CERFACS

CENTRE EUROPÉEN DE RECHERCHE ET DE FORMATION AVANCÉE EN CALCUL SCIENTIFIQUE

Portage sur GPU d'un code Fortran HPC grâce à OpenACC

J. Legaux

Séminaire « En route vers l'exascale »
23 mai 2019

joeffrey.legaux@cerfacs.fr

www.cerfacs.fr

CERFACS

CENTRE EUROPÉEN DE RECHERCHE ET DE FORMATION AVANCÉE EN CALCUL SCIENTIFIQUE

Acknowledgements

This work would not have been possible without these people:

NVIDIA:

F. Pariente, F. Courteille, S. Chauveau

IBM:

P. Vezolle, L. Lucido

Cellule de veille technologique GENCI:

G. Hautreux

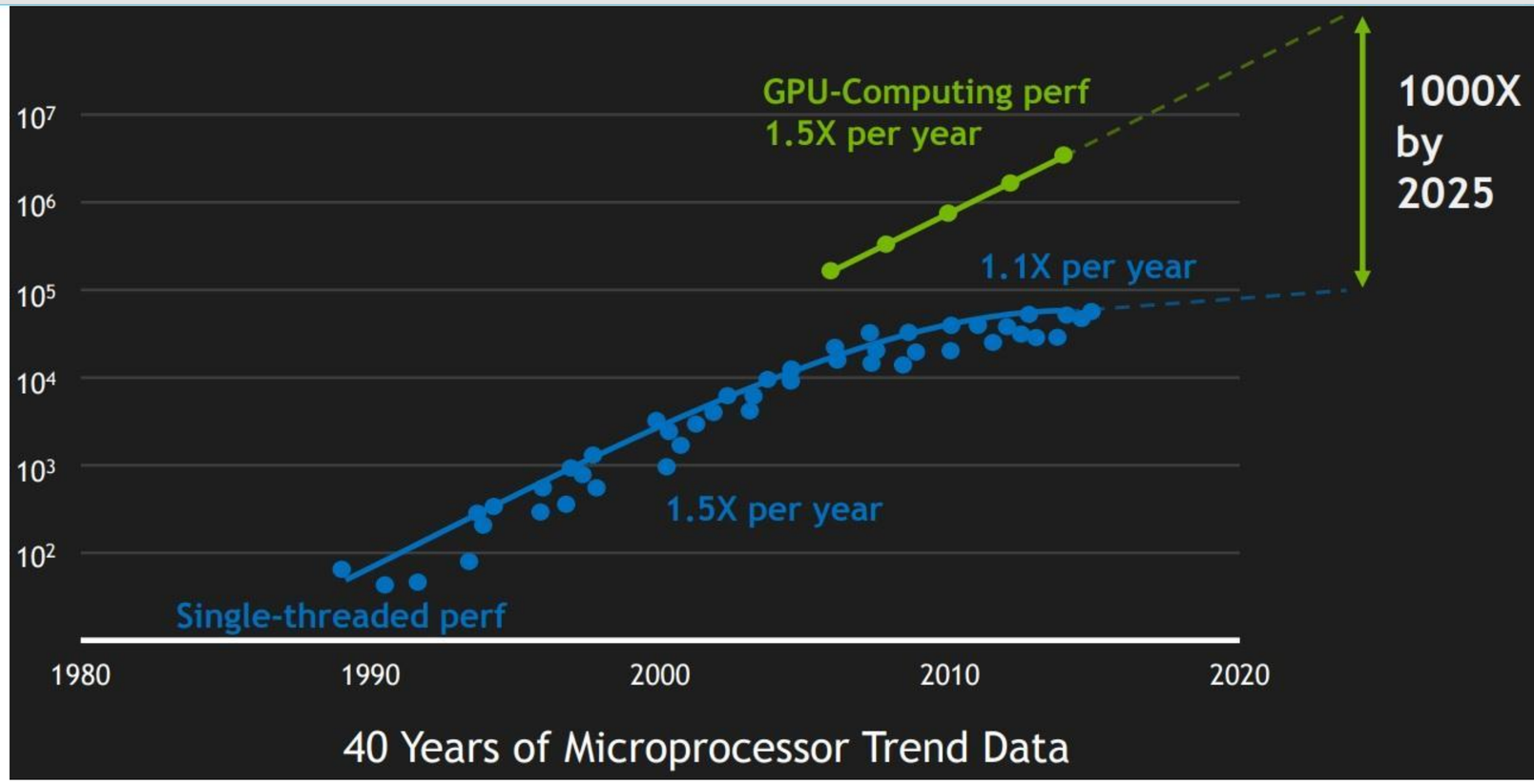
CERFACS:

I. d'Ast, N. Monnier, G. Staffelbach

Legacy codes and GPU

GPUs are gaining the favors of the HPC community thanks to their steady performance growth.

One has the choice to either write a new specific code, or port a legacy one



Legacy codes
+ widely established community
+ experience, validation, software environment…
- GPU port might prove challenging

Our focus : combustion with AVBP

Applications of AVBP

- Prediction of combustion in highly complex cases

Test cases

- The “Simple” test (3M)

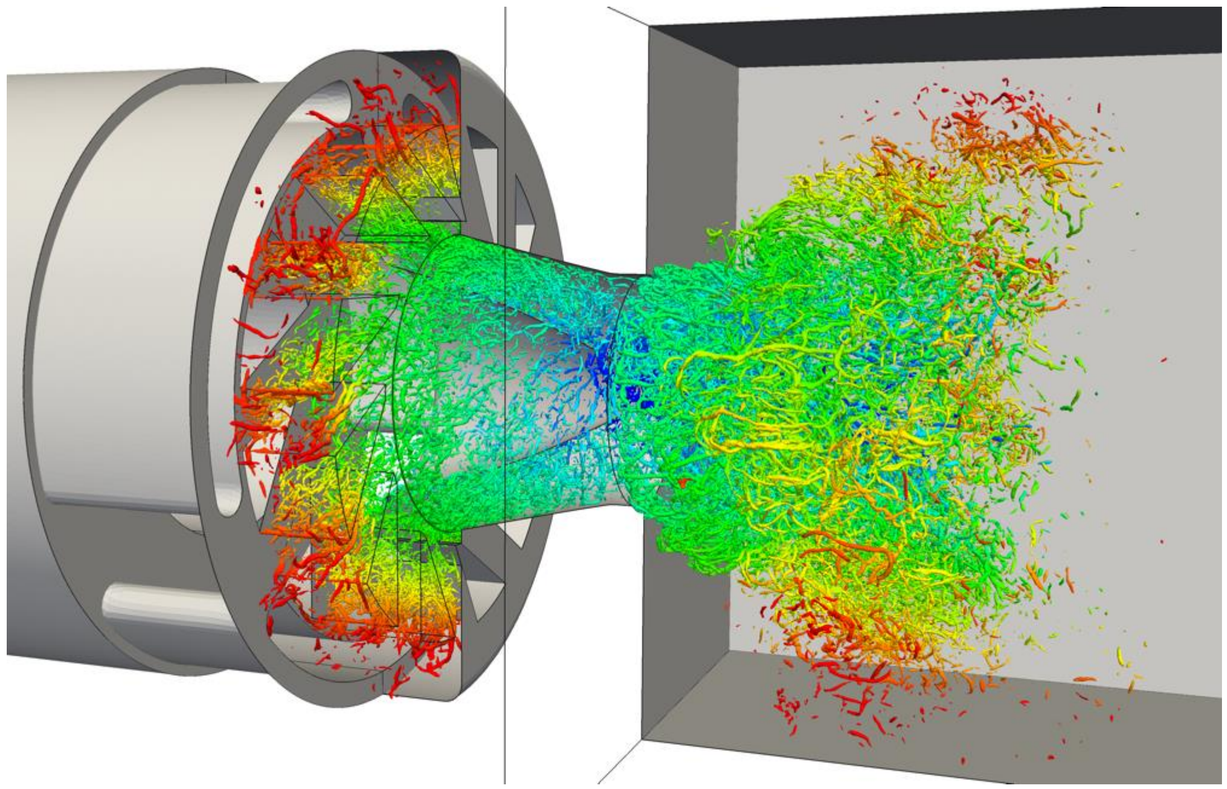
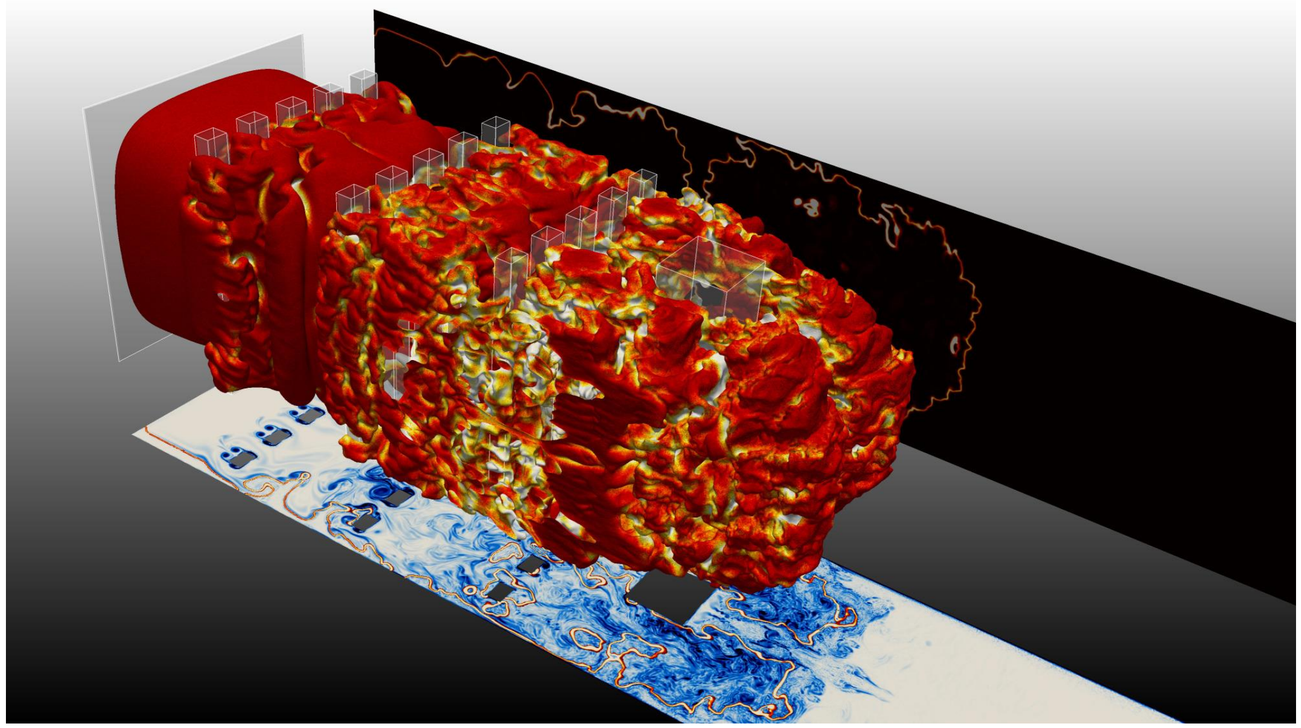


Image courtesy of V. Moureau (CORIA/ CNRS)

- Gas turbine simulation

- The “Explo 20MAO” test (20M)



Quillatre et al.

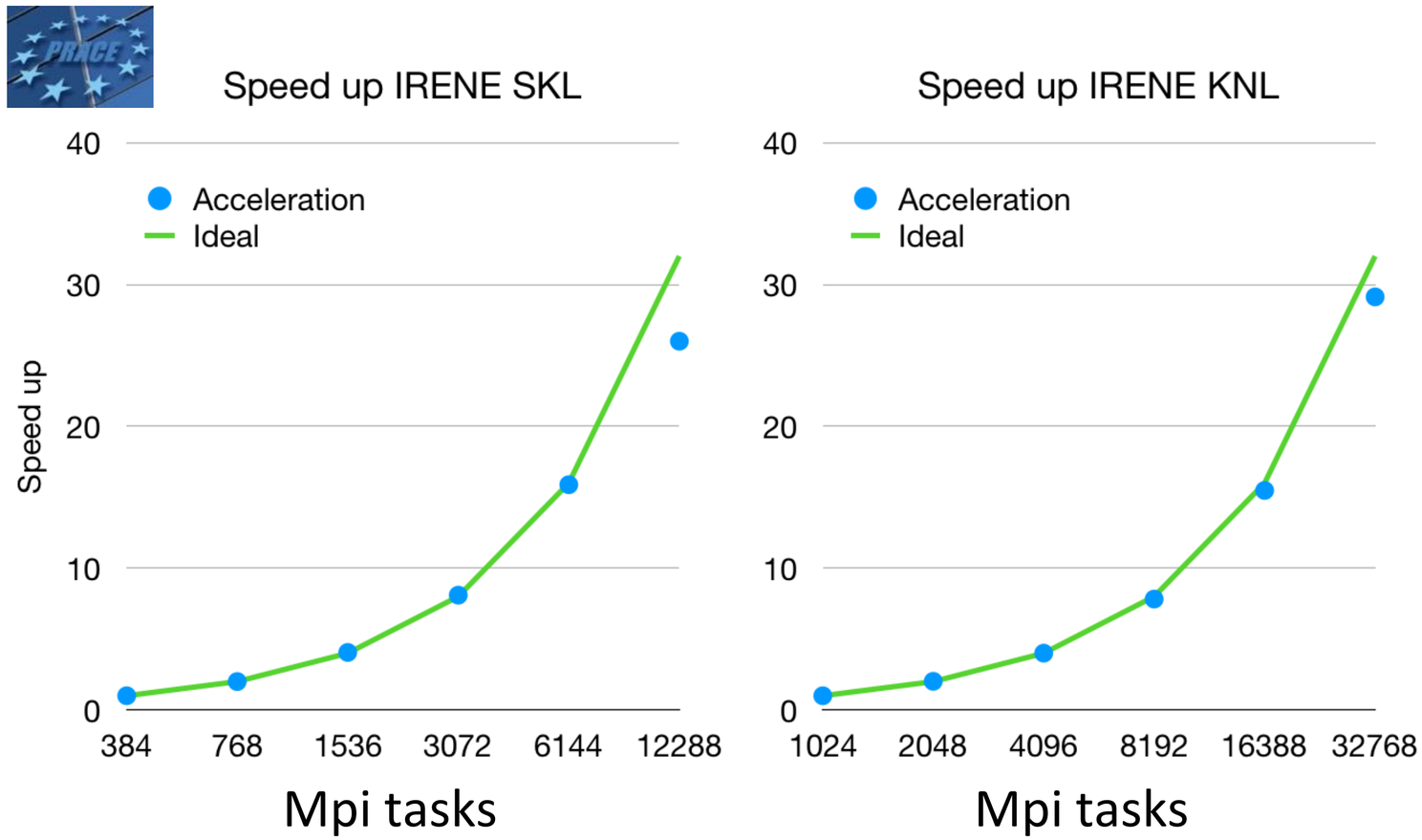
- Explosion in a confined space.

The AVBP development team

- Started in **1997**
- 300 users
- 20 **new** users/dev per year
- 2 ‘constant’ maintainers

➡ Almost all new devs are usually from academia and from a *CFD background not CSE!*

HPC and AVBP



- Excellent strong scaling response on CPU systems with full MPI
- Current record 200k cores 90% scaling

- How can we take advantage of rapidly expanding and performant CPU+GPU systems ? Ex: Jean Zay (IDRIS)

Constraints

- Code needs to **remain "simple"**
  - Very large fortran code
  - Active development and in "production"
  - **Limited HPC developers** and needs to be compatible with "CFD" students
  - Code needs to remain portable

Our choice : OpenACC

- The current fortran code basis must be kept
  - CUDA is not suitable
  - Directive programming models are perfectly suited

- **OpenACC vs OpenMP**
  - Simpler syntax for GPUs
  - Active (and enthusiastic) support from Nvidia and PGI
  - OpenMP too limited at the time (started in 2017)

How does AVBP work ?

- Fortran + MPI (+ OpenMP) + Parallel HDF5 I/O

Input/Read

Decomposition and distribution of data

Numerical Scheme

Main loop

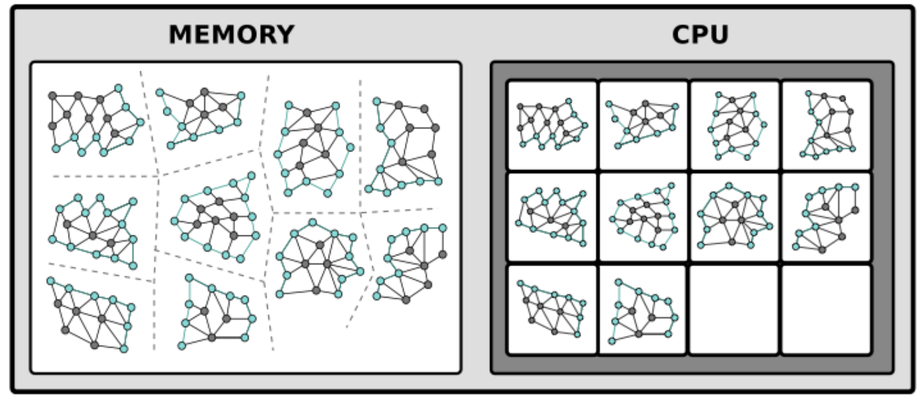
- Convective Scheme
- Diffusion Scheme
- Boundary treatment
- Physical models

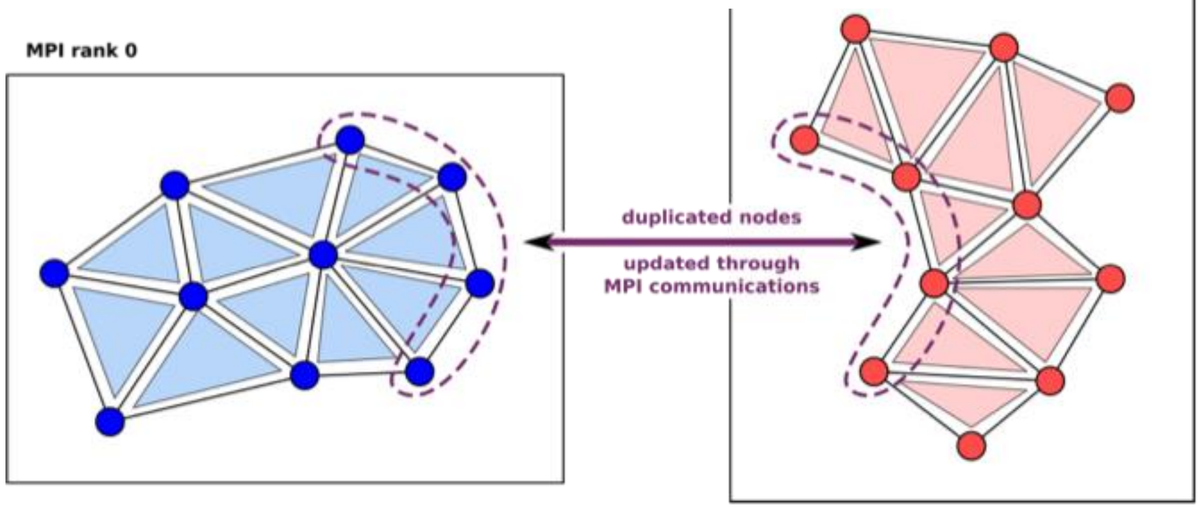
Postprocessing and I/O

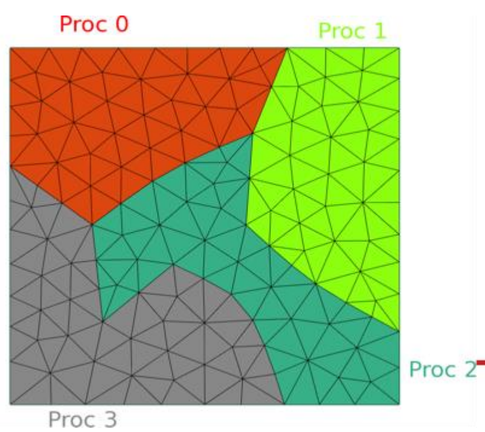
END ?

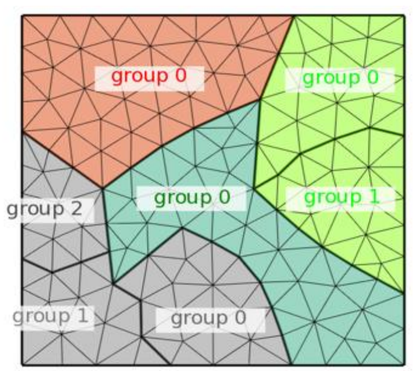
NO

YES





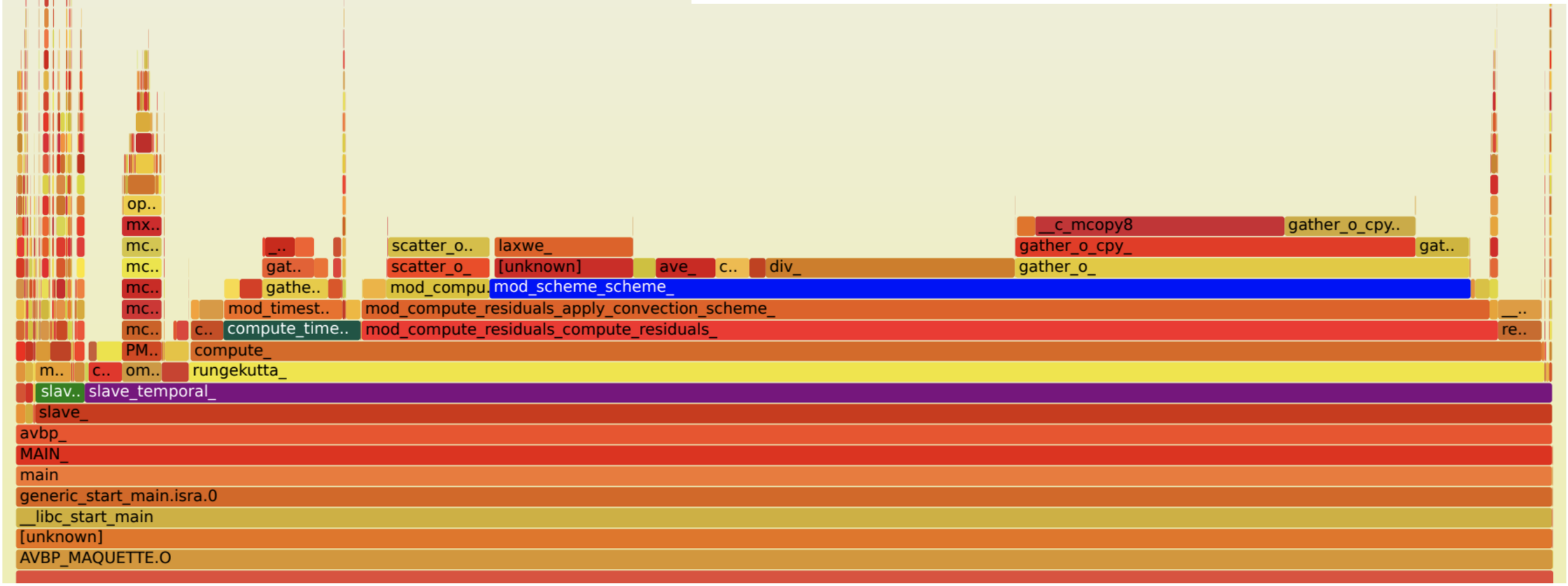




Extending AVBP for GPUs

Power 8 CPU only

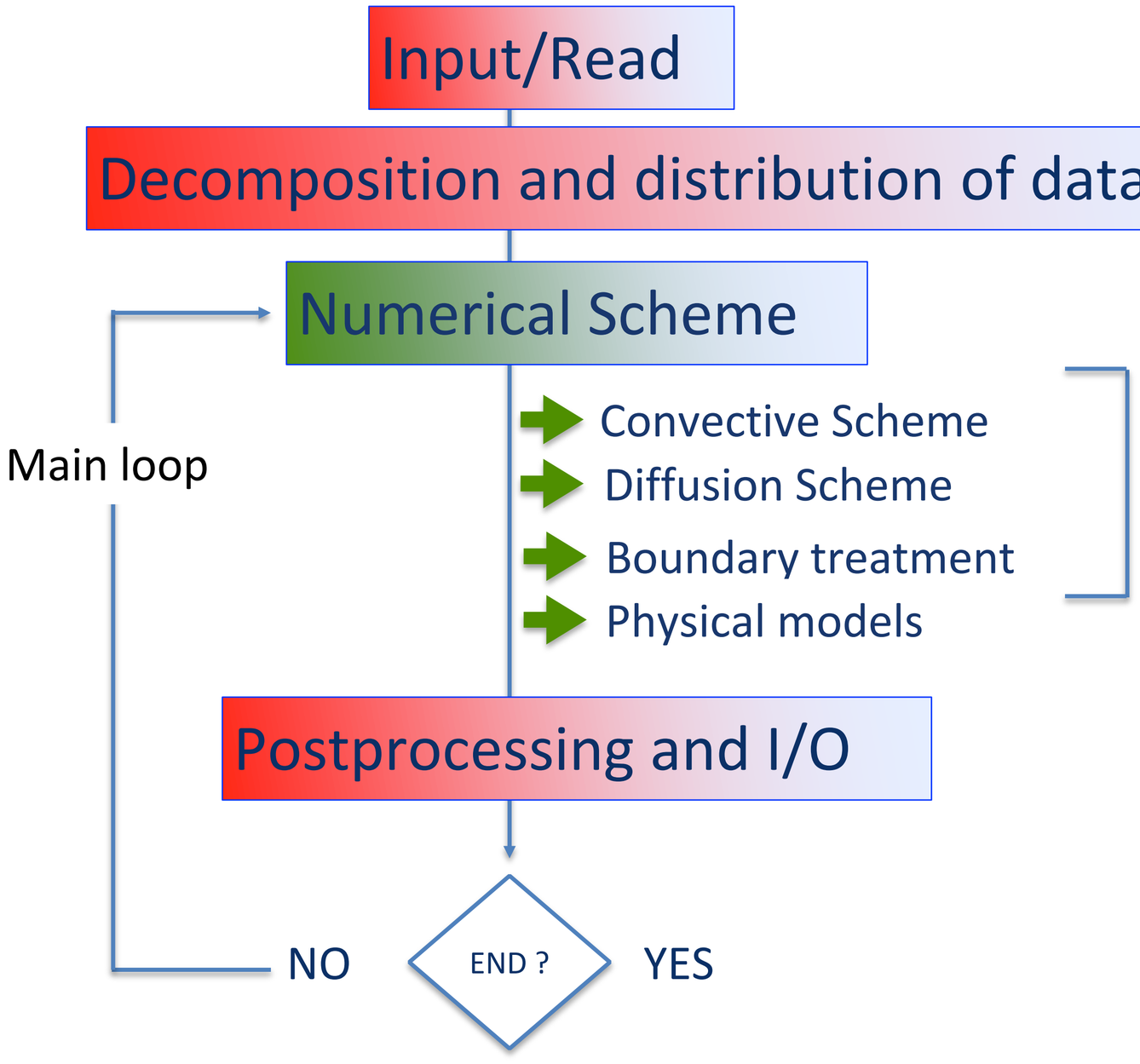
- Simplified source workflow profile
  - Deeper is higher
  - Scheme = 64% (usually around 80%)



Courtesy of Lucido et al.

Extending AVBP for GPUs

- Fortran + MPI (+ OpenMP) + Parallel HDF5 I/O



- **High I/O sections incompatible with GPU**
- **Compute intensive kernels compatible with GPUs**

Extending AVBP for GPUs

- Typical structure and Most intensive kernels

MAIN LOOP

```
DO n = 1, ngroup
  Call scheme (global R data, global RW data)
END DO
```

```
USE module only scheme_data

CALL function1(global_R_data,global_RW_data,
scheme_data)
..
CALL function…(global_R_data,global_RW_data,
scheme_data)
```

```
USE module only internal_data
DO i=1,ncells
  X[i] = B* X[i] +. A*Y[i]
END DO
```

Extending AVBP for GPUs

- Typical structure and Most intensive kernels

MAIN LOOP

```
DO n = 1, ngroup
  Call scheme (global R data, global RW data)
END DO
```

COARSE GRAIN

```
USE module only scheme_data

CALL function1(global_R_data,global_RW_data,
scheme_data)
..
CALL function…(global_R_data,global_RW_data,
scheme_data)
```

```
USE module only internal_data
DO i=1,ncells
  X[i] = B* X[i] +. A*Y[i]
END DO
```

FINE GRAIN

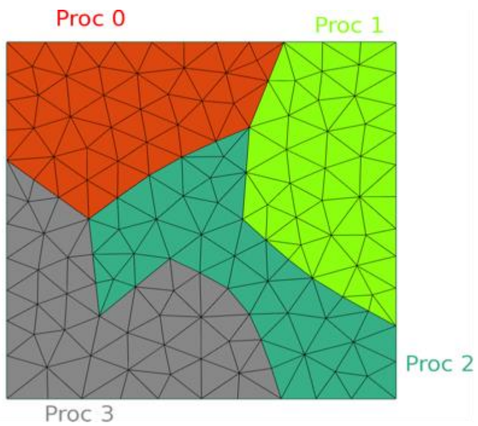
Coarse grain approach

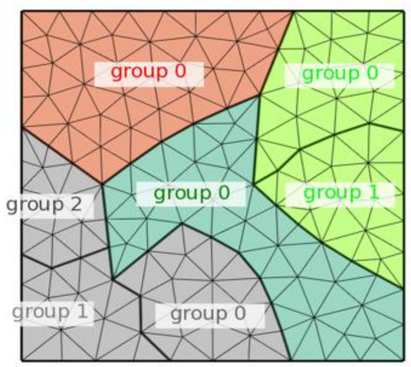
- Derived from current most effective OpenMP implementation

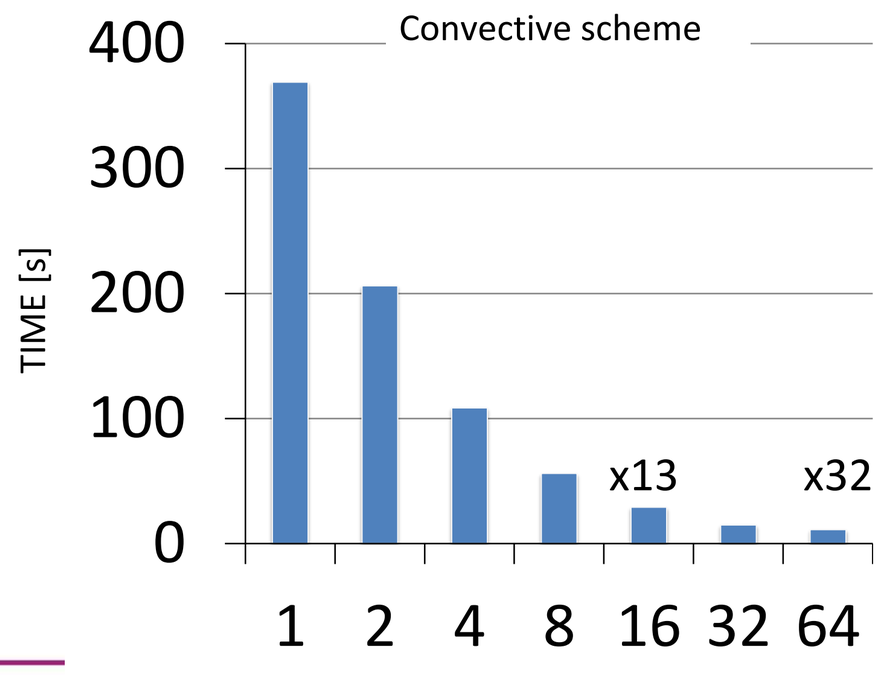
```
!$OMP PARALLEL DO (..)
DO n = 1, ngroup
  Call scheme
END DO
```

```
!thread 1
 Call scheme (group(1))
 Call scheme (group(2))
  …
 Call scheme (group(…))
```

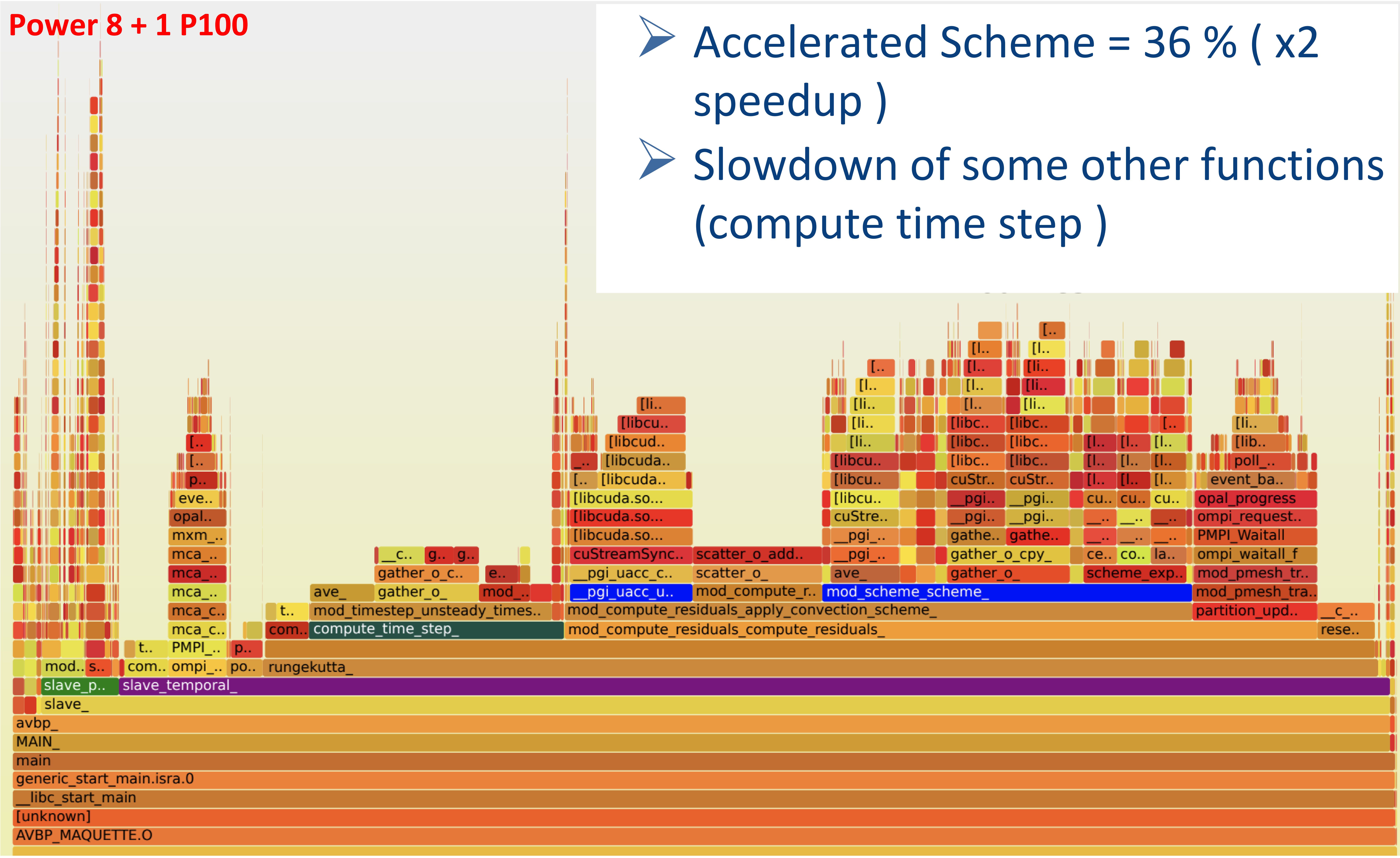
```
!thread …
 Call scheme (group(…))
   …
 Call scheme (group(…))
```







Coarse grain performance



- Accelerated Scheme = 36 % (x2 speedup)
- Slowdown of some other functions (compute time step)

Courtesy of Lucido et al.

Extending AVBP with OpenACC

- Coarse grain implementation encouraging but unsuccessful

  - ✦ Very few directives but …

  - ➡ **Modifications** of **high-level data structures**

  - ➡ “black box”

  - ➡ Some minor limitations of PGI ACC observed
    - ➡ Simple workflow is ok
    - ➡ Some workflows use features of the language currently incompatible with PGI ACC

Extending AVBP with fine grain

- Switch to small kernels
  - ✓ Only target computation-heavy loops in the code
  - ✓ Identify arrays that are used for those computations
  - ✓ **Explicitly manage** memory exchanges of those arrays between CPU and GPU memories
  - ✓ Explicitly offload the concerned loops to the GPU

- A tedious, **step-by-step work, but easy to check**
  - ✓ Each loop can easily be isolated, ported one at a time for debugging, optimisation or precision evaluation

But First … a more vector friendly structure…

- Code was started when long vector processors were no longer ‘the future’.

```
DO  n = 1, nnode
  DO nv = 1, nvert
    DO e = 1, neq
      array(e, nv, n) = …
    END DO
  END DO
END DO
```

- **Loops are unstructured and build for short array lengths**
- **Typical values for the loop:**
  - nnode : several millions to billions (mesh dependent)
  - nvert : 3 to 8 (mesh element type dependent)
  - neq : 1 to 15 (physics)

But First … a more vector friendly structure…

- Code was started when long vector processors were no longer ‘the future’.

```
DO  n = 1, nnode
  DO nv = 1, nvert
    DO e = 1, neq
      array(e, nv, n) = …
    END DO
  END DO
END DO
```

```
DO e = 1, neq
  DO nv = 1, nvert
    DO n = 1, nnode
       array(n, nv, e) = …
    END DO
  END DO
END DO
```

- **Typical values for the loop:**
  - nnode : several millions to billions (mesh dependent)
  - nvert : 3 to 8 (mesh element type dependent)
  - neq : 1 to 15 (physics)

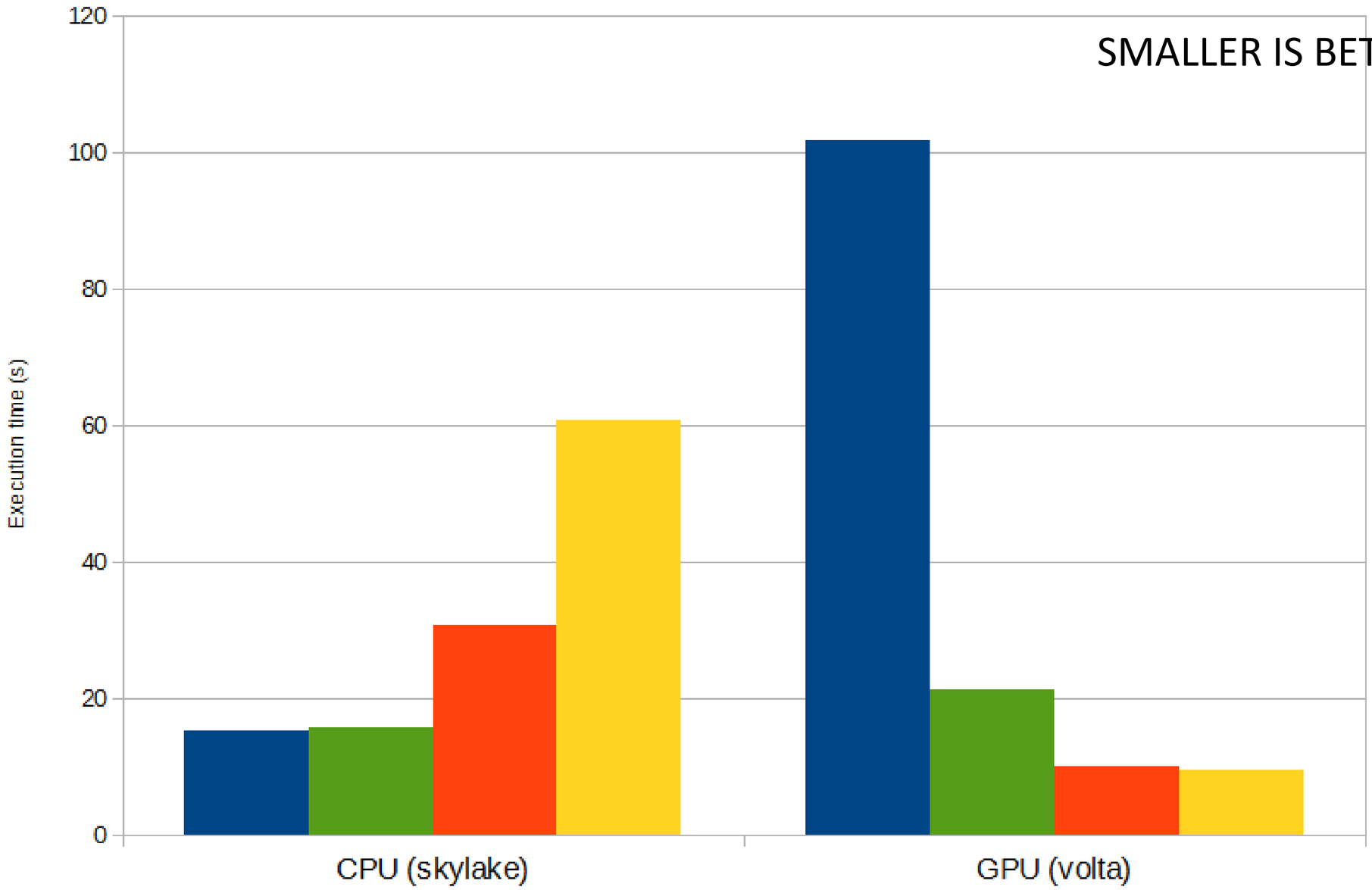
- **very large inner loop, small outer loop**
  - the inner loop can use many warps of 32 cores
  - outer loops can be distributed over the SMP

- Far better potential usage of GPUs **WITH LARGE DATASETS**

- **+ broad benefit for vector operations (SIMD, …)**

But First … a more vector friendly structure…

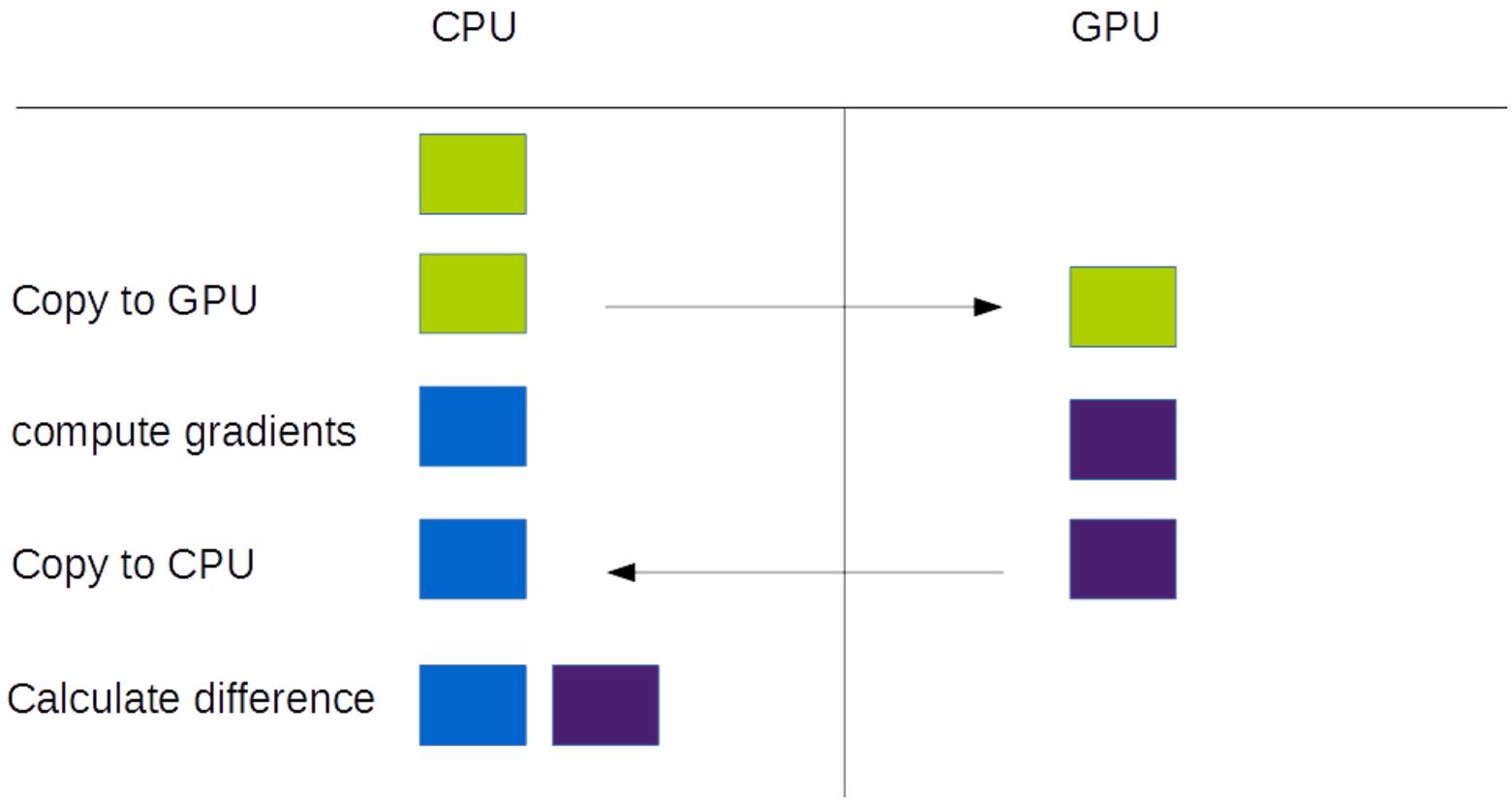
- Performance of the gradients computation for the SIMPLE case per cell group size



Side effect : Better vector performance on x86 as well (1.7x) but with « small groups »

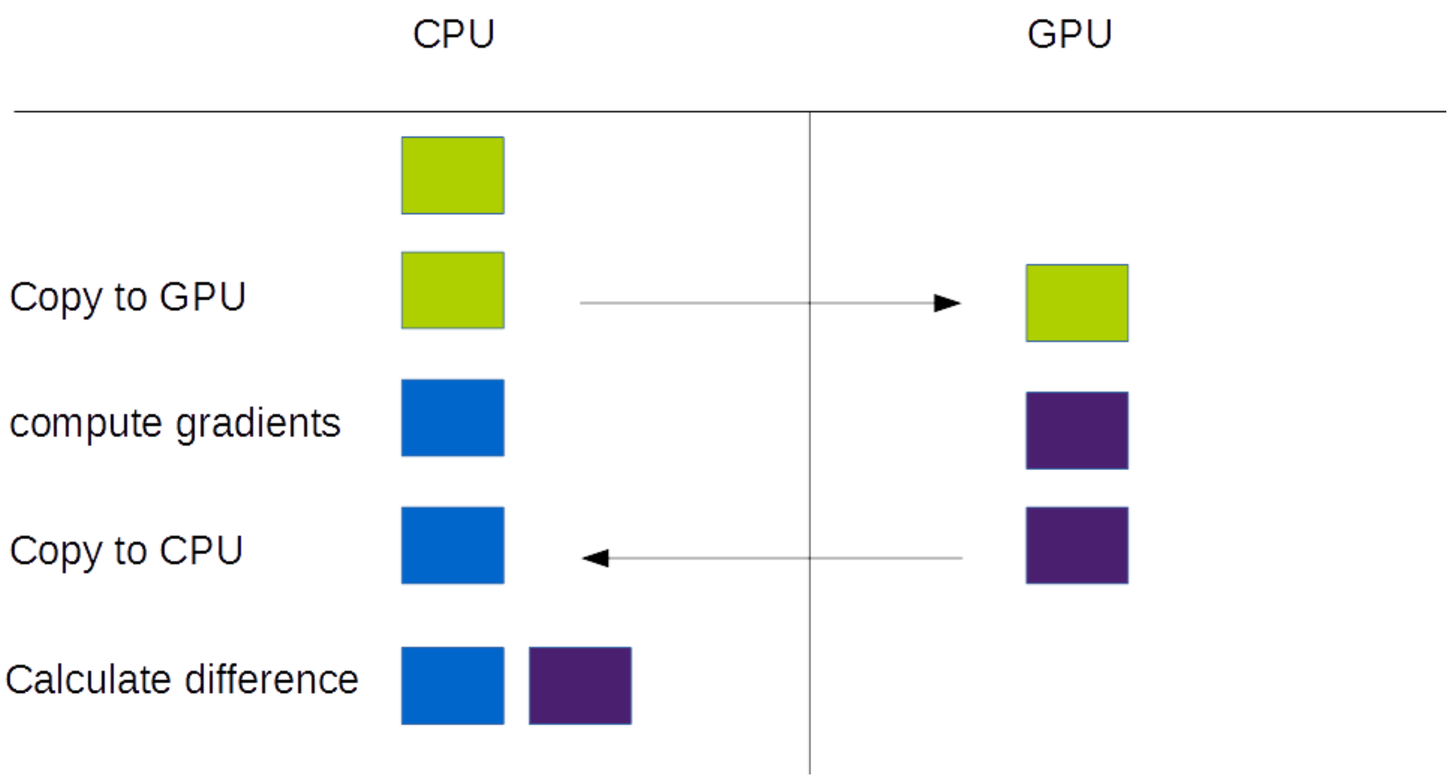
Validating results

- Different architecture and different parallelism.
- Order of operations can not be guaranteed



- Simulation outputs : not a direct access to computed arrays
- Duplicate computations, on-the-fly comparisons
- **PCAST — PGI Compiler Assisted Software Testing** can now automate this

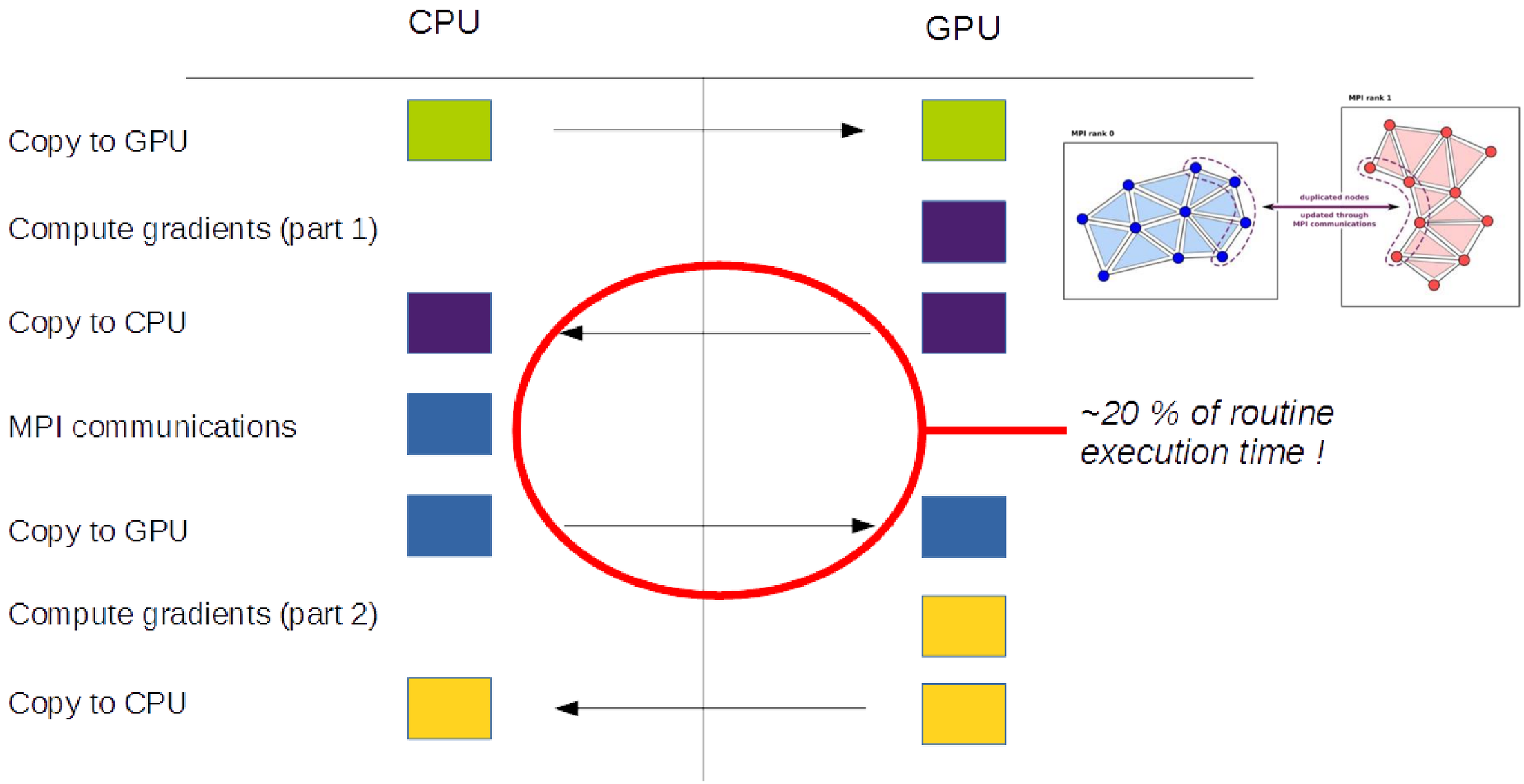
Validating results



- **Most cases (80% are strictly identical), the rest are between 10^-11 and 10^-23 (results from V100).**
- Most errors seem to be cumulative or different operation approximations.
- Overall behavior is currently acceptable for large scale simulations.

Kernel execution

- MPI in the code is negligible but structure requires synchronisations between partitions : copies ..



Handling MPI

- Direct MPI calls on GPU using cuda-aware implementation: available on most recent MPI libraries (OpenMPI, MVAPICH,IBM Spectrum)

```
!$ACC HOST_DATA USE_DEVICE(tmp_buf_recv)
CALL MPI_Irecv(tmp_buf_recv(ofs),cnt,mpi_real_type,rank,tag,&
                    comm,mpi_reqs(i),ierr)
!$ACC END HOST_DATA
```

- However, we still need to handle the MPI buffers construction/manipulation

Handling MPI

Transfers use unstructured arrays

```
lp = 1
dp = 1
DO i=1,runlist_cnt
  cnt = runlist(i)
  SELECT CASE(cnt)
  CASE(1)
    DO j=1,list_length
    lid = indices(lp)
    ofs(1)=(dep_data(dp)…)
    DO k=1,neq
         field_ptr(k,lid)=
            recv_buf(ofs(1)+k)
    END DO
      dp=dp+2
      lp=lp+1
  END DO
CASE(2)
...
  dp=dp+4
...
```

- Routines that build/extract data to and from buffers used in MPI communications
- **Fundamentally bad for GPU parallelism**
  - Variable execution path
  - Counter variables that get unpredictible increments
  - Data movements depend on the correctness and order of the counter variables increments
  - !$ACC lead to sequential execution, forcing parallelisation leads to wrong order of operations

Handling MPI

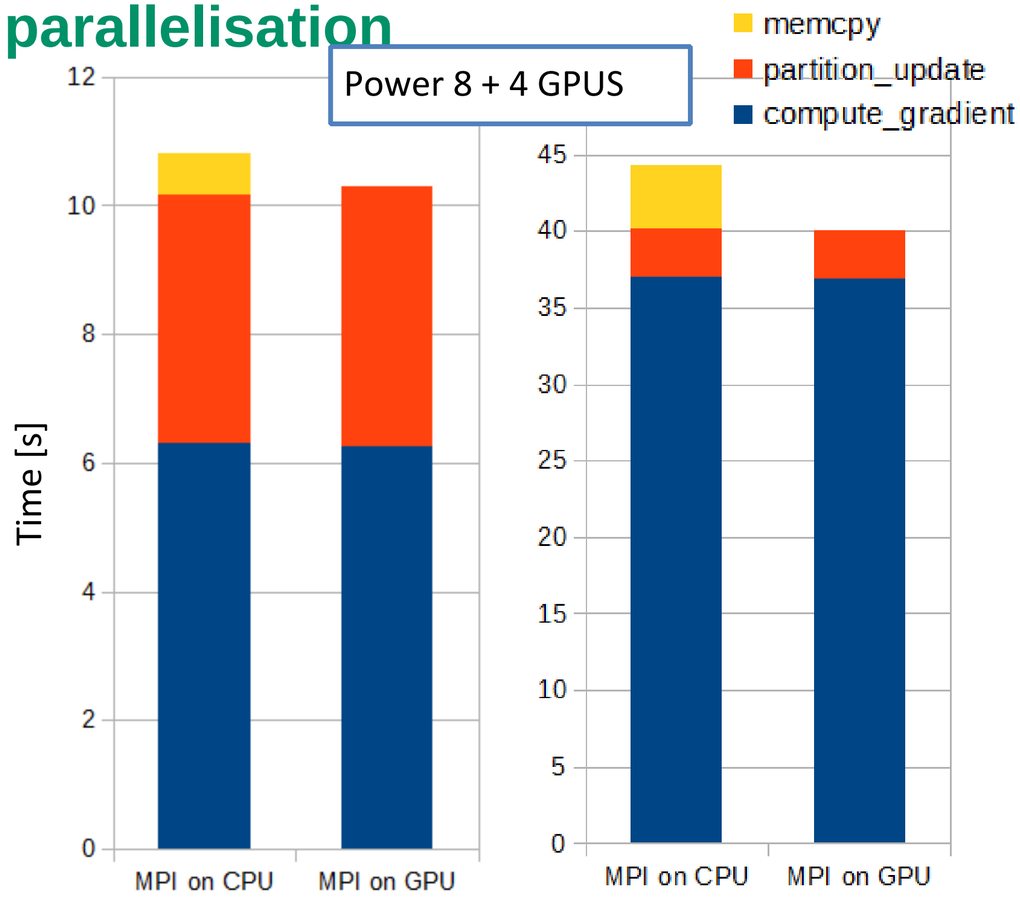
Complete rewrite of the transfer module

```
DO i=1,runlist_cnt
  lp(i) =…     dp(i)= …
DO i=1,runlist_cnt
  cnt = runlist_depcnt(i)
  !$ACC LOOP PRIVATE (dp, lp, s, ofs) VECTOR(128)
  DO j=1,runlist_length(i)
    dp = runlist_dp(i) + (j-1) * cnt * 2
    lp = …
    DO l = 0, cnt
      ofs(cnt) = dep_data(dp…)
    DO k=1,neq
      DO l=1,cnt
        s(k) = s(k) + recv_buf(ofs(l) + k)
DO k=1,neq
  !$ACC ATOMIC UPDATE
  field_ptr(k,lid(lp))=s(k
```

➡**Precompute boundary counters**

➡ **Express the counters independently for each iteration**

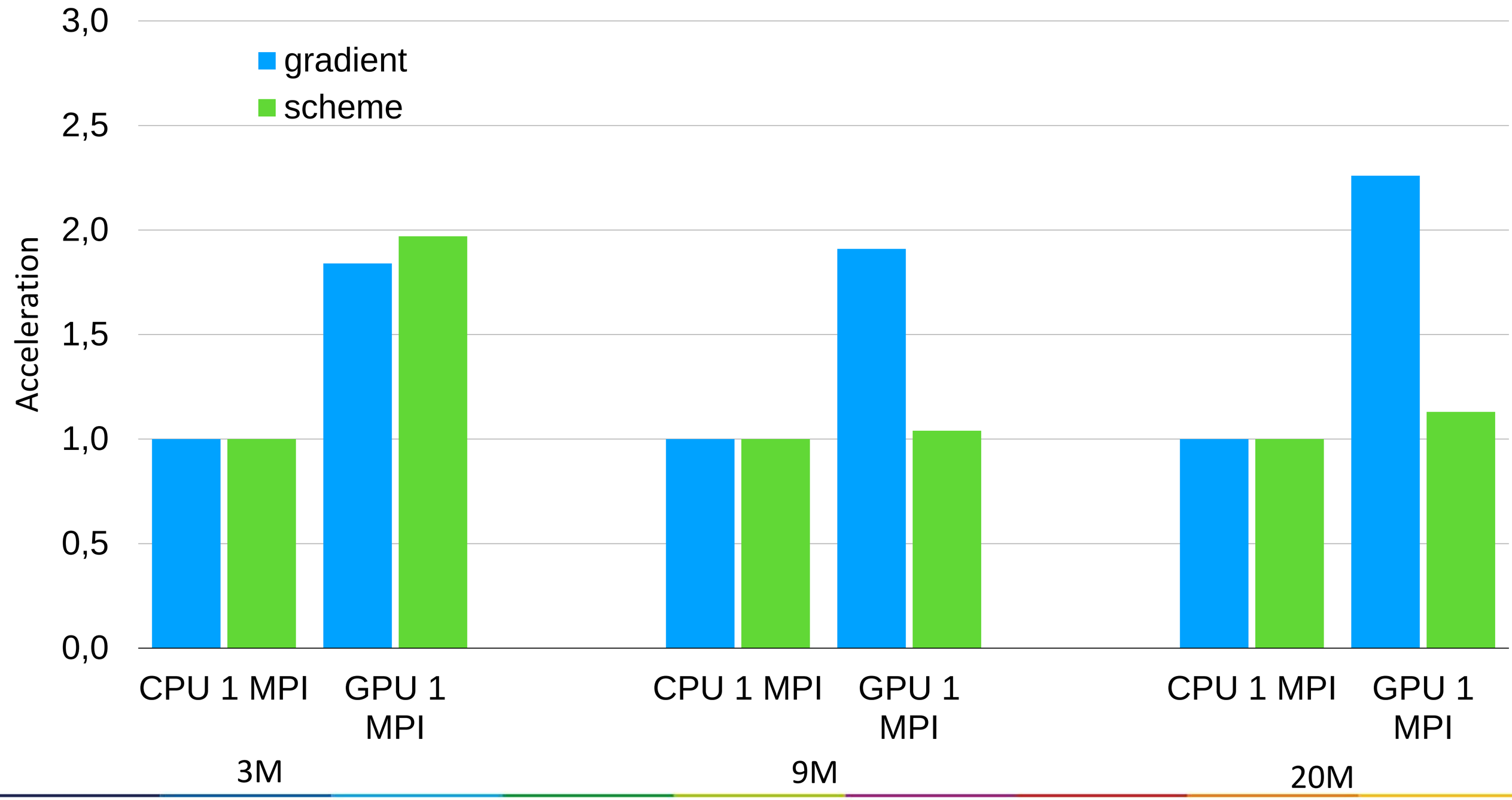
➡ **The counters can them be privatised for each iteration, allowing full parallelisation**



This is the only code that has been rewritten explicitly for GPU . Rest of the code remains identical for CPU.

Performance

1 Skylake + V100

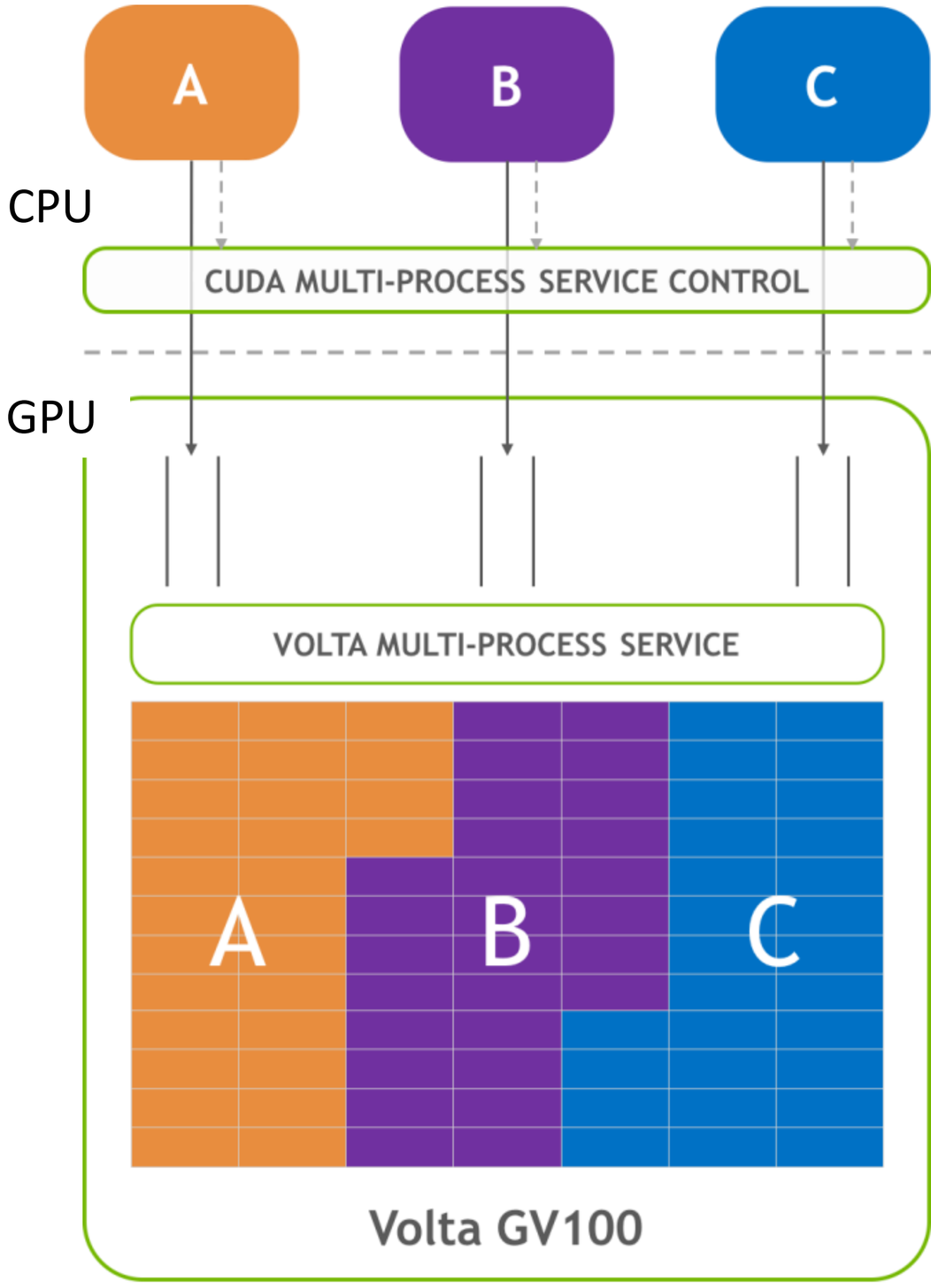


GPU faster than CPU … but not everywhere … Low occupancy of GPU < 30%/50%

MPI for performance

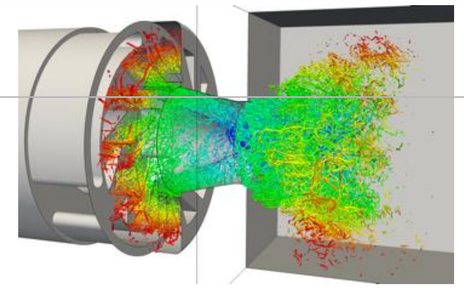
- 1 MPI per GPU not efficient.
- Occupancy below 50%
- Multi-Process Service (MPS) allows for multiple concurrent MPIs on GPU:
  - ✓ Share the ressources
  - ✓ Split of the workload
  - ✓ Computation / communication overlap

- AVBP already suited for multi MPIs



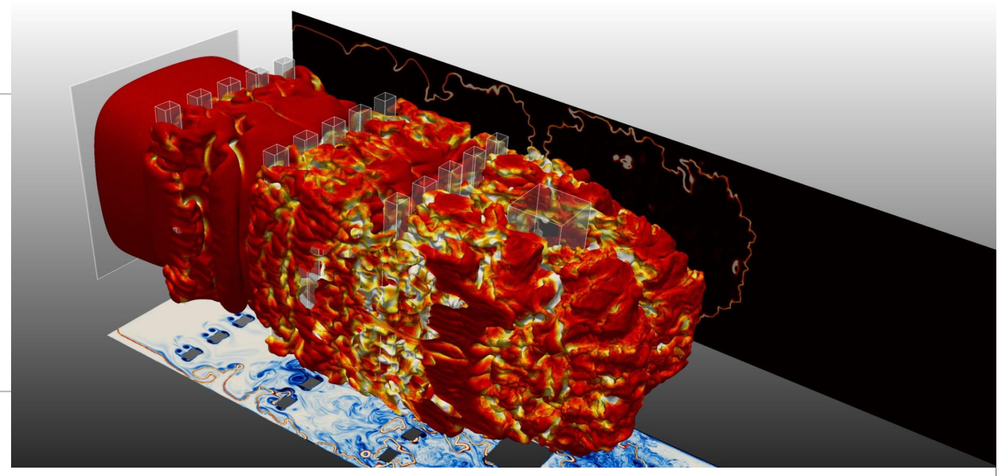
Acceleration 3M cells case

1 Skylake + V100



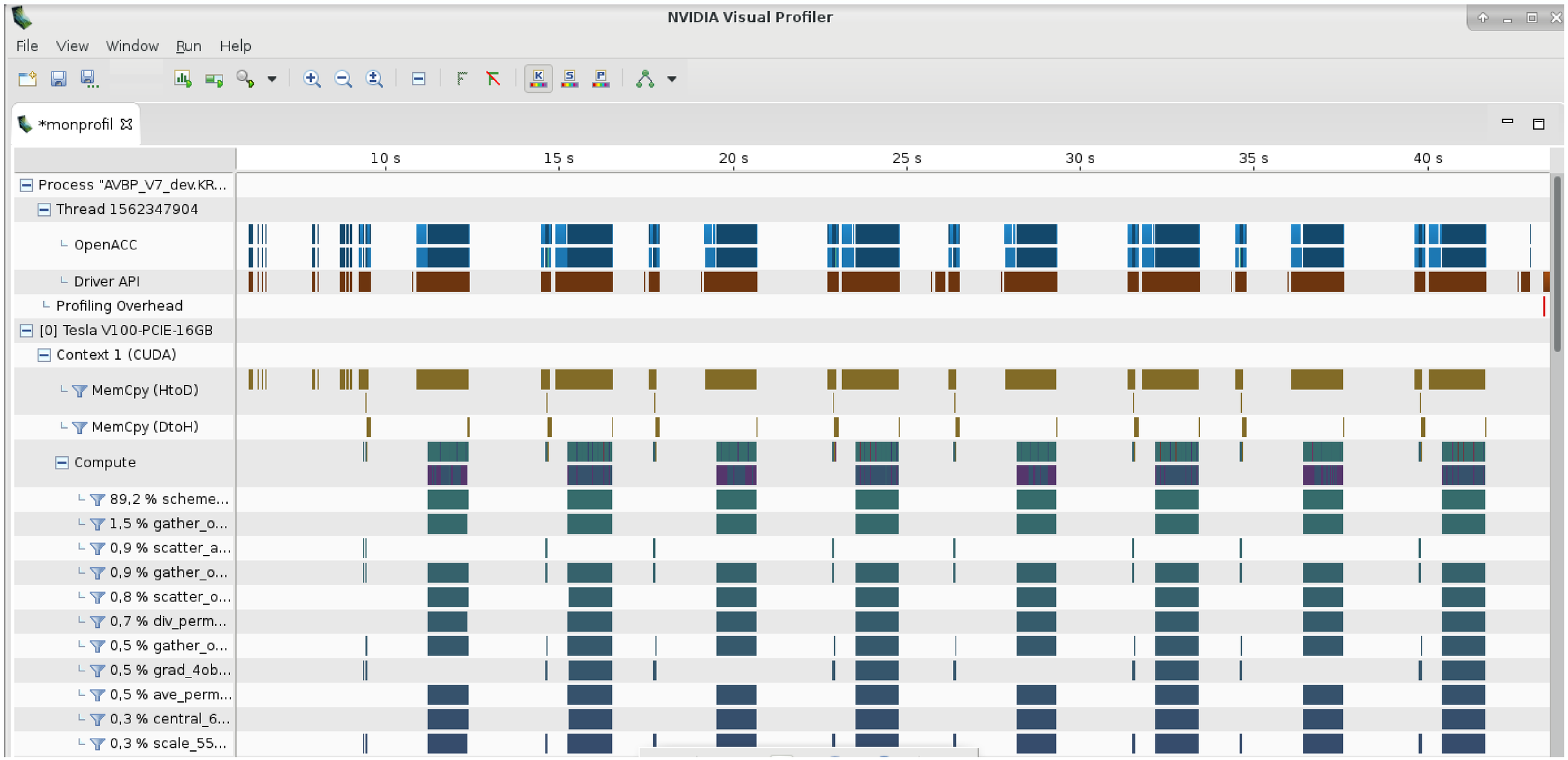
Excellent scaling for the large kernel. Limited for the « medium size one »

Acceleration 20M cells case



Not enough GPU memory for more than 8 MPI ranks (1 GPU = 16 GB)

What is happening ?



- Extensive compute capability
- Significant memcpy HtoD for scheme kernel for larger datasets.

Conclusions

- OpenACC allows for a simple but efficient porting of legacy fortran codes to GPU with almost no code duplication.

- Currently : equilibrium between 1 (full) CPU vs 1 GPU

- Kernels are faster but memory exchange are impairing
  - Better coverage should solve this

- If changes in the code are required, they are often beneficial for modern CPUs too.

Perspectives

- Full GPU port is ongoing
  - Increased coverage of iterations in order to reduce copies and increase GPU load

- Target on the Jean Zay computer
  - Collaboration Progres GENCI/IDRIS, CERFACS, HPE, Centre of Excellence EXCELLERAT

- Optimizations (asynchronous kernels, nested structure instead of MPS, …)

- OpenMP 5 ? when widely available

THANK YOU