Performance and portability of accelerated lattice Boltzmann applications with OpenACC

E. Calore S. F. Schifano R. Tripiccione

INFN Ferrara and Università degli Studi di Ferrara, Italy

Parma - 9th June, 2016

Outline

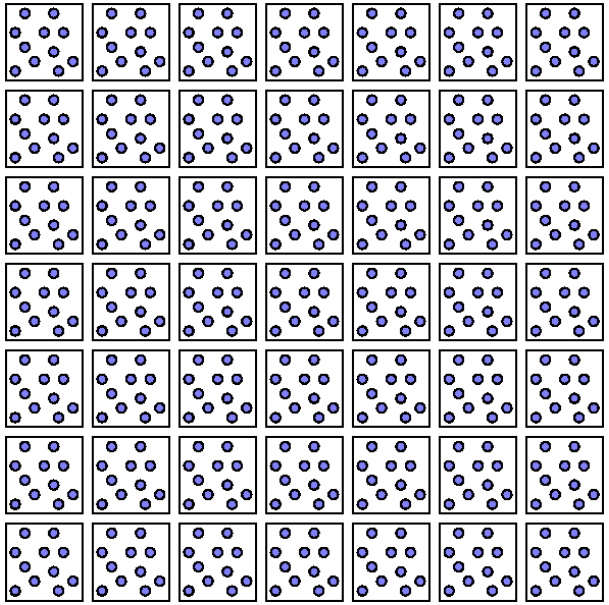
The D2Q37 Lattice Boltzmann Model

- Lattice Boltzmann method (LBM) is a class of computational fluid dynamics (CFD) methods
- LBM methods simulate a discrete **Boltzmann** equation, which under certain conditions, reduce to the **Navier-Stokes** equation
- **virtual particles** called **populations** arranged at edges of a discrete and regular grid are used to simulate a synthetic and simplified dynamics
- the interaction is implemented by two main functions applied to the virtual particles: **propagation** and **collision**
- 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();
```



Embarassing parallelism

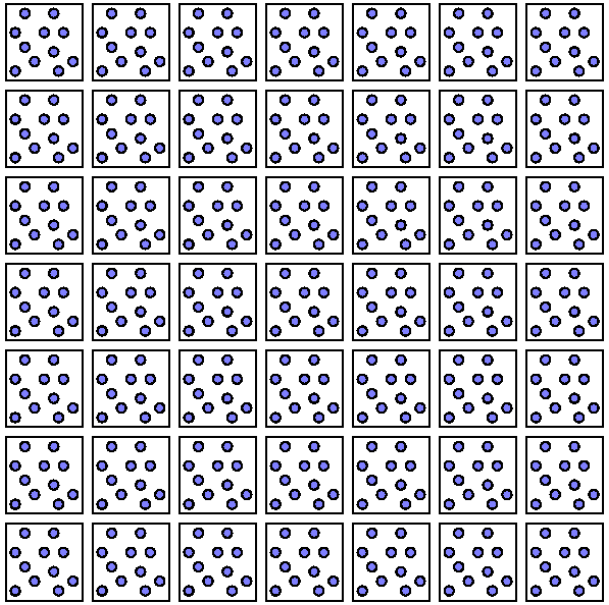
Sites can be processed in parallel applying in sequence propagate and collide

Challenge

Design an efficient implementation to exploit a large fraction of available peak performance on different heterogeneous architectures

Computational Scheme of LBM

```
foreach time-step

  foreach lattice-point
    propagate();

  foreach boundary-point
    bc();

  foreach lattice-point
    collide();
```



Embarassing parallelism

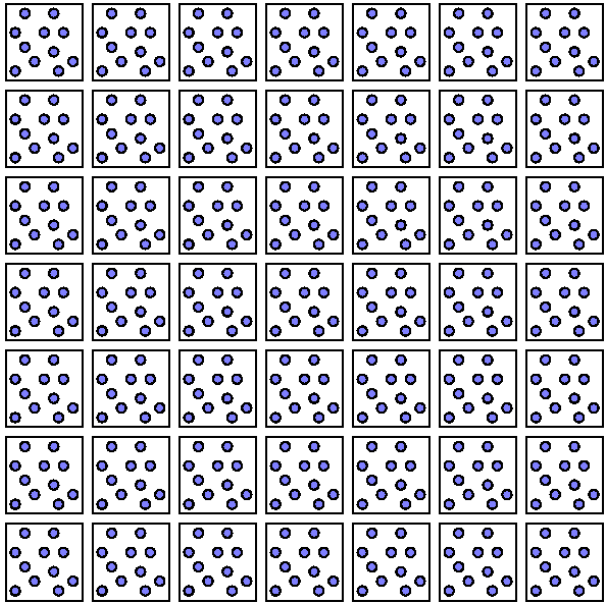
Sites can be processed in parallel applying in sequence propagate and collide

Challenge

Design an efficient implementation to exploit a large fraction of available peak performance on different heterogeneous architectures

Computational Scheme of LBM

```
foreach time-step

  foreach lattice-point
    propagate();

  foreach boundary-point
    bc();

  foreach lattice-point
    collide();
```



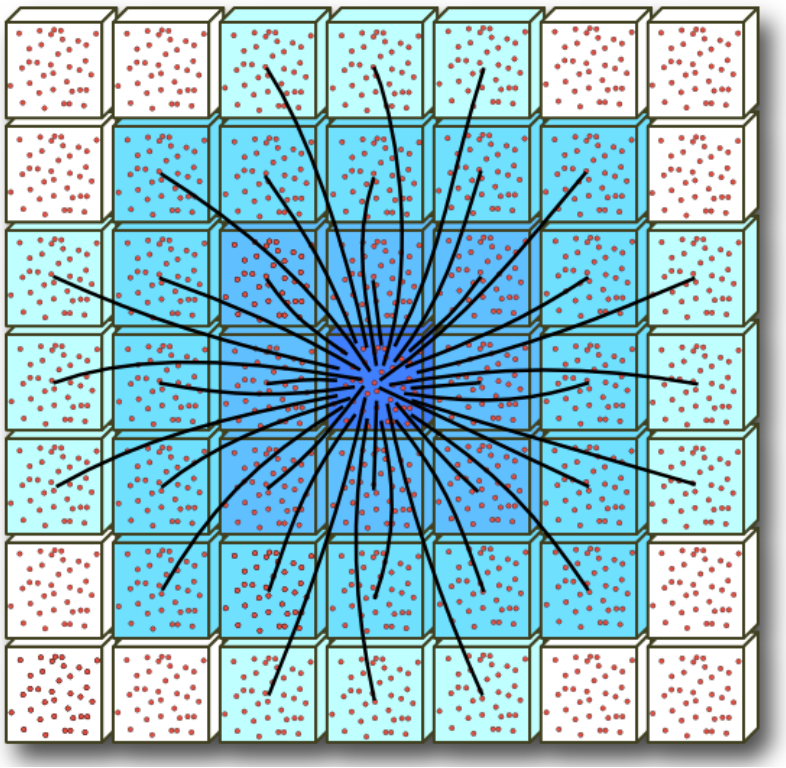
Embarassing parallelism

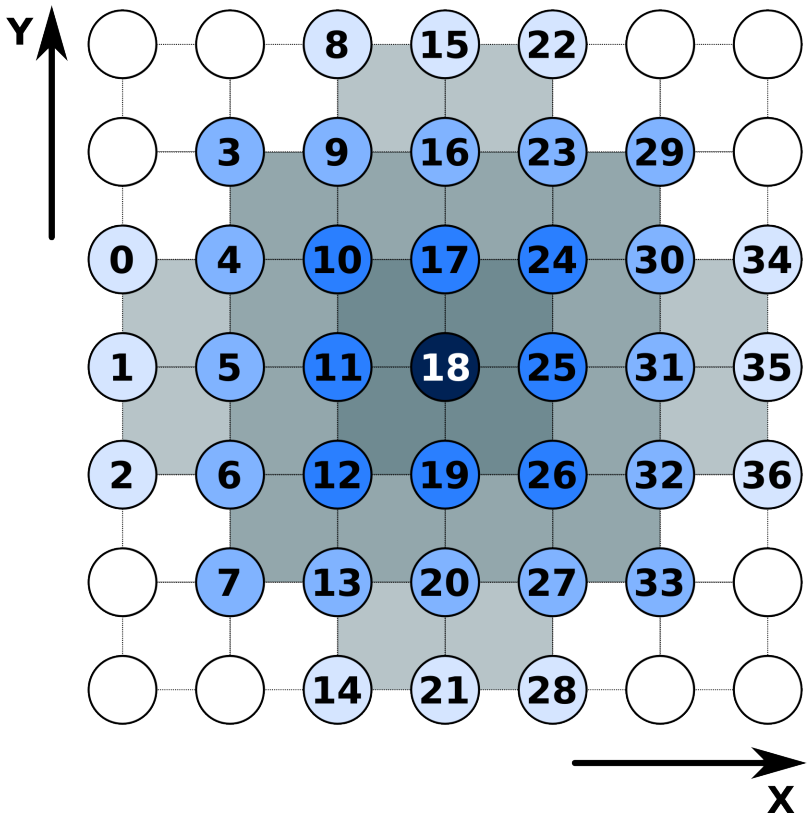
Sites can be processed in parallel applying in sequence propagate and collide

Challenge

Design an efficient implementation to exploit a large fraction of available peak performance on different heterogeneous architectures

D2Q37: propagation scheme

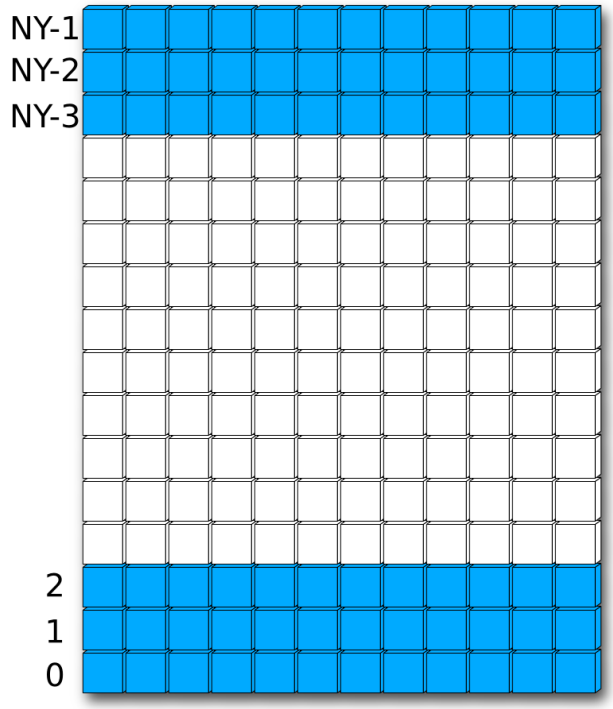




- require to access neighbours cells at distance 1,2, and 3,
- generate memory-accesses with **sparse** addressing patterns.

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

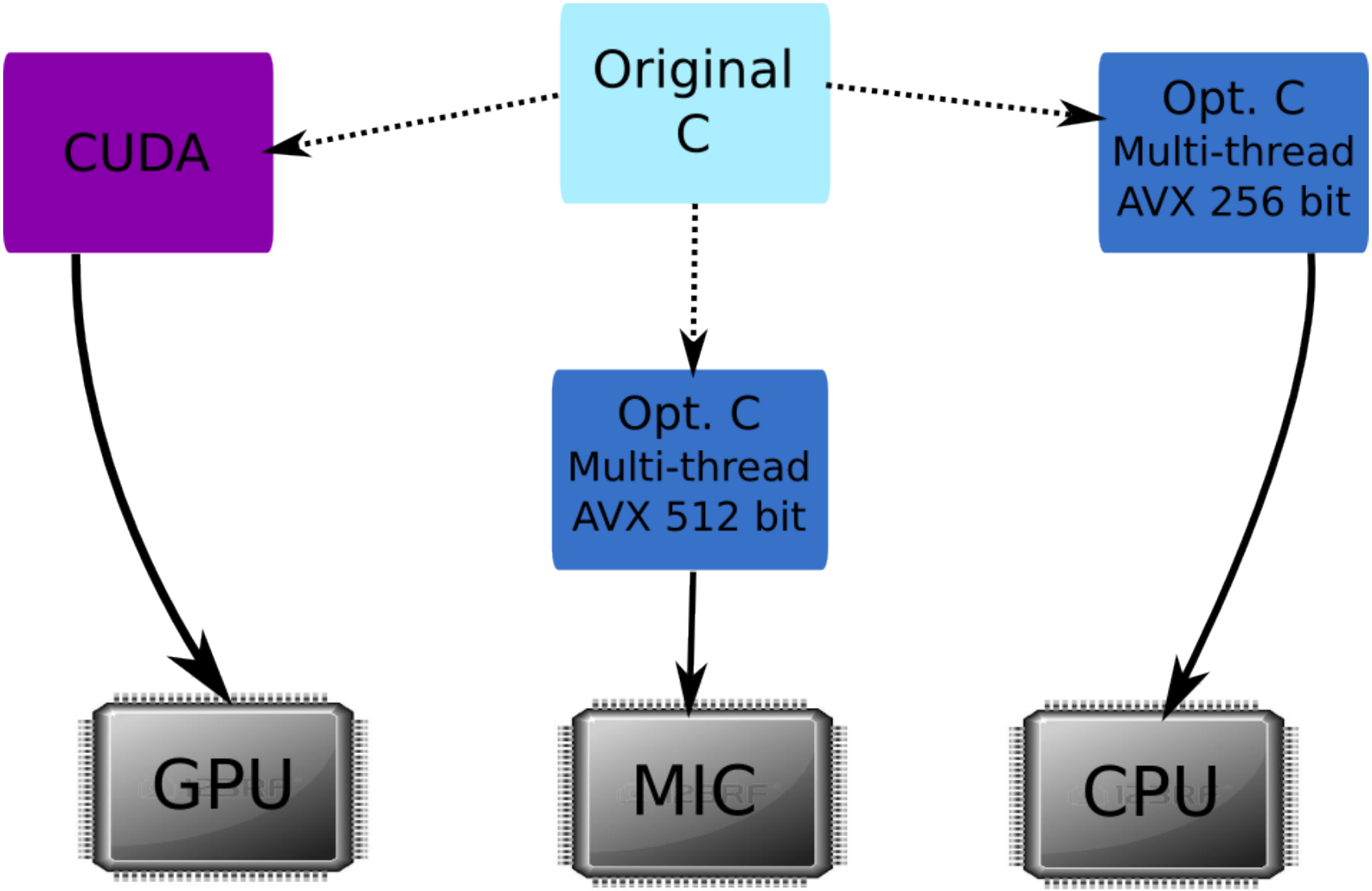


This step (bc) is computed before the collision step.

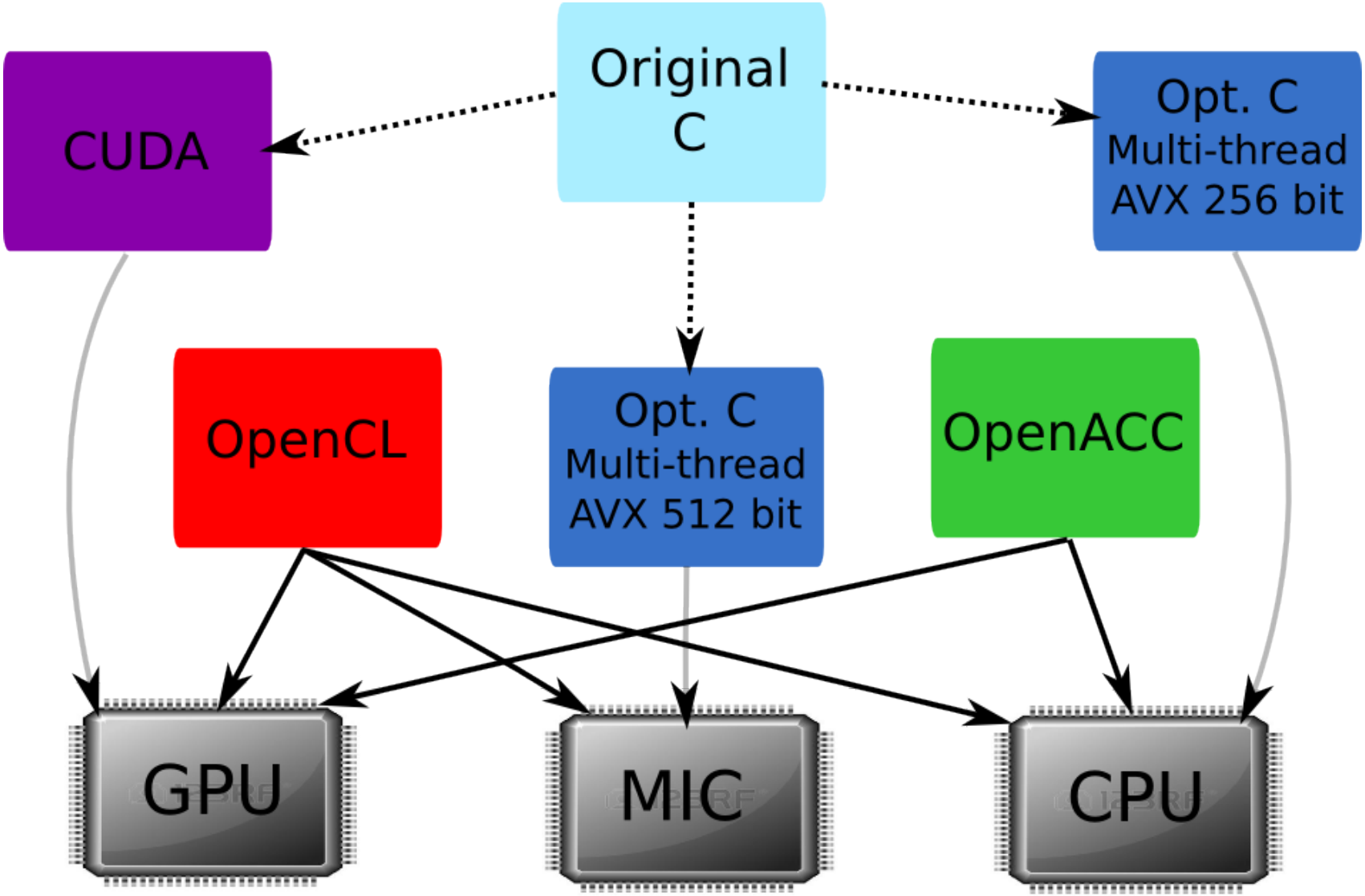
D2Q37 collision

- 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



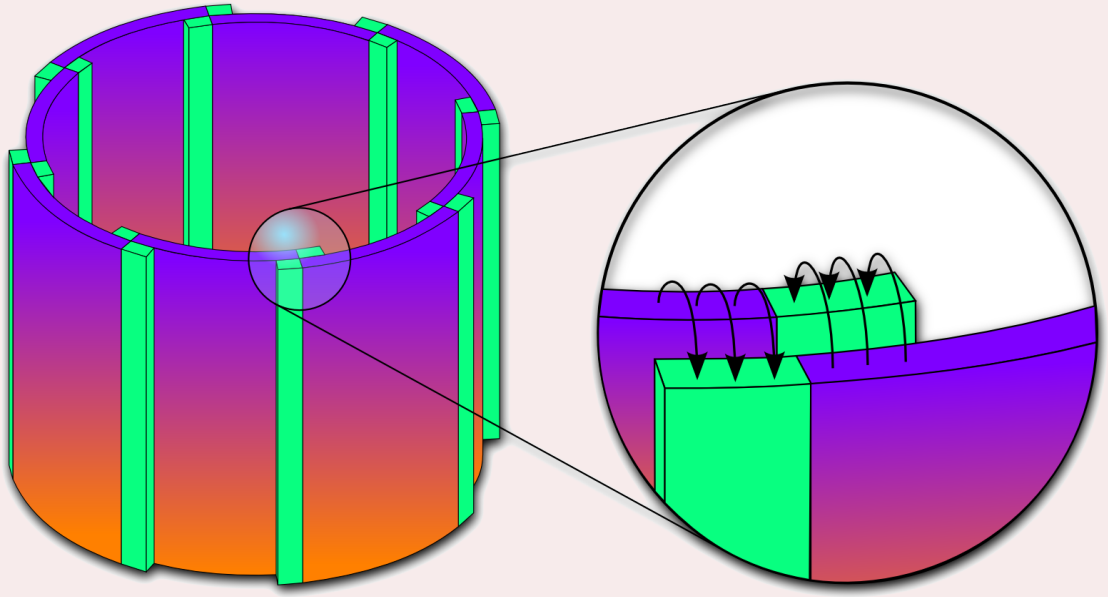
OpenACC example

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];

 ...
```

Multi-GPU implementation using MPI

Before each iteration, processes swap halos with neighbours

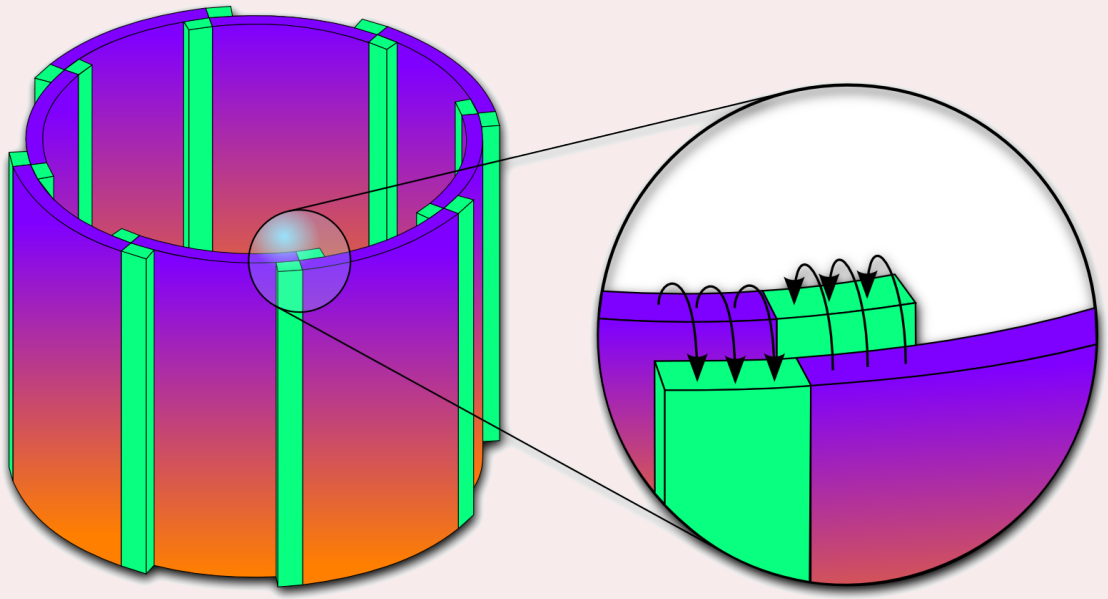


Propagate can not start before halo exchange is completed...

...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...

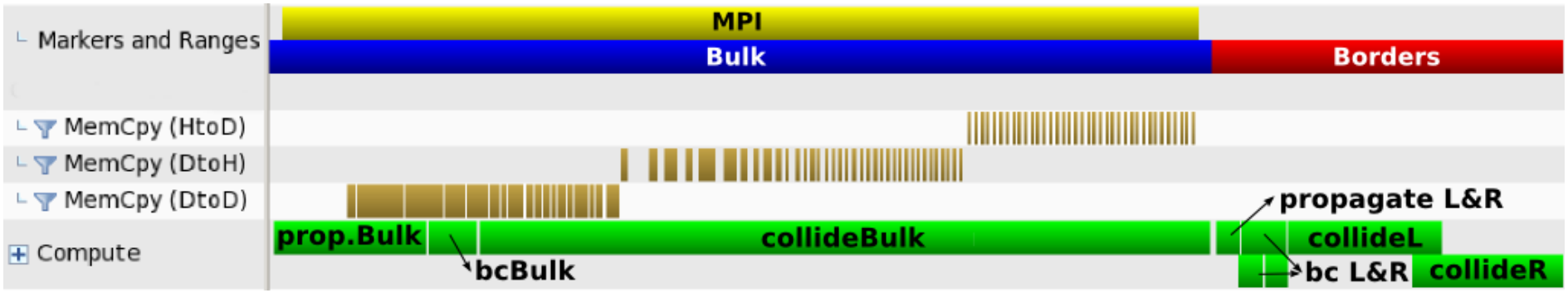
...on sites which are near to the halos!

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

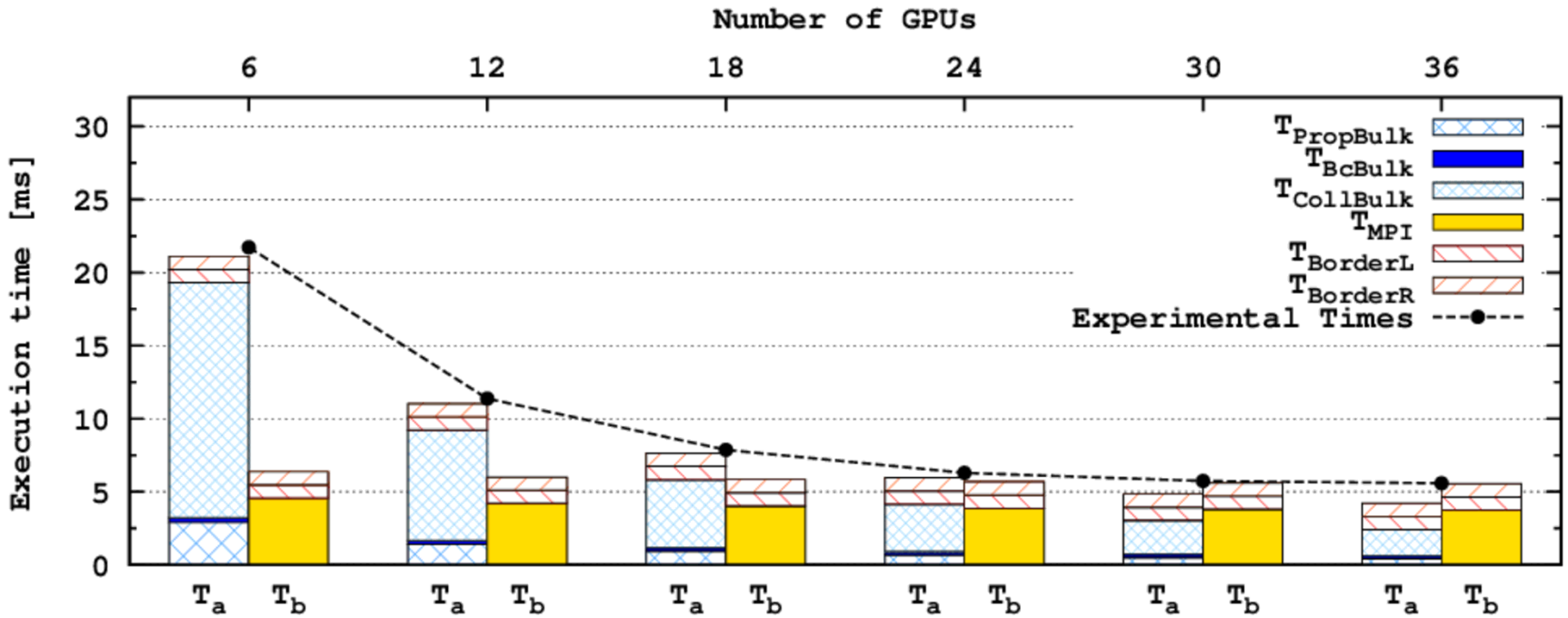
We model the execution time of the whole program as $T \approx \max\{T_a, T_b\}$

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

Scalability Model

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

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

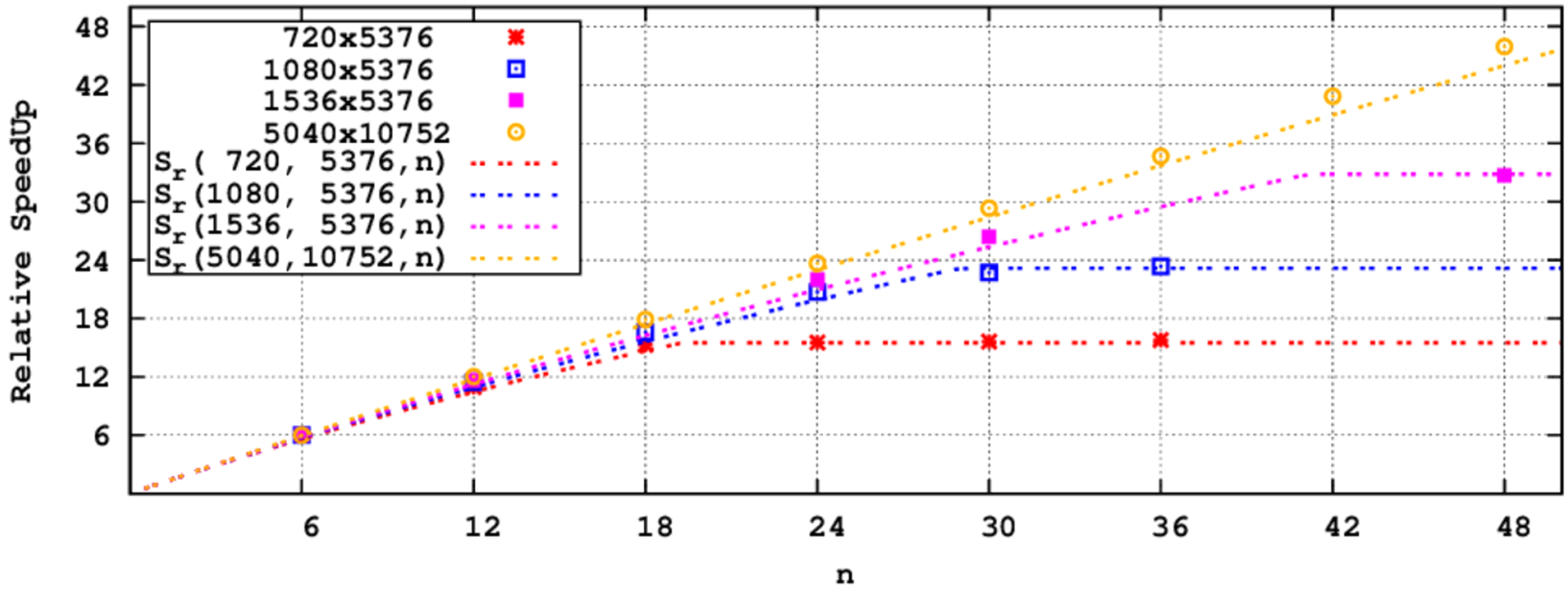


A General Model to predict Strong Scaling

Given that: bulk processing is proportional to $(L_x \times L_y)$; boundary conditions scale as L_x; communication and borders processing scales as L_y;

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

$$S_r(L_x, L_y, n) = \frac{T(L_x, L_y, 1)}{T(L_x, L_y, n)}$$



Straightforward portability

Target architecture for the compilation is specified by appropriate options to the PGI compiler:

- `-ta=nvidia` for NVIDIA GPUs
- `-ta=radeon` for AMD GPUs
- `-ta=multicore` for *x*86 multicore CPUs (since PGI 15.9)

But, what about performances?

Straightforward portability

Target architecture for the compilation is specified by appropriate options to the PGI compiler:

- `-ta=nvidia` for NVIDIA GPUs
- `-ta=radeon` for AMD GPUs
- `-ta=multicore` for *x*86 multicore CPUs (since PGI 15.9)

But, what about performances?

Architectures specs and results

	Intel Xeon	NVIDIA K80	AMD S9150
processor codename	E5-2630 v3	GK210	Hawaii XT
#physical-cores	8	13 x2	44
#logical-cores	16	2496 x2	2816
nominal clock Freq. (GHz)	2.1	0.562	0.900
Max Boosted clock Freq. (GHz)	2.6	0.875	–
Nominal GFLOPS (DP)	269	935 x2	2530
Boosted GFLOPS (DP)	331.56	1455 x2	–
Mem Bandwidth (GB/s)	59	240 x2	320

	E5-2630 v3	GK210	Hawaii XT
propagate [GB/s]	10	155	216
collide [MLUPS]	8	55	53

Architectures specs and results

	Intel Xeon	NVIDIA K80	AMD S9150
processor codename	E5-2630 v3	GK210	Hawaii XT
#physical-cores	8	13 x2	44
#logical-cores	16	2496 x2	2816
nominal clock Freq. (GHz)	2.1	0.562	0.900
Max Boosted clock Freq. (GHz)	2.6	0.875	–
Nominal GFLOPS (DP)	269	935 x2	2530
Boosted GFLOPS (DP)	331.56	1455 x2	–
Mem Bandwidth (GB/s)	59	240 x2	320

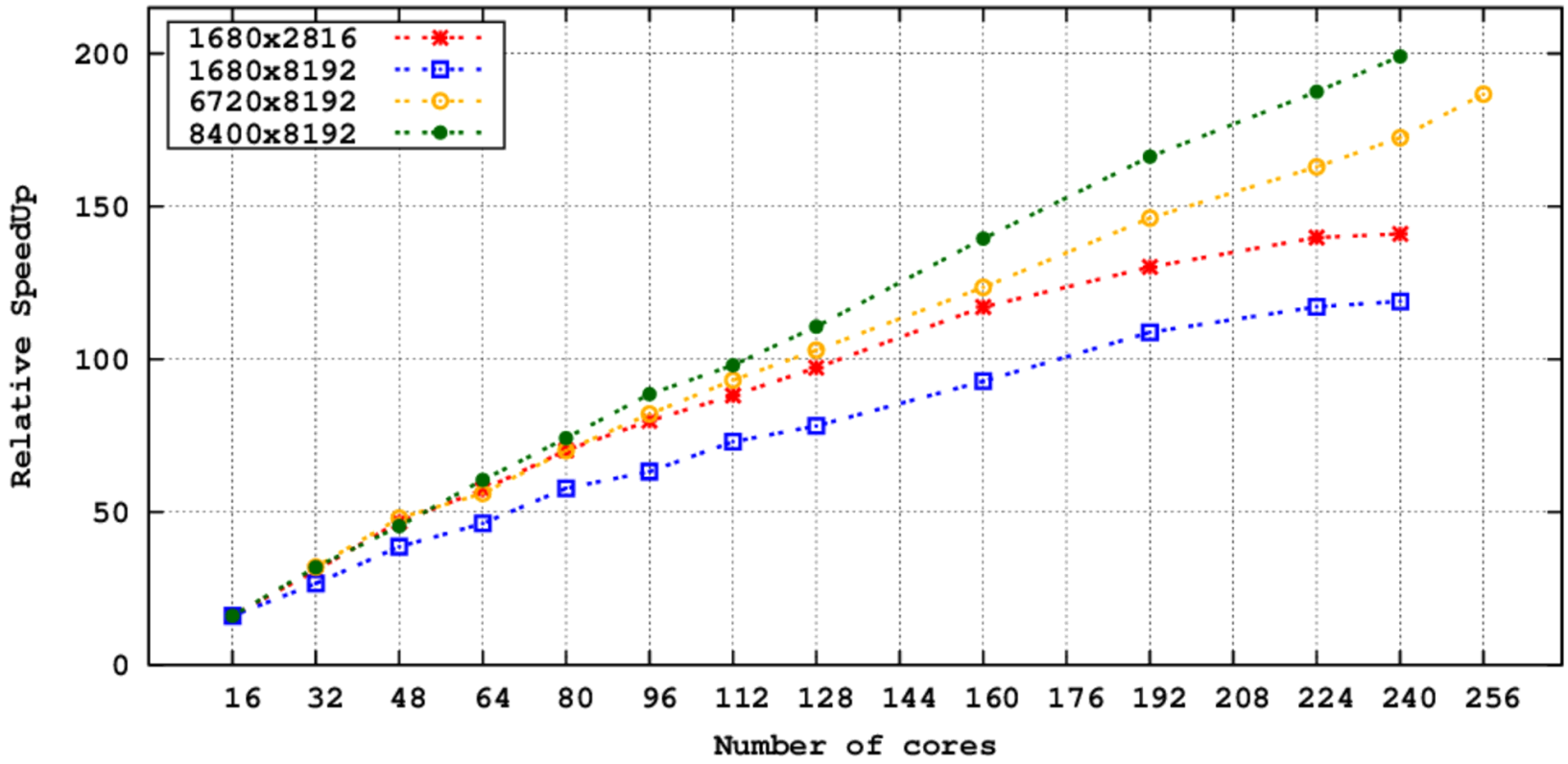
	E5-2630 v3	GK210	Hawaii XT
propagate [GB/s]	10	155	216
collide [MLUPS]	8	55	53

Using a bit of architecture specific optimizations

```
#pragma acc kernels present(prv) present(nxt)
#pragma acc loop gang independent
for ( ix=HX; ix < (HX+SIZEX); ix++) {
  site_i = (ix*NY);

  #pragma acc loop seq independent
  for ( ipop=0; ipop<37; ipop++ ) {

    #pragma acc loop vector independent
    for ( iy=HY; iy < (HY+SIZEY); iy++) {
      nxt[ipop*NX*NY+site_i+iy] = prv[ipop*NX*NY+site_i+iy+OFFSET[ipop]];
    }

  }
}
```

	E5-2630 v3	GK210	Hawaii XT
propagate v1 [GB/s]	10	155	216
propagate v2 [GB/s]	32	145	223
collide v1 [MLUPS]	8	55	53
collide v2 [MLUPS]	12	6	5

Scalability on a CPU cluster



Performance comparison of various programming frameworks on various processors.

	E5-2630 v3			GK210		Hawaii XT	
compiler model	ICC 15 Intrinsics	ICC 15 OMP	PGI 15.10 OACC	NVCC 7.5 CUDA	PGI 15.10 OACC	GCC OCL	PGI 15.10 OACC
`propagate` [GB/s]	38	32	32	154	155	232	216
$\mathcal{E}_p$	65%	54%	54%	64%	65%	73%	70%
`collide` [MLUPS]	14	11	12	117	55	76	54
`collide` [GFLOPs]	92	71	78	760	356	494	351
$\mathcal{E}_c$	28%	22%	24%	52%	24%	19%	14%
Tot perf. [MLUPS]	11.5	9.2	9.8	80.7	45.6	63.7	47.0

Conclusion

- we have successfully ported, tested and benchmarked a complete LBM code using OpenACC
- we obtained a good code maintainability, thanks to minimal code modification w.r.t. plain C version
- code is portable across different accelerators
- performances are portable with some limitations

Conclusion

- we have successfully ported, tested and benchmarked a complete LBM code using OpenACC
- we obtained a good code maintainability, thanks to minimal code modification w.r.t. plain C version
- code is portable across different accelerators
- performances are portable with some limitations

Conclusion

- we have successfully ported, tested and benchmarked a complete LBM code using OpenACC
- we obtained a good code maintainability, thanks to minimal code modification w.r.t. plain C version
- code is portable across different accelerators
- performances are portable with some limitations

Conclusion

- we have successfully ported, tested and benchmarked a complete LBM code using OpenACC
- we obtained a good code maintainability, thanks to minimal code modification w.r.t. plain C version
- code is portable across different accelerators
- performances are portable with some limitations

Thanks for Your attention

	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
MLUPS	285.32	248.86	282.72	129.10	32.98	64.84
GB/s	168.91	147.33	167.37	76.42	19.53	19.19
$\mathcal{E}_p$	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
FLOP/site	6472	6613	6504	7600	7600	7600
MLUPS	99	29	50	54	29	29
$\mathcal{E}_c$	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.

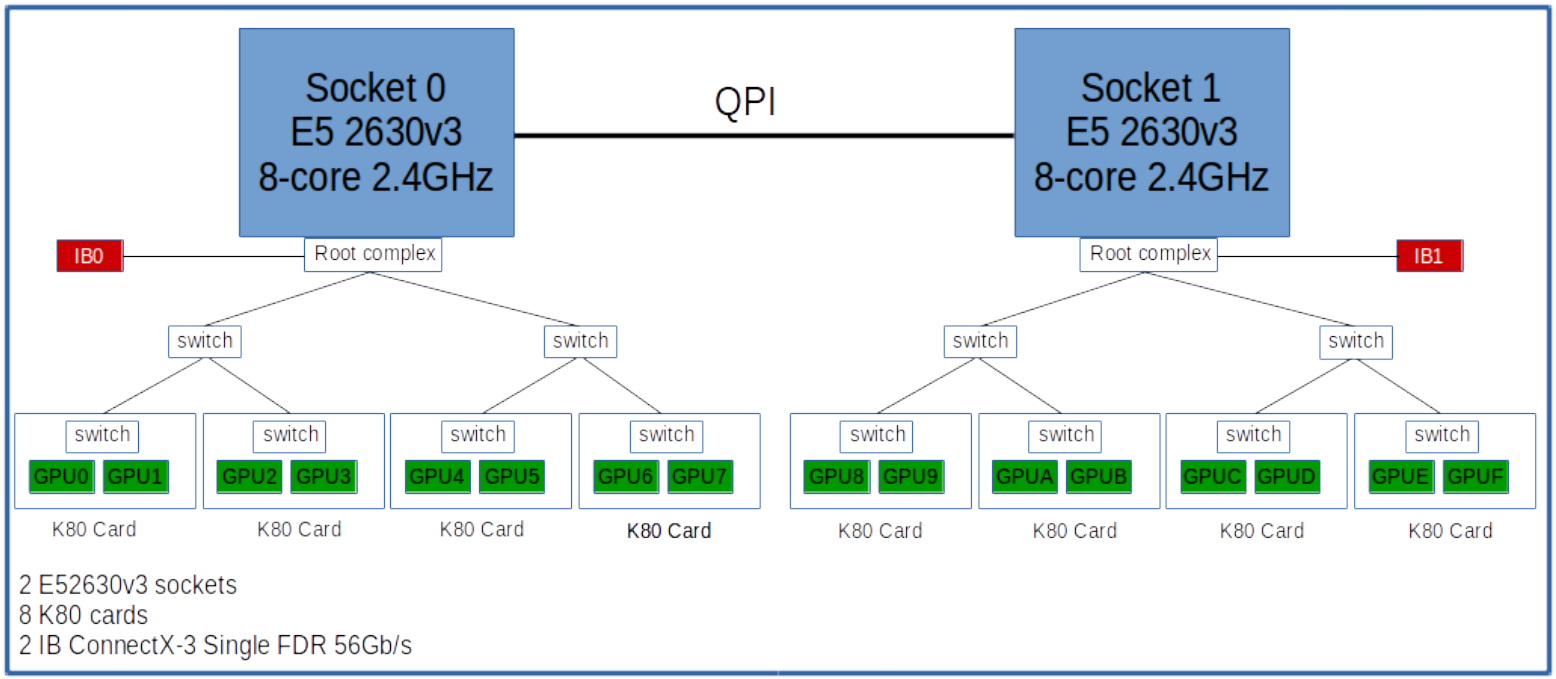
Cluster Overview

Compute nodes:

Supermicro SYS-4028GR-TR

- 2 x Intel Xeon E5-2630v3
- 8 x NVIDIA K80 (2xGPU)
- 2 x Mellanox ConnectX-3 Single FDR 56Gb/s Infiniband cards

Nodes Overview



Network Overview