A portable LQCD Monte Carlo code using MPI and OpenACC

Enrico Calore

Università degli Studi di Ferrara and INFN Ferrara, Italy

Bari, December 13-15th, 2017

The XVII workshop on
Statistical Mechanics and non Perturbative Field Theory

Outline

1. Introduction
   - Scientific applications
   - HPC hardware and software trends
   - OpenACC at a glance
2. Design and Implementation with OpenACC
   - Initial planning
   - Analysis and design
   - Implementation
3. Results and Conclusions

Outline

1. Introduction
   - Scientific applications
   - HPC hardware and software trends
   - OpenACC at a glance
2. Design and Implementation with OpenACC
   - Initial planning
   - Analysis and design
   - Implementation
3. Results and Conclusions

HPC community specificity

Concerning HPC Scientifc Software,
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 better performances.

HPC community specificity

Concerning HPC Scientifc Software,
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 better performances.

HPC community specificity

Concerning HPC Scientifc Software,
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 better performances.

HPC community specificity

Concerning HPC Scientifc Software,
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 better performances.

Making decisions in uncertain times

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

Although reluctant to embrace not consolidated technologies, the quest for performances lead to start using languages such as CUDA or OpenCL

Proprietary languages prevent code portability:
$\Rightarrow$ need to maintain multiple code versions

Open spec. languages may not be supported by all vendors:
$\Rightarrow$ need to re-implement the code
$\Rightarrow$ need to maintain multiple code versions

Making decisions in uncertain times

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

Although reluctant to embrace not consolidated technologies, the quest for performances lead to start using languages such as CUDA or OpenCL

Proprietary languages prevent code portability:
$\Rightarrow$ need to maintain multiple code versions

Open spec. languages may not be supported by all vendors:
$\Rightarrow$ need to re-implement the code
$\Rightarrow$ need to maintain multiple code versions

Making decisions in uncertain times

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

Although reluctant to embrace not consolidated technologies, the quest for performances lead to start using languages such as CUDA or OpenCL

Proprietary languages prevent code portability:
$\Rightarrow$ need to maintain multiple code versions

Open spec. languages may not be supported by all vendors:
$\Rightarrow$ need to re-implement the code
$\Rightarrow$ need to maintain multiple code versions

Making decisions in uncertain times

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

Although reluctant to embrace not consolidated technologies, the quest for performances lead to start using languages such as CUDA or OpenCL

Proprietary languages prevent code portability:
$\Rightarrow$ need to maintain multiple code versions

Open spec. languages may not be supported by all vendors:
$\Rightarrow$ need to re-implement the code
$\Rightarrow$ need to maintain multiple code versions

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` directives syntax.

OpenMP recently added support also for accelerators

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` directives syntax.

OpenMP recently added support also for accelerators

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` directives syntax.

OpenMP recently added support also for accelerators

The use of OpenACC as a prospective solution

The case of Lattice QCD

- Existing versions of the code targeting different architectures:
  - $\Rightarrow$ C++ targeting x86 CPUs
  - $\Rightarrow$ C++/CUDA targeting NVIDIA GPUs

The faced challenge is to design and implement one version:

- with good performances on present best performing architectures;
- portable across different available architectures;
- easy to maintain, allowing scientists to change/improve the code;
- portable, or easily portable on future unknown architectures.

The use of OpenACC as a prospective solution

The case of Lattice QCD

- Existing versions of the code targeting different architectures:

  $\Rightarrow$ C++ targeting x86 CPUs

  $\Rightarrow$ C++/CUDA targeting NVIDIA GPUs

The faced challenge is to design and implement one version:

- with good performances on present best performing architectures;
- portable across different available architectures;
- easy to maintain, allowing scientists to change/improve the code;
- portable, or easily portable on future unknown architectures.

The use of OpenACC as a prospective solution

The case of Lattice QCD

- Existing versions of the code targeting different architectures:

  $\Rightarrow$ C++ targeting x86 CPUs

  $\Rightarrow$ C++/CUDA targeting NVIDIA GPUs

The faced challenge is to design and implement one version:

- with good performances on present best performing architectures;
- portable across different available architectures;
- easy to maintain, allowing scientists to change/improve the code;
- portable, or easily portable on future unknown architectures.

The use of OpenACC as a prospective solution

The case of Lattice QCD

- Existing versions of the code targeting different architectures:

  $\Rightarrow$ C++ targeting x86 CPUs

  $\Rightarrow$ C++/CUDA targeting NVIDIA GPUs

The faced challenge is to design and implement one version:

- with good performances on present best performing architectures;
- portable across different available architectures;
- easy to maintain, allowing scientists to change/improve the code;
- portable, or easily portable on future unknown architectures.

The use of OpenACC as a prospective solution

The case of Lattice QCD

- Existing versions of the code targeting different architectures:
  - $\Rightarrow$ C++ targeting x86 CPUs
  - $\Rightarrow$ C++/CUDA targeting NVIDIA GPUs

The faced challenge is to design and implement one version:

- with good performances on present best performing architectures;
- portable across different available architectures;
- easy to maintain, allowing scientists to change/improve the code;
- portable, or easily portable on future unknown architectures.

The use of OpenACC as a prospective solution

The case of Lattice QCD

- Existing versions of the code targeting different architectures:

  $\Rightarrow$ C++ targeting x86 CPUs

  $\Rightarrow$ C++/CUDA targeting NVIDIA GPUs

The faced challenge is to design and implement one version:

- with good performances on present best performing architectures;
- portable across different available architectures;
- easy to maintain, allowing scientists to change/improve the code;
- portable, or easily portable on future unknown architectures.

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];

}
```

Plain C 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* directives 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* directives identifies the region to run on the accelerator and how to manage data transfers.

Outline

1 Introduction
- Scientific applications
- HPC hardware and software trends
- OpenACC at a glance

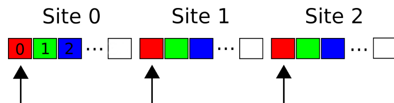
2 Design and Implementation with OpenACC
- Initial planning
- Analysis and design
- Implementation

3 Results and Conclusions

Planning the memory layout for LQCD : AoS vs SoA

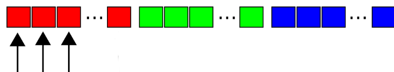
First version in C++ targeting CPU based clusters adopts **AoS**:

```
//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;
```



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

1) Vectorization is not possible when using AoS data layout
Intel Xeon E5-2620v2 implements AVX instructions
Intel Xeon E5-2630v3 implements AVX2 and FMA3 instructions

C. Bonati, E. Calore, S. Coscetti, M. D'Elia, M. Mesiti, F. Negro, S. F. Schifano, R. Tripiccione, *Development of Scientific Software for HPC Architectures Using OpenACC: The Case of LQCD*, IEEE/ACM SE4HPCS 2015. doi: 10.1109/SE4HPCS.2015.9

Fermions vectors data structure

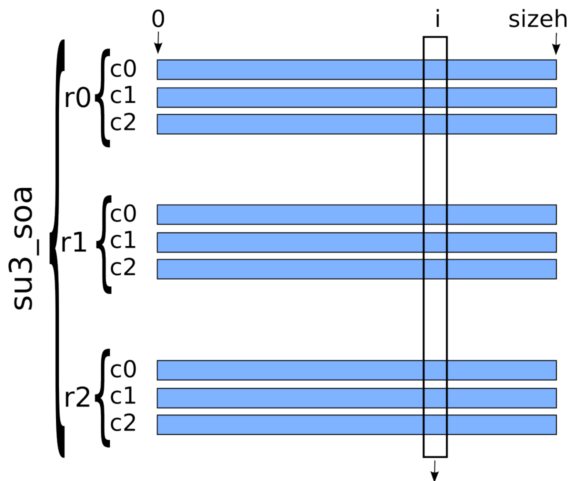
```
typedef struct {

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

} vec3_soa_t;
```

Since C99 float/double standard complex data type:

```
     Real Part         Img Part
*  ___________  *  ___________  *
|    Double     |    Double     |
|  (8 bytes)    |  (8 bytes)    |
*  ___________  *  ___________  *
```

Gauge field matrices data structure

```
typedef struct {

  vec3_soa r0;
  vec3_soa r1;
  vec3_soa r2;

} su3_soa_t;
```



Dirac Operator Implementation

Most of the running time in a LQCD simulation is spent applying the Dirac Operator:

- 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 vector-$SU(3)$ matrices multiplications.

Strongly memory-bound operation (< 1 FLOP/Byte)

Dirac Operator Implementation

Most of the running time in a LQCD simulation is spent applying the Dirac Operator:

- 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 vector-$SU(3)$ matrices multiplications.

Strongly memory-bound operation ($<$ 1 FLOP/Byte)

OpenACC example for the Deo function

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

 int hx, y, z, t;

 #pragma acc kernels present(u) present(out) present(in) present(bfield)
 #pragma acc loop independent gang collapse(2)
 for(t=0; t<nt; t++) {
   for(z=0; z<nz; z++) {
     #pragma acc loop independent vector tile(TDIM0,TDIM1)
     for(y=0; y<ny; y++) {
       for(hx=0; hx < nxh; hx++) {

         ...
```

Nested loops over the lattice sites annotated with OpenACC directives.

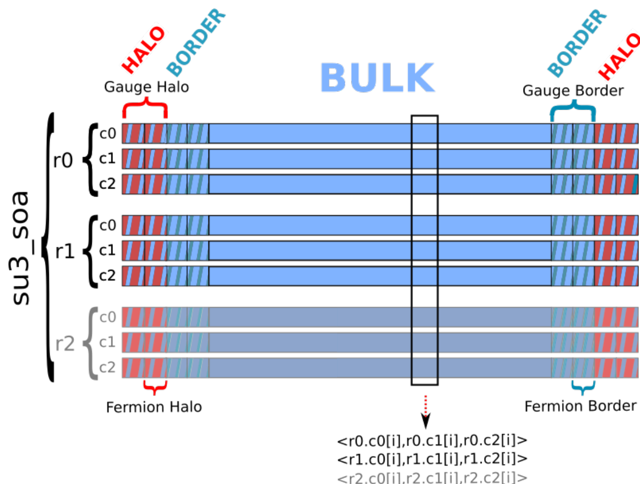
Single Device Performance

Lattice	Processor (CPU or GPU)							
	NVIDIA GK210		NVIDIA P100		Intel E5-2630v3		Intel E5-2697v4	
	SP	DP	SP	DP	SP	DP	SP	DP
24^4	4.43	8.62	1.58	2.90	70.44	94.42	51.13	66.87
32^4	4.02	9.54	1.32	2.40	79.05	100.19	43.90	54.88

Table: Measured execution time per lattice site [ns] for the Dirac operator, on several processors, in single and double precision. PGI Compiler 16.10.

C. Bonati, E. Calore, S. Coscetti, M. D'Elia, M. Mesiti, F. Negro, S. F. Schifano, G. Silvi, R. Tripiccione, *Design and optimization of a portable LQCD Monte Carlo code using OpenACC* International Journal Modern Physics C, 28, 1750063 (2017). doi: 10.1142/S0129183117500632

Multi Device Implementation

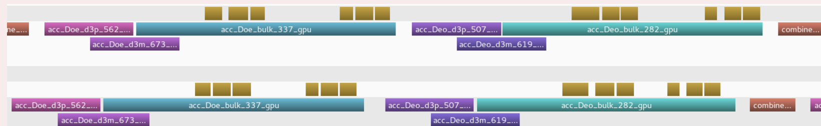


Different kernels/functions for borders and bulk operations

Overlap between computation and communication

One dimensional tailing of a $32^3 \times 48$ Lattice across:

$8\times$ GPUs



Local lattice: $32^3 \times 6$ per GPU

$12\times$ GPUs

Local lattice: $32^3 \times 4$ per GPU

Relative Speedup on NVIDIA K80 GPUs

Dirac Operator in double precision

32^3 x 48
32^3 x 64

Relative Speedup

Number of GPUs

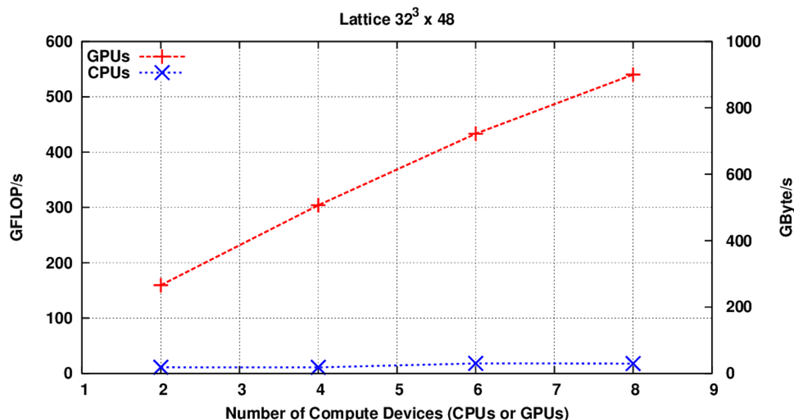
C. Bonati, E. Calore, M. D'Elia, M. Mesiti, F. Negro, F. Sanfilippo, S. F. Schifano, G. Silvi, R. Tripiccione, *Portable multi-node LQCD Monte Carlo simulations using OpenACC*, International Journal Modern Physics C, Being submitted.

Outline

1. Introduction
   - Scientific applications
   - HPC hardware and software trends
   - OpenACC at a glance
2. Design and Implementation with OpenACC
   - Initial planning
   - Analysis and design
   - Implementation
3. Results and Conclusions

Strong Scaling Results

Roberge Weiss simulation over a $32^3 \times 48$ lattice, with mass 0.0015 and beta 3.3600, using mixed precision floating-point.



Using 2 CPUs we measure a $\approx 14\times$ increase in the execution time wrt using 2 GPUs and the gap widens for more devices.

Conclusions

LQCD Monte Carlo

- a single code version able to run on different architectures
- capability to run on several computing devices / nodes
- still regular plain C code if ignoring directives
- performance comparable to CUDA implementations on NVIDIA GPUs

Future works

- investigate performance of multi-dimensional tiling
- experiment different compilers targeting Intel CPUs (e.g. GCC 7)
- introduce optimizations for Intel CPUs and MICs without impacting GPU performance

Conclusions

LQCD Monte Carlo

- a single code version able to run on different architectures
- capability to run on several computing devices / nodes
- still regular plain C code if ignoring directives
- performance comparable to CUDA implementations on NVIDIA GPUs

Future works

- investigate performance of multi-dimensional tiling
- experiment different compilers targeting Intel CPUs (e.g. GCC 7)
- introduce optimizations for Intel CPUs and MICs without impacting GPU performance

Thanks for Your attention