HAL
open science

Automated Code Generation for Lattice QCD Simulation

Denis Barthou, Gilbert Grosdidier, Konstantin Petrov, Michael Kruse, Christine Eisenbeis, Olivier Pène, Olivier Brand-Foissac, Claude Tadonki, Romain Dolbeau

To cite this version:

Denis Barthou, Gilbert Grosdidier, Konstantin Petrov, Michael Kruse, Christine Eisenbeis, et al.. Automated Code Generation for Lattice QCD Simulation. [Research Report] RR-8417, INRIA. 2013, pp.13. hal-00918812

HAL Id: hal-00918812

https://inria.hal.science/hal-00918812v1

Submitted on 15 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automated Code Generation for Lattice QCD Simulation

Denis Barthou, Gilbert Grosdidier, Konstantin Petrov, Michael Kruse, Christine Eisenbeis, Olivier Pène, Olivier Brand-Foissac, Claude Tadonki, Romain Dolbeau

RESEARCH REPORT
N° 8417
December 15, 2013
Project-Team Parall

ISSN 0249-6399 ISRN INRIA/RR--8417--FR+ENG

Automated Code Generation for Lattice QCD Simulation

Denis Barthou*, Gilbert Grosdidier†, Konstantin Petrov‡, Michael Kruse‡, Christine Eisenbeis‡, Olivier Pène§, Olivier Brand-Foissac§, Claude Tadonki¶, Romain Dolbeau‖

Project-Team Parall

Research Report n° 8417 — December 15, 2013 — 10 pages

Abstract: Quantum Chromodynamics (QCD) is the theory of strong nuclear force, responsible of the interactions between sub-nuclear particles. QCD simulations are typically performed through the lattice gauge theory approach, which provides a discrete analytical formalism called LQCD (Lattice Quantum Chromodynamics). LQCD simulations usually involve generating and then processing data on petabyte scale which demands multiple teraflop-years on supercomputers. Large parts of both, generation and analysis, can be reduced to the inversion of an extremely large matrix, the so-called Wilson-Dirac operator. For this purpose, and because this matrix is always sparse and structured, iterative methods are definitely considered. Therefore, the procedure of the application of this operator, resulting in a vector-matrix product, appears as a critical computation kernel that should be optimized as much as possible. Evaluating the Wilson-Dirac operator involves symmetric stencil calculations where each node has 8 neighbors. Such configuration is really hindering when it comes to memory accesses and data exchanges among processors. For current and future generation of supercomputers the hierarchical memory structure make it next to impossible for a physicist to write an efficient code. Addressing these issues in other to harvest an acceptable amount of computing cycles for the real need, which means reaching a good level of efficiency, is the main concern of this paper. We present here a Domain Specific Language and corresponding toolkit, called QIRAL, which is a complete solution from symbolic notation to simulation code.

Key-words: LQCD, HPC, code generation, accelerator, parallel, matrix, linear system

* University of Bordeaux, LaBRI / INRIA Bordeaux Sud-Ouest, denis.barthou@inria.fr
† University of Paris Sud, Laboratoire de lAccelerateur Lineaire, gilbert.grosdidier@lal.fr
‡ INRIA Saclay, firstname.name@inria.fr
§ University of Paris Sud, Laboratory of Theoretical Physics, firstname.name@lpt.fr
¶ Mines ParisTech - CRI, claude.tadonki@mines.paristech.fr
‖ CAPS Entreprise, romain.dolbeau@caps-entreprise.com

RESEARCH CENTRE SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Automated Code Generation for Lattice QCD Simulation

Résumé : Ce travail concerne la génération de code à partir d'un formalisme exprimé en latex. Nous présentons notre méthodologie et sa mise en oeuvre, puis nous illustrons son fonctionnement sur de nombreux examples.

Mots-clés : LQCD, génération de code, accélérateur, parallélisme, matrice, système linéaire

I. Introduction

In a context of fast moving parallel architectures, the design of an optimized LQCD simulation, and in particular of an efficient inversion function is complex. This requires to design, select and combine iterative methods and preconditioners adapted to the problem and the target architecture, to optimize data layout and organize parallelism between nodes, cores, accelerators and SIMD units. In order to harness all resources of the hardware, orchestrating the work on many cores and accelerators, using different levels of parallelism, complex memory hierarchies and interconnect networks takes a large part of the tuning time, often at the expense of the exploration of new algorithms/preconditioners. Indeed, testing new methods can only be achieved with large enough data sets, requiring efficient parallel codes. Several codes and libraries have been designed for Lattice QCD (tmLQCD [**?**] as part of the European Twisted Mass Collaboration or Chroma [**?**] as part of the USQCD effort to name a few) and many works have been proposed on code optimization for Lattice QCD, among them recent works on Blue Gene/Q [**?**], Intel Xeon Phi [**?**] or clusters of GPUs [**?**]. While taking architectural features into account is crucial for high performance simulations, the key to exascale simulations also lies in new algorithms, new data layouts and communication patterns. However, designing new iterative methods, combining existing ones, changing data layout within these frameworks and tools is difficult and requires a significant code rewriting effort. This clearly hinders the adaptation of code to new parallel machines, limiting performance and the expected scientific results.

This paper proposes a domain-specific language (DSL), QIRAL, for the description of Lattice QCD simulations, and its compiler to generate parallel code. A domain-specific language can help to separate the high level aspects of the simulation from machine-dependent issues. The contribution of QIRAL is to address this twofold challenge:

- Propose to physicists a domain-specific language expressive enough to enable the description of different models and algorithms, and more importantly, expressive enough to enable algorithmic exploration by composing different algorithms and preconditioners as well as the design of new algorithms.
- Generate from this description efficient codes for parallel machines. Explicit parallelism and data layout are automatically generated and can be guided by the user. The code generated by QIRAL targets shared memory parallel machines, corresponding to one node of larger Lattice QCD simulations. This code uses OpenMP and a library for efficient SIMD operations.

With a higher level description of the Lattice QCD formulation, it becomes easier to try new algorithmic ideas, the high level code is easier to maintain and develop, and therefore makes numerical simulation accessible to a large number of users, not necessarily high performance computing experts. We show on several architectures, from Nehalem-EX with 128 cores to the Xeon Phi accelerator that the code generated with QIRAL competes in terms of parallel efficiency and performance with tmLQCD, while QIRAL provides an easier framework for the writing of algorithms and the adaptation to new architectures.

This paper is organized as follows: first we describe the DSL in Section II, describe the high-level compiler in Section III. Then the optimizations for locality, parallelism and SIMDization are presented in Section IV. Comparisons with related works are in Section V and experimental results, comparing with tmLQCD and describing strong scalability are shown in Section VI.

The whole project, under the name PetaQCD [**?**], was partly funded by a grant from ANR, through the program COSINUS-2008, from 2009 up to 2011.

II. The QIRAL Domain-Specific Language

As one of the purposes of the QIRAL DSL is to give scientists a familiar tool to describe the problem in scope, it makes sense to take an existing system of symbolic notation as the basic language. There are two such systems in most disciplines, LaTeXand Mathematica. While we are not attached to a particular one, we chose to use LaTeX-like syntax where certain additional macros have been defined. Therefore the QIRAL description can be processed either using the QIRAL compiler to produce program source code or alternatively, by the LaTeXtypesetter to produce its documentation, as shown in Figure 1. This means we revive the principle of *literate programming* coined by Donald Knuth [**?**]. For instance, the algorithm in Figure 4 is a QIRAL program included into this document as processed by LaTeX. The description of the language given in the following complements a description previously presented by the author's [**?**].

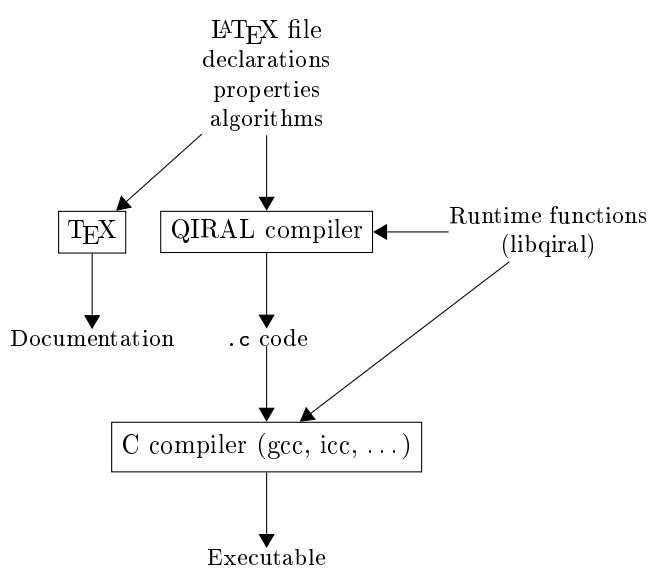

Fig. 1. QIRAL compiler

QIRAL is an array language for linear algebra, dedicated to the manipulation of sparse matrices defined through tensor products and direct sums of dense matrices as they appear in

Lattice QCD. The language relies on specificities of Lattice QCD: the inversion computation is a stencil computation, on a regular, 4D Cartesian mesh (the lattice). The Dirac operator, used for this inversion, is a sparse but regularly structured matrix that can be seen as a diagonal of dense matrices. This operator transforms values from the lattice into new values. These values are made of 12 complex values, indexed by their spin (4 indices) and color (3 indices). Hence for a $24^3 \times 48$ lattice, the Dirac operator is a matrix of $(24^3 \times 48 \times 12)^2$ complex values. As it is sparse, its structure carries the parallelism of the computation, hence the language and the compiler captures this structure through the operators building the matrix.

Elements of the language are declarations, equations, algorithms and the goal. Declarations declare symbols and functions with their type. Basic types are boolean, integers, real ($\mathbb{R}$), complex ($\mathbb{C}$) vectors (`V`), matrices (`M`), indices and index sets. Vectors and matrices are defined over index sets either defined through the notation `V1`$[is]$, where is is the possibly multi-dimensional index set for vector `V1`, or deduced through type inference. A particular element of a vector is accessed by the use of an index: `V1[I1]`. Figure 2 shows the declaration of the constants used for Lattice QCD, and the definition of Dirac operator as a matrix. The two other matrices, P_e and P_o are projections, keeping only black or white elements of the lattice, like a 4D checkerboard.

Constant:

$$\begin{aligned} Dirac, P_e, P_o, \gamma_5 &\in M, \\ L, S, C, even &\in Indexset, \\ \gamma &\in Index->M, \\ U &\in Index->M, \\ \kappa, \mu, &\in \mathbb{R}, \\ D &\in Indexset \end{aligned}$$

Variable: $s \in Index, d \in Index$

$$\begin{aligned} Dirac &= I_{L\otimes C\otimes S} \\ &+ 2 * i * \kappa * \mu * I_{L\otimes C} \otimes \gamma_5 \\ &+ -\kappa * \sum_{d\in D} ((J_L^{-d} \otimes I_C) * \bigoplus_{s\in L} U[s \otimes d]) \otimes (I_S - \gamma[d]) \\ &+ -\kappa * \sum_{d\in D} ((J_L^{d} \otimes I_C) * \bigoplus_{s\in L} U[s \otimes -d]) \otimes (I_S + \gamma[d]) \\ P_e &= P_{even,L} \otimes I_{C\otimes S} \\ P_o &= P_{!even,L} \otimes I_{C\otimes S} \end{aligned}$$

Fig. 2. Definitions of the Dirac matrix on a Lattice L in QIRAL, and the two projections for even and odd elements (P_e and P_o respectively) of this lattice.

Equations are used to define variables or functions. Figure 3 describes nearly all properties and definitions on the constant and functions used for the simulation. For instance, the function "invertible" is defined for only some expressions.

Algorithms are given as possible definitions for statements

Constant:

$$dx, dy, dz, dt \in Index$$

$$\begin{aligned} D &= \{dx, dy, dz, dt\} \\ \text{isPeriodic}(L) &= true \\ U[s \otimes d]^\dagger &= U[(s+d) \otimes -d] \\ U[s \otimes -d]^\dagger &= U[(s+-d) \otimes d] \\ \text{Preconditioner1}(Dirac) &= Pe \\ \text{Preconditioner2}(Dirac) &= Po \\ \gamma[d]^\dagger &= \gamma[d] \\ \text{diagonal}(\gamma_5) &= true \\ \gamma_5 * \gamma_5 &= I_S \\ \gamma_5 * \gamma[d] &= -\gamma[d] * \gamma_5 \\ \text{invertible}(I_S + c * \gamma_5) &= true \\ \text{invertible}(I_S - c * \gamma_5) &= true \\ \text{invertible}(-(c * I_S) + i * \gamma_5) &= true \\ \gamma_5^\dagger &= \gamma_5 \\ \text{type}(\gamma[d]) &= S \times S \\ \text{type}(U[s \otimes d]) &= C \times C \\ \text{type}(\gamma_5) &= S \times S \\ vol(S) &= 4 \\ vol(C) &= 3 \end{aligned}$$

Fig. 3. Identities of constants used for Lattice QCD.

or expressions. For instance, the conjugate gradient algorithm in Figure 4 provides the code that compute expressions of the form $x = A^{-1} * b$, when A and b are given. It outputs the value of x, i.e. solve the linear system $Ax = b$.

Input : $A \in M, b \in V$
Output : $x \in V$
Constant: $\epsilon \in \mathbb{R}$
Match : $x = A^{-1} * b$
Var : $r, p, Ap, z \in V, \alpha, \beta, n_r, n_z, n_{z1} \in \mathbb{R}$
$r = b$;
$z = A^\dagger * r$;
$p = z$;
$x = 0$;
$n_z = (z \mid z)$;
$n_r = (r \mid r)$;
while $(n_r > \epsilon)$ **do**
  $Ap = A * p$;
  $\alpha = n_z/(Ap \mid Ap)$;
  $x = x + \alpha * p$;
  $r = r - \alpha * Ap$;
  $z = A^\dagger * r$;
  $n_{z1} = (z \mid z)$;
  $\beta = n_{z1}/(n_z)$;
  $p = z + \beta * p$;
  $n_z = n_{z1}$;
  $n_r = (r \mid r)$;

Fig. 4. Conjugate Gradient, normal residual method (CGNR).

The initial statement, in the **Match** clause, is then defined

(and replaced) by the pseudo-code. The **Var** keyword declares the type of local variables. This algorithm is written using the "algorithm2e" package in LaTeX, and is not specific to Lattice QCD. The user has the possibility to write new algorithms for Lattice QCD or any other algorithm found in common literature. The QIRAL compiler finds automatically how to compute for instance $A * p$ when A is instantiated with the Dirac operator.

Most often the validity of an algorithm depends on prerequisites, special properties the inputs must have. These prerequisites are declared in a clause **Require** and is proved by the QIRAL compiler. The following example illustrates this prerequisite mechanism. Figure 5 describes the Schur complement method that is used as a preconditioner for the conjugate gradient. The condition invertible$(P_e * A * P_e^t)$ is proved automatically by the compiler when A matches the matrix $Dirac$. To prove this, the property defined previously for the function "invertible" is used. If the compiler is not able to prove the requirements attached to an algorithm, the algorithm is not applied and an error is generated. Notice that

Input : $A, P_e, P_o \in M, b \in V$
Output : $x \in V$
Match : $x = A^{-1} * b$
Constant: $D_{11}, D_{12}, D_{21}, D_{22} \in M$
Var : $v_1, v_2, x_1, x_2 \in V$
Require : invertible$(P_e * A * P_e^t)$
$D_{21} = P_o * A * P_e^t$;
$D_{11} = P_e * A * P_e^t$;
$D_{22} = P_o * A * P_o^t$;
$D_{12} = P_e * A * P_o^t$;
$v_1 = P_e * b$;
$v_2 = P_o * b$;
$x_2 = (D_{22} - D_{21} * D_{11}^{-1} * D_{12})^{-1} * (v_2 - D_{21} * D_{11}^{-1} * v_1)$;
$v_1 = P_e * (2 * \kappa * b)$;
$x_1 = D_{11}^{-1} * (v_1 - D_{12} * x_2)$;
$x = P_e^t * x_1 + P_o^t * x_2$;

Fig. 5. Definition of Schur complement method.

on this preconditioning, the statements involve computation of inverse matrices. For the expression D_{11}^{-1}, the QIRAL compiler can prove automatically that D_{11} is diagonal (when A is the Dirac operator), and knows how to invert this matrix. For the computation of the expression $(D_{22} - D_{21} * D_{11}^{-1} * D_{12})^{-1}$, an iterative method has to be applied.

The goal defines the initial code and the list of algorithms to apply. The algorithms are composed from right to left.

Input : $bb \in V$
Output : $xx \in V$
Templates: CGNR schur
$xx[L \otimes C \otimes S] = Dirac^{-1} * bb[L \otimes C \otimes S]$;

For this goal here the preconditioner `schur` is applied on the initial statement, and then the CGNR algorithm. It is possible to chain multiple algorithms, used to apply preconditions before the solvers. The index set $L \otimes C \otimes S$ represents the Cartesian product of these sets and the domain for the vector bb. At this level, there is no implicit data layout for vectors and matrices. The vector bb could be either a 4D array of structures, one dimension for each dimension of L and the structure representing elements indexed by C and S, or a 1D array of structures, or just a large 1D array of complex values. This is orthogonal to the expression of the algorithm.

The output of QIRAL compiler is a function in C and OpenMP pragmas representing the computation described in the goal, and taking as parameters bb and xx. All other constant values (in particular constant matrices) are assumed to be global.

III. Implementation Details

The QIRAL compiler is based on a rewriting system, Maude [**?**]. The different steps of this transformation are explained in this section.

A. Algorithms composition and expression simplification

Algorithms are translated into rules of the rewriting system, while equations define the equational theory for the rewriting system. The first step consists in transforming LaTeXinput into a Maude program. Additional modules, defining usual algebraic simplifications are added to this code.

The first step parses the LaTeXinput and captures only what is described in predefined environments, for algorithms, definitions and the goal. Syntactic verification as well as type checking is performed. The output generated is a Maude module, with equations corresponding to definitions, rules corresponding to algorithms, and an unique Maude statement, corresponding to the goal.

The algorithms declared in the goal are applied, in turn, to the statements provided. These statements are terms for Maude. The **Match** clause is the left-hand side (lhs) of the rule, while the pseudo-code corresponds to a term that is the right-hand side (rhs) of the rule. Any statement matching the lhs will then be rewritten in the rhs. In order to identify this match, Maude unifies the input variables of the lhs with the real values, corresponding to the binding of the formal parameters of the actual function parameters of a function call. If a **Require** clause exists, it constitutes the condition for the rewriting. The first statement is provided by the goal, then algorithms are applied successively.

Definitions and properties are considered by Maude as defining the equational theory for the rewriting system. Actually, these equations are handled as automatic rewriting rules: Maude automatically applies all possible equations, rewriting their lhs into rhs, until the term is normalized. For instance, the property $x*(y+z) = x*y+x*z$, stating the distributivity of $+$ over $*$, will only be used to distribute the operators, not to factorize terms.

The main objective of this formal rewriting is to eliminate all terms that are equal to zero. In Lattice QCD, the Dirac matrix used in the problem is sparse, but built from dense matrices with a regular structure. To obtain such simplifications, an additional module defines properties for the linear algebra operators, on complex, vectors and matrices: Addition is

commutative and associative, multiplication is associative and distributes over the addition, binary subtraction is converted to unary minus, transposition distributes over the addition and multiplication, etc. Moreover, some properties are also defined for permutation and projection matrices, in particular to handle Schur preconditioning.

B. Generating Element-wise Computation

The term obtained after the first application and simplification step still manipulates matrices and vectors representing the whole lattice. Parallel loops are generated to iterate over the lattice component of the index sets used by all vectors. The different phases of the QIRAL compilation are driven by meta-rewriting rules. The user here can change the parameters for the compiler and choose for instance 4D iterators of the domain (building one loop for each dimension of L), 1D iterator (linearized space) or any combination, performing tiling for instance. For this, the user has to indicate in the properties that the lattice is decomposed into sub-lattices:

$$L = L_1 \otimes L_2$$

where L_1 and L_2 are two index sets. L_2 represents then sub-lattices and L_1 is the iteration domain, iterating through these different blocks.

The generation of loops for Lattice QCD is straightforward, as all array statements are parallel. All such loops are therefore parallel, possibly nested. Parallelism is expressed through OpenMP.

C. Open Issues

A number of difficulties may arise during this high level compilation step, since the range of properties and algorithms that can be described is not limited.

- Convergence: the properties the user define have to form a convergent rewriting system. For instance if the properties $x*(y+z) = x*y+x*z$ and $x*y+x*z = x*(y+z)$ are simultaneously present, the compiler is unable to know which one is to apply (no priority). Maude is not able to detect such situation. For algorithms, preconditions, there is no such issue since the user defines the sequence of algorithms to apply.
- Confluence: this only concerns properties and definitions. Confluence means that any order of application of the properties leads to the same resulting term. Some recent works [**?**], [**?**] have shown that this can be checked automatically. The impact is that depending on the order of the properties given by the user, there is a risk the code generated is not the same. This has not occurred in QIRAL so far.

These two limitations are therefore more theoretical than practical. More work though has to be done in order to provide a high quality development environment to users, in particular for the identification of the reasons why an algorithm cannot be applied to a particular code (the requirements are not fulfilled).

IV. Code Optimizations

Once parallel loops are generated, several code optimizations are applied. These transformations are also described in the rewriting system, with built-in modules of QIRAL, in a similar way to a compiler such as Stratego [**?**]. Current optimizations range from loop transformations, versioning and data layout. The output of this phase is a C-code with OpenMP.

A. Improving Locality

Loops fusion is a transformation to reduce reuse distances, hence improving locality. To check if fusion is valid, a simple dependence analysis, based on dependence distance, is computed. The fusion method is applied on consecutive independent loops that share the same iterators, and is applied on all code until no more fusion is possible. This simple strategy is sufficient for Lattice QCD generated codes.

Following this fusion, the regions of arrays that are written/read by all loops, and the regions that are inputs/outputs of loops are computed. All arrays that are used only in one loop are scalar promoted. The resulting values are allocated on the stack, and aligned for further vectorization. This reduces memory consumption.

Both transformations are applied automatically.

B. Versioning Matrix Multiplication

The computation involve many multiplications of vectors by constant matrices, accounting for transformations on spin and color (S and C index sets respectively). These matrices are small, of size 3×3 and 4×4 respectively, and the latest have only 2 non-null elements per line, these elements being among $\{1, -1, i, -i\}$. Therefore specialization of these product is necessary in order to obtained better performance. These matrix-vector multiplications appear in expressions of the form $(M_1 \otimes M_2) \cdot V$ with M_1 and M_2 the two matrices multiplied by a tensor product, and V is a 12 element vector. In this case the QIRAL compiler uses the identity

$$(M_1 \otimes M_2) * V = M_1 * V * M_2^t$$

where on the lhs, V is considered a matrix of size 3×4 and $*$ stands for the matrix product. Therefore, instead of using general matrix multiplication, QIRAL compiler finds these occurrences and calls versioned matrix multiplications. Specializing such multiplications for these particular sizes, in particular performing SIMDization, is essential for performance. These functions correspond to the hot-spot of the codes generated by QIRAL. The codes of these functions are hand-written in `libqiral` library as presented in Figure 1.

Other expressions can be replaced by library calls, and QIRAL changes expressions on vectors and matrices into BLAS calls (or specialized BLAS). The fact that the QIRAL compiler automatically identifies these functions in the code generated from the different algorithms facilitates the optimization process and is an asset of QIRAL. The optimization of these functions in `libqiral`, specializations of BLAS, can indeed be achieved by an expert in high-performance computing, independent of any Lattice QCD context.

C. Optimizing Data layout and Iterators

The choice of data layout is essential for performance. The dimensionality of the data structures (how many dimensions to the array) and the temporal locality loosely depend on the way the index domains are iterated. In QIRAL, loop iterators and number of loops are selected according to the transformation on data layout to achieve. Three different transformations are explored by the QIRAL compiler.

Tiling is a well established approach to reduce the overhead impact of memory accesses or inter-node communications. In our code generation framework, tiling can be specified by decomposing the lattice into sub-lattices and iterating in each sub-lattice in turn. As the Lattice QCD computation is an 8 point stencil (2 neighbors in each of the 4 dimensions), tiling leads to spacial reuse of each element read by the stencil, provided the tile fits into the cache. Note that as all loops are initially parallel, by tiling we create innermost sequential loops, and keep the outermost loops parallel. When using an accelerator, tiling is essential for effective and optimal data transfer [**?**]. An important point here is that for given a tile, we may need to pack the necessary data to proceed with. This is required for instance when we need to distribute the tiles among several processors or threads, or to offload tiled calculations into accelerator units. Another technical aspect, which sounds non-trivial, is the organization of the computation related to the border of the tiles. In most of the cases, data communications occur only between the borders. Consequently, the computation of a tile has to be split into two parts, one for the bulk and another for the border. We should be able to generate code accordingly in future work.

Even-odd preconditioning (or Schur complement method) leads to computation on only one element out of 2 in the lattice. These elements are accessed as the white squares of a 4D checkerboard. The QIRAL compiler analyzes the set of elements accessed for each array, and arrays are allocated only for the elements accessed. The loop domains are also redefined accordingly, thus no conditionals are created. Half-lattice arrays are therefore created when even-odd preconditioning is applied. This avoids useless memory strides, improving spacial locality, and reduces the in-memory working-set.

Finally, as the lattice is a 4D lattice, 2 options are possible: either vectors are indexed with a 4D index (meaning there are four loops), or their indices are linearized. As the 4D space is a torus, the computation of the index for all the neighbors requires a modulo arithmetic operation. This is simpler with a 4D index than with a 1D linearized index. Both versions can be generated, and combined with tiling or even-odd transformations.

Figure 6 shows an example of the code automatically generated by the QIRAL compiler. Note that all statements are calls to functions. The QIRAL compiler has matched expressions with these functions, all corresponding to BLAS functions or specialized BLAS. These functions are implemented in the `libqiral` library.

D. Hand-tuned Optimizations

As presented in the previous section, the QIRAL compiler generates code for Lattice QCD that calls dedicated versions of BLAS functions. This versioning is due to constant and simple matrices, special sizes or coefficients. Different versions for these functions are implemented: one corresponds to a naïve C implementation, letting the compiler perform optimizations. This has the advantage of simplifying the port to new architectures. Other versions are obtained by SIMDization of these functions, either through the Intel SPMD Program Compiler (ISPC) [**?**], or through manually-written code using compiler intrinsics.

The compiler ISPC requires adding a few extra keywords to standard C in order to help SIMDization. Back-ends for ISPC are Intel architectures with SSE, AVX, AVX-2 and in future versions, Xeon Phi SIMD. For complex value manipulation, ISPC tries to generate SIMD code with real and imaginary parts in different vectors. For complex values allocated contiguously, this requires the use of strided loads and stores (named GATHER and SCATTER in Intel ISA), not available on Sandybridge for instance.

Alternatively, complex values can be loaded into the same vectors, storing imaginary and real parts contiguously. Depending on the instruction set, complex multiplication is then implemented either using a shuffle operation, or directly dedicated SIMD instructions. Use of shuffle operations or of these instructions requires on the architectures considered the use of intrinsics.

Besides SIMDization, for shared-memory non-uniform (NUMA) architectures, memory allocation has to take into consideration the precise cores that are using this memory. The use of NUMA policy library is a way to distribute memory among the cores, according to their predefined affinity. This hand-optimization is conducted on all structures allocated before the execution of the function generated by QIRAL.

V. Related Works

Since it is a heavily CPU time consuming type of application, LQCD was the focus of many developments and papers in the last 15 years. To name a few, some recent works target mostly Intel processors like Chroma [**?**] on Xeon Phi [**?**], for a Wilson-Dirac Lagrangian type, or mostly the BlueGene family like tmLQCD [**?**] on the BlueGene/Q supercomputer [**?**] with the twisted-mass Lagrangian type. A whole bunch of works also appeared in the last years targeting the GPUs, [**?**], [**?**], and the latter are definitely part of our own road map: QIRAL will have a great impact on the GPU coding due to the flexibility it provides, specifically for data-layout.

The Spiral language [**?**] was obviously the precursor of the QIRAL tool and the model for this work on a LQCD-dedicated DSL, specifically designed for sparse matrices with very small conditioning factor. Qiral initial work was described in [**?**]. It is a multi-disciplinary project [**?**] merging fruitfully Lattice QCD theory labs like LPT/CNRS, High Energy Physics people like LAL/IN2P3/CNRS and HPC Parallel Computing experts coming from several INRIA teams (Orsay, Bordeaux) and

```
#pragma omp parallel for
for(iL = 0 ; iL < L / 2 ; iL += 1)
  {
    double complex tmp[12] __attribute__((aligned(0x1000))) ;
    double complex IDschur9[12] __attribute__((aligned(0x1000))) ;
    double complex IDschur10[12] __attribute__((aligned(0x1000))) ;
    xgemmGAMMAp1dx(& IDschur5[12 * layouteven(sup(idxodd(iL), dx, L))],tmp);
    xgemmfast0(U[uup(idxodd(iL), dx, L)],tmp,0,IDschur10);
    xgemmGAMMAp1dy(& IDschur5[12 * layouteven(sup(idxodd(iL), dy, L))],tmp);
    xgemmfast(U[uup(idxodd(iL), dy, L)],tmp,1,IDschur10);
    xgemmGAMMAp1dz(& IDschur5[12 * layouteven(sup(idxodd(iL), dz, L))],tmp);
    xgemmfast(U[uup(idxodd(iL), dz, L)],tmp,1,IDschur10);
    xgemmGAMMAp1dt(& IDschur5[12 * layouteven(sup(idxodd(iL), dt, L))],tmp);
    xgemmfast(U[uup(idxodd(iL), dt, L)],tmp,1,IDschur10);
    xcopy(12, IDschur9, IDschur10) ;
    xscal(12, kappa, IDschur9) ;
     ...
  }
```

Fig. 6. Excerpt of the code generated by the QIRAL compiler for the CGNR algorithm with the Schur preconditioning.

CAPS-Entreprise. Pochoir [**?**] is also a similar tool targeting at stencils.

But while Spiral model was inspiring to QIRAL, the former is only targeting at Signal Line Processing, where the latter is fully managing the algorithm, together with the data layout issues, and this part is crucial to improve computing speed. This new tool will allow theory physicists to propose, validate and evaluate the efficiency of their new algorithms more easily and quickly. It will speedup as well the inversion coding for new types of Lagrangian [**?**] (Clover [**?**], Overlap [**?**], and so on).

To summarize, even if the current version is this running on a single node with threads (no MPI), the very huge effort achieved for the data-layout will allow for an easier mixed implementation (MPI-like, OpenMP), and the performances in terms of CPU speed for the global algorithm is very attractive, even for rather large QCD lattices (up to 2^{25}-node lattice already tested). This is very important, since there was always a huge gap between small and large lattices when comparing their behavior, because this application is both memory-size and memory-bandwidth bound.

VI. Performance Results

The following architectures are used for the experiments:

Name	Processor	# of cores (hyperthreading)	Clock in Ghz
Nehalem-EX	Intel X7560	4×32 (2)	2.26
SandyBridge1	E5-2680	16 (2)	2.7
SandyBridge2	E5-2687W	16 (2)	3.1
Haswell	i5-4670T	4 (1)	2.3
Xeon Phi	SE10P	60 (4)	1.1

The Nehalem-EX and the Sandybridge1 correspond to Curie machines from the TGCC Supercomputer center, Curie Fat and Curie Thin respectively[1]. The same compiler, ICC, is used for the compilation of tmLQCD and QIRAL generated codes.

[1]http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm

Several iterative methods are written with QIRAL. Figure 7 presents some of these methods, for two architectures: CGNR, CRNE, MCR1 and MCR2 with some preconditioners: Schur and preMCR. We observe that while MCR2 exhibits the best time per iteration, the method takes more time to converge than CGNR and Schur. This shows that the best method cannot be determined only by benchmarking a single iteration, but it is necessary to run all iterations. This justifies the need for an automatic approach to the generation of parallel codes, able to run for large data-sets, in order to compare different methods. Besides, the second plot of Figure 7 shows that the relative difference may vary according to the architecture. While the absolute best method is still the same (here CGNR combined with Schur), this stresses the fact that the algorithmic solution may be chosen depending on the target architecture.

In order to compare tmLQCD with the code generated by QIRAL, the same algorithm is used for both (CGNR and Schur preconditioning). Performance is displayed for all architectures as the total execution time multiplied by the number of cores. Due to the fact that tmLQCD is using MPI, there is no version for Xeon Phi. Besides, the tmLQCD code uses in-line assembly code with SSE3 instructions. Adapting this code for newer SIMD extensions is more difficult than adapting intrinsics as used by QIRAL. Indeed for intrinsics, part of the optimization work still relies on the compiler: register allocation, generation of FMAs, scheduling. The code generated by QIRAL has been quickly ported to these architectures, and then code tuning has focused on the library used by QIRAL (with versioned BLAS), using intrinsics and aggressive in-lining.

Figure 8 presents timing results on different architectures, comparing tmLQCD code with QIRAL generated code. For QIRAL, the "hand-optimized library" corresponds to the best version obtained, using intrinsics (AVX, AVX2, Xeon Phi) for Sandybridge, Haswell and Xeon Phi architectures. The Nehalem EX version does not use SSE SIMD intrinsics. This explains why QIRAL/Nehalem EX version is more than two times slower than tmLQCD. For Xeon Phi, the performance

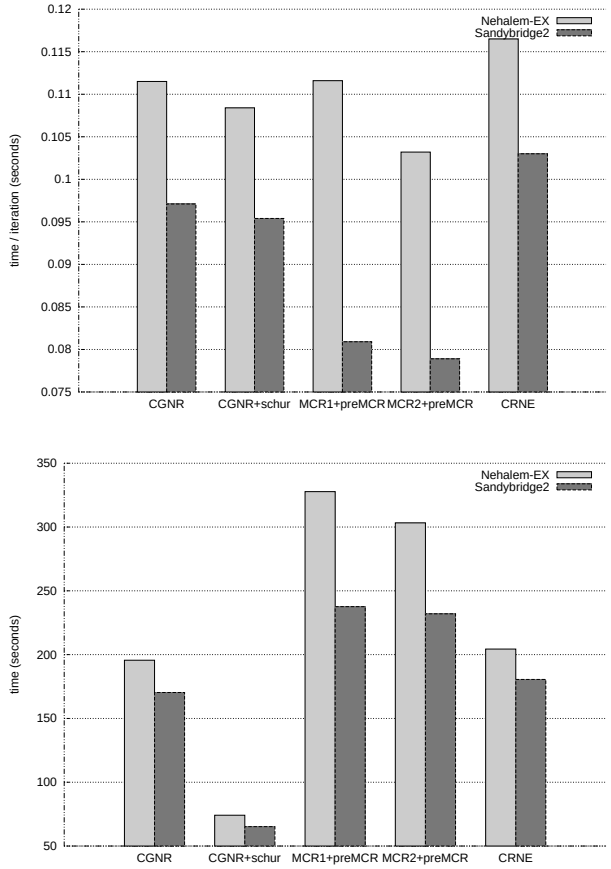

Fig. 7. Comparison between different iterative methods, on Nehalem-EX and Sandybridge 2 architectures. Top figure: Time in second per iteration. Bottom figure: Total execution time.

displayed corresponds to the use of all the 60 cores, and a linear speed-up can be observed by using an increasing number of cores. The ISPC compiler has been used to generate SIMD version of matrix multiplication of size 3×4 on complexes. The compiler is still in development, does not fully work for Xeon Phi. Figure 8 shows that the level of performance reached with ISPC is not competing with the level for hand-tuned intrinsics.

The strong scalability of the code generated by QIRAL is evaluated on Xeon Phi and Nehalem-EX architectures. Figure 9 shows efficiency results for different number of cores. Note that the size of the lattice is different for both architectures, reflecting the need for different granularity. The efficiency for the Xeon Phi is compared to the run on 4 cores, with 4 threads each. This explains why for some number of threads, the efficiency goes beyond 1. The code scales well upto the 60 cores (240 threads). For the Nehalem-EX machine, the efficiency is higher than 95% up to 32 cores, and then drops quickly. The reason is that a 128-core node is structured with 4 groups of 4 octo-cores, connected through a switch. Going

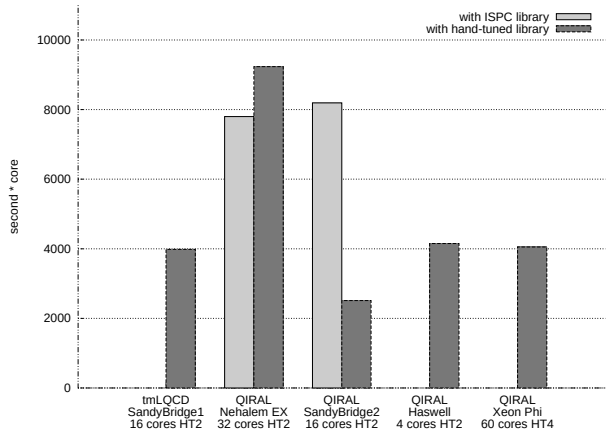

Fig. 8. Normalized performance for the inversion on different architectures, with QIRAL and tmLQCD codes. Performance is shown in sec.core, lower is better. The execution time is obtained by dividing this performance by the number of cores. The same method, a conjugate gradient with Schur preconditioning is used in all cases, with a lattice of size $24^3 * 48$ and an error of 10^{-14}.

through the switch has a high penalty in terms of performance.

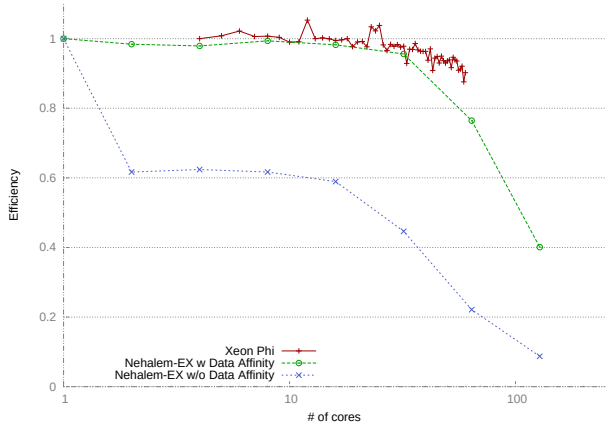

Fig. 9. Efficiency of the code generated by QIRAL on Xeon Phi and Nehalem EX, according to the number of cores used. For both architectures, the method used is the conjugate gradient with Schur preconditioning. The lattice size for the Xeon Phi is $24^3 * 48$ and for the Nehalem-EX, $64^3 * 128$. On the Nehalem-EX the efficiency is measured with and without NUMA-aware memory allocation.

VII. Conclusion

The contribution of this paper is a new domain-specific language, QIRAL, for the automatic code generation of OpenMP codes for Lattice QCD simulations. QIRAL language offers to physicists the possibility to implement iterative methods and preconditioners, literally "from the book" using LaTeX,

or design new ones, and test them on large parallel shared memory machines or on accelerators such as the Xeon Phi. The language enables the composition of preconditioners and iterative methods, and the compiler checks automatically the validity of application for each method. This makes possible a more systematic exploration of the algorithmic space: indeed, it removes from the physicists the burden of long and stressful validations of their new code since it will be automatically generated, then safer, and the time-to-market for a viable product will be much shorter. The QIRAL compiler generates OpenMP parallel code using BLAS or specialized versions of BLAS functions. Further hand-tuning is possible on the code generated by QIRAL, and we have shown that the performance on various multi-core architectures and on Xeon Phi accelerator it compares or outperforms the performance of a hand-tuned Lattice QCD application, tmLQCD.

Among the perspectives of this work, the automatic generation of a communication code for multi-node computation would enable to run Lattice QCD simulations on a larger scale. Besides, the fine tuning of the library functions used by QIRAL on different architectures, in particular their SIMDization, could be improved.

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399