QDP++ on Cell BE

WEI WANG

June 8, 2009

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2009
Abstract

The Cell BE provides large peak floating point performance with a novel heterogeneous multi-core architecture. Many studies are exploring this high performance on computationally demanding scientific problems including Lattice Quantum Chromodynamics (QCD). Lattice QCD is one of the important areas in particle physics and a lot of scientific software has been written to solve it. Chroma is a Lattice QCD framework which provides a library and many applications to do Lattice QCD. In this project, various versions of QDP++, the data parallel layer, in Chroma are tried to port on Cell BE with different configurations.

The “cbexlc” is a single source compiler provided by IBM for the Cell BE. It can recognize the OpenMP directives. The newest version of the QDP++ library supports threading with the OpenMP programming model. To what extent, the threaded QDP++ library can utilize the new compiler to generate threads on the SPEs automatically is not well known. This project investigates this threading mechanism (OpenMP) of the QDP++ and analyzes the problems discovered in the porting process.

A prototype is developed to evaluate the complexity and performance of porting threaded QDP++ on the Cell BE. This prototype is implemented according to the threading interface in the threaded QDP++ and the logic of Wilson Dslash Operator in QDP++. For the comparison of the performance of the OpenMP implementation, this prototype also implements the same algorithm with the libspe2 model.

The results show that the Cell BE is powerful in dealing with the computationally massive applications in both single and double precision. However, current OpenMP threading framework is not suitable for the single source compiler programming model because of the incomplete support of the cbexlc. And the developers need substantial platform specific knowledge and low level programming skills for the sake of the high performance of QDP++ on the Cell BE.
Contents

Chapter 1 Introduction ......................................................... 1

Chapter 2 Cell BE ................................................................. 3
  2.1 Cell BE Architecture .................................................. 4
  2.1.1 PPE ................................................................. 4
  2.1.2 SPE ............................................................... 5
  2.1.3 Element Interconnect Bus ............................................ 7
  2.2 Programming on Cell .................................................. 8
  2.2.1 SDK and Library .................................................... 8
  2.2.2 Programming Models .............................................. 9
  2.2.3 Lib SPE2 ......................................................... 10
  2.2.4 MFC DMA ....................................................... 12
  2.2.5 Single Source Compiler .......................................... 16

Chapter 3 Lattice QCD and QDP++ ......................................... 18
  3.1 Lattice QCD ............................................................ 18
  3.2 QDP++ and Chroma ................................................... 20
  3.3 Wilson DSlash Operator ............................................... 20
  3.4 Threads in QDP++ .................................................... 22

Chapter 4 Portability .......................................................... 25
  4.1 Testing Environment ................................................... 25
  4.2 Compilation of QDP++ and Wilson DSlash ....................... 26
  4.2.1 Source Code ....................................................... 26
  4.2.2 Compilation Options .............................................. 26
Acknowledgements

Chris Maynard and Jon Hill provided supervision for this project. They gave me a lot of invaluable guidance in project management as well as technical assistance. Chris and Jon organized two workshops for me, one is about the programming on the Cell and the other is on the Lattice QCD. It is the first time for me to attend the scientific networking function abroad. This experience will definitely be one of my most cherish memories in the whole life.

I also got very useful advice regarding as the QDP++ and Lattice QCD from Bálint Joó and Xu Guo. With their help, I was encouraged to step into this complicate physical field and got a lot of new knowledge during the process.

I’m grateful to EPCC for providing a good study and research environment. I have really enjoyed life in EPCC during the past year. I also made a lot of new friends from all over the world in EPCC. Everyone here is brilliant.

Last but not least, I need to show my sincere gratitude to my dear parents. Without their sponsorship, I would not have had the wonderful opportunity to enjoy this splendid life in the UK.
Chapter 1

Introduction

Originally designed for multi-media and video game applications, the cell processor is potentially suitable for traditional high performance computing applications. The theoretical peak performance of the cell processor reaches 204.8 GFlop/s for single precision floating point number. For double-precision floating point operations, the performance of Cell BE drops by an order of magnitude to 14 GFlop/s. However, the PowerXCell 8i variant specifically designed for double-precision reaches 102.4 GFlop/s in double-precision calculations [1]. To what extent this high peak performance can be utilized in scientific applications has been investigated by several studies [2,3]. However, the difficulties in programming on the Cell processor prevent the exploitation of this potential. Moreover, the complicate programming models on cell obstacle the possibility of porting existing applications.

IBM published a new compiler “cbexlc” to enhance the programmability on Cell BE, while continuing to provide high performance. This compiler provides a single source compilation programming model by supporting OpenMP directives. To what extent the “cbexlc” compiler can support the real computation applications is not well known.

The Lattice Quantum Chromodynamics (LQCD) is a highly computationally demanding problem from particle physics [4]. Many libraries and applications are designed for the Lattice QCD. One of the most significant libraries is QDP++, which is part of Chroma framework. In this project, the QDP++ library is used as the test case to explore the possibility and performance to port scientific applications on the real Cell BE machine with the “cbexlc” compiler.

Because the limitation the “cbexlc” compiler, current threading model in the QDP++ can not be ported to the Cell BE successfully. In order to fulfil the target of the project, a simple prototype is developed according to the Wilson DSlash Operator logic. Three methods are implemented in this prototype: a normal scalar operation libspe2 version, a SIMD libspe2 code, and an OpenMP implementation.

The thesis is organized as follows. The architecture and programming models of the Cell BE are introduced in Chapter 2. Chapter 3 give a whole picture of the Lattice QCD and the QDP++ library. The portability of different versions of QDP++ library is described in Chapter 4. The implementation and the performance test are illustrated in Chapter 5. This Chapter also evaluates the difficulties involved in programming on the Cell BE
with the merit lines of codes (LOC). The conclusion of this project is presented in the last Chapter.
Chapter 2

Cell BE

Cell Broadband Engine Architecture, commonly abbreviated as Cell BE is jointly developed by Sony Computer Entertainment, Toshiba, and IBM, (Alliance STI). The Cell BE accelerates vector processing applications and multimedia greatly by combining a general purpose Power Architecture core with eight streamlined co-processing elements, Synergistic Processing Elements (SPEs).

![Cell BE board](image)

**Figure 2.1 the Cell BE board** [5]

Eight synergistic processing elements (SPEs), the power processing element (PPE) on the left, and the PPE’s L2 cache in the top-left.

As shown in **Figure 2.1** [5], eight SPEs are lined two sides of the high-bandwidth bus; the PPE has a 512K L2 cache connected with it. The SPE core uses in-order execution without branch predication. With this simplicity, more computing units can be put on a single board to achieve high efficiency/watt. The Cell architecture design also prioritizes bandwidth over latency with novel memory coherence architecture and favours peak
computational throughput over simplicity of programs with low level programming techniques.

For these reasons, Cell is widely regarded as a challenging environment for software development. In order to confront these challenges, IBM provides a comprehensive Linux-based Cell development platform and multiple developing models to assist developers. The potential of using STI Cell processor as a building block for future high performance computing systems has been investigated by several studies.

2.1 Cell BE Architecture

As shown in Figure 2.2, the Cell processor consists of four components: the main processor which is a two-way simultaneous multithreaded Power ISA compliant core (PPE), eight Synergistic Processing Elements (SPEs) that are fully functional co-processors, external input and output processing elements, and a specialized high-bandwidth circular data bus which is called the Element Interconnect Bus (EIB) to connect the PPE, SPEs, memory controller (MIC) and I/O interfaces.

![Figure 2.2 64-bit Power Architecture with VMX](image)

2.1.1 PPE

The PPE consists of a 64-bit RISC, dual-threaded PowerPC processor which is similar to a hyper threaded Pentium 4. The PPE consists of two main units, Power Processor Unit (PPU) and Power Processor Storage Subsystem (PPSS), as shown in Figure 2.3. The PPU is capable of running two threads simultaneously and can be appeared to software as a two microprocessors with shared dataflow. PPSS deals with the memory requests access from PPU or requests from other processors, such as SPUs, and I/O devices.
The PPE has a conventional virtual memory which is viewed as a huge array of bytes indexed from 0 to \(2^{264}-1\). The PPE’s cache memory system contains two levels of on-chip cache preserving global coherence across the system, 64 KB in level 1 and 512 KB for level 2. The processor also supports IBM’s VMX SIMD units to accelerate the multimedia application. PPE is not particularly powerful but is responsible for the overall control of whole system by running the OS to manage the computation of multiple SPEs to achieve good performance.

### 2.1.2 SPE

It can be seen in Figure 2.2 that each SPE is composed of a “Synergistic Processing Unit” (SPU), and a dedicated hardware DMA engine, known as the “Memory Flow Controller” (MFC).

For the current implementation of Cell BE, each SPU consists of a RISC processor with 128-bit SIMD unit for both single and double precision instructions [7] and a 256 KB embedded SRAM for data and instruction which is called the "Local Store" (LS). As a specialized SIMD processor, all loads, stores and arithmetic instructions are operated on 128-bit vectors [8]. As a result, there will be a high cost for computations of scalar arithmetic compared to those with vector arithmetic. Vector floating point instructions in Cell BE include not only normal add and multiply operations, but a fused multiply-add operation which takes the same time (6 cycles) to complete as add or multiply instruction.

For a perfectly pipelined program with single precision floating point, the SPE is capable of dealing with one vector multiply-add instruction per cycle [9]. For the system operating on 3.2 GHz product, each SPE gives a theoretical peak performance of 25.6 GFlop/s for single precision floating point operations. Multiplied by 8, the theoretical overall peak of SPEs arrives at 204.8GFlop/s. For double-precision floating point operations, the performance of Cell BE drops by an order of magnitude to 14 GFlop/s.
However, the PowerXCell 8i variant specifically designed for double-precision reaches 102.4 GFlop/s in double-precision calculations [1].

Unlike conventional CPU caches, the LS is not transparent to software and contains no hardware structures to predict which data to load. The address space of LS can support up to 4 GB and can be accessed by software directly. Each SPE has a large unified 128-bit, 128 entry register file which can store any type of data. Vectors can be transferred between LS and register file by SPE load/store operations on a 16B-aligned LS address. In a single clock cycle, an SPE can operate on 16 8-bit integers, 8 16-bit integers, 4 32-bit integers or a single precision floating-point number as well as a memory operation.

The system memory can not be accessed by the SPU directly. The 64-bit virtual memory addresses generated by the SPU must be passed to the SPE MFC unit from the SPU in order to invoke a DMA operation within the system address space. Cell SDK provides put and get functions to implement DMA operations by sending arguments to certain special registers efficiently. The DMA engine is able to support a maximum 16 concurrent requests of up to 16KB originating remotely or locally with a cost of around 20 cycles for each MFC command. As part of the globally coherent memory address space, addresses of local DMA requests are translated by a Memory Management Unit (MMU) before being sent on the bus.

![Figure 2.4 DMA & multi-bufferring](image)

MFC performs data transfers asynchronously, so it is possible for SPE to overlap computation and data movement. A double buffering is utilized to pipeline the memory accesses. As shown in Figure 2.4, Thread 1 issues a DMA to fetch data into one buffer, at the same time, Thread 2 work on the pre-fetched data in the other. In this way, stalls...
due to memory accesses can be reduced greatly and high effective bandwidth for real applications can be delivered.

Moreover, each SPE contains a dedicated DMA management queue which is capable of scheduling long sequences of data transfers while not interfering with the SPU's ongoing computations. A structure containing the list of addresses and sizes are composed in the LS and the location of this structure is passed to the MFC command. These DMA queues provide additional flexibility in the control model.

### 2.1.3 Element Interconnect Bus

The various on-chip system elements of Cell BE are connected by an internal communication bus, known as EIB. The EIB interconnects the PPE, eight SPE coprocessors, MIC and two off-chip I/O interfaces.

![Figure 2.5 EIB Data Topology](image)

Currently, the EIB is implemented as a circular ring which comprises four 16 Bytes wide unidirectional channels counter-rotating in pairs as shown in Figure 2.5 [9].

Physically overlapping all processor elements, the four 16B data ring connects 12 bus elements, two in the clockwise direction and two in the counter-clockwise direction. The central arbiter can support up to three concurrent data transfers for each data ring when data transfer pattern permits. The effective channel rate is 16B for every two system clock cycles, because the EIB runs at half of the system clock rate. At the maximum 12 concurrent transactions, with three active transfers for each of the four rings, the peak EIB bandwidth is 96B per clock.

**Peak EIB Bandwidth = \(3 \times 4 \times 16B \div 2 \text{ system cycles per transfer} = 96B \text{ per clock}\)**

For a Cell processor running at 3.2 GHz, the theoretical peak EIB Bandwidth can reach \(96B \times 3.2 \text{ GHz} = 307.2 \text{ GB/s}\). However, some technical constraints are involved in the arbitration mechanism for packets accepted onto the bus, as explained by IBM Systems Performance Group:
Each unit on the EIB can simultaneously send and receive 16B of data every bus cycle. The maximum data bandwidth of the entire EIB is limited by the maximum rate at which addresses are snooped across all units in the system, which is one per bus cycle. Since each snooped address request can potentially transfer up to 128B, the theoretical peak data bandwidth on the EIB at 3.2 GHz is \(128 \times 1.6 \text{ GHz} = 204.8 \text{ GB/s}\) [10].

Since there are twelve elements around each ring, the longest distance from one element to another one is six steps. Data flows on an EIB channel must move around the ring step by step. In addition, longer communication distance can reduce the available concurrency and influence the overall performance of EIB. So a data transfer requiring more than six steps to complete must take the shorter route of current channel in the other direction.

### 2.2 Programming on Cell

#### 2.2.1 SDK and Library

The PPE and SPEs have different roles in the programming on Cell BE. Typically, the PPE runs the Operating System to generate and manage the user-mode threads executing on the SPEs. For each SPE, only one single program context is supported at any time. The SPE is specifically designed and optimized for computation-intensive applications. Data and instruction need to be predicted and transferred into the LS of the SPE by explicit DMA operations. These previously loaded data and instructions can then be used by the SPU.

IBM provides a software development kit (SDK) for the Cell Broadband Engine. The SDK contains many significant tools required for the software development on Cell BE. It includes a Cell simulator that bases on a Linux execution environment image. Two chains of compilation tools are also available in the SDK. One is GNU compilation tools for PPU and SPU, including C/C++ compiler, linker; and the other one consists of IBM XLC compiler (C and C++) and IBM XLF compiler (Fortran). In order to get the runtime profile information of the programs running on the Cell BE, the SDK integrates a lot of performance tools, such as CellPerfCount and oprofile.

A low-level, standardized programming library, libspe2, is available for applications to access and manage the SPEs. Based on the low level library, many other libraries or frameworks that facilitate developing and executing parallel applications are also included in the SDK, for example, Accelerated Library Framework library (ALF) and Data communication and Synchronization library (DaCS). Except for these standard libraries, the SDK contains libraries which are developed for the purpose of high performance and domain specific applications:

- Standardized SIMD math libraries are developed for the PPU’s Vector/SIMD Multimedia Extension and the SPU;
- Mathematical Acceleration Subsystem (MASS) libraries are used to support both long and short (SIMD) vectors;
- Fast Fourier Transform (FFT) library;
Basic Linear Algebra Subprograms (BLAS) library;
Linear Algebra PACKage (LAPACK) library;
Monte Carlo Random Number Generator library;

2.2.2 Programming Models

With the rich tools and libraries provided by the SDK, multiple levels of programming models are supported for the Cell BE architecture. The SPE management library (libspe2) provides low-level functionalities for loading codes to SPEs and managing the execution of programs on the SPEs. All synchronizations and data movements are left to programmers to handle explicitly.

The “workblock” (WB) programming model is implemented based on the Accelerated Library Framework (ALF) [11]. In ALF, the main program runs on a host processor and WBs are dispatched to and executed on accelerator processors. The ALF is a flexible programming model and can be implemented on various architectures. For example, on the Cell BE, the PPE can take the role of host processor and provides eight accelerators, SPEs; the RoadRunner [12] machine is to be an x86/Cell hybrid. The hybrid design utilizes dual-core Opteron server processors with the standard AMD64 architecture to work as host processors; and makes use of a Cell BE processor using Power Architecture attached to each Opteron core to work as accelerators.

The software caching model exploits a software caching algorithm to transfer data into and out of the LS of SPEs explicitly. This programming model is suitable for programs with unpredictable memory access patterns, or having datasets larger than the size of LS. Software caching is implemented in the libcach library included in the Cell SDK rather than compiler level, so explicit cache manipulation calls of the library are needed for this model.

The single compiler programming models are investigated to reduce the complexity of programming on Cell BE. The commercial product, RapidMind SDK [13], provides a high level and relatively simple programming model for multi-core architectures. RapidMind has well defined inputs, outputs and scope. Not dividing code into PPE and SPE code separately, this model makes subprograms for Cell inlined with the main program in a single source file. Moreover, the source code can be compiled by a single compiler. IBM has published impressive performance results from a compiler known as the “single source” compiler. This compiler uses OpenMP directives to specify parallel regions to be executed in parallel and the compiler is responsible for low level tasks, such as automatic program partitioning, data virtualization, code overlay, Auto-SIMDization and etc [14]. Many other approaches to develop or port code for the Cell BE are available by Buttari et al [15].

In this project, two models for programming on Cell BE will be used for the exploitation of the potential performance of QDP++ on Cell BE: libspe2 and Single Source Compiler.
2.2.3 Lib SPE2

The SPE Runtime Management Library (libspe2) is a standardized low-level application programming interface (API) for access to the Cell BE’s SPEs. The libspe2 provides a neutral API for the underlying operating system. Applications invoke functions in libspe2 to create and manipulate the SPE context. A SPE context is a logical representation of an SPE and is handled as a base object for the operations of libspe2. Then the SPE context will be scheduled from applications onto the physical SPE with the help of Operating System according to the scheduling policy and priority [7].

![Figure 2.6 Data and control Flow for PPE and SPE](image)

(1) The SPE program is loaded to the LS. (2) SPEs are instructed to execute the SPE program by PPE program. (3) Required data is transferred from the main memory to the LS. (4) SPEs deal with data got from main memory. (5) Results of SPEs are sent to the main memory from the LS. (6) SPE program informs the termination to PPE.

Typically, an SPE can be used by an application in the following scheme. An SPE context is created by PPE with the function call `spe_context_create`, in which a pointer to the newly created SPE context is returned on success. Next, an SPE executable object is loaded into the SPE context LS using `spe_program_load`. Then the method `spe_context_run` transfers control to the OS for the actual scheduling of the SPE context to a physical SPE and executes the binary code on SPE. This function call is synchronous to the OS and will block until the completion of the task on the SPE. Finally, the SPE context will be destroyed by `spe_context_destroy`. The control and data flow of PPE and SPE program is illustrated in Figure 2.6.

In order to utilize the SPEs sufficiently, applications are necessary to utilize multiple SPEs concurrently, because the high peak floating point performance originates from the eight SPEs on Cell BE. The standard method is to create multiple threads for concurrent SPE contexts required for the program.
for ( i=0; i < NUM_THREADS; i++ ) {
    spe_contexts[i] = spe_context_create( 0, NULL );
    spe_program_load( spe_contexts[i], &hello_spu );
    thread_args[i].spe_context = spe_contexts[i];
    thread_args[i].argp = NULL;
    thread_args[i].envp = NULL;
    pthread_create( &threads[i], NULL, &spe_thread,
                    &thread_args[i] );
}

Code 2.1 Pack the SPE Context information

void *spe_thread( void *voidarg ) {
    thread_args_t *arg = (thread_args_t *)voidarg;
    unsigned int runflags = 0;
    unsigned int entry = SPE_DEFAULT_ENTRY;
    spe_context_run( arg->spe_context, &entry, runflags,
                     arg->argp, arg->envp, NULL );
    pthread_exit( NULL );
}

Code 2.2 Invoke the SPE Program

As shown in Code 2.1, the context information for each SPE is stored in a data structure array Thread_Args. After wrapping the context information, each thread will run one SPE context at a time, as demonstrated in Code 2.2. Finally, the PPE program will wait for all N threads to terminate and destroy the SPE context one by one, as shown in Code 2.3.

for ( i=0; i < NUM_THREADS; i++ ) {
    pthread_join( threads[i], NULL );
    spe_context_destroy( spe_contexts[i] );
}

Code 2.3 Wait for the end of execution

The communications and data transfers between PPE and SPEs are implemented by passing parameters from PPE to SPEs with function call spe_context_run.
#include <lipspe2.h>
//reference to the SPU image
extern spe_program_handle_t hello_spu
Int main (void)
{
    ...........
    //run SPE context
    rc = spe_context_run(speid, &entry, 0, argp, envp, &stop_info);
    ...........
}

#include <stdio.h>
Int main (unsigned long long speid, unsigned long long argp, unsigned long long envp)
{
    ...........
    //Do some interesting things
    ...........
}

Figure 2.7 Parameter Relationship between PPU and SPE program

As shown in Figure 2.7, speid is a pointer to the SPE context that will be run on the SPEs. The initial address of the execution of SPU’s instructions where the SPE program should start is set up by entry; when the SPE program terminates, entry stores the address where the SPU stops execution. The pointer to application specific data argp is passed to the SPE program as the second parameter. The pointer to environment specific data envp is passed to the SPE program as the third parameter. The correspondence of these parameters between PPU code and SPU code is marked with red line in Figure 2.7.

2.2.4 MFC DMA

The SPE is connected to EIB and exchanges data with main memory and other SPEs with the help of Memory Flow Controller (MFC). The MFC commands allow codes executing on SPEs communicate with main memory and other devices and provide the synchronization mechanism among the devices in the system.

The LS of each SPE has a Real Address (RA) within the range of the global memory space and the SPE program locates its LS address with a Local Store Address (LSA). With this addressing mechanism, privileged applications on the PPE are able to map LS units into the Effective Address Space (EA). So devices generating EAs, including the PPE and SPEs, can access the LS as regular component in the main storage system. Because of the map between LS and RA, the SPE can utilize DMA operations to transfer data between its own LS unit to other SPE’s LS unit directly. It is an efficient way to transfer data, because the DMA makes use of the high performance interconnects bus (EIB) to transfer data from one SPE to another without involving the main memory. However, the code running on the SPU can only fetch instructions and load/store data from its own LS.
The address of the main memory is represented in effective address and is translated into a special real address of the System by MMU, as shown in Figure 2.8 [5]. The MFC commands can be issued either by the code running on the SPU by channel instructions or code executing on the PPE with some stores and loads to MMIO registers. The commands issued by SPUs are queued in MFC SPU Command Queue and have an ‘l’ suffix, while the MMIO-initiated commands by the PPE are queued in MFC Proxy Command Queue and are suffixed with an ‘s’. The direction of data transfers of DMA commands are referenced from the point of view of the SPE. Therefore, get commands are those reading data from main memory to LS and put commands are those writing data to main memory from LS.

```c
(void) mfc_get( volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag, uint32_t tid, uint32_t rid)
(void) mfc_put(volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag, uint32_t tid, uint32_t rid)
```

Taking the “get” operation as an example, the pointer “ls” holds a 32b address in the LS for fetched data, while “ea” specifies the 64b Effective Address where data is fetched in the main memory. The 16B alignment for both the EA and the LS is mandatory, but 128B alignment is preferable for the sake of the performance. The transfer sizes can be 1, 2, 4, 8, 16, or a multiple of 16, but no larger than 16kB per DMA transfer. Peak performance is achieved when both the effective address and the local storage address are 128 bytes aligned and the transfer is an even multiple of 128 bytes [16]. The DMA reads and writes are non-blocking commands. They use a 5b tag-group identifier “tag” to check the status and wait for the completion of DMA commands. Tagging is not mandatory for DMA commands, but may be useful when using barrier or fence to synchronize the order of MFC commands in a command queue.
For the DMA transfers with sizes greater than 16kB, the limitation of 16kB per DMA transfer can be eliminated with a simple wrapper of multiple `mfc_get` commands as shown in Code 2.4. A loop with multiple `mfc_get` commands is used to divide the large block into multiple smaller 16kB blocks.

Libspe2 provides an alternative but efficient way to transfer large block of data with a single function call, DMA list. The DMA list command utilizes a DMA list structure to specify the size of each DMA transfer “LTS” and the low-order 32b of an effective address “EAL” in the form of map pair {LTS, EAL}. The list structure can hold up to 2048 transfer elements, each of which can transfer 16kB at most. That means the largest size of each DMA list transfer size can be up to 2048×16kB = 32768kB (32MB).

```c
void get_large_region(void *ls, uint64_t ea, uint32_t size, uint32_t tag, uint32_t tid, uint32_t rid) {
    int blocksize = 16 * 1024;
    int blocks = size / blocksize;
    int remainder = size % blocksize;
    for (int i = 0; i < blocks; ++i) {
        mfc_get(ls + i * blocksize, ea + i * blocksize,
                blocksize, tag, tid, rid);
    }
    if (remainder > 0) {
        mfc_get(ls + blocks * blocksize, ea + blocks * blocksize,
                remainder, tag, tid, rid);
    }
}
```

**Code 2.4 Wrapper of Multiple mfc_get commands**
As shown in Code 2.5, spu_mfcdma32 intrinsic is used to issue the DMA list command to get data from the main memory to local store:

\[
\text{mfc_getl (volatile void *lsa, void * list, uint32_t listsize, uint32_t tag, uint32_t tid, uint32_t rid)}
\]

The starting address of LS is specified by “lsa”. The address in LS of each block is incremented according to the size of data transferred by each list element. The effective address of the DMA list commands is generated by two parts. One is the high address of effective address (EAH) which is specified in the command initiating the DMA list, it defines a 4GB area in the main memory; the low address of EA (EAL) for each transfer element is listed in the structure \{LTS, EAL\}. 

Code 2.5 DMA List Logic

```c
struct dma_list_elem{
    unsigned int size;
    unsigned int ea_low;
};
struct dma_list_elem list[16] __attribute__((aligned (8)));
void get_large_region(void *dst, unsigned int ea_low,
    unsigned int nbytes){
    unsigned int i = 0;
    unsigned int tagid = 0;
    unsigned int listsize;
    if (!nbytes)
        return;
    while (nbytes > 0) {
        unsigned int sz;
        sz = (nbytes < 16384) ? nbytes : 16384;
        list[i].size = sz;
        list[i].ea_low = ea_low;
        nbytes -= sz;
        ea_low += sz;
        i++;
    }
    listsize = i * sizeof(struct dma_list_elem);
    spu_mfcdma32((volatile *)dst, (unsigned int) &list[0],
        listsize, tagid, MFC_GETL_CMD);
}
```
The relationship between the DMA command parameters, the local store in SPE and the main memory is shown in Figure 2.9. The DMA list structure maintains the connection between LS and the main memory. The data blocks in LS are contiguous, while the corresponding blocks in the main memory may be scattered. In this point of view, the DMA list commands can be seen as collective operations for data movement. For “getl” operation, the scattered or continuous list of blocks in main memory can be gathered into a continuous block in LS from different effective address. The same situation with “putl” applies; the contiguous block of local address can be transferred to a scattered or continuous list of EAs using the single DMA list command.

The list keeping the information of DMA list transfers is saved in local store memory. It can be reused by other DMA commands. For instance, the same list information can be used in a “getl” and a subsequent “putl” command.

2.2.5 Single Source Compiler

In order to enhance the programmability while keeping providing high performance of Cell BE, IBM utilizes the parallelization infrastructure of IBM XL compiler to develop a new compiler specified for Cell BE architecture, CBEXLC.

The CBEXLC compiler supports the OpenMP programming model by providing the programmers with the abstraction of a single shared memory address space. The regions of code which can be parallelized are specified using OpenMP directives and the compiler takes charge of duplicating and dispatching code for heterogeneous cores in Cell BE, the detailed process is described in the following.
Figure 2.10 Compilation Process of CBEXLC [14]

As shown in Figure 2.10 [14], the CBEXLC compiles the code with OpenMP directives in two phases, outlining and cloning. In first stage, the compiler recognizes the parallel code sections which are tagged with “#pragma omp” and outlines these sections by creating a new function. The new created function contains the copy of the original parallel code. After that, the compiler replaces the original code section with the outlined function call. In the outlining pass, machine independent optimizations are performed for the outlined parallel code. In second phase, the compiler clones the outlined code and makes inter-procedural analysis in the link step. Now, two copies of the outlined parallel code exist, one is for PPE and the other is for SPE, and they can be optimized independently. As this phase occurs in the link pass, various versions of codes on different platforms can be generated with different libraries, such as for the PPE and the SPE.

A runtime library is provided for CBEXLC to enable the OpenMP codes execute in parallel mode. These runtime functions in the library are inserted into the program according to OpenMP directives, and are responsible for the initialization of working threads, distribution of tasks, and synchronization of shared data, etc. The master thread runs on the PPE, and the working threads are generated by the master and distributed to the SPEs.

The GNU compiler also supports OpenMP code on newly published version (from version 4.2.1) with the compilation flag “-fopenmp”. However the GNU compiler has not provided sufficient support for Cell BE architecture. That is, the OpenMP code can be only compiled and executed in multiple threads mode on the PPE, but the master thread can not distribute the slave threads onto the SPEs, the SPEs know nothing about what is happening on the PPE.
Chapter 3

Lattice QCD and QDP++

3.1 Lattice QCD

Quantum Chromodynamics (QCD) is a significant part of the standard model of particle physics. It is a theory of describing the interactions of quarks and gluons, which are basic elements to make up hadrons (proton, neutron, etc). There are two particular features for QCD, asymptotic freedom and confinement. Asymptotic freedom means that quarks move mostly as free non-interacting particles in high energy scattering. Confinement means that when quarks are separated the forces between them will not diminish. So separating two quarks needs an infinite amount of energy.

Lattice Quantum Chromodynamics (Lattice QCD) is a well-established non-perturbative approach to simulate the interactions of composite particles on a four dimensional space-time lattice. In lattice QCD, space-time is not a continuous four dimensional space but is represented by a crystalline lattice, vertices (sites) connected by lines (links). Quarks reside only on sites, each of which is represented by 4 component spinor (complex vector); Gluons reside on the links, each of which is represented by a $3\times3$ complex matrix. The motion of the particles is simulated by partial difference equations and is replaced with finite difference. The resulting matrix equation (of order the volume of the 4D Lattice) is very badly conditioned. In the course of the simulation, this equation has to be solved many times for different configurations of the Gluon field. This makes Lattice QCD simulation extremely computationally demanding.
To serve this computational demand, massively parallel and highly scalable machines have been custom-built and optimized solely for the simulation of Lattice QCD, such as QCDOC [17] and QPACE [18]. Each QCDOC node is the PowerPC-based QCDOC ASIC, developed by IBM Technology Library, with a peak speed of 1 GFlop/s, as shown in Figure 3.1. The nodes communicate via high-speed serial links in a 6-dimensional torus with nearest neighbours [19].

The QPACE bases on the IBM powerXCell 8i processor to form a 3D torus architecture. An FPGA-based and application-optimized network processor coupled these Cell processors tightly, as shown in Figure 3.2[18].
3.2 QDP++ and Chroma

Written in C++, Chroma is a Lattice QCD framework which provides a library and applications to do Lattice QCD. Chroma depends on platform-neutral QCD Data Parallel library (QDP++) to provide lattice valued physics data objects and manipulation methods. QDP++ hides message passing layer to distribute computational tasks automatically on parallel machines. QDP++ also provides expressions, shifts, memory management and IO functions for upper layer scientific applications. Third party libraries are provided for the optimization of numerically intensive work. All the libraries are organized into loose layers, as illustrated in Figure 3.3 [20].

![Figure 3.3 QDP++ and Chroma Framework](image)

A linear algebra library (QLA), a message passing library (QMP) and a threading parallel library QMT compose the 1st level of SciDAC QCD packages. They are used by the 2nd layer which contains C data parallel library (QDP/C) and QDP I/O library. Except for QMP and QLA, QDP++ depends on QIO to provide I/O functions, and moreover, QDP++ library relies on all linear algebra operations rather than QLA solely. Highly optimized modules for kernel calculations are contained in Level 3. Scientific applications, such as Chroma, are built on the top of the above components. The highly modular and layered design allows scientists to concentrate on science and write application code which looks like maths, without concerning the underlying implementation of common linear algebra operations and computation parallelization.

3.3 Wilson DSlash Operator

The calculation of the actions of Wilson Dirac operator, which is referred as “DSlash”, is the most intensively computation part for the simulation of Lattice QCD. The Wilson DSlash Operator can be implemented with the lattice valued physics data objects and manipulation methods provided by QDP++. In the directory QDPHOME/examples,
there is a Wilson DSlash program for testing, which contains two implementations of the
Wilson DSlash operator. A simple implementation of the Wilson DSlash Operator with
QDP++ is illustrated in the following code:

\begin{verbatim}
chi[cb[ch]] = spinReconstructDirMinus(u[3]) * shift(spinProjectDirMinus(psi), FORWARD, 0) +
spinReconstructDirPlus(shift(adj(u[3])) * spinProjectDirPlus(psi), BACKWARD, 0)) +
spinReconstructDirMinus(u[3]) * shift(spinProjectDir1Minus(psi), FORWARD, 1) +
spinReconstructDirPlus(shift(adj(u[3])) * spinProjectDir1Plus(psi), BACKWARD, 1)) +
spinReconstructDir2Minus(u[3]) * shift(spinProjectDir2Minus(psi), FORWARD, 2) +
spinReconstructDir2Plus(shift(adj(u[3])) * spinProjectDir2Plus(psi), BACKWARD, 2)) +
spinReconstructDir3Minus(u[3]) * shift(spinProjectDir3Minus(psi), FORWARD, 3) +
spinReconstructDir3Plus(shift(adj(u[3])) * spinProjectDir3Plus(psi), BACKWARD, 3))
\end{verbatim}

Code 3.1 Wilson Dslash implementation 1

As shown in Code 3.1, the programming code is similar to the original mathematical
formula. The scientist does not need not to care about the implementation details. The
inputs of the DSlash procedure include a LatticeFermion object \( \psi \) containing 4 complex
3-vectors for each site and four LatticeColorMatrix objects \( U \) consisting of a 3x3
complex array for each link. The computation involves a sum over spinors \( \sigma \) multiplied by
gauge links \( U \) through the spin projector \( \pm \). After calculating, the
results are stored in a LatticeFermion object \( \chi \).
Another implementation “dslash2” can also be found in the same program. As shown in Code 3.2, two complex 3-vectors are stored in temporary LatticeHalfFermion objects for each site. They are used to represent the projected form of data.

Except for the testing program in QDP++, several libraries are developed for the operation of Wilson DSslash, including sse_wilson_dslash and cpp_wilson_dslash. The underlying implementation mechanism is similar to the simple testing program, but additional optimizations are added into the libraries according to the specific architectures.

### 3.4 Threads in QDP++

In QDP++ library, optimized functions are wrapped so that they can be dispatched to multiple threads for parallelism. In Chroma framework shown in Figure 3.2, there are two libraries utilized to enable the implementation of threaded QDP++. One is the OpenMP supporting multiple threading works with directives. The other one is the software library QMT which provides OpenMP like fork-join multi-threaded APIs [22].
The current implementation exploits a wrapper of the optimized function and the parameters needed for the optimized function to be dispatched into multiple threads. The optimized function `func` is wrapped into a standardized form as follows.

```c
void func(int lo, int hi, int myId, MyArgType* args);
```

The parameters `lo` and `hi` are the low and high indices for current thread. The parameter `args` pointing to the parameters used in the threaded function is wrapped in the structure `MyArgType`. In QDP++, the `MyArgType` is a template class which can be replaced according to the specific applications.

```c
/* Thread worker argument structure */
struct ThreadWorkerArgs {
    void *res;  /* k_Spinor: either read*/
    void *psi;  /* k_Spinor: write*/
    void *u;    /* k_Gauge field - suitably packed*/
    void *s;    /* k_Shift table*/
    int cb;     /* k_Checkerboard (source)*/
};
```

Code 3.3 Parameter Wrapper used in cpp_wilson_dslash

For example, in cpp_wilson_dslash library, the `MyArgType` is specified to structure `ThreadWorkerArgs` as shown in Code 3.3.

The QMT style of dispatcher uses the library function “`qmt_call`” to facilitate the task distribution.

```c
qmt_call((qmt_userfunc_t)func, int N, (void *)args)
```

The parameter `N` is the total amount of the sites on the Lattice, and the parameters `lo` and `hi` in the function wrapper are set by the `qmt_call` according to the number of the threads and the parameter `N`.

```
#pragma omp parallel \
    shared(numSiteTable, threads_num, a) \
    private(chucksize, myId, low, high) \
    default(shared)
{
    threads_num = omp_get_num_threads();
    chucksize = numSiteTable/threads_num;
    myId = omp_get_thread_num();
    low = chucksize * myId;
    high = chucksize * (myId+1);
    func(low, high, myId, &a);
}
```

Code 3.4 Dispatcher of OpenMP Threads

For OpenMP style of dispatcher, the parameters `lo` and `hi` are calculated with the normal OpenMP programming model in the parallel code section, as listed in Code 3.4.
The library QDP++ contains a unified interface, *dispatchToThreads*, for generating and managing threads for different mechanisms. Various methods to dispatch optimized function into multiple threads are selected according to the compilation configuration. The compilation flag “--enable-openmp” can enable the building of OpenMP dispatcher and “--with-qmt=<DIR>” selects QMT threads library install in DIR to generate threads. If no options are configured at compilation time, the interface will do nothing while just calls the optimized function directly.
Chapter 4

Portability

4.1 Testing Environment

The machine used in the project is IBM BladeCenter® QS22 which is based on the innovative multi-core IBM PowerXCell 8i processor. It is a new generation processor based on the Cell Broadband Engine Architecture. The QS22 provides extraordinary double precision floating point processing power, so the QS22 can make applications get results faster and with more precision.

<table>
<thead>
<tr>
<th>IBM BladeCenter QS22 at a glance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Form factor</strong></td>
</tr>
<tr>
<td><strong>BladeCenter Compatibility</strong></td>
</tr>
<tr>
<td><strong>Processors</strong></td>
</tr>
<tr>
<td><strong>Number of processors</strong></td>
</tr>
<tr>
<td><strong>L2 cache</strong></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td><strong>Internal disk storage</strong></td>
</tr>
<tr>
<td><strong>Optional external storage</strong></td>
</tr>
<tr>
<td><strong>Networking</strong></td>
</tr>
<tr>
<td><strong>I/O upgrade</strong></td>
</tr>
<tr>
<td><strong>Optional connectivity</strong></td>
</tr>
<tr>
<td><strong>Operating systems</strong></td>
</tr>
<tr>
<td><strong>Warranty</strong></td>
</tr>
</tbody>
</table>

Table 4.1 IBM BladeCenter QS22 Features [22]
As illustrated in Table 4.1 [22], the Cell machine for the project has two high-performance IBM PowerXCell 8i processors and sixteen enhanced double precision SPE cores, eight for each processor. The memory capacity for the machine in EPCC is 7 GB DDR2 memory.

4.2 Compilation of QDP++ and Wilson DSlash

4.2.1 Source Code

Multiple versions of QDP++ source code can be obtained from the USQCD website [20]. A CVS server is also available for the direct source code access with the following CVS configuration:

   export CVSROOT=:pserver:anonymous@cvs.jlab.org:/group/lattice/cvsroot

This is an anonymous account access and has no write ability to write files back to the CVS repository. The current Lattice QCD source code package provides several CVS modules for users to check out as shown in Appendix B. Three versions of qdp++ code are used in this project, one with the tag “2009-03-04 14:08 edwards” is partly threaded, the second one is the branch without threading with the tag “2008-09-18 14:15 chen” and the latest version which is modified according to the testing results of the first two versions.

The QDP++ library has dependencies with several third party libraries for the successful building for different environments. One of such libraries is LibXML2, it is mandatory for QDP++ to generate and manipulate the portable data result in XML format. Details of the configuration for LibXML2 are illustrated in Appendix C1.

Basically, there are two types of Wilson DSlash libraries, SSE Wilson DSlash and pure cpp version Wilson DSlash library. The host processor of Cell BE belongs to the Power PC serial which has no support for Streaming SIMD Extensions (SSE). So a cpp version Wilson DSlash library is checked out from Lattice QCD CVS repository with the module name: cpp_wilson_dslash for this project.

4.2.2 Compilation Options

Aiming to run on various platforms, the QDP++ has a lot of compilation options. For example, “--enable-parallel-arch=scalar/parscalar” is used to build QDP++ on parallel architecture, whether workstations or parallel machine with scalar; options for the optimizations performed on specific parallel machines are provided as well, including “--enable-sse” for optimized code with Intel SSE instructions, “--enable-3dnow” to execute optimized code with AMD 3DNow instructions, “--enable-qcdoc” to enable QCDOC Optimisations, and “--enable-bgl” to perform BlueGeneL Optimizations. For the threaded version, the mechanism to generate threads is selected by the option “--enable-openmp” and “--with-qmt=<DIR>”, the former one can enable the building of OpenMP dispatcher and the latter selects QMT threads library install in DIR to generate threads. The default configuration for the three libraries used in this project is listed in Appendix C.
4.2.3 Compilers

Three compilers are used to examine the portability and the performance of the QDP++ library, including GNU compiler (gcc/g++), XLC normal compiler (ppuxlc, ppuxlc++), and XLC cbexlc compiler (cbexlc/cbexlc++). The version of GNU compiler is V4.3.0 which has support for OpenMP code, the version of XLC compiler serial installed on Cell machine is V10.1. In order to automatically compile LibXML2, QDP++, cpp_wilson_dslash and execute test programs with different compilers and configurations, a shell script is written to facilitate the compilation and execution. It can be found in Appendix C.

4.3 Portability Results

For the exploration of the portability of QDP++ code on real Cell BE machine, three compilers, two “robust” versions of QDP++ (threading and no threading), two options for threaded QDP++ (with --enable-openmp and without --enable-openmp), one version cpp_wilson_dslash library and one libxml2 library are examined in the project. In order to illustrate the result clearly, the results are listed in the tables, because there are too many combinations (3×2×2=12) needed to test. In addition, a modified version of QDP++ code is also tested with the testing results and bug reports of the first two versions.

The library cpp_wilson_dslash uses the object classes and methods in QDP++, so, for each version of the QDP++ and each compilation configuration, cpp_wilson_dslash need to be compiled again. The original codes of cpp_wilson_dslash library contain several compilation bugs, and are modified for the successful compilation; the changes are listed in Appendix D.

The results for different configurations are organized in the following tables. The first three columns of the table show the compilation results of the three libraries with different OpenMP configurations. The compilation and execution results of Wilson DSlash testing program are listed in the last two columns.

<table>
<thead>
<tr>
<th></th>
<th>LibXML2</th>
<th>QDP_T_OMP</th>
<th>CPP_D_OMP</th>
<th>QDP_D_T(C/E)</th>
<th>CPP_D_T(C/E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P/P</td>
<td>P/F</td>
</tr>
<tr>
<td>XLC</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>F/F</td>
<td>F/F</td>
</tr>
<tr>
<td>CBE</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F/F</td>
<td>F/F</td>
</tr>
</tbody>
</table>

Table 4.2 Threaded QDP and CPP_WilsonDSlash with openmp

QDP_T_OMP: the threaded QDP++ library which enables the openmp flag
CPP_D_OMP: cpp_wilson_dslash library enabling the openmp flag
QDP_D_T(C/E): the testing program in QDP++ with the compilation result (C) and the execution result (E)
CPP_D_T(C/E): the testing program in cpp_wilson_dslash with the compilation result (C) and the execution result (E)

As shown in the Table 4.2, the libraries used in this test are the threaded QDP++ library with compilation option --enable-openmp (Q_T_OMP), cpp_wilson_dslash enabling
OpenMP. The result are represented by P and F, P means pass and F means fail respectively. The compilation and execution results (C/E) for test program \textit{t_dslash} in QDP++ (QDP\textunderscore D\textunderscore T) and cpp\_wilson\_dslash (CPP\_D\textunderscore T) are illustrated in the last two columns. The results for the Threaded version QDP++ without openmp flag and the unthreaded version QDP++ can included in Appendix E.

It can be seen from the results that gcc have better support for the three libraries and can compile the three libraries successfully. The situation for ppuxlc and cbexlc is not optimistic because of the complicated templates in QDP++ and cpp\_wilson\_dslash libraries. The following is a typical compilation error about template in QDP++.

\textit{The function template parameter "Arg" has been found to have two values: "QDP::u\_arg<QDP::PScalar<QDP::PScalar<QDP::RScalar<int> > >,QDP::PScalar<QDP::PScalar<QDP::RScalar<int> > >,QDP::OpAssign,QDP::UnaryNode<QDP::OpIdentity,QDP::Reference<QDP::QDPTyp..." and "u\_arg<T,T1,Op,RHS>".}

The complex templates confuse the compilers to choose the correct class to replace in appropriate positions. Dr Bálint Joó, a Chroma developer from Jefferson Lab, proposed a solution which makes this situation a little better. The solution was to explicitly templatize the dispatch\_to\_threads functions. Instead of letting compiler try to infer template parameters in the function dispatch\_to\_threads(....), the modified library explicitly specifies template parameters by change this function into dispatch\_to\_threads<T,T1,Op,RHS>( ) in 2 places in the qdp\_scalar\_specific.h file.

There were a lot of struct unordered\_XXXXX\_user\_arg {} which had 'references' as members ie:

\begin{verbatim}
struct unordered_fred_arg {
    const OLattice<T>& x;
};
\end{verbatim}

The trouble with these is that if there is no constructor function, reference members can in-principle lead to 'non-assigned' references, which are like null pointers. Original version initialized the members using the following statement:

\begin{verbatim}
unordered_fred_arg a = { x };  
\end{verbatim}

GCC can compile this statement without complaint. But for XLC compilers, it caused a warning. The solution of Balint is trawling through the library files and adding a constructor to all these structs to make them into eg:

\begin{verbatim}
struct unordered_fred_arg {
    unordered_fred_arg( const OLattice<T>& x_ ) : x(x_) {}  
    const OLattice<T>& x;
};
\end{verbatim}
Adding a constructor like this makes the compiler less confused. As far it is concerned, the object now has to be initialized on constructions and the constructor initializes any potential reference members. Then all the initialization statements need to be changed to:

\[ \text{unordered_fred_arg}(x); \]

After these changes, GNU compiler (gcc/g++) and XLC compiler (ppuxlc/ppuxlc++) can compile all the libraries successfully. But the CBE compiler (cbexlc/cbexlc++) can not do this correctly, because it can not compile the fundamental library LibXML2 with errors like:

```
./libs/libxml2.a(globals.o): In function `__xmlOutputBufferCreateFilenameValue':
/home/s0897477/Cell/download/libxml2-2.6.19/globals.c:(.text+0x0): multiple definition of `xmlSaveNoEmptyTagsThrDef'
```

The solution used in this project is to replace the CBE compiler (cbexlc) with XLC compiler (ppuxlc) to compile the libxml2. QDP++ and cpp_wilson_dslash still use the CBE compiler (cbexlc). With this tiny change, all the libraries can be compiled successfully, if the option `--enable-openmp` is disabled (Table 4.3).

<table>
<thead>
<tr>
<th></th>
<th>LibXML2</th>
<th>QDP_Latest</th>
<th>CPP_D</th>
<th>QDP_D_T(C/E)</th>
<th>CPP_D_T(C/E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P/P</td>
<td>P/P</td>
</tr>
<tr>
<td>XLC</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P/P</td>
<td>P/P</td>
</tr>
<tr>
<td>CBE</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P/P</td>
<td>P/P</td>
</tr>
</tbody>
</table>

Table 4.3 Modified QDP and CPP_WilsonDSlash without openmp

However, a problem occurs when the “openmp” flag is enabled (Table 4.4), errors like the following are generated:

```
1586-502 (S) Function pointer call identified in function "_ZN3QDP19dispatch_to_threadsINS_8user_argINS_7PScalarINS2_INS_7RScalarIiEEEEaaaaEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE...
```

Table 4.4 Modified QDP and CPP_WilsonDSlash with openmp
IBM staff have confirmed that “there a limitation that current OpenMP compiler (cbxelc) doesn’t allow calling through function pointers in the parallel (OpenMP) regions” [23].

4.4 Performance for different compilers

Two Wilson DSlash Operator testing programs in QDP++ and cpp_wilson_dslash libraries are used to examine the performance of QDP++ on Cell BE with respect to different compilers.

Figure 4.1 Performance of DSlash in QDP++ with gcc and openmp

Figure 4.2 Performance of DSlash in cpp_wilson_dslash with gcc and openmp
Current GNU compiler supports OpenMP directives by using the compilation flag “fopenmp”, but to what extent the GNU compilers can support OpenMP on Cell BE is not well known. In order to investigate the OpenMP support of GNU compiler on Cell BE, the QDP++ and cpp_wilson_dslash libraries are compiled with the “--enable-omp” option using gcc/g++ with “fopenmp”. By exporting the thread number with “export OMP_NUM_THREADS = $number”, the performance of “dslash” testing program in two libraries is illustrated in Figure 4.1 and Figure 4.2. It can be seen from Figure 4.1 that the performance of dslash program in threaded QDP++ goes down with the increasing number of the threads. That means the benefit of parallelism is less than the overhead of threading for the testing program; and for cpp_wilson_dslash, the peak performance appears when the number of threads equals to 4.

From the two testing results, it can be concluded that GNU compiler does not support OpenMP on Cell completely. It can generate OpenMP threads only on PPE, while not utilizing any SPE on the Cell BE. As shown in Figure 4.2, the peak performance comes from the two hyper-threaded cores (like four cores), two threads runs on each hyper-threaded core.

The XLC compiler “ppuxlc” is able to compile the threaded libraries but can not generate multiple threads on Cell, even on PPE. The compiler “cbexlc” even can not compile the code with threads.
Chapter 5

Prototype Implementation and Performance

5.1 Issues in implementation on QDP++

In the initial project plan, the first step is to try and port a recent version QDP++ library to the Cell BE. In second phase, the performance of the QDP++ library with OpenMP threading is tested. Finally, the libspe2 programming model is utilized on the same code to compare the performance with single source compiler model.

There are three issues which present obstacles of varying degree to this plan. Firstly, the original library uses a lot of meta-programming technologies such as nested C++ templates. IBM XLC serial compilers can not recognize some complex templates automatically. So some implicit definition statements need to be changed with explicit ones as introduced in section 4.3.

The second one also involves the IBM compilers. As mentioned in Section 4.3, errors occur when compiling the libraries with OpenMP threading enabled. It results from no support on function pointers for “cbexlc” compiler in an OpenMP parallel region, while the current implementation of the threaded code (OpenMP or QMT) uses function pointers in the uniform invoking interface dispatchToThreads (Section 3.4). After testing this issue with real code on both Cell BE and HPCx (Appendix F) with the same serial IBM compilers, this issue has been confirmed by the staff in IBM [23]. This prevents porting the OpenMP threaded versions of QDP++ to the Cell BE.

The last issue has to do with the libspe2 programming model. In current threaded QDP++ implementation, an interface introduced in Section 3.4 is utilized for threading of the optimized functions. This interface is invoked whenever and wherever these optimized functions are called. However, for libspe2 programming model, each function however, needs to be separated from the original code as a single piece of code for the SPE. Moreover, the libspe2 model needs explicitly transferred data between main memory and the LS of the SPE for the sake of high performance. So every data structure used in QDP++ or cpp_wilson_dslash libraries must be known clearly in advance. However, QDP++ uses a lot of nested templates to build the data structures which are hard for libspe2 to recognize and transfer. So the current threaded implementation of the QDP++/OMP can not be ported to the Cell BE in a 16 week project.
5.2 Simulation of the Problem

Based on the above three issues, a simple program is developed as a prototype to simulate the operations on Lattice. The performance of the prototype program is then used to evaluate the possibility and performance of porting current QDP++ library to Cell. Two programming models are examined, including single source compilation (OpenMP) and libspe2 programming models.

The prototype follows the style of the threading interface used in QDP++ and cpp_wilson_dslash. The OpenMP version replaces the function pointer with the function body. For the libspe2 implementation, the function pointer is replaced by a SPE context handler:

```c
static void dispatchToThreads(spe_program_handle_t func,
void * sorce,
void * result,
void * u,
void * s,
int cb,
int n_sites);
```

This handler points to the SPE executable code which is compiled into a dynamic library.

As mentioned in Section 3.3, the inputs of the DSlash procedure include a LatticeFermion object \( \psi \) containing 4 complex 3-vectors for each site and four LatticeColorMatrix objects \( \mathbf{u} \) consisting of a 3×3 complex array for each link. The computation involves a sum over spinors \( \sum (\pm \delta \gamma^5 \delta \mathbf{U}) \) multiplied by gauge links \( \mathbf{U} \) through the spin projector \( \gamma^\alpha \). After calculating, the results are stored in a LatticeFermion object \( \chi \). The computation is numerically expensive and the computation rate of each site arrives at 1392 Flops/site [20]. The underlying operation for each site is a complex number multiplication of a matrix and a vector, and the result is kept in a vector. To simplify the problem in the prototype, the multiplication is replaced as the following assumption:

Lattice: A M×M matrix with spinor and links matrix for each element in a 2-dims space

Spinors: 4 sets of complex 3vectors is replaced with a 12 real vector \( \mathbf{B} \)

\((b_1, b_2 \ldots b_{11}, b_{12})^T\)

Matrices on links (\( \mathbf{U} \)):9 real number / link * four links is represented by a 12* 12 matrix \( \mathbf{A} \):
The result of each site is stored in a real vector \( C = B \times A \). The computation amount for each site is 288Flops/site: 12 multiplications, 12 additions for each element in result vector.

Basically, there are four steps in the prototype to complete the simulation. In the first step, the PPE divides the data set into blocks and generates the block information (high and low index of the block) according to the number of available SPEs. Then this block information is transferred to the SPEs. In the second step, every SPE gets its own block information to start the DML list command to transfer data from the main memory to the LS. Thirdly, each SPE follows the process introduced in section 5.2 to evolve the data set. In the end, the final result is sent back to the main memory from the LS of the SPEs.

The multiplication of the simulation above is implemented in two different ways to exploit the influence of different algorithms. Taking a 4-dims matrix as an example, the difference of \( Y = A \times X \) is described as follows.

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 4 & 5 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 3 & 4 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

The first one is dot product, as shown in Figure 5.1. For each element in the result vector, the corresponding row vector register is multiplied by the x-vector, then the product result is performed a sum reduction to get the final result in y-vector. In the prototype, this algorithm (named after "scalar") does not utilize the SIMD intrinsic of libspe2.
The second one assumes that each column of the matrix, the x-vector and y-vector in Figure 5.2 is gathered in a vector register separately. Each element in the x-vector replicates its value into all slots in a register. This replicated register is multiplied by the column register. Then the result is added to the result register y-vector. The algorithm (named after “simd”) exploits the SIMD intrinsic to improve the performance.

Except for the two methods above, the prototype implements the same simulation with the OpenMP programming model (named after “openmp”). All the synchronizations and data transfers are left to the “cbexlc” compiler to complete.

As to the data decomposition, 1D block decomposition in the time dimension is used to divide the lattice in John C. Spray’s dissertation. The work is distributed across the SPEs. This is inherently load-balanced as long as T is a multiple of the number of SPEs [24].

In this project, the decomposition is based on the sites in the Lattice, each SPE get a data set based on two variables, low and high index of the row, as shown in Figure 5.3. This is used to compatible with the threading interface in QDP++ and cpp_wilson_dslash. Each block is distributed into each SPE’s LS, and the communication of halos can be implemented with DMA command which supports direct access to other SPEs. However, this is not completed in current prototype because of limited time.
5.3 Tuning Parameters

The prototype program is run on an IBM BladeCenter® QS22 described in section 4.1. Each result is based on an average over five runs, in which the highest and the lowest ones are eliminated.

Three parameters are tuned to investigate their influence on the performance. The parameters can be specified with multiple options provided by the prototype program. The count of the iteration of the simulation is decided by the option “ITERATIONS”. The number of the SPEs is set with the option “THREADS”. The size of the matrix simulating the size of the Lattice is determined by parameter size “M”. In addition, the type of the real number is defined with “#define REAL float”, so it can be changed to double precision easily. The program is run on 1, 2, 4, 8 SPEs to provide an indication of the influence of different parameters and the effectiveness of the parallelisation strategy.

The performance of the prototype is evaluated by the computation rate (MFlops/s) which can be obtained from the following formula:

\[ \text{MFlops/s} = 288 \times M \times M \times \text{Iterations} / \text{time} \]

For libspe2 implementation, the value of “time” includes the time spent on the division of the data set, getting data from main memory, computation and sending back the result from the LS. The time for the OpenMP code is the execution time of the parallel section.

5.3.1 Tuning Iterations

The libspe2 programming model needs transferring data between main memory and the LS of the SPEs explicitly. The time spending on this explicit transfer is calculated in the execution time of the program. So there is a trade-off between this overhead and the benefit from multiple threads.
It can be seen from Figure 5.4, the performance of the “scalar” and ”simd” drops with the increase of the number of threads. That means the overhead of the explicit transfer of data is larger than the benefit of parallel computation when the simulation is performed only once. When it comes to the “openmp”, the compiler takes charge of the data transfer, synchronization and optimizations and generates the SPE stubs. When the program starts to execute, it will take more time on running codes generated by the compiler to distribute the SPE stubs and transfer data between the main memory and the LS of SPEs. So the value keeps steadier but lower than “scalar” and “simd”. In order to make full use of the SPEs, the task done in each SPE must be massive enough to eliminate the overhead mentioned above.

The value of the “ITERATIONS” is tuned from 1 to 1E5 to evaluate when the benefit of parallel computation overwhelms the load of transferring data from main memory to the LS of the SPEs.

As shown in Figure 5.5, the performance for each SPE increases for all the three methods in the prototype. When the number of iterations arrives over 1E4, the performance tends to level off. There is a dramatic increase for the “simd” code when the number of iterations equals to 1E3. The peak performance of the “simd” code in the prototype arrives at 11.795GFlops/s with 8 SPEs (1.47GFlops/s×8SPEs).

The result is far below the theoretical peak performance for single precision, 204.8GFlops/s. The reason for this is that the computation in the prototype is not massive enough compared to the real scientific code. There are only 288 floating point operations for each site, but in the real code the number is 1392 Flops/site [20]. The Cell BE has no support on branch predication and out-of-order execution, so if the computation for loop is too simple, or there are too many branches in the computation, the performance will be
affected to some extent. After unrolling the inner loop four times, the performance of each SPE increases to 3.34GFlops/s. That means the peak performance for 8 SPEs arrives at 26.27GFlops.

It also can be found from the Figure 5.5 that the performance of “**simd**” is about 5 times as much as that of “**scalar**” code, and more than 20 times better than that of “**openmp**” code. The source of the huge power originates from the SIMD intrinsic. After interchanging the rows and columns of the matrix U, each column in the original matrix can be stored in the vector declared with the following statement.

```
MY_ALIGN(vector REAL u[12][12/VECSIZE],128);
```

<table>
<thead>
<tr>
<th>Scalar Code</th>
<th>SIMD Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for(i=0; i&lt;100; i++)</code></td>
<td><code>for(i=0; i&lt;100; i++)</code></td>
</tr>
<tr>
<td><code>{</code></td>
<td><code>{</code></td>
</tr>
<tr>
<td><code>for(m=0; m&lt;12; m++)</code></td>
<td><code>for(m=0; m&lt;12; m++)</code></td>
</tr>
<tr>
<td><code>{</code></td>
<td><code>{</code></td>
</tr>
<tr>
<td><code>for(n=0; n&lt;12; n++)</code></td>
<td><code>n</code></td>
</tr>
<tr>
<td><code>chi[i][n] += u[m][n] * psi[i][m];</code></td>
<td><code>temp = chi[i][m];</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>chi[i][w] = spu_madd(u[w][0], psi0, temp);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>chi[i][m+1] = spu_madd(u[m][1], psi1, temp);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>chi[i][m-1] = spu_madd(u[m][2], psi2, temp);</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

Figure 5.6 SIMD optimization of the vector-vector multiplication

The value of “**VECSIZE**” is decided by the type of real number, 4 for single precision and 2 for double precision. Vector floating point instructions in Cell BE include not only normal add and multiply operations, but a fused multiply-add operation which takes the same time (6 cycles) to complete as add or multiply instruction. For a perfectly pipelined program with single precision floating point, the SPE is capable of dealing with one vector multiply-add instruction per cycle [9]. After vectorization, the normal scalar vector-vector multiplication loop is replaced by the SIMD intrinsic spu_madd (A, B, C) for each row in the matrix, as shown in Figure 5.6.

### 5.3.2 Size of Data Set

In the prototype, the data transfer and computation are divided into two phases. As to the transfer phase, the size of each DML list transfer should not be larger than the size of the LS (256KB), otherwise, segment fault will come out. In order to transfer larger block to each SPE, an outer loop is added in the SPE code to divide the large block distributed to each SPE into multiple blocks.
The size of data set for each site in the prototype equals to 672B for single-precision $(144+12+12)*4$, in which the size of the matrix U is 122, the size of the spinor vector is 12 and the size of the result 12. So the largest number of sites each SPE can handle every DMA transfer is about $390(256*1024/672)$. In the prototype, the size of the small blocks is set to 256, as shown in the code above. If the number of the sites distributed to the SPE is larger than 256, the SPE needs transferring data multiple times.

```c
for(k=0; k<sites; k+=256)
{
    transfer_large_region((void *)&psi,
                (unsigned int)speArg.psi+12*sizeof(REAL)*log,
                (12*sizeof(REAL)*sitesN), 1);
    transfer_large_region((void *)&u,
                (unsigned int)speArg.u+144*sizeof(REAL)*log,
                sizeof(REAL)*144*sitesN, 1);
    //start of computation/
    //............................../
    //end of computation/
    transfer_large_region((void *)&chi,
                (unsigned int)speArg.res+12*sizeof(REAL)*log,
                (12*sizeof(REAL)*sitesN), 0);
}
```

Figure 5.7 influence of different size of “Lattice”

When fixing the number of the SPEs to 8 and the count of the iterations to 1000, the size of the data set is changed from 16×16 to 1024×1024. As shown in Figure 5.7, the performance for each SPE initially increases with the size of the Lattice, leveling off at around 256×256.

As mentioned in Chapter 2, at the maximum 12 concurrent transactions, with three active
transfers for each of the four rings, the peak EIB bandwidth is 96B per clock. For a Cell processor running at 3.2 GHz, the theoretical peak EIB Bandwidth can reach up to 204.8GB/s between all the PEs. However, the peak main memory bandwidth available to all the SPEs is effectively 25.6GB/s. All the SPEs need simultaneously competing for access to main memory. By changing the iterations of the computation to 0 and set the size of the data set to 512×512, the data transfer rate of each SPE in the prototype can be obtain using the following formula:

$$168 \times \text{sizeof(REAL)} \times M \times M / \text{secs} / 1.0E^9$$

The result is about 19.66GB/s which is similar magnitude to the peak main memory bandwidth, 25.6GB/s. So it is reasonable to conclude that the main memory bandwidth is the main bottleneck to prevent the increase of the performance.

5.3.3 Float Precision VS Double Precision

The theoretical peak performance of the cell processor reaches 204.8GFlop/s for single precision floating point number. The earlier version Cell BE does not support double-precision floating point operations very well: the performance drops to 14GFlops/s. However, the PowerXCell 8i variant used in this project is specifically enhanced for the double-precision. The performance for double-precision calculations reaches half of the single precision floating point computations, up to 102.4 GFlop/s [1].

![Figure 5.8 Single Precision Performance of the Prototype](image-url)
From Figure 5.8 and Figure 5.9, the shape of the performance curve for single-precision is similar with that of double-precision. And the value for single precision is almost twice of the value for double precision. So the performance of the PowerXCell 8i on double precision can be well exploited in the Lattice QCD.

5.3.4 Performance of OpenMP Model

The OpenMP implementation is tested on both Cell BE and Ness. The size of the data set is set to 32*32 and the iteration of the main loop is set to $1E^4$. 
The performance for the same OpenMP code of the prototype varies on different platform with different compiler. The code running on the Ness is compiled with the “pgcc –O3 –mp” and the code executing on the Cell is compiled with “cbexc –O3”.

It can be seen from Figure 5.10 that the performance on Ness is much better than that on the Cell with same number of threads. For example, the performance for 8 threads on Ness is more than 4 times as well as that on the Cell. But the higher performance one Ness, 2.215GFlops/s, is still very low compared with 12GFlops/s of “simd” code on the Cell. The situation for double precision is same as the single precision, as shown in Figure 5.11.

![Figure 5.11 OpenMP code Performance on Cell and Ness for Double Precision](image)

### 5.4 Code Complexity Analysis

The source code of the three methods in the prototype is evaluated to get the picture of the tradeoffs between the performance and the development effort. The project uses the number of lines of code (LOC) as the merit to assess the code complexity. The LOC is a count of the number of statements in which whitespace, newlines and comments are excluded.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD Unrolling</td>
<td>218</td>
</tr>
<tr>
<td>SIMD</td>
<td>165</td>
</tr>
<tr>
<td>Scalar</td>
<td>143</td>
</tr>
<tr>
<td>OpenMP</td>
<td>23</td>
</tr>
</tbody>
</table>
Table 5.1 LOC for different implementations of the prototype

Table 5.1 shows the results of code complexity of the three implementations of the prototype. For the OpenMP implementation, which does not have SPE code, the code in the parallel section is used for the metric calculation. For the libspe2 implementation, whether normal scalar one or the SIMD one, the metric calculation includes both the PPE code and the SPE code. The PPE code contains those generating and controlling the SPE context code, and the SPE code include the explicit DMA list data transfer code and the computation code.

The libspe2 implementation with SIMD has almost 6 times as many LOC as the OpenMP implementation. The normal scalar version of the libspe2 implementation in this prototype shares the same PPE code and data transfer code in the SPE code with SIMD code. The computation for this prototype is relatively simple, and the computation logic contains no more than 10 LOC in the main loop. So the LOC for SIMD and scalar implementation is very close. The LOC of the most inner loop in the scalar version is 1, but the same logic in SIMD is 15 for single precision and 30 for double precision. When the data layout for the real application is abnormal, more SIMD intrinsic are needed to shuffle the data to get a better performance. Moreover, loop unrolling is often used to enlarge the body size of the loop for the improvement of the performance. After unrolling the loop twice, the LOC of SIMD explodes to 218, a 32.12% increase compared to the original SIMD code.

Overall, the low-level libspe2 implementation which obtains the best performance has long and complex code, while the high level OpenMP code which performs poorly has short and simple code.
Chapter 6

Conclusions and Future Work

In John Spray's project [24], he provided performance of four programming models on a "vaxpby" operation (z = ax+by operation in QDP++). But the work did not show whether the threaded QDP++ with OpenMP can be ported on the real Cell machine. It also did not evaluate how we can make use of the OpenMP programming model on the threaded QDP++ with the single source compiler “cbexlc”. John also provided a high performance Wilson Dslash for the Cell BE, in which the data decomposition is based on the time dimension only. However, the data decomposition implemented in current threaded QDP++ is based on the Lattice with high and low index.

The most important outcome of this project was the portability result of threaded QDP++ with OpenMP from Chapter 4 and the prototype evaluation for the real application in Chapter 5. The former one detected and fixed some bugs to port the QDP++ and cpp_wilson_dslash library with OpenMP to the Cell. It also provided a reasonable configuration and testing process of the QDP++ and cpp_wilson_dslash library.

The latter one provided a proof-of-concept for running LQCD calculations with single or double precision floating point numbers on the Cell processor. However, the OpenMP implementation of the two libraries can not be ported to the Cell with current threading interface in the 16 weeks project, because the single source compiler “cbexlc” does not have the support on the function pointer in OpenMP parallel region. Only after replacing all the function pointers in the QDP++ with the explicit function code, can the OpenMP implementation be ported to the Cell.

The result of the prototype also showed that the performance of the OpenMP implementation is not only much lower than the performance of the low level lipspe2 implementation, but lower than the same OpenMP code on Ness. The result of the prototype illustrates that the main memory bandwidth is the main bottleneck of the performance (Figure 5.7). This can be improved by utilizing the double buffer technology to implement the overlapping of the data transfer and computation. However, there is not enough time for the project to so more simulation with the prototype. In order to make full use of the Cell BE, the application must explore the power of the vector floating point instructions for the Cell BE. However, this programming complexity is far beyond the ability of the scientists without the specific knowledge of the Cell BE. Moreover, code optimisations also can be done on the low
level programming model libspe2, such as loop unrolling. But again, same with SIMD intrinsic, this technology will result in the code size’s explosion. Chapter 5 evaluated this complexity with the lines of the codes in the prototype.

In conclusion, the OpenMP programming model with the “cbexlc” compiler is not suitable for the newest threaded versions of QDP++. There still many challenges for the scientific application programmer to make sufficient use of the Cell BE and significant platform-specific knowledge is demanded to obtain good performance.
### Appendix A

#### Definition

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell BE</td>
<td>Cell Broadband Engine</td>
</tr>
<tr>
<td>PPE/U</td>
<td>PowerPC Element/Unit</td>
</tr>
<tr>
<td>SPE/U</td>
<td>Synergistic Processing Element/Unit</td>
</tr>
<tr>
<td>EIB</td>
<td>Element Interconnect Bus</td>
</tr>
<tr>
<td>PPSS</td>
<td>Power Processor Storage Subsystem</td>
</tr>
<tr>
<td>MIC</td>
<td>Memory Interface Controller</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>b</td>
<td>bit</td>
</tr>
<tr>
<td>B</td>
<td>Byte</td>
</tr>
<tr>
<td>QPACE</td>
<td>Quantum Chromodynamics Parallel Computing on the Cell Broadband Engine</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
</tbody>
</table>
Appendix B  Modules in QDP++

* qmp: SciDAC QMP/C message passing interface
* qmt: SciDAC QMT multi-thread package
* qio: SciDAC QIO/C intermediate level input-output package
* qla: SciDAC QLA linear algebra library
* qdp: SciDAC QDP/C data parallel library
* qdp++: SciDAC QDP/C++ data parallel library
* qopqdp: SciDAC QOP high level operations library for QDP/C
* chroma: Main production physics code over QDP++ using C++
* bagel_qdp: Generates optimized code using the Bagel package
* szin: SZIN code (over C/M4)
Appendix C Building Scripts

C1. Cross-compiling libxml for Cell

${LIBXMLHOME}/configure --prefix=${HOME}/Cell/install --host=none
--build=none --disable-shared --without-zlib --without-python --without-readline
--without-threads --without-history --without-reader --without-writer --with-output
--without-ftp --without-http --without-pattern --without-catalog --without-docbook
--without-icnv --without-schemas --without-schematic --without-modules
--without-xptr --without-xinclude CC=CCOMPILER CXX=CXXCOMPILER
CFLAGS="" CXXFLAGS=""

C2. Cross-compiling QDP++ for Cell

${QDPHOME}/configure --enable-precision=double --prefix=${HOME}/Cell/install
--with-libxml2=${HOME}/Cell/install --enable-openmp
--enable-parallel-arch=scalar
CC=CCOMPILER CXX=CXXCOMPILER CFLAGS="" CXXFLAGS=""

C3. Cross-compiling cpp_wilson_dsllash for Cell

${CPPHOME}/configure --prefix=${HOME}/Cell/install --enable-parallel-arch=scalar
--enable-openmp --with-qdp=${HOME}/Cell/install CC=CCOMPILER
CXX=CXXCOMPILER CFLAGS="" CXXFLAGS=""

C4. Compilation and Running Scripts for Cell

#!/bin/bash -e
CELLDIR=${HOME}/Cell
SCRIPT=${CELLDIR}/script
SOURCE=${CELLDIR}/download
TARGET=""
TIME=`date +%d-%H-%M`
source ~/.bashrc
if [ ! -d ${SCRIPT} ];then
    echo "$SCRIPT does not exist"
    exit 0
    #or mkdir dir, copy script
fi
if [ ! -d ${SOURCE} ];then
    echo "$SOURCE does not exist"
    exit 0
    #or mkdir download, copy source code
fi
echo "#### compile and run without openmp ####"
echo "#### compile qdp++ threaded version ####"
for COMPILER in gcc ppgcc cbe
do
  for TARGET in libxml2 qdp cpp
    do
      echo "build ${TARGET}" 
      echo "......start clean ${TARGET} with compiler ${COMPIELR}......" 
      pushd ${SCRIPT}/${TARGET}
      echo "now in the directory \"pwd\"" 
      if [ ! -d error ]; then 
        mkdir error 
      fi 
      rm -rf ./error/*.* 
      make clean 
      echo "......end make clean ${TARGET} with compiler ${COMPIELR}......" 
      echo "......start configure ${TARGET} with compiler ${COMPIELR}" 
      echo "now in the directory \"pwd\"" 
      sh ./config-${TARGET}-${COMPIELR}.sh 
      2> ./error/${TARGET}-${COMPIELR}-conf-${TIME}.error
      #gmake uninstall 
      echo "......end configure ${TARGET} with compiler ${COMPIELR}" 
      echo "......start install ${TARGET} with compiler ${COMPIELR}" 
      gmake -j 4 all 2> ./error/${TARGET}-${COMPIELR}-all-${TIME}.error 
      gmake -j 4 install 2> ./error/${TARGET}-${COMPIELR}-ins-${TIME}.error 
      echo "......end install ${TARGET} with compiler ${COMPIELR}" 
      popd 
      if [ "$TARGET" == "qdp" ]; then 
        pushd ${SCRIPT}/qdp/examples 
        make clean 
        if [ ! -d error ]; then 
          mkdir error 
        fi 
        if [ ! -d ${SCRIPT}/result ]; then 
          mkdir ${SCRIPT}/result 
        fi 
        rm -rf ./error/*.* 
        make t_dslashm 2> ./error/${COMPILER}-${TIME}.error 
        ./t_dslashm > ${SCRIPT}/result/QDP-${COMPILER}-${TIME}.result 
        popd 
      fi 
    echo "......end for ${TARGET}......" 
  done 
pushd ${SCRIPT}/cpp/tests 
make clean 
if [ ! -d error ]; then 
  mkdir error 
fi 
if [ ! -d ${SCRIPT}/result ]; then 
  mkdir ${SCRIPT}/result 
fi 
rm -rf ./error/*.* 
make t_dslash 2> ./error/${COMPILER}-${TIME}.error 
./t_dslash > ${SCRIPT}/result/QP-CPP-${COMPILER}-${TIME}.result 
make time_dslash 2> ./error/tim-${COMPILER}-${TIME}.error 
./time_dslash > ${SCRIPT}/result/tim-${COMPILER}-${TIME}.result 
popd 

done
Appendix D  Deubs for DSlash

1. Error of SSE
   Remove #include <xmmintrin.h> in cpp_dslash_scalar_64bit_c.h
2. Error of #define ALIGN __attribute__((aligned(16))) in XLC
   Need to change the position of ALIGN from the end to the middle:
   GaugeMatrix (*gauge_field)[4] ALIGN = (GaugeMatrix(*[4])a->u;
   -->
   GaugeMatrix ALIGN (*gauge_field)[4] = (GaugeMatrix(*[4])a->u;
   Line 82, Line 229 in cpp_dslash_scalar_64bit.cc
   Line 76, Line 148 in cpp_dslash_3d_scalar_64bit.cc
   Line 120, Line 396 in cpp_clover_scalar_64bit.cc
3. Error about ShiftTable
   Delete class CPlusPlusWilsonDslash::ShiftTable; in cpp_clover_scalar.h and add the
defination of class ShiftTable in file cpp_dslash_types.h scalar _workstation builds
## Appendix E  Portability Results

<table>
<thead>
<tr>
<th></th>
<th>LibXML2</th>
<th>QDP_T</th>
<th>CPP_D</th>
<th>QDP_D_T(C/E)</th>
<th>CPP_D_T(C/E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P/P</td>
<td>P/F</td>
</tr>
<tr>
<td>XLC</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>F/F</td>
<td>F/F</td>
</tr>
<tr>
<td>CBE</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

**Threaded QDP and CPP_WilsonDSlash without openmp**

<table>
<thead>
<tr>
<th></th>
<th>LibXML2</th>
<th>QDP</th>
<th>CPP_D</th>
<th>QDP_D_T(C/E)</th>
<th>CPP_D_T(C/E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P/P</td>
<td>P/F</td>
</tr>
<tr>
<td>XLC</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P/P</td>
<td>P/F</td>
</tr>
<tr>
<td>CBE</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F/F</td>
</tr>
</tbody>
</table>

**QDP and CPP_WilsonDSlash without threading**
Appendix F  Xlc on HPCx

xlc_r array_add.c -q64 -O2 -qarch=pwr4 -qtune=pwr4 -qsmp=noauto -o array_add and attached
is the run script and output ...

```
#@ shell = /bin/ksh
#
#@ job_name = my_openmp_run
#
#@ job_type = parallel
#@ cpus = 1
#@ node_usage = not_shared
#
#@ wall_clock_limit = 00:20:00
#@ account_no = z001
#
#@ output = $(job_name).$(schedd_host).$(jobid).out
#@ error = $(job_name).$(schedd_host).$(jobid).err
#@ notification = never
#
#@ queue
#
export XLSPMOPTS=spins=0:yields=0
export OMP_NUM_THREADS=16
./my_omp_executable
```
References


[20] www.usqcd.org


[23] Wang Chen, IBM private communication