Design and optimization of a portable Lattice QCD Monte Carlo code for heterogeneous HPC architectures, using OpenACC

C. Bonati[1], E. Calore[2], S. Coscetti[1], M. D'Elia[3], M. Mesiti[3], F. Negro[1], S. F. Schifano[2], G.Silvi[2] and R. Tripiccione[2]

1) INFN Pisa,
2) INFN and Università degli Studi di Ferrara,
3) INFN and Università di Pisa

Italy

Enrico Calore

Barcelona (Spain), May 19th, 2017

PRACE Scientific and Industrial Conference 2017 – PRACEdays17

Outline

Outline

Hardware in HPC

A large fraction of modern HPC systems computing power is provided by highly parallel accelerators, such as GPUs and MICs.

The quest for performances lead to start using languages not so well consolidated and/or proprietary.

- Proprietary languages prevent code portability:
  $\Rightarrow$ need to maintain a code version for each architecture
- Open Specifications languages may not be supported by all vendors:
  $\Rightarrow$ need to maintain multiple code versions
  $\Rightarrow$ need to re-implement the code if they "die"

Hardware in HPC

A large fraction of modern HPC systems computing power is provided by highly parallel accelerators, such as GPUs and MICs.

The quest for performances lead to start using languages not so well consolidated and/or proprietary.

- Proprietary languages prevent code portability:
  $\Rightarrow$ need to maintain a code version for each architecture
- Open Specifications languages may not be supported by all vendors:
  $\Rightarrow$ need to maintain multiple code versions
  $\Rightarrow$ need to re-implement the code if they "die"

Hardware in HPC

A large fraction of modern HPC systems computing power is provided by highly parallel accelerators, such as GPUs and MICs.

The quest for performances lead to start using languages not so well consolidated and/or proprietary.

- Proprietary languages prevent code portability:
  $\Rightarrow$ need to maintain a code version for each architecture
- Open Specifications languages may not be supported by all vendors:
  $\Rightarrow$ need to maintain multiple code versions
  $\Rightarrow$ need to re-implement the code if they "die"

Hardware in HPC

A large fraction of modern HPC systems computing power is provided by highly parallel accelerators, such as GPUs and MICs.

The quest for performances lead to start using languages not so well consolidated and/or proprietary.

- Proprietary languages prevent code portability:
  $\Rightarrow$ need to maintain a code version for each architecture
- Open Specifications languages may not be supported by all vendors:
  $\Rightarrow$ need to maintain multiple code versions
  $\Rightarrow$ need to re-implement the code if they “die”

Scientific Software Engineering

HPC Scientific Software development has to adapt to specific characteristics:

- Software lifetime may be very long; even tens of years.
- Software must be portable across current and future HPC hardware architectures, which are very heterogeneous (e.g. CPU, GPU, MIC, etc.).
- Software has to be strongly optimized to exploit the available hardware for maximum performances.
- What the software does, has to be always under control, thus higher level programming languages could be a problem.

Scientific Software Engineering

HPC Scientific Software development has to adapt to specific characteristics:

- Software lifetime may be very long; even tens of years.
- Software must be portable across current and future HPC hardware architectures, which are very heterogeneous (e.g. CPU, GPU, MIC, etc.).
- Software has to be strongly optimized to exploit the available hardware for maximum performances.
- What the software does, has to be always under control, thus higher level programming languages could be a problem.

Scientific Software Engineering

HPC Scientific Software development has to adapt to specific characteristics:

- Software lifetime may be very long; even tens of years.
- Software must be portable across current and future HPC hardware architectures, which are very heterogeneous (e.g. CPU, GPU, MIC, etc.).
- Software has to be strongly optimized to exploit the available hardware for maximum performances.
- What the software does, has to be always under control, thus higher level programming languages could be a problem.

Scientific Software Engineering

HPC Scientific Software development has to adapt to specific characteristics:

- Software lifetime may be very long; even tens of years.
- Software must be portable across current and future HPC hardware architectures, which are very heterogeneous (e.g. CPU, GPU, MIC, etc.).
- Software has to be strongly optimized to exploit the available hardware for maximum performances.
- What the software does, has to be always under control, thus higher level programming languages could be a problem.

Scientific Software Engineering

HPC Scientific Software development has to adapt to specific characteristics:

- Software lifetime may be very long; even tens of years.
- Software must be portable across current and future HPC hardware architectures, which are very heterogeneous (e.g. CPU, GPU, MIC, etc.).
- Software has to be strongly optimized to exploit the available hardware for maximum performances.
- What the software does, has to be always under control, thus higher level programming languages could be a problem.

Lattice Quantum Chromodynamics (LQCD)

Quantum Chromodynamics (QCD) is the quantum field theory that describes strong interactions in the Standard Model of particle physics.

It describes the interactions of six different species (flavors) of quarks.

Lattice Quantum Chromodynamics (LQCD) Monte Carlo code

- LQCD represent a typical HPC grand challenge.
- Physics results strongly limited by available computational resources.
- Several generations of parallel machines were developed on purpose for LQCD simulations.

Lattice Quantum Chromodynamics (LQCD)

Quantum Chromodynamics (QCD) is the quantum field theory that describes strong interactions in the Standard Model of particle physics.

It describes the interactions of six different species (flavors) of quarks.

Lattice Quantum Chromodynamics (LQCD) Monte Carlo code

- LQCD represent a typical HPC grand challenge.
- Physics results strongly limited by available computational resources.
- Several generations of parallel machines were developed on purpose for LQCD simulations.

Lattice Quantum Chromodynamics (LQCD)

Two different initial versions: **C++** for CPUs and **CUDA** for NVIDIA GPUs

(state-of-the-art discretization; tree-level Symanzik improved action for the gauge part; and the stout-improved “staggered” action for the fermion part)

Our goal was to have:

- Just one code version.
- Being able to run on several architectures.
- Avoid major code changes when new processors will be available.
- Have roughly the same level of efficiency.

Looking for an acceptable trade-off between:
maintainability, portability and efficiency

Lattice Quantum Chromodynamics (LQCD)

Two different initial versions: **C++** for CPUs and **CUDA** for NVIDIA GPUs

(state-of-the-art discretization; tree-level Symanzik improved action for the gauge part; and the stout-improved “staggered” action for the fermion part)

Our goal was to have:

- Just one code version.
- Being able to run on several architectures.
- Avoid major code changes when new processors will be available.
- Have roughly the same level of efficiency.

Looking for an acceptable trade-off between:
maintainability, portability and efficiency

Lattice Quantum Chromodynamics (LQCD)

Two different initial versions: **C++** for CPUs and **CUDA** for NVIDIA GPUs

(state-of-the-art discretization; tree-level Symanzik improved action for the gauge part; and the stout-improved “staggered” action for the fermion part)

Our goal was to have:

- Just one code version.
- Being able to run on several architectures.
- Avoid major code changes when new processors will be available.
- Have roughly the same level of efficiency.

Looking for an acceptable trade-off between:
maintainability, portability and efficiency

Outline

OpenACC example: the Saxpy function

```
{
    my_saxpy(x, y);

}
```

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

  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.

OpenACC example: the Saxpy function

```
{
    my_saxpy(x, y);

    acc_async_wait(1);
}
```

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

  #pragma acc kernels 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.

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.

The use of OpenACC as a prospective solution

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

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

NVIDIA seems committed to develop PGI and also GCC supports OpenACC

The use of OpenACC as a prospective solution

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

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

NVIDIA seems committed to develop PGI and also GCC supports OpenACC

The use of OpenACC as a prospective solution

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

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

NVIDIA seems committed to develop PGI and also GCC supports OpenACC

LQCD Hot spots analysis

Most of the running time in a simulation is spent executing the Dirac Operator i.e. two functions:

- D_{eo}: reads from even sites of the lattice and writes in odd ones.
- D_{oe}: reads from odd sites of the lattice and writes in even ones.
- Both perform mainly vector-$SU(3)$ matrices multiplications.

To have since the beginning an estimation of performance loss:

implementation started from the D_{eo} and D_{oe} functions

LQCD Hot spots analysis

Most of the running time in a simulation is spent executing the Dirac Operator i.e. two functions:

- D_{eo}: reads from even sites of the lattice and writes in odd ones.
- D_{oe}: reads from odd sites of the lattice and writes in even ones.
- Both perform mainly vector-$SU(3)$ matrices multiplications.

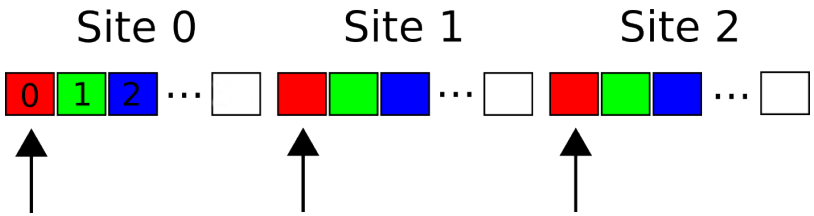
To have since the beginning an estimation of performance loss:

implementation started from the D_{eo} and D_{oe} functions

Planning the memory layout for LQCD : AoS vs SoA

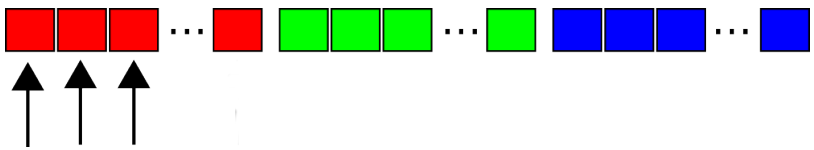
First version in C++ targeting CPU based clusters adopts **AoS**:

```
//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];
```



Later version in C++/CUDA targeting NVIDIA GPU clusters adopts **SoA**:

```
//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;
```



A micro-benchmark to assess the $SU(3)$ matrix - fermion multiplication performance

Testing data layout and data type

Table: Execution time [*ms*] to perform 32^4 vector-$SU(3)$ multiplications (DP)

Data		NVIDIA	Intel E5-2620v2		Intel E5-2630v3	
Type	Layout	K20 GPU	Naive	Vect.	Naive	Vect.
Complex	AoS	8.75	30.16	*n.a.*[1]	20.47	*n.a.*[1]
	SoA	1.45	45.75	32.21	18.69	13.93
Double	SoA	1.48	106.90	38.58	43.69	16.08

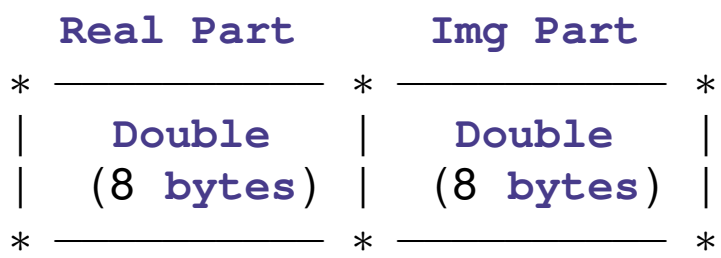
Intel Xeon E5-2620v2 implements AVX instructions
Intel Xeon E5-2630v3 implements AVX2 and FMA3 instructions

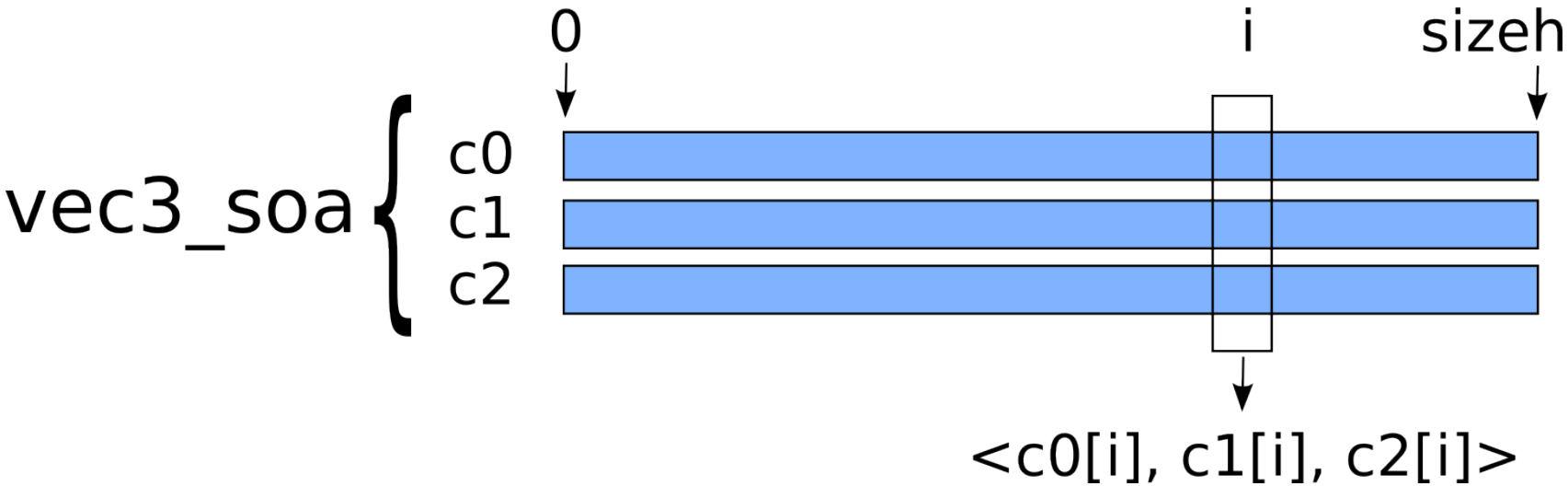
1) Vectorization is not possible when using AoS data layout

Fermions vectors data structure

```
typedef struct {

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

} vec3_soa_t;
```

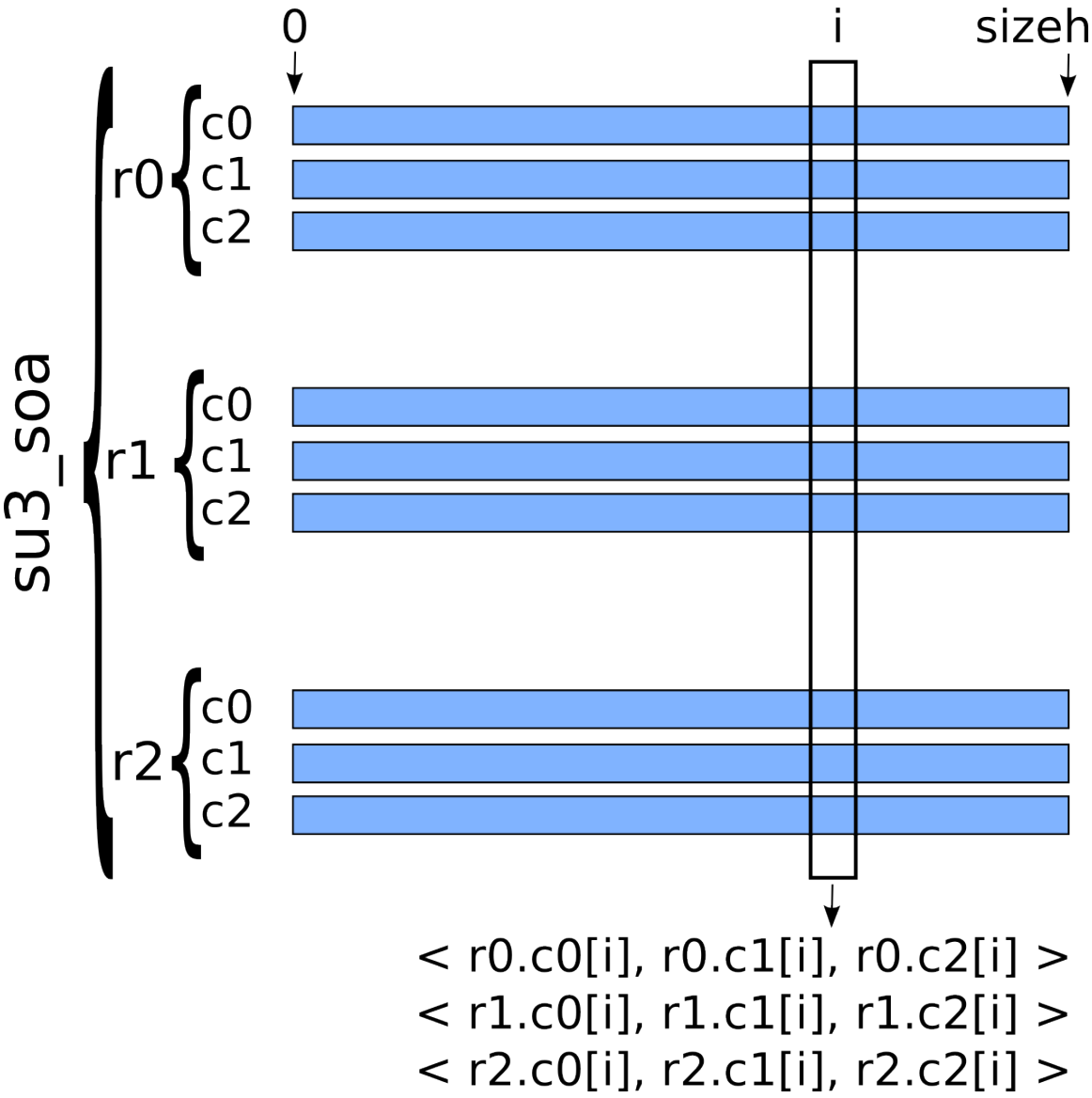
Since C99 float/double standard complex data type:





Gauge field matrices data structure

```
typedef struct {

  vec3_soa r0;
  vec3_soa r1;
  vec3_soa r2;

} su3_soa_t;
```



OpenACC example for the Deo function

```
void acc_Deo( __restrict const su3_soa * const u,
              __restrict vec3_soa * const out,
              __restrict const vec3_soa * const in,
              __restrict const double_soa * const backfield){
  int hd0, d1, d2, d3;
  #pragma acc kernels present(in) present(out)
                      present(u)  present(backfield) async(1)
  #pragma acc loop independent gang(GANG)
  for(d3=0; d3<nd3;d3++) {
    #pragma acc loop independent vector tile(TILE0,TILE1,TILE2)
    for(d2=0; d2<nd2; d2++) {
      for(d1=0; d1<nd1; d1++) {
        for(hd0=0; hd0 < nd0h; hd0++) {
          ...
}}}}}
```

directive vector tile divides the computational domain in sub-lattices (tiles), each processed within a compute unit in order to allow data re-use.

Outline

Dirac Operator Performance

Performance of the Dirac Operator on different CPUs and GPUs:

Lattice	Processor (CPU or GPU)									
	NVIDIA GK201		NVIDIA P100		Intel E5-2630v3		Intel E5-2697v4		AMD W9100	
	SP	DP	SP	DP	SP	DP	SP	DP	SP	DP
$32^2 \times 8 \times 32$	6.22	9.63	1.77	3.07	72.18	99.17	38.92	54.90		
$32^3 \times 8$	5.73	8.59	1.22	2.48	72.81	101.46	77.33	103.87		
24^4	5.91	10.85	1.58	2.90	70.44	94.42	51.13	66.87	22.16	41.43
32^4	5.69	8.60	1.32	2.40	79.05	100.19	43.90	54.88	20.40	41.74
$32^3 \times 36$	5.88	8.94	1.46	2.54	83.12	107.47	38.82	50.29		

Table: Execution time per lattice site, for the Dirac operator, in [ns]

Preliminary energy-awareness studies

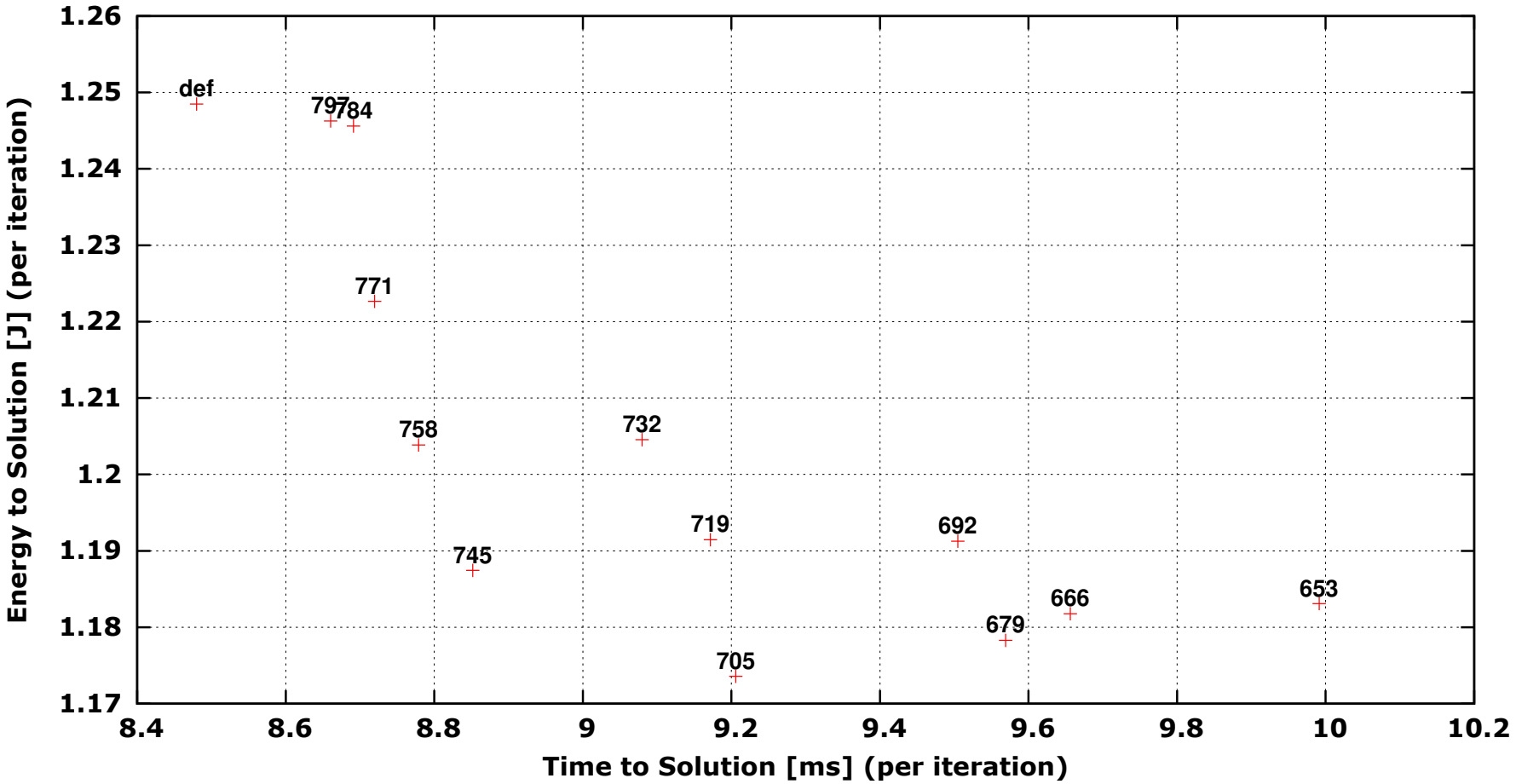


Figure: Energy consumption of the Dirac Operator versus execution time, on one GPU of an NVIDIA K80 board for different GPU frequencies (as labels).

Full Simulation Performance

OpenACC code compared with earlier GPU-optimized CUDA

Lattice	am	β	CUDA	OpenACC	Variation
$32^3 \times 8$	0.0125	5.55	392.69	490.74	+25%
24^4	0.0125	5.55	303.80	328.07	+8%
32^4	0.001	5.52	8973.82	8228.36	-8%

Table: Execution time [sec] of a full trajectory of a complete Monte Carlo simulation for several typical physical parameters, running on one GPU of a NVIDIA K80 board.

Here we use the unimproved staggered fermions as the CUDA code does not support the more advanced improvements available in the OpenACC version.

Outline

Conclusions and future works

Conclusions:

- we developed an OpenACC implementation for LQCD simulations
- we reached the goal of a single code version able to run on different architectures without code changes
- performances are comparable to the CUDA code on NVIDIA GPUs
- performances on Intel CPUs can be enhanced
- compilers support for AMD GPUs is not yet very mature

Future works:

- we are working on the MPI implementation allowing multi-node / multi-accelerator simulations
- we hope to be soon able to run efficiently also on Intel KNL

Conclusions and future works

Conclusions:

- we developed an OpenACC implementation for LQCD simulations
- we reached the goal of a single code version able to run on different architectures without code changes
- performances are comparable to the CUDA code on NVIDIA GPUs
- performances on Intel CPUs can be enhanced
- compilers support for AMD GPUs is not yet very mature

Future works:

- we are working on the MPI implementation allowing multi-node / multi-accelerator simulations
- we hope to be soon able to run efficiently also on Intel KNL

Thanks for Your attention