Designing and Optimizing LQCD codes using OpenACC

Claudio Bonati[1], *Enrico Calore*[2], *Simone Coscetti*[1], *Massimo D'Elia*[1,3], *Michele Mesiti*[1,3], *Francesco Negro*[1], *Sebastiano Fabio Schifano*[2,4], *Raffaele Tripiccione*[2,4]

[1]INFN - Sezione di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
[2]INFN - Sezione di Ferrara, via Saragat 1, 44122 Ferrara, Italy
[3]Università di Pisa, Largo Bruno Pontecorvo, 3, 56127 Pisa, Italy
[4]Università di Ferrara, via Saragat 1, 44122 Ferrara, Italy

DOI: http://dx.doi.org/10.3204/DESY-PROC-2014-05/27

An increasing number of massively parallel machines adopt heterogeneous node architectures combining traditional multicore CPUs with energy-efficient and fast accelerators. Programming heterogeneous systems can be cumbersome and designing efficient codes often becomes a hard task. The lack of standard programming frameworks for accelerator based machines makes it even more complex; in fact, in most cases satisfactory performance implies rewriting the code, usually written in C or C++, using proprietary programming languages such as CUDA. OpenACC offers a different approach based on directives. Porting applications to run on hybrid architectures "only" requires to annotate existing codes with specific "pragma" instructions, that identify functions to be executed on accelerators, and instruct the compiler on how to structure and generate code for specific target device. In this talk we present our experience in designing and optimizing a LQCD code targeted for multi-GPU cluster machines, giving details of its implementation and presenting preliminary results.

1 Introduction

Lattice Quantum Chromodynamics (LQCD) simulations enable us 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 kernels of the LQCD algorithms are embarassingly parallel, however the challenge of designing and running efficient codes is not easy to met.

In the past years commodity processors were not able to provide the required computational power for LQCD simulations, and several generations of parallel machines have been specifically designed and optimized for this purpose [1, 2]. Today, multi-core architecture processors are able to deliver several levels of parallelism providing high 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 performamces of LQCD codes [3].

More generally, massively-parallel machines based on heterogeneous nodes combining traditional powerful multicore CPUs with energy-efficient and fast accelerators are ideal targets for LQCD simulations and are indeed commonly used. Programming these heterogeneous systems can be cumbersome, mainly because of the lack of standard programming frameworks for accelerator based machines. In most of the cases, reasonable efficiency requires that the code is re-written targeting a specific accelerator, using proprietary programming languages such as CUDA for nVIDIA GPUs.

OpenACC offers a different approach based on directives, allowing to port applications onto hybrid architectures by annotating existing codes with specific "pragma" directives. A perspective OpenACC implementation of an LQCD simulation code would grant its portability across different heterogeneous machines without the need of producing multiple versions using different languages. However the price to pay for code portability may be in terms of code efficiency.

In this work we explore the possible usage of OpenACC for LQCD codes targeting heterogeneous architectures, estimating the performance loss that could arise with respect to an architecture-specific optimized code. To pursue this goal, we wrote the functions accounting for most of the execution time in an LQCD simulation using plain C and OpenACC directives. As well known, the most compute intensive computational kernel is the repeated application of the Dirac operator (the so called D slash operator $\not{D}$). We wrote and optimized the corresponding routines and then compared the obtained performance of our C/OpenACC implementation with an already existing state-of-the-art CUDA program [4].

In Sec. 2 and Sec. 3 we briefly review the OpenACC programming standard and the LQCD methods respectively, while in Sec. 4 we provide the details of our implementation. In Sec. 5 we present our performance results.

2 OpenACC

OpenACC is a programming framework for parallel computing aimed to facilitate code development on heterogeneous computing systems, and in particular to simplify porting of existing codes. Its support for different architectures relies on compilers; although at this stage the few available ones target mainly GPU devices, thanks to the OpenACC generality the same code can be compiled for different architectures when the corresponding compilers and run-time supports become available.

OpenACC, like OpenCL, 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*. Following *pragma* instructions the compiler generates one or more *kernel* functions – in the OpenCL sense – that run in parallel as a set of threads.

OpenACC is similar to the OpenMP (Open Multi-Processing) language in several ways [5]; both environments are directive based, but OpenACC targets accelerators in general, while at this stage OpenMP targets mainly multi-core CPUs.

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.

Various directives are available, allowing fine tuning of the application. As an example, the number of threads launched by each device function and their grouping can be fine tuned by the *vector*, *worker* and *gang* directives, in a similar fashion as setting the number of *work-items* and *work-groups* in OpenCL. 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 more details on OpenACC see [6].

3 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.

The basic idea of LQCD is to discretize the continuum QCD on a four dimensional lattice, in such a way that continuum physics is recovered as the lattice spacing goes to zero. Fermions are problematic in this respect: a famous no-go theorem by Nielsen and Ninomiya can be vulgarized by saying that in the discretization procedure some of the properties of the fermions will be lost; they will be recovered only after the continuum limit is performed. Several ways to circumvent this difficulty exist and this is the reason for the current lattice fermions zoology: Wilson, staggered, domain-wall and overlap fermions (see e.g. [7]). 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. [7]). 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 η is a vector of Gaussian random numbers and $\not{D}$ is the discretized version of the Dirac matrix, whose dimension is given by the total lattice size, which is typically $\mathcal{O}(10^5 - 10^6)$. By using an even-odd sorting of the lattice sites, the matrix $\not{D}$ can be written in block form

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

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.

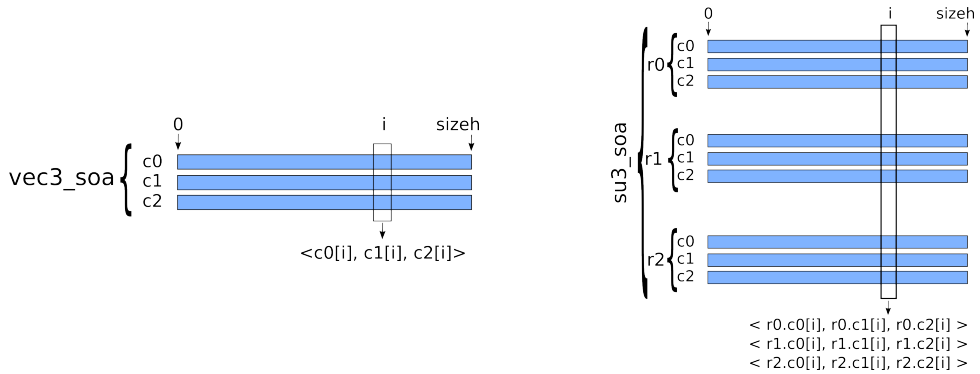

Figure 1: Memory allocation diagrams for the data structures of vectors (left) and $SU(3)$ matrices (right). Each vector or matrix component is a double precision complex value.

Before going on to present some 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). The starting point for the development of the OpenACC code was the CUDA code described in [4], which adopts all these specific optimizations.

4 Code Implementation

We coded the D_{eo} and the D_{oe} functions using plain C; OpenACC directives instruct the compiler to generate one GPU kernel function for each of them. The function bodies of the OpenACC version strongly resemble the corresponding CUDA kernel bodies, trying to ensure a fair comparison between codes which perform the same operations.

For both the CUDA and OpenACC versions, each GPU thread is associated to a single lattice site; 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. Consequently each kernel function operates on half of the lattice points.

Data structures are allocated in memory following the *SoA* (Structure of Arrays) layout to obtain better memory coalescing for both vectors and matrices as shown in Fig. 1. The basic data element of both the structures is the standard C99 double complex.

Listing 1 contains a code snippet showing the beginning of the 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 1: CUDA

```
__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);

  ...
```

Listing 2 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 still control the *thread blocks* size using the vector and gang clauses. In particular, DIM_BLK_X, DIM_BLK_Y and DIM_BLK_Z 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.

Listing 2: OpenACC

```
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++) {

          ...
```

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 register spilling.

5 Results and Conclusions

We prepared a benchmark code able to repeatedly call the D_{eo} and the D_{oe} functions, one after the other, using the OpenACC implementation or the CUDA one. The two implementations

were compiled respectively with the PGI compiler, version 14.6, and the nVIDIA nvcc CUDA compiler, version 6.0.

The benchmark code was run on a 32^4 lattice, using an nVIDIA K20m GPU; results are shown in Tab. 1, where we list the sum of the execution times of the D_{eo} and D_{oe} operations in nanoseconds per lattice site, for different choices of *thread block* sizes. All computations were performed using double precision floating point values.

	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

Table 1: Execution time in (nsec per lattice site) of the D_{eo} + D_{oe} functions, for the CUDA and OpenACC implementations running on an nVIDIA K20m GPU and using floating point double precision throughout.

Execution times have a very mild dependence on the block size and 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 with respect to CUDA is expected, given the higher level of the OpenACC language. 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.

The development of a complete LQCD simulation code fully based on OpenACC is now in progress.

References

[1] H. Baier, et al. *Computer Science - Research and Development*, 25(3-4):149–154, 2010. DOI:10.1007/s00450-010-0122-4.

[2] F. Belletti, et al. *Computing in Science and Engineering*, 8(1):50–61, 2006. DOI:10.1109/MCSE.2006.4.

[3] G.I. Egri, et al. *Computer Physics Communications*, 177(8):631–639, 2007. DOI:10.1016/j.cpc.2007.06.005.

[4] C. Bonati, et al. *Computer Physics Communications*, 183(4):853–863, 2012. DOI:10.1016/j.cpc.2011.12.011.

[5] S. Wienke, et al. In *Lecture Notes in Computer Science*, volume 8632 LNCS, pages 812–823, 2014. DOI:10.1007/978-3-319-09873-9-68.

[6] `http://www.openacc-standard.org/`.

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