Towards a Unified CPU-GPU code hybridization: A GPU Based Optimization Strategy Efficient on Other Modern Architectures

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

S. Vialle, J. Ryan

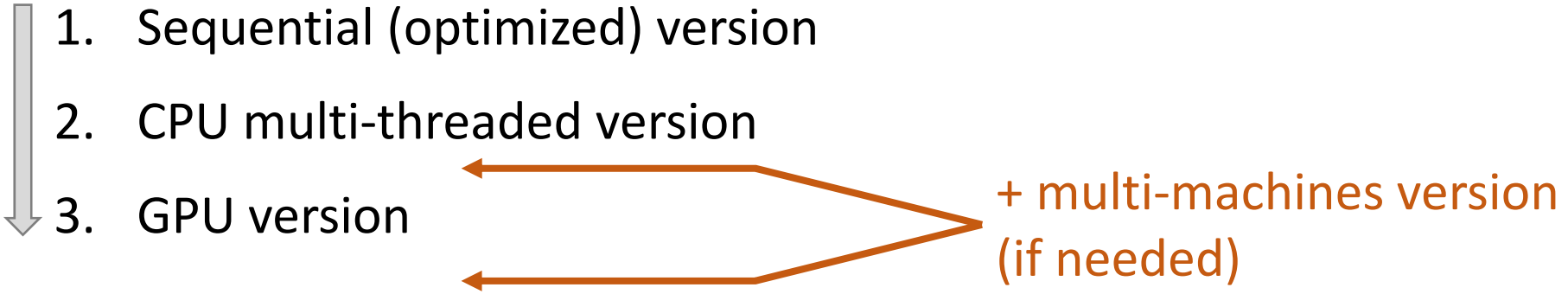
Acks.: CEA/DIF, IDRIS, GENCI, NVIDIA, Région Lorraine

Introduction and context

Most scientific simulations

- Require intensive computations
- Are frequently based on parallel iterative processes

Classical development process:



We propose to invert the parallel development process:

- **Begin with the GPU version**
- **To rapidly derive efficient CPU versions**: vectors + threads (+ MPI)

Test case application: Discontinuous Galerkin schemes for Computational Fluid Dynamics

Representative of some compute-intensive applications

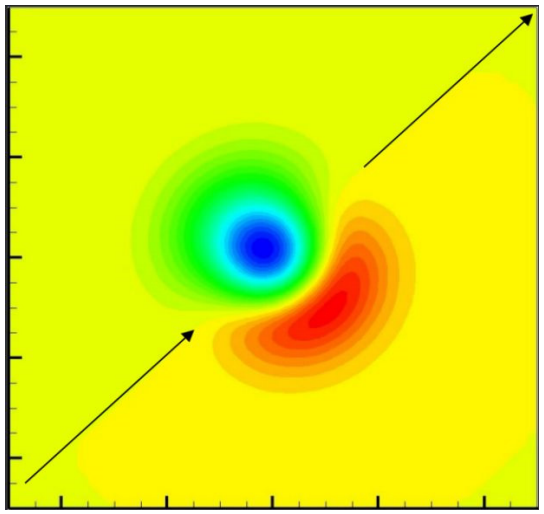
We consider the 2D time-dependant compressible Navier-Stokes equations

Example:

Yee's vortex with the full NS equations

Numerical method:

- **Space:** 2nd order Discontinuous Galerkin,
- **Time:** 3rd order TVD Runge-Kutta,
- **Boundaries:** X and Y-periodic.



Test case application: Discontinuous Galerkin schemes for Computational Fluid Dynamics

Representative of some compute-intensive applications

We consider the 2D time-dependant compressible Navier-Stokes equations

→ Storage space per array of variables: $N_{coefs} \times N_{eq} \times N_{cells}$

N_{coefs}: number of polynomial coefficients (6 for 2D second order polynomials)

N_{eq}: number of equations (4 in our case)

$N_{cells} = N_x \times N_y$: number of cells in the x and y directions

→ 3 double precision arrays: current state, previous time-step and time-derivatives

→ **Memory cost ≈ $3 \times (6 \times 4 \times N_{cells}) \times 8$ Bytes**

Ex : **2001 x 2001 mesh ≥ 2.3 GBytes**

2731 × 2731 mesh ≥ 4.3 GBytes

Unified Development Approach

GPU and CPU have different architectures, BUT :

Today both use **vectorization**:

- *Vectors* for CPU
- *Warps* for GPU

→ Efficient computing scheme on GPU should provide:

- Compliant CPU scheme with minor adaptations
- Efficient CPU execution

Optimized GPU programming:

- is low-level (« close to the metal »)
- low-level API (CUDA, OpenCL) allows accurate ctrl of computations

→ Clear impact of optimization attempts on GPU

→ Interest of using GPU as the first development step

Unified Development Approach

Example of GPU kernel translated into a vectorized CPU function

2D Grid of blocks of CUDA threads → CPU nested loops

2D CUDA-GPU code

```
1  void __global__ function(...)
2  {
3    //Thread index in x-direction
4    int tidx = ...;
5    //Thread index in y-direction
6    int tidy = ...;
7    //Logical condition
8    //to make computations
9    if(tidx<Nx && tidy<Ny)
10   {
11     ... //Copy to CPU code
12   }
13 }
```

2D CPU code (vectorized with icc)

```
1  void function(...)
2  {
3    //Loop on y-direction
4    #pragma omp for
5    for(tidy=0; tidy<Ny; tidy++)
6    {
7    //Loop on x-direction
8      #pragma simd
9      for(tidx=0; tidx<Nx; tidx++)
10     {
11       [...] //Insert from GPU code
12     }
13   }
14 }
15
```

Optimization guideline overview

4 crucial GPU optimizations applied to the CPU version

1. Reduce accesses to distant memories
2. Merge kernels which share the same memory patterns
3. Simplify and factor computations
4. Align data (including tiling: CPU specific optimization)

+ CPU specific optimizations

a. Vectorization tuning
b. Data locality improvement

Optimization guideline: GPU → CPU (1)

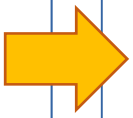
1. Reduce accesses to distant memories

Transfer redundant accesses to global/DRAM memory into registers

→ GPU avoids unnecessary accesses to global memory

→ CPU improves data cache locality

```
__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;
}
```

→ Only 1 read and 1 write to **`results[]`** array

Ex: GPU perf ×3.97, CPU perf ×1.36

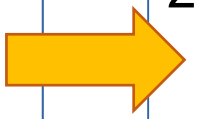
Optimization guideline: GPU → CPU (2)

2. Merge kernels which share the same memory patterns

→ GPU limits its number of accesses to distant memories by improving the re-use of distant variables

→ CPU increases its cache re-use

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. All fluxes in X
   c. All fluxes in Y
   d. Integral+Runge-Kutta

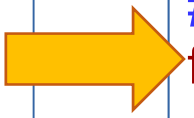
Ex: GPU perf ×1.75, CPU perf ×1.37

Optimization guideline: GPU → CPU (3)

3. Simplify and factor computations

- Suppress non essential variables
- Store recurrent computations in inter-mediate vars (mult by const, const square roots,...)
- Control whether or not a static 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*...
  ...
}
```

→ The most time-consuming development step

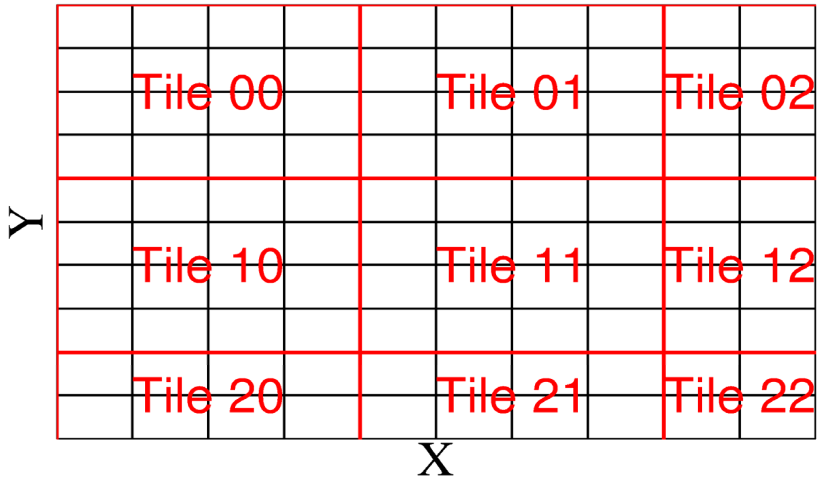
Ex: GPU perf ×1.21, CPU perf ×1.82

Optimization guideline: GPU → CPU (4)

4. Align data

- GPU: align data access on warp size : 32 (static size)
  → ensures coalescent memory accesses
  → GPU perf ×1.03 on our problem

- CPU:
  1. Add a tiling algorithm inside each thread computations
     → CPU perf ×1.23
  2. Use static tile size
     → CPU perf ×1.19



Tiling for one OpenMP thread

Static tile size → static nb of loop iterations → better vectorization

Optimization guideline: CPU specific (1)

a. Vectorization tuning

Previous optimizations steps tend to create huge vectorized loops !

→ high increase of register pressure

→ not suitable for CPU vectorization…

So … we adjusted our CPU vectorization strategy

1. by splitting SIMD loops into smaller loops
2. #pragma simd → #pragma vector always

   tells the compiler to perform auto-vectorization if the loop does not carry dependencies

→ CPU perf ×1.03

Compromise *distant memory accesses reduction* / *register pressure* for optimal CPU vectorization

Optimization guideline: CPU specific (2)

b. Data locality improvement

MPI-OpenMP version of the CPU code → to improve data locality

- Memory locations of data used in (only) one MPI process and its threads will be close to their associated cores
- This optimization is well suited to **NUMA** architectures *(and many machines are becoming NUMA machines !)*

→ CPU perf ×1.03

The domain decomposition has been performed along the y axis by using a ghost-cell technique to share boundaries of neighbouring domains between MPI processes

Experimental steps & performances (1)

Code versions	CPU E5-1650v3 (6th)	Speedup vs seq. CPU	Progressive speedup	GPU K20Xm	Speedup vs seq. CPU	Progressive speedup
Seq. CPU v (1th)	8.35 x10^4 cus	1.00	-			
CPU v1 (OMP+simd pragmas)	0.59 x10^6 cus	7.01	7.01			
GPU v1 (CUDA)				1.44 x10^6 cus	17.21	17.21

cus: Cell Update/s

2001 × 2001 cells

100 steps

Experimental steps & performances (2)

Code versions	CPU E5-1650v3 (6th)	Speedup vs seq. CPU	Progressive speedup	GPU K20Xm	Speedup vs seq. CPU	Progressive speedup
Seq. CPU v (1th)	8.35 x10^4 cus	1.00	-			
CPU v1 (OMP+simd pragmas)	0.59 x10^6 cus	7.01	7.01			
GPU v1 (CUDA)				1.44 x10^6 cus	17.21	17.21
Reduced accesses to distant mem.	0.81 x10^6 cus	9.63	1.36	5.71 x10^6 cus	68.36	3.97
Merged funcs with same mem pattern	1.09 x10^6 cus	13.05	1.37	10.0 x10^6 cus	119.62	1.85
Simplification of computations	2.01 x10^6 cus	24.05	1.82	12.1 x10^6 cus	145.00	1.21
CPU Only: Tiling (cache optim)	2.48 x10^6 cus	29.72	1.23	-	-	-
Data alignment	2.86 x10^6 cus	34.25	1.19	12.5 x10^6 cus	149.53	1.03

cus: Cell Update/s

2001 × 2001 cells

100 steps

Experimental steps & performances (3)

Code versions	CPU E5-1650v3 (6th)	Speedup vs seq. CPU	Progressive speedup	GPU K20Xm	Speedup vs seq. CPU	Progressive speedup
Seq. CPU v (1th)	8.35 x10^4 cus	1.00	-			
CPU v1 (OMP+simd pragmas)	0.59 x10^6 cus	7.01	7.01			
GPU v1 (CUDA)				1.44 x10^6 cus	17.21	17.21
Reduced accesses to distant mem.	0.81 x10^6 cus	9.63	1.36	5.71 x10^6 cus	68.36	3.97
Merged funcs with same mem pattern	1.09 x10^6 cus	13.05	1.37	10.0 x10^6 cus	119.62	1.85
Simplification of computations	2.01 x10^6 cus	24.05	1.82	12.1 x10^6 cus	145.00	1.21
CPU Only: Tiling (cache optim)	2.48 x10^6 cus	29.72	1.23	-	-	-
Data alignment	2.86 x10^6 cus	34.25	1.19	12.5 x10^6 cus	149.53	1.03
CPU Only: Tuning on vectorization	2.96 x10^6 cus	35.44	1.03			
CPU Only: MPI + OMP (3 proc x 2th)	3.05 x10^6 cus	36.52	1.03			

cus: Cell Update/s

2001 × 2001 cells

100 steps

Experimental testbed

Hardware:

- Bi-socket E5-2698v3
- Bi-socket E5-2680v4

Xeon

- KNL 7250

Xeon-phi

- K20Xm
- K40
- P100 540GB/s
- P100 720GB/s

GPU

Application:

- 2D Discontinous Galerkin (NS equations)
- Grid size: 2731×2731

Operating System:

- Linux

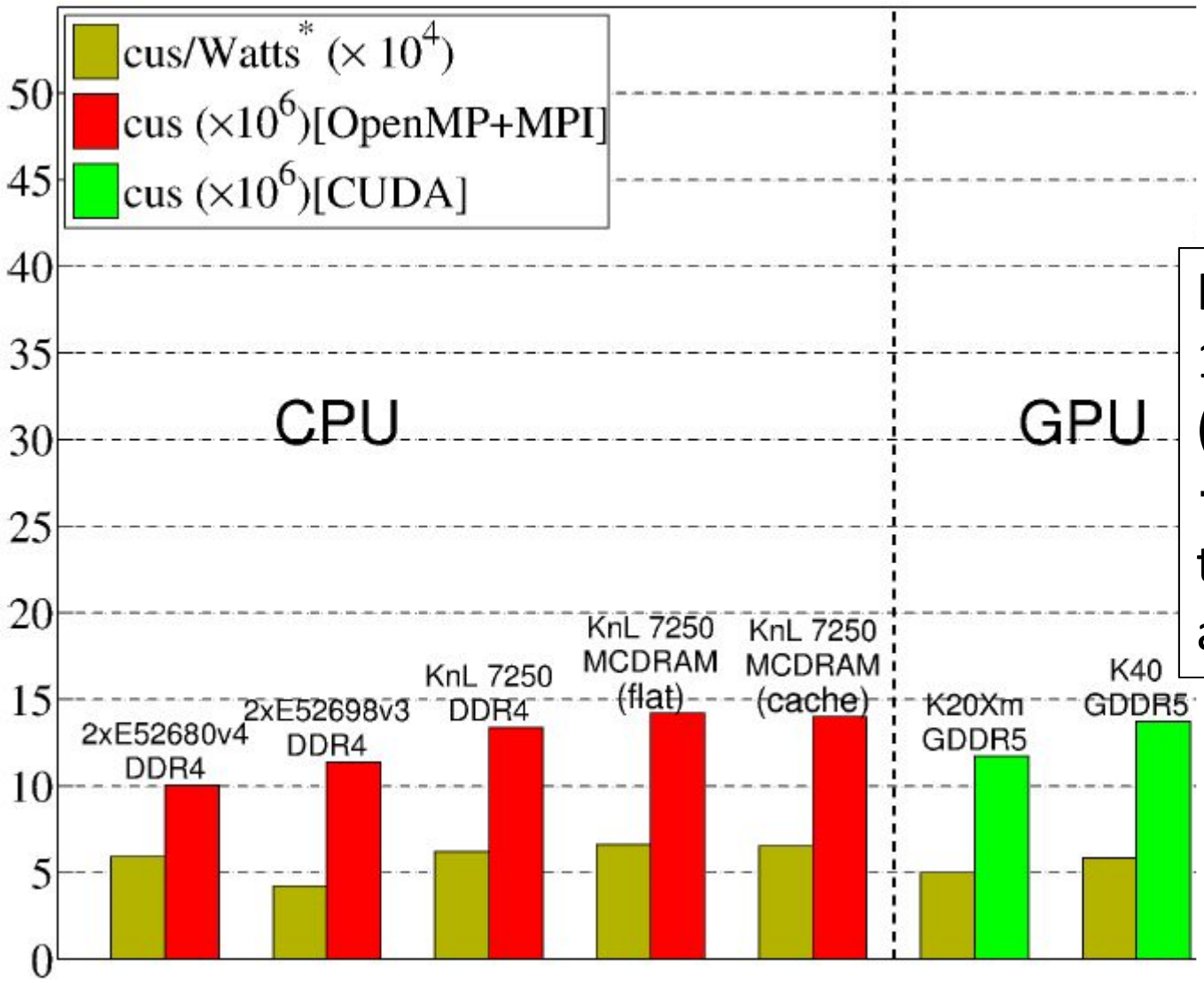
Compilers:

- CUDA 8
- ICC

ICC options:

-O3
-march=native
-fma
-align

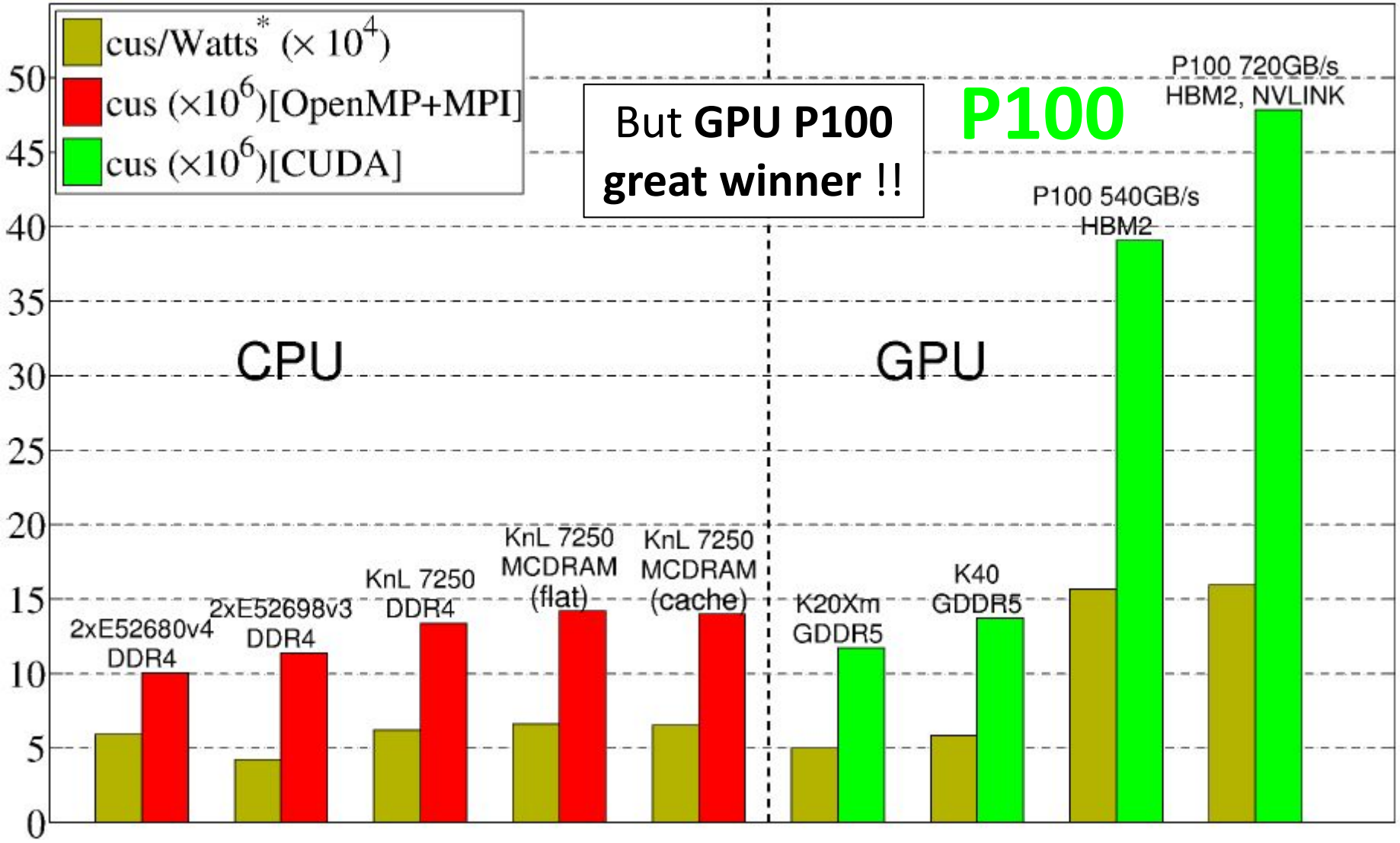
Experimental final performances (1)



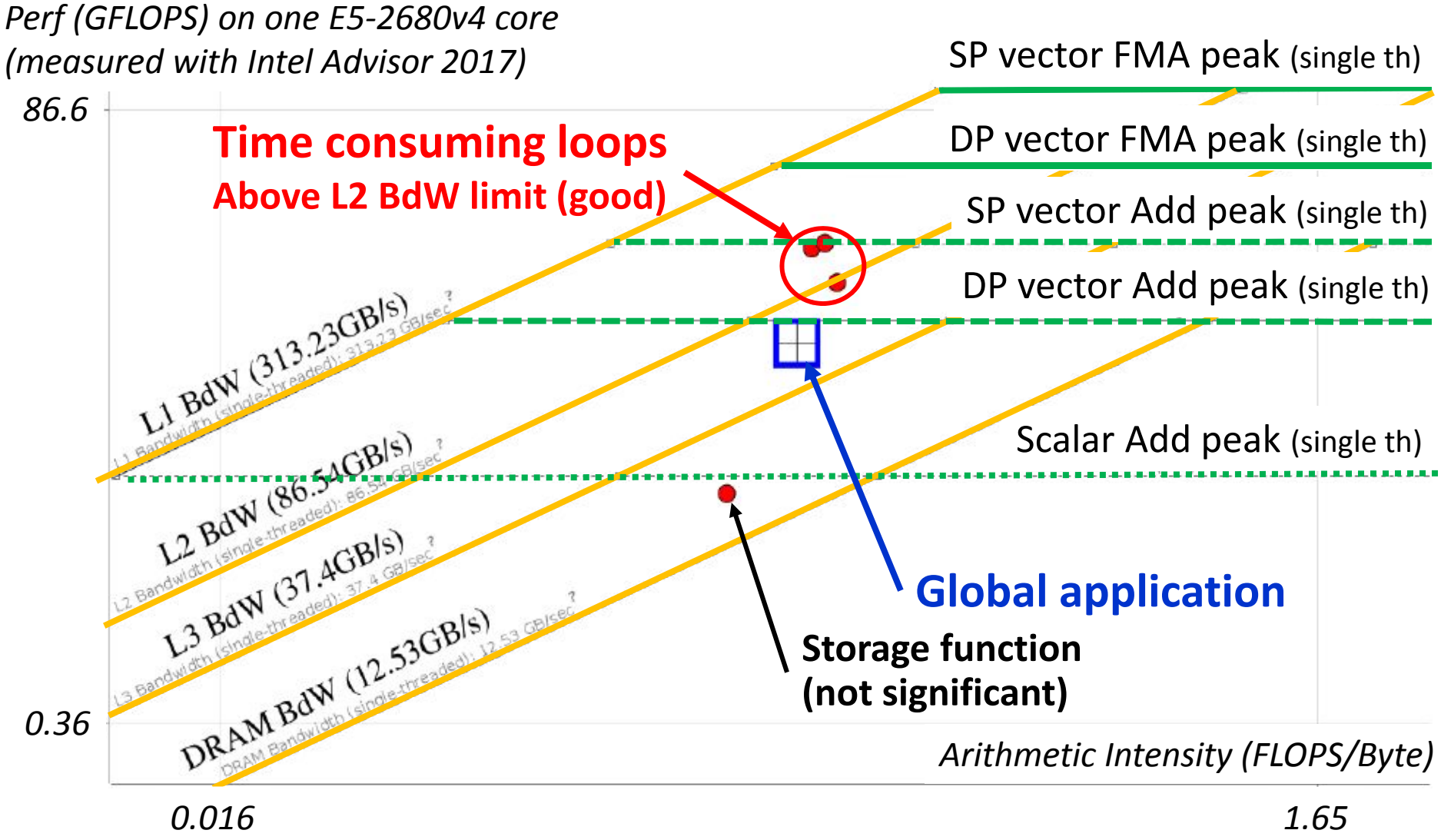
Applying our optimisation guideline

KNL 7250 with 16GB MCDRAM (≈500 GB/s)
→ a little bit faster than GPU K20Xm and GPU K40

Experimental final performances (2)



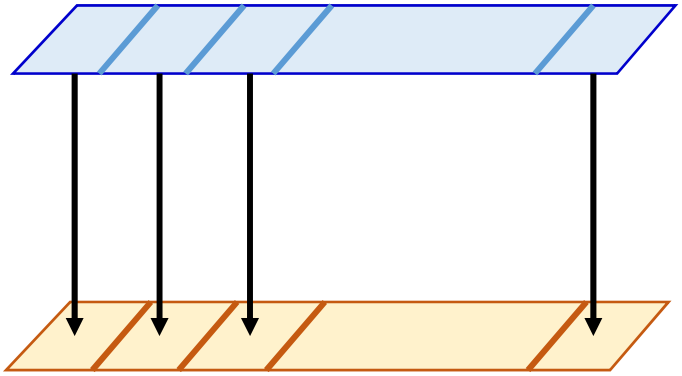
Experimental single-threaded cache roofline profile



Foundation of Hybrid solution

GPU

A large block of n_{th} light threads



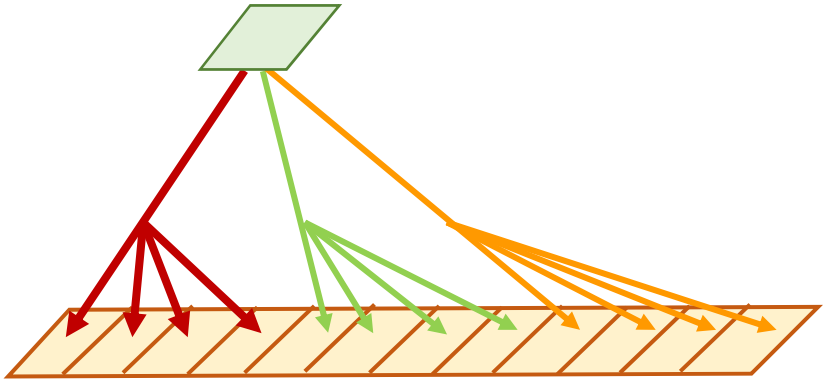
An array of *n* data

$n = n_{th} \times VSIZ$

Ex: 12 = 12 × 1

CPU

One CPU thread with VSIZ vector units



An array of *n* data

$n = n_{th} \times nb_{steps} \times VSIZ$

Ex: 12 = 1 × 3 × 4

Hybrid implementation

Common src code for CPU & GPU

2D CUDA-GPU code (GPU.cu)

```
1  #define VSIZ 1
2  //Include the kernel
3  #define DEFGPU
4  #include "kernel.h"

6  void __global__ gpu_function(
7    double *__restrict__ val)
8  {
9    //Thread indexes
10   int tidx = ...;
11   //Logical condition
12   //to make computations
13   if(tidx<Nx)
14   {
15     kernel<VSIZ>(val,tidx);
16   }
17 }
18
```

Common vectorized kernel (kernel.h)

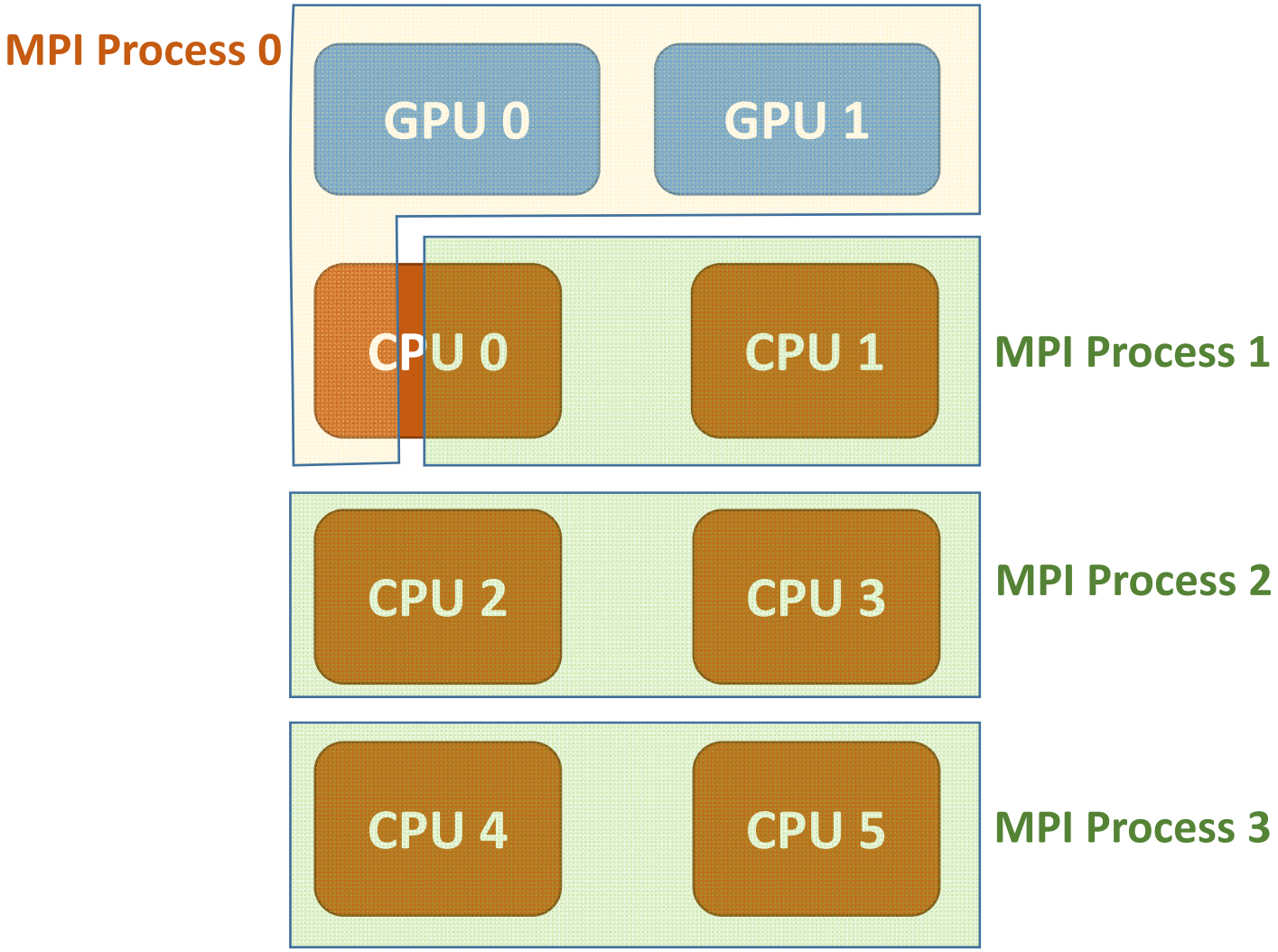
```
1  template<const int VSIZ>
2  //Conditional compilation
3  #ifdef DEFGPU
4  __device__
5  #endif
6  __inline void kernel(
7    double *__restrict__ val,
8    const int tidx)
9  {
10   //Vectorized loop
11   #pragma vector always
12   #pragma unroll
13   for(vec=0; vec<VSIZ; vec++)
14   {
15    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

2D CPU code (CPU.cpp)

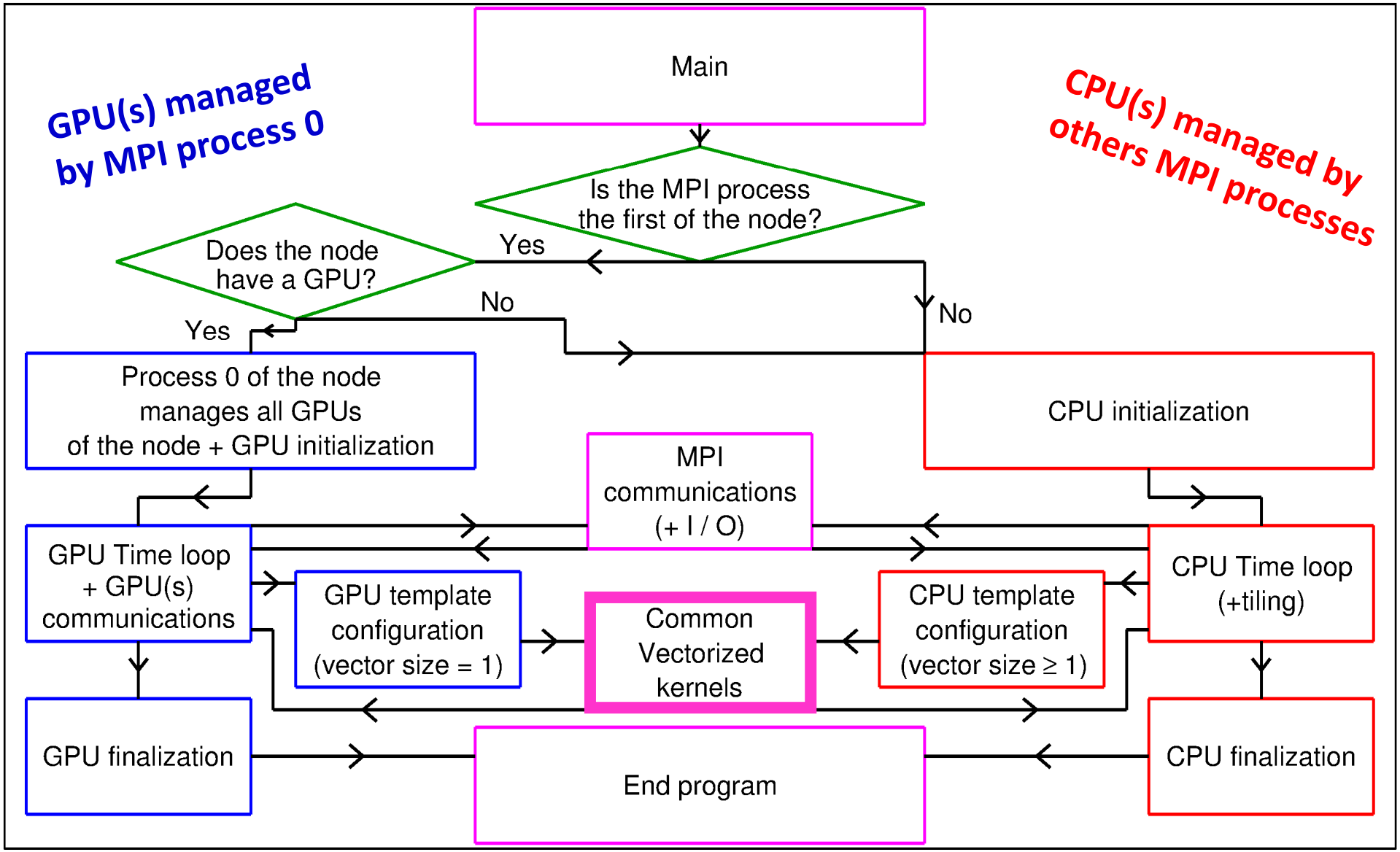
```
1  #define VSIZ 32
2  //Include the kernel
3  #include "kernel.h"

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 }
18
```

- **C++ template** code, parametered with the nb of elements processed by each thread
- **Compiler directives** (ignored when not using the right compiler and architecture)

Hybrid system



Sketch of hybridized code



Experimental Hybrid Performance

Devices	CPU run: 2xE5-2680v4	GPU run: P100 (540GB/s)	Hybrid run: 2xE5-2680v4 + P100
Perf (x10^6 cus)	**10.4**	**37.0**	**46.0**

Maximum perf : 10.4 + 37.0 = **47.4**

Close to ideal (hybrid) performances

Conclusion & Perspective

- General programming guideline for modern computing devices
  CPU, NUMA CPU, GPU, Xeon-Phi KNL
- Highly efficient with the Discontinuous Galerkin problem
  Should also be efficient on other highly vectorizable problems
- Common (hybrid) computing kernel for all devices
  Inserted in a hybrid source code (for CPU+GPU machines)

→ GPU development also valuable for other (vector) targets
→ Increases the interest of developing first on GPU

Perspective:

(Half-) Automatic code generation for CPU from the GPU one

Towards a Unified CPU-GPU code hybridization:
A GPU Based Optimization Strategy Efficient on Other Modern Architectures

Questions ?

We propose to invert the parallel development process:

- Begin with the GPU version
- To rapidly derive efficient CPU versions: vectors + threads (+ MPI)