ORB5 on GPU:

summary, status, plans, lessons learned

Thomas Hayward-Schneider[1]

with input from the ORB5 team[1,2,3]

(pt.1 figures taken from Ohana et al. CPC 2020)

[1] Max Planck Insitute for Plasma Physics, DE

[2] SPC, EPFL Lausanne, CH | [3] University of Warwick, UK

Outline

- Background: ORB5
- Background: ORB5 on GPU
- Mixing CPUs and GPUs with ORB5
- Issues

Background - ORB5

ORB5: Global, gyrokinetic, EM, PIC code

- Phase-space markers (particles)
- Fields on FE mesh (Fourier filtered)
- Long lived `Fortran`+`MPI` code
- Modernized, refactored, ported to OpenMP (multi-core) + OpenACC (GPU)
  - Work done by SPC+CSCS [Ohana et al., CPC 2020]
- Code described in [Lanti et al., CPC 2020]

ORB5 Parallelism: 3 levels

1. Domain decomposition
2. Domain cloning
3. Multi-threading

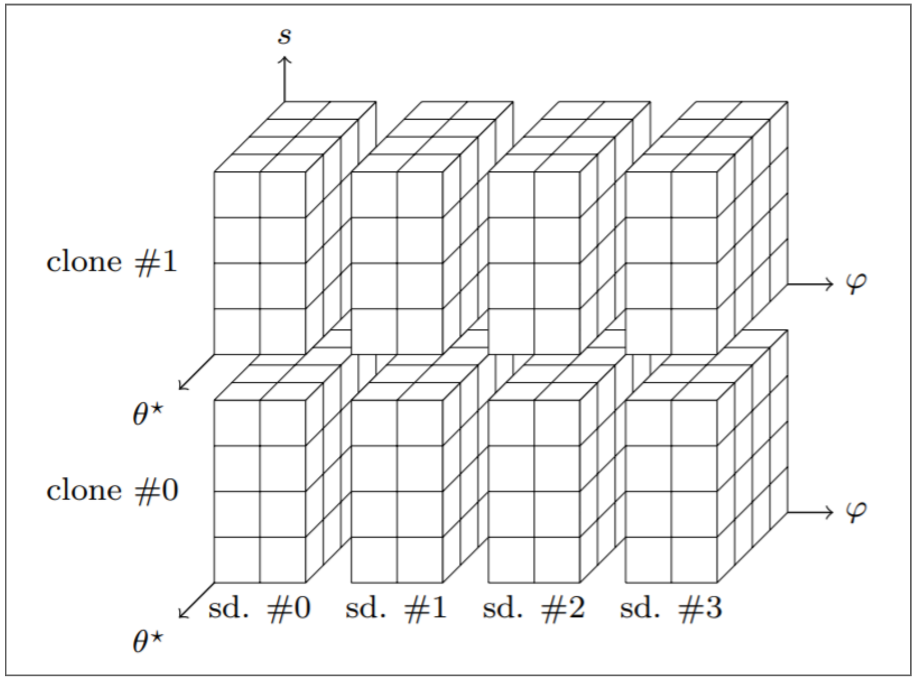
Cost/Performance Assumptions:

- Numerical cost
  ~ O(N_p) + O(N_g) + O(1) + O($F(N_g, N_p)$)
- Particles: pushing particles (linear in N_p)
- Grid: Solving fields (linear; $N \log N$)
- Particles/Grid: Particles $\leftrightarrows$ Fields (~ linear in N_p)
  Leading order contributions: ~ N_p
  Next contributions: ~N_g
  Assume leading term dominant

Parallelizing particle operations

1. Split toroidal planes by MPI (~128-512-way parallelism)
2. Duplicate torus (MPI) (arbitrary)
3. Each plane-clone runs on 1 MPI rank (process)
   = 1 core in pure MPI
   = several cores in OpenMP hybrid
   = 1 core + 1 GPU on GPU machines
   Multithreading originally applied only to particle operations

Spatial mesh parallelization

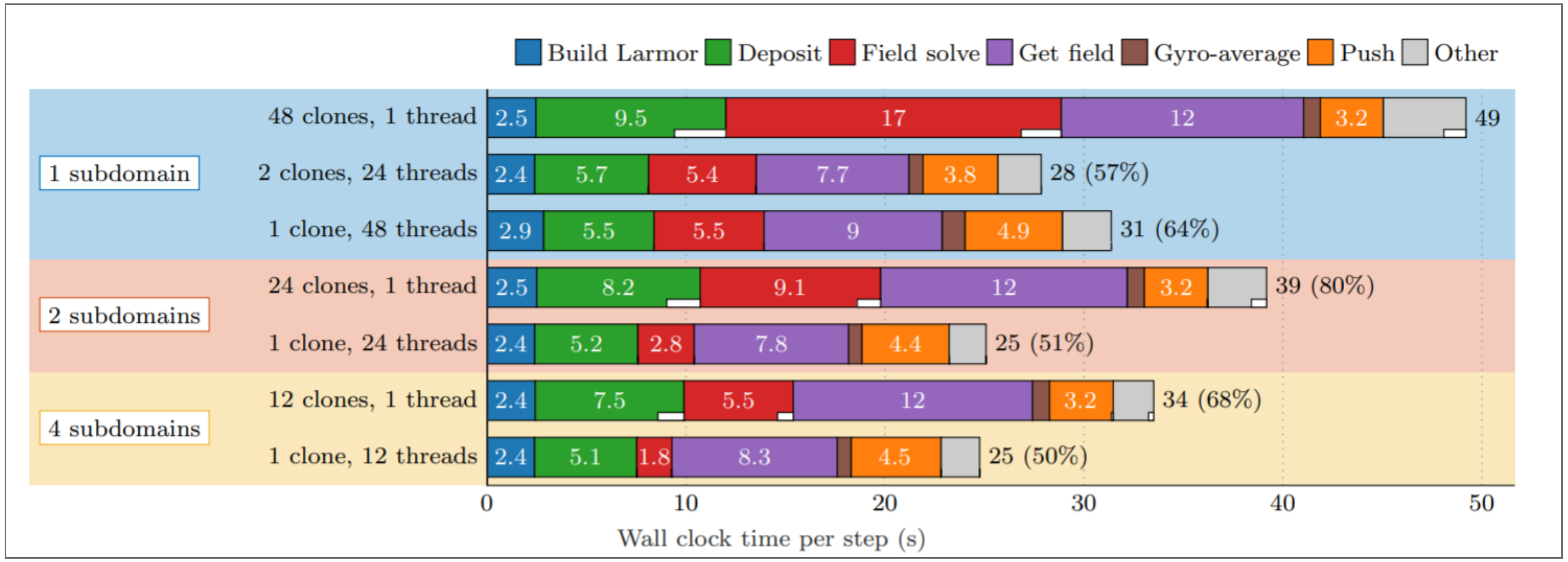


Within 1 MPI rank (here: 8 ranks: 2x clones; 4x decomposition):
`OpenMP` (CPUs) or `OpenACC` (GPUs) on markers

Aside: A word on OpenACC

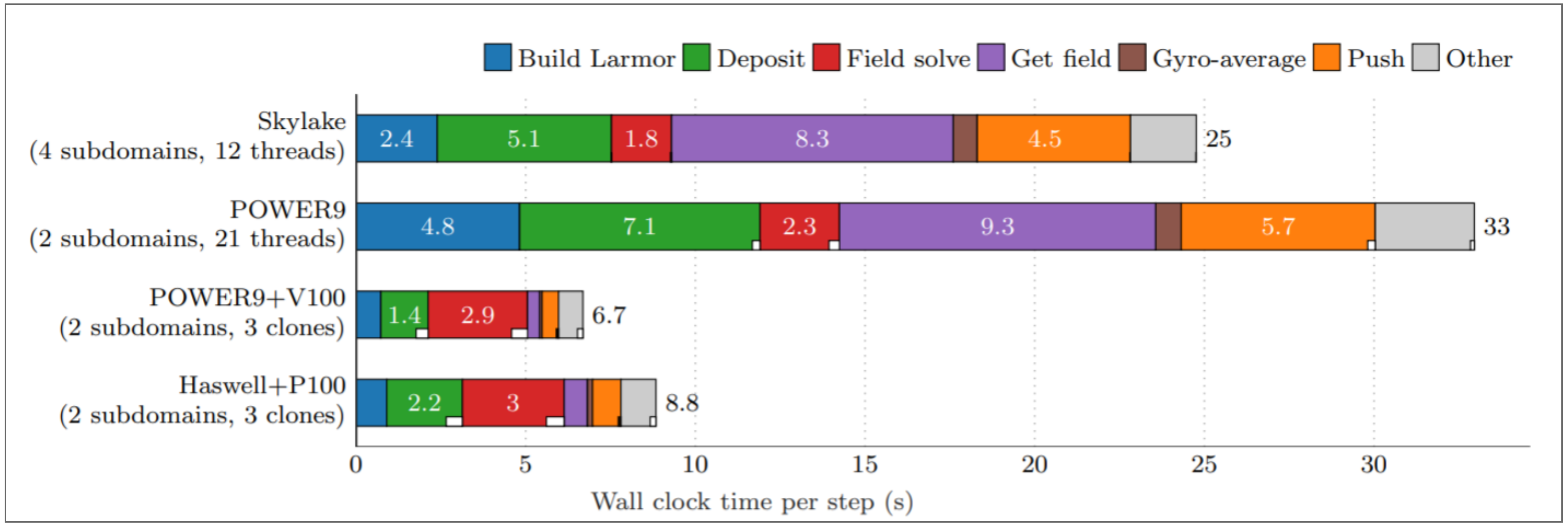
- Directive-based paradigm for GPU offloading
- Single source model
  - Huge advantages in avoiding code divergence
  - Nevertheless: Still some burden for developers
  - Automated testing framework (essentially) mandatory for any CPU+GPU code
- Supported primarily by Nvidia hardware + compiler (formerly PGI)
- Recent versions of OpenMP have approximately same features

Performance numbers (Multithreading)



- MPI+OpenMP hybrid beats pure MPI in total.

Performance numbers (GPU)



Note: Memory capacity of GPUs (here: 16GB) sets minimum parallelization [1].
Slow field solver wouldn't warrant such parallelism

[1] See later for further complications

Part 1 Summary

- OpenACC effectively accelerates (dominant) particle parts of ORB5 PIC code on GPUs
- Sufficiently successful that particle parts not necessarily still dominant
- GPU memory capacity becomes issue – adds lower bound on parallelization
  - Particles live on GPUs to minimize CPU $\leftrightarrows$ GPU transfer bottlenecks
- Light-touch port: < 500 `!$acc` directives

Beyond OpenMP ***or*** OpenACC

OpenMP and OpenACC in ORB5

- In general, OpenMP ***and*** OpenACC directions are around particle loops in ORB5
  - If compiling for CPU, OpenMP directives are used
  - If compiling for GPU, OpenACC directives are used
- Not many other places in code accelerated with either OpenMP/OpenACC
  Question arose in 2020: Can we accelerate (when running on GPUs) some operation which is tricky to write on a GPU?

Aside: code sample

```fortran
! Standard OpenMP loop
!$omp do
do ip=1,num_particles(species_i)
   ...
end do
!$omp end do
```

Build ORB5 with `OPENMP` to enable

Aside: code sample

```fortran
! Standard OpenACC loop
!$acc parallel loop
do ip=1,num_particles(species_i)
   ...
end do
!$acc end parallel loop
```

Build ORB5 with `OPENACC` to enable

Aside: code sample

```
! Typical ORB5 particle loop
!$acc parallel loop
!$omp parallel do
do ip=1,num_particles(species_i)
   ...
end do
!$omp end parallel do
!$acc end parallel loop
```

Build ORB5 with `OPENACC` **OR** `OPENMP`.
Twin directives on the same loops make the options incompatible.

Aside: code sample (mixing)

```
!$acc parallel loop
!$omp parallel do
do ip=1,num_particles(species_i)
   ...
end do
!$omp end parallel do
!$acc end parallel loop

!$omp parallel do
do ix=1,100
   ... ! Some other loop to accelerate
end do
!$omp end parallel do
```

How can we compile this to accelerate the first loop on the GPU and the second loop on the CPU?

Aside: code sample (mixing)

```
!$acc parallel loop
!pomp parallel do     ! this is now just a comment
do ip=1,num_particles(species_i)
   ...
end do
!pomp end parallel do
!$acc end parallel loop

!$omp parallel do
do ix=1,100
   ...
end do
!$omp end parallel do
```

Change `!$omp` to `!pomp` if it coexists with `!$acc`.
Now compiles, but need to fix OpenMP marker loop.

Aside: code sample (mixing)

```
#ifndef _OPENACC
#define pomp $omp
#endif

!$acc parallel loop
!pomp parallel do
do ip=1,num_particles(species_i)
   ...
end do
!pomp end parallel do
!$acc end parallel loop

!$omp parallel do
do ix=1,100
   ...
end do
```

Use preprocessor to change `!pomp` back to `!$omp`
(all pre-existing `omp` declarations were converted).

Example:

- New nonlinear collision operator [P. Donnel et al., PPCF 2020]
- Ported to GPUs, wants to be used on combination with quadtree smoothing algorithm.
- Quadtree smoothing is not ported to GPU, and it’s not obvious how to do so efficiently/flexibly/quickly.

Version -1

Code crashes when calling quadtree if compiled for GPU

Version 0

Check quadtree is disabled if compile for GPU

Nice that the code doesn't crash, but no progress

Version 1

When calling QT:

```
do species_i=1,nspecies:
    copy marker_data(:,species_i)          GPU->CPU
    call serial_QT(species_i)
    copy marker_data(weights,species_i)    CPU->GPU
```

Slow, serialized, but works.
Fine(?) if QT is called rarely

Version 2

When calling QT:

```
start copy of marker_data(:,species_i) GPU->CPU
do species_i=1,nspecies:
    wait for data(species_i)
    call OpenMP_QT(species_i)
    start copy of marker_data(weights,species_i) CPU->GPU
wait for copies.
```

OpenMP + OpenACC (+ async data movement).
Acceptable even if QT called every step

Simple test #1

1 node (2xSkylake + 2xV100, 20 threads per socket)

Option	**Speed [steps]**
no QT	131
v1	40
v2	98

QT speedup $(S_{no} - S_{v1})/(S_{no} - S_{v2})$ ~ 3

More realistic test:

Daint (1xHaswell + 1xP100, 12 threads per socket)

Option	**Time [s]**
no QT	687
v1	1760
v2	866

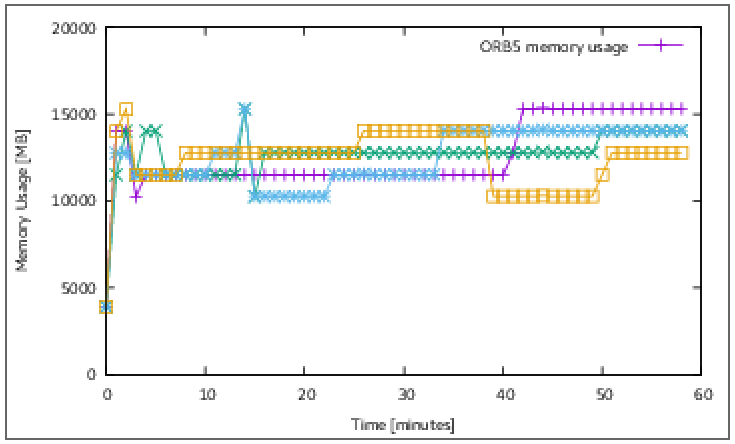
QT speedup = $(T_{v1} - T_{no})/(T_{v2} - T_{no})$ ~ 6

Issues

Memory limitations

- 16GB per GPU very limiting, given particle pushing speed of V100.
- For good performance, should fill GPUs as much as possible with markers.
- Overfilling => crash. Memory usage fluctuates in time. Difficult to understand/debug (OpenACC? Buffers?)
- Hard to fill more than ~2/3 GPU memory.

Memory fluctuations?



- GPU Memory usage reported by `nvidia-smi` tools difficult to explain
- Hard to reconcile even with memory debugging data
- Feedback on diagnosing GPU memory usage very welcome

Compilers

- PGI, now Nvidia HPC-SDK only compiler with sufficient `OpenACC` support
  - compiler issues outstanding.
  - `HDF5-mpi` library issues outstanding.
- Long term: migrate `OpenACC` => `OpenMP 4.5+` ?
  - for multi-vendor support

Strong vs Weak scaling

- Scaling in general affected by earlier assumption (Particles vs Grid)
  - depends heavily on physics studied
    - linear high-n Alfvén eigenmode studies among “worst” affected
- Strong scalability of code limited by field solver
  - Mixing OpenACC+OpenMP opened as avenue to help here
- More fundamentally GPUs help with throughput (weak scaling) rather than latency (strong scaling) [e.g. J. Brown *Excalibur 2020*]

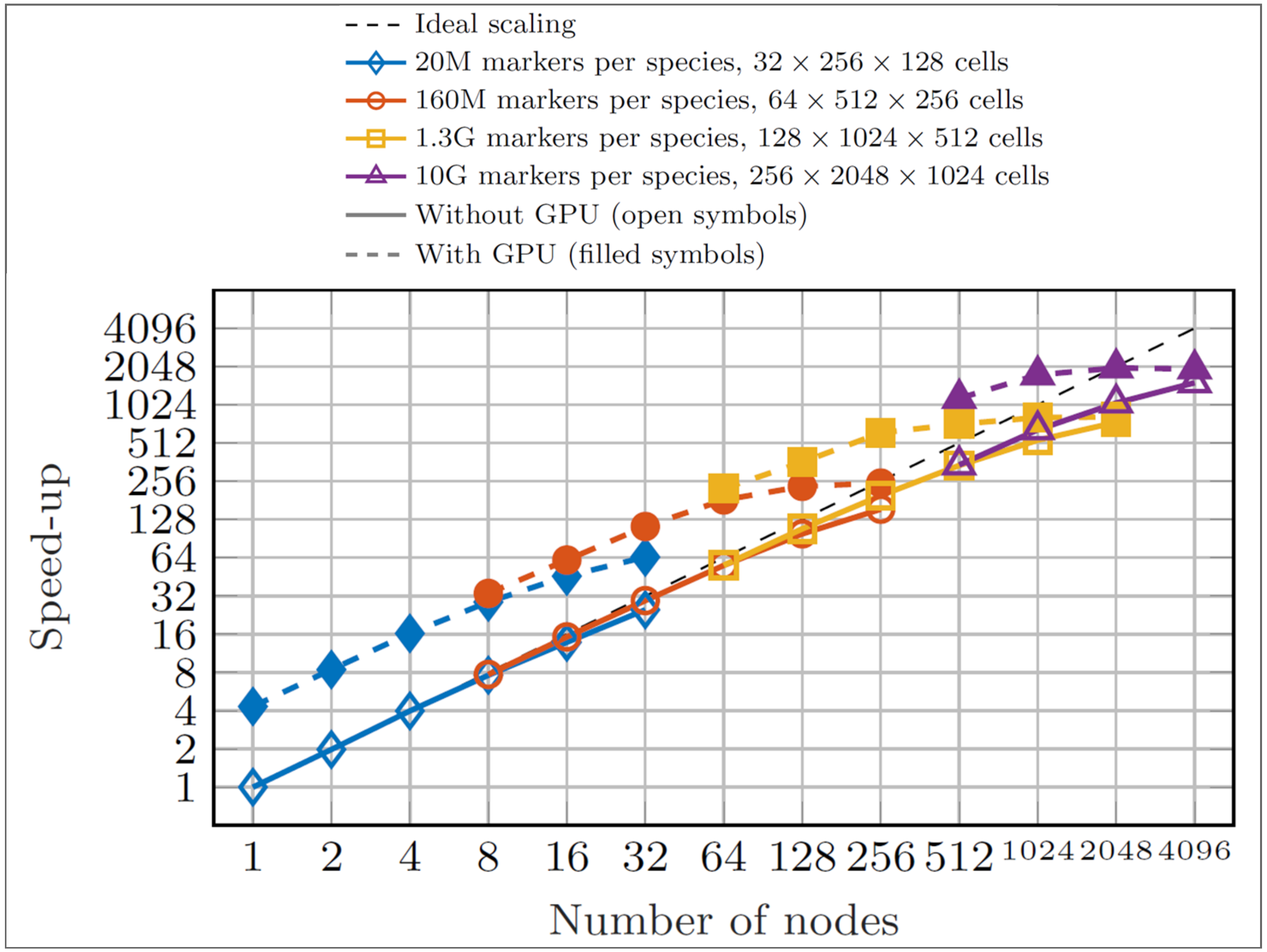
Summary

- ORB5 ported to GPUs with OpenACC
  - Particles live on GPUs, rest of code on CPU
  - Details in Ohana et al., CPC 2020
- Memory capacity limiting factor on m100
  - Sets minimum parallelization / maximum problem
- Strong scalability now (often) limited by field solver
  - Mixing OpenACC+OpenMP path forward

* Limited compiler support: as of now, not building with latest compilers

Backup

Summit parallel performance (hybrid strong/weak)


Ideal scaling
20M markers per species, 32 × 256 × 128 cells
160M markers per species, 64 × 512 × 256 cells
1.3G markers per species, 128 × 1024 × 512 cells
10G markers per species, 256 × 2048 × 1024 cells
Without GPU (open symbols)
With GPU (filled symbols)
Speed-up
Number of nodes