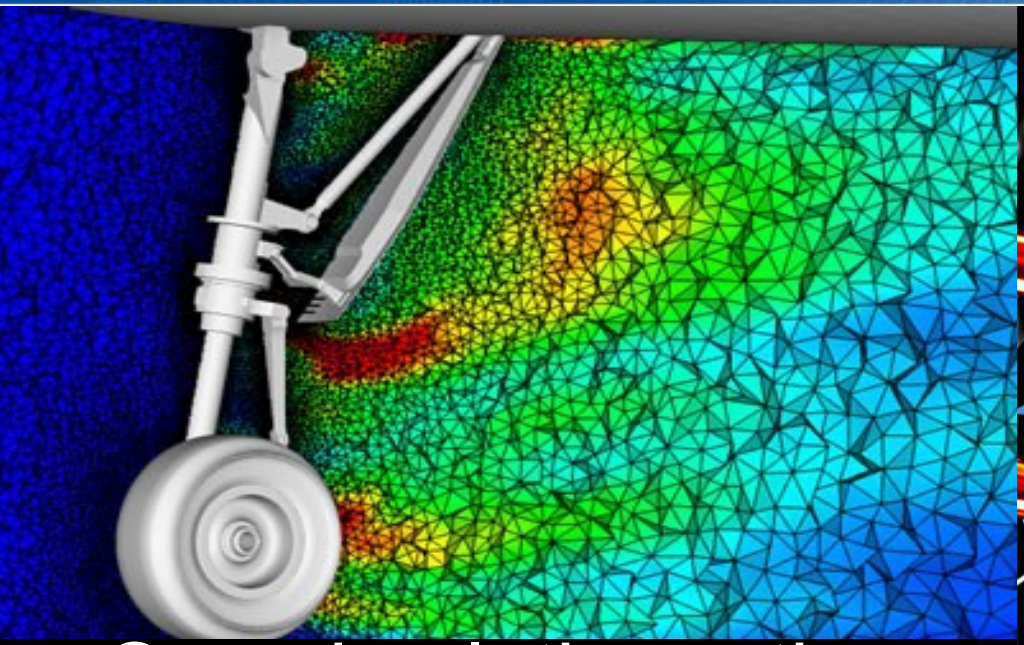
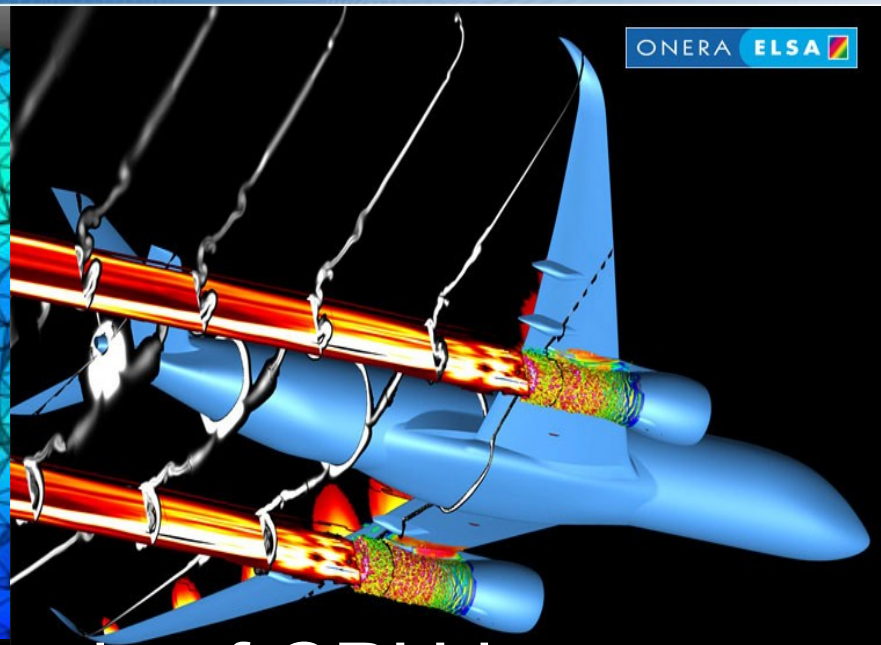
A Simple Guideline for Code Optimizations on Modern Architectures with OpenACC and CUDA

L. Oteski, G. Colin de Verdière,
S. Contassot-Vivier, S. Vialle, J. Ryan

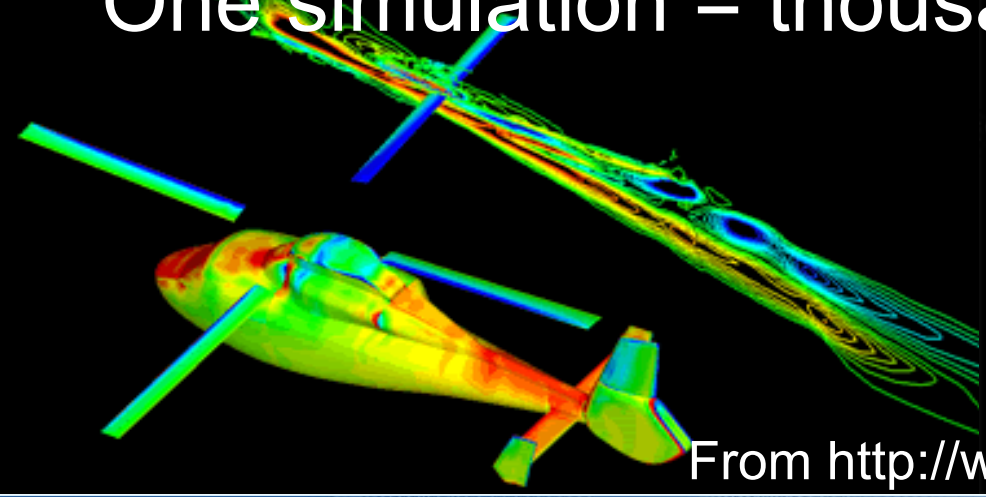
Acks.: CEA/DIFF, IDRIS, GENCI, NVIDIA, Région Lorraine.

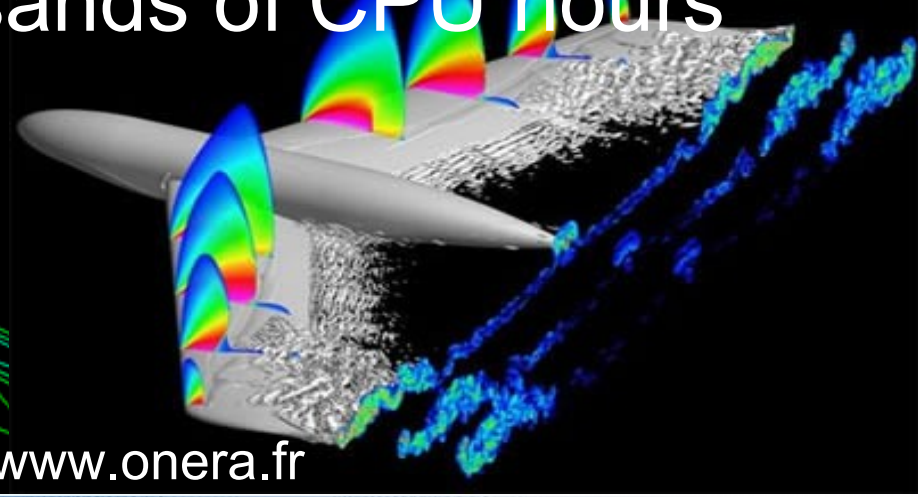
CFD at ONERA





One simulation = thousands of CPU hours





From http://www.onera.fr

Why optimize?

Cost of a CPU hour ~$0.05→$0.10 [Walker (2009)].

Example: 1000 cores for 1 month ~$36,000→$72,000.

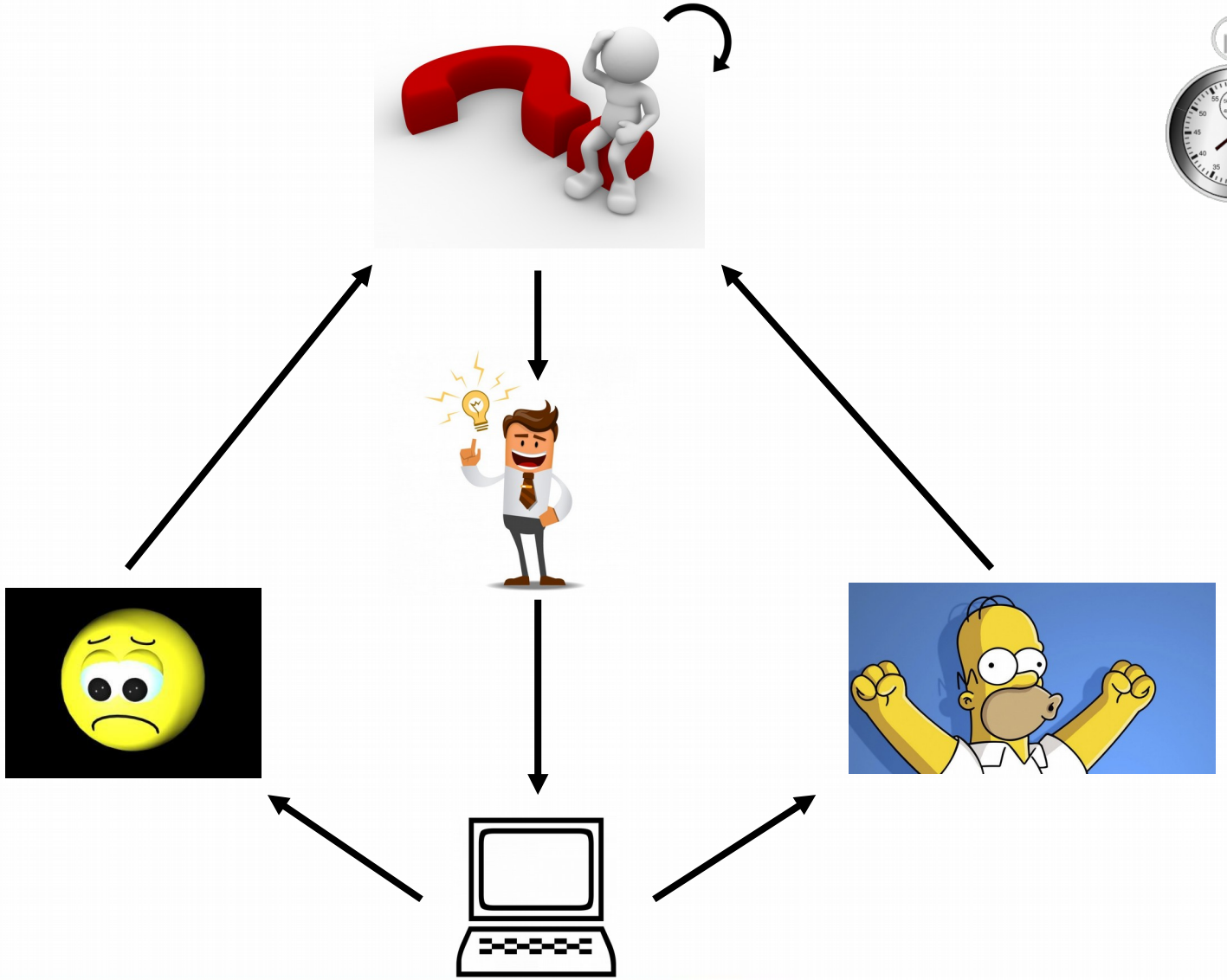
Dividing application runtime **by 2** allows to:

→be **twice** cost-efficient,

→get **faster** results,

→**increase** the problem size **by 2**.

Optimization process



CPU-GPU differences

CPU: few # of cores, vector machines, hard to manage registers,

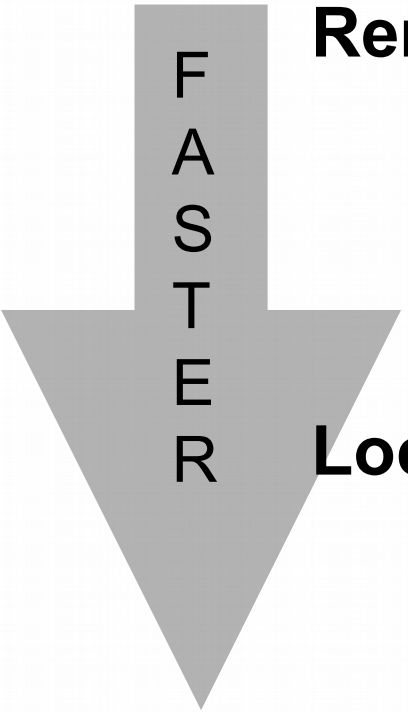
GPU: huge # of cores, vector machines, easy to manage registers (--ptxas-options=-v with nvcc).

Question:

Due to vector approaches, could GPU and CPU be considered as similar devices?

→ Could the GPU be the ideal development device?

GPU Memories



Remote memories:

Host (CPU) ~8→160 GB/s

Global Memory ~250→720 GB/s

Local memories:

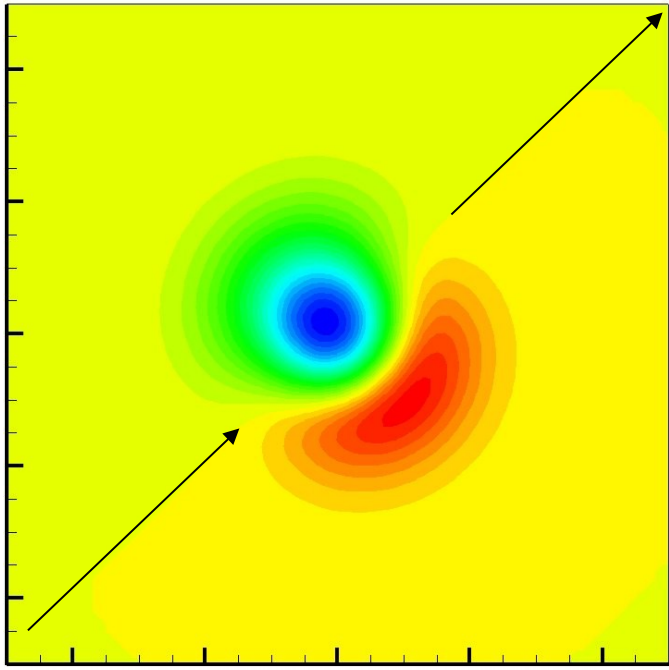
Shared, constant, texture ≥ 1 TB/s

Registers ≥ 10 TB/s

→ **4 orders of magnitude**

A prototype flow

Yee's vortex with the full Navier-Stokes equations,



2 dimensional structured mesh,

Numerical method:

→ **Space:** 2nd order Discontinuous Galerkin,

→ **Time:** 3rd order TVD Runge-Kutta,

→ **Boundaries:** X and Y-periodic.

Test case configuration

2001x2001 mesh, 100 time-steps.

CPU: Intel Xeon E5-1650v3 (6 cores),

Compiler:

icc 2016, -O3 -march=native

GPU: K20Xm (2688 CUDA cores)

Compilers:

nvcc CUDA-8.0, -O3

pgc++ 16.10, -O3 -acc
-ta=tesla:cuda8.0,nollvm,nordc

Parallel paradigm

CPU:

OpenMP + vectorization pragmas,

→ each thread: one part of the mesh.

GPU:

full CUDA,

full OpenACC,

→ each CUDA block: one part of the mesh.

Experimental protocol

Developments with the full CUDA version

GPU-CUDA

```
int tidx=threadIdx.x+...;
int tidy=threadIdx.y+...;
if(tidx<Nx && tidy<Ny)
{
    [...]
}
```

Copy

CPU-OpenMP

```
#pragma omp for
for(int tidy=0;tidy<Ny;tidy++)
{
#pragma simd
    for(int tidx=0;tidx<Nx;tidx++)
    {
        [...]
    }
}
```

Paste

Experimental protocol

Developments with the full CUDA version

GPU-CUDA

```
int tidx=threadIdx.x+...;
int tidy=threadIdx.y+...;
if(tidx<Nx && tidy<Ny)
{
    [...]
}
```

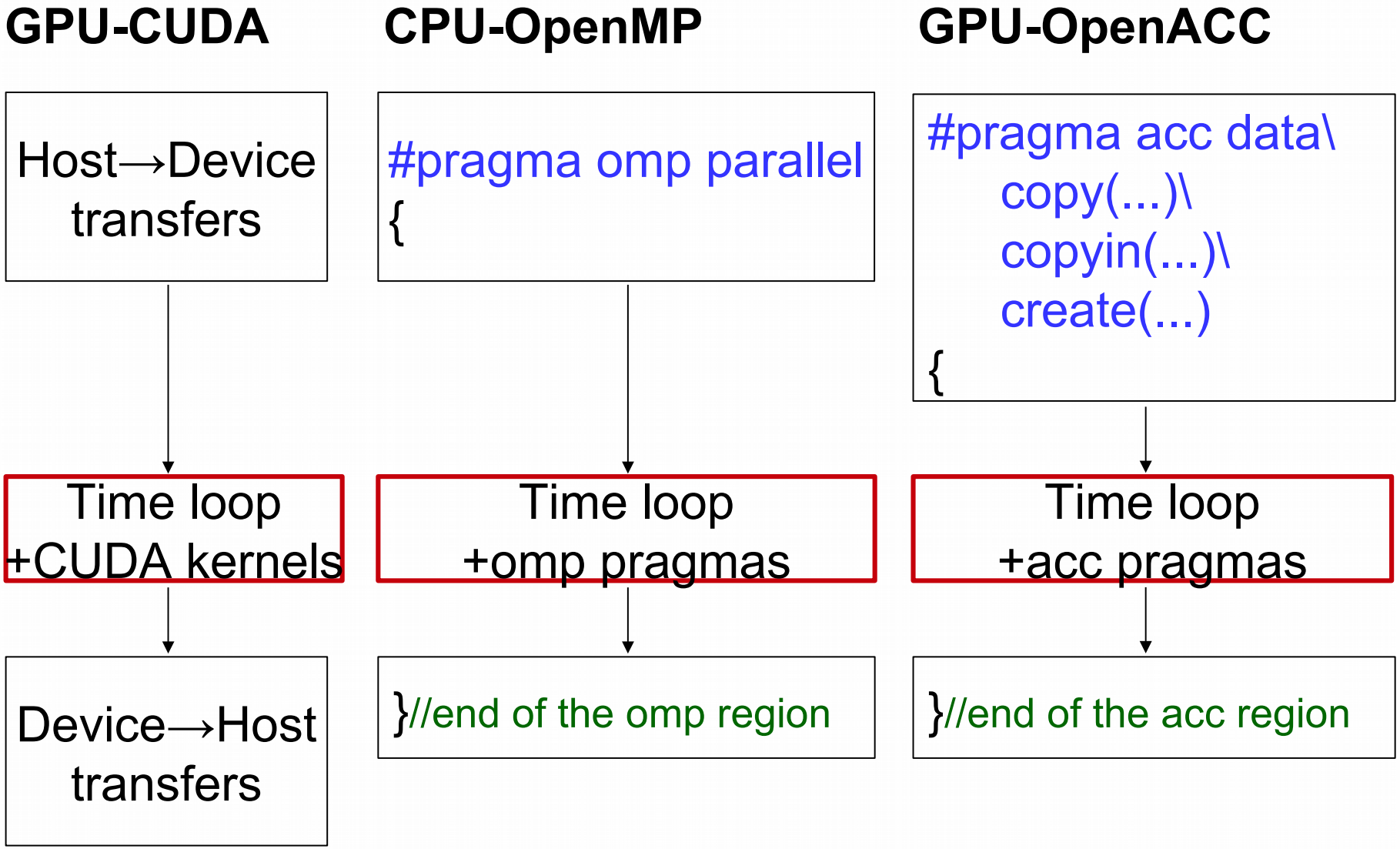
Copy

GPU-OpenAcc

```
#pragma acc parallel present(...)
{
#pragma acc loop gang independent
    for(int tidy=0;tidy<Ny;tidy++){
#pragma acc loop vector independent
        for(int tidx=0;tidx<Nx;tidx++)
        {
            [...]
        }
    }
}
```

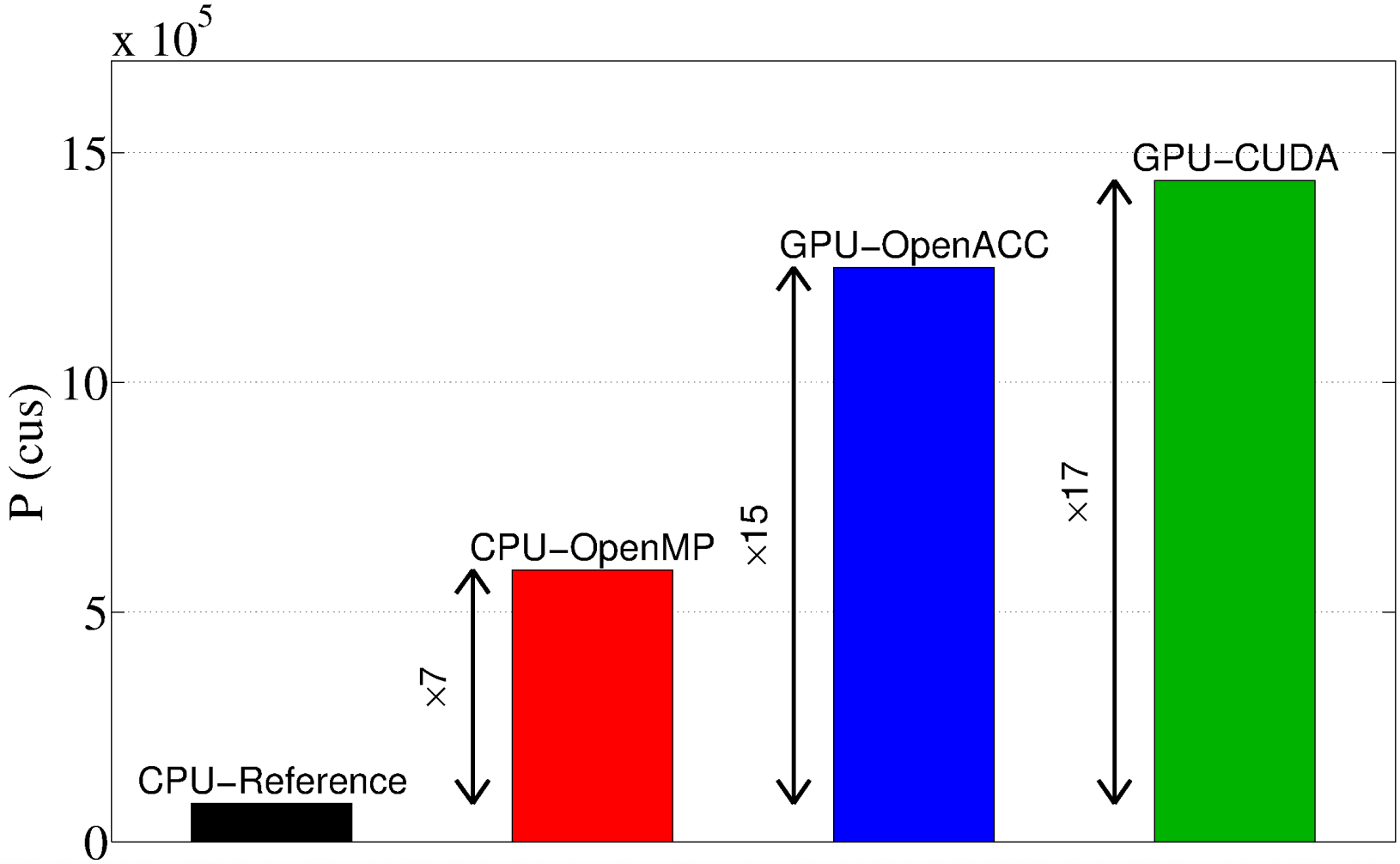
Paste

Step 0: Code adaptation



Step 0: Code adaptation

P(cell updt. per sec. [cus])=#of points x number of it. / time (sec)



Step 1: Reduce remote accesses

Load reused remote variables in registers (time locality),

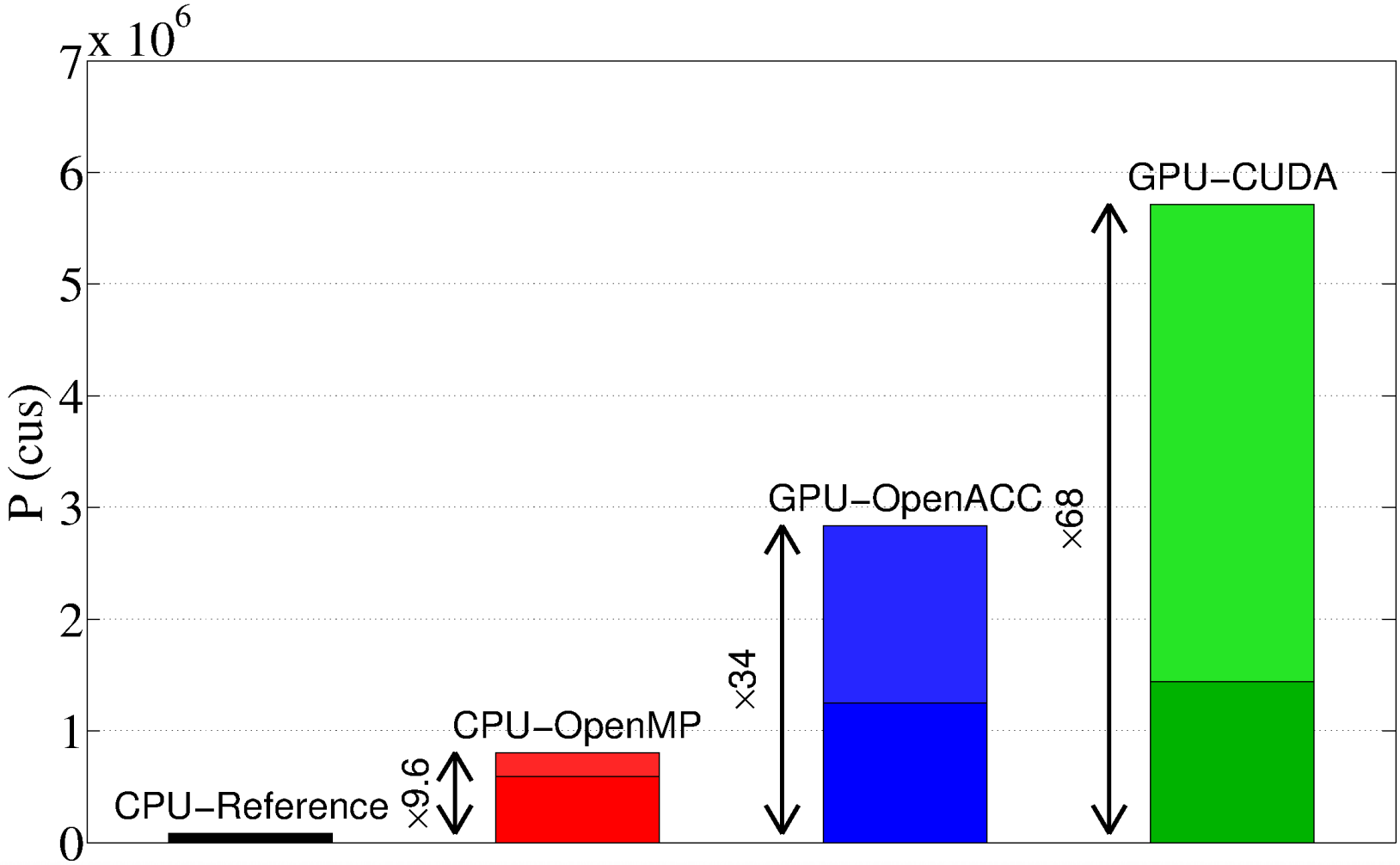
Use __restrict__ keyword on pointers.

```
__global__
void func(double *results, int N)
{
    int tidx=threadIdx.x+...;
    […]
    for(int i=0;i<N;i++)
    {
        […]
        results[tidx]+=...
    }
}
```

```
__global__
void func(double *__restrict__ results, int N)
{
    int tidx=threadIdx.x+...;
    […]
    double loc_result=results[tidx];
    for(int i=0;i<N;i++)
    {
        […]
        loc_result+=...
    }
    results[tidx]=loc_result;
}
```

Step 1: Reduce remote accesses

P(cell updt. per sec. [cus])=#of points x number of it. / time (sec)



Step 2: Merge kernels

Merge kernels which share similar memory patterns.

1. Compute the time-step

2. For s Runge-Kutta step:

   a. Convective fluxes in X,
   b. Convective fluxes in Y,
   c. Convective integral,

   d. Compute local viscosity,
   e. Viscous fluxes in X,
   f . Viscous fluxes in Y,
   g. Viscous integral,

   h. Runge-Kutta propagation.

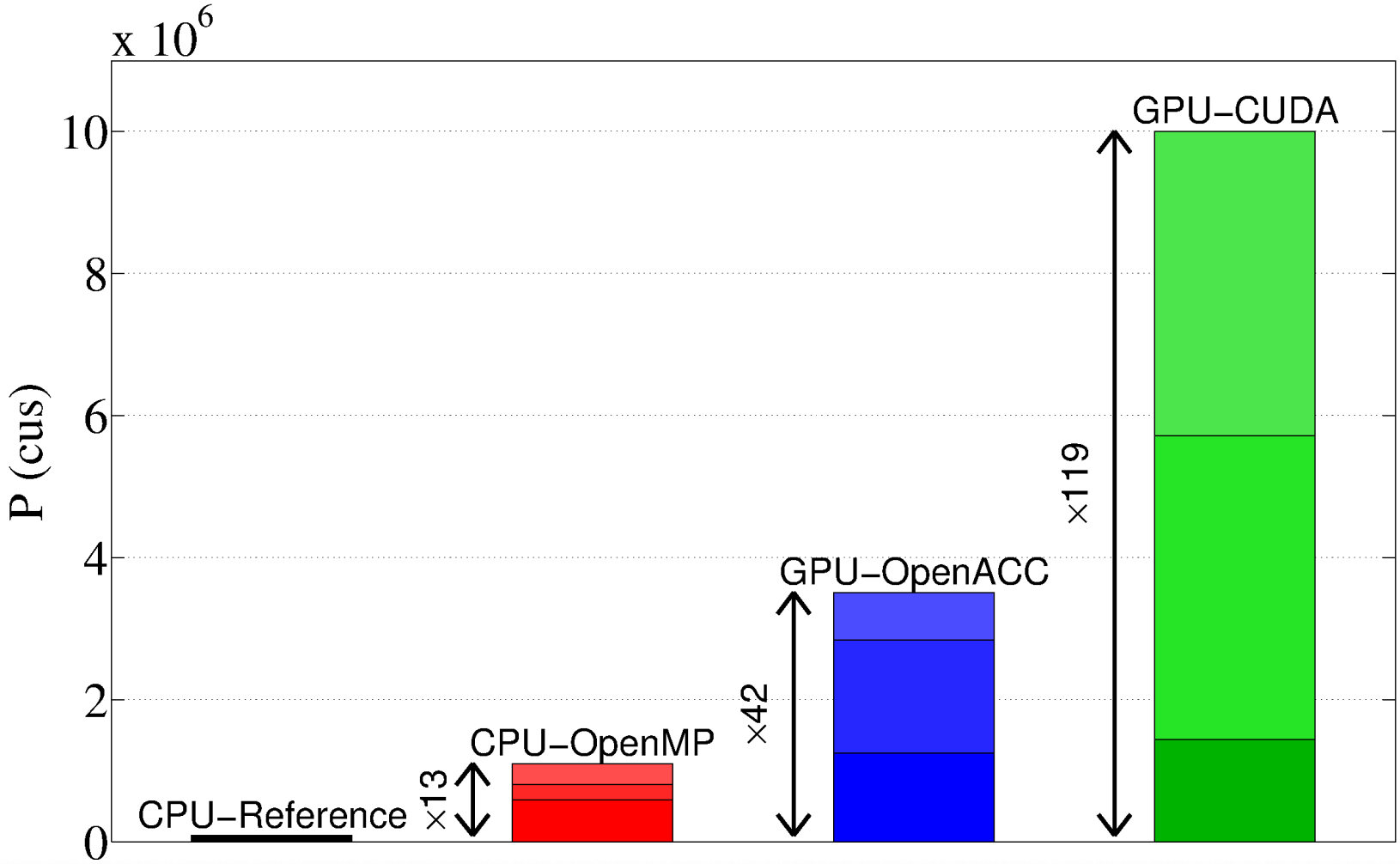
1. Compute the time-step

2. For s Runge-Kutta step:

   a. Compute local viscosity,
   b. Fluxes in X,
   c. Fluxes in Y,
   d. Integral+Runge-Kutta.

Step 2: Merge kernels

P(cell updt. per sec. [cus])=#of points x number of it. / time (sec)



Step 3: Factor computations

Transform divisions into multiplications,

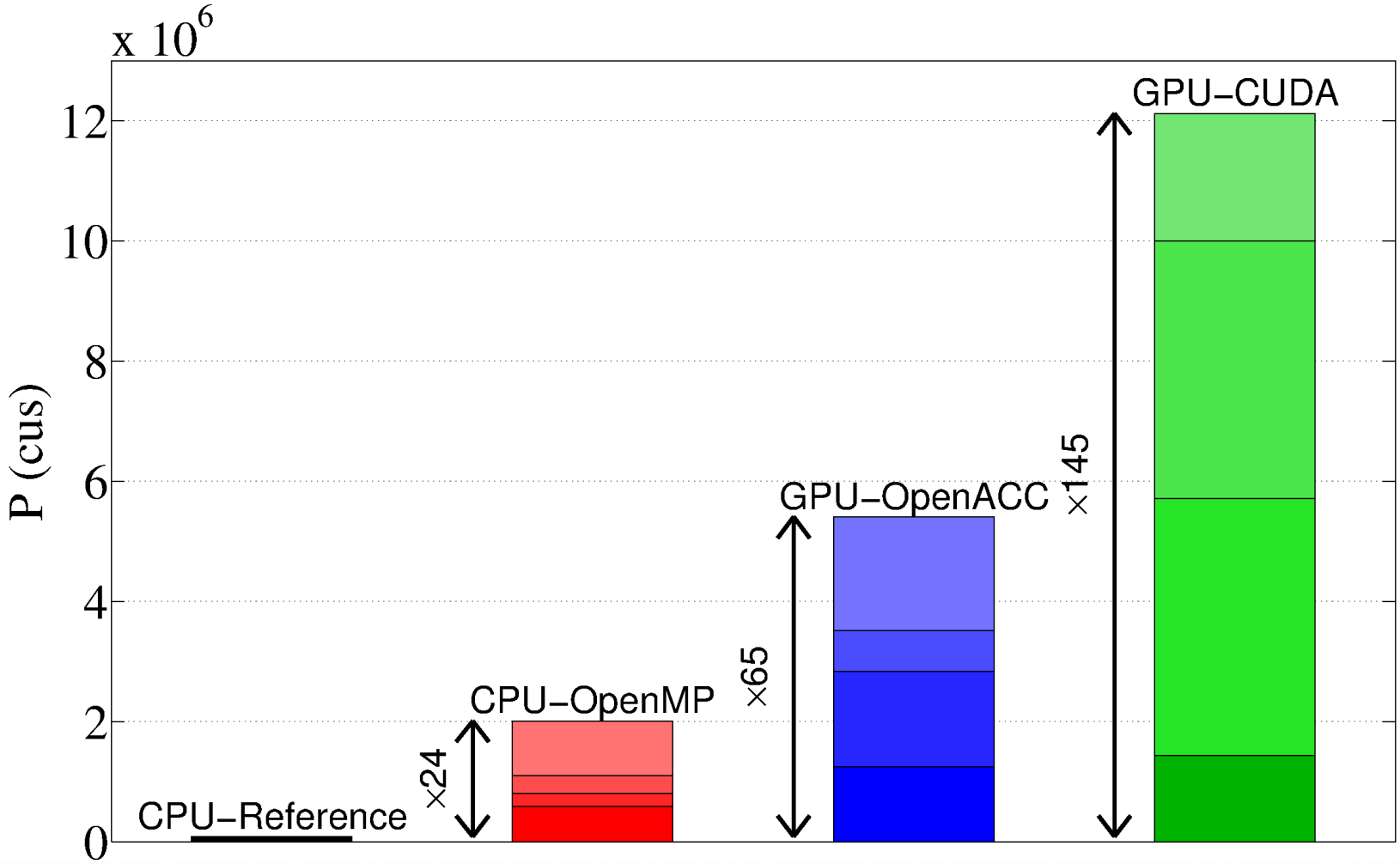
Control whether or not a loop should be unrolled.

```
#pragma unroll
for(int i=0;i<SIZE;i++)
{
    […]
    double val=[...]
    double c0=1/(sqrt(0.5)*val)*[...]
    double c1=1/(3*val)*[...]
    [...]
}
```

```
double isqrt0p5=1/sqrt(0.5);
double inv3=1/3;
#pragma unroll //Keep it ?
for(int i=0;i<SIZE;i++)
{
    […]
    double val=[...]
    double ival=1/val;
    double c0=isqrt0p5*ival*[...]
    double c1=inv3*ival*[...]
    [...]
}
```

Step 3: Factor computations

P(cell updt. per sec. [cus])=#of points x number of it. / time (sec)



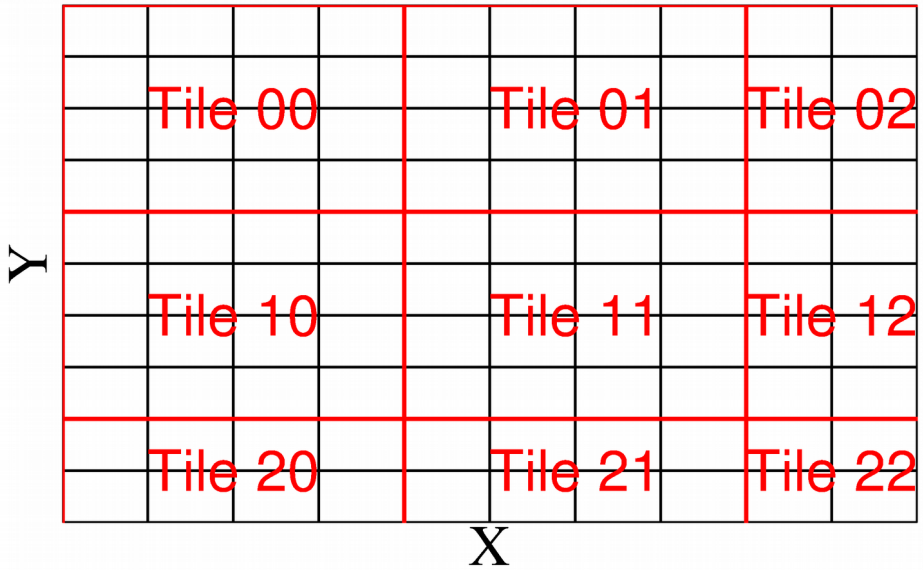
Step 4: Align data (+tiling for CPU)

GPU:

Align data on the size of a warp → coalescence.

CPU:

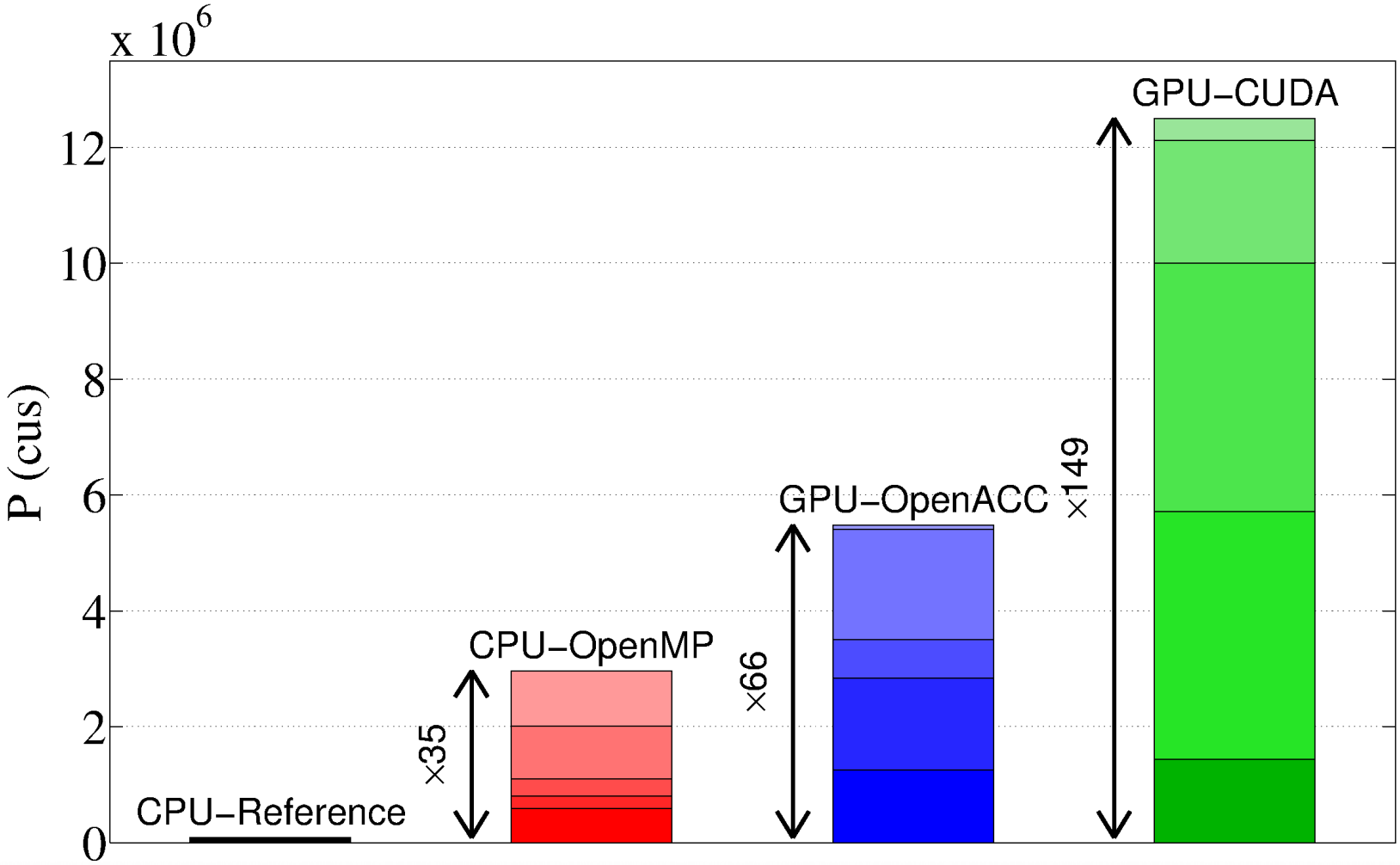
Tiling (Block) algorithm + fixed block size.



Tiling algorithm for OpenMP threads.

Step 4: Align data (+tiling for CPU)

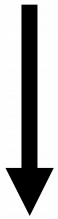
P(cell updt. per sec. [cus])=#of points x number of it. / time (sec)



Step 5: Miscellaneous

OpenACC PGI 16.10:

-O3 / O1 / O2 / O4 compilation → registers overflow.



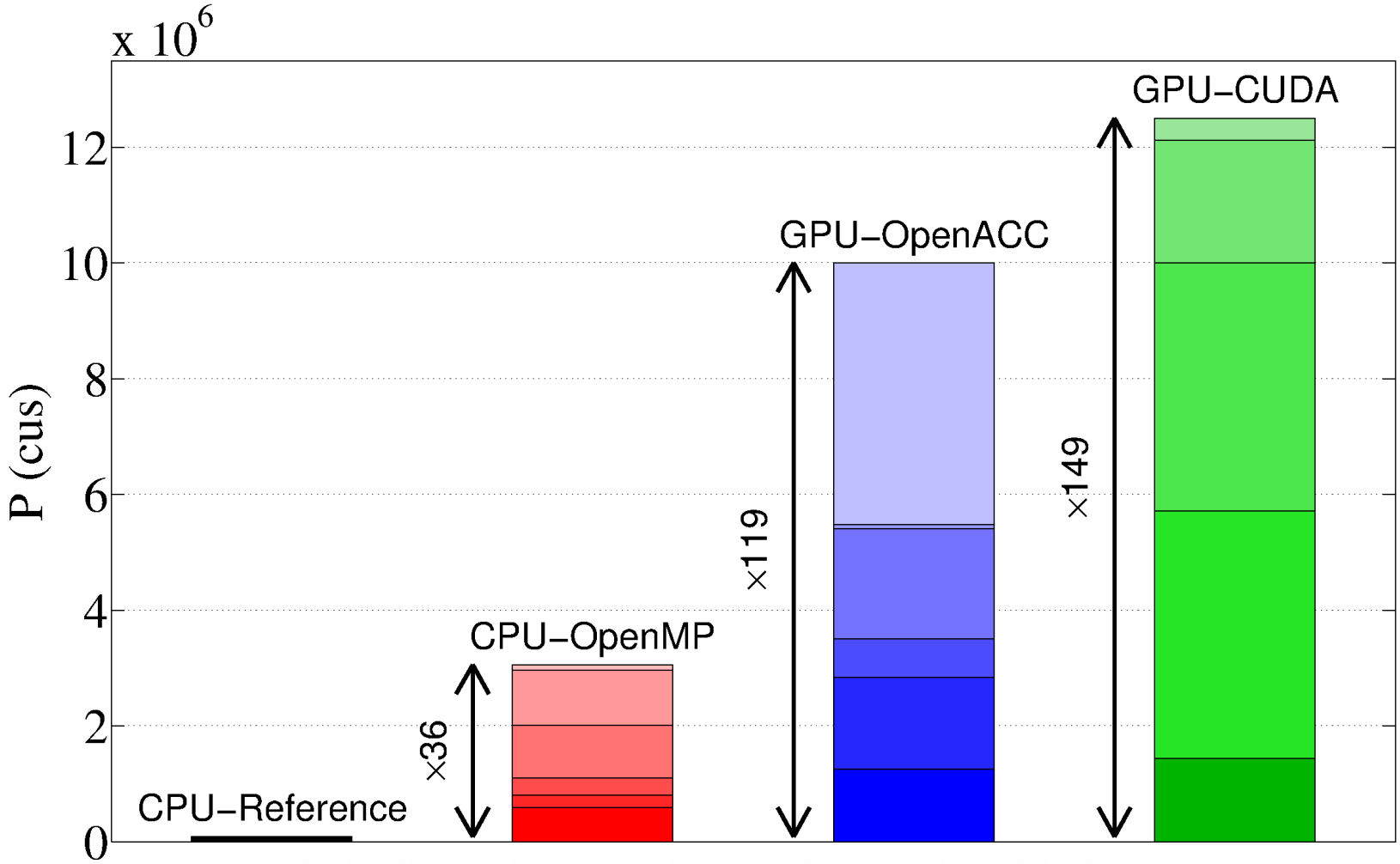
-O0 = +82% performances (corrected in v17).

CPU:

Introduce MPI to improve data locality.

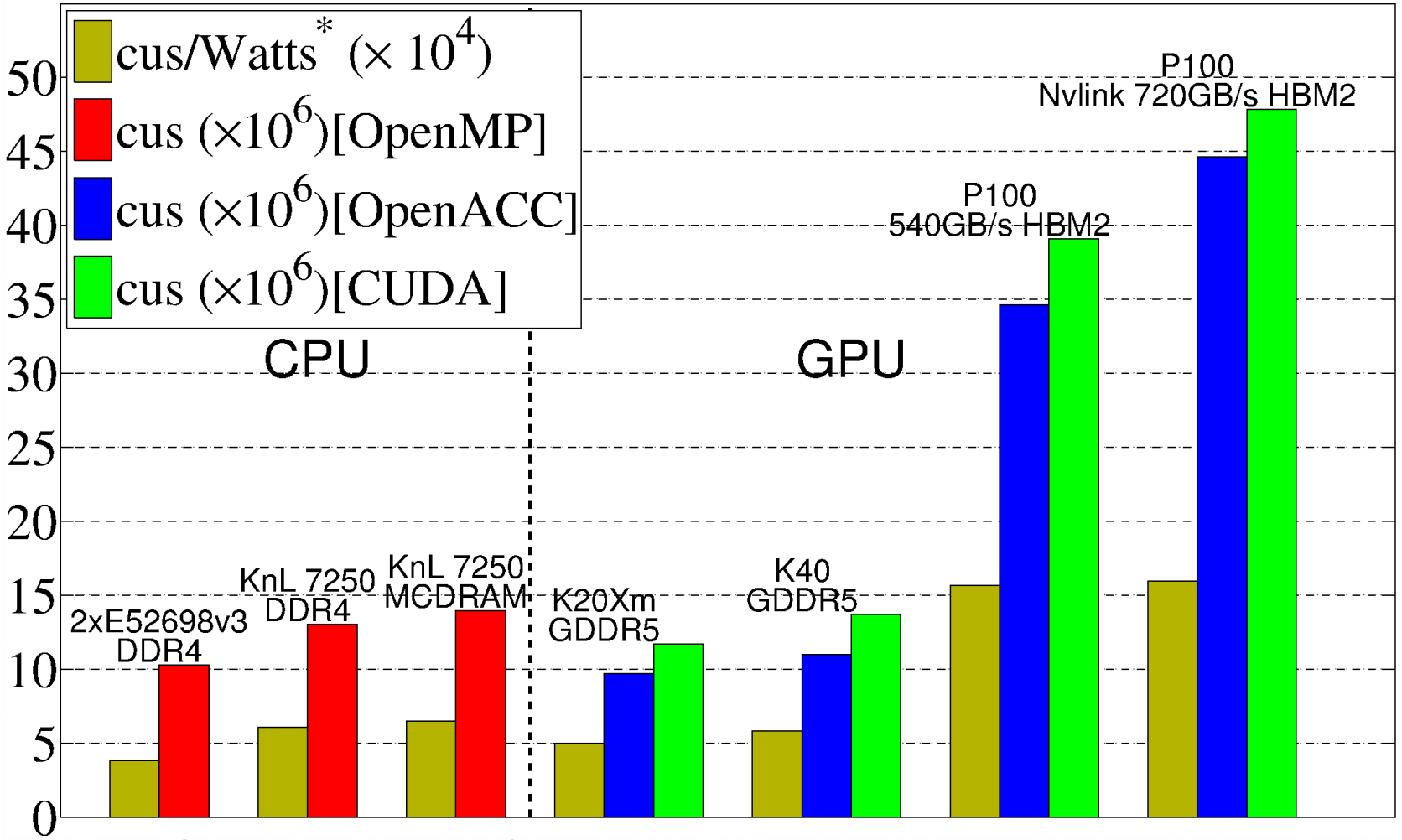
Step 5: Miscellaneous

P(cell updt. per sec. [cus])=#of points x number of it. / time (sec)



Running on various devices

P(cell updt. per sec. [cus])=#of points x number of it. / time (sec)



*TDP evaluated from constructor's documentations

Conclusion

→ **GPUs can be** used to efficiently optimize CPU codes,

→ **4 steps** to optimize:

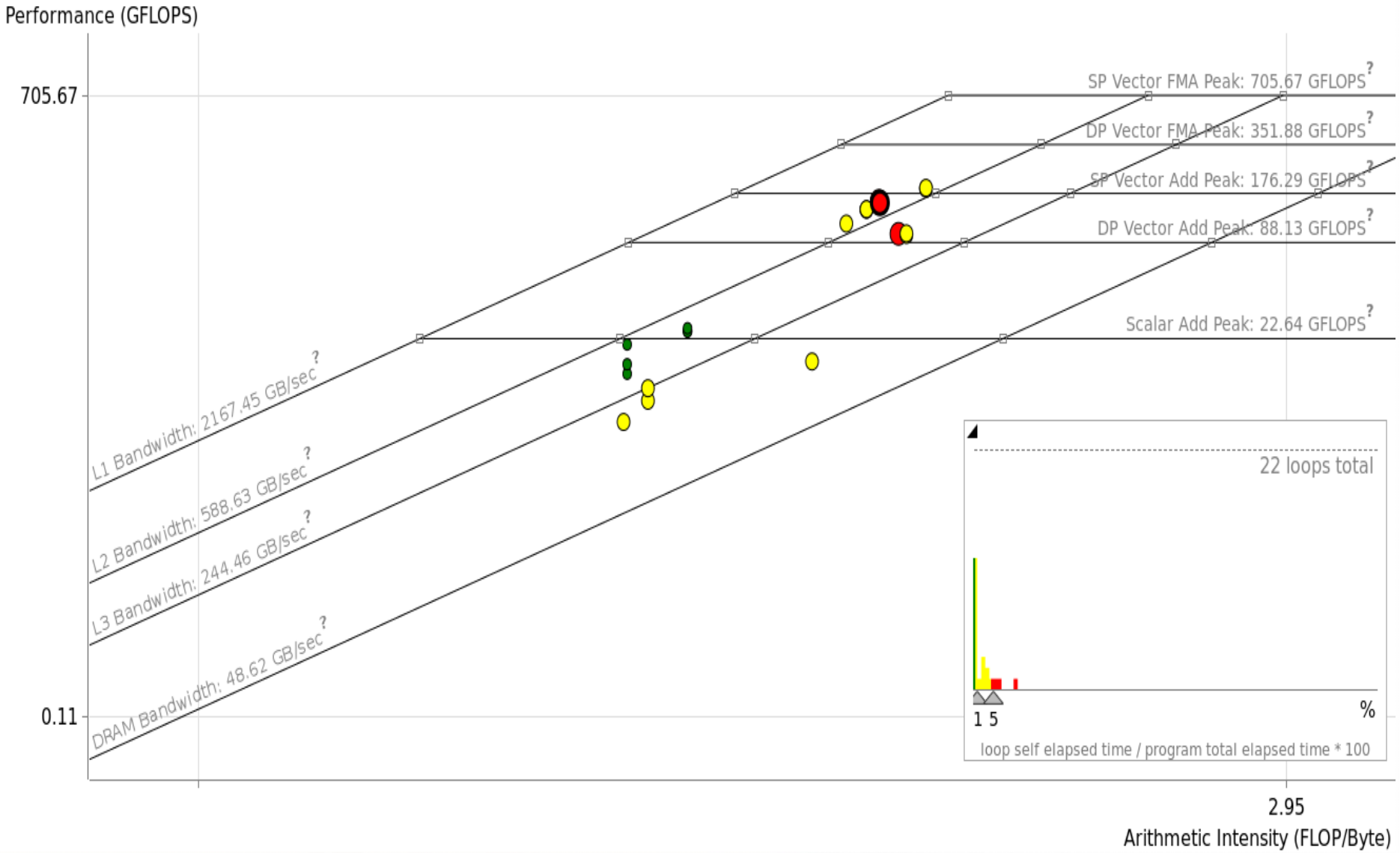
- **Reduce** access to remote memories,
- **Merge** your kernels,
- **Factor** your computations,
- **Align** your data.

To go deeper into GPU optimization: [Woolley (2013)].

Thank you for your attention.

Contact: ludomir.oteski@onera.fr

CPU Cache roofline (Intel Advisor)



Optimization steps

Ref. Case: 4785s

Modif.	CPU (6th)	Speedup	OpenACC	Speedup	CUDA	Speedup
Adaptation	677s	7.06	320s	14	278s	17.2
Reduce accesses	497s	9.63	141s	33.9	70s	68.4
Merge kernels	364s	13.1	114s	42	40s	119
Factor. Comput.	199s	24	74s	64.7	33s	145
Tiling	161s	29.7	--	--	--	--
Alignment	135s	35.4	73s	65.5	32s	149
OpenMP +MPI	131s	35.5	--	--	--	--
PGI -O0 compil.	--	--	40s	119	--	--