AN EFFICIENT CELL LIBRARY FOR LATTICE QUANTUM CHROMODYNAMICS

Claude Tadonki
Linear Accelerator Laboratory/CNRS/IN2P3
University of Orsay, Faculty of Sciences, Bat. 200
91898 Orsay Cedex (France)
claude.tadonki@u-psud.fr

Gilbert Grosdidier
Linear Accelerator Laboratory/CNRS/IN2P3
University of Orsay, Faculty of Sciences, Bat. 200
91898 Orsay Cedex (France)
Gilbert.Grosdidier@cern.ch

Olivier Pene
Theoretical Physics Laboratory
University of Orsay, Faculty of Sciences, Bat. 210
91405 Orsay Cedex (France)
olivier.pene@th.u-psud.fr

ABSTRACT
Quantum chromodynamics (QCD) is the theory of subnuclear physics, aiming at modeling the strong nuclear force, which is responsible for the interactions of nuclear particles. Numerical QCD studies are performed through a discrete formalism called LQCD (Lattice Quantum Chromodynamics). Typical simulations involve very large volume of data and numerically sensitive entities, thus the crucial need of high performance computing systems. We propose a set of CELL-accelerated routines for basic LQCD calculations. Our framework is provided as a unified library and is particularly optimized for an iterative use. Each routine is parallelized among the SPUs, and each SPU achieves its task by looping on small chunks of arrays from the main memory. Our SPU implementation is vectorized with double precision data, and the cooperation with the PPU shows a good overlap between data transfers and computations. Moreover, we permanently keep the SPU context and use mailboxes to synchronize between consecutive calls. We validate our library by using it to derive a CELL version of an existing LQCD package (tmLQCD). Experimental results on individual routines show a significant speedup compared to standard processor, 11 times better than a 2.83 GHz INTEL processor for instance (without SSE). This ratio is around 9 (with QS22 blade) when considering a more cooperative context like solving a linear system of equations (usually referred as Wilson-Dirac inversion). Our results clearly demonstrate that the CELL is a very promising way for high-scale LQCD simulations.

Keywords
LQCD, linear algebra, parallelism, CELL

1. INTRODUCTION
Quantum chromodynamics (QCD) [16], the theory of the strong nuclear force, can be numerically simulated on massively parallel supercomputers using the method of lattice gauge theory (LQCD), see Vranas et al [15]. A LQCD simulation chain involves basic linear algebra computations on large scale entries. Moreover, basic operations are repeated so many times following a stopping criteria which is either purely numerical or based on the physical meaning of the result. An example of important kernel is the inversion of the so-called Dirac operator, which is an important step during the synthesis of a statistical gauge configuration sample. Indeed, in the Hybrid Monte Carlo (HMC) algorithm [9], it appears in the expression of the so-called fermionic force, used to update the momenta associated with the gauge fields along a trajectory.

A common way to parallelized LQCD applications is to partition the lattice into sublattices and then assign each sublattice to a computing node (see [4, 10]). This yield a standard SPMD model which is then mapped onto a given parallel machine. Thus, tuning an individual computing node to efficiently perform a critical part of the simulation is a good way towards a powerful cluster. Number of authors have studied the LQCD implementation on various supercomputers [15]. For the special case of solving the Wilson-Dirac system, a mixed-precision solution accelerated with GPUs is proposed by Clark[4]. A domain decomposition approach associated with the deflation technique is studied by Luscher[10]. A prospective overview of QCD implementation on the CELL is reported in [1]. A specific study of the Dirac operator (the most CPU consuming kernel) on the CELL (simulator) is reported in [5].

LQCD computation kernels are built up from basic linear algebra routines, with the arguments being organized following specific data structures. Hence, comes the idea of a computing library dedicated to LQCD. A well known example of such a library is the so-called QDP++[13], which provides a data-parallel programming environment suitable for Lattice QCD. In this work, we propose a CELL-accelerated library for the same purpose. Our main concern is performance. Indeed, having a more powerful node yields the additional advantage of a smaller computing network request, which significantly reduce inter-nodes communication overhead. This aspect is crucial in LQCD performance, otherwise people consider very large supercomputers and suffer from communication penalty.

The rest of the paper is organized as follows. The next section presents an overview of the CELL Broadband Engine, followed in section 3 by LQCD linear algebra foundations and reference data structures. In section 4, we provide a generic acceleration scheme considered for each selected
routines. Next, we explain in section 5 how the whole library is organized. In section 6, we describe how to use the library. Section 7 is devoted to benchmark reports, and section 8 concludes the paper.

2. THE CELL BROADBAND ENGINE

The CELL [3, 12] is a multi-core chip that includes nine processing elements. One core, the POWER Processing Element (PPE), is a 64-bit Power Architecture. The remaining eight cores, the Synergistic Processing Elements (SPEs), are Single Instruction Multiple Data (SIMD) engines (3.2GHz) with 128-bit vector registers and 256 KB of local memory, referred to as local store (LS). Figure 1 provides a synthetic view of the CELL architecture [2].

![Figure 1: Overview diagram of the CELL](image)

Programming the CELL is mainly a mixture of single instruction multiple data parallelism, instruction level parallelism and thread-level parallelism. The chip was primarily intended for digital image/video processing, but was immediately considered for general purpose scientific programming (see [17] for an exhaustive report on the potential of the CELL BE for several key scientific computing kernels). A specific consideration for QR factorization is presented in [7]. Nevertheless, exploiting the capabilities of the CELL in a standard programming context is really challenging. (see [14])

3. BACKGROUND AND PRELIMINARIES

3.1 Foundations

**Definition** Given a complex square matrix $A$ and an integer $\mu$, we state the followings definitions:

- $|A|$ the order of the matrix $A$
- $I_A$ the identity matrix of order $|A|$  
- $A^T$  
- $\hat{\mu} = e_\mu$ ($\mu^{th}$ vector of the canonical basis)

**Definition** Given two matrices $A$ and $B$, the tensor product $C = A \otimes B$ is defined by

$$C = (a_{ij}B)$$

(1)

**Property** Given three complex matrices $A$, $B$ and $C$, we have (see [14])

**Associativity:**

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C$$

(2)

**Normal factors decomposition:**

$$A \otimes B = (A \otimes I_B)(I_A \otimes B) = (I_A \otimes B)(A \otimes I_B)$$

(3)

Tensor product is rarely computed explicitly as it involves a huge amount of memory and lot of redundancies. Instead, an implicit approach is considered depending on the final targeted operation.

We now consider five $4 \times 4$ special matrices, called Dirac $\gamma$-matrices, which are given by

$$\gamma_0 = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix}$$

$$\gamma_1 = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix}$$

(4)

$$\gamma_2 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_3 = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & 0 & i \\ 0 & i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix}$$

(5)

$$\gamma_5 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

(6)

The basic Wilson-Dirac operator can be schematised by the following formula

$$D\psi(x) = A\psi(x) - \frac{1}{2} \sum_{\mu=0}^{4} \left[ (I_4 - \gamma_\mu) \otimes U_{x,\mu}\psi(x + \hat{\mu}) + (I_4 + \gamma_\mu) \otimes U_{x,-\mu}\psi(x - \hat{\mu}) \right]$$

(7)

where

- $A$ is a $12 \times 12$ complex matrix of the form $\alpha I_{12} + \beta (\nu \otimes \tilde{\gamma})$, where $\alpha, \beta$ are complex coefficients and $\nu$ a $3 \times 3$ complex matrix
- $x$ is a given point of the lattice, which is a finite subset of $\mathbb{N}^4$
- $\psi$ (called quark field or Wilson vector) is a 12-components complex vectors in the lattice
- $U_{x,\mu}$ is a $3 \times 3$ complex matrix (called gluon field matrix or gauge matrix) at $(x, \mu)$.

Operation 7 is the most time consuming kernel as it involves a significant amount of floating point operations and its is done so many times. However, other linear algebra operations are also time consuming because they are applied on very large arrays of complex numbers. One advantage when
using the CELL on such memory intensive processing is the possibility of overlapping data transfers and computations. We now state on data structures commonly considered in LQCD, which also correspond to what we have considered.

3.2 Data structures

The data structures used in LQCD are based on the following canonical types.

typedef struct
  { double re,im; }
typedef complex;

typedef struct
  { complex c00,c01,c02,c10,c11,c12,c20,c21,c22; }
su3;

typedef struct
  { complex c0,c1,c2; }
su3_vector;

typedef struct
  { su3_vector s0,s1,s2,s3; }
spinor;

typedef struct
  { su3_vector s0, s1; }
halfspinor;

Note that equivalent data structures can be also used as we manipulated the arrays through pointers on double. This yields the equivalences of figure 2.

<table>
<thead>
<tr>
<th>Original type</th>
<th>Equivalent type</th>
<th>size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex c;</td>
<td>double c[2];</td>
<td>16</td>
</tr>
<tr>
<td>su3 u;</td>
<td>double u[18];</td>
<td>144</td>
</tr>
<tr>
<td>su3_vector v;</td>
<td>double v[6];</td>
<td>48</td>
</tr>
<tr>
<td>spinor s;</td>
<td>double s[24];</td>
<td>192</td>
</tr>
<tr>
<td>halfspinor h;</td>
<td>double h[12];</td>
<td>96</td>
</tr>
</tbody>
</table>

Figure 2: Data structures equivalence

The inputs/outputs of the routines implemented in our library are either double, complex, or spinor. At execution, the DMAs operate on arrays of spinor. Thus, since a spinor has size 192 = 16 × 12, aligning the entry pointer of the whole array of spinors is sufficient to have the DMA of any portion being aligned. In the case of Wilson-Dirac operator, we also transfer su3 data (U matrices) and unsigned int data representing the indices of the right hand side spinors (see equation (7)). From ((7)), we see that 8 U matrices and 8 indices are needed for each spinor component. Thus, we transfer 8K su3 and 8K unsigned int, which are both suitable sizes for DMA (multiple of 16 bytes). Let now see how each of our CELL-accelerated routines are built.

4. GENERIC ACCELERATION SCHEME

For a given routine to be accelerated, we proceed as follows:

1.) We derive a SIMD version of the code using SPE intrinsics. In double precision, we manipulate vector registers with two double values. An example is provided in Algorithm 1 for subtracting two su3_vector (array of 3 complex numbers).

Algorithm 1 SIMD subtracting of two su3 vector
1:  
2:  
3:  
4:  
5:  
6:  
7:  

2.) On the SPE, the main routine performs the global loop following the typical sequence

DMA_get + SIMD Computation + DMA_put

This sequence is executed on small portions of the input arrays (the maximum that fits into the local store) until the whole arrays (on main memory) are visited.

3.) We apply a double buffering technique as described in Algorithm 2 in order to overlap data transfers and computations.

Algorithm 2 Generic double buffering scheme
1:  
2:  
3:  
4:  
5:  
6:  
7:  
8:  
9:  
10:  
11:  
12:  

The generic DMA mechanism of Algorithm 2 takes into account two-ways transfers. We need to make sure that a partial result as been sent back to the main memory before we reuse the same space for the next computation. That’s why we split the SPE computation into two parts. The first part should overlap with the DMA_put of the previous result, while the second one overlap with the DMA_get for the next input. For reduction operations like the scalar product or the square norm, this additional strategy is not needed because we don’t send any data during the whole computation (we just send the final result at the end). However, if we chose to keep the global mechanism for reduction operations, then we will be able to do an assignement operation at the same time. Thus, we provide the routine CELL_square_norm_assign(spinor *S, spinor *R, int N) which calculates the square norm of the spinor arrays R while copying it into S. We then eliminate the cost of the memory copy by hiding it into the square norm calculation. This local optimization is not negligible as memory accesses is a critical part of LQCD computations.

From the PPU side, the task is partitioned and distributed onto the different SPEs. We use a generic master data structure for all our routines. The corresponding variables, one
for each SPE, contain scalar values (if any) and pointers to the input/output arrays. The SPE first initiates a DMA to get its own master structure, and then uses it to perform it task following the scheme of Algorithm 2. The master structure remains the same (we only change its values) during different calls. By this way, all participating SPEs remain active during a complete session with our library. We now provide more details about this aspect.

5. GLOBAL ORGANIZATION

First, the SPE code aggregates all the routines of our library and the main routine (Figure 3) just selects which one to call using an indicator sent by the PPE using the mailbox mechanism.

```c
int main(int argc){
    unsigned int which_code;
    /* SPE asks what to do */
    which_code = spu_read_in_mbox();
    while(which_code!=-1){
        switch(which_code){
            case 0: routine_0( argp); break;
            case 1: routine_1( argp); break;
            ... 
            case n: routine_n( argp); break;
            default: break;
        }
        /* SPE tells he has finished */
        spu_write_out_mbox(1);
        /* SPE waits for next run */
        which_code = spu_read_in_mbox();
    }
    return 0;
}
```

Figure 3: SPE main program

At the initialization of the library, the PPU creates and loads the context on each participating SPE. During a session with our library, when the user requests the execution of PPUroutine_id, the PPU prepares the data and mails id to the SPEs. Each SPE receives id, executes PPUroutine_id, and waits for the next request. Figure 4 displays the list of routines yet implemented in our library (see [6] for their original specifications).

```c
void CELL_QCD_INIT();
void CELL_QCD_FINALIZE();
complex CELL_scalar_prod();
double CELL_scalar_prod_r();
double CELL_square_norm();
double CELL_square_norm_assign();
void CELL_assign_diff_mul();
void CELL_assign();
void CELL_assign_diff_mul_add_r();
void CELL_assign_diff_mul_serie();
void CELL_Hopping_Matrix();
void CELL_eo_tm_inv_psi();
void CELL_diff();
void CELL_mul_one_pm_imu_sub_mul_gamma5();
void CELL_mul_one_pm_imu_inv();
build_dependence_indices();
```

6. HOW TO USE THE LIBRARY

The library is intended to be integrated into C (or C++) codes. Only one change is required in the code (if not yet done). This concerns memory allocations, 16 bytes alignment is mandatory. Our suggestion is to use `#define` statements to change every `malloc` into `malloc_align` and `free` into `free_align`. This is done in the file `cell_lqcd.h`, which also contains the declaration of all routines. The steps necessary to use the library are the followings:

- include the file `cell_lqcd.h` into the code
- compile the code and link it with the library `ppe_lqcd.o` `spu_lqcd.a` `-lspe2 -lmisc`

Typical use of the library follows the sequence below:

- `CELL_QCD_INIT();` (called once at the beginning)
- different calls to the routines
- `CELL_QCD_FINALIZE();` (called once at the end)

If `CELL_Hopping_Matrix()` needs to be used, then a call to `build_dependence_indices()` is necessary to build up the dependence indices following formula (7). The result of this procedure is an array of indices that will be passed as an argument to `CELL_Hopping_Matrix()`. We chose to separate it from `CELL_Hopping_Matrix()` because it is done once, based on geometry parameters (so we avoid useless redundant calculation). We now report some illustrative benchmark results.

7. PERFORMANCE RESULTS

We present some performance results using our library in a $32 \times 16^3$ lattice (hence 131 072 sites). We use 8 SPUs simultaneously.

```
<table>
<thead>
<tr>
<th>Routine</th>
<th>Intel 2.83GHz</th>
<th>CELL BE QS20</th>
<th>CELL BE QS22</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell_diff</td>
<td>0.0110</td>
<td>0.00177</td>
<td>0.00183</td>
</tr>
<tr>
<td>cell_scalar_prod</td>
<td>0.0060</td>
<td>0.00148</td>
<td>0.00110</td>
</tr>
<tr>
<td>cell_assign_diff_mul</td>
<td>0.0080</td>
<td>0.00176</td>
<td>0.00175</td>
</tr>
<tr>
<td>cell_Hopping_Matrix</td>
<td>0.1210</td>
<td>0.01755</td>
<td>0.00650</td>
</tr>
</tbody>
</table>
```

Figure 4: Timings (seconds) of individual routines

We see from figure 4 that our implementation is globally competitive. The difference between QS20 and QS22 is perceptible only with the `Hopping_Matrix()` because of its significant computing load. This is in our favor, because `Hopping_matrix()` implements the Wilson-Dirac operator, which consumes more than 80% of the computation time when dealing with linear system solving. Indeed, figure 5 displays the elapsed times for solving a Wilson-Dirac system using iterative methods (GCR: Generalized Conjugate Residual and CG: Conjugate Gradient) on our $32 \times 16^3$ lattice.

```
<table>
<thead>
<tr>
<th>Routine</th>
<th>Intel 2.83GHz</th>
<th>CELL BE QS20</th>
<th>CELL BE QS22</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCR (67 iterations)</td>
<td>27 s</td>
<td>5.58 s</td>
<td>3.68 s</td>
</tr>
<tr>
<td>CG (685 iterations)</td>
<td>362 s</td>
<td>42 s</td>
<td>20 s</td>
</tr>
</tbody>
</table>
```

Figure 5: Timings of the Wilson-Dirac inversion
We first compare with an Intel 2.83 Ghz (without SSE and using one core) just to be close to the 3.2 GHz frequency of an SPE. We globally see a promising speedup (around 9). The case of CG is more impressive because the algorithm mainly relies on the Wilson-Dirac operator, which is the best accelerated routine of our library. Now, with an improvement in the DMA organization, we got the following best accelerated routine of our library. Now, with an improvement in the DMA organization, we got the following best accelerated routine of our library.

Acknowledgments. We wish to express our gratitude to Urbach Carsten, one of the main author of the tmLQCD implementation. We wish to express our gratitude to Claude Tadonki and Bernard Philippe, Implementing Wilson-Dirac operator on the cell broadband engine, ICS ’08: Proceedings of the 22nd annual international conference on Supercomputing, pp. 4-14, Island of Kos, Greece, 2008.

Figure 6: Dirac operator on the CELL

Figure 7: Dirac operator Intel i7 quadcore 2.83 Ghz

We see from figure 6 that our implementation of the Dirac operator scales very well on a QS20 and suffers from a slowdown on a QS22. We think, at this stage of our work, that this is due to an inappropriate SPU allocation, since we use a Dual Cell Based Blade. This should be easy to fix and then provide a scalable implementation on a QS22 too. Moreover, even with our optimal DMA organization, this part is still highly dominant on a QS22. The main idea we have in mind to overcome this is the use of the SU(3) reconstruct mechanism. This will significantly reduce the volume of exchanged data and increase the SPE computation load, thus a more balanced implementation.

8. CONCLUSION

Lattice QCD simulations at higher resolution (bigger lattices) is a challenging computing task. Meaningful physical information are expected from very large-scale simulations. Thus, an efficient system for LQCD computation is highly expected. Supercomputers are indeed considered for this task, but as usual, the more nodes we have, the more we exacerbate the inter-processor communication overhead. This acts in favor of a more powerful computing node. Accelerated nodes clearly appear as good candidates, provided the existence of efficient libraries. Our work is a contribution towards this requested achievement. Our analysis and methodology reveal number of key points and technical strategies that can be generalized for any accelerators based implementation.

Acknowledgments. We wish to express our gratitude to Urbach Carsten, one of the main author of the tmLQCD package for providing the code and exchanging with us. We also thank our colleagues of the PetaQCD project for our regular and fruitful discussions. We are grateful to Lionel Lacassagne (IEF through a past ANR project OCHELLE) and the CC-IN2P3 at Lyon for the access to their CELL machines.

9. REFERENCES