Designing and Optimizing LQCD code using OpenACC

E Calore, S F Schifano, R Tripiccione

Enrico Calore

University of Ferrara and INFN-Ferrara, Italy

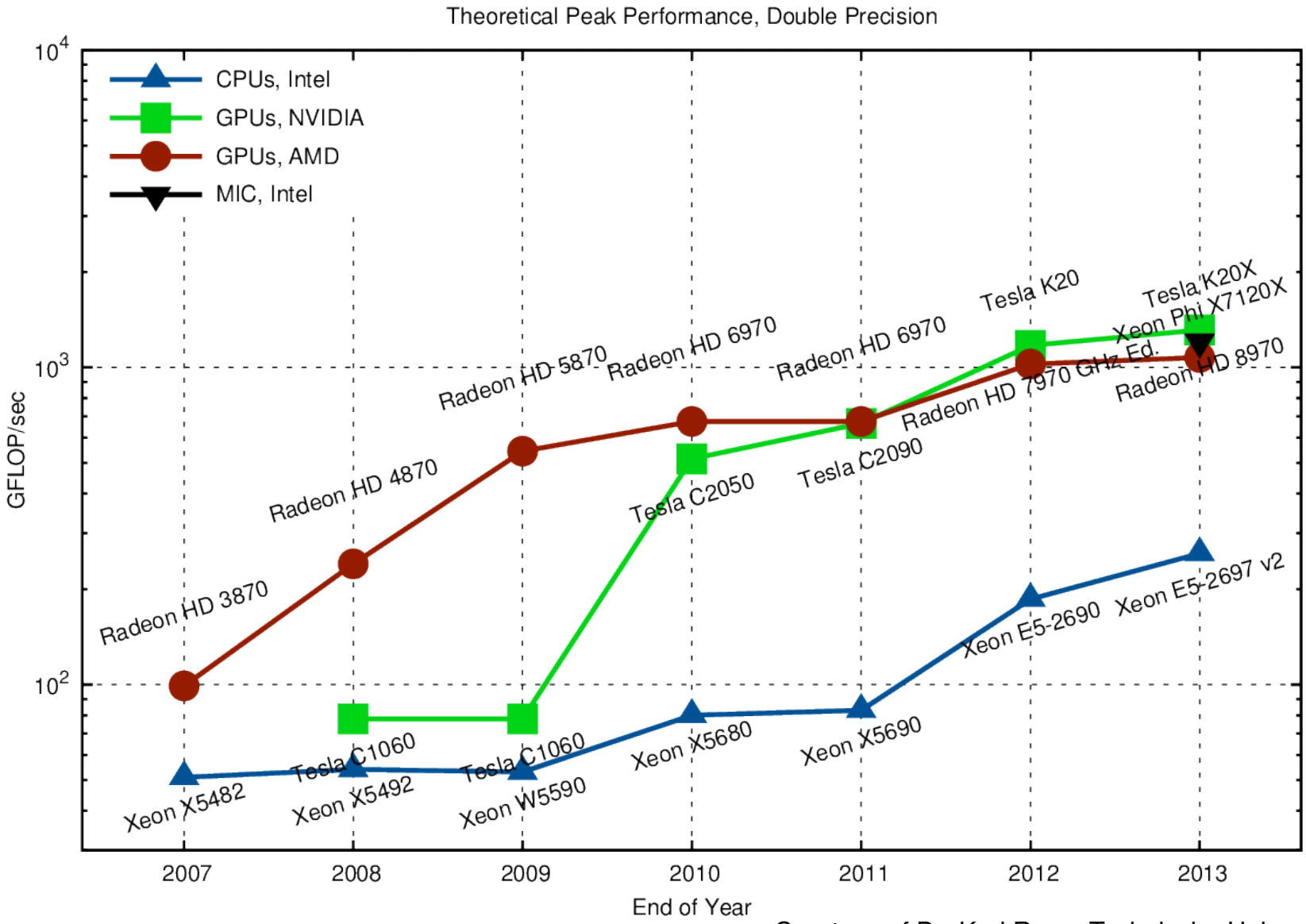
GPU Computing in High Energy Physics

Pisa, Sep. 10th, 2014

Outline

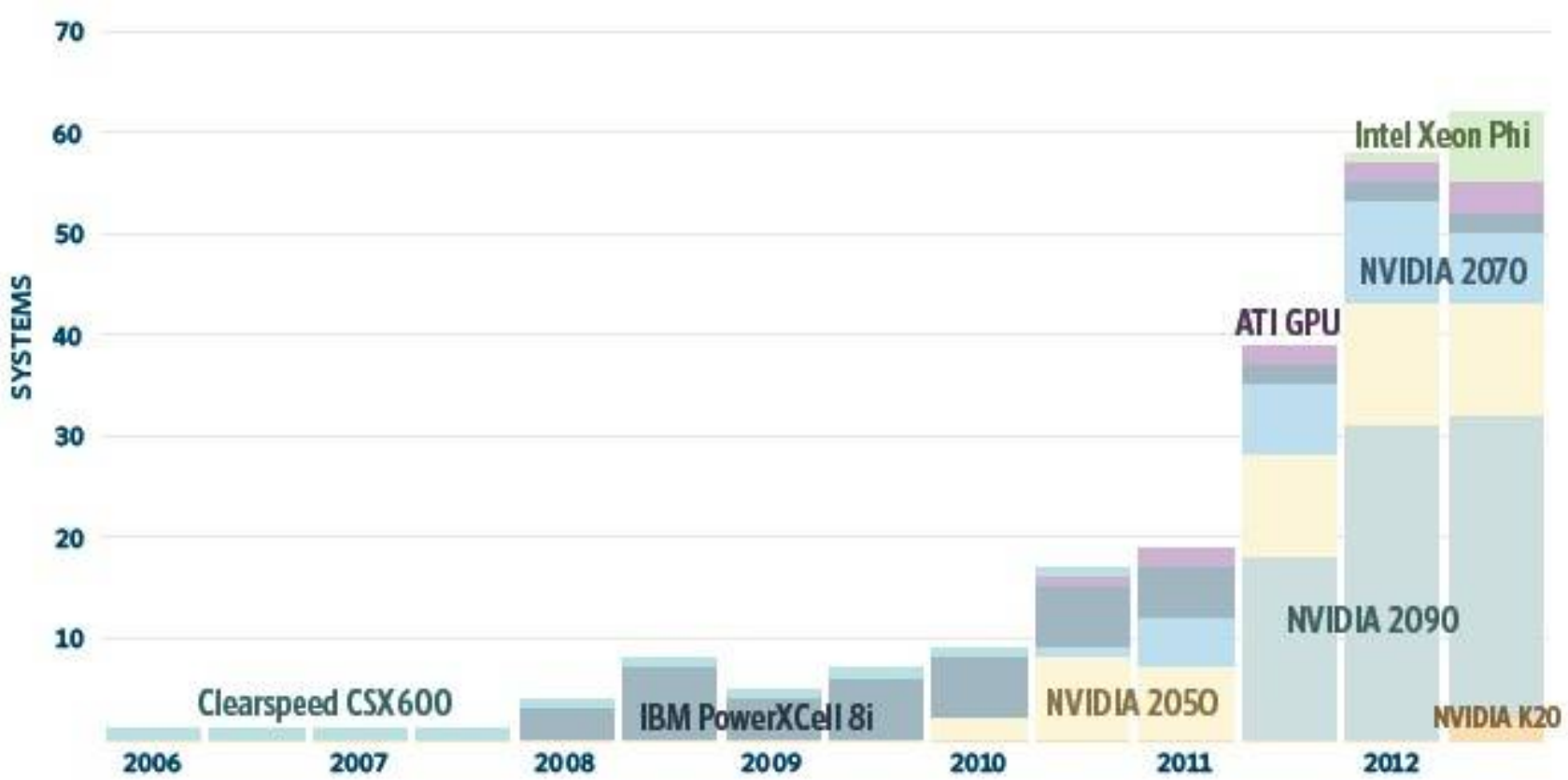
Outline

GPUs and MICs performances are growing



Courtesy of Dr. Karl Rupp, Technische Universität Wien

GPUs and MICs use in HPC is growing



Accelerator architectures in the Top500 Supercomputers

Outline

How to get our code ready for future HPC systems?

Given that:

- available parallelism in CPUs is increasing
- accelerator architectures are quickly evolving
- CPUs and Accelerators are getting closer
- is hard to predict if one architecture will prevail and, if it is the case, which one will

Code has to:

- be able to exploit hardware parallelism at different levels
- be portable across different architectures
- not be subject to (excessive) performance degradation due to its portability

OpenCL (Open Computing Language):

- The same code can be run on CPUs, GPUs, MICs, etc.
- Functions to be offloaded on the accelerator have to be explicitly programmed (as in CUDA)
- Data movements between host and accelerator has to be explicitly programmed (as in CUDA)
- NVIDIA do not support it anymore

OpenACC (for Open Accelerators):

- The same code (will probably) run on CPUs, GPUs, MICs, etc.
- Functions to be offloaded are “annotated” with `#pragma` directives
- Data movements between host and accelerator could be managed automatically or manually
- Support is still limited, but seems to be quickly growing

Why it is worth to use OpenACC

Code modifications could be minimal

- Thanks to the annotation of pre-existing C code using `#pragma` directives.
- Programming efforts needed mainly to re-organize the data structures and to efficiently design data movements.

If it will be superseded, programming efforts would not be lost

- OpenMP community is working towards the native support for accelerators in the language (maybe in several years).
- Switching between directive based languages should be just a matter of changing the `#pragma` clauses.
- Also other directive based languages would benefit from data re-organization and efficiently designed data movements.

NVIDIA is pushing for its adoption and is strongly committed to develop PGI

Outline

OpenACC example: the Saxpy function

```
{
  my_saxpy(x, y);


}
```

```
void my_saxpy(float * x, float * y) {


  #pragma acc kernels loop
  for (int i = 0; i < N; ++i)
    y[i] = a*x[i] + y[i];

}
```

OpenACC code computing a *saxpy* function on vectors *x* and *y*. *#pragma* clause identify the region to run on the accelerator.

OpenACC example: the Saxpy function

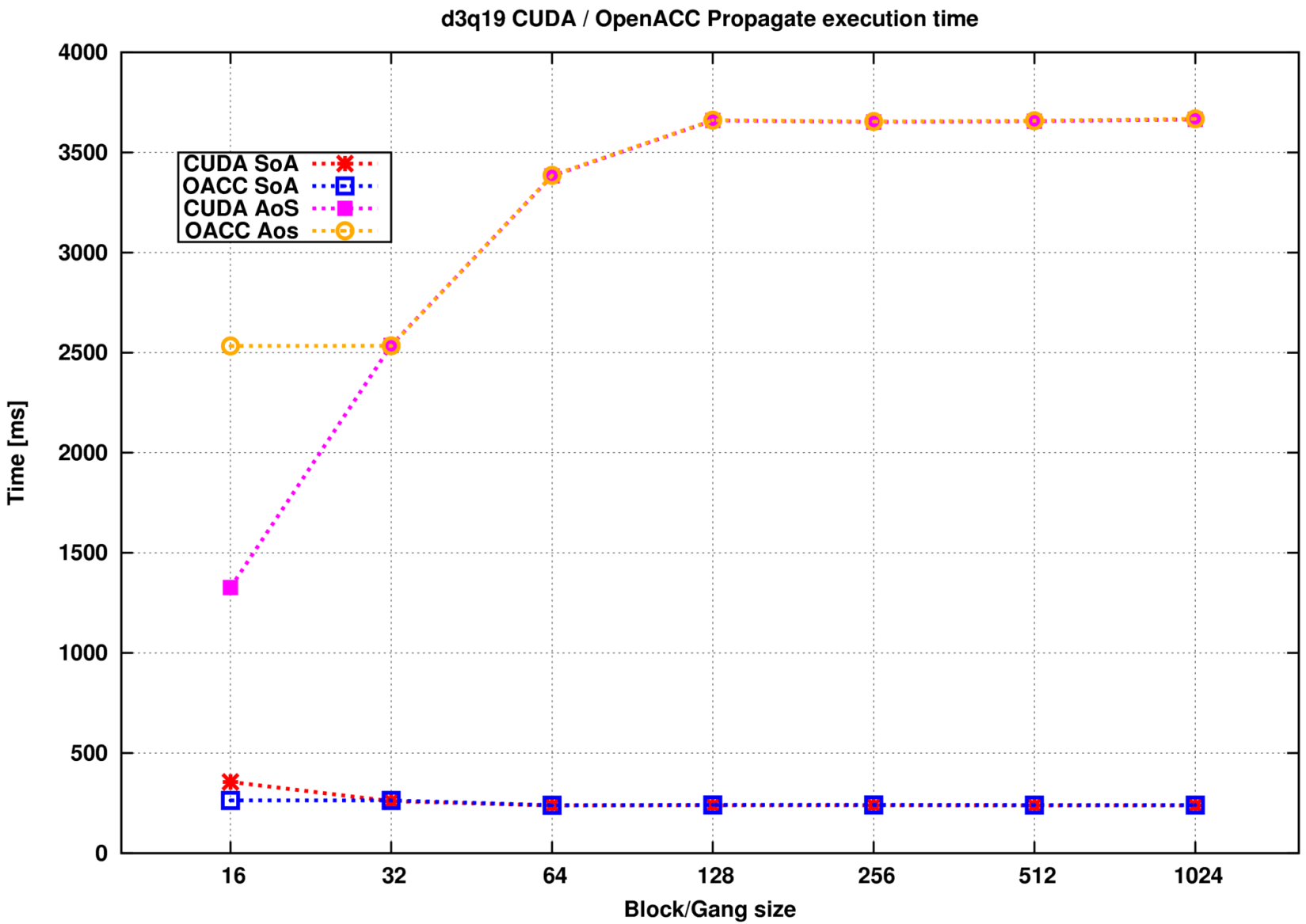
```
#pragma acc copyin(x), copy(y)
{
   my_saxpy(x, y);

   acc_async_wait(1);
}
```

```
void my_saxpy(float * x, float * y) {

  #pragma acc kernels present(x) present(y) async(1)
  #pragma acc loop gang vector(256)
  for (int i = 0; i < N; ++i)
    y[i] = a*x[i] + y[i];

}
```

OpenACC code computing a *saxpy* function on vectors *x* and *y*. *#pragma* clauses identifies the region to run on the accelerator and how to manage data transfers.

Outline

AoS vs SoA in a 3D Lattice Boltzmann Application



Memory layout for LQCD : AoS vs SoA

```
//fermions stored as AoS:
typedef struct {

  double complex c1; // component 1
  double complex c2; // component 2
  double complex c3; // component 3

} vec3_aos_t;

vec3_aos_t fermions[sizeh];
```

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

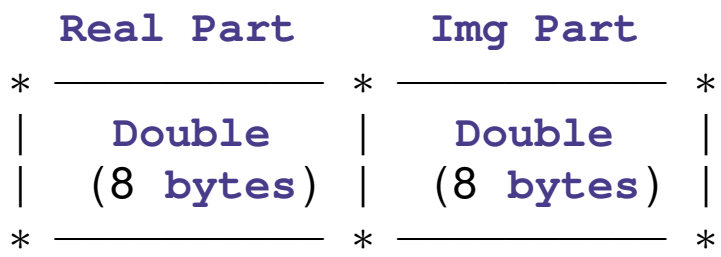
```
//fermions stored as SoA:
typedef struct {

  double complex c0[sizeh]; // components 1
  double complex c1[sizeh]; // components 2
  double complex c2[sizeh]; // components 3

} vec3_soa_t;

vec3_soa_t fermions;
```

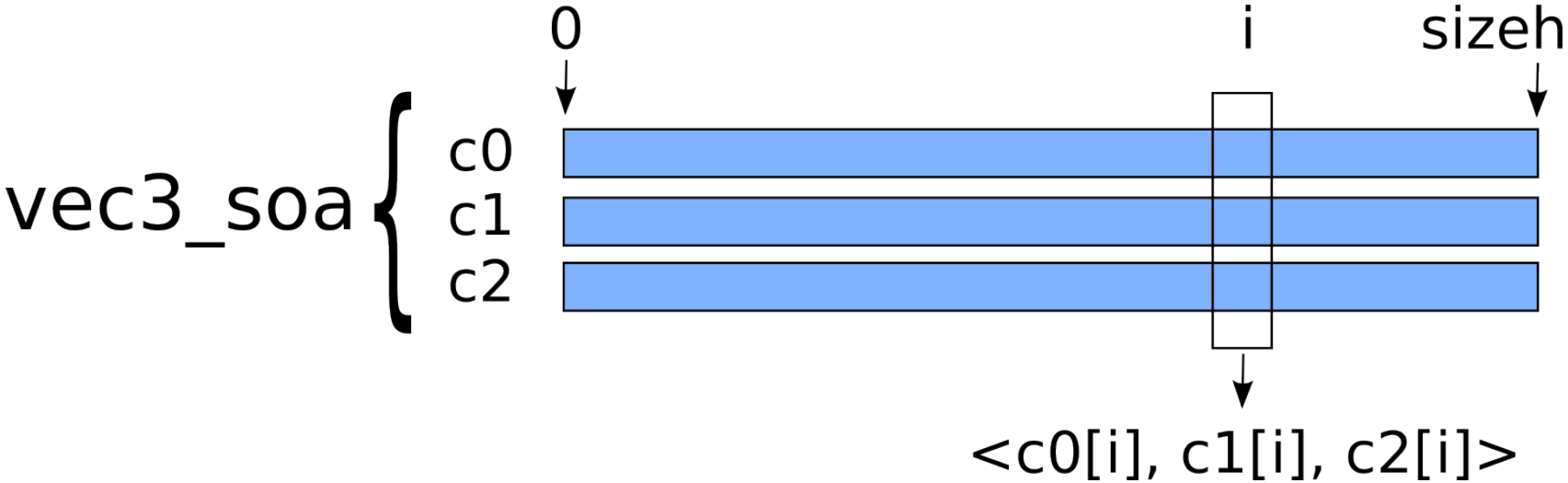
SoA: corresponding populations of different sites are allocated at contiguous memory addresses, enabling coalescing of accesses, and making use of full memory bandwidth.

Fermions vectors data structure

```
typedef struct {

  double complex c0[sizeh];
  double complex c1[sizeh];
  double complex c2[sizeh];

} vec3_soa_t;
```

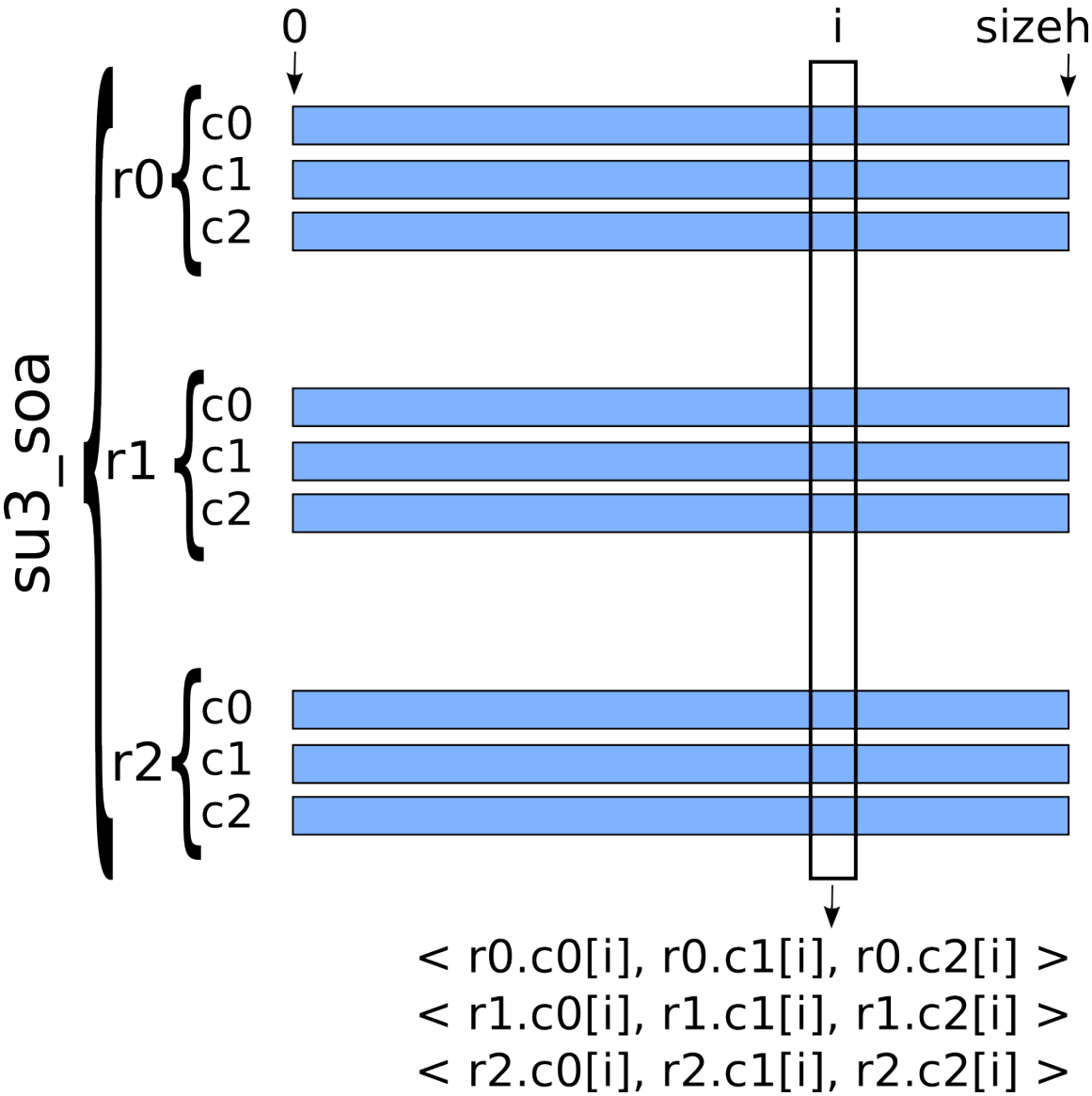
Since C99 float/double standard complex data type:





Gauge field matrices data structure

```
typedef struct {

  vec3_soa r0;
  vec3_soa r1;
  vec3_soa r2;

} su3_soa_t;
```



Outline

CUDA example for the Deo function

```
__global__ void Deo(const __restrict su3_soa_d * const u,
                    __restrict vec3_soa_d * const out,
                const __restrict vec3_soa_d * const in) {

  int x, y, z, t, xm, ym, zm, tm, xp, yp, zp, tp, idxh, eta;

  vec3 aux_tmp;
  vec3 aux;

  idxh = ((blockIdx.z * blockDim.z + threadIdx.z) * nxh * ny)
       + ((blockIdx.y * blockDim.y + threadIdx.y) * nxh)
       +  (blockIdx.x * blockDim.x + threadIdx.x);

  t =   (blockIdx.z * blockDim.z + threadIdx.z) / nz;
  z =   (blockIdx.z * blockDim.z + threadIdx.z) % nz;
  y =   (blockIdx.y * blockDim.y + threadIdx.y);
  x = 2*(blockIdx.x * blockDim.x + threadIdx.x) + ((y+z+t) & 0x1);

  ...
```

Outline

OpenACC example for the Deo function

```
void Deo(const __restrict su3_soa * const u,
                __restrict vec3_soa * const out,
                const __restrict vec3_soa * const in) {

  int hx, y, z, t;

  #pragma acc kernels present(u) present(out) present(in)
  #pragma acc loop independent gang(nt)
  for(t=0; t<nt; t++) {
    #pragma acc loop independent gang(nz/DIM_BLK_Z) vector(DIM_BLK_Z)
    for(z=0; z<nz; z++) {
      #pragma acc loop independent gang(ny/DIM_BLK_Y) vector(DIM_BLK_Y)
      for(y=0; y<ny; y++) {
        #pragma acc loop independent vector(DIM_BLK_X)
        for(hx=0; hx < nxh; hx++) {

          ...
```

Outline

Execution times for a 32^4 lattice

	Deo + Doe	
Block-size	CUDA	OpenACC
8,8,8	7.58	9.29
16,1,1	8.43	16.16
16,2,1	7.68	9.92
16,4,1	7.76	9.96
16,8,1	7.75	10.11
16,16,1	7.64	10.46

Time in [ns per site], run on an NVIDIA K20m GPU using double precision;

OpenACC code compiled using PGI 14.6

Execution times summary

Lattice size	Thread Block size	CUDA	OpenACC
16^4	8x8x8	7.27	9.86
32^4	8x8x8	7.58	9.23
48^4	8x8x8	7.86	9.11
64x32x32x16	16x8x4	7.59	9.16
"	32x8x2	7.62	9.12
"	32x4x4	7.54	9.06
"	32x4x2	7.61	10.56
"	32x2x2	7.71	10.18
32x16x16x16	16x8x4	7.45	9.78

Time in [ns per site], run on an NVIDIA K20m GPU using double precision;

OpenACC code compiled using PGI 14.6

Outline

Prospective multi-GPU Lattice: $48 \times 48 \times 48 \times 96$

Local Lat l_x, l_y, l_z, l_t	No. of GPUs	Block Size	Mem [MB]	Data Trans. [48B]	*Tc* [*ms*]	*Td* [*ms*]
$48 \times 48 \times 48 \times 6$	16	$8 \times 8 \times 8$	415	$2 \times (48 \times 48 \times 48)$	6.16	$\simeq 1.6$
$48 \times 48 \times 24 \times 12$	16	$8 \times 8 \times 8$	415	$2 \times (48 \times 48 \times 24)$ $+2 \times (48 \times 48 \times 12)$	6.21	$\simeq 1.8$
$48 \times 48 \times 24 \times 6$	32	$8 \times 8 \times 8$	208	$2 \times (48 \times 48 \times 24)$ $+2 \times (48 \times 48 \times 6)$	3.30	$\simeq 1.4$
$48 \times 24 \times 24 \times 48$	8	$8 \times 8 \times 8$	830	$4 \times (48 \times 24 \times 48)$ $+2 \times (48 \times 24 \times 24)$	12.26	$\simeq 2.5$
$24 \times 24 \times 24 \times 96$	8	$8 \times 8 \times 8$	830	$6 \times (24 \times 24 \times 96)$	17.24	$\simeq 3.0$
$16 \times 16 \times 16 \times 96$	27	$8 \times 8 \times 8$	369	$6 \times (16 \times 16 \times 96)$	5.38	$\simeq 1.8$

Data transfers expressed in number of fermions (i.e. 48 bytes).

Thanks for Your attention