https://doi.org/10.1051/epjconf/201817509008

Portable LQCD Monte Carlo code using OpenACC

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

[1] *Università di Pisa and INFN Sezione di Pisa, Largo Pontecorvo 3, I-56127 Pisa, Italy*
[2] *Università degli Studi di Ferrara and INFN Sezione di Ferrara, Via Saragat 1, I-44122 Ferrara, Italy*
[3] *Jülich Supercomputing Centre, Forschungszentrum Jülich, 52428 Jülich, Germany*

Abstract. Varying from multi-core CPU processors to many-core GPUs, the present scenario of HPC architectures is extremely heterogeneous. In this context, code portability is increasingly important for easy maintainability of applications; this is relevant in scientific computing where code changes are numerous and frequent. In this talk we present the design and optimization of a state-of-the-art production level LQCD Monte Carlo application, using the OpenACC directives model. OpenACC aims to abstract parallel programming to a descriptive level, where programmers do not need to specify the mapping of the code on the target machine. We describe the OpenACC implementation and show that the same code is able to target different architectures, including state-of-the-art CPUs and GPUs.

1 Introduction

The use of processors based on multi- and many-core architectures is common practice in High Performance Computing (HPC). Many variants of these processors exist, differing mainly in the number and architecture of the cores.

Conventional CPUs accommodate tens of fat cores sharing a large on-chip cache. These cores include several levels of caches and complex control structures, able to perform hardware optimization techniques (branch-speculation, instruction scheduling, register renaming, etc). Vector instructions are also supported by these cores, with a moderate level of data parallelism: 2 to 4 vector elements are processed by one vector instruction. This architecture is reasonably efficient for many type of regular and non-regular applications and convey a level of performance of the order of hundreds of GigaFlops per processor. On the other side of the spectrum we have Graphics Processor Units (GPU), available as accelerator boards attached to conventional CPUs. GPUs incorporate thousands of slim cores able to efficiently support regular streams of computation, and reach performances of the order of many TeraFlops. GPUs are extremely aggressive in terms of data-parallelism, implementing vector units with large vector sizes (16 and 32 words are presently available options).

Halfway between these two architectures, we find the Intel *Many Integrated Cores* (MIC) architecture based on several tens of slim cores. In this case, cores are similar to their fat counterparts, but their design has been simplified removing many hardware control structures (instruction scheduler,

[⋆] Speaker, e-mail: g.silvi@fz-juelich.de register renaming, etc) and adopting wider vector units, able to process up to 4 or 8 vector elements in parallel.

Today computing centers have not reached a common consensus on the "best" processor option for HPC systems. Architecture choices are not driven only by performances but also by cost of ownership and energy aspects which are becoming increasingly critical parameters[1].

In this scheme, the development of applications would greatly benefit from the availability of an individual code version, written in an appropriate programming framework, able to offer portability in terms of code and performance across several architectures. A single code version is of great convenience notably for scientific applications, where code changes and development iterations are not rare, so keeping several architecture-specific code versions up-to-date is a laborious and error prone effort[2].

In this work we describe the implementation of a Lattice QCD (LQCD) Monte Carlo code with tree-level improved gauge action and stout-smeared action for staggered fermions designed to be efficient and portable across several processor architectures.

Lattice QCD simulations is a typical and well known HPC grand challenge, where physics results are strongly limited by available computational resources[3, 4]; over the years, several generations of parallel machines, optimized for LQCD, have been developed[5, 6], while the development of LQCD codes running on many core architectures, in particular GPUs, has seen large efforts in the last decade [7–9]. Our target is to have a single code able to run on several processors without any major code change while looking for an acceptable trade-off between portability and efficiency[10]. OpenACC has been chosen as programming model, as it currently has a larger compiler support, in particular targeting NVIDIA GPUs, which are widely used in HPC clusters and for scientific computations. The migration of our code to OpenMP4, if needed, as soon as compiler support becomes more mature, is expected to be a simple effort.

2 Numerical challenges

The fundamental problem of LQCD is the evaluation of expectation values of given functions of the fields, $O[U]$, that is integrals of the form

$$\int \mathscr{D}U\mathscr{D}\phi\mathscr{D}H O[U] \exp\left(-\frac{1}{2}H^2 - S_g[U] - \phi^\dagger M[U]^{-1/4}\phi\right) \tag{1}$$

where the exponent S_g is the discretization of the action of the gauge fields (usually written as a sum of traces of products of $U_\mu(n)$ along closed loops) and $\det(M)$ describes the gluon-quark interaction. Here, $M[U]$ is a large and sparse structured matrix (i.e. containing both color and space-time indexes).The momenta term is a shorthand to indicate the sum of $-\mathrm{Tr}(H_\mu(n)^2)/2$ over the whole lattice.

The most time-consuming single step of the whole algorithm is the solution of a linear system

$$M[U]\varphi = b \tag{2}$$

This calculation is needed to compute the forces appearing in the equations of motion and also to evaluate ΔS, and one usually employ Krylov solvers. In the case of staggered fermions, corresponding to (1), it is customary to use the so-called Rational HMC (RHMC) algorithm[11], in which the algebraic matrix function appearing in (1) is approximated by a rational function. In this case one replaces (2) by r equations (r is the order of the adopted approximation)

$$(M[U] + \sigma_i)\varphi_i = b\ , \quad i \in \{1, \ldots, r\}\ , \tag{3}$$

	Xeon E5-2630 v3	Xeon E5-2697 v4	K80-GK210	P100
Year	2014	2016	2014	2016
Architetcure	Haswell	Broadwell	Kepler	Pascal
#physical-cores / SMs	8	18	13 × 2	56
#logical-cores / CUDA-cores	16	36	2496 × 2	3584
Nominal Clock (GHz)	2.4	2.3	562	1328
Nominal DP performance (Gflops)	≈ 300	≈ 650	935 × 2	4759
LL cache (MB)	20	45	1.68	4
Total memory supported (GB)	768	1540	12 × 2	16
Peak mem. BW (ECC-off) (GB/s)	69	76.8	240 × 2	732

Table 1. Selected hardware features of some of the processors used in this work: the Xeon-E5 systems are two recent multi-core CPUs based on the Haswell and Broadwell architecture, the K80 GPU is based on the *Kepler* architecture while the P100 GPU adopts the *Pascal* architecture.

where the real numbers σ_i are the poles of the rational approximations. These equations can again be solved by Krylov methods: by exploiting the shift-invariance of the Krylov subspace it is possible then to write efficient algorithms that solve all the equations appearing in (3) at the same time, using, at each iteration, only one matrix-vector product[12].

For most of the discretizations adopted in QCD (and in particular for the one we use, the so-called "staggered fermions), the matrix $M[U]$ can be written in block form

$$M = m\,I + \begin{pmatrix} 0 & D_{oe} \\ D_{eo} & 0 \end{pmatrix}, \qquad D_{oe}^{\dagger} = -D_{eo}\ ; \tag{4}$$

matrices D_{oe} and D_{eo} connect only even and odd sites. It is thus convenient to use even/odd preconditioning[13]; in this case, (2) is replaced by:

$$(m^2\,I - D_{eo}D_{oe})\varphi_e = b_e; \tag{5}$$

φ_e is defined only on even sites and the matrix is positive definite (because of (4)), so we can use the simplest of the Krylov solvers: the conjugate gradient.

Over the years, many improvements of this basic scheme have been developed; these are instrumental in reducing the high computational cost of actual simulations but their implementation is straightforward, once the basic steps of the "naive" code are ready.

2.1 Implementation

Intel multi-core CPUs and NVIDIA GPUs considered in this work are listed in Table 1 together with details of the systems.

We have developed a mini-application benchmark of the Dirac operator[14] to initially assess the performance level achievable with OpenACC. This operator is the fundamental building block of the Krylov solver, commonly accounting for at least 40% of the running time, and reaching up to 80% in low temperature configurations. The Dirac operator make use of three functions: deo, doe (corresponding respectively to the application of functions D_{eo} and D_{oe} defined in (4)) and a *zaxpy*-like function which is quite negligible in terms of execution time. From this early tests the performance of the OpenACC versions of the double precision deo and doe functions were found to be comparable

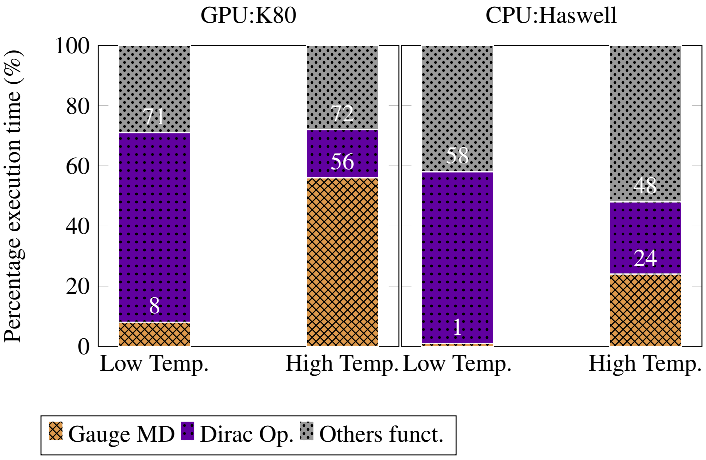

Figure 1. Percentage of the execution time of a selection of computationally heavy steps of our OpenACC code on two different architectures for low and finite temperature simulations.

with the CUDA ones[14]. This promising start was a strong indication that also for LQCD the higher portability of the OpenACC implementation is not associated with a severe loss of performance, and propel us to proceed to an OpenACC implementation of the full RHMC code. As a side benefit, the use of the OpenACC programming model greatly simplified the implementation of algorithmic improvements.

Implementation of these new features started with the coding and testing of the improvements on a single threaded version. Subsequently to the validation of the algorithm, the acceleration is switched on by annotating the code with #pragma directives. Finally a further correctness check was performed to check for errors in the OpenACC directives. In order to have a more readable code, the most complex kernels have been divide in several functions. Small functions can be used in kernels if declared as static inline, but for larger ones we had to use the routine seq OpenACC directive.

We have followed two different approaches to parallelize kernels. Those using data belonging to nearest (and/or next-to-nearest) neighbors have been parallelized via the #pragma acc loop directive on 4 nested loops, one for each dimension. This allows to use 3D thread blocks, which should improve data reuse between threads thus reducing bandwidth requirements, which is the major performance bottleneck. The other kernels, i.e. the ones performing only single-site operations, have been parallelized using a single cycle running on the lattice sites.

After the implementation of a first full working OpenACC simulation, we implemented various optimization iterations, in particular for the performance critical steps. These include not only the Dirac operator, but also the gauge part of the molecular dynamics steps, since their relative impact on the overall execution time is not neglegible. This is shown in Fig.1 in particular for the high temperature case.

3 Performance analysis

We test the performance of our code on different architectures using two different benchmarks taking into account the most computational expensive parts of the code. The first benchmark evaluates the performance of the Dirac operator and the second evaluates the performance of the gauge part of the

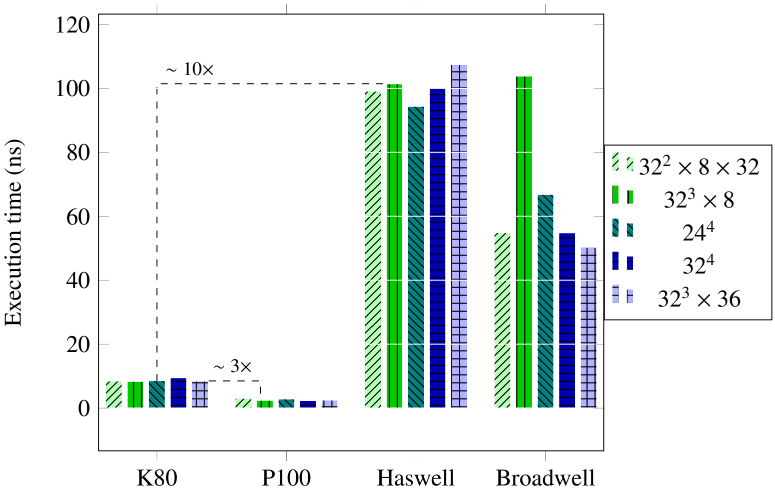

Figure 2. Measured execution time per lattice site [ns] for the Dirac operator for several architectures and several typical lattice sizes in double precision.

molecular dynamics step. The execution time per site of the Dirac operator for different lattice sizes in shown in Fig. 2. Exactly the same code version has been run on all platforms without requiring any change; it has been just re-compiled with different flags instructing the PGI 16.10 compiler which architecture to target and which tile dimensions would best fit for each of them.

We tested two different NVIDIA GPUs: the K80 based on the Kepler architecture and the P100 board based on the Pascal architecture. For the K80 the single precision version takes $\approx 4ns$ per lattice site, while the double precision version requires $\approx 8.5ns$. We measure on the P100 $\approx 1.5ns$ for single and $\approx 2.5ns$ for double precision, improving approximately by a factor 3× over the K80. This results perfectly scales with architecture performance of P100 that has ≈ 4.3× more cores and ≈ 3× more memory bandwidth, see Table 1.

For the Intel CPUs, we have compared two different processors, the 8-core E5-2630v3 CPU based on Haswell architecture, and the 18-core E5-2697v4 CPU based on Broadwell. Due to computing resources of the CPUs being 3× lower than on GPUs, see Table 1, a performance drop is expected. However, the actual performance drop measured on both CPUs is much larger than the expected theoretical figure; indeed time per site is approximately 10× or larger on the Haswell than on one K80 GPU. The Broadwell performs approximately a factor 2× better compared to Haswell, at least for some lattice sizes. We have identified two main reasons for this non-optimal behavior, and both of them point to some still immature features of the PGI compiler when targeting x86 architectures, that – we expect – should be soon resolved:

- **Parallelization** - the compiler is only able to split outer-loops across different threads, while inner-loops are executed serially or vectorized within each thread. This explains why on the Broadwell CPU running on a lattice $32^2 \times 8 \times 32$ we have a performance 2× better than for a $32^3 \times 8$ lattice, which has the same volume but allows the latter to split the outer loop only on 8 threads.
- **Vectorization** - as reported by the compilation logs, the compiler fails to vectorize the `deo` and `doe` functions computing the Dirac operator. It states to be unable to vectorize due to the use of "mixed data-types". To verify if this is related to the way we have coded these functions, we have translated

the OpenACC pragmas into the corresponding OpenMP ones – without changing the C code – and compiled using the Intel compiler (version 17.0.1). In this case the compiler succeeds in vectorizing the two functions, running a factor 2× faster compared to the OpenACC version compiled by PGI compiler.

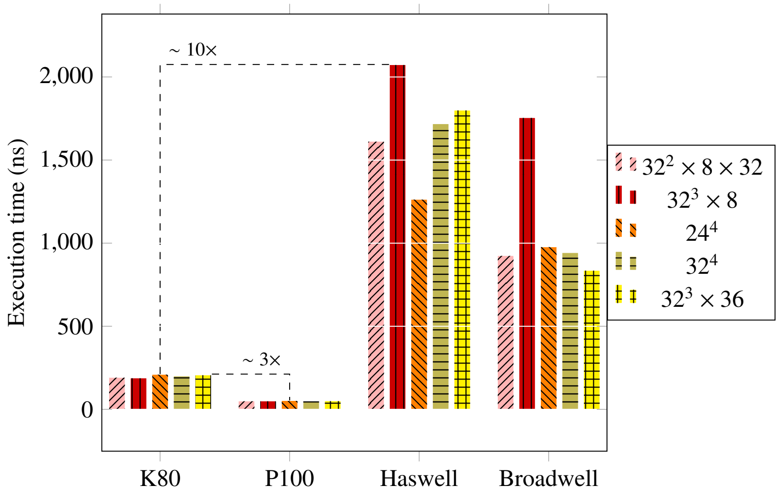

Figure 3. Measured execution time per lattice site [ns] for the pure gauge part of Molecular Dynamics evolution for several processors and lattice sizes in double precision.

Fig.3 shows the execution time of the gauge part of the molecular dynamics step. As already remarked this is one of the two most time-consuming steps. It is possible to notice that the update time per site is quite stable for the different lattice sizes tried. Going from the NVIDIA K80 to the P100 the time improves by a factor ≈ 3×, while between Haswell and Broadwell we have roughly a factor ≈ 1.5× / 2.0×.

Further we mention that we have also been able to compile and run our code on an AMD FirePro W9100 GPU and on the latest version of the Intel Xeon Phi, the Knights Landing (KNL). However, for this architecture, results are still preliminary. In more details, the compiler itself crashes when compiling the code for the AMD GPU for some specific lattice sizes; for the KNL, specific compiler support is still not present. This processor is still able to run the code compiled for the Haswell architecture, although not using the 512-bit vectorization. These problems forbid us to perform a systematic comparison of performance for these architectures. Once again, we believe that this is due to some immaturity of the compiler, for which we expect will be resolved in future versions.

Fig.4 addresses the question of the efficiency costs (if any) of our architecture-portable code; it compares the execution time for a *full* Monte Carlo step (in double precision) of the OpenACC code and a previously developed CUDA implementation[9], optimized for NVIDIA GPUs. Although the two codes are not exactly in a one to one correspondence, the implementations are similar enough to make the test quantitatively meaningful. One immediately sees that the performances of the two implementations are comparable and the use of OpenACC is not associated to a dramatic performance loss, the differences between the execution times of the two versions being of the order of $10 \div 20\%$.

The worst case is the one of the $32^3 \times 8$ lattice, in which OpenACC is about 25% slower than CUDA. Since we are comparing an high-level version of the code with one specifically developed for

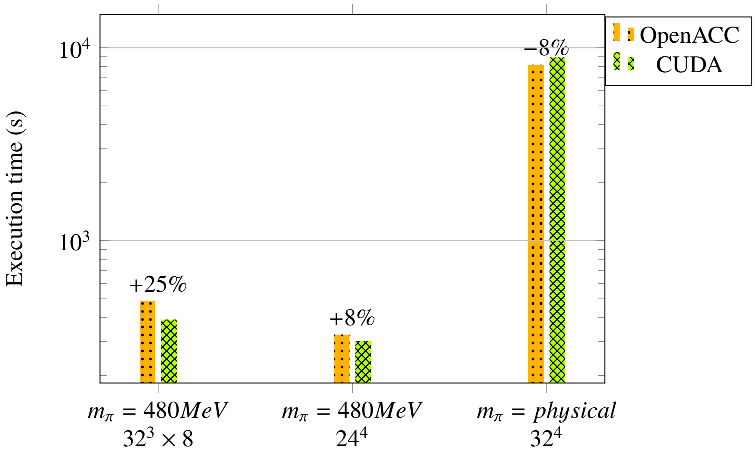

Figure 4. Execution time [sec] of a full trajectory of a complete Monte Carlo simulation for several typical physical parameters, running on one GPU of a NVIDIA K80 system. We compare the OpenACC code developed in this paper and and earlier GPU-optimized CUDA code [9]. Here we use the standard Wilson action and unimproved staggered fermions as the CUDA code does not support the action improvements available in the OpenACC version.

NVIDIA GPUs, this would not be a dramatic loss, however in this case the comparison is also not completely fair. Indeed for high temperature simulations the gauge part of the Molecular Dynamic step starts to be the computationally heaviest task and, in the CUDA implementation, part of it had been explicitly hard coded in single precision.

For the low-temperature test cases the differences between the CUDA and the OpenACC implementation are much smaller; furthermore, in one case the OpenACC version is the fastest one. A possible explanation of this behavior is the following: in the CUDA version unidimensional blocks are used to parallelize the Dirac operator, while in the OpenACC implementation three-dimensional block structures are adopted, that fit more the larger cache of recent GPUs and, especially on larger lattices, improves data reusage.

4 Concluding Remarks

In this work we have developed a full state-of-the-art production-grade code for Lattice QCD simulations with staggered fermions, using the OpenACC directive-based programming model. Our implementation does include all steps of a complete simulation, and most of them run on accelerators, minimizing the transfer of data to and from the host. It has been used the PGI compiler, which supports the OpenACC standard and is able to target almost all current architectures relevant for HPC computing, although with widely different levels of maturity and reliability. Exactly the same code runs successfully on NVIDIA many-core GPUs and Intel multi-core CPUs, and for both architectures we have measured roughly comparable levels of efficiency. Also, the performance of the complete code is roughly the same as that of an equivalent code, specifically optimized for NVIDIA GPUs and written in the CUDA language. Our code also runs on AMD GPUs and on the KNL Intel Phi processor, even if the compilation and run-time environment for these processors is still unable to deliver production-grade codes; in these cases there are strong indications that these problems come from a residual immaturity of the compilation chain expected to be resolved soon.

Some further comments are in order: i) using a directive-based programming model, we are able to target different computing platforms presently used for HPC, avoiding to rewrite the code when moving from one platform to another; ii) the OpenACC standard provides a good level of hardware abstraction requiring the programmer to only specify the function to be parallelized and executed on the accelerator; the compiler is then able to exploit the parallelism according to the target processor, hiding from the programmer hardware optimization details; iii) the OpenACC code has roughly the same level of performance of that implemented using a native language such as CUDA for NVIDIA GPUs, allowing to efficiently exploit the computing resources of the target platform.

References

[1] O. Villa, D.R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero et al., *Scaling the power wall: a path to exascale*, in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for* (IEEE, 2014), pp. 830–841

[2] 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 open acc: The case of lqcd*, in *Software Engineering for High Performance Computing in Science (SE4HPCS), 2015 IEEE/ACM 1st International Workshop on* (IEEE, 2015), pp. 9–15

[3] C. Bernard, N. Christ, S. Gottlieb, K. Jansen, R. Kenway, T. Lippert, M. Lüscher, P. Mackenzie, F. Niedermayer, S. Sharpe et al., Nuclear Physics B-Proceedings Supplements **106**, 199 (2002)

[4] G. Bilardi, A. Pietracaprina, G. Pucci, F. Schifano, R. Tripiccione, *The potential of on-chip multiprocessing for QCD machines*, in *International Conference on High-Performance Computing* (Springer, 2005), pp. 386–397

[5] P.A. Boyle, D. Chen, N.H. Christ, M. Clark, S. Cohen, Z. Dong, A. Gara, B. Joo, C. Jung, L. Levkova et al., *QCDOC: a 10 teraflops computer for tightly-coupled calculations*, in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (IEEE Computer Society, 2004), p. 40

[6] H. Baier, H. Boettiger, M. Drochner, N. Eicker, U. Fischer, Z. Fodor, A. Frommer, C. Gomez, G. Goldrian, S. Heybrock et al., Computer Science-Research and Development **25**, 149 (2010)

[7] N. Cardoso, P. Bicudo, Journal of Computational Physics **230**, 3998 (2011)

[8] T.W. Chiu, T.H. Hsieh, Y.Y. Mao, T. Collaboration et al., Physics Letters B **702**, 131 (2011)

[9] C. Bonati, G. Cossu, M. D Elia, P. Incardona, Computer Physics Communications **183**, 853 (2012)

[10] M.G. Lopez, V.V. Larrea, W. Joubert, O. Hernandez, A. Haidar, S. Tomov, J. Dongarra, *Towards achieving performance portability using directives for accelerators*, in *Accelerator Programming Using Directives (WACCPD), 2016 Third Workshop on* (IEEE, 2016), pp. 13–24

[11] M. Clark, A. Kennedy, Z. Sroczynski, Nuclear Physics B-Proceedings Supplements **140**, 835 (2005)

[12] B. Jegerlehner, arXiv preprint hep-lat/9612014 (1996)

[13] T.A. DeGrand, P. Rossi, Computer Physics Communications **60**, 211 (1990)

[14] 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 open acc: The case of lqcd*, in *Software Engineering for High Performance Computing in Science (SE4HPCS), 2015 IEEE/ACM 1st International Workshop on* (IEEE, 2015), pp. 9–15