Efficient SIMDization and data management of the Lattice QCD computation on the Cell Broadband Engine¹

Khaled Z. Ibrahim* and François Bodin
IRISA/INRIA, Campus de Beaulieu, Rennes, France

Abstract. Lattice Quantum Chromodynamic (QCD) models subatomic interactions based on a four-dimensional discretized space–time continuum. The Lattice QCD computation is one of the grand challenges in physics especially when modeling a lattice with small spacing.

In this work, we study the implementation of the main kernel routine of Lattice QCD that dominates the execution time on the Cell Broadband Engine. We tackle the problem of efficient SIMD execution and the problem of limited bandwidth for data transfers with the off-chip memory. For efficient SIMD execution, we present runtime data fusion technique that groups data processed similarly at runtime. We also introduce analysis needed to reduce the pressure on the scarce memory bandwidth that limits the performance of this computation.

We studied two implementations for the main kernel routine that exhibit different patterns of accessing the memory and thus allowing different sets of optimizations. We show the attributes that make one implementation more favorable in terms of performance.

For lattice size that is significantly larger than the local store, our implementation achieves 31.2 GFlops for single precision computations and 16.6 GFlops for double precision computations on the PowerXCell 8i, an order of magnitude better than the performance achieved on most general-purpose processors.

Keywords: Lattice QCD calculations, Cell Broadband Engine, SIMD computer architecture

1. Introduction

Simulating Lattice Quantum Chromodynamic (QCD) aims at understanding the strong interactions that bind sub-nuclear matter (quarks and gluons) together to form stable nuclear matter (hadrons) [19]. In Lattice QCD, a four-dimensional space–time continuum is simulated, where the quantum field (composed of quarks) resides at the lattice sites and the gauge field (composed of gluons) resides at the links between these sites. Lattice spacing needs to be small to obtain reliable results which requires enormous amount of computations because of the increase of the number of lattice sites. Figure 1 shows the discretization of the

four-dimensional space–time continuum of the Lattice QCD. Efficient implementation of a main kernel routine, responsible for computing the actions of Wilson–Dirac operator, is of critical importance for the simulation of Lattice Quantum Chromodynamics (Lattice QCD) [4,6,19].

Efficient use of accelerators for scientific computing is usually addressed by researchers. Among recently attractive technologies are graphic processing units (GPUs) [2,3,9,12,15] and the Cell Broadband Engine (BE) [1,4,10,20]. The Lattice QCD community, as well as other high performance computing communities, started exploring the possibility of using these accelerators to build cost effective supercomputers for scientific computations.

Using GPUs, for instance, has been investigated [7,8] for Lattice QCD, especially with the advent of general-purpose programming environment such as Cuda [11] for NVidia graphic cards. The main challenge in these environments is the over-protection that

¹This work is based on an earlier work: Implementing Wilson–Dirac operator on the Cell Broadband Engine, presented in International Conference on Supercomputing, © ACM, 2008; <http://doi.acm.org/10.1145/1375527.1375532>.

*Corresponding author: Khaled Z. Ibrahim, IRISA/INRIA, Campus de Beaulieu, Rennes 35042, France. Tel.: +33 2 9984 7110; Fax: +33 2 9984 7171; E-mail: kibrahim@irisa.fr.

Fig. 1. Four dimensional space-time Lattice QCD with odd and even sub-fields marked by different colors.

most manufacturers adopt to hide their proprietary hardware designs.

There is a great interest of the computational science community in the Cell BE family of processors, including those interested in the simulation of the Lattice QCD. For Lattice QCD analytical model is developed [4] to predict the performance on the Cell BE of the main kernel routine, as well as multiple implementations to determine the potential performance that can be achieved based on the Cell family of processors are currently announced [10,14]. The first supercomputing facilities breaking the petaflop barrier, the Roadrunner of Los Alamos National Laboratory [13], is based on PowerXCell 8i processors as accelerators. The top machines of the Green500 list [16] in 2008 are also powered by the PowerXCell 8i processor.

In this study, we introduce an implementation of the main kernel routine for simulating Lattice QCD. In this implementation, we try to provide answers to two main questions; the first question is how to SIMDize the computation in an efficient way; the second question is how to distribute the lattice data and how to handle memory efficiently.

For efficient SIMDization, we introduce *runtime data fusion*, a technique that aligns data at runtime that cannot be aligned optimally at compile time. Furthermore, while allocating lattice data on the main memory, we introduce analysis for data frames to create optimized DMA requests that removes redundancy of data transfers as well as improves contiguity of memory accesses. We investigated two techniques in doing the kernel computation; the first technique is called *full-spinor* which does the computation of one output site of the lattice in one phase of computation. The second technique, called *half-spinor*, does the computation in two phases. These two implementations have

different memory access patterns and reuse distances of the same memory location. The amount of memory references differs in these two implementations. We investigated the tradeoffs affecting the efficiency of these implementations to the Cell BE both for code SIMDization and for managing direct memory transfers. We detail how the full-spinor implementation can deliver higher overall performance at 31.2 GFlops in single precision computation and 8.8 GFlops in double precision computation for lattice size that is at least $16^3 \times 16$ per chip. The PowerXCell 8i is able to deliver 16.6 GFlops in double precision based on our implementation.

The rest of this paper is organized as follows: Section 2 introduces the Cell Broadband Engine architecture and the used software development environment. Section 3 introduces the main computation kernel routine. The SIMDization problem is tackled in Section 4. Section 5 details the proposed memory layout and the analysis leading to optimizing the memory transfers. We comment on the scalability of the introduced implementation in Section 6. We compare the Cell family of processors with other architectures for Lattice QCD computation in Section 7. Section 8 concludes this paper.

2. Cell Broadband Engine and its software development environment

In this study, we target developing an efficient implementation of the main kernel routine of simulating Lattice QCD on the Cell Broadband Engine (Cell BE). We used the IBM Cell BE SDK 3.0 [5]. We explored our implementation on the Cell BE as well as the new

Fig. 2. Architecture of the Cell Broadband Engine family of processors.

generation PowerXCell 8i with enhanced double precision.²

We used the simulator provided in the SDK to analyze the performance of our implementation and we measured the performance on an IBM BladeCenter[®] QS20 system with dual-Cell BE processors, running at 3.2 GHz.

Figure 2 outlines the basic components of the Cell BE family of processors. The Cell BE chip is composed of multiple cores; a PowerPC compatible master processor, with dual SMT (PPE) and eight synergistic processing elements (SPEs).

The execution unit on the PPE can handle control-flow intensive codes while the execution unit on the SPE is optimized to handle SIMD computations. Each SPE has a 128 register file; each register is 16-byte wide. The SPE has a small (256 kB) special memory called *local store*.

The main data is usually stored in the external memory and data are transferred back and forth with the main memory through special DMA APIs. Each SPE has two pipelines, one is mainly specialized in doing integer and floating point (FP) operations, even pipeline, and the other is mainly specialized in doing shuffling, branching and load/store operations, odd pipeline.

3. Lattice QCD main kernel routine

The main kernel routine for the Lattice QCD is for computing the actions of Wilson–Dirac operator. This

²For PowerXCell 8i, the performance reported is based on the IBM Cell simulator.

computation is performed on a spinor field (representing quarks) coupled by a gauged field (representing gluons). We ported the code adopted in the ETMC collaboration, see for instance [17,18].

Computing the actions of Wilson–Dirac operator is the most time consuming operation in simulating Lattice QCD. Equation (1) outlines the computation of the actions of Wilson–Dirac operator. This computation involves a sum over spinors (ψ_i) multiplied by gauge links ($U_{i,\mu}$) through the spin projector ($I \pm \gamma_\mu$).

$$\chi_i = \sum_{\mu=\{x,y,z,t\}} \kappa_\mu U_{i,\mu} (I - \gamma_\mu) \psi_{i+\hat{\mu}} + \kappa_\mu^* U_{i-\hat{\mu},\mu}^\dagger (I + \gamma_\mu) \psi_{i-\hat{\mu}}, \quad (1)$$

where $\gamma_x = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}$, $\gamma_y = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$, $\gamma_z = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}$, $\gamma_t = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$, I is a unity matrix, and κ_μ represents the hopping term that controls the convergence to a solution. $U^\dagger$ represents the conjugate transpose of a complex matrix U .

The representation of each gauge link is a special unitary SU(3) matrix (3×3 complex variables). Each spinor is represented by four SU(3) vectors composed of three complex variables. The routine implementing this computation is called Hopping_Matrix.

Parallelization of the Hopping_Matrix routine involves dividing the lattice into two subfields odd and even, as shown in Fig. 1. Each spinor of the odd subfield is surrounded by spinors of the even subfield

and vice versa. The computation sweeps on spinors from one subfield making the other subfield temporarily constant, thus breaking data dependency. The odd and even subfields are stored in separate grids to improve spatial contiguity of memory accesses. The computation involves gathering the data of the surrounding spinors and gauge field matrices to compute an output spinor. The computation order can follow the storage order of the output spinors. This ordering of memory accesses is one of the most bandwidth friendly orderings that will prove critical to performance, as will be detailed in Section 5.

Even though Eq. (1) shows regular computation across all sites of the lattice, the computation usually faces the challenge of the low ratio of floating-point operations to memory references. This makes this computation memory bandwidth and latency bounded.

Another implementation of this kernel decomposes the computation of each output spinor into two phases. In the first phase, described by Eq. (2), each input spinor is visited once and the intermediate results are stored in a ϕ structure. The results of the first phase are visited in the second phase based on the order of the output spinors, described by Eq. (3). The objective of this decomposition is to have a regular access for the input spinors' structure. All read data of this implementation have a good spatial locality, while irregularity is associated with data writes of the structure ϕ , in the first phase of computation. In contrast the first implementation, given by Eq. (1), has irregular reads of the input spinors, while the reuse distance of the read data is much shorter.

$$\begin{aligned}\phi_{i,\mu}^+ &= \kappa_\mu U_{i,\mu}(I - \gamma_\mu)\psi_{i+\hat{\mu}}, \\ \phi_{i,\mu}^- &= (I + \gamma_\mu)\psi_{i-\hat{\mu}},\end{aligned}\quad (2)$$

Fig. 3. Full spinor computation: computation involves pulling (reading) data with irregular access pattern from the surrounding spinors.

$$\chi_i = \sum_{\mu=\{x,y,z,t\}} \phi_{i,\mu}^+ + \kappa_\mu^* U_{i-\hat{\mu},\mu}^\dagger \phi_{i,\mu}^-. \quad (3)$$

In Section 4, we discuss the problem of efficiently SIMDizing these implementations, while the data alignment and communication are investigated in Section 5. In the following section, we will detail the attributes of the two main techniques used to perform the Wilson–Dirac kernel computation.

3.1. Computation models for the Wilson–Dirac kernel routine

The natural way of doing the computation of the Wilson–Dirac operator is to read (pull) the data of the surrounding sites to an output spinor and then do the computation over these data. The input data are streamed from eight different directions of the space–time continuum surrounding the output spinors. Given that each input spinor appears in the computation of eight output spinors, the data can be organized such that contiguity of access is observed along some of the continuum dimensions. This contiguity cannot be realized for all the dimensions simultaneously.

Collecting the input data involves a pattern of read accesses that can be less predictable by the hardware, as shown in Fig. 3. The code we ported refers to this scheme of computation as *full-spinor* technique, expressed by Eq. (1). Practical sub-lattice size (in the order of hundreds of megabytes) cannot fit in a processor cache and the access pattern of data critically limits the performance. The reuse distance of a significant percentage (more than 25%) of the input spinors is relatively short and lies within the size of level-two cache for most processor architectures.

Fig. 4. The two phases of the half spinor computations: phase one involves data that are pushed (written) with irregular access pattern to the surrounding spinors. Phase two involves regular access to the intermediate half spinors to compute the output spinors.

The alternative technique, given by Eqs (2) and (3), splits the computation into two phases, as shown in Fig. 4. In the first phase, each input spinor is visited once and the computation involving this spinor is written (pushed) to the neighboring output sites. The output half-spinor results are put in intermediate data structures. The order of writes to the intermediate structures follows an irregular pattern of accesses similar to that of the reads of spinors for the full-spinor technique. The performance is less affected by irregular data writes on most general-purpose processor. Data from the intermediate half-spinor structures are read (pulled) in the second phase to compute output spinors. The second phase of computation follows contiguous access of the output spinors and the intermediate half-spinor structures computed in the first phase. The code we ported names this method *half-spinor* technique.

Even though the half-spinor technique introduces about 7% additional memory traffic in addition to the large allocated intermediate data structure, the performance of the half-spinor technique is superior to the full-spinor technique by about 15–25% on most general-purpose architectures, for instance on Intel Xeon and Intel Itanium. The main difference between the two techniques is that the irregular access pattern is associated with the data reads for the full-spinor technique while it is associated the data writes in the half-spinor technique.

In this study we explored porting these two techniques of computation of the Wilson–Dirac operator to test their efficiency on the Cell BE architecture.

4. SIMDizing the main kernel computations on the Cell Broadband Engine

The main problem that prevents SIMDizing the computation of the main kernel routine of the Lattice QCD efficiently is the multiple patterns of accessing the same data, due to the spin projector in Eq. (1), making no single representation optimal at runtime. Each gauge link $SU(3)$ matrix is accessed twice in the positive and the negative directions. The computation may involve the original matrix or the conjugate transpose of the matrix. A gauge link matrix is usually stored in only one representation. The problem is exacerbated for spinors because each spinor is accessed in eight different contexts depending on the space direction. Each access involves different spinor vectors and operations alternating between vector addition, subtraction, conjugate addition and conjugate subtraction.

Aligning data such that all these computations are performed optimally at the same time is not possible. Different earlier approaches for SIMDizing this code align the data on one layout and then use shuffle operations to change the alignment of data at runtime to perform the needed computations, for instance the SSE2 implementation of the HMC method by Urbach et al. [18].

Another problem is that data are represented by 3×3 complex matrices and four 3×1 complex vectors, which do not map perfectly to power-of-2 data alignment. For the Cell BE processor data should be aligned to the 16-byte boundaries to be efficiently accessed. DMAs are also better aligned to 128-byte boundaries.

Efficient SIMDization of the code requires alignment of the data in a way that reduces the dependency between instructions, reduces the number of shuffle instructions and allows efficient instructions such as multiply-add to be executed.

In this work, we introduce *runtime data fusion* as a solution for the above problems of data alignment. We show that the performance can be greatly improved using this technique.

4.1. Runtime data fusion

Two conventional representations of complex structures are commonly used; the first combines the real and the imaginary parts into one structure; the second separates the real and the imaginary parts into two separate arrays. Figure 5 shows these data layouts for storing a 3×1 complex vector and 3×3 complex matrix for double precision variables aligned to 16-byte words. The complex matrix represents a gauge field link, while the complex vector represents one of four parts of the spinor data structure. Multiplication of vector of complex variables by matrix of complex variables is used intensively in the kernel computation.

Using structure of arrays representation for Lattice QCD will require a large number of source arrays in the memory, at least 42 arrays in the case of the full-spinor technique. This large number of source arrays will lead to fragmented DMA requests that reduce the observed bandwidth especially that only 16 DMA (or

DMA list) requests only can be concurrently issued. Enqueuing more DMA requests is a blocking process.

In Table 1, we list the number of instructions that are needed to do a matrix–vector multiplication. We consider the average instruction count for doing vector $\times$ matrix multiply and vector $\times$ transposed-conjugate matrix multiply. For the first representation, the real and the imaginary parts face different treatments, thus requiring shuffles. The second representation separates the real and the imaginary parts into two separate arrays, which involves additional shuffles for transposing the matrix.

The computational requirements based on these alignments favor separating the real and the imaginary parts. The number of floating point operations is reduced because of the possibility to use multiply-add and multiply-subtract instructions. These instructions

Table 1
Instruction decomposition for vector–matrix multiply on SPE

	Merged	Separate	Fused
add, sub	18	12	6
madd, msub	0	10	9
mul	22	10	9
Compute/reduce shuffles	31	14	0
Transpose/conjugate shuffles	3	5	0
Fuse shuffles	0	0	6

Note: The count represents the average for vector $\times$ matrix and vector $\times$ transposed-conjugate matrix.

Fig. 5. Multiple representations of structures based on complex variables aligned to 16-byte words: (a) the base representation; (b) the separate representation; (c) the fused representation. On the right the shuffle operations needed to multiply complex variables are shown.

cannot be used with the first layout because the real and the imaginary parts share the same 16-byte word.

Even with the larger number of instructions, the first layout is favored by the perfect alignment within the boundary of a memory word. The second layout requires to use padding (25% of the total space for spinors and 10% for the gauge field links), or otherwise the data will not be perfectly aligned. Increasing the size of the data because of alignment can severely reduce the performance of this application because it is bandwidth limited as will be detailed in Section 5. Not aligning data to 16-byte boundaries will severely penalize loading and storing data and will nullify the computation benefit achieved by reducing the instruction count. We adopted the first alignment of complex data as the base for comparison, especially that the original implementation (optimized for SIMDization on Intel SSE2) adopted it throughout the whole code for simulating Lattice QCD.

Considering the fused version in Fig. 5, the number of floating-point instructions can be reduced significantly and no shuffling is needed except at the fusion and scatter stages. The fusion introduced in Fig. 5(c) shows the two-way matrix fusion for complex variables of double precision. The fused matrix removes the need for shuffling in case of conjugate access to the array elements because the real and the imaginary parts are aligned to separate words. Transposing or conjugating the matrix does not involve additional shuffles.

This fused alignment is unfortunately not possible at compile time for all data, especially when it requires having a unique order of access. For the full-spinor technique, given a spinor i , we need to determine a unique spinor that will always precede it in the computation and a unique spinor that will always follow it. The surrounding spinors in every access context can be different and it will be a very large space overhead to keep multiple coherent copies. The gauge field

data structure can generally be fused at compile time. For the half-spinor technique, it is possible to fuse the spinor information at compile time, but accesses to the intermediate half-spinor arrays require runtime data fusion. In our experiment, we chose between runtime data fusion and compile time data fusion based on the access pattern to each particular data structure.

Because of the performance benefit associated with data fusion and the difficulty of doing it statically for most data, we consider the following proposal for *runtime data fusion*:

1. Data are fused at runtime for a number of structures depending on the number of variables per memory word, which usually involves some startup shuffle operations.
2. An optimized kernel of computation is written assuming fused data. Fused data are kept alive in registers as long as they are needed.
3. The final results for the optimized kernel (the output spinors) are then scattered back before storing them to the memory.

The steps involved in code transformations to support runtime data fusion are shown in Fig. 6. Templates for fusing complex variables are also shown on the right side of Fig. 6. Our technique involves fusing unrolled code and fusing data that can not be fused statically. The fusion process involves grouping data that will be accessed with the same pattern of access on SPU word boundary (i.e., aligning them to a 16-byte boundary). For single precision computations, 4-byte floating point, spinors are computed in groups of 4 output spinors. Consequently, the input gauge links and the input spinors are combined into groups of four (4-way fusion). For double precision, optimal alignment requires fusing the computation for two output spinors (2-way fusion).

Runtime fusion adopts the same data structure used for the base; no padding is needed. The fused data only

Fig. 6. Code transformations for *runtime data fusion* computation on the left side figure. Data templates based on runtime fusion on the right side.

exist during computation, living in registers. The main feature of the Cell SPE that allows this technique is the large register file. Merging data structures in the beginning of computation incurs minimal overhead if all fused data are kept in registers as long as they are needed for computations. Runtime fusion can prove difficult for other processors with SIMD instruction set that have small register count, for instance Intel SSE. We need to keep not only input fused data but also intermediate results in registers, especially that most data are not accessed frequently. During the computation of a group of two spinors in double precision for the full-spinor technique, almost 6 kB of memory are accessed while the register file can hold only 2 kB. Knowing that some additional registers are needed to hold shuffle patterns, intermediate results and other bookkeeping operations, careful register liveness analysis is needed to minimize the possibility of spilling the register file to the memory. We did this analysis on a basic block size of 2–2.4 kilo instructions. In our implementation, we managed to use almost 110 of the SPE 128-bit registers without the need to spilling fused data or intermediate results to the memory.

Figure 7 shows the dynamic instructions decomposition for four implementations of the base complex alignment and the fused versions. Dynamic instructions take into account all instances of each instruction encountered during the execution. We have two computation templates in terms of the shuffles needed, one for the double precision and the 2-way single precision, and the other kernel is for the 2-way double precision and the 4-way single precision. Perfect fusion kernel, 4-way single and 2-way double, provides better chance for reducing the shuffles of data. For the full-spinor

technique, the shuffling is reduced to less than 37% of the original shuffle operations for the single precision computations when we use the 4-way fusion and less than 22% for the double precision computations. For the half-spinor technique, the shuffling is reduced to less than 25% of the original shuffle operations for the single precision computations when we use the 4-way fusion and less than 13% for the double precision computations. The half-spinor technique observes more reduction in shuffles compared with the full-spinor technique because the regular access of the spinors data allows fusing part of the data statically. In general, the dynamics instructions decrease for the half-spinor technique (summing the two phases of computation) compared with the full-spinor technique by 6% for single precision and increase by 4% for double precision. Runtime shuffle operations are less for the half-spinor while there are additional overhead instructions due to splitting the computation into two phases.

Not using fusion for single precision, not presented, will incur additional overheads during transposing matrices and will provide a much worse performance.

Figure 7 shows also reductions in floating point operations, in addition to the reduction in shuffle instructions, because runtime fusion of data allows using multiply-add or multiply-subtract more frequently as clarified earlier in Table 1.

The impact on the average execution cycles per spinor computation is given in Table 2 for the full-spinor technique and Table 3 for the half-spinor technique. For these performance cycles, we aligned all data in the local store of the Cell SPE. The execution cycles are reduced significantly for the versions with fusion compared with the versions with less or no fu-

Fig. 7. Decomposition of dynamic instructions per spinor computation for the even and the odd pipelines of the Cell SPE. Two runtime fusion kernels are shown for both single precision and double precision computations based on the full-spinor and the half-spinor techniques.

Table 2
Performance summary for single and double precision computations of the full-spinor technique

	Cycles per spinor	GFlops	Bytes/FP	GB/s	Cycles/frame	LD/ST efficiency (%)
2-way single on Cell	794	51.84	0.935	48.73	50,816	98.28
4-way single on Cell	517	79.62	0.935	74.85	33,088	82.70
1-way double on Cell	7741	5.32	1.79	9.52	247,712	97.79
2-way double on Cell	4681	8.79	1.79	15.74	149,792	99.14
1-way double on PowerXCell 8i	1757	23.43	1.79	41.94	56,224	97.79
2-way double on PowerXCell 8i	824	49.96	1.79	89.42	26,368	97.28

Table 3
Performance summary for single and double precision computations of the half-spinor technique for the two phases of computation

	Cycles per phase	GFlops	Bytes/FP	GB/s	Cycles/frame	LD/ST efficiency (%)
2-way single Cell BE, phase 1	454	43.27	1.04	45.07	58,166	93.95
2-way single Cell BE, phase 2	473	45.47	0.95	43.30	60,537	83.02
4-way single Cell BE, phase 1	217	90.75	1.00	94.53	27,730	92.82
4-way single Cell BE, phase 2	225	95.37	0.91	90.83	28,862	83.50
1-way double Cell BE, phase 1	3830	5.13	2.00	10.27	245,146	95.67
1-way double Cell BE, phase 2	4111	5.23	1.83	9.56	263,129	96.63
2-way double Cell BE, phase 1	2213	8.88	2.00	17.77	141,652	97.14
2-way double Cell BE, phase 2	2464	8.72	1.83	15.95	157,751	98.13
1-way double PowerXCell 8i, phase 1	1031	19.06	2.00	38.12	66,013	94.74
1-way double PowerXCell 8i, phase 2	1126	19.09	1.83	34.91	72,092	97.63
2-way double PowerXCell 8i, phase 1	424	46.40	2.00	92.80	27,119	94.29
2-way double PowerXCell 8i, phase 2	425	50.62	1.83	92.56	27,190	96.68

sion. Considering the full-spinor technique, the reduction in execution cycles for single precision is 35% for the 4-way fusion compared with the 2-way fusion. For double precision, the reduction is 40% on the Cell BE and 53% on the PowerXCell 8i. For the half-spinor technique, the reductions in execution cycles for using fusion versus not using it are 52%, 41% and 61% for single precision on Cell BE, double precision on Cell BE and double precision on PowerXCell 8i, respectively. The access pattern associated with the half-spinor technique allows to reduce the execution cycles and to improve the performance.

In Tables 2 and 3, we define the efficiency of the memory instructions as the percentage of the minimum load/store instructions, needed by the computations to the actual load/store instructions executed during computations. As shown in the table, the memory instructions efficiency is very high in our implementation, ranging from 82.7% to 99%. Careful register liveness analysis is needed to achieve this but it is also facilitated by the large register file available on the Cell BE that makes it possible to hold intermediate results of the computation.

Tables 2 and 3 also show the performance in GFlops for these implementations and the bandwidths needed

to access the memory. Apparently, if the data are requested from the main memory system outside the Cell BE chip, then the memory subsystem will not be able to afford these bandwidths.³ Only double precision on the Cell BE requires moderate bandwidth because the execution of double precision computation is severely penalized during the issue stage of instruction execution. These tables show that the performance, in general, will be bounded by bandwidth; which justifies our choice of avoiding implementations based on data alignment that requires large padding.

As we consider lattice sizes that does not fit in the local stores of the SPEs combined, we need to process the data in chunks, we call frames. In Tables 2 and 3, a frame of data represent the maximum amount of data can be processed by the SPE without DMA intervention. The size of the frame depends on many factors, for instance if double buffering is used or not, the amount of storage needed by the code and the overhead of intermediate data, the use of single vs. double precision, etc. For the following discussion, a frame is defined as the data needed to compute 64 output spinors

³Padding is added for single precision gauge field matrix to make the number of elements even (less than 4% additional overhead).

Fig. 8. Kernel routine execution time breakdown for one spinor based on single precision and double precision computations on the Cell BE and the PowerXCell 8i architectures.

of single precision or 32 output spinors of double precision.

Efficient handling of memory defines the limits of the achievable performance on the Cell architectures for the Lattice QCD. This topic will be explored in details in the next section.

Figure 8 shows the decomposition of the execution cycles at runtime. For double precision on Cell BE, the performance is dominated by the issue stall. The performance improvement for the 2-way fusion is attributed to the reduction in the number of issued FP instructions due to using multiply-add and multiply-subtract as one instruction instead of two separate instructions. For single precision on the Cell BE and double precision on the PowerXCell 8i, the shuffle operations are reduced (fewer cycles in the critical path of the execution time is needed by the odd pipeline and fewer stall cycles on shuffles). Additionally, some nops are removed from the critical path of execution on the PowerXCell 8i with double precision computation.

Using runtime data fusion with the full-spinor technique, the un-overlapped odd pipeline cycles (implying stall of the even pipeline, responsible for FP operations) is reduced by 19%, 63% and 66% for single precision on the Cell BE, double precision on the Cell BE and double precision on the PowerXCell 8i, respectively. With the half-spinor technique, the corresponding reductions are 68%, 75% and 81%, respectively.

Most of the stalls are removed in the fused versions compared with the non-fused versions because runtime data fusion decouples the execution of the even and odd pipeline of the SPEs during most of the execution cycles. Additionally, within each pipeline, especially the odd that is responsible for FP operations, there

is a large number of independent instructions within the large basic of computing the spinor. The performance associated with the half-spinor technique is better than that of the full-spinor technique by 15% for single precision computation, while no significant difference exists for double precision computation. In the next section, the effect of data exchange with the external memory is explored.

5. Lattice QCD memory management

Traditionally, the computation of Wilson–Dirac operator is parallelized by aligning part of the lattice in a computing element’s memory and the results of computation from this computing element are communicated to the neighboring computing elements. Assigning part of the lattice per local store has the advantage of reducing the DMA requests with the memory. Applying the same model on the Cell BE is challenged by the following:

- The local store associated with the SPE is very small in size. The number of output spinors that can be computed in the local store is no more than 64 double precision spinors or 128 single precision spinors. This leads to handling trivially small sub-lattice size.
- The computation-to-communication ratio of a small lattice is severely small. The results on each local store will need to be communicated not only to the other SPEs sharing the chip but also to the SPEs on other Cell chips. The need for simulating lattice of millions of spinors leads to the need

for tens of thousands of Cell processors making the slow communication a dominating factor for performance.

- The synchronization between the SPEs will be very frequent. In Tables 2 and 3, we show the number of cycles needed to finish one frame of computation. A frame of computation is composed of 64 single precision spinors or one frame of 32 double precision spinors for full-spinor technique and double of these numbers for the half-spinor technique. Computing a frame requires cycles in the order of tens of thousands. At 3.2 GHz, these cycles leaves trivially small amount of inter-communication time and cannot scale well on a parallel machine. In our experiments, we noticed that the variation in execution time due to the wait for DMA is very large which makes it undesirable to have frequent synchronization.

The conventional approach to program the Cell BE is to store the data in the main memory, outside the Cell chip, then to bring frames of data for processing. Each SPE takes the responsibility of doing the computation for a part of the dataset. To compute one spinor, the data communicated with the external memory is 1504 bytes (assuming 16-byte alignments) for single precision computations and 2880 bytes for double precision. This leads to bytes/FP ratio equals to 0.94 and 1.79 for single precision and double precision computations, respectively for the full-spinor technique. Considering solely the bandwidth restriction of the Cell memory system at 25.6 GB/s, the upper limit on performance will be 27.4 GFlops for single precision computation and 14.3 GFlops for double precision. For the half-spinor technique, the data accessed is increased by 7% but the access pattern for the half-spinor technique is associated with moving the irregular access pattern to memory writes instead of associating the irregular access with the memory reads in the case of the full-spinor technique.

Putting data in the main memory solves the limited storage size of the SPE, but direct application of this approach will assign SPE computing threads the job of requesting data to operate upon. The data need to be aligned in contiguous memory regions to facilitate fetching them.

As explained earlier for the full-spinor technique, each input spinor generally appears in the computation of eight output spinors. The context of access (relative access with other spinors) is not the same for each of these spinor access scenarios. Because of coherence

we cannot have multiple copies easily for spinors; instead we need to compute indices of spinor locations before fetching them. An implementation of this approach on the Cell SPE will face the following obstacles:

- It is extremely inefficient to do address calculation in the SPE for spinors because of the need for control flow and integer operations for many spinor sites. These computations cannot be easily SIMDized.
- It is not possible to compute indices of spinors and store them in the local store because of the limited size of the local store. Alternatively, storing them on the main memory will add an additional step of fetching indices from the main memory in the critical path of execution, in addition to stressing the memory scarce bandwidth.
- Computing the location of each spinor individually then requesting a group of them will be associated with DMA fragmentation which can severely impact the performance and reduces the effective bandwidth observed during execution.

Similar problems exist for the half-spinor technique, associated with accessing the half-spinor intermediate array. Because storing spinors in the main memory is the solution suitable traditionally for the Cell BE and since there is redundancy on accessing the data (especially for spinors), we analyzed the data available within each frame targeting the following objectives:

- Reducing the stress on the scarce bandwidth.
- Improving contiguity of DMA requests (having fewer DMAs/DMA lists).

The next section shows that frame analysis can lead to achieve these goals at least in part.

5.1. Contiguity analysis of the data space

The data accessed during spinors computation belongs mostly to the gauge field and the spinor field. Accessing each of these data structures has different attributes, as follows:

- For the gauge field:
 1. Each gauge link is surrounded by two spinors.
 2. The gauge field is not updated during the computation of the Wilson-Dirac operator. Consequently, each gauge link can have multiple copies in separate locations because no coherence mechanism is needed between these copies.

3. The gauge links can be reordered arbitrary to improve the performance. Because two spinors are surrounding each link then preserving contiguity within one spinors subfield will require at most two copies for each gauge link.
4. The spinors surrounding a gauge link always belong to two separate spinor subfields (one in the odd subfield and the other in the even subfield). One of these spinors is updated while the other is considered constant (belonging to the constant subfield). During a sweep (even-to-odd or odd-to-even) each gauge field link appears only once.

Based on the above, the best way to access the gauge field is to replicate the gauge field matrices based on the spatial locality for each sweep of computation. This alignment can be done at compile time. Two gauge fields can be created, one is ordered for contiguity during the odd-even sweep of computation and a redundant copy is ordered for the even-odd sweep. In one sweep, no redundancy of data exists and data are organized to guarantee contiguity of access. For the half-spinor technique, the computation is split into two phases. Half of the gauge field links are visited in each phase. The gauge links used in each sweep of computation (odd-to-even or even-to-odd) should be split further into two additional arrays; one is accessed in the first phase of the half-spinor computation and the second is accessed in the second phase.

- For the spinor field:

1. Each spinor is surrounded by eight gauge links. Each spinor is accessed in eight different contexts.
2. Half the spinors are updated in each sweep of computation.

Multiple copies of the same spinor will require coherent update for all copies which will add load to the potentially overloaded memory system.

For spinors, we considered analyzing the frames of input spinors accessed during computation for contiguity and redundancy. Figure 9 shows part of the indices accessed in one frame of input spinors in the full-spinor technique, or one frame of output half-spinors in the half-spinor technique. For the full-spinor technique, the input spinors are stored in an array of $(lattice\ volume/2) \times 1$. For the half-spinor technique, the output half-spinor array is of the dimension $(lattice\ volume/2) \times 8$ and the indices represent the y -index in a two dimensional indexing.

For the full-spinor technique, computing an output spinor requires accessing one row of the input spinors. Looking at the indices of each row, we find no spatial locality. Looking across neighboring output spinors, we observe some contiguity for the input spinors streamed from the same dimension of the space, for instance the columns for $+T$, $-T$, $+X$ and $-X$. We can issue DMA requests from the memory based on their spatial locality in memory to improve the contiguity of access from the external memory. Knowing that the local store is not arranged like cache, the non-contiguous access during execution of

Fig. 9. Analysis of access pattern for a frame of spinors.

spinors in the local store is fortunately not penalized. For the half-spinor technique, we need to organize the half-spinor arrays in a column major layout to be able to issue the DMAs (using DMA list APIs) with contiguous access of the external memory.

Contiguity of access favors issuing DMAs based on column major ordering for spinors from the main memory while accessing spinors in a row-major fashion from the SPE local store during computations.

The second observation is that some of the indices within the frame are repeated for the full-spinor technique. Redundancy in the read data can be exploited to save bandwidth. No such redundancy exists with the half-spinor technique because of the x -indices are different (not shown in the figure).

Bandwidth can be saved for the full-spinor technique by bringing the non-redundant part of the frame and then by using the very fast local store to local store transfers. This reduces the stress on the memory system thus reducing the effect of the limited bandwidth on performance. We similarly search for redundant data that can be moved between consecutive frames. Effectively, two copy lists are created; the first is used for copying redundant spinors within the same frame; and the second is used to copy spinors between consecutive frames.

Because the values of spinors changes during computation, while their locations remain fixed, the analysis to create optimized DMA lists and to create copy lists is needed only once at the start of the program execution. This job is done more efficiently on PPE because the computation is control-flow intensive. The analysis on the PPE also uses the physical addresses of the spinors within the SPE local store so that absolute addresses are given back to the SPE to improve the performance.

Each SPE receives the addresses where its DMA lists are located in the start of creating its thread. In

our implementation, we allocate a buffer for information about DMAs for 64 frames in the local store (occupying about 32 kB). Each frame represents the data necessary for computing 64 single precision spinors or 32 double precision spinors. In total, a DMA buffer carries information about 2048–4096 output spinors for the full-spinor technique. The eight SPE can hold the optimized DMA information for a lattice size $16^3 \times 16$ of single precision (or half of that for double precision). In the half-spinor technique, the local store allows processing double the amount of spinors but in two stages.

To handle larger lattice sizes the optimized DMA buffer, created in the beginning of the program, can be fetched in pieces. The overhead of this process is minimal since it occurs once every 64 frames of computation and the memory request involves a small amount of data. The layout of the data in the local store including the optimized DMA buffer is shown in Fig. 10.

Benefiting from these optimized DMAs depends on whether the bandwidth is the performance bottleneck or not. Figure 11(a, b) shows the use of double buffering to overlap computation with communication. For single precision computation on the Cell BE, the DMA performance is in the critical path of execution. For double precision on the Cell BE, the computations hide the latency for the DMA completely. For the PowerX-Cell 8i, simulations show that the wait for DMA will be put back in the critical path of execution for double precision computations. The optimized DMA, shown on Fig. 11(c), reduces the stress on the DMA and introduces additional work to the computing thread. The overall execution time can be lowered if DMA is bounding the performance.

For the full-spinor technique, the computation engine with the optimized DMA and the inter-frame analysis is shown in Algorithm 1. The computation uses double buffering with the optimized DMA to

Fig. 10. Data layout inside the local store of a Cell SPE. The figure shows the 2-way fusion access pattern for the full-spinor technique.

Fig. 11. Double buffering for computing spinors in single-precision and double-precision computations. Optimized double-buffering is based on the frame analysis shown in Fig. 9. (a) Double-buffering (single precision on Cell BE, double precision on PowerXCell 8i); (b) double-buffering (double precision on Cell BE); (c) optimized double-buffering for DMA sensitive performance.

overlap computation and communication whenever possible. It also interleaves sending the results to the memory with the computation whenever possible. For the half-spinor technique, two major phases are used. The two phases are separated by an additional barrier synchronization.

5.2. Performance with DMA

In this section, we present the performance considering DMA measured on IBM BladeCenter[®] QS20 system, based on the 4-way fusion for single precision and the 2-way fusion for double precision. We also report the performance estimated by the simulator for the PowerXCell 8i with enhanced double precision engine.

We show the performance with three implementations of the DMA:

- Base DMA with no optimization: the row major access for spinors is adopted thus accessing fragmented data. We show the performance based on the full-spinor technique.
- DMA with contiguity: the spinor data in the local store are aligned in column major format and the DMAs are requested from memory for all data in contiguous fashion whenever possible. We show

this DMA scheme with the full-spinor and the half-spinor techniques. For the half-spinor technique, the intermediate half-spinor array is also aligned in column-major layout. We used the minimum number of DMA lists possible in this implementation.

- Optimized DMA: only non redundant data are requested from the memory in addition to the earlier contiguous access to memory. The data are complemented either from the data brought with the current frame or from the data available in the previous frame. This technique can be applied only to the full-spinor technique.

Figure 12 shows the execution time breakdown for the three DMA schemes on the Cell BE and the PowerXCell 8i. Enqueuing unoptimized DMA requests contributes large percentage of the execution time ranging from 50% to 90%. The main reason for this large delay is that only 16 DMA requests are allowed at a time. Exceeding this limit causes the SPE execution to stall waiting for the completion of DMAs and thus reducing the chance of overlapping computation with communication. Another problem with fragmented DMAs is that it is not always aligned to 128-byte boundaries of the address space. The frag-

Algorithm 1. SPE computation of spinors with optimized DMA for the full-spinor technique.

```

REQUEST_OPTIMIZED_DMAS(0)
for  $i \leftarrow 0$  to  $sublattices - 1$  do
  phase  $\leftarrow 0$ 
  WAIT_OPTIMIZED_DMAS( $i$ )
5:  ISSUE_FRAME_INPUTS_DMAS(phase,0)
  phase  $\leftarrow phase \oplus 1$ 
  for  $k \leftarrow 0$  to  $frames\_per\_spe - 2$  do
    ISSUE_FRAME_INPUTS_DMAS(phase,
       $k + 1$ )
    phase  $\leftarrow phase \oplus 1$ 
10:  WAIT_INPUTS_RECIEVED(phase)
    COMPLEMENT_TRANSFER( $k$ )
    WAIT_OUTPUTS_SENT(phase)
    KERNEL_COMPUTE(phase)
    TRANSFER_FROM_PREV( $k + 1$ )
15:  ISSUE_FRAME_WRITE_DMA(phase)
  end for
  phase  $\leftarrow phase \oplus 1$ 
  WAIT_INPUTS_RECIEVED(phase)
  COMPLEMENT_TRANSFER( $k$ )
20:  WAIT_OUTPUTS_SENT(phase)
  if  $i + 1 < sublattices$  then
    REQUEST_OPTIMIZED_DMAS( $i + 1$ )
  end if
  KERNEL_COMPUTE(phase)
25:  ISSUE_FRAME_WRITE_DMA(phase)
end for
WAIT_OUTPUTS_SENT(phase)
BARRIER_ALL_PROCS()

```

mentation with single precision computation is more severe compared with the fragmentation with double precision. Aligning data to 128-byte boundary is not advantageous because it will reduce the number of spinors that can be computed per frame to half.

Contiguous DMA requests are created by observing the contiguity on spinors accessed in the same frame of spinors. Enqueuing time becomes negligible and the wait time for DMAs shows up as the limiting factor for single precision on the Cell BE with about 62% of the execution time and is estimated for double precision on the PowerXCell 8i to be 66% of the execution time. The double precision performance on the Cell BE is not affected by the latency of the DMA.

Optimized DMA reduces the stall for DMA by 30–40%. Buffer repairs (complements) are less than 5% of the total execution time for single precision on the Cell BE and double precision on the PowerXCell 8i. For double precision on the Cell BE, the buffer repair takes 2.5% of the execution time.

The performance of the half-spinor technique is generally similar to the full-spinor technique on the Cell BE when the bandwidth is not critical to performance, for instance for double precision on the Cell BE. For single precision on Cell BE and double precision on PowerXCell 8i, there is a better chance to reduce the pressure on the bandwidth by removing redundancy in the case of the full-spinor technique, as discussed earlier in Section 5.

The full-spinor technique is generally not the best performing technique on general-purpose processor with multilevel cache hierarchy because it is associated with less regular reads of the data. The half-spinor technique slightly increases the bandwidth for the sake of regularizing the memory reads and makes the ir-

Fig. 12. Execution time breakdown for with different DMA for fragmented DMA, contiguous DMA and optimized DMA. Each group is normalized to the execution time with fragmented DMAs.

Table 4

Performance of Wilson–Dirac operator routine in terms of speed (GFlops) and bandwidths (GB/s) for fragmented full-spinor, contiguous half-spinor, contiguous full-spinor and optimized full-spinor schemes

	Single on Cell BE				Double on Cell BE				Double on PowerXCell 8i			
	Frag.	Half	Full	Opt.	Frag.	Half	Full	Opt.	Frag.	Half	Full	Opt.
GFlops	8.41	22.90	22.81	31.20	4.34	8.85	8.75	8.56	6.02	12.10	11.95	16.60
GB/s	7.86	21.87	21.33	23.58	7.78	16.92	15.68	12.40	10.74	21.70	21.60	24.05

regularity associated with the writing of the intermediate structure. Our study shows an interesting trade-off that architecture like Cell BE is offering compared with cache-based processors. The explicit management of the memory transfers offered by the Cell BE makes bandwidth friendly technique more desirable. The computation demonstrated in this study is bandwidth limited because of the highly optimized kernel we developed. The main challenge is how to arrange the DMA transfers such that the maximum bandwidth available by the system is observed. It is conceivable that a technique of higher bandwidth can provide better performance if it is associated with contiguous and aligned accesses.

Table 4 summarizes the performance in GFlops and the bandwidth measured in GB/s. Compared with contiguous DMA, optimized DMA delivers 36% performance improvement for single precision. PowerXCell 8i achieves 38% performance improvement for optimizing DMA on double precision computation compared with using contiguous DMA access alone. The improvement in performance is higher than the saving in bandwidth because other savings are associated with reducing the requested data such as reducing the queuing delay, reducing fragmentation repairs and reducing the memory controller occupancy.

For double precision on the Cell BE, the best performance is achieved with contiguous DMA without removing redundancy because copying spinors is added to the critical path of execution. With redundancy removal, the reduction in performance is about 2.5% while almost 26% of the bandwidth is saved which potentially can save power consumption.

Another observation is that the saving in bandwidth is different for single precision (24%) compared with double precision (26%). The bandwidth saving is dependent on the fragmentation of data and the amount of redundancy within a frame of spinors. The fragmentation for double precision frame is lower compared with single precision frame because of the larger size of the data structures representing the spinors and the gauge field links. On the other hand, the redundancy within a frame is dependent on the frame size where for sin-

Fig. 13. Redundancy within a frame of spinors.

gle precision it is 64 while it is 32 for double precision for the full-spinor technique. The bigger the frame size the better the chance to find redundancy within a frame. Figure 13 shows the increase in the amount of redundancy within a frame with the increase of frame size. Increasing the lattice size also requires increasing frame size to find the same amount of redundancy.

If the local store size increases with the future generations of the Cell processor, then the effectiveness of optimizing DMA, by removing redundancy, is expected to increase.

6. Performance scaling of the introduced implementation

In this section, we discuss the scaling of the implementation presented earlier in two dimensions. The first dimension is the scaling of performance within the Cell processor, illustrating the balance between the computational resource and memory bandwidth for the Cell chip vs. the requirements of the computation. The second dimension of scaling is the inter-nodes scaling, considering building a system that tackle a realistic lattice size.

6.1. SPEs utilization

The utilization of the SPEs within the Cell processor is dependent on the precision of computation and the

generation of the Cell family of processors. The SPEs are fully utilized for double precision computation on the Cell BE. For single precision computation on the Cell BE and double precision on the PowerXCell 8i, SPEs stall for large percentage of the execution cycles. The main reason is that the cycles that the memory controller will be fully occupied bringing one frame of data, based on 25.6 GB/s, are not less than 11.5 kCycles. Having eight SPEs per Cell BE, each SPE will have 92.2 kCycles to do computation. These cycles are about 3 times the cycles needed for computing a single precision frame on the Cell BE, or 3.5 times the cycles for double precision on the PowerXCell 8i. This shows why the SPEs will be underutilized in these cases.

Figure 14 shows the performance achieved by partial use of the Cell BE computational resources. The experiments were done on IBM Blade Center[®] QS20 system varying the used SPE count from 1 to 8.

For single precision calculation using the full-spinor technique, it is apparent that as few as four SPEs can achieve most of the performance the Cell processor can afford, considering the 4-way fusion. The performance slows down by 5% if we increase the SPEs from four to eight because we use more contending SPEs on the limited bandwidth. For single precision computation, we achieve the same maximum throughput (set by the bandwidth) for the 2-way fusion and the 4-way fusion. For single precision, 2-way fusion, we achieve the maximum throughput with 6 SPEs compared with 4 SPEs for the 4-way fusion. For double precision, the 2-way fusion steadily shows better performance compared with no fusion.

For single precision computation, the half-spinor technique delivers better performance up to two SPEs compared with the full-spinor technique. The full-spinor takeover starts with the use of three SPEs because the bandwidth becomes the dominant factor for performance.

Fig. 14. Scalability of performance with the count of the used SPEs for the full-spinor technique.

Although the flattening throughput is disappointing (at 15% of the peak throughput), it allows an additional degree of freedom for designing a multi-cell system to simulate the Lattice QCD problem. To handle the imbalance of the computational resources with the bandwidth for Lattice QCD on the Cell BE, the following alternatives can be explored:

- The memory of some of the SPEs can be used as an extended memory for the other SPEs, for instance, to hold optimized DMA information. The inter-SPE bandwidth is much higher than the bandwidth with the external memory. The Cell processor will then be able to hold information about a very large lattice size (for instance $32^3 \times 64$).
- In addition to the computing SPEs, other SPEs can manage their local storage as additional buffers for frames. Creating extended frames increases the chance of detecting redundancy within each frame and thus reducing the pressure on the memory bandwidth.
- Few SPEs can be used and the others are turned off to save power, or computing threads can migrate between SPEs to reduce hotspots on the Cell BE chip. The runtime fusion can be viewed as power-saving optimization.

6.2. Scaling of the proposed scheme on a large scale system

Simulation of the Lattice QCD problem cannot normally be realized on few nodes. A realistic lattice sizes will be at least of the dimensions $128^3 \times 256$. Given that the physical memory attached to a node cannot exceed few gigabytes allowing limited sub-lattice size, we will need a system of 8192 nodes to realize the lattice mentioned earlier, each solving a sub-lattice of the size $16^3 \times 16$.

Traditionally, simulating the Lattice QCD is bounded by the performance of a single node. We estimate that with the performance that can be delivered by accelerators such as the Cell family of processors, the bottleneck will be pushed to the interconnection network. Even though the communication for the Lattice QCD follows nearest neighbor pattern, the communication time between these nodes can be larger than computation time. A state of the art communication network such as the one used in BlueGene/P can become a bottleneck for performance if it is computationally powered by the Cell BE. We estimate that the communication time will be two times the computation time of a Cell-based node with the conventional three-dimensional topology.

Four-dimensional (4D) network topology is needed for the Lattice QCD while preserving the link speed to cut the communication time. If this 4D interconnection network is realized, we will still need techniques to overlap computation with communication. Building a customary network or developing special algorithmic techniques for an optimized communication will become a necessity. This topic requires specialized study that is beyond the scope of this work.

7. Comparison with other architectures and computation schemes for the Lattice QCD

The Cell family of processors offers an attractive alternative in terms of the raw performance and the power efficiency. Defining the power consumption efficiency as the number of watts needed to deliver one GFlop of computation, the Cell BE can achieve a power consumption efficiency of 3.21 Watts/GFlop for single precision computation based on the performance achieved in this study for Lattice QCD. The PowerX-Cell 8i power consumption efficiency for double precision is about 5.75 Watts/GFlop.

The best competitor to the Cell is PowerPC 450 adopted in BlueGene family of super-computers. At 30% of the peak performance per chip, the power consumption efficiency is about 10 Watts/GFlop. Other architectures will have significantly higher power consumption, reaching up to 30 Watts/GFlop.

Traditionally, the implementation of this routine on most architectures exploits a small fraction of the floating-point computational power because the computation is bandwidth limited. For instance, an SSE2 implementation on Intel Xeon is in the range of 0.8 GFlops per core of double precision computa-

tions [8]. The performance on BlueGene/P is measured at 0.6–1.1 GFlops per core [6] depending on the lattice size. Unfortunately, the performance does not scale linearly with the number of cores because of the dependency on the memory bandwidth. But some researchers were able to achieve 30% of the peak performance per chip [6] at about 4 GFlops.

Using NVidia G80 GPUs to implement this routine delivers 6.2 GFlops [8] (28 W/GFlop) of single precision computations. One of the main limitations on the use of this architecture is the limited bandwidth between the main memory and the GPU. Considering the data exchange time of input and output spinors through PCI express $16\times$ at 60% sustained performance of the peak 4 GB/s per direction and assuming an infinite computational power of the GPU makes the maximum achievable performance about 20 GFlops of single precision. This upper estimate assumes that the gauge links are not exchanged with the main memory (75% of the data referenced) as they are less frequently updated. If we consider the bandwidth between the GPU processors and the GPU memory banks (at about 76–84 GB/s) the performance cannot exceed 15 GFlops for the computation presented in this study.

The use of the Cell BE is also under consideration by many Lattice QCD groups. An analytical model to predict the performance limits of simulating Lattice QCD is developed [4]. This model predicted that the performance will not exceed 34% of the theoretical peak performance. Some simplified kernels were also tested on the Cell BE [10] showing a peak performance estimate of 20 GFlops of single precision computation. These studies confirmed the fact that the computation of Lattice QCD is bandwidth limited and tried to predict the performance of a real implementation. A concurrent study by Spray et al. [14] showed an implementation of a similar kernel routine that achieves 45 GFlops of single precision performance on the Cell for sub-lattices that fit within the local store. This assumption alleviates the need to retrieve the gauge field from the memory, and only few spinors are exchanged through DMA between SPEs' local stores. Our optimized kernel can achieve 79.6 GFlops for similar lattice sizes, showing a speedup of 77%.

One major advantage of our proposal in contrast with other proposals is that it is not bounded by the local store size in determining the sub-lattice size that can be assigned to a Cell BE based node. As discussed earlier we need to simulate large lattice sizes that is at least $128^3 \times 256$. If we try to build a system that simulate this lattice based on a scheme that tries to fit sub-

lattices in the local store then we will need about 350 K Cell processors. In contrast, the proposed scheme will require at most 8 K cell processors.

Additionally, we need less communication between computing nodes because we have larger lattice volume-per-chip in our implementation. For Lattice QCD, the computation is proportional to the volume of the lattice, while the communication is proportional to the surface of the lattice. The volume usually grows quicker than the surface when we increase the dimensions of the lattice. With our proposed scheme it is easy to increase the size of the sub-lattice assigned per chip based on the size of the external memory which will reduce the number of nodes needed to compute a full lattice. The new generation PowerXCell 8i allows accessing more external physical memory because of the improved on chip memory controller. In contrast, schemes that require increasing the local store size to accommodate a larger sub-lattice per chip will face a challenge especially with the trend observed in not increasing the local store size in the new generation PowerXCell 8i compared with the previous generation Cell BE.

Although the Cell family of processors shows a strong potential for scientific computing in terms of performance and power efficiency, the programming of these chips requires a tedious effort to extract their full potential. Tools that allow automatic transformation for code to the Cell are yet to mature, and finding an efficient way to program these chips is usually a tricky business. The amount of details that are given to the developer can be confusing but they allow full control of all the details of the computation and the memory management. We believe that this ease of programming problem will fade by time as the community gets more experiences with handling computation on this architecture and many of the optimizations will certainly qualify for automation in optimizing compilers.

8. Conclusions

In this study, we ported two implementations of the computation kernel of the Lattice QCD to the Cell Broadband Engine; the full-spinor and the half-spinor. We additionally studied two special design processes; the first process is SIMDizing the code to achieve good performance on this architecture; the second process involves optimizing DMA requests. We presented runtime data fusion technique for code SIMDization. This

technique aligns data at runtime to reduce the data shuffles and to allow efficient instructions, such as multiply-add, to be used.

The Lattice QCD computation favors the full-spinor technique because it is possible to optimize its access pattern to make it friendlier to the memory bandwidth. For the full-spinor technique, we showed that we can achieve on the Cell BE 79.6 GFlops for single precision computation, 8.8 GFlops for double precision computation and 50 GFlops for double precision on the PowerXCell 8i. Considering DMA without analysis makes the attained performance on the Cell BE no more than 8.4 GFlops for single precision, 4.3 GFlops for double precision and 6 GFlops on the PowerXCell 8i.

We show analysis for frames of data that can save 26% percent of the bandwidth, exploiting the small reuse distance for the full-spinor technique in addition to improving contiguity of access. The PPE undertakes the job of analyzing data frames for contiguity and redundancy removal once in the beginning of the program and then transfer to the SPEs optimized DMA requests and buffer repairs. With optimized DMA, we achieved 31.2 GFlops for single precision, 8.8 GFlops for double precision on the Cell BE and 16.6 GFlops for double precision on the PowerXCell 8i.

Acknowledgment

This work was funded in part by ANR projects: POPS, PARA and QCDNext. We would like to thank the Centre Informatique National de l'Enseignement Supérieur (CINES), France for allowing us to access their IBM BladeCenter® QS20 machines to conduct our experiments. We would like also to thank André Sez nec and the anonymous reviewers for their insightful comments and suggestions to improve this work.

References

- [1] D.A. Bader and V. Agarwal, FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine, in: *Proc. of the 14th IEEE International Conference on High Performance Computing (HiPC)*, Bangalore, India, December 2007, pp. 172–184.
- [2] F. Banterle and R. Giacobazzi, A fast implementation of octagon abstract domain on graphics hardware, in: *Proc. of the 14th International Static Analysis Symposium (SAS07)*, Kongens Lyngby, Denmark, August 2007, pp. 315–332.
- [3] R.G. Belleman, J. Bedorf and S.P. Zwart, High performance direct gravitational N -body simulations on graphics processing units – II: An implementation in CUDA, *Journal of New Astronomy* **13**(2) (2008), 103–112.

- [4] F. Belletti, G. Bilardi, M. Drochner, N. Eicker, Z. Fodor, D. Hierl, H. Kaldass, T. Lippert, T. Maurer, N. Meyer, A. Nobile, D. Pleiter, A. Schaefer, F. Schifano, H. Simma, S. Solbrig, T. Streuer, R. Tripiccion and T. Wettig, QCD on the Cell Broadband Engine, in: *The XXV International Symposium on Lattice Field Theory*, Regensburg, Germany, July 2007.
- [5] Cell SDK 3.0, <http://www.ibm.com/developerworks/power/cell/index.html>, October 2007.
- [6] J. Doi, Performance evaluation and tuning of Lattice QCD on the next generation BlueGene, in: *The XXV International Symposium on Lattice Field Theory*, Regensburg, Germany, July 2007.
- [7] G.I. Egri, Z. Fodor, C. Hoelbling, S.D. Katz, D. Nogradi and K.K. Szabo, Lattice QCD as a video game, *Computer Physics Communications* **177**(8) (2007), 631–639.
- [8] K.Z. Ibrahim, F. Bodin and O. Pene, Fine-grained parallelization of Lattice QCD kernel routine on GPUs, *Journal of Parallel and Distributed Computing* **68**(10) (2008), 1350–1359.
- [9] A. Kolb and N. Cuntz, Dynamic particle coupling for GPU-based fluid simulation, in: *Proc. of the 18th Symposium on Simulation Technique*, Erlangen, Germany, September 2005, pp. 722–727.
- [10] S. Motoki and A. Nakamura, Development of QCD code on a cell machine, in: *Proc. of Science*, October 2007.
- [11] NVIDIA Cuda 1.1, <http://developer.nvidia.com/object/cuda.html>, 2007.
- [12] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn and T.J. Purcell, A survey of general-purpose computation on graphics hardware, in: *Eurographics 2005*, State of the Art Reports, Dublin, Ireland, August 2005, pp. 21–51.
- [13] Roadrunner Supercomputing Facility, <http://www.lanl.gov/roadrunner/>, 2008.
- [14] J. Spray, J. Hill and A. Trew, Performance of a Lattice Quantum Chromodynamics Kernel on the Cell Processor, *Computer Physics Communications* **179**(9) (2008), 642–646.
- [15] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco and K. Schulten, Accelerating molecular modeling applications with graphics processors, *Journal of Computational Chemistry* **28**(16) (2007), 2618–2640.
- [16] The Green500 List, <http://www.green500.org/>.
- [17] C. Urbach, Lattice QCD with two light Wilson Quarks and Maximal Twist, in: *The XXV International Symposium on Lattice Field Theory*, Regensburg, Germany, July 2007.
- [18] C. Urbach, K. Jansen, A. Shindler and U. Wenger, HMC algorithm with multiple time scale integration and mass preconditioning, *Computer Physics Communications* **174**(2) (2006), 87–98.
- [19] P. Vranas, G. Bhanot, M. Blumrich, D. Chen, A. Gara, P. Heidelberger, V. Salapura and J.C. Sexton, The BlueGene/L Supercomputer and Quantum ChromoDynamics, *Proc. of the 2006 ACM/IEEE conference on Supercomputing*, Tampa, FL, USA, June 2006, pp. 50–57.
- [20] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands and K. Yelick, Scientific computing kernels on the cell processor, *International Journal of Parallel Programming* **35**(3) (2007), 263–298.