PROCEEDINGS OF SCIENCE

Lattice QCD code Bridge++ on arithmetic accelerators

S. Motoki[*a], **S. Aoki**[b], **T. Aoyama**[c], **K. Kanaya**[d,e], **H. Matsufuru**[a], **T. Miyamoto**[b], **Y. Namekawa**[f], **H. Nemura**[f], **Y. Taniguchi**[d], **S. Ueda**[g], **and N. Ukita**[f]

[a] *Computing Research Center, High Energy Accelerator Research Organization (KEK), Tsukuba 305-0801, Japan*

[b] *Yukawa Institute for Theoretical Physics, Kyoto University, Kyoto 606-8502, Japan*

[c] *Kobayashi-Maskawa Institute for the Origin of Particles and the Universe (KMI), Nagoya University, Nagoya 464-8602, Japan*

[d] *Faculty of Pure and Applied Sciences, University of Tsukuba, Tsukuba 305-8571, Japan*

[e] *Center for Integrated Research in Fundamental Science and Engineering (CiRfSE), University of Tsukuba, Tsukuba 305-8571, Japan*

[f] *Center for Computational Sciences, University of Tsukuba, Tsukuba 305-8577, Japan*

[g] *Theory Center, IPNS, High Energy Accelerator Research Organization (KEK), Tsukuba 305-0810, Japan*

E-mail: smotoki@post.kek.jp

OpenCL and OpenACC are generic frameworks for heterogeneous programming combining CPU and accelerator devices such as GPUs. We test incorporation of these two frameworks into the Bridge++, a general-purpose code set for numerical simulations of lattice QCD, to offload the time-consuming fermion matrix inversion to accelerator devices. We discuss advantages and disadvantages of these frameworks from the viewpoints of constructing reusable components based on the object-oriented programming and of tuning the code to achieve a high performance.

The 33rd International Symposium on Lattice Field Theory
14 -18 July 2015
Kobe International Conference Center, Kobe, Japan

[*]Speaker.

PoS(LATTICE 2015)040

1. Introduction

Recent accelerator devices such as GPUs and Xeon Phi enable us to construct parallel cluster computers of large computational power with relatively low cost. Accelerator devices have been adopted in lattice simulations [1, 2]. However, a big programming effort is required to offload heavy tasks of hot spots to accelerator devices in recent QCD codes with hybrid parallelization for multi-node and multi-thread machines. To achieve a high performance, we also need to optimize the code according to the architecture of each accelerator device. On the other hand, the large variety of combinations of different physics projects and hardware architectures as well as numerical algorithms forced the lattice community to develop general-purpose code sets for QCD simulations. It is thus important to implement accelerator devices into a general-purpose code set.

To keep applicability to a wide range of physics problems and numerical algorithms, the components of the code specific to each hardware should be separated and encapsulated into reusable ingredients with the least interference. A guideline to develop such a code set is the object-oriented programming (OOP) [3, 4, 5]. In 2009, we have started a project to develop a general-purpose code set for lattice QCD simulations that is widely applicable while based on uniform design policy. The code set, named Bridge++, is written in C++ and developed so as to achieve readability, extensibility, and portability, while simultaneously keeping sufficient performance for productive research. The project also aims at compiling the latest knowledge of modern programming techniques.

In this study, we incorporate accelerator devices into Bridge++. We adopt generic programming frameworks for general accelerator devices. Several open source libraries have been developed to use some specific accelerator. An example is the QUDA [6] for NVIDIA GPUs, which is a CUDA-based library for lattice QCD. Here, we want to develop a code for general accelerator devices by establishing techniques to fully make use of their performance. Among available frameworks, OpenCL and OpenACC are attractive because of their portability. Application of OpenCL [7, 8, 9, 10, 11] and OpenACC [12, 11] to lattice QCD have been started. We note these two frameworks have contrasting features: OpenCL uses explicit API-based controls of the device, while OpenACC is directive-based and a compiler generates procedures to use the device. In this paper, we apply OpenCL and OpenACC in Bridge++ to offload a linear equation solver to accelerator devices. From a point of view of constructing reusable components based on OOP and performance, we evaluate feasibility of these two frameworks in Bridge++ through practical implementation.

2. Implementation

2.1 Strategy to use accelerator devices

To introduce accelerator devices into a simulation code, the following steps are necessary to be executed.

(1) Get information of accelerators and setup environment.

(2) Setup kernel code.

(3) Allocate memory space for data on device.

(4) Transfer data from host to device.

(5) Execute kernel code.

(6) Transfer data from device to host.

(7) Free the memory space on device.

In QCD simulations, we often encounter bottlenecks in the steps (4) and (6), because of a limited bandwidth between host and device. This is a severe problem in the fermion matrix inversion. To minimize the data transfer in these steps, preconditioning using a single precision solver [6] or domain-decomposition [13] are frequently employed. Since our current public version of Bridge++ code assumes double precision for the gauge and fermion fields, we first generalize the field container class to adopt any type of the data. As a counter part of a class `Field`, which represents the field data, we introduce new template classes `AField<REALTYPE>` on the host and `AField_dev<REALTYPE>` on the accelerator device for a data type `REALTYPE`. At the construction of an `AField_dev` instance, a memory space on the device is allocated. The linear algebraic operations as well as the fermion operators corresponding to the above step (5) are also prepared with the same interfaces as on the host. The solver algorithms are implemented with the C++ template for both host and device field classes.

We also apply so-called coalesced memory access by changing the data layout on the device from that on the host. To reduce memory transfer, the third column of SU(3) matrix in gauge field is not transferred from the global memory but calculated on-the-fly using the relation $v_3 = (v_1 \times v_2)^*$ where $U = (v_1, v_2, v_3) \in SU(3)$.

2.2 Implementation with OpenCL

OpenCL (Open Computing Language) is an open standard framework for a parallel programming in heterogeneous platforms. By specification, OpenCL is composed of the run-time APIs and the OpenCL C language. The APIs control devices from the host programs. The OpenCL C is for the device codes. On devices, threads run in parallel executing the same program. A thread is called *work-item*, and a specified number of work-items are grouped to form a *work-group*. The memories on the device are classified into four types: (i) Global memory: readable and writable from all the work-items and from the host, (ii) Constant memory: read-only from all work-items and readable and writable from the host, (iii) Local memory: shared by work-items within a work-group, and (iv) Private memory: exclusively used by a work-item. The total number of work-items and the size of work-group are tunable parameters at run-time.

Each step of the work flow in Sec. 2.1 is handled through OpenCL APIs. At the initialization step (1), one needs to obtain the information of platforms and setup *contexts* and *command queues* at the beginning of a program. If a run-time compiler is used, as we adopt, a kernel code is compiled at the step (2). We encapsulate OpenCL APIs for these steps in a `DeviceManager` class, which simplifies the procedures and switches the frameworks easily. This `DeviceManager` class also wraps managements of device memory objects and data transfer between host and device. Each object accesses to the device memory through the interfaces provided by the `DeviceManager` class. The device code, described in OpenCL C language, is embedded as a string object at the compilation of Bridge++, and then on-line compiled at the run-time. Using this mechanism, several parameters and functions described as macros are replaced at the run-time. Changing the data layout is realized by replacing the macro definition of an index, without modifying the kernel code. At construction of an instance of `AField_dev<REALTYPE>` class, the associated memory space

on the device is allocated through the `DeviceManager` class (the step (4) of the workflow), and is released at the destruction (the step (7)). This class also contains methods to transfer data between host and device (the steps (4) and (6)).

Using this `DeviceManager` class, as well as the fermion operators implemented similarly, the solver algorithms can be constructed commonly to those on the host processor. The linear-algebraic operations (for the step (5)) are prepared as methods. The kernel codes used in these methods are compiled and cached through the `DeviceManager`, when the first instance is constructed.

2.3 Implementation with OpenACC

OpenACC is a directive-based extension of programming languages. The specification is defined for C/C++ and Fortran. A compiler analyzes OpenACC directives, and generates procedures for offloading data and tasks to accelerators. OpenACC assumes three levels in the processor: *gang*, *worker*, and *vector*. For example, in the case of NVIDIA Tesla architecture, they correspond to the streaming multi-processor, warp, and thread.

The following three kinds of directives are crucial.

Specification of parallel region Two directives, `kernels` and `parallel`, are defined to specify which part of the code is to be executed in parallel. `kernels` directive entrusts a compiler to analyze dependencies of variables. On the other hand, `parallel` directive does to users. We use the latter in our implementation.

Memory allocation and data transfer `data` directive is a representative example. From OpenACC 2.0, `enter` and `exit` directives are added, which allocate and free a memory space on the device. Data transfer between host and device is executed by `update` directive. We make use of these OpenACC 2.0 directives, so that before the parallel region the clause of `data` directive is always `present`.

Specification of parallelized loop `loop` directive specifies the `parallel` region. With a clause, one can identify which of gang, worker, and vector is assigned to the loop, and variables that are private to the loop. Collapsing loops and specification of reduction can also be indicated with clauses.

Since OpenACC libraries control devices implicitly by insertion of directives, we introduce no new classes corresponding to the `DeviceManager` class for the case of OpenCL. Instead, we simply add OpenACC directives to our code. The steps (1) and (2) of the work flow in Subsection 2.1 are automatically incorporated by the compiler. As in the case of OpenCL, we define `AField_dev<REALTYPE>` class that represents field data on the device. At construction of an instance of this class, the constructor allocates a memory space on the device by `enter` directive (the step (3)). This device memory space is freed in the destructor by `exit` directive (the step(7)). To transfer data between the host and device, member functions are defined using the `update` directive (the steps (4) and (6)). This implementation enables explicit control of memory allocation and data transfer through an abstract interface. The kernel code to execute the step (5) is generated by a compiler at the `parallel` directive.

	Tesla K40 (Kepler)	Tesla M2090 (Fermi)	FirePro W9100
Performance(float)	4290 GFlops	1330 GFlops	5240 GFlops
Performance(double)	1430 GFlops	665 GFlops	2620 GFlops
Memory bandwidth	288 GB/s	177 GB/s	320 GB/s
Number of cores	2880	512	2816
Software	CUDA 5.5/PGI 14.10	CUDA 5.5/PGI 14.10	SDK v.2.9/PGI 14.10

Table 1: Hardware and software environment.

In the `AField_dev<REALTYPE>` class, linear algebraic methods and fermion matrix multiplications are implemented in this manner. By replacing corresponding objects in the OpenCL version with them, the solver algorithms work without any modification.

3. Performance

We report the sustained performance obtained by Bridge++ using OpenCL and OpenACC. Table 1 summarizes our test environment. Two types of accelerators are tested, NVIDIA and AMD. OpenCL is managed by CUDA and AMD APP SDK, and OpenACC by PGI compiler.

As a representative example, we study the performance of a multiplication of the Wilson operator (represented as "mult"), and a conjugate gradient (CG) solver for Wilson fermions on a single device. Our lattice size is $16^3 \times 32$.

Performance with OpenCL In Table 2, we summarize the performance of our code on NVIDIA and AMD GPUs [1]. This is an update of the result reported in Ref. [10, 11]. The run-time parameters for thread grouping are adjusted for each device. Except for the NVIDIA Tesla M2090 case, the results of the single precision are almost twice the double precision, indicating the performance is determined by the data transfer between the device memory and cores. While the performance of the Wilson matrix multiplication is comparable for NVIDIA Tesla K40 and AMD GPU, the performance of the solver shows amplified differences. This is presumably caused by inefficient implementation of the reduction in the inner-product. At present, the BLAS methods are implemented in the Bridge++ code. We apply two step reduction, first reducing to a coarse-grained array and then taking a full reduction. These reductions are found to affect the performance significantly. In the case of Tesla M2090 for double precision, in addition to adjusting the run-time parameters, we slightly modify the code so as to use a local memory instead of registers for a part of variables. This improves the performance of the Wilson matrix multiplication about 70 %, while still much less than half the number for single precision. It may imply inefficient register assignment in the original code for this architecture. Further tuning of the code and architecture dependent optimization are underway.

Performance with OpenACC In Table 2, we also quote the present performance of OpenACC implementation of Bridge++. The fermion multiplication is less efficient than the OpenCL version,

[1] In Ref. [5] we quoted incorrect performance data for Tesla K40, caused by that the position of time counter was changed by compiler optimization.

	OpenCL		OpenACC	
operation	float	double	float	double
NVIDIA Tesla K40 (Kepler):				
Wilson mult	232 GFlops	121 GFlops	196 GFlops	39.1 GFlops
CG solver	160 GFlops	86.0 GFlops	123.7 GFlops	35.0 GFlops
NVIDIA Tesla M2090 (Fermi):				
Wilson mult	154 GFlops	39.2 GFlops	149 GFlops	22.9 GFlops
CG solver	107 GFlops	33.7 GFlops	84.3 GFlops	21.1 GFlops
AMD FirePro W9100:				
Wilson mult	179 GFlops	110 GFlops	123 GFlops	19.0 GFlops
CG solver	120 GFlops	79.5 GFlops	130 GFlops	20.6 GFlops

Table 2: Performance with OpenCL and OpenACC for Wilson fermion matrix mult and CG solver on a $16^3 \times 32$ lattice.

in particular for the double precision case. The latter may be due to non-optimal assignment of variables to registers. By reducing the number of local variables, the performance indeed approaches the half the values of the float cases. To achieve a higher performance, a more careful tuning seems to be in order.

4. Discussion and conclusion

We reported our implementation of OpenCL and OpenACC into Bridge++, and discussed advantages and disadvantages of these frameworks.

We found that OpenCL enables us to control devices in detail, but requires complicated setup procedures, such as preparation of contexts, command queues, as well as a compilation of kernel codes. These procedures can be, however, encapsulated in a management class. While C++ template programming is not available in the current OpenCL C language, on-line compiler can achieve polymorphism. In tuning the code, it is also convenient that the memory type of a variable can be specified explicitly.

In contrast to the case of OpenCL, OpenACC is much easier to be introduced. However, we had to spend much more time to tune the code with OpenACC, because of its indirect control of devices by a compiler. The rapid improvement of the OpenACC compiler may dissolve this problem. We also note that the present OpenACC compiler is not sufficiently mature in processing the C++ template syntax, which required us involved coding to enable static polymorphism.

Both frameworks allow us to offload time-consuming computational tasks, keeping the object-oriented code structure. The interfaces implemented in Bridge++ are sufficiently simple. Although the performance is better with OpenCL at present, that with OpenACC in single precision is also acceptable. For the double precision case with OpenACC, more tuning is required. Further tuning is ongoing, taking account of each device architecture.

Acknowledgment

The code was developed and tested on HA-PACS at the University of Tsukuba under a sup-

port for its Interdisciplinary Computational Science Program, and workstations installed at KEK Computing Research Center. This project is supported by H20 Grant-in-Aid for Scientific Research on Innovative Areas 'Research on the Emergence of Hierarchical Structure of Matter by Bridging Particle, Nuclear and Astrophysics in Computational Science', Joint Institute for Computational Fundamental Science and HPCI Strategic Program Field 5 'The origin of matter and the universe'. This work is supported in part by the Grand-in-Aid for Scientific Research of the Japan (Nos.20105005, 25400284, 15K05041, 15K05068).

References

[1] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi and K. K. Szabo, Comput. Phys. Commun. **177**, 631 (2007) doi:10.1016/j.cpc.2007.06.005 [hep-lat/0611022].

[2] M. A. Clark, PoS LAT **2009**, 003 (2009) [arXiv:0912.2268 [hep-lat]].

[3] Bridge++ website, `http://bridge.kek.jp/Lattice-code/`.

[4] S. Ueda *et al.*, PoS LATTICE **2013**, 412 (2014); J. Phys. Conf. Ser. **523**, 012046 (2014).

[5] S. Ueda *et al.*, PoS LATTICE **2014**, 036 (2015).

[6] M. A. Clark, R. Babich, K. Barros, R. C. Brower and C. Rebbi, Comput. Phys. Commun. **181**, 1517 (2010) doi:10.1016/j.cpc.2010.05.002 [arXiv:0911.3191 [hep-lat]].

[7] M. Bach, V. Lindenstruth, O. Philipsen and C. Pinke, Comput. Phys. Commun. **184**, 2042 (2013) doi:10.1016/j.cpc.2013.03.020 [arXiv:1209.5942 [hep-lat]].

[8] V. Demchik and N. Kolomoyets, arXiv:1310.7087 [hep-lat].

[9] M. Di Dipierro, PoS LATTICE **2013**, 043 (2014).

[10] S. Motoki *et al.*, Procedia Computer Science 29, 1701-1710 (2014).

[11] H. Matsufuru *et al.* (Bridge++ Project), Proc. Comp. Sci. **51**, 1313 (2015) [doi:10.1016/j.procs.2015.05.316].

[12] P. Majumdar, PoS LATTICE **2013**, 031 (2014) [arXiv:1311.2719 [hep-lat]].

[13] Y. Osaki and K. I. Ishikawa, PoS LATTICE **2010**, 036 (2010) [arXiv:1011.3318 [hep-lat]].