Development of lattice QCD simulation code set “Bridge++” on accelerators

S. Motoki\textsuperscript{1}, S. Aoki\textsuperscript{2}, T. Aoyama\textsuperscript{3}, K. Kanaya\textsuperscript{4}, H. Matsufuru\textsuperscript{6}, Y. Namekawa\textsuperscript{5}, H. Nemura\textsuperscript{5}, Y. Taniguchi\textsuperscript{4}, S. Ueda\textsuperscript{7}, and N. Ukita\textsuperscript{5}

\textsuperscript{1} Graduate School of Computer Science and Engineering, University of Aizu, Japan. shinji-m@u-aizu.ac.jp
\textsuperscript{2} Yukawa Institute for Theoretical Physics, Kyoto University, Japan. saoki@yukawa.kyoto-u.ac.jp
\textsuperscript{3} Kobayashi-Maskawa Institute for the Origin of Particles and the Universe (KMI), Nagoya University, Japan. aoym@kmi.nagoya-u.ac.jp
\textsuperscript{4} Graduate School of Pure and Applied Sciences, University of Tsukuba, Japan. {kanaya, taniguchi}@ccs.tsukuba.ac.jp
\textsuperscript{5} Center for Computational Sciences, University of Tsukuba, Japan. {namekawa, nemura, ukita}@ccs.tsukuba.ac.jp
\textsuperscript{6} Computing Research Center, High Energy Accelerator Research Organization (KEK), Japan. hideo.matsufuru@kek.jp
\textsuperscript{7} Theory Center, IPNS, High Energy Accelerator Research Organization (KEK), Japan. sueda@post.kek.jp

Abstract

We are developing a new code set “Bridge++” for lattice QCD (Quantum Chromodynamics) simulations. It aims at an extensible, readable, and portable workbench, while achieving high performance. Bridge++ covers popular lattice actions and numerical algorithms. The code set is constructed in C++ with an object oriented programming. In this paper, we describe our code design focusing on the use of accelerators such as GPGPUs. For portability our implementation employs OpenCL to control the devices while encapsulates the details of manipulation by providing generalized interfaces. The code is successfully applied to several recent accelerators.

Keywords: Lattice QCD, HPC, GPGPU, OpenCL

1 Introduction

Numerical simulations have been playing an essential role in physics researches. Simulations occasionally provide the only way to explore the dynamics. Quantum Chromodynamics (QCD), a fundamental theory of the strong interaction between quarks [1], is a typical example. The coupling between quarks increases at a large distance, i.e. at a low energy. It deteriorates an
analytic study by the perturbation theory that is an expansion in the coupling. The essential features of QCD, such as confinement and spontaneous chiral symmetry breaking, can be explored only by numerical simulations.

The formulation of QCD for numerical simulations is called lattice QCD [2]. It is a field theory on a discretized space-time, namely on a lattice. The lattice QCD simulations have been developed in tandem with development of computers [3]. A recent lattice QCD simulation becomes tremendously realistic. It can be performed on the physical point with real quark masses. A lattice technique is also applied to theories beyond QCD.

Although lattice simulations are an indispensable tool for theoretical analysis, the programming technique has become more involved. A complicated tuning of the simulation code is required to make use of the full performance. In view of this situation, we started a project to develop a new common code set “Bridge++” for lattice QCD simulations [4, 5]. We employ C++ to make use of the object-oriented design. The code was released in July 2012. It is vigorously being developed to involve more functions, brush up its design and the implementation.

In this paper, we describe our attempt to incorporate arithmetic accelerators such as GPGPUs and Xeon Phi into Bridge++. We adopt OpenCL (Open Computing Language) as a framework to control accelerator devices. OpenCL enables us to write a generic code for multi-platform.

This paper is organized as follows. In the next section, we briefly introduce the lattice QCD simulations focusing on the linear equation solver. It consumes most of the simulation cost and thus the target to be offloaded to accelerator devices. Section 3 summarizes our policy and the current status of Bridge++ code. In Section 4, we describe the OpenCL implementation in Bridge++ to exploit accelerator devices. The last section is devoted to the summary and our future prospects.

2 Lattice QCD simulations

The fundamental degree of freedom in QCD is quarks and gluons. A quark is the finest component of matter. A gluon mediates the strong interaction between quarks. An analytic study of QCD is difficult due to the self-interaction among gluons. The coupling increases at a large distance. It leads to a breakdown of analytic approaches by the perturbation.

A very powerful tool for non-perturbative analysis of QCD was proposed by K.G. Wilson as a field theory on a lattice [6]. Using lattice QCD, Monte Carlo simulations can be applied. It has been extensively used to understand the QCD dynamics and for phenomenological calculations. In lattice QCD, the gluon field is represented by link variables, \( U_\mu(n) \in \text{SU}(3) \), where a vector \( n = (n_x, n_y, n_z, n_t) \) stands for a lattice site and \( \mu = x, y, z, t \) (Figure 1). In computer simulations
the size of a lattice is finite, \( n_\mu = 1, 2, \cdots, L_\mu \). The quark field is represented as a complex vector on a lattice site, which carries 3 components of color and 4 components of spinor. Applying the path integral quantization, an expectation value of observable \( O \) is given by a functional integral

\[
\langle O \rangle = \frac{1}{Z} \int D\bar{\psi}D\psi DU_\mu O(\bar{\psi}, \psi, U_\mu)e^{-S_{\text{lat}}} = \frac{1}{Z} \int DU_\mu O(U_\mu) \det(D)e^{-S_G},
\]

where \( Z \) is the partition function. \( S_{\text{lat}} = S_G + S_F \) is the lattice QCD action with the gluon part \( S_G \) and the quark part \( S_F = \bar{\psi}D\psi \) with fermion matrix \( D \) (see below). The quark field is anticommuting Grassmann field. It can be integrated out by hand so that one arrives at the second equality of Eq. (1). The quark propagator \( D^{-1} \) expresses a propagation from a site to another site. Hadron spectrum can be extracted from the hadron correlation functions that are composed of quark propagators. The expectation value is calculated by generating ensemble of gluon fields with Monte Carlo method under the probability,

\[
P(U) \propto \det(D[U])e^{-S_G[U]},
\]

where \( U \) represents a gauge configuration \( \{U_\mu(x)\} \). There is a standard algorithm called Hybrid Monte Carlo algorithm to generate an ensemble with Eq. (2) that is composed of molecular dynamics for gluon field followed by a Metropolis test to remove the finite step size error. At every step of the molecular dynamics, one needs to solve a linear equation \( Dx = b \) to evaluate the contribution of \( \det(D) \). It costs most of the simulation time. Once the ensemble of independent configurations \( U^{(i)}(i = 1, \ldots, N) \) are in hand, an expectation value of the physical observable is obtained by

\[
\langle O \rangle = \frac{1}{N} \sum_i O[U^{(i)}].
\]

The quark propagator is obtained by an inversion of the quark operator. Here, we introduce the Wilson quark as an example. The Wilson quark action \( S_F = \bar{\psi}D\psi \) is constructed with a Wilson fermion operator \( D_W \) that connects a site \( m \) to \( n \),

\[
D_W(m, n) = \delta_{m,n} - \kappa \sum_{\mu=1}^4 \left[ (1 - \gamma_\mu)U_\mu(m)\delta_{m+\hat{\mu},n} + (1 + \gamma_\mu)U_\mu^\dagger(m - \hat{\mu})\delta_{m-\hat{\mu},n} \right],
\]

where \( \gamma_\mu \) is a 4 x 4 matrix acting on the spinor space, \( \hat{\mu} \) the unit vector in \( \mu \)-direction, and \( \kappa \) is a parameter related to the quark mass \( m_q \) through \( \kappa = 1/2(4 + m_q) \). The matrix \( D_W \) contains only the coupling with the neighboring sites, and thus large sparse complex matrix of the degree \( 3 \cdot 4 \cdot L_x \cdot L_y \cdot L_z \cdot L_t \). There is a variety of quark actions reflecting how much the symmetries in the continuum theory is respected as well as how improved so as to approach the continuum limit rapidly. Examples of fermion actions will appear in Table 1 in the next section.

3 Lattice QCD code Bridge++

The development of Bridge++ code set was started in 2009 [5]. Although there were several public lattice QCD code sets, to catch up with the rapid progress of computer architectures and to correspond to variety of applications, we decided to develop a new general-purpose code set that has the followings features:
• Readability: the structure is transparent so as to be understandable even for beginners.
• Extensibility: the code is easy to be modified for testing new ideas.
• Portability: the code runs not only on a laptop PC but also on supercomputers.
• High-performance: the code has high performance enough for productive researches.

To achieve these goals, we adopt an object oriented design using C++ programming language. The machine dependent part of the code is hidden as much as possible. The first version of the code was released in July 2012. It is actively developed to extend functions, refine the implementation, and improve the performance. The code as well as useful documents are available at our website [4].

3.1 Object-oriented programming

Object-oriented programming (OOP) is a guiding principle of Bridge++. Basic ingredients of the program are “objects” which are defined as sets of data fields and methods. A functionality is provided by an interaction between objects. The data field is a characteristic variable of the object, called a member data or a member variable. The method is a function that defines a behavior of the object. OOP is characterized by the following properties: (a) Encapsulation: data is handled through the interface, which defines how to use the object, (b) Inheritance: object is expandable by adding new functions, and (c) Polymorphism: objects with the same kind of behavior can be handled through the same interface. These are mechanisms to maximize reusability of the code. Using OOP, an interface is separated from details of the implementation. It localizes a machine specific part of the code, which is inevitable due to the optimization. The polymorphism allows us to implement an algorithm with a set of objects that have a common interface.

There is a compilation of wisdom to make use of these virtues of OOP. An example is so-called design patterns [7, 8]. The design pattern is a kind of programming idiom that frequently appears as a good solution to certain kind of problems. Their efficiency has been realized after the GoF’s publication [7] that classified such idioms into 23 design patterns. The design pattern enables us to use the benefit of OOP easily, as well as to make our code transparent.

In the following, we introduce the Bridge pattern as an example applied to the solver algorithms and the fermion matrices. There are numbers of fermion matrices and solver algorithms. The best solver algorithm depends on a type of the fermion matrix. The solver algorithm should be implemented so as to be applied to any kind of fermion operators, and simultaneously it must be changed easily. This is a typical situation that the Bridge pattern is applied. The Bridge pattern is one of GoF’s design patterns that decouples an interface from its implementation. They can be switched independently.

Figure 2 shows the class diagram of the Bridge pattern based on UML (Unified Modeling Language). The interface of the fermion operator is \texttt{Fopr.mult()} is a function that applies the fermion operator to a given vector and returns the resultant vector. The practical implementation of \texttt{mult()} is given in a subclass of \texttt{Fopr}, such as \texttt{Fopr.Wilson} for the Wilson fermion and \texttt{Fopr.Overlap} for the overlap fermion (another type of fermion matrix). Similarly, the base class \texttt{Solver} defines the interface of the solver algorithms. \texttt{solve()} function returns the solution of a linear equation for a given source vector. Again the implementation is given in subclasses, \texttt{Solver.CG} and \texttt{Solver.BiCGStab}. Bridge++ users can select any combination, such as \texttt{Fopr.Wilson} with \texttt{Solver.CG}. In practice, one sometimes requires nested calls of the linear solvers and fermion matrices, for example, for multi-precision preconditioning that is applied on accelerators. Abstraction with OOP is helpful to implement the code in a unified manner.
3.2 Features of Bridge++

Parallelization For a current computational environment, parallelization is inevitable. Large-scale computers consist of many nodes, which communicate with each other via high speed network. In Bridge++, we adopt MPI (Message Passing Interface) for distributed memory parallelization, i.e. for parallel nodes. Inter-node communication, such as broadcast and one-to-one data transfer, is performed through the functions of the Communicator class, which wraps API functions of MPI. If a machine specific efficient library is available, the implementation of Communicator class can be switched with it. For an environment without MPI, so-called stub implementation of Communicator class is provided. Bridge++ works on a single node as well as parallel nodes.

Multi-threading The shared memory parallelization of Bridge++ is now under development. Hybrid parallelization with MPI and multi-threading has been considered. Two multi-thread libraries, Pthread and OpenMP, are compared based on a working implementation of the Wilson fermion operator. While Pthread can accomplish better performance, it is not easy to apply it to the entire code set. OpenMP has been selected as a primary method to multi-threading.

I/O format In Bridge++, simulation parameters are given by ASCII files in YAML format. The parameters are held in Parameter object and passed to each object. Extraction of values from files is handled by a parameter manager class. For the I/O of field objects, several formats are available including ILDG standard format for the gauge configuration. ILDG (International Lattice Data Grid) is an activity to promote sharing configuration data and standardizing data format and description of metadata [9]. The standard output is classified with a verbose level. At present 4 levels are prepared. The output level is selected for each object.

3.3 Current status

Our development status of Bridge++ is summarized in Table 1. The current public version contains major algorithms and observables, though the fermion actions are limited to Wilson and clover (improved Wilson) fermions. Several other fermion actions have been implemented and now being tested. These fermion actions can be combined with an improvement called a link smearing. For linear algorithms, iterative algorithms, such as CG, BiCGStab and GMRES(m), are available. Multi-shift solver with CG algorithm and eigensolver with Implicitly restarted Lanczos algorithm are ready. A variety of popular algorithms to generate configurations have also been implemented.
<table>
<thead>
<tr>
<th>Gauge action</th>
<th>plaquette, rectangle loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fermion action</td>
<td>Wilson, clover, twisted mass*, staggered*, domain-wall*, overlap*</td>
</tr>
<tr>
<td>Smearing</td>
<td>clover w/ isospin chemical potential</td>
</tr>
<tr>
<td>Schrödinger functional</td>
<td>(APE, HYP) × (maximal projection, stout)</td>
</tr>
<tr>
<td>Linear algebra</td>
<td>plaquette/rectangle, Wilson/clover</td>
</tr>
<tr>
<td>Eigen solver</td>
<td>CG, BiCGStab, GMRES(m)</td>
</tr>
<tr>
<td>Shift solver</td>
<td>implicitly restarted Lanczos</td>
</tr>
<tr>
<td>Configuration generation</td>
<td>multi-shift CG</td>
</tr>
<tr>
<td>Physical observable</td>
<td>HMC, RHMC, multi-time step integrator w/ leapfrog and Omelyan</td>
</tr>
</tbody>
</table>

Table 1: Our development status. * mark means the function that has already been implemented and being confirmed. Detailed explanation is in our website [4].

The performance tuning of Bridge++ is in progress. On Hitachi SR16000, the rate to the peak performance is about 5% on 1 node (32 cores). On Blue Gene/Q, the public version runs with 2-3% on 32-nodes. The latest code under development gives 10% for the most time consuming solver part.

4 Arithmetic accelerators in Bridge++

4.1 Programming frameworks for accelerators

Arithmetic accelerators, such as GPUs and Xeon Phi, are rapidly increasing their performance. The TOP500 ranking clearly reflects this trend. A prominent feature of the arithmetic accelerator is that one can acquire a large numerical power with less cost. Their high electric power efficiency is also attracting. In Green500 ranking, the top ten ranks are occupied by machines containing accelerators.

Lattice QCD simulations have exploited arithmetic accelerators. After pioneering works [10, 11], many attempts have been made to obtain the full performance. It includes a development of a machine designed for lattice simulations, QPACE [12], which is composed of Cell B.E.s. For NVIDIA GPUs, QUDA library for lattice QCD has become a popular tool [13, 14, 15]. Many results, including a development of algorithms suitable to accelerators, have been reported in the yearly symposium on Lattice Field Theory, [16, 17].

Thanks to various SDKs, it is not difficult to develop a code of lattice QCD simulations for a specific architecture of the accelerators. However, it is still challenging to integrate a framework to use accelerators into a code set developed in aiming at wide purpose of application, without specifying the architecture of accelerator. We investigate how the devices of accelerators are abstracted and integrated as a generic framework in a code set.

Although there are many kinds of framework to use accelerators, it is nontrivial to keep portability for various kinds of architecture. A general framework to use accelerators is offloading the operation and data from CPU to accelerators using programming framework. CUDA SDK is provided by NVIDIA. Directive (pragma) based frameworks are also developed so as to implicitly offload the program, such as OpenACC and HMPP. One of our goals is to seek for the programming techniques independent of environment and encapsulate the implementation specific to architecture. As the first implementation to make use of the accelerators, we adopt the OpenCL framework. It possesses portability to various environments, while a detailed handling of accelerator devices is possible by rich APIs and data types. In future, we plan to implement
several frameworks their interfaces that enable us to change these frameworks appropriately to an architecture and its programming environment.

4.2 OpenCL framework

OpenCL (Open Computing Language) is a standard framework for a parallel programming[18]. The specifications are determined by Khronos Group with contributions by AMD, NVIDIA, and so on. An advantage of OpenCL is its portability to various architecture, CPUs, GPUs and other computing devices such as Xeon Phi, FPGAs and Cell/B.E. OpenCL offers an abstract hardware layer. It allows programmers to develop applications without knowing details of underlying processor architectures. (See also Figure 3).

Here we briefly summarize the general scheme of programming in OpenCL. An OpenCL application is composed of a host program and a device program working on single or multiple computing devices. Such an environment is called “platform”. The host device, usually CPUs, controls a memory access and operation on GPUs and CPUs. When the platform is homogeneous as multi-cored processors, one of the cores plays the role of host device and controls the other cores, to which OpenCL can apply.

The most prominent feature of OpenCL is that the specifications are standardized independently from the architecture of processors. Using such a language and APIs, one can develop a parallelized applications that works on various chips and operation systems. This is indispensable for portability that we regard as one of the most important feature of Bridge++. It has another advantage that one needs to learn just one programming scheme. Although the performance might be a trade-off, OpenCL has rich APIs that enables code tuning specific to the target architecture. One can start with a generic code and incrementally tune it to the adopted environment.
Figure 4: Schematic structure of Bridge++ to handle the accelerator.

4.3 Implementation in Bridge++ with OpenCL

Programs that off-loads operation and data to devices have at least the following procedures.

1. Generating device memory object
2. Transferring data to device memory
3. Executing device program
4. Collecting data from device memory
5. Deleting device memory object

If each class carries out these procedures, overheads may be large due to construction and destruction of objects. Thus we develop a class that manages memory objects on the devices and transferring data between the host processor and the devices. This DeviceManager class is implemented in a Singleton pattern (to ensure the instance generated only once) and wraps OpenCL APIs so as to simplify the procedures to use the devices. In addition, resources such as device programs are also managed in unified manner. Recursive calls of the program is easily performed. An image of our approach is summarized in Figure 4.

The DeviceManager class consists of two characteristic elements. One is the mechanism to handle the device memory objects by wrapping the APIs of OpenCL. The memory object is managed with uniquely associated ID using the ‘map’ template class in the C++ Standard Template Library (STL). The second ingredient is methods (member functions) that handle the device program from the host program through generalized interfaces. Example of an interaction between host and device programs is schematically displayed in Figure 5. These implementations aim at keeping portability by avoiding a direct use of the objects, APIs, and
grammars specific to a framework in the main part of the code. This provides an example of the Adapter pattern, one of the GoF’s design patterns.

4.4 Performance of accelerator program

We measure the performance of our code on NVIDIA GPU, AMD GPU, and Intel Xeon Phi (the specifications are summarized in Table 2) without any modifications to each architecture. The performance is measured for multiplications of Wilson fermion operator on $16^3 \times 32$ lattice. The whole lattice is put on a single accelerator device. The sustained performance values are AMD Radeon HD 7970: 22GFlops(0.5%), NVIDIA GeForce GTX Titan: 16GFlops(0.3%), and Intel Xeon Phi 5110P: 25GFlops(1%).

The current performance is not sufficient in the original version of the code that has not yet optimized to the accelerator architectures. To obtain good performance on these devices, it is required to elaborate the data layout so that the load and store of the data to the accelerator cores is appropriately performed. In this regard, we have to apply modifications of data structure specific to the architecture of each accelerator device. Mechanisms to absorb the variety of such modifications have been prepared in our framework. Tuning on the aforementioned architectures is now underway.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Radeon HD 7970</td>
<td>AMD</td>
<td>Southern Islands</td>
<td>925</td>
<td>947</td>
<td>3789</td>
<td>3</td>
<td>254</td>
</tr>
<tr>
<td>GeForce GTX Titan</td>
<td>NVIDIA</td>
<td>Kepler</td>
<td>876</td>
<td>1570</td>
<td>4709</td>
<td>6</td>
<td>258</td>
</tr>
<tr>
<td>Xeon Phi 5110P</td>
<td>Intel</td>
<td>MIC</td>
<td>1053</td>
<td>1011</td>
<td>2022</td>
<td>8</td>
<td>320</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GFlops</th>
<th>Performance (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>0.5</td>
</tr>
<tr>
<td>16</td>
<td>0.3</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Accelerator specifications and results of performance.

5 Summary

In this paper, we described the design of our code set for lattice QCD simulations ‘Bridge++’. We adopt C++ to utilize the object-oriented design. Bridge++ handles accelerator devices by use of OpenCL. OpenCL framework is valuable to keep the portability of Bridge++. The main program controls devices through a generalized interface. The APIs are wrapped into a device manager class. We demonstrated that our code successfully runs on several accelerators while for a practical application the performance tuning respecting each architecture is necessary.

Acknowledgment

In addition to the authors of this paper, this project has been contributed by many of our colleagues, as listed in a document enclosed in the source code. This project is supported
by Joint Institute for Computational Fundamental Science and HPCI Strategic Program Field
5 ‘The origin of matter and the universe’. The code was developed and tested on Hitachi
SR16000 and IBM System Blue Gene/Q at KEK under a support of its Large-scale Simulation
Program (Nos.12/13-15, 13/14-19), Hitachi SR16000 at YITP in Kyoto University, K-computer
at RIKEN Advanced Institute for Computational Science, HA-PACS at University of Tsukuba
under a support for its Interdisciplinary Computational Science Program (No.13a-25) and FX10
at University of Tokyo. This work is supported in part by the Grand-in-Aid for Scientific
Research of the Japan (Nos.20105005, 24540250, 25400284).

References

[2] There are many textbooks of lattice QCD simulations, for example, I. Montvay and G. Münster,
Quantum Fields on a Lattice (Cambridge Univ. Press, 1994); T. DeGrand and C. DeTar, Lattice
[3] A. Nakamura, Lattice QCD Simulations as an HPC Challenge., in High-Performance Computing,
[5] S. Ueda et al., in proceedings of the 15th International Workshop on advanced computing and anal-
ysis techniques in physics, May 16-21, 2013, Beijing, China; proceedings of the 31st International
Symposium on Lattice Field Theory, 29 July - 3 August 2013, Johannes Gutenberg University
Mainz, Germany.
[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-
Oriented Software. (Addison-Wesley, 1995).
[8] A. Trott and J. R. Shalloway, Design Patterns Explained: A New Perspective on Object-Oriented
Design, 2nd ed. (Addison-Wesley Professional, 2004)
for High Performance Computing, Networking, Storage and Analysis (SC), 2011 [arXiv:1109.2935
[hep-lat]].
[18] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous sys-
tems,[Online]. http://www.khronos.org/opencl