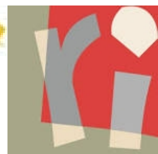
Towards an Efficient CPU-GPU Code Hybridization: a Simple Guideline for Code Optimizations on Modern Architecture with OpenACC and CUDA

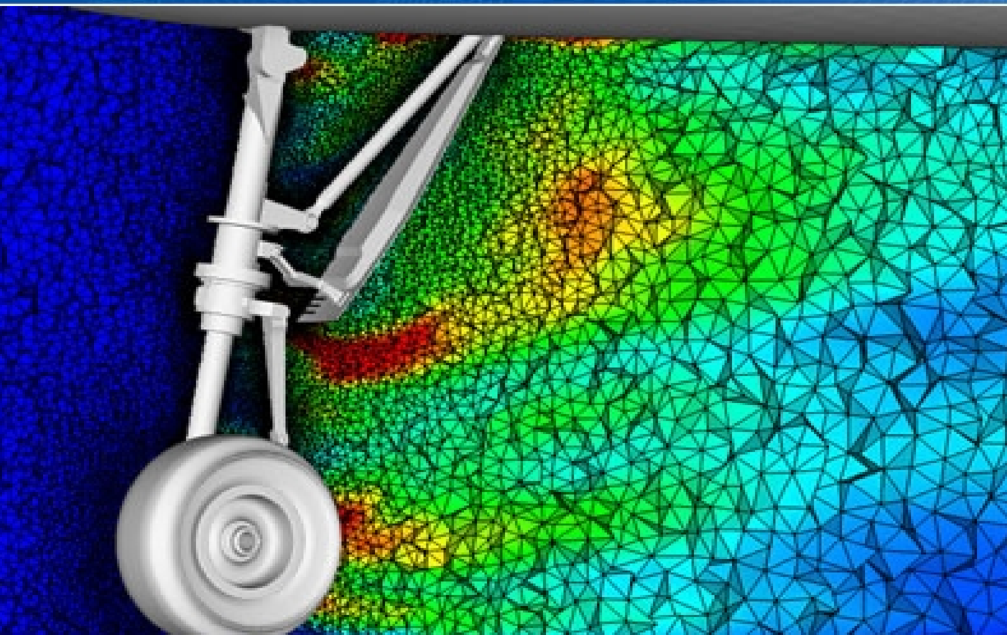
L. Oteski, G. Colin de Verdière,

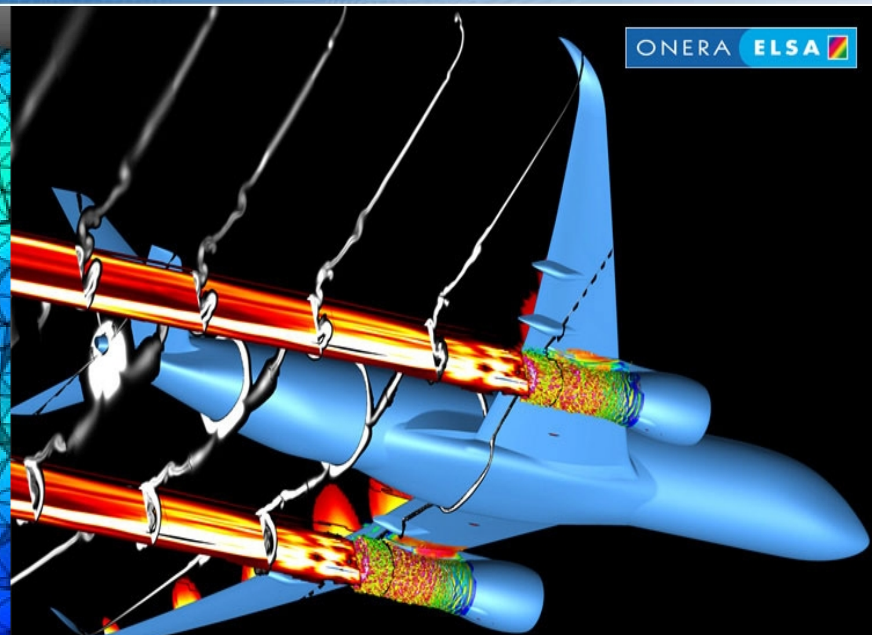
S. Contassot-Vivier, S. Vialle, J. Ryan

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

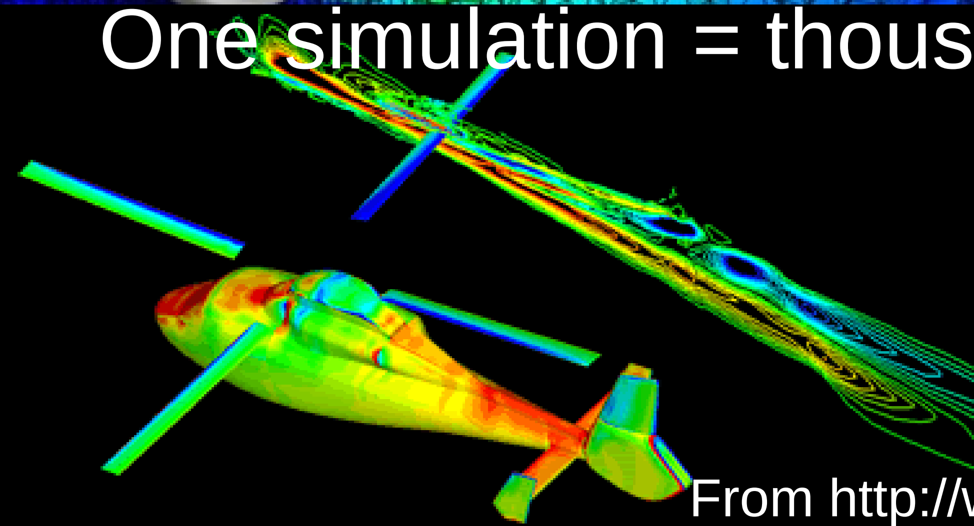


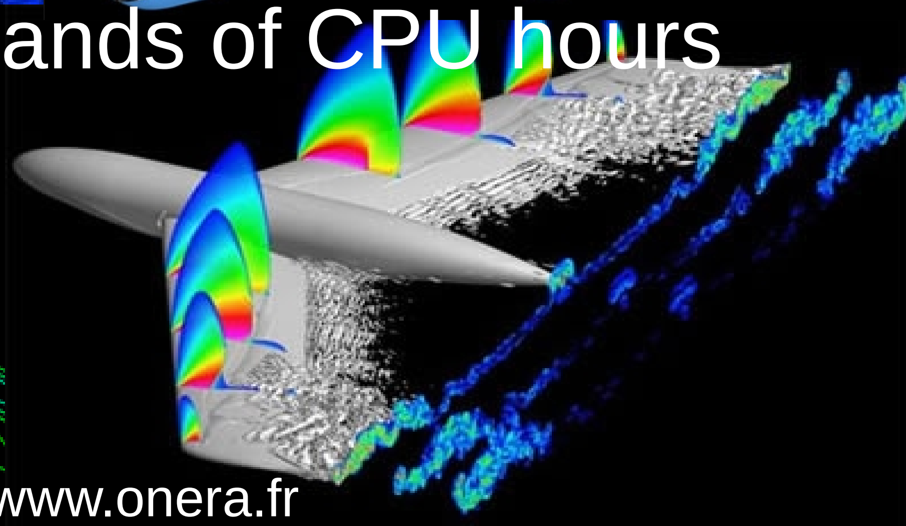
CFD at ONERA





One simulation = thousands of CPU hours





From http://www.onera.fr

GT CNRS GeorgiaTech-CNRS UMI 2958
CentraleSupélec
Loria Laboratoire lorrain de recherche en informatique et ses applications
cea
ONERA THE FRENCH AEROSPACE LAB

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 allows to:

→be more **cost-efficient**,

→get **faster** results,

→**increase** the problem size.

GT CNRS GeorgiaTech-CNRS UMI 2958
CentraleSupélec
Loria
cea
ONERA THE FRENCH AEROSPACE LAB

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 used to optimize the CPU code?

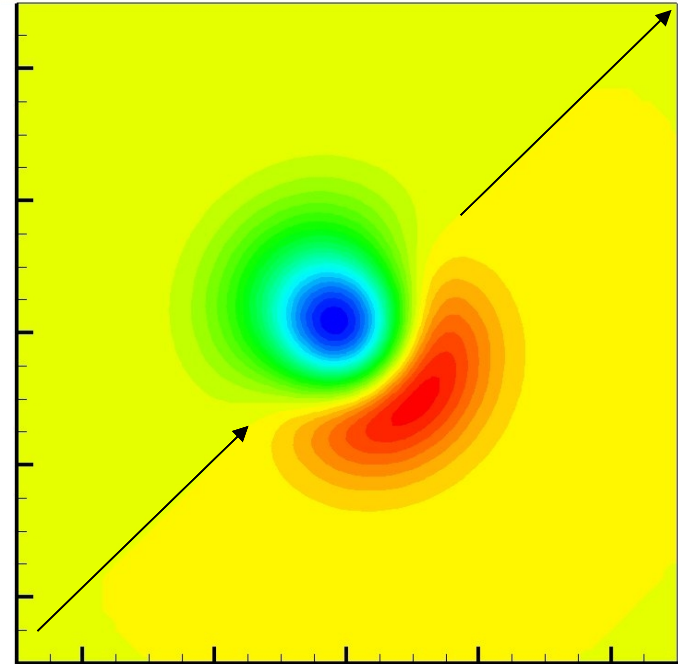
GT CNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria cea ONERA THE FRENCH AEROSPACE LAB

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.

GT CNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria cea ONERA THE FRENCH AEROSPACE LAB

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

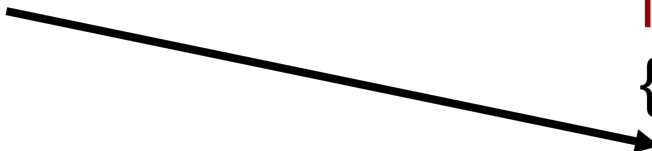
Experimental protocol

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

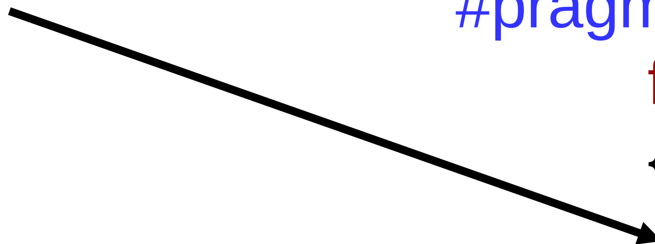
GT CNRS GeorgiaTech-CNRS UMI 2958 · CentraleSupélec · Loria · cea · ONERA THE FRENCH AEROSPACE LAB

Experimental protocol

Developments with the full CUDA version

GPU-CUDA

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



GPU-OpenACC

```
#pragma acc parallel
{
#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

GT CNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria cea ONERA THE FRENCH AEROSPACE LAB

Step 0: Code adaptation

GPU-CUDA

Host → Device transfers
cudaMemcpy(...)

Time loop
+CUDA kernels

Device → Host transfers
cudaMemcpy(...)

CPU-OpenMP

```
#pragma omp parallel
{
```

Time loop
+omp pragmas

```
}//end of the omp region
```

GPU-OpenACC

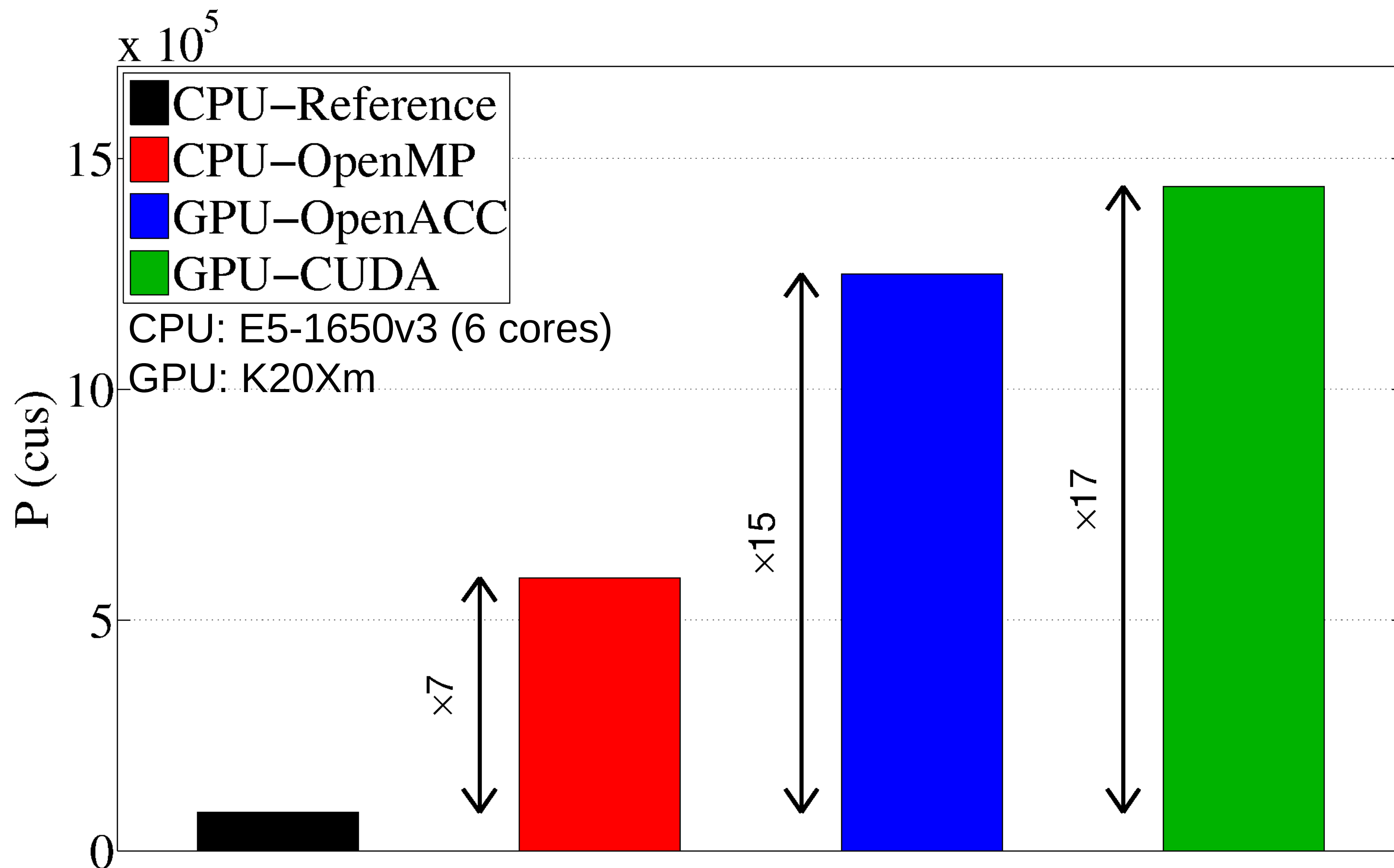
```
#pragma acc data\
      copy(...)\
      copyin(...)\
      create(...)
{
```

Time loop
+acc pragmas

```
}//end of the acc region
```

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.

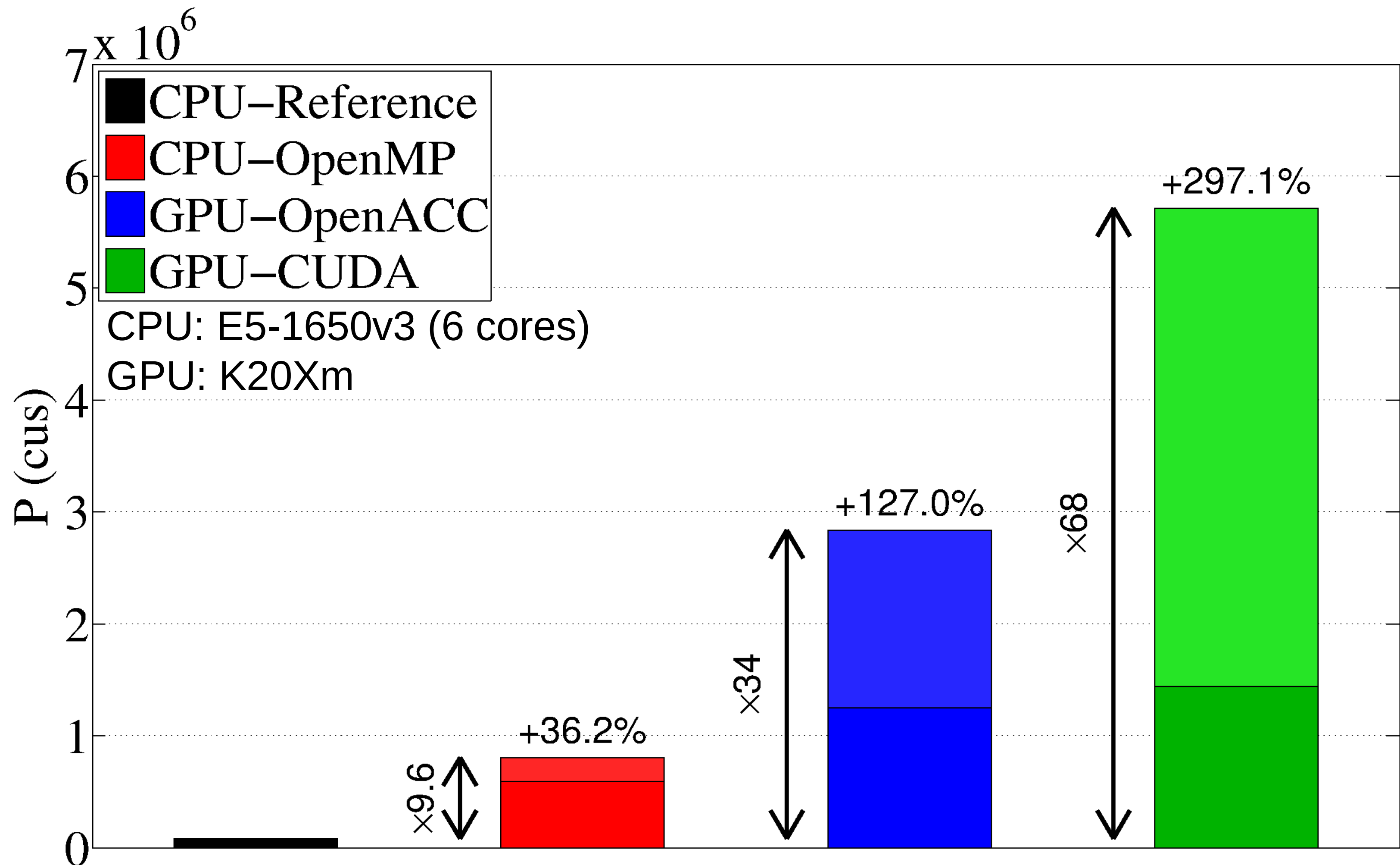
```
__global__
void func(double *results, int N)
{
      int tidx=threadIdx.x+...;
      …
      for(int i=0;i<N;i++)
      {
            …
            results[tidx]+=...
      }
}
```

```
__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;
}
```


GT CNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria cea ONERA THE FRENCH AEROSPACE LAB

Step 1: Reduce remote accesses

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



ONERA
THE FRENCH AEROSPACE LAB
CentraleSupélec
Loria
cea
GT CNRS GeorgiaTech-CNRS UMI 2958

Step 2: Merge similar 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

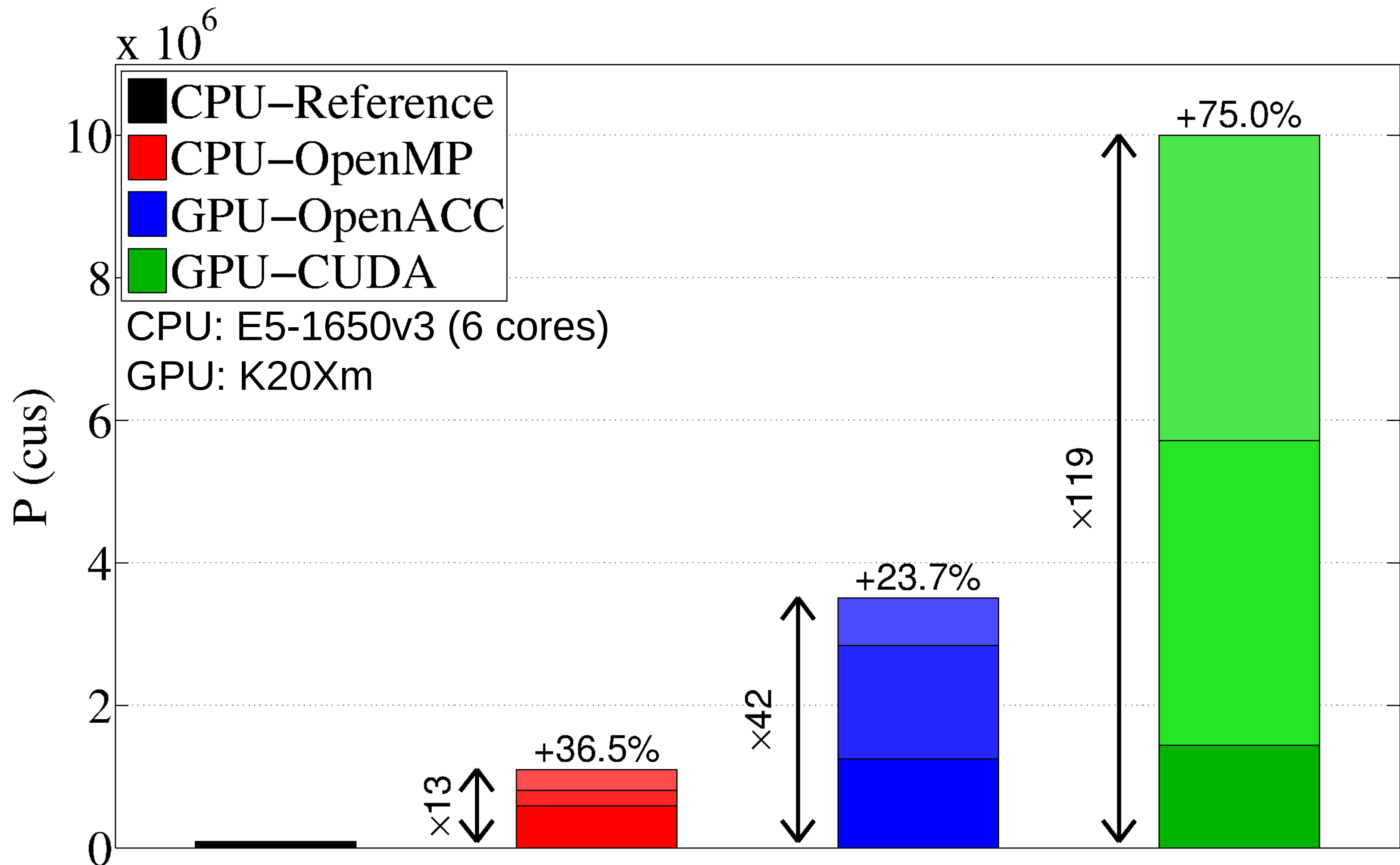
2. For s Runge-Kutta step:

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

GT CNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria cea ONERA THE FRENCH AEROSPACE LAB

Step 2: Merge similar kernels

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



GTCNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria cea ONERA THE FRENCH AEROSPACE LAB

Step 3: Factor computations

Transform divisions into multiplications,

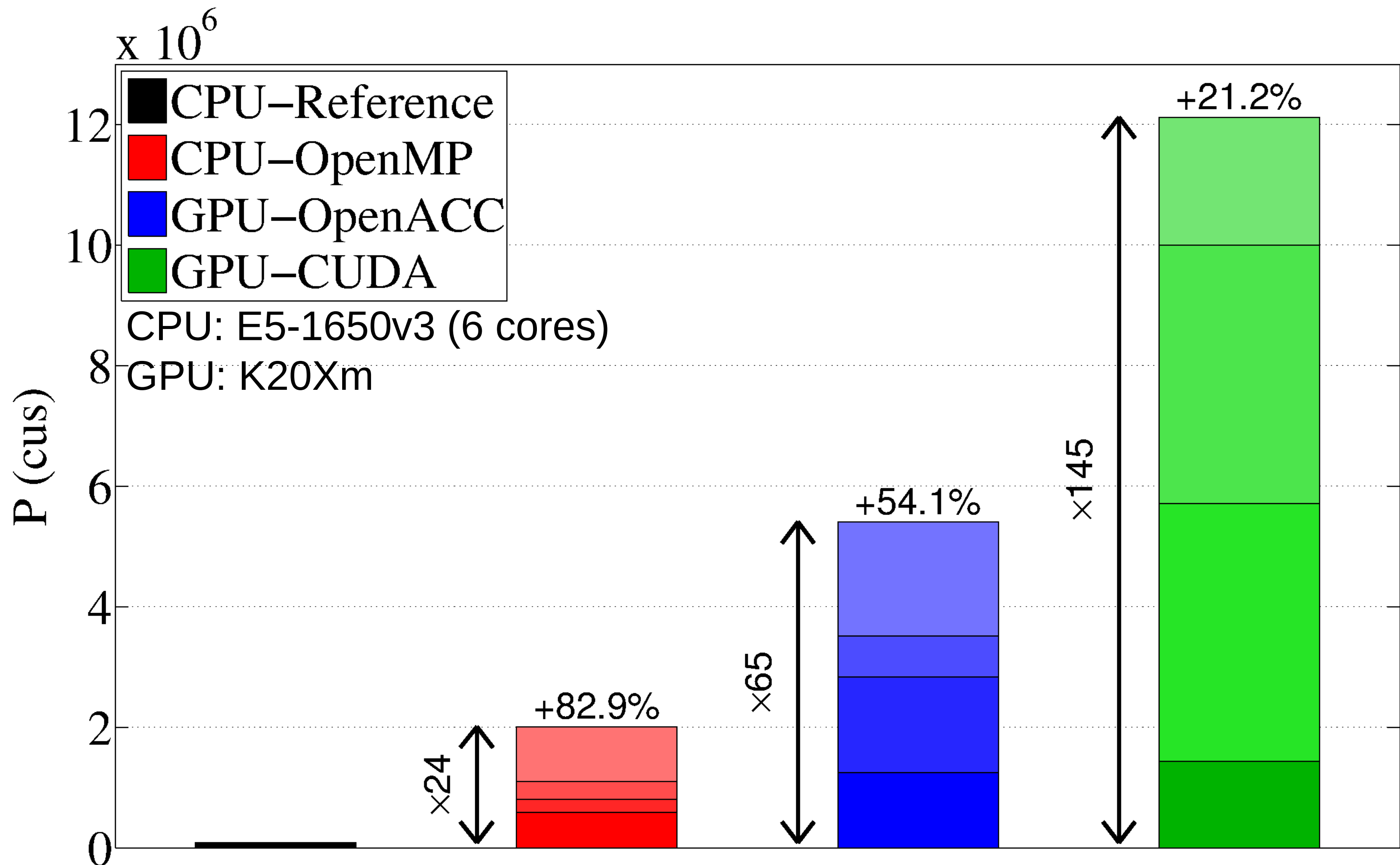
Control whether or not a loop should be unrolled.

```
#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)



CentraleSupélec

Loria

ONERA
THE FRENCH AEROSPACE LAB

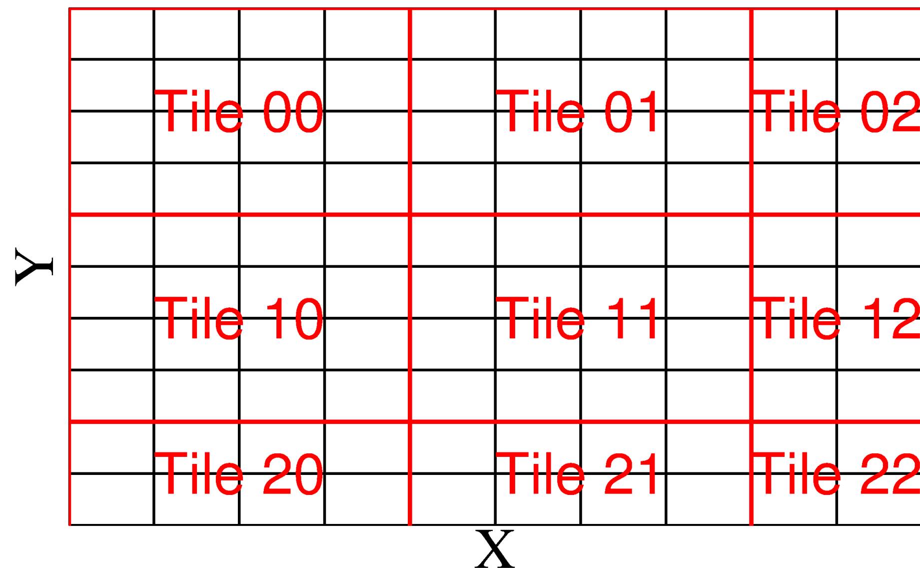
Step 4: Align data (+tiling for CPU)

GPU:

Align data on the size of a warp → coalescence.

CPU:

Tiling (Block) algorithm + fixed block size.

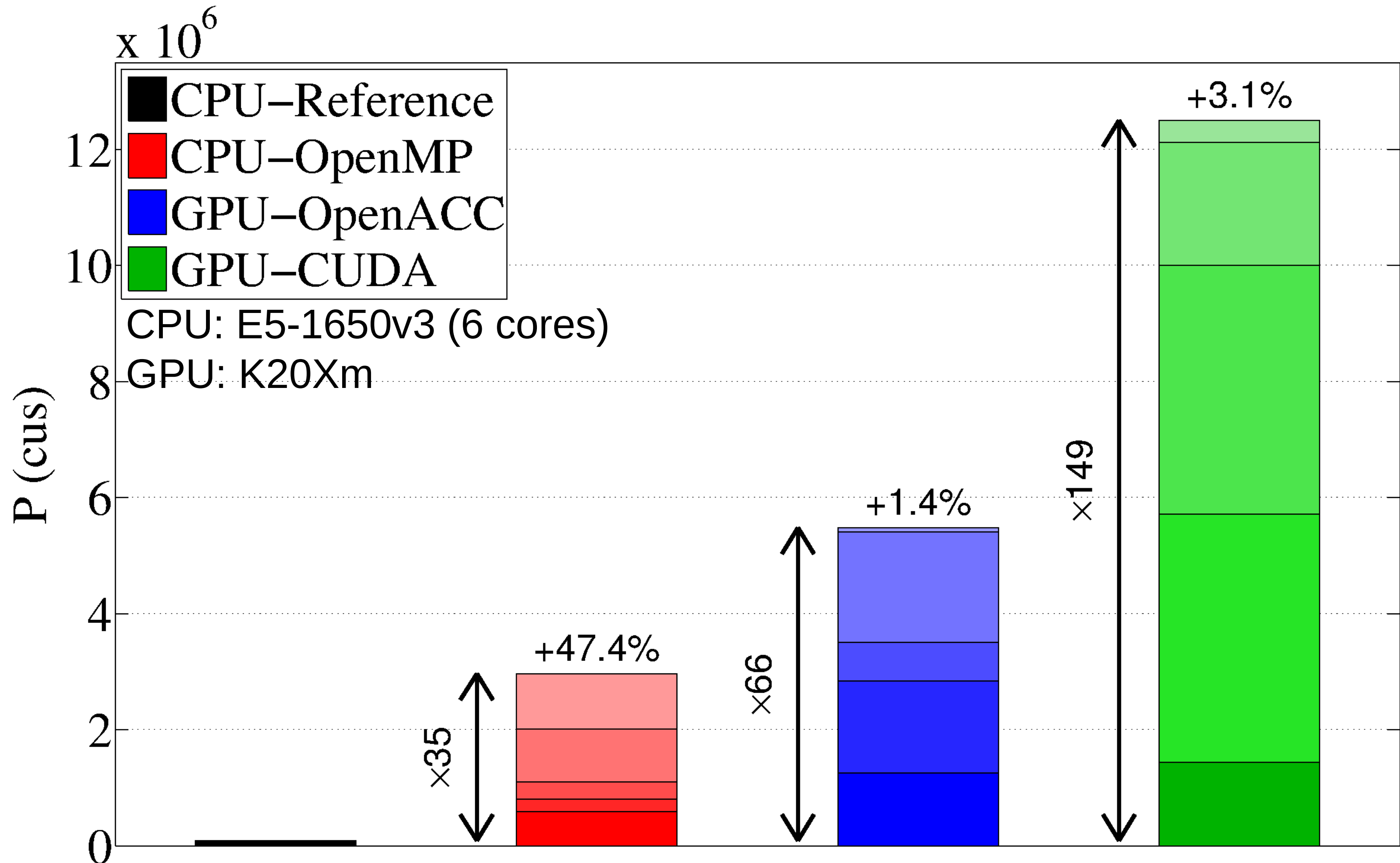


Tiling algorithm for OpenMP threads.

GTCNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria cea ONERA THE FRENCH AEROSPACE LAB

Step 4: Align data (+tiling for CPU)

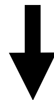
P(cell updt. per sec. [cus])=#of points x number of it. / time (sec)



Step 5: Miscellaneous

OpenACC PGI 16.10:

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



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

CPU:

Introduce MPI to support several nodes and to improve data locality on each one.

#pragma simd on huge loops creates huge register pressure:

→ not suitable for CPU vectorization.

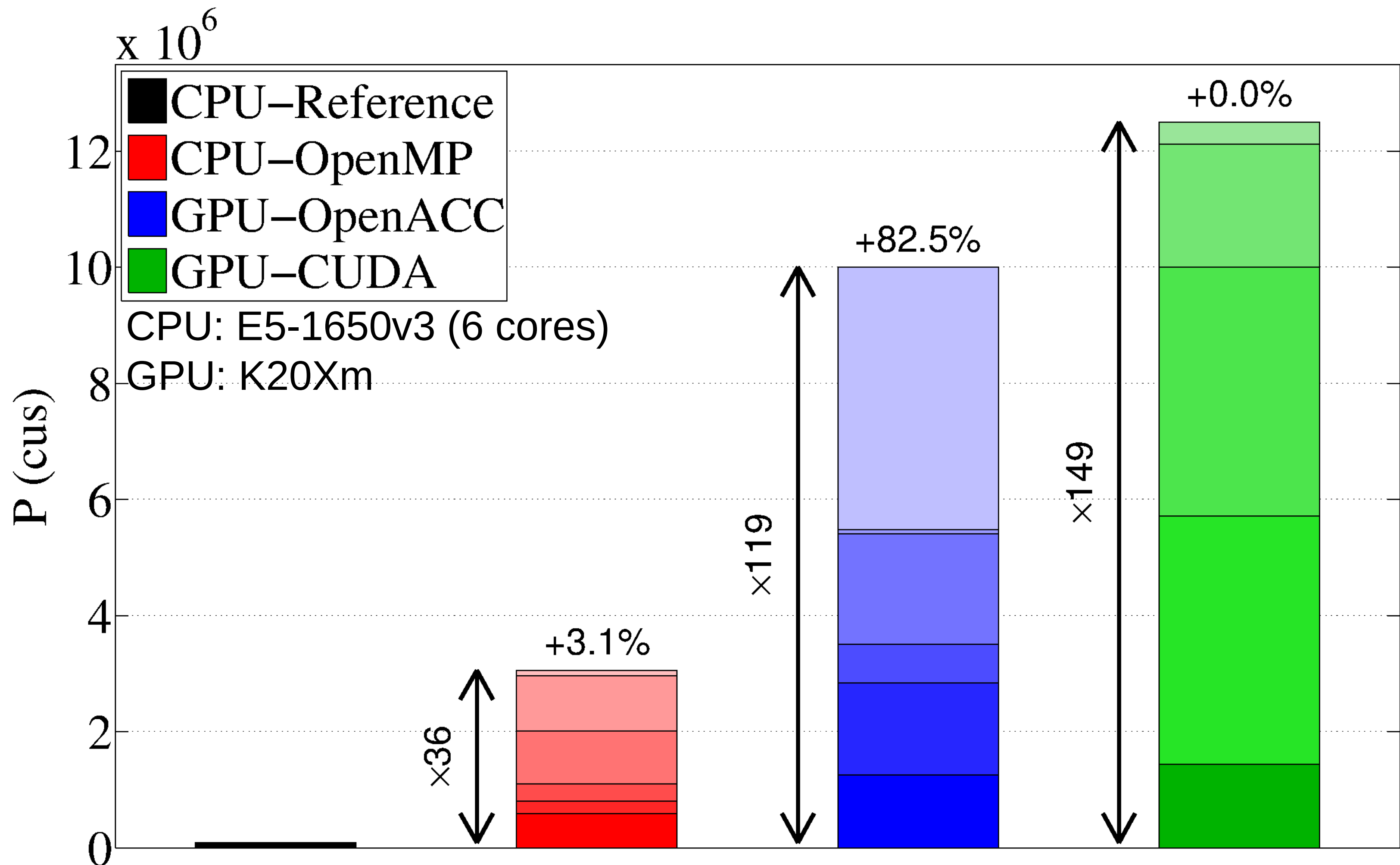
→ split SIMD loops into smaller loops

→ adjust vectorization with #pragma vector always.

GT CNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria cea ONERA THE FRENCH AEROSPACE LAB

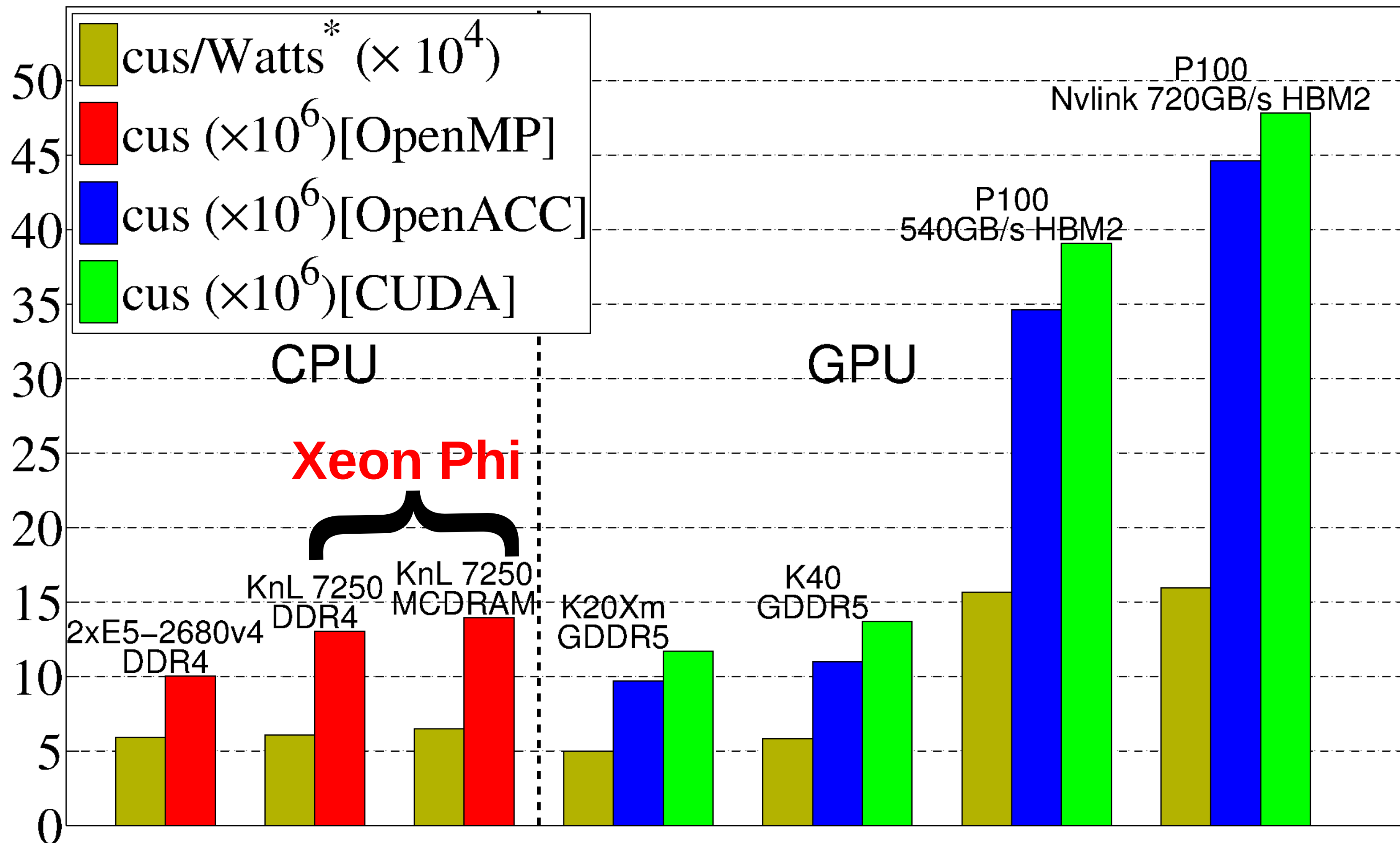
Step 5: Miscellaneous

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

Vectorized OpenACC code

Run on 2xE52680v4 (CPU):
OpenACC code for GPU with pgc++ and
-ta=multicore -tp=x64 (CPU execution)
→ 2.72 x 10^6 cus → ~25% performances of the Intel version.

CPU code with pgc++ and
-ta=multicore -tp=x64 (CPU execution)
→ 5.4 x 10^6 cus → ~50% performances of the Intel version.

Therefore:
OpenACC for CPUs must be developed on smaller loops than GPU kernels.
Add tiling algorithms to improve CPU cache efficiency.

GT CNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria cea ONERA THE FRENCH AEROSPACE LAB

What's next?

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

→this optimization strategy leads to highly similar kernels.

Could it be possible to use the **same kernels for both** CPU and GPU **within a single version of the code**?

GT CNRS GeorgiaTech-CNRS UMI 2958
CentraleSupélec
Loria Laboratoire lorrain de recherche en informatique et ses applications
cea
ONERA THE FRENCH AEROSPACE LAB

A unified development approach

Use C++ templates and pre-compilation variables to define which version should be used.

2D CUDA-GPU code (GPU.cu)	Common vectorized kernel (kernel.h)	2D CPU code (CPU.cpp)

2D CUDA-GPU code (GPU.cu)

```
1  #define VSIZ 1
2  //Include the kernel
3  #include "kernel.h"
4
5  void __global__ gpu_function(
6    double *__restrict__ val)
7  {
8    //Thread indexes
9    int tidx = ...;
10   //Logical condition
11   //to make computations
12   if(tidx<Nx)
13   {
14     kernel<VSIZ>(val,tidx);
15   }
16 }
17
18
```

Common vectorized kernel (kernel.h)

```
1  template<const int VSIZ>
2  //Conditional compilation
3  #ifdef DEFGPU
4  __device__
5  #endif
6  __inline void kernel(
7    double *__restrict__ val,
8    const int tidx)
9  {
10   //Vectorized loop
11   #pragma vector always
12   #pragma unroll
13   for(vec=0; vec<VSIZ; vec++)
14   {
15     ...
16     val[VSIZ*tidx+vec] = ...;
17   }
18 }
```

2D CPU code (CPU.cpp)

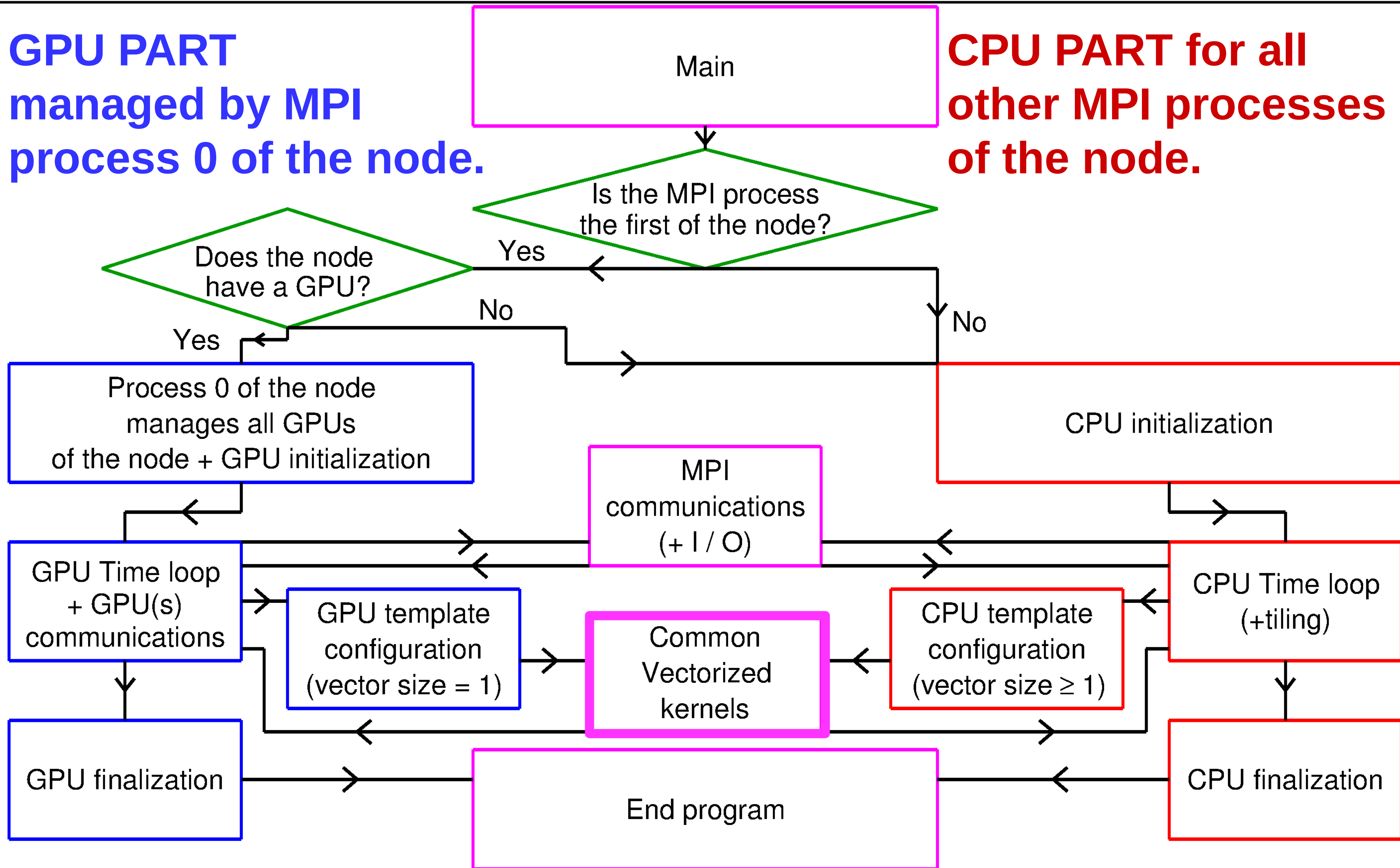
```
1  #define VSIZ 32
2  //Include the kernel
3  #include "kernel.h"
4
5  void cpu_function(
6    double *__restrict__ val)
7  {
8    //CPU loop
9    for(int tidx=0;
10       tidx<Nx;
11       tidx+=VSIZ)
12   {
13     kernel<VSIZ>(val,tidx);
14   }
15 }
16
17
18
```

Compiler directives (ignored when not using the right compiler),

Template parameter to control the vector size (VSIZ=1 for GPU, VSIZ ≥ 1 for CPU),

Inline keyword to avoid conflicts in function names.

Sketch of the hybridized code



Work-balancing

Poor balance between performances of CPUs and GPUs.

→ Micro-benchmark inside the code to balance work between processes:

Performances of MPI process i with the actual work-balance.

Work for MPI process i.

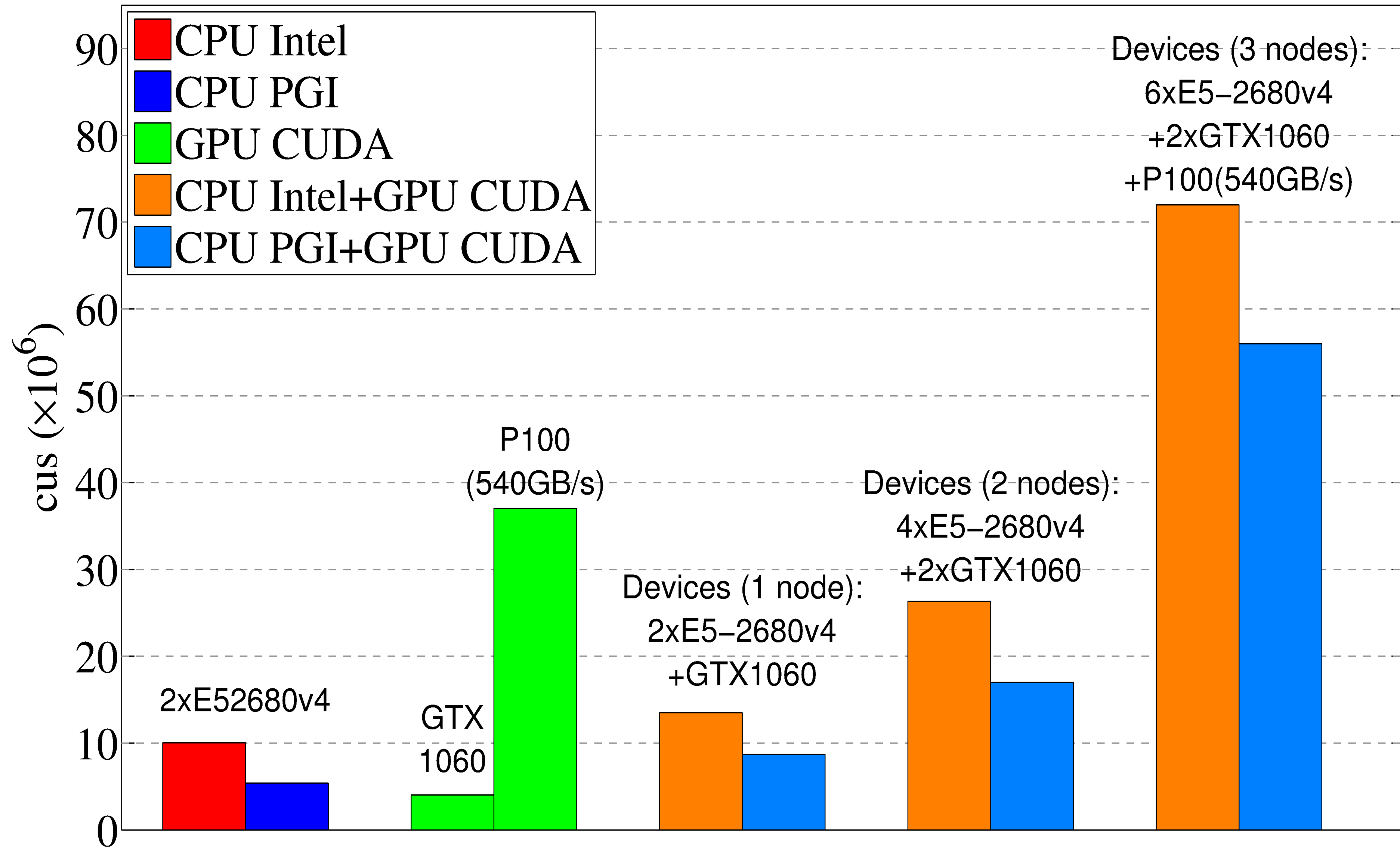
$$W_i = \frac{P_i}{\sum_{i=0}^{n_p} P_i} W_T$$

Total work to be done.

Sum of performances with the actual work-balance.

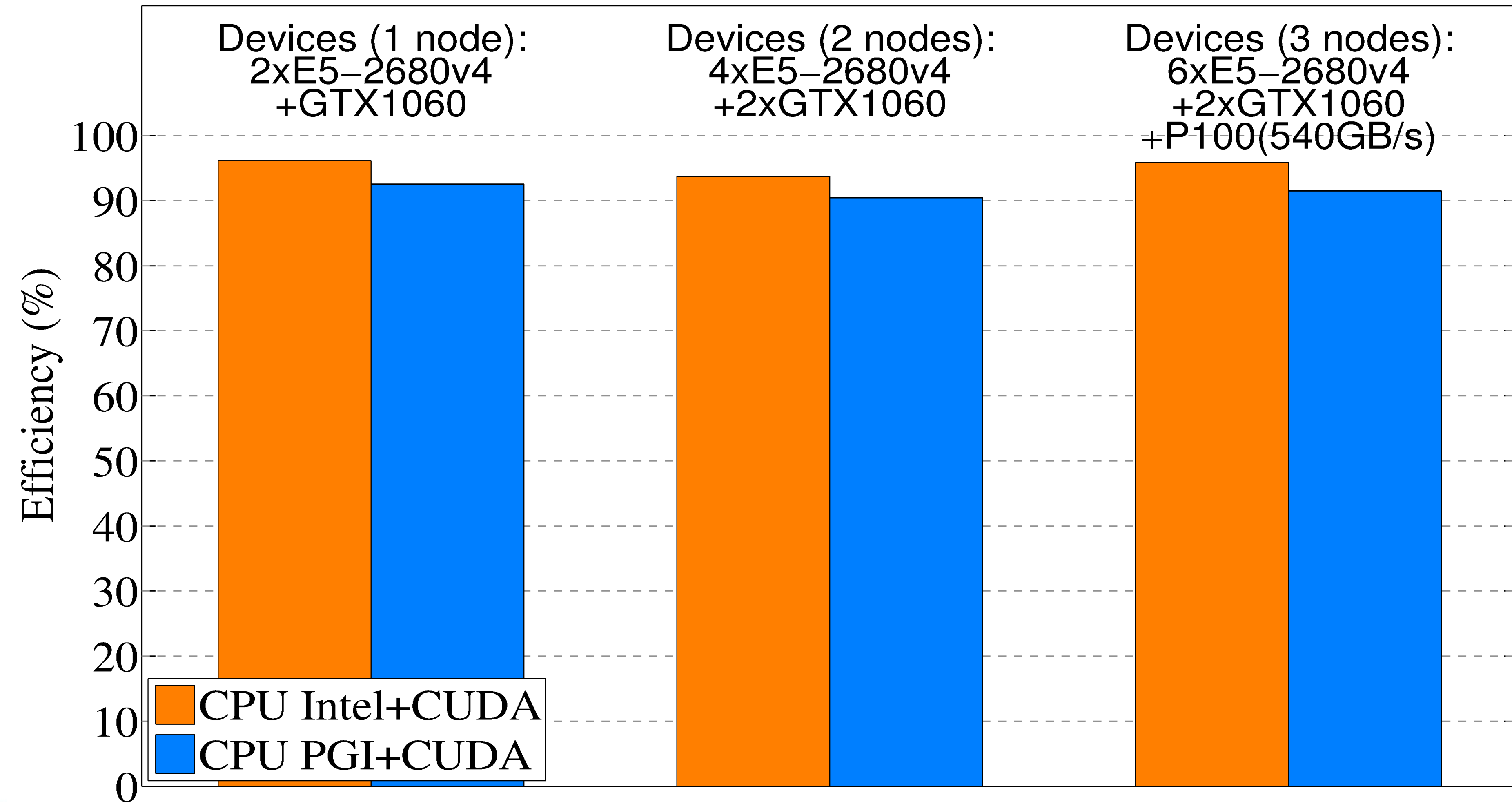
GT CNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria Laboratoire lorrain de recherche en informatique et ses applications cea ONERA THE FRENCH AEROSPACE LAB

Performances



Strong scalability

Efficiency (%) = (effective performance/max. performance)*100



Conclusion

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

→ **4 steps** to optimize:

Reduce access to remote memories,

Merge kernels with similar data access pattern,

Factor your computations,

Align your data.

Future work: move to unstructured meshes

→ ranking on engine's performances may change.

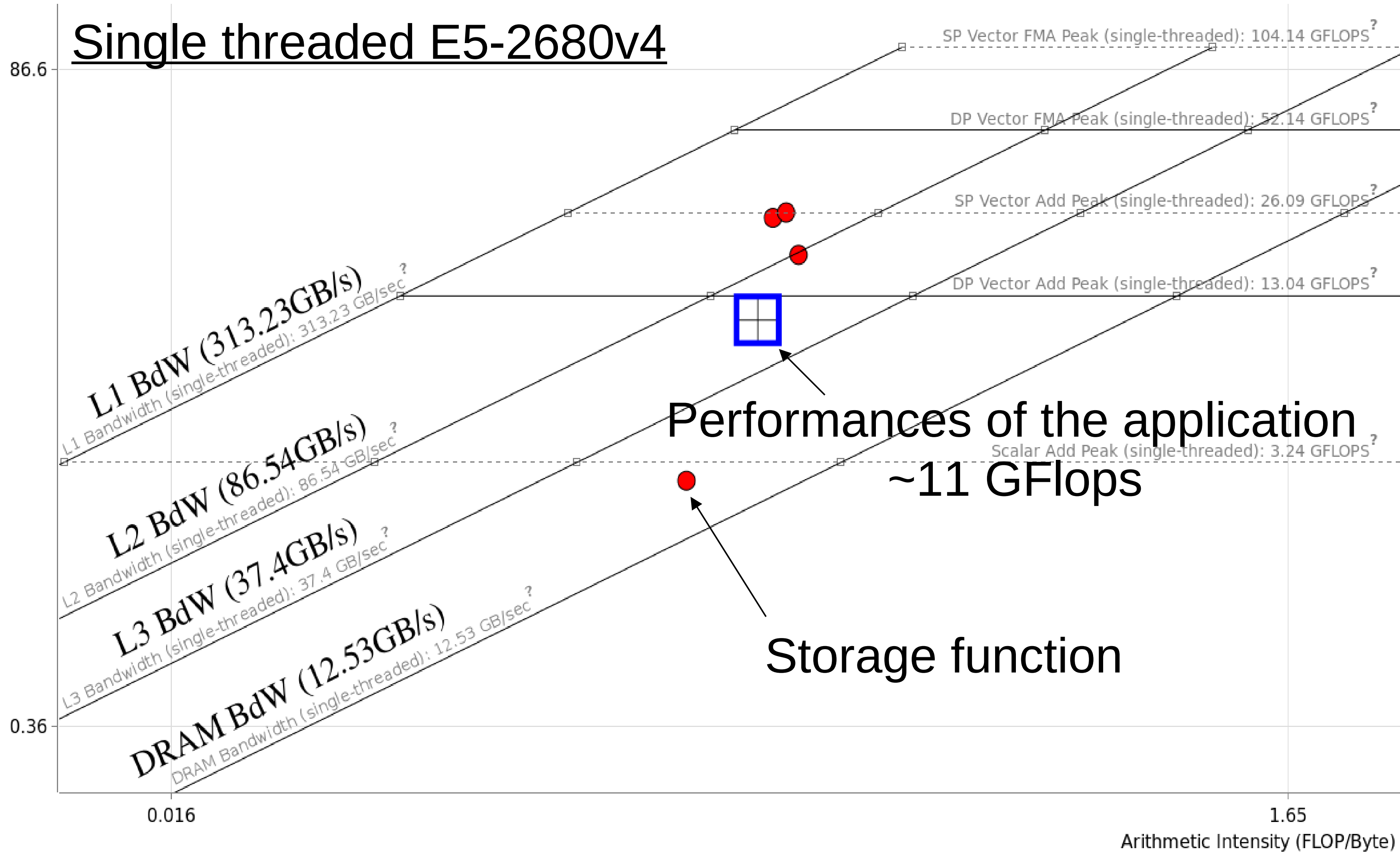
To go deeper into GPU optimization: [Woolley (2013)].

GT CNRS GeorgiaTech-CNRS UMI 2958 CentraleSupélec Loria Laboratoire lorrain de recherche en informatique et ses applications cea ONERA THE FRENCH AEROSPACE LAB

Thank you for your attention.

Contact: ludomir.oteski@onera.fr

Cache roofline (Intel Advisor 2017)



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	--	--