Portability, Efficiency and Maintainability: the case of OpenACC

Enrico Calore, Sebastiano Fabio Schifano, Raffaele Tripiccione

University of Ferrara and INFN-Ferrara

SUMA workshop

February 13, 2015

Trento, ITALY

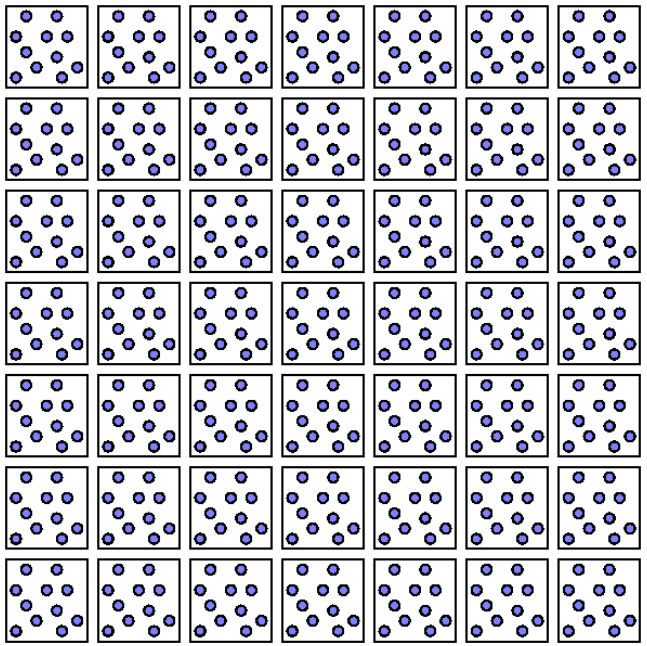
The D2Q37 Lattice Boltzmann Model

- Lattice Boltzmann method (LBM) is a class of computational fluid dynamics (CFD) methods
- simulation of synthetic dynamics described by the discrete **Boltzmann** equation, instead of the **Navier-Stokes** equations
- a set of **virtual particles** called **populations** arranged at edges of a discrete and regular grid
- interacting by **propagation** and **collision** reproduce – after appropriate averaging – the dynamics of fluids
- D2Q37 is a D2 model with 37 components of velocity (populations)
- suitable to study behaviour of **compressible** gas and fluids optionally in presence of **combustion** [1] effects
- correct treatment of *Navier-Stokes*, heat transport and perfect-gas ($P = \rho T$) equations

[1] chemical reactions turning cold-mixture of reactants into hot-mixture of burnt product.

Computational Scheme of LBM

```
foreach time_step

  foreach lattice_point
    propagate();

  foreach boundary_point
    bc();

  foreach lattice_point
    collide();

endfor
```



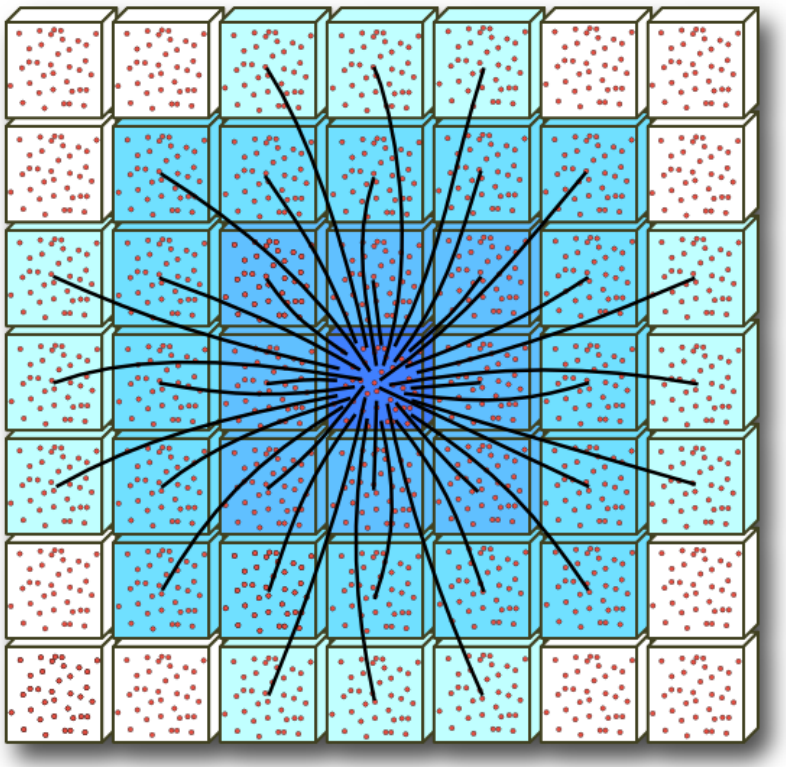
Embarassing parallelism

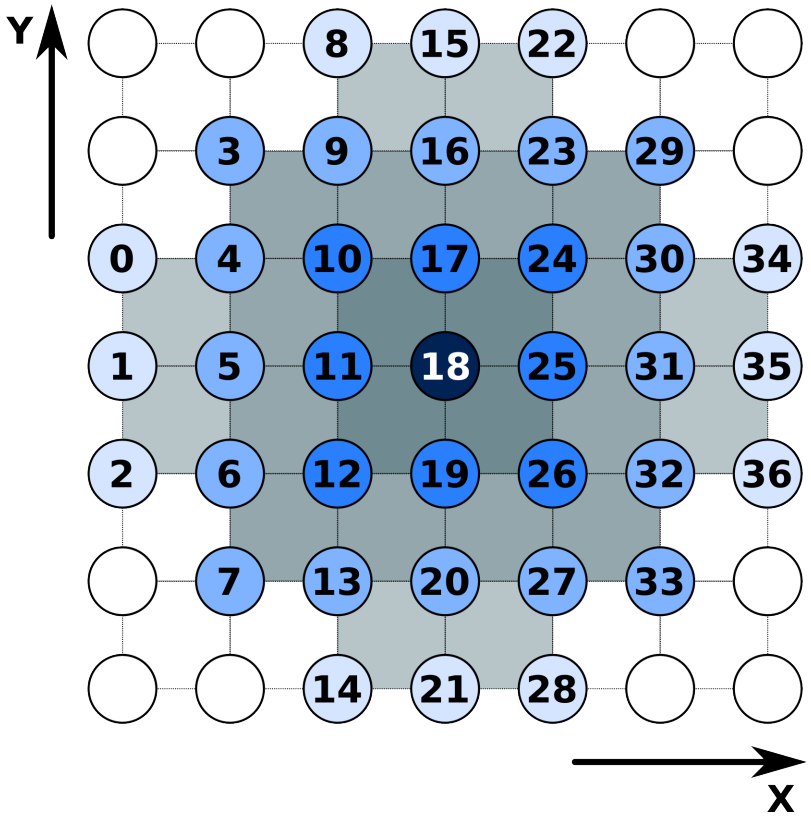
All sites can be processed in parallel applying in sequence propagate and collide.

Challenge

Design an efficient implementation to exploit a large fraction of available peak performance.

D2Q37: propagation scheme

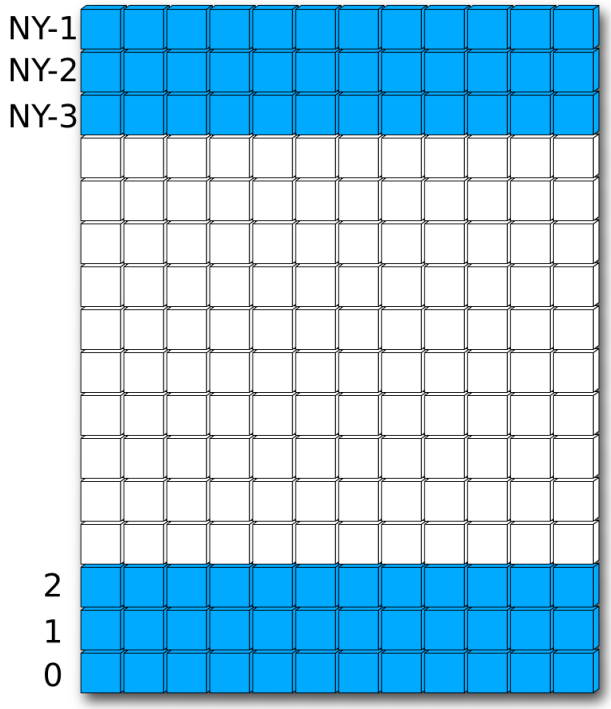




- require to access neighbours cells at distance 1,2, and 3,
- generate memory-accesses with **sparse** addressing patterns.

D2Q37: boundary-conditions

- we simulate a 2D lattice with period-boundaries along x-direction
- at the top and the bottom boundary conditions are enforced:
  - to adjust some values at sites $y = 0 \ldots 2$ and $y = N_y - 3 \ldots N_y - 1$
  - e.g. set vertical velocity to zero

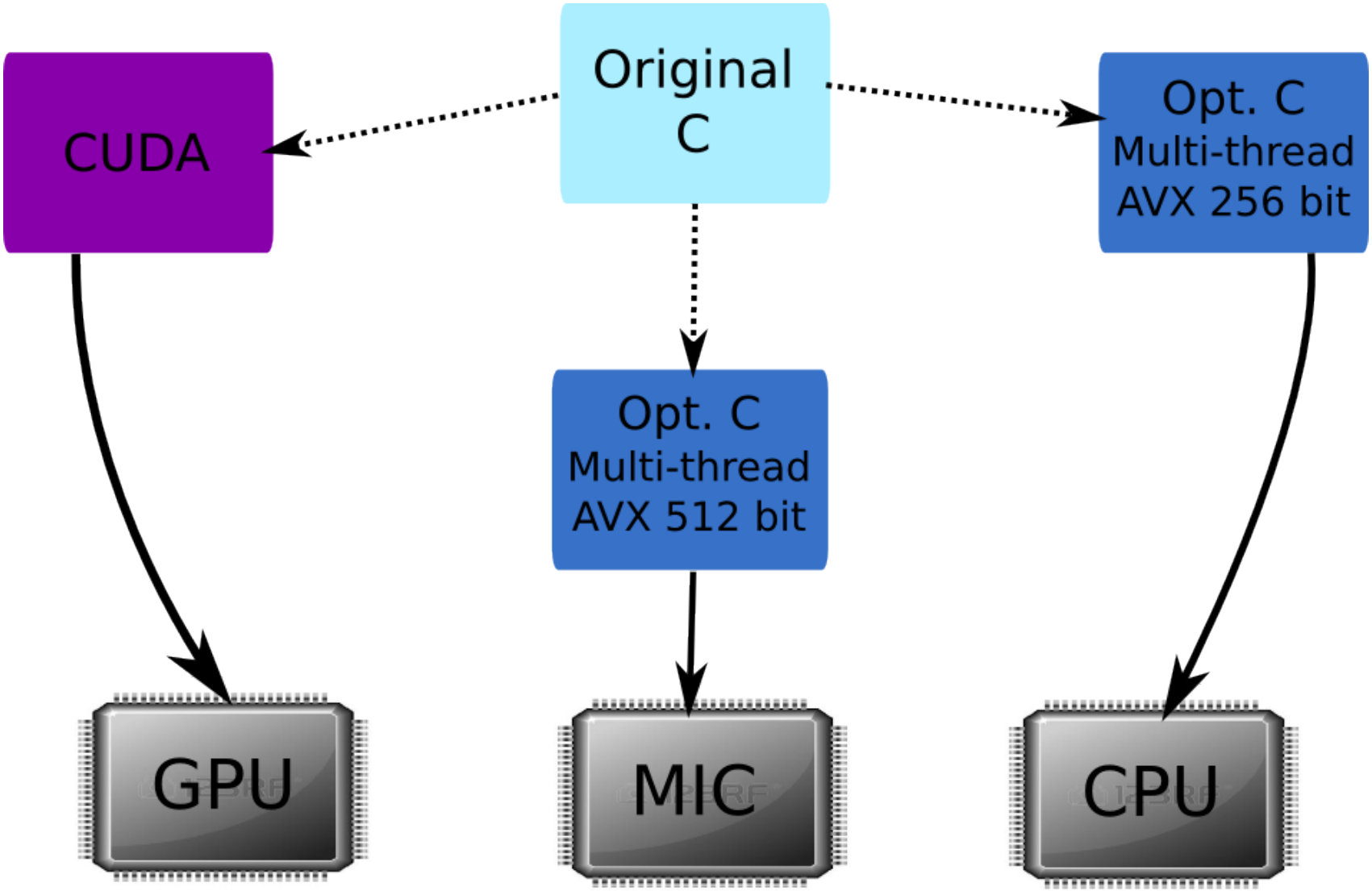


This step (bc) is computed before the collision step.

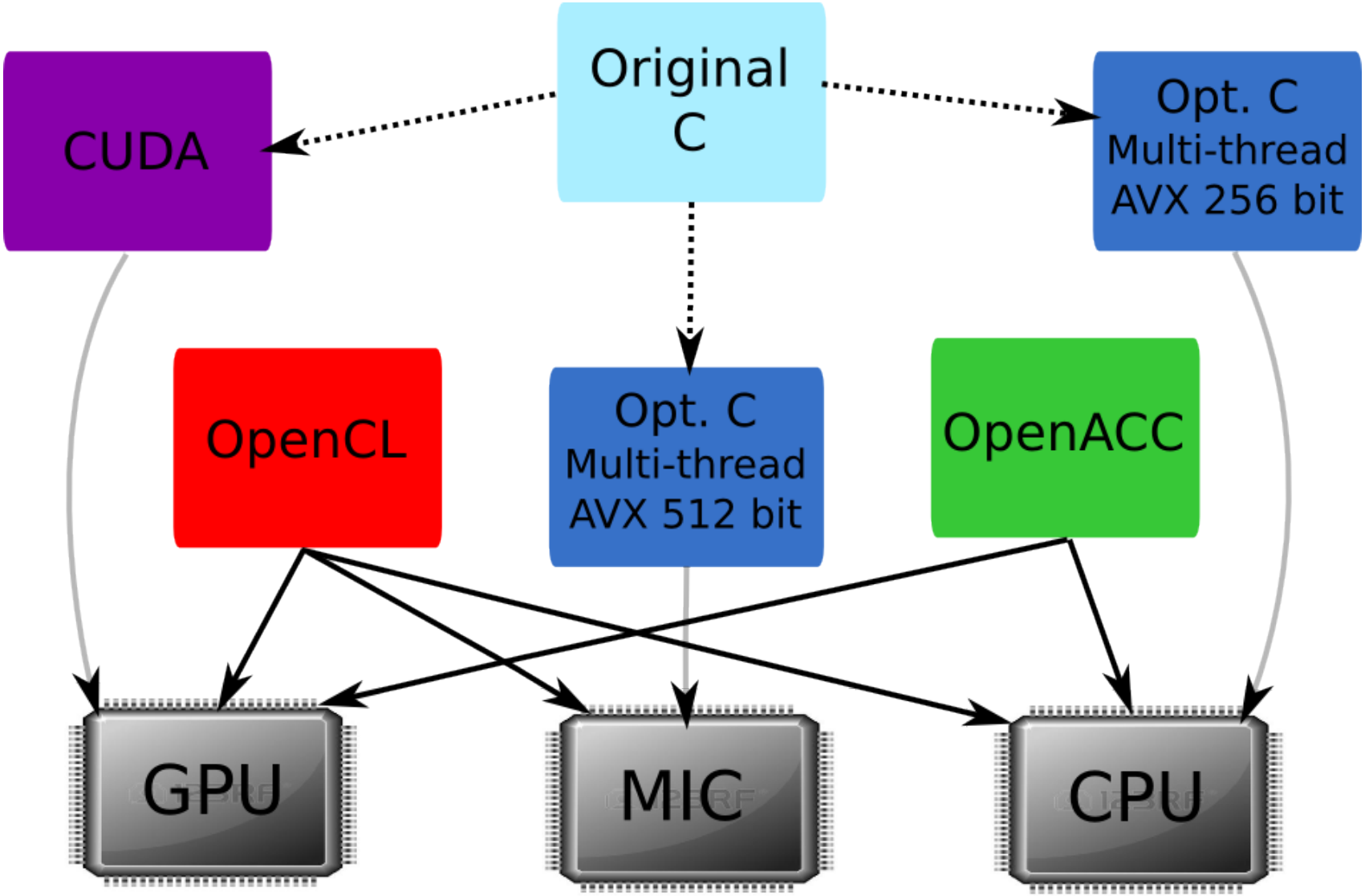
D2Q37 collision

- collision is computed at each lattice-cell
- computational intensive: for the D2Q37 model requires $\approx$ 7600 DP operations
- completely local: arithmetic operations require only the populations associate to the site

Code implementations



Code implementations



Towards an hardware independent code

- OpenCL
  - Framework for writing programs that execute across heterogeneous platforms (CPUs, GPUs, MICs, FPGAs, etc.)
  - Open standard developed by the not-for-profit Khronos group, supported by Apple, Intel, AMD, (NVIDIA), etc.
  - Apparently NVIDIA do not support it anymore
- OpenACC
  - Directive based programming standard for heterogeneous parallel computing
  - Developed by Cray, CAPS, Nvidia and PGI
  - At the moment it addresses only NVIDIA GPUs and some AMD GPUs

Independence may have costs in terms of complexity and performance

Towards an hardware independent code

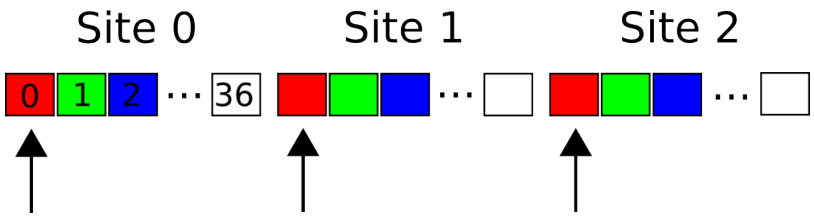
- OpenCL
  - Framework for writing programs that execute across heterogeneous platforms (CPUs, GPUs, MICs, FPGAs, etc.)
  - Open standard developed by the not-for-profit Khronos group, supported by Apple, Intel, AMD, (NVIDIA), etc.
  - Apparently NVIDIA do not support it anymore
- OpenACC
  - Directive based programming standard for heterogeneous parallel computing
  - Developed by Cray, CAPS, Nvidia and PGI
  - At the moment it addresses only NVIDIA GPUs and some AMD GPUs

Independence may have costs in terms of complexity and performance

Memory layout for LB : AoS vs SoA

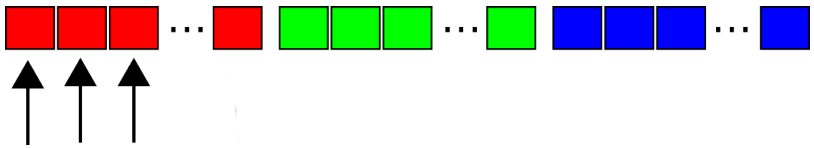
AoS: corresponding populations of different sites are interleaved, causing strided memory-access and leading to coalescing issues.

```
//lattice stored as AoS:
typedef struct {
  double  p1; // population 1
  double  p2; // population 2
   ...
  double p37; // population 37
} pop_t;

pop_t lattice2D[SIZEX*SIZEY];
```

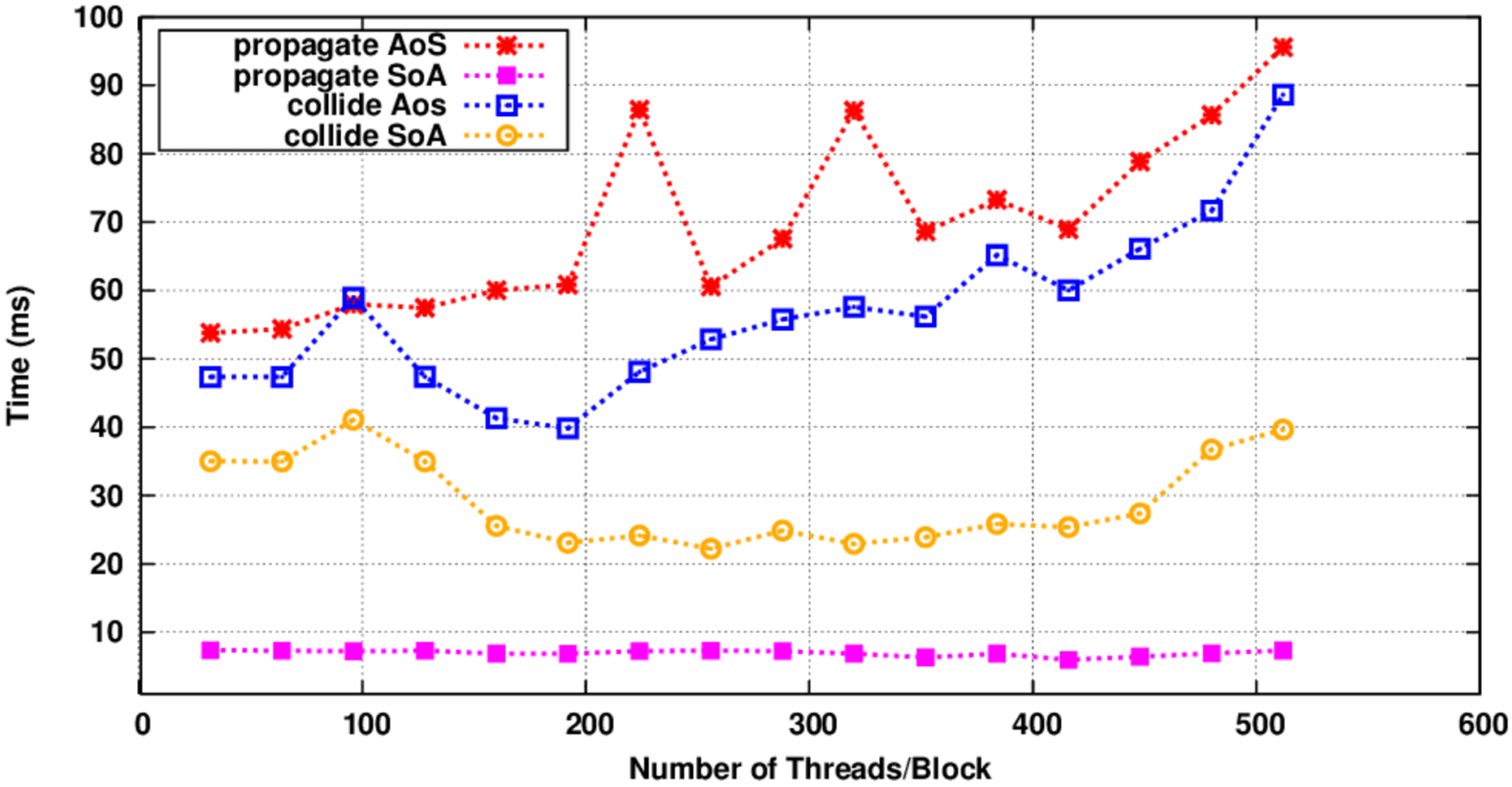


SoA: corresponding populations of different sites are allocated at contiguous memory addresses, enabling coalescing of accesses, and making use of full memory bandwidth.

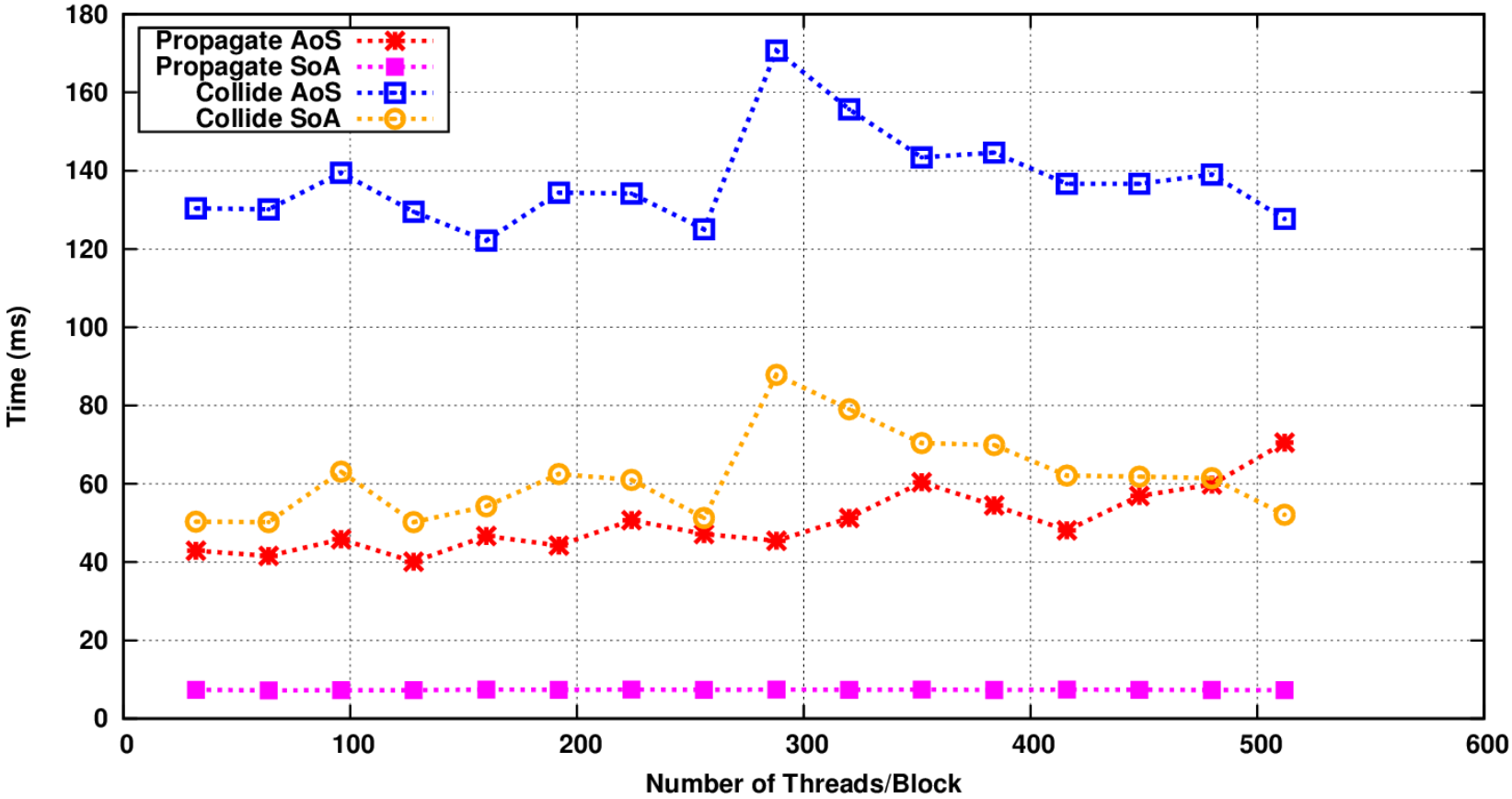
```
//lattice stored as SoA:
typedef struct {
  double  p1[SIZEX*SIZEY]; // pop 1 array
  double  p2[SIZEX*SIZEY]; // pop 2 array
   ...
  double p37[SIZEX*SIZEY]; // pop 37 array
} pop_t;

pop_t lattice2D;
```



AoS vs SoA CUDA



AoS vs SoA OpenACC



OpenCL/“CUDA” example

Propagate device function:

```
__kernel void propagate(__global const data_t* prv, __global data_t* nxt) {

  int ix,           // Work-item index along the X dimension.
      iy,           // Work-item index along the Y dimension.
      site_i;       // Index of current site.

  // Sets the work-item indices (Y is used as the fastest dimension).
  ix = (int) get_global_id(1);
  iy = (int) get_global_id(0);

  site_i = (HX+3+ix)*NY + (HY+iy);

  nxt[           site_i] = prv[           site_i - 3*NY + 1];
  nxt[   NX*NY + site_i] = prv[   NX*NY + site_i - 3*NY    ];
  nxt[ 2*NX*NY + site_i] = prv[ 2*NX*NY + site_i - 3*NY - 1];
  nxt[ 3*NX*NY + site_i] = prv[ 3*NX*NY + site_i - 2*NY + 2];
  nxt[ 4*NX*NY + site_i] = prv[ 4*NX*NY + site_i - 2*NY + 1];
  nxt[ 5*NX*NY + site_i] = prv[ 5*NX*NY + site_i - 2*NY    ];
  nxt[ 6*NX*NY + site_i] = prv[ 6*NX*NY + site_i - 2*NY - 1];

  ...
```

OpenACC example

Propagate function:

```
inline void propagate(const data_t* restrict prv, data_t* restrict nxt ) {

  int ix, iy, site_i;

  #pragma acc kernels present(prv) present(nxt)
  #pragma acc loop gang independent
  for ( ix=HX; ix < (HX+SIZEX); ix++) {
    #pragma acc loop vector independent
    for ( iy=HY; iy<(HY+SIZEY); iy++) {

      site_i = (ix*NY) + iy;

      nxt[           site_i] = prv[           site_i - 3*NY + 1];
      nxt[   NX*NY + site_i] = prv[   NX*NY + site_i - 3*NY    ];
      nxt[ 2*NX*NY + site_i] = prv[ 2*NX*NY + site_i - 3*NY - 1];
      nxt[ 3*NX*NY + site_i] = prv[ 3*NX*NY + site_i - 2*NY + 2];
      nxt[ 4*NX*NY + site_i] = prv[ 4*NX*NY + site_i - 2*NY + 1];
      nxt[ 5*NX*NY + site_i] = prv[ 5*NX*NY + site_i - 2*NY    ];
      nxt[ 6*NX*NY + site_i] = prv[ 6*NX*NY + site_i - 2*NY - 1];

 ...
```

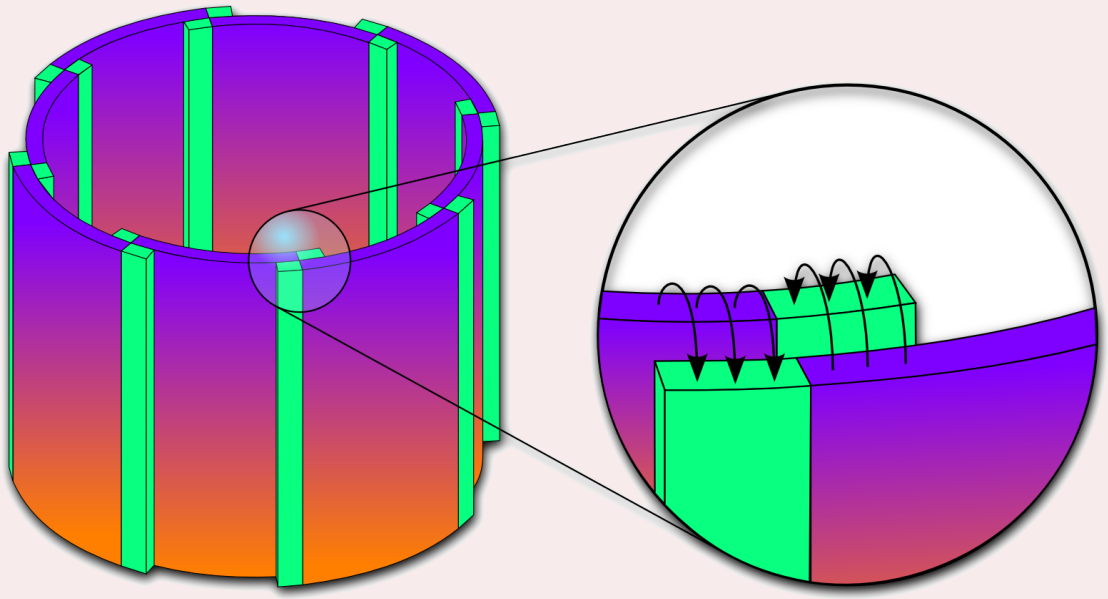
Performance portability

	Tesla K40			Xeon-Phi 7120	E5-2699-v3	
Code Version	CUDA	OCL	OACC	OCL	1CPU C	2CPU C
$T_{\text{Pbc+Prop}}$ [msec]	13.78	15.80	13.91	30.46	120.71	61.40
$\mathcal{E}_p$	59%	51%	58%	22%	29%	28%
T_{Bc} [msec]	4.42	6.41	2.76	3.20	1.62	0.80
T_{Collide} [msec]	39.86	136.93	78.65	72.79	136.24	67.95
$\mathcal{E}_c$	45%	13%	23%	34%	34%	34%
T_{WC}/iter [msec]	58.07	159.14	96.57	106.45	259.79	131.88
MLUPS	68	25	41	37	15	30

Table: Performance comparison between NVIDIA K40 GPU, Intel Xeon-Phi 7120, single and dual Intel multi-core CPUs (Haswell-v3 micro architecture); the lattice size is 1920×2048 points.

Multi-GPU implementation using MPI

Before each iteration, processes swap halos with neighbours

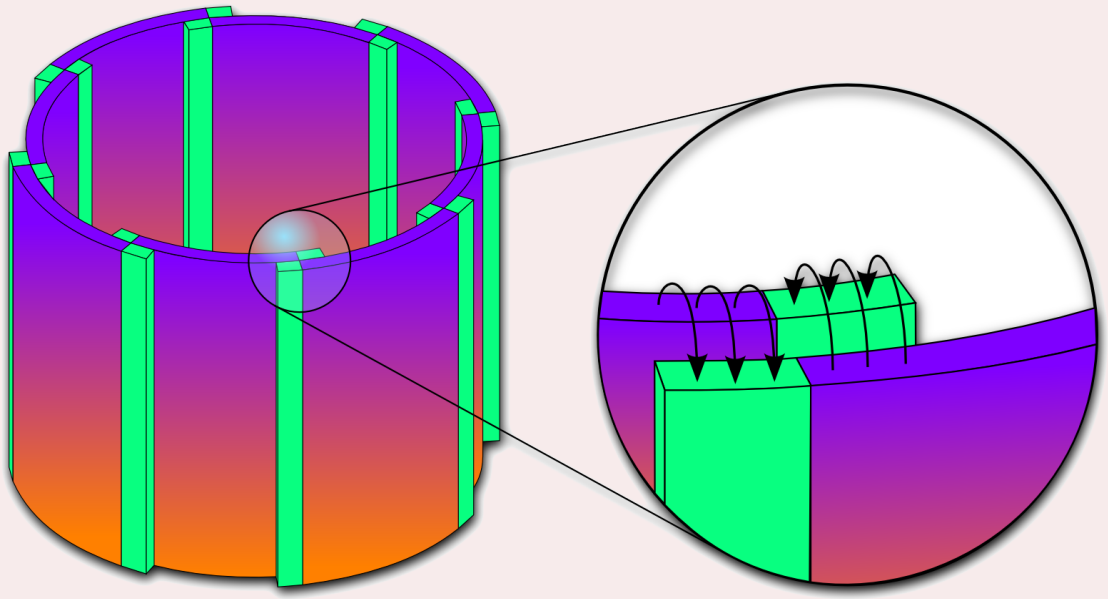


Propagate can not start before halo exchange is completed...

...on sites which are near to the halos!

Multi-GPU implementation using MPI

Before each iteration, processes swap halos with neighbours



Propagate can not start before halo exchange is completed...

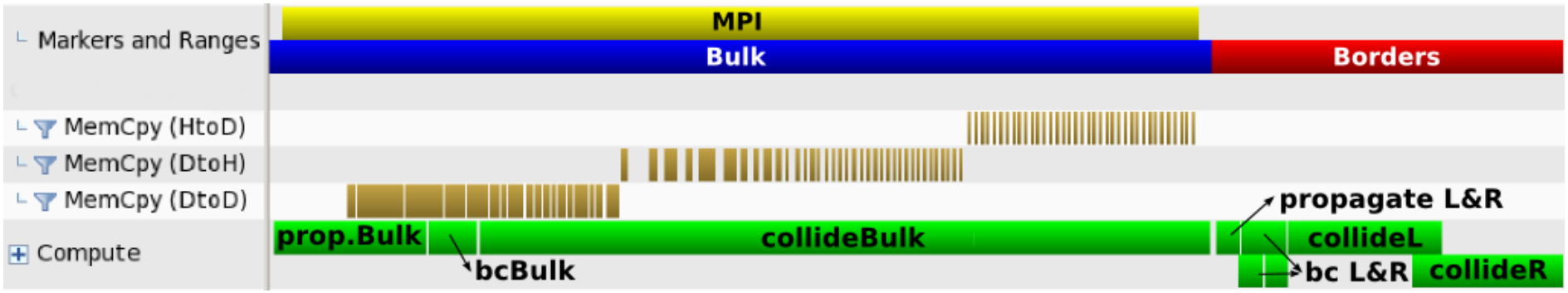
...on sites which are near to the halos!

Overlapping communications with Bulk processing

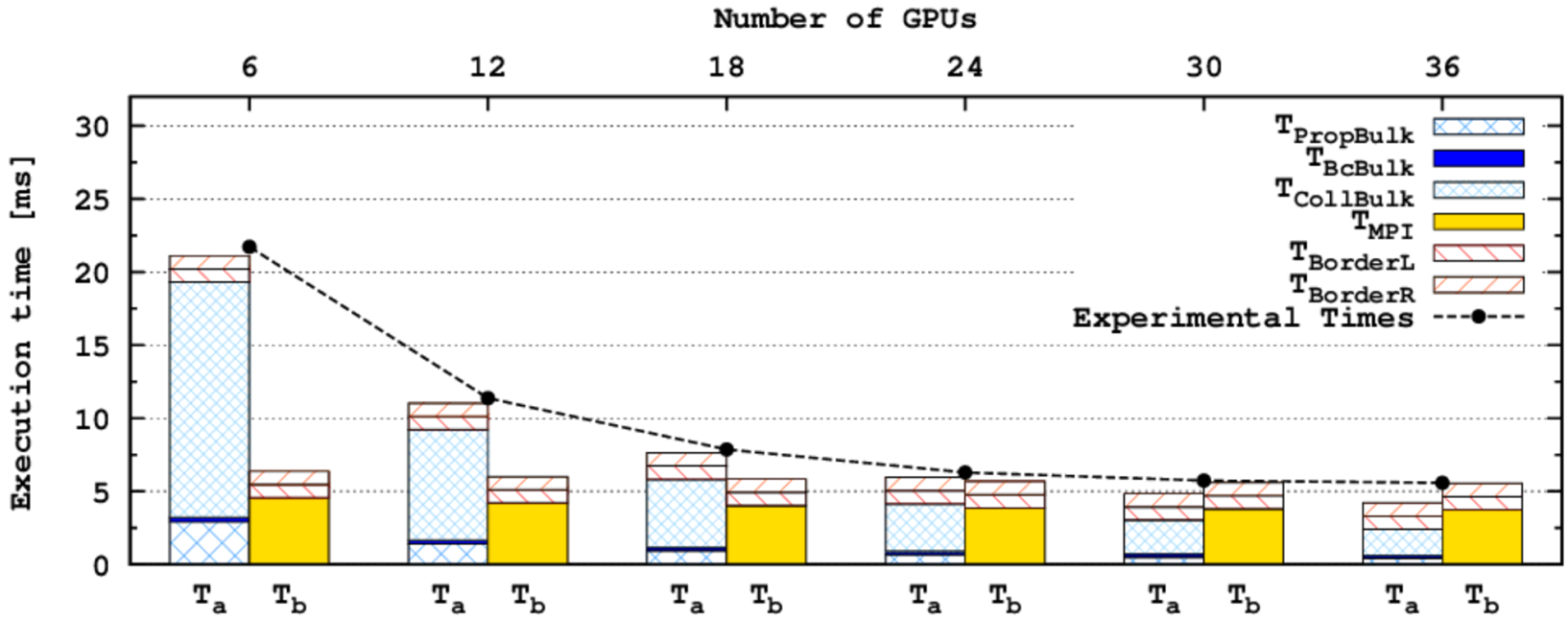
```
// processing of bulk
propagateBulk( f2, f1 ); // async execution on queue (1)
bcBulk( f2, f1 );        // async execution on queue (1)
collideInBulk( f2, f1 ); // async execution on queue (1)

#pragma acc host_data use_device(f2) {
  for ( pp = 0; pp < 37; pp++ ) {
    MPI_Sendrecv ( &(f2[...]), 3*NY, ... );
    MPI_Sendrecv ( &(f2[...]), 3*NY, ... );
  } }

// processing of the three leftmost columns
propagateL( f2, f1 );    // async execution on queue (2)
bcL( f2, f1 );           // async execution on queue (2)
collideL( f1, f2 );      // async execution on queue (2)

// processing of the three rightmost columns
propagateR( f2, f1 );    // async execution on queue (3)
bcR( f2, f1  );          // async execution on queue (3)
collideR( f1, f2  );     // async execution on queue (3)
```

Overlapping communications with Bulk processing

```
// processing of bulk
propagateBulk( f2, f1 ); // async execution on queue (1)
bcBulk( f2, f1 );        // async execution on queue (1)
collideInBulk( f2, f1 ); // async execution on queue (1)

#pragma acc host_data use_device(f2) {
  for ( pp = 0; pp < 37; pp++ ) {
    MPI_Sendrecv ( &(f2[...]), 3*NY, ... );
    MPI_Sendrecv ( &(f2[...]), 3*NY, ... );
  } }

// processing of the three leftmost columns
propagateL( f2, f1 );    // async execution on queue (2)
bcL( f2, f1 );           // async execution on queue (2)
collideL( f1, f2 );      // async execution on queue (2)

// processing of the three rightmost columns
propagateR( f2, f1 );    // async execution on queue (3)
bcR( f2, f1  );          // async execution on queue (3)
collideR( f1, f2  );     // async execution on queue (3)
```



Scalability Model



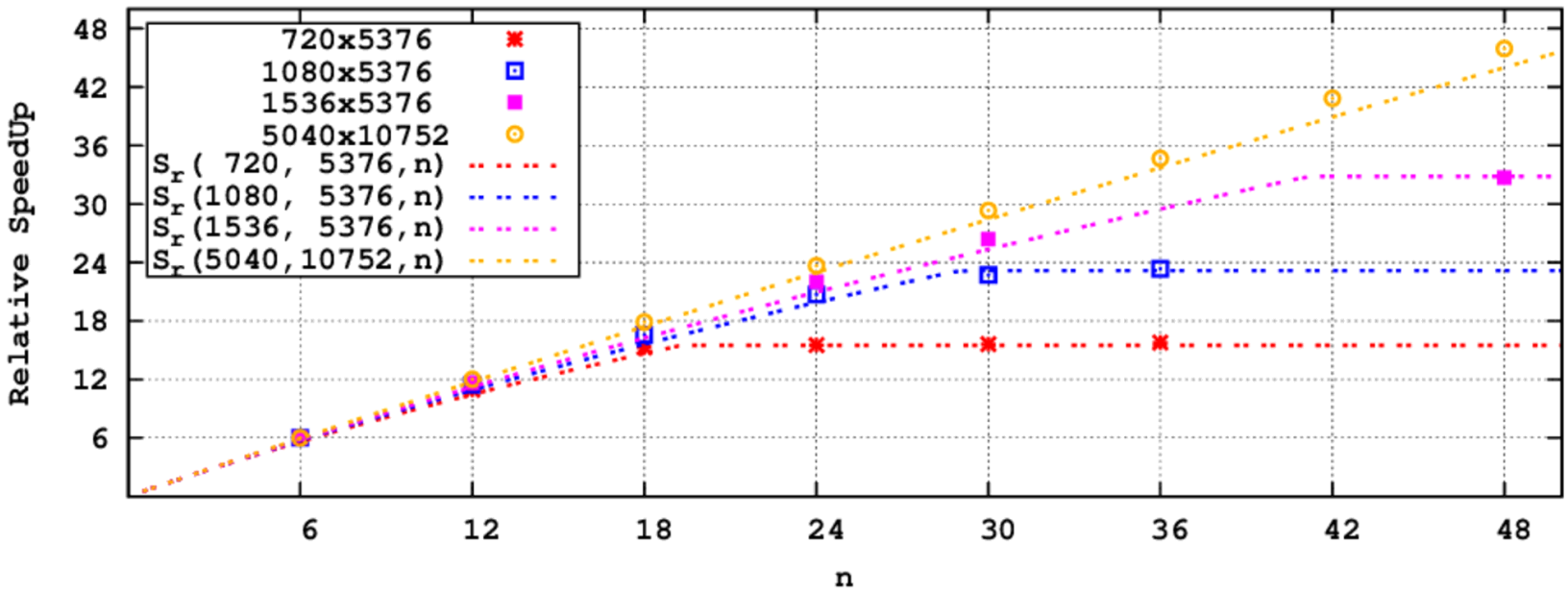
We model the execution time of the whole program as $T \approx \max\{T_a, T_b\}$

$$T_a = T_{\text{bulk}} + T_{\text{borderL}} + T_{\text{borderR}} \ , \qquad T_b = T_{\text{MPI}} + T_{\text{borderL}} + T_{\text{borderR}}$$

Scalability Model to predict Strong Scaling

Assumption: bulk processing is proportional to $(L_x \times L_y)$; boundary conditions scale as L_x; communication and borders processing scales as L_y; so, on n GPUs:

$$T(L_x, L_y, n) = \max\left\{\alpha\frac{L_x}{n}L_y + \beta\frac{L_x}{n}, \quad \gamma L_y\right\} + \delta L_y$$



Conclusion

OpenACC

- grants code portability (at the moment across NVIDIA and AMD GPUs)
- permits good code maintainability, thanks to minimal code modification wrt plain C version
- provides good performance portability
- allows for easy multi-node / multi-accelerator implementation
- will be probably incorporated in the OpenMP standard in the future

Thanks for Your attention