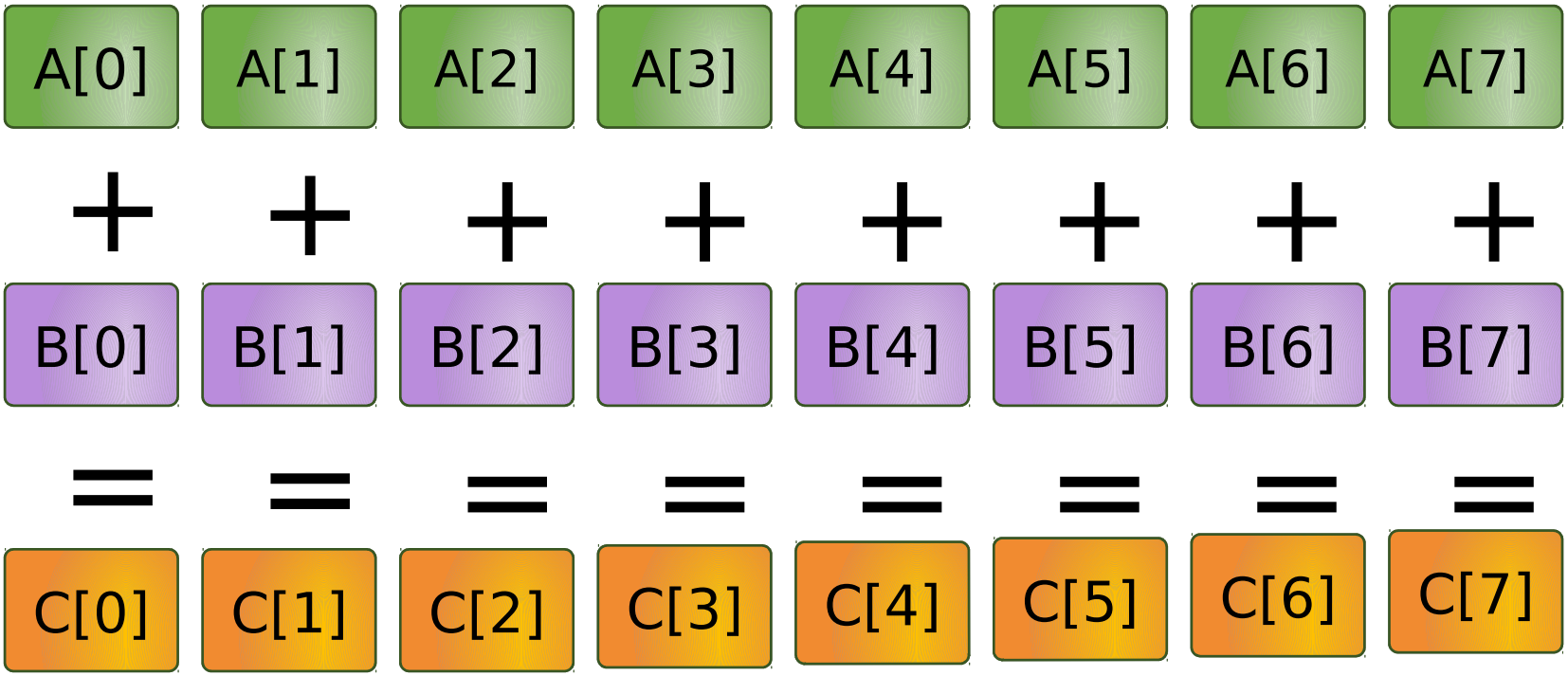
Understand the vectorized treatment of data

In a scalar loop, the core will perform each sum one by one...



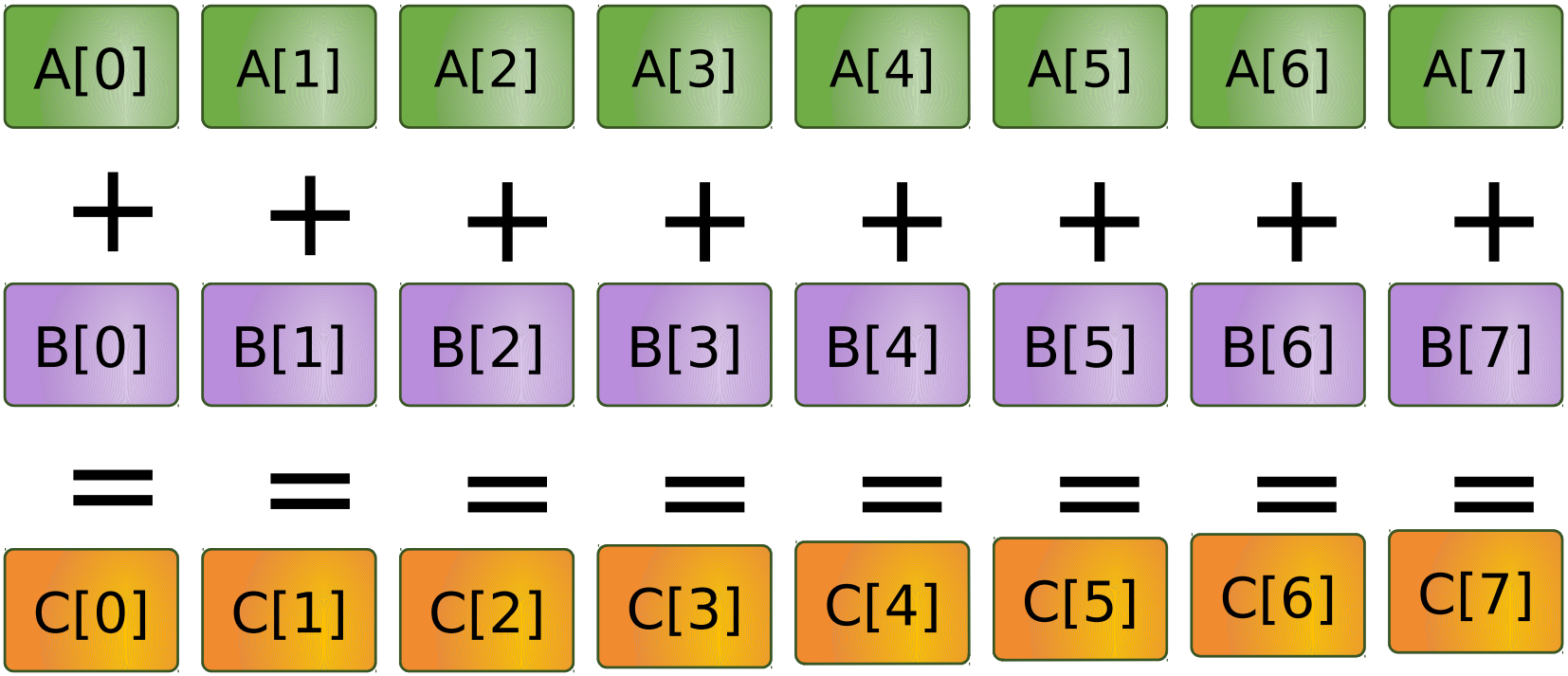
Understand the vectorized treatment of data

In a scalar loop, the core will perform each sum one by one until the end



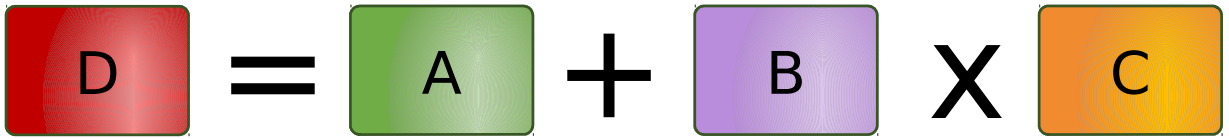
Understand the vectorized treatment of data

The vectorized version performs the sum of all elements at once



Understand the vectorized treatment of data

- Most modern processors perform both an addition and a multiplication in a single vectorized cycle (referred o as FMA for Fused-Add-Multiply instruction)



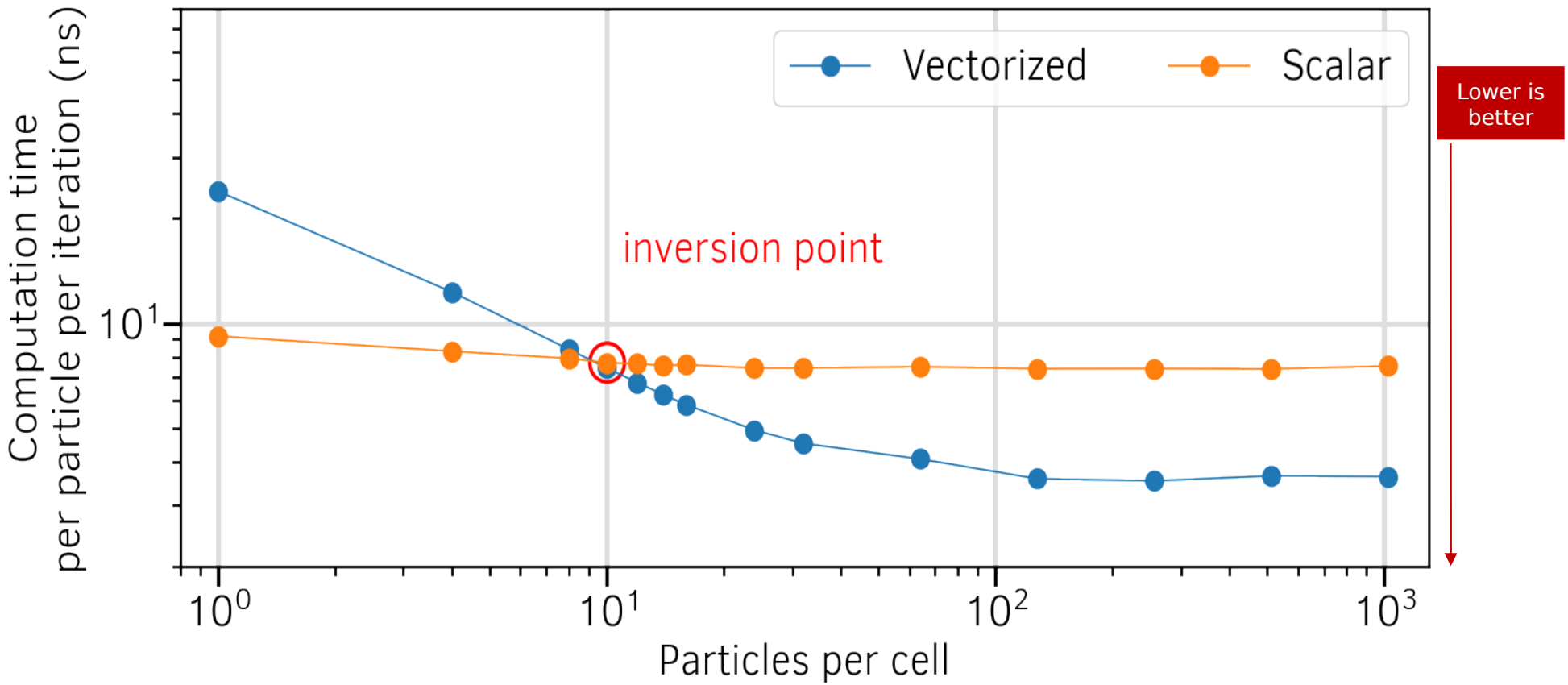
- If-branch can also be vectorized using masks
- Largest vectors are composed of 8 double-precision floats (AVX512 for instance)

How to change the vecto

- Specify at compile time the proper options:
  - the architecture (-march=haswell for example)
  - The instruction sets: -xCORE-AVX2 for example
- There are plenty of examples in /scripts/compile_tools/machine
- Namelist: activate it with the bloc Vectorization() in your namelist (more details in the doc)

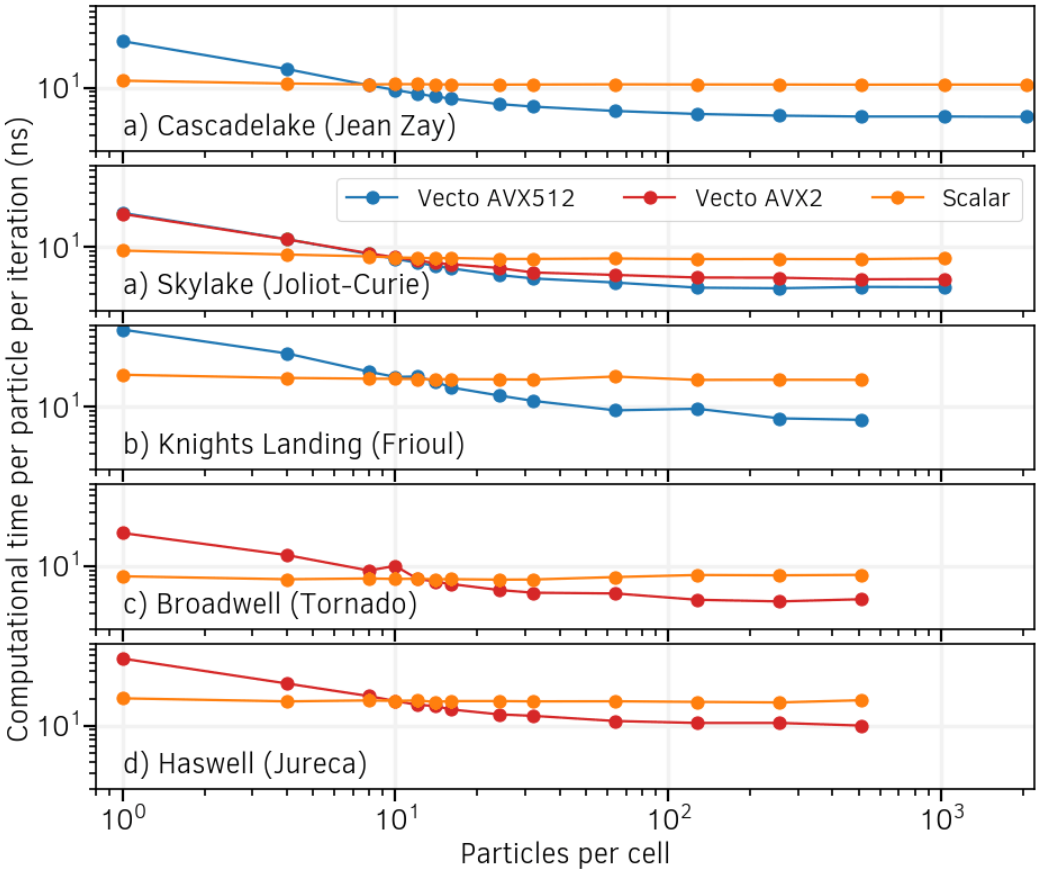
Vectorized versus scalar operator implementations

Thermal plasma 3D benchmark on a Skylake node (2 MPIs x 24 OMPs)



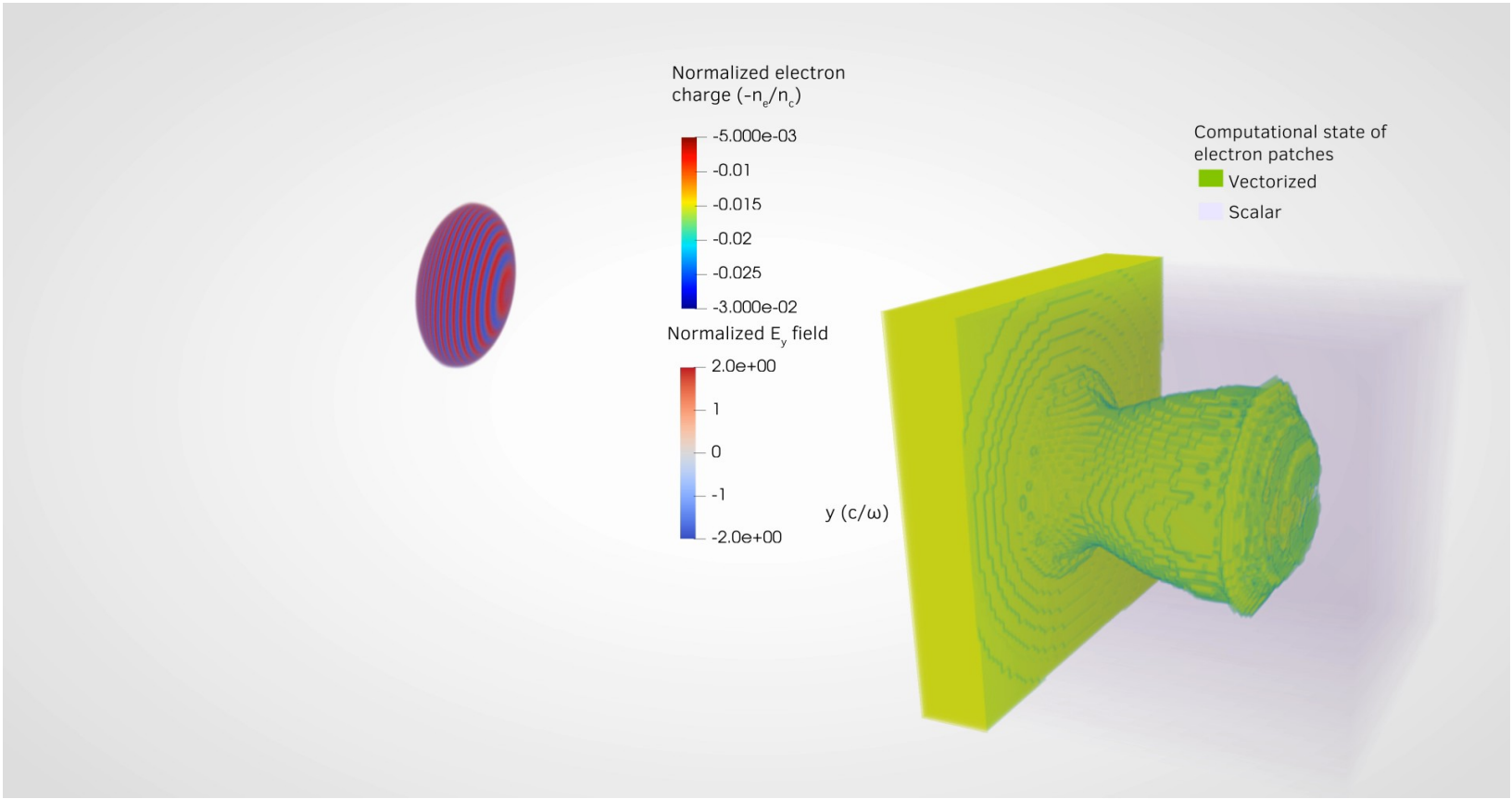
[1] A. Beck, *et al.* , *Adaptive SIMD optimizations in particle-in-cell codes with fine-grain particle sorting*, Computer Physics Communications 244, 246-263 (2019) arXiv:1810.03949

Vectorization results



Particle computational cost as a function of the number of particles per cell. Vectorized operators are compared to their scalar versions on various cluster architectures. Note that the Skylake compilations accepts both AVX512 and AVX2 instruction sets.

Adaptive vectorization



GPU Acceleration

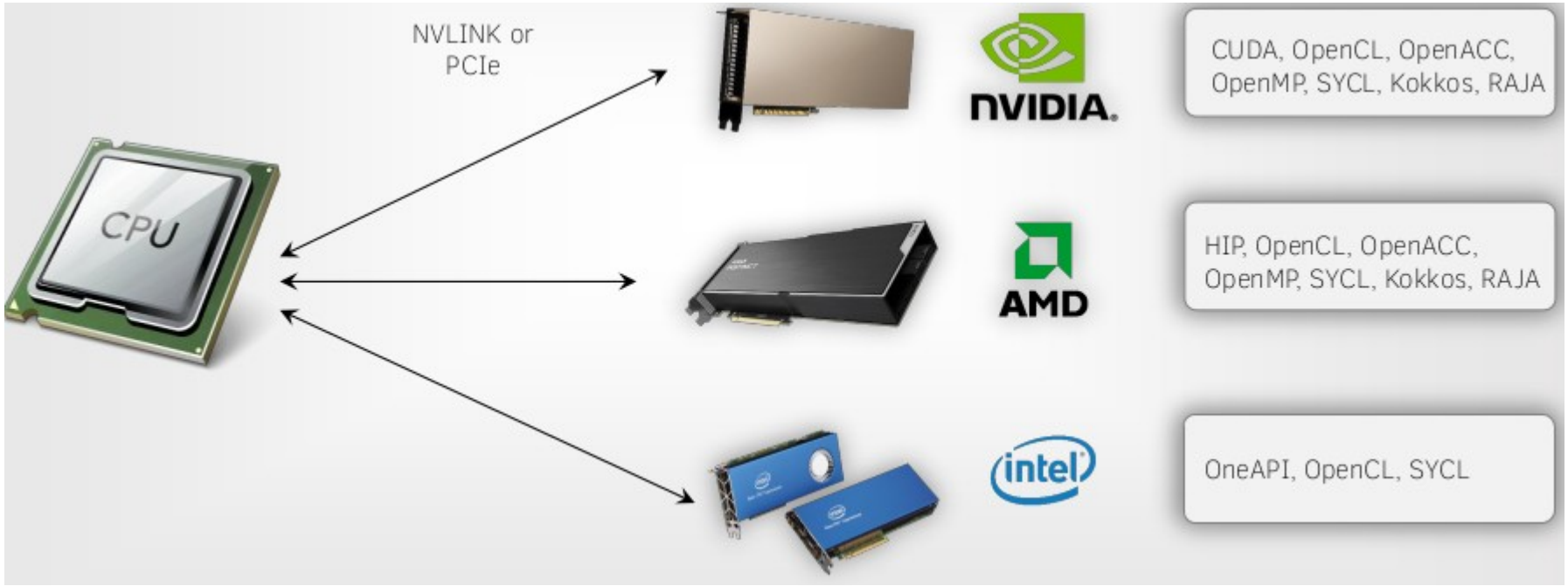
As most new supercomputers are GPU based, A lot of the development effort of SMILEI has been focused on offloading computations on GPUs.

Either using Pragmas (with OpenAcc or OpenMP) or using CUDA/HIP.

(6 team members so far involved in this endeavour)

What is a GPU?

A GPU works as an accelerator and needs a CPU for system tasks (IO, network communication...)



State of GPU support (5.0)

Currently supported:

- Both AMD & NVIDIA GPUs
- Cartesian geometry: 2D & 3D
- Order 2
- Moving window
- Diagnostics

Not yet supported:

- AM geometry
- Envelope
- PML
- Additional physics modules: Ionization, QED, collisions
- Dynamic load balancing

How to accelerate your computations with GPU?

- Options need to be specified at compile time:

  - architecture with -arch=SM_70 for example
  - -acc &/or -cuda, or nothing depending on the CUDA & NVHPC version

- In the namelist: add “gpu_computing = True” to main()
- Additional information:

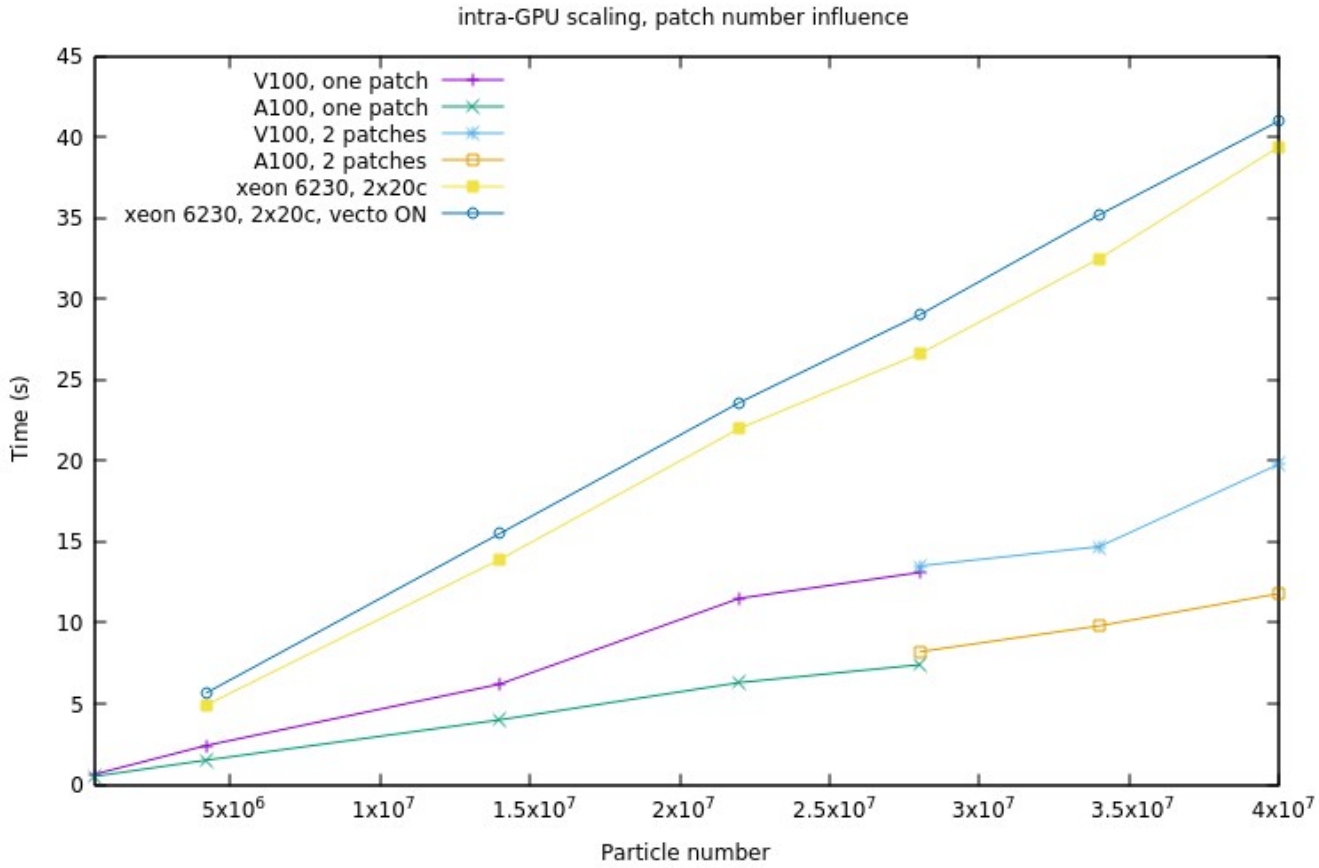
  - https://smileipic.github.io/Smilei/index.html

  - see the doc with sphinx + make doc

Scaling intra node

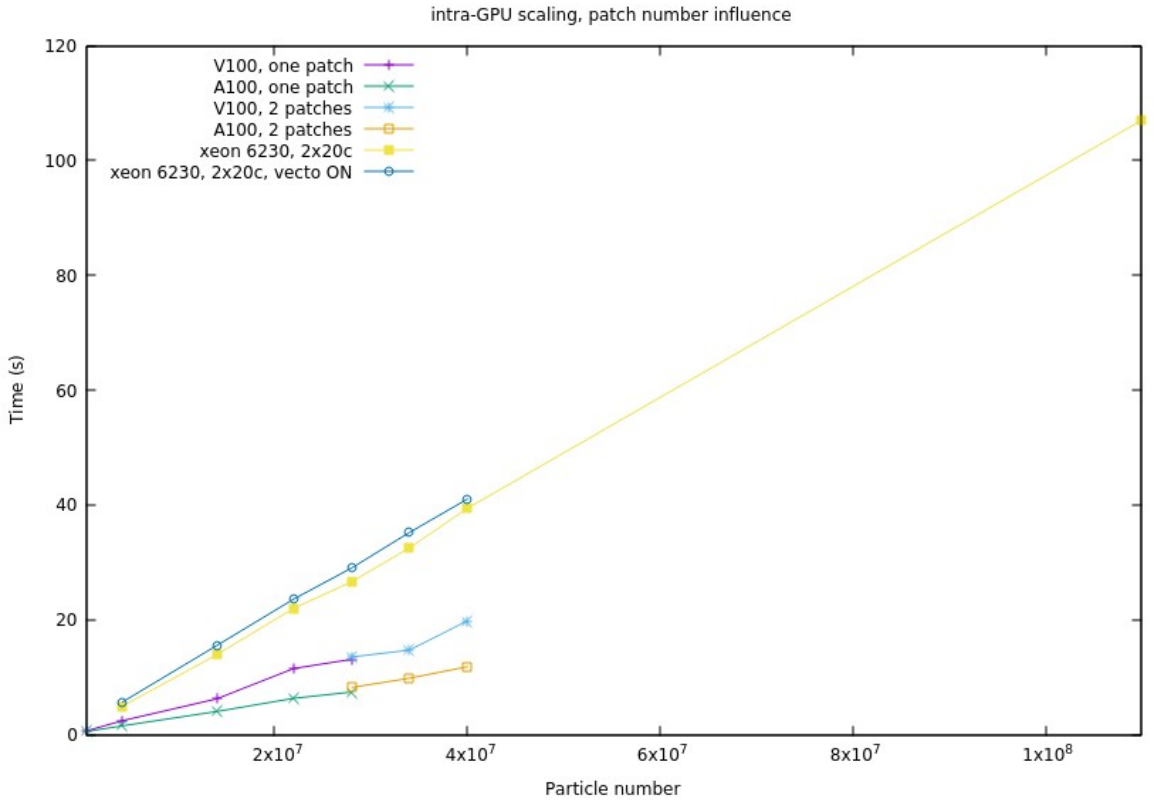
Case used:

3D thermal plasma
N cells in every direction
8 ppc
2 species



Scaling intra node

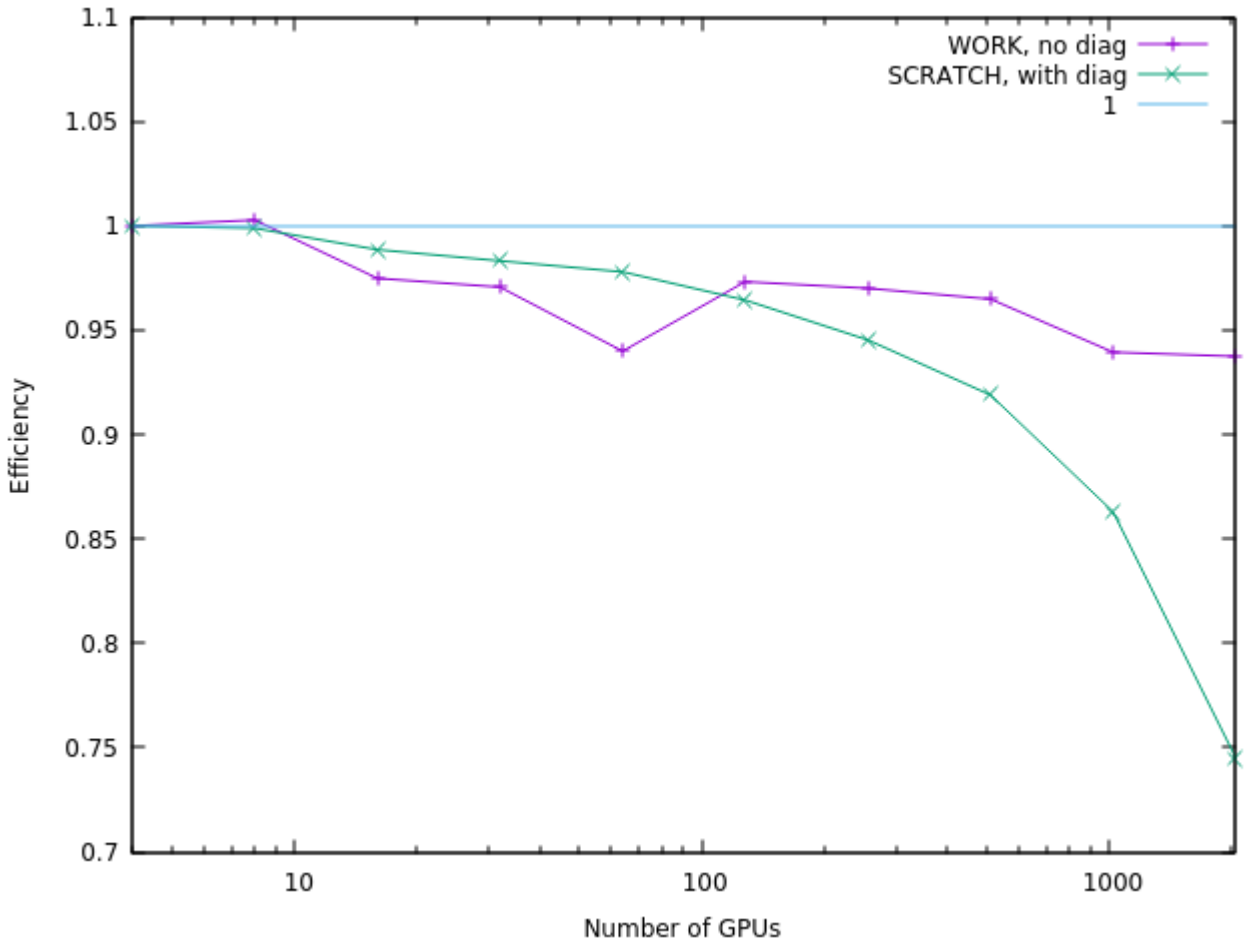
Note: CPU node's RAM capacity is much bigger than a GPU's



Weak scaling

Example on Adastra:

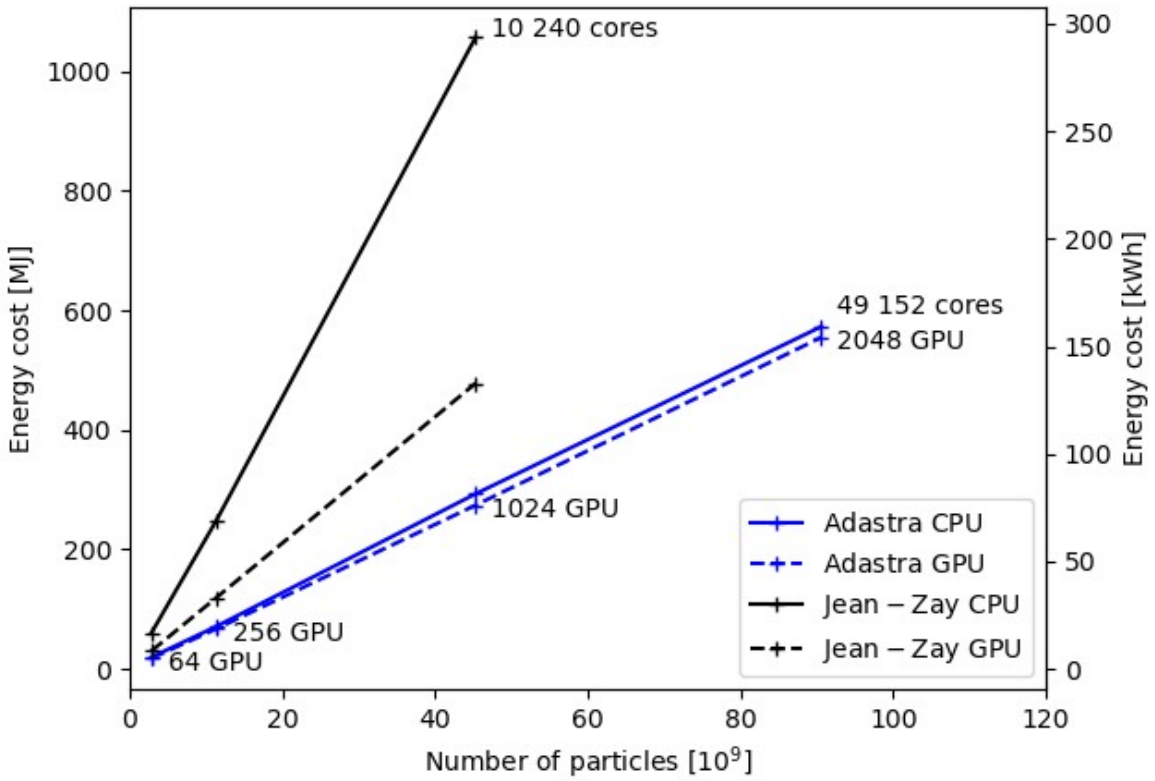
- 2D case
- Scaling done from 4 to 2048 GPUs
- 128 x 128 x (900pcc) x 3 species ~45e6 particles per GPU



Measuring energy efficiency

Tracking the energy of your job can be done with, for example on Jean-ZAY:

sacct -j JOBID --format=elapsed,consumedenergy,consumedenergyraw



GPU offloading guidelines

Guidelines On CPU:

- Have reasonably small patches. Small patches are beneficial to efficient load balancing and cache use, but they increase the synchronization costs. The optimal patch size depends strongly on the type of simulation. Use small patches (down to 6x6x6 cells) if your simulation has small regions with many particles. Use larger patches (typically 100x100 or 25x25x25 cells) otherwise.
- For high performances, each process should own more patches than threads.
- Have only as many MPI processes as sockets

Guidelines On GPU:

- One patch, 4 is certain circumstances
- One MPI process per GPU
- Fill the GPU as much as possible
- Increasing the number of patch allows the possibility to fill more RAM

For both: create as little as possible outputs/diagnostics because of their overhead
If you do, use the fastest filesystem available (scratch)

Thank you for your attention!

Thanks for supporting this event

Contributing labs, institutions & funding agencies