Development of scientific software for HPC architectures using OpenACC: the case of LQCD

Claudio Bonati*, Enrico Calore†, Simone Coscetti*, Massimo D'Elia‡, Michele Mesiti‡,
Francesco Negro*, Sebastiano Fabio Schifano† and Raffaele Tripiccione†
* INFN Sezione di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
† INFN and Università di Ferrara, Via Saragat 1, 44122 Ferrara, Italy
‡ INFN and Università di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy

***Abstract*—Many scientific software applications, that solve complex compute- or data-intensive problems, such as large parallel simulations of physics phenomena, increasingly use HPC systems in order to achieve scientifically relevant results. An increasing number of HPC systems adopt heterogeneous node architectures, combining traditional multi-core CPUs with energy-efficient massively parallel accelerators, such as GPUs. The need to exploit the computing power of these systems, in conjunction with the lack of standardization in their hardware and/or programming frameworks, raises new issues with respect to scientific software development choices, which strongly impact software maintainability, portability and performance. Several new programming environments have been introduced recently, in order to address these issues. In particular, the OpenACC programming standard has been designed to ease the software development process for codes targeted for heterogeneous machines, helping to achieve code and performance portability. In this paper we present, as a specific OpenACC use case, an example of design, porting and optimization of an LQCD Monte Carlo code intended to be portable across present and future heterogeneous HPC architectures; we describe the design process, the most critical design choices and evaluate the tradeoff between portability and efficiency.**

I. Introduction

Several scientific communities use a significant part of their resources in scientific software design and development, in order to meet their compute- or data-intensive requirements.

Some SE (Software Engineering) studies in the literature have focused on the software development methodologies used by these communities [1]–[4] highlighting that an actual SE model is rarely followed. This may be due to the lack of knowledge about SE techniques, but also to the fact that most SE best practices were not designed taking into account scientific software peculiarities [2].

Nevertheless, common practices, and thus a software development methodology, can be identified in most scientific software development projects. In particular a rapid IID (Iterative and Incremental Development) is often preferred with respect to the traditional waterfall discrete phased model [3]. The design, development and testing phases are often not clearly distinct and the source code is in most cases continuously evolving in subsequent iterations. As opposed to small evolutionary improvements, codes typically undergo frequent major modifications. Moreover, in some cases actual production releases do not even exist.

Very different software applications could be defined as "scientific", but in the rest of this work we will focus on the subset of scientific HPC (High Performance Computing) software applications targeted to computer clusters; often referred as *codes* in the HPC community. The software development process adopted for this kind of applications, commonly follows the same methodologies previously introduced, but presents further particular characteristics [4]:

- code lifetimes may be very long (i.e. even tens of years); this implies a careful choice of the adopted software languages and frameworks, avoiding those which may not be supported long enough;
- codes must run on multiple current and future HPC systems, so portability is a major concern; on the other hand, computing performance has a very high priority since available HPC computing resources are limited, and it is often the case that portability and performance are conflicting requirements.
- higher level programming languages and abstractions are often avoided in favor to a better understanding of what the code actually does and to allow for optimizations (e.g. plain C and Fortran are the most used languages yet);

With the advent of modern heterogeneous HPC systems equipped with hardware accelerators, a large fraction of the computing power becomes accessible only by the use of specific codes able to run on these accelerators. Consequently, the expectation of a relevant performance increase has led scientific software developers to overcome the initial skepticism about recently introduced languages and software frameworks. Several scientific HPC software developers have undertaken the programming effort to port existing software (or to create new applications) using specific languages able to exploit hardware accelerators. In most cases, various research groups opted for (re-)writing their codes, using proprietary programming

978-1-4673-7082-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SE4HPCS.2015.9

languages such as CUDA. Unfortunately this choice badly impacts on code portability, limiting code usage to a single vendor's accelerators and/or on code maintainability, forcing to write and mantain specific implementations for different hardware architectures. This is a serious issue in our case, as the continous evolution of the codes means that maintaining several implementations, and ensuring their correctness, is a complex, difficult and error prone task.

New languages and software frameworks have been introduced recently, aiming to grant code portability, in order to develop a single source code able to run on different architectures, such as OpenCL. Various scientific applications have indeed been ported to this language [5], but several drawbacks have arisen also in this case, as for example the high code verbosity, the need to take hardware architecture details into account, but moreover its limited support due to marketing reasons. Indeed, one of the most important accelerator manufacturer has recently dropped support for the OpenCL language. This is forcing developers who wrote scientific applications using OpenCL to port their software (again, in some cases) to other languages.

In this work we present our experience in the re-design, implementation and optimization of an existing scientific HPC software, previously implemented in two versions: a C++ one, running on ordinary CPUs and a C++/CUDA one able to exploit NVIDIA GPUs accelerators [6]. Our final aim is to produce a single implementation able to run on heterogeneous present and future HPC systems and exhibiting a fair trade off between code and performance portability.

The software application described in this work is a Lattice Quantum Chromodynamics (LQCD) Monte Carlo code. LQCD simulations enable researchers to investigate aspects of the Quantum Chromodynamics (QCD) physics that would be impossible to systematically investigate in perturbation theory. The computation time for LQCD simulations is a strong limiting factor, bounding for example the usable lattice size. Fortunately enough, the most time consuming functions of the LQCD algorithms are embarassingly parallel, if run on a shared memory system. However the challenge of designing and running efficient codes is not easily met, in particular if code portability is required and the lattice has to be partitioned across independent nodes.

The language that we adopt is OpenACC [7], which offers a different approach w.r.t. CUDA or OpenCL, based on directives. Porting applications to OpenACC "simply" requires to annotate existing codes with specific "pragma" instructions, which identify functions to be executed on accelerators, and instruct the compiler on how to structure and generate code for a specific target accelerator [8]. Despite this apparent simplicity, the annotated code has to be carefully designed to obtain satisfactory performances, having in mind at least some of the hardware details which are common to most accelerators. As we will detail later the data layout is of paramount importance as well as a wise scheduling of data transfers between host and accelerator memories.

In this paper we describe the methodology adopted to reach our goal, describe the advantages in terms of portability and maintainability of the produced code, but we also quantitatively assess the performance price that one has to pay w.r.t. the corresponding code written using an architecture-specific programming environemt. This paper is structured as follows: in Sec. II and Sec. III we briefly review the LQCD methods and the OpenACC programming standard respectively, while in Sec. IV we describe the development process and provide the details of our implementation. In Sec. V we present our conclusions.

II. Lattice QCD

Lattice QCD (LQCD) is a nonperturbative regularization of Quantum Chromodynamics (QCD) which enables us to tackle some aspects of the QCD physics that would be impossible to systematically investigate by using standard perturbation theory, like for instance the confinement problem or chiral symmetry breaking.

For many years commodity systems have not been able to provide the computational power needed for LQCD simulations, and several generations of parallel machines have been specifically designed and optimized for this purpose [9]–[11].

Today, multi-core architecture processors are able to deliver several levels of parallelism providing large computing power that allow to tackle larger and larger lattices, and computations are commonly performed using large computer clusters of commodity multi- and many-core CPUs. Moreover, the use of accelerators such as GPUs has been successfully explored to boost performances of LQCD codes [12]. More generally, massively-parallel machines based on heterogeneous nodes combining traditional powerful multi-core CPUs with energy-efficient and fast accelerators are ideal targets for LQCD simulations.

The basic idea of LQCD is to discretize the continuum four dimensional space of QCD on a four dimensional discrete lattice, in such a way that continuum physics is recovered as the lattice spacing goes to zero. The discretization of the fermionic part of the action is not completely trivial and several strategies have been developed in the past: Wilson, staggered, domain-wall and overlap fermions (see e.g. [13]). In the following we will specifically refer to the staggered formulation, which is commonly adopted for the investigation of QCD thermodynamics.

The discretized problem can be studied by means of Monte-Carlo techniques and, in order to generate configurations with the appropriate statistical weight, the standard procedure is the Hybrid Monte Carlo algorithm (HMC, see e.g. [13]). HMC consists of the numerical integration of the equations of motion, followed by an accept/reject step. The computationally more intensive step of this algorithm is the solution of linear systems of the form $\not{D}\psi = \eta$, where ψ is the fermion field (defined at all lattice sites), η is a vector of Gaussian random numbers and $\not{D}$ is the discretized version of the Dirac operator; $\not{D}$ is a matrix with as many rows and columns as the total lattice size, which has typically $\mathcal{O}(10^5 - 10^6)$ sites. By using

an even-odd sorting of the lattice sites, the matrix $\not{D}$ can be written in block form

$$\not{D} = \begin{pmatrix} m\mathbb{1} & D_{oe} \\ D_{eo} & m\mathbb{1} \end{pmatrix}, \quad D_{oe}^{\dagger} = -D_{eo} \ ,$$

where m is the fermion mass. The even-odd preconditioned form of the linear system is $(m^2 - D_{eo}D_{oe})\psi_e = \eta_e$, where now ψ_e and η_e are defined on even sites only.

This is still a very large sparse linear problem, which can be conveniently solved by using the Conjugate Gradient method or, more generally, Krylov solvers. The common strategy of this class of solvers is the following: one starts from an initial guess solution, then iteratively improves its accuracy by using auxiliary vectors, obtained by applying D_{oe} and D_{eo} to the solution of the previous step.

From this brief description of the target algorithm it should be clear that, in order to have good performances, a key point is the availability of very optimized routines for the D_{oe} and D_{eo} operators. As a consequence these routines appear as natural candidates for a comparison between different code implementations.

Before going on to present the details about the code and the performances obtained, we notice that both D_{oe} and D_{eo} still have a natural block structure. The basic elements of these operators are $SU(3)$ matrices and this structure can be used, e.g., to reduce the amount of data that have to be transferred from the memory (the algorithm is strongly bandwidth limited).

III. OpenACC

OpenACC is a programming framework for parallel computing aimed to facilitate code development on heterogeneous computing systems [7]. It is a valuable tool in particular to simplify porting of existing codes.

Similarly to OpenCL, OpenACC provides a widely applicable abstraction of actual hardware, making it possible to run the same code across different architectures. Contrary to OpenCL, where specific functions (called *kernels*) have to be explicitly programmed to run in a parallel fashion (e.g. as GPU threads), OpenACC is based on pragma directives that help the compiler identify those parts of the source code that can be implemented as *parallel functions*.

OpenACC support for different hardware architectures relies on compilers; at this point in time there are only few OpenACC compilers (e.g. from PGI and Cray) and they mainly target GPU devices. One hopes however that, thanks to the OpenACC generality, compilers and run-time support will quickly become available for a growing set of different architectures. As an example, GCC (GNU Compiler Collection) is expected to support OpenACC from version 5.0 onwards. Following *pragma* instructions, the compiler generates one or more *kernel* functions – in the OpenCL sense – that run in parallel as a set of threads.

Listing 1 shows a simple example of the *saxpy* operation of the *Basic Linear Algebra Subprogram* (BLAS) set. This example contains the *pragma acc kernels* clause which identify the code to run on the accelerator; in this case the iterations of the for-loop are parallelized and the function execution is offloaded at run-time from the host-CPU to an attached accelerator (e.g. a GPU). More directives are available, allowing a finer tuning of the application. As an example, the number of threads launched by each device function and their grouping can be tuned by the *vector*, *worker* and *gang* directives, in a similar fashion as setting the number of *work-items* and *work-groups* in OpenCL.

Listing 1. Sample OpenACC code computing a *saxpy* function on vectors x and y. *Pragma* clauses instruct data transfers and identifies the region to run on the accelerator.

```
#pragma acc data copyin(x), copy(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];

 #pragma wait(1);
}
```

Data transfers between host and device memories are automatically generated, when needed, entering and exiting the annotated code regions. Even in this case data directives are available to allow the programmer to obtain a finer tuning, e.g. increasing performance by an appropriate ordering of the transfers. For example, in Listing 1 the clause *copyin(ptr)* copies the array pointed by *ptr* from the host memory into the accelerator memory before entering the following code region; while *copy(ptr)* perform the additional operation of copying it also back to the host memory after leaving the code region.

An asynchronous directive *async* is available as well, instructing the compiler to generate asynchronous data transfers or device function executions; a corresponding clause (i.e. *#pragma wait(queue)*) allows to wait for completion.

The reader may have noticed that OpenACC is similar to the OpenMP (Open Multi-Processing) language in several ways [14]; both environments are directive based, but OpenACC targets accelerators in general, while at this stage OpenMP targets mainly multi-core CPUs, even if the OpenMP project is clearly moving towards accelerators support.

Regular C/C++ or Fortran code, already developed and tested on traditional CPU architectures, can be annotated with OpenACC pragma directives (e.g. *parallel* or *kernels* clauses) to instruct the compiler to transform loop iterations into distinct threads, belonging to one or more functions to run on an accelerator. OpenACC is thus particularly well suited for scientific HPC software development for several reasons:

- it allows to maintain a single code version, able to run on multiple architectures;
- in contrast to the OpenCL's verbosity, the programming overhead to offload code regions to accelerators is limited to few pragma lines.
- the code annotated with OpenACC pragmas can be still compiled and run ignoring pragmas.

Detractors may argue that also OpenCL some years ago seemed to be a promising standard for the HPC community,

while nowadays its support seems declining; why should OpenACC be different? We can not be sure about a wide adoption of the OpenACC standard in HPC for the next years, neither we can be sure that most hardware manufacturers will support it. Despite of this, we believe it to be one of the best languages to use in this field at this point in time due to its favorable risk-benefit ratio, as it is based on a small set of changes w.r.t. a plain C/C++ or Fortran code. Indeed, if main hardware manufacturers drop their support for OpenACC in the future, one would still have a fully functional and updated reference version of the code written, for example, in plain C. Also, it would be reasonably straigthforward to port the code to a different directive based language, since it would probably be "just a matter" of translating pragma clauses. This is indeed what is in some sense expected for the near future. The fact that OpenACC mimics the widely known and used OpenMP standard [14] is not just a coincidence, OpenACC has been created and implemented by several members of the OpenMP ARB (Architecture Review Board) in order to address their immediate customer needs, but OpenMP and OpenACC are actively merging their specification while continuing to evolve. A first step at merging has been made with the release of OpenMP 4.0 and according to the OpenMP developers, OpenACC implementations can be considered to be a beta test of the OpenMP accelerator specification providing early implementation experience [15].

The similarities between the OpenACC and OpenMP standards are particularly relevant especially for the HPC community, where new technologies that can coexist with older ones have a greater chance of success than ones requiring complete buy-in at the beginning [4]. OpenMP is in fact considered a consolidated standard in the HPC community.

IV. Software Design and Implementation

As previously introduced, the development process started from two existing different code versions written in C++ and C++/CUDA, targeted respectively to CPUs and nVIDIA GPUs architectures.

Code development had started earlier with a C++ version targeted to CPU architectures; later a new implementation was derived, targeting GPUs in order to better exploit computational resources of GPU based HPC systems. After the fork, the two implementations were optimized independently for the two different architectures. After several development iterations, in particular due to optimizations, the two implementations started to diverge significantly. This increasing divergence led to an increasing difficulty in keeping the two versions consistent and up to date as long as new functions were added or modified.

The advent of the OpenACC language, introduced in Sec. III, was therefore seen as an opportunity to merge the two different versions in a single implementation which could ease the development and maintenance processes. Moreover, the use of OpenACC instead of CUDA would have increased the code portability (or at least it would not have affected it). On the other side, a negative consequence which has to be evaluated is the performance loss that one has to pay, moving from architecture specific codes to generic ones [16].

A. Initial Planning

In contrast to most scientific software development processes starting from scratch [3], the existence of two existing implementations, granted us the possibility to perform an initial requirements gathering phase and an analysis of the different optimization techniques which were adopted to obtain the best performances on both the architectures.

The main difference between the two available implementations was the memory data layout. In particular, to improve performance on GPUs, data was organized adopting the SoA (Structure of Array) data layout in the C++/CUDA implementation, while an AoS (Array of Structure) was still used in the CPU one.

Data layout is known to be highly important to allow memory coalescing in GPUs, where vectorization is performed on "long vectors" (e.g. 32 elements) [17]. In practice, threads executed in a single SIMD (Single Instruction Multiple Data) instruction need to access contiguous memory regions to fully exploit memory bandwidth and to hide memory latencies. Actually, also CPU devices should benefit from a SoA data layout when using vectorization [18], but this is rarely enforced.

The use of the SoA data layout seemed to be the best choice, but being a fundamental design decision with strong impact on the code structure and on its performance, experimental data was collected to support it. Apart from the data layout, also the fundamental data type had to be chosen. Most of the computations in a LQCD simulation involve complex arithmetic, thus complex data type could be used with built-in complex arithmetic functions, or otherwise, ordinary floating point data could be used with custom functions. Built-in functions are surely more convenient from the programming point of view, but may not be the best choice when looking for performance.

B. Analysis and design

As previously mentioned in Sec. II, the most heavily used data structures in this algorithm are arrays of complex valued 3×3 matrix (containing the $SU(3)$ matrices) and arrays of 3 complex valued component vectors (containing the fermions). We implemented a micro-benchmark in order to assess the performance of a $SU(3)$ matrix - fermion multiplication using the AoS or SoA data layouts and additionally, using double or double complex data types.

This benchmark was implemented using CUDA and plain C. It demonstrated significantly better results, on an nVIDIA K20 GPU, when using SoA data layout, w.r.t. AoS, while showing minor differences between the different data types. On an Intel Xeon E5-2620v2 CPU, using a naive implementation we obtained an opposite result, showing a better performance when using AoS. However, using pragma assisted vectorization, we were able to almost close the gap between the two implementations [1]. Finally, on a newer Intel Xeon E5-2630v3

[1] Vectorization is not possible when using AoS data layout

CPU (implementing AVX2 and FMA3 instructions), the SoA data layout has better performance for both the naive and in particular for the vectorized versions, w.r.t. the AoS one. Results are reported in Tab. I.

TABLE I
EXECUTION TIME $[ms]$ TO PERFORM 32^4 VECTOR-$SU(3)$ MULTIPLICATIONS IN DOUBLE PRECISION USING DIFFERENT DATA LAYOUTS AND TYPES.

Data Type	Layout	nVIDIA K20 GPU	Intel E5-2620v2 Naive	Intel E5-2620v2 Vect.	Intel E5-2630v3 Naive	Intel E5-2630v3 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

Since the CPU market is moving towards the adoption of wider vector instructions (e.g. AVX2, AVX-512), when using vectorization, we expect the SoA data layout to outperform the AoS one, by even higher factors, on future CPU processors.

Based on these results and considerations, we adopted the SoA data layout, for both fermions and $SU(3)$ matrices, organized as shown respectively in Fig. 1 and Fig. 2. Concerning the data type, we adopted the standard C99 double complex, as this allowed to use built-in functions for complex arithmetic without affecting negatively the performances.

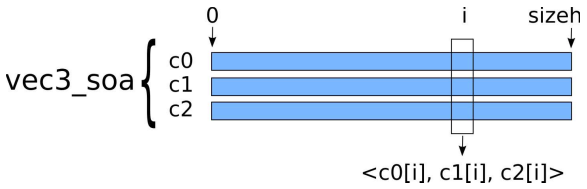

Fig. 1. Memory data layout for the data structures of vectors. Each vector component is a double precision complex value.

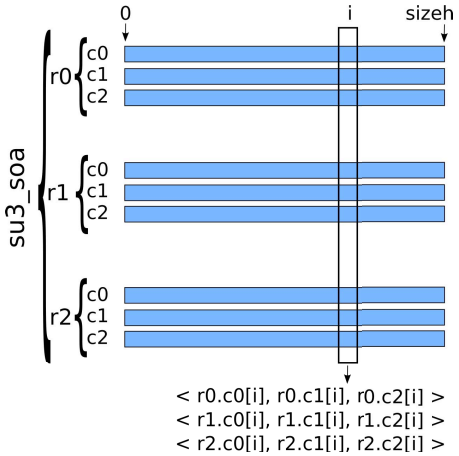

Fig. 2. Memory data layout for the data structures of $SU(3)$ matrices. Each matrix component is a double precision complex value.

Another fundamental design decision which would have affected code performance and maintainability, was between plain C and an appropriate C++ revision. Other languages were not an option since the developers involved in the project were experienced mainly in C/C++.

In the HPC community, the plain C99 language is still one of the most used and there are solid reasons for that [4]. Object Oriented (OO) languages in general are commonly avoided since the abstraction they provide limits the programmer visibility about what the software is actually doing from the point of view of memory usage and instruction execution. This limitation prevents the programmer from identifying code inefficiencies and to fine tune the software in her quest for performance [4].

The use of plain C was also driven by the fact that although OpenACC could be used in C++ codes, this language is not yet fully supported in accelerated code regions by available compilers; e.g. full STL (Standard Template Library) support is still work in progress.

An analysis of existing codes revealed that for both the GPU and CPU targeted versions no particular OO capabilities of C++ were used and despite of the use of C++ syntax, plain C99 could be easily used instead.

C. Implementation Phase

After this preliminary phase, the software development started porting the two functions accounting for the largest fraction of the computing time in a LQCD simulation. As mentioned in Sec. II these functions implement the D_{eo} and D_{oe} operators. The D_{eo} function reads even sites of the lattice and writes odd ones, while D_{oe} vice versa. The use of these two distinct functions grants the possibility to parallelize each of them on numbers of threads as large as half the lattice size, which can run independently.

Both the functions were implemented using plain C, avoiding indirect addressing which would undermine performances in a SIMD scenario. OpenACC directives were used to instruct the compiler to generate one GPU kernel function for each of them.

For both the CUDA and OpenACC versions, each GPU thread is associated to a single lattice site; consequently in the D_{eo} function all GPU threads are associated to even lattice sites, while in the D_{oe} function all GPU threads are associated to odd lattice sites. Therefore each kernel function operates on half of the lattice points.

Listing 2 contains a code snippet showing the beginning of the previously existing CUDA function implementing the D_{eo} operation. Note in particular the mechanism to reconstruct the x, y, z, t coordinates identifying the lattice point associated to the current thread, given the CUDA 3-dimensional thread coordinates (nt, nx, ny, nz are the lattice extents and $nxh = nx/2$).

Listing 3 shows the OpenACC version of the same part of the D_{eo} function. The four for loops, each iterating over one of the four lattice dimensions, and the pragma directive preceding each of them are clearly visible. In this case the explicit evaluation of the x, y, z, t coordinates is not needed, since we use here standard loop indices; however we can

Listing 2. Heading of the CUDA implementation of 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, idxh;

 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);
  ...
```

Listing 3. Heading of the OpenACC implementation of 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/DIMZ) vector(DIMZ)
  for(z=0; z<nz; z++) {
   #pragma acc loop independent gang(ny/DIMY) vector(DIMY)
   for(y=0; y<ny; y++) {
    #pragma acc loop independent vector(DIMX)
    for(hx=0; hx < nxh; hx++) {
     ...
```

still control the *thread blocks* size using the vector and gang clauses. In particular, DIMX, DIMY and DIMZ are the desired dimensions of the *thread blocks*. Since we execute the code on an nVIDIA GPU, as for the CUDA case, also in this case, each GPU thread is actually addressed by 3 coordinates.

D. Testing and Evaluation

After this second phase, for results testing and performance evaluation, we prepared a mini-app able to repeatedly call the D_{eo} and the D_{oe} functions, one after the other, using the OpenACC implementation or the CUDA one. Apart from the numerical results testing, a measurement was required at this stage to evaluate the performance loss w.r.t. the CUDA version. Portability and maintainability of the code was a major concern, but a performance loss higher than 50% would not have been considered acceptable. Notice that as previously mentioned, the performance of these functions roughly shape the performance of the whole software application.

The two implementations were compiled respectively with the PGI compiler, version 14.6, and the nVIDIA nvcc CUDA compiler, version 6.0. The mini-app code was run on a 32^4 lattice, using an nVIDIA K20m GPU; results are shown in Tab. II, where we list the sum of the execution times for the D_{eo} and D_{oe} operations in nanoseconds normalized per lattice site, for different choices of *thread block* sizes. All computations were performed using double precision floating point values.

TABLE II
EXECUTION TIME [$nsec$ PER LATTICE SITE] OF THE D_{eo} + D_{oe} FUNCTIONS, FOR THE CUDA AND OPENACC IMPLEMENTATIONS, USING FLOATING POINT DOUBLE PRECISION THROUGHOUT.

	D_{eo} + D_{oe} Funtions	
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

Execution times have a very mild dependence on the block size, which seems to influence differently the CUDA and OpenACC versions; this is probably due to the different loop unrolling policies (i.e. CUDA permits explicit loops unrolling, while in OpenACC it is a compiler choice) and the different register usage.

The execution times for the OpenACC implementation are in general slightly higher; if one considers the best *thread block* sizes both for CUDA and OpenACC, the latter is $\simeq 23\%$ slower. A slight performance loss w.r.t. CUDA was expected, given the higher level of the OpenACC language and its generality. In this respect, our results are very satisfactory, given the lower programming efforts needed to use OpenACC and the increased code maintainability given by the possibility to run the same code on CPUs or GPUs, by simply disabling or enabling pragma directives. Moreover, OpenACC code performance is expected to improve in the future also due to the rapid development of OpenACC compilers, which at the moment are yet in their early days.

E. Discarded iterations

We experimented with other potentially useful optimizations, e.g. combining the D_{eo} and D_{oe} routine in a single function D_{eoe} and mapping it onto a single GPU *kernel*, but the performance was roughly one order of magnitude lower, mainly because of overheads associated to excessive register spilling.

F. Further iterations

After the first development iteration we obtained a mini-app able to perform the D_{eo} and D_{oe} operations with interesting performances w.r.t. the previously existing CUDA version.

A new iteration started afterword, in order to nestle the OpenACC D_{eo} and D_{oe} functions into the previously existing C++ code to have the possibility to evaluate the new implementation in the context of a full LQCD simulation. This was accomplished implementing some auxiliary functions able to rearrange data stored in the previously used data structures to the new one and vice versa. To obtain this, the produced C functions annotated with OpenACC pragmas had to be called from the C++ implementation. This lead to a temporary hybrid application with very bad performances (due to data

rearrangements) which on the other side was very handy for the subsequent development iterations.

Further iterations were indeed accomplished porting new parts of the previously existing application to plain C annotated with OpenACC pragmas, in this hybrid application. Testing for numerical correctness could then be performed very easily at the end of every iteration, with respect to the previously existing versions.

The development of a complete LQCD simulation code fully based on C/OpenACC is now in progress. When all the functions are ported, all of them will be able to use the new data layout and to offload computations to the GPU where needed, so auxiliary functions used to rearrange data structures will be removed and performances will be finally measured again.

Eventually, in a new iteration, MPI (Message Passing Interface) will be introduced to obtain a multi-node implementation, able to split the lattice and the computation across multiple heterogeneous nodes and multiple accelerators. Thanks to the *#pragma acc data* clauses and asynchronous kernel execution, OpenACC will allow to fine tune data transfers between nodes, overlapping communications and computations.

From that point onwards the code will reach a production status and previous C++ versions will be decommissioned. Subsequent iterations will be performed to add new functions and/or to modify the software application behavior according to scientific needs.

V. Conclusions

In this work we discussed the development process of a scientific software application able to exploit modern, and future, heterogeneous HPC systems. We highlighted the software development characteristics associated to scientific HPC software applications and as a real use case we described the development of a LQCD simulation software. We presented the most important development phases and in particular we focused on the design and analysis necessary to find a tradeoff between portability, efficiency and maintainability.

In general, for portable HPC software applications, to chose the best performing data layout during the design and analysis phase has a strong impact on performances. Unfortunately it depends on the hardware architecture, but to develop a portable code, a single data layout has to be adopted. However, since processors, in general, are moving towards architectures featuring longer vector instructions, the use of the SoA data layout appear a good best practice for performance.

For HPC software applications, lower level languages, such as plain C, although lacking a lot of features available in OO languages, are commonly a better choice. These languages grant the programmer to more easily foresee how the code would map to machine instructions, allowing for finer optimizations. Furthermore such languages are also easier to vectorize from the compiler point of view.

We adopted plain C indeed and moreover, to offload computations to hardware accelerators, we used the OpenACC language. OpenACC is particularly well suited to the scientific HPC software development model, allowing for a single portable code implementation and for a synthetic way to express code parallelism. This translate to a cleaner code, resembling a pure plain C, apart from pragma instructions. Furthermore, these instructions resemble the widely known OpenMP pragmas and additionally, in the near future, the two standards will probably merge. Also from the performance point of view, the loss with respect to an hardware specific language seems to be affordable and well repaid by portability and maintainability. Performance loss may vary for different codes, but it should be of the same order, in particular for memory bounded codes.

Concerning the LQCD software application described in this work, its development is still ongoing, but a multi-node implementation with competitive performances, fully based on C/OpenACC and MPI, will soon be available.

References

[1] E. S. Mesh and J. S. Hawker, "Scientific software process improvement decisions: A proposed research strategy," in *Software Engineering for Computational Science and Engineering (SE-CSE), 5th International Workshop on*. IEEE, 2013, pp. 32–39.

[2] D. Kelly, "A Software Chasm: Software Engineering and Scientific Computing," *IEEE Software*, vol. 24, no. 6, pp. 120–119, Nov 2007.

[3] J. Segal, "Models of scientific software development," in *First International Workshop on Software Engineering in Computational Science and Engineering*, May 2008.

[4] V. R. Basili *et al.*, "Understanding the high-performance-computing community: A software engineer's perspective," *IEEE Software*, vol. 25, no. 4, pp. 29–36, 2008.

[5] E. Calore, S. F. Schifano, and R. Tripiccione, "A portable OpenCL Lattice Boltzmann code for multi-and many-core processor architectures," *Procedia Computer Science*, vol. 29, pp. 40–49, 2014.

[6] C. Bonati, G. Cossu, M. D'Elia, and P. Incardona, "QCD simulations with staggered fermions on GPUs," *Computer Physics Communications*, vol. 183, no. 4, pp. 853–863, 2012.

[7] "OpenACC directives for accelerators." [Online]. Available: http://www.openacc-standard.org/

[8] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC - First Experiences with Real-World Applications," in *Euro-Par 2012 Parallel Processing*, ser. LNCS, 2012, vol. 7484, pp. 859–870.

[9] H. Baier *et al.*, "QPACE: Power-efficient parallel architecture based on IBM PowerXCell 8i," *Computer Science - Research and Development*, vol. 25, no. 3-4, pp. 149–154, 2010.

[10] F. Belletti *et al.*, "Computing for LQCD: Ape NEXT," *Computing in Science and Engineering*, vol. 8, no. 1, pp. 50–61, 2006.

[11] P. Boyle *et al.*, "Overview of the QCDSP and QCDOC computers," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 351–365, March 2005.

[12] G. Egri *et al.*, "Lattice QCD as a video game," *Computer Physics Communications*, vol. 177, no. 8, pp. 631–639, 2007.

[13] T. DeGrand and C. DeTar, *Lattice methods for quantum chromodynamics*. World Scientific, 2006.

[14] S. Wienke, C. Terboven, J. Beyer, and M. S. Müller, "A pattern-based comparison of OpenACC and OpenMP for accelerator computing," in *Euro-Par 2014 Parallel Processing*, ser. LNCS, 2014, vol. 8632, pp. 812–823.

[15] "How does OpenMP relate to OpenACC?" 2015. [Online]. Available: http://openmp.org/openmp-faq.html#OpenACC

[16] E. Calore, S. Schifano, and R. Tripiccione, "On Portability, Performance and Scalability of an MPI OpenCL Lattice Boltzmann Code," in *Euro-Par 2014: Parallel Processing Workshops*, ser. LNCS, 2014, vol. 8806, pp. 438–449.

[17] R. Strzodka, "Abstraction for AoS and SoA layout in C++," in *GPU Computing Gems: Jade Edition*, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2011.

[18] "A Guide to Auto-vectorization with Intel C++ Compilers," 2012. [Online]. Available: https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers