XcalableACC: an Integration of XcalableMP and OpenACC

Akihiro Tabuchi and H. Murai, M. Nakao, T. Odajima, and T. Boku

Abstract XcalableACC (XACC) is an extension of XcalableMP for accelerated clusters. It is defined as a diagonal integration of XcalableMP and OpenACC, which is another directive-based language designed to program heterogeneous CPU/accelerator systems. XACC has features for handling distributed-memory parallelism, inherited from XMP, offloading tasks to accelerators, inherited from OpenACC, and two additional functions: data/work mapping among multiple accelerators and direct communication between accelerators.

Akihiro Tabuchi
Fujitsu Laboratories Ltd., 4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa 211-8588, Japan, e-mail: tabuchi.akihiro@fujitsu.com

Hitoshi Murai
RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan, e-mail: h-murai@riken.jp

Masahiro Nakao
RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan, e-mail: masahiro.nakao@riken.jp

Tetsuya Odajima
RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan, e-mail: tetsuya.odajima@riken.jp

Taisuke Boku
Center for Computationl Sciences, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan, e-mail: taisuke@ccs.tsukuba.ac.jp

Contents

1 Introduction

This document defines the specification of XcalableACC (XACC) which is an extension of XcalableMP version 1.3[1] and OpenACC version 2.5[2]. XcalableACC provides a parallel programming model for accelerated clusters which are distributed memory systems equipped with accelerators.

In this document, terminologies of XcalableMP and OpenACC are indicated by **bold font**. For details, refer to each specification[1, 2].

The works on XACC and the Omni XcalableACC compiler was supported by the Japan Science and Technology Agency, Core Research for Evolutional Science and Technology program entitled “Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era” in the research area of “Development of System Software Technologies for Post-Peta Scale High Performance Computing.”

1.1 Hardware Model

The target of XcalableACC is an accelerated cluster, a hardware model of which is shown in Fig. 1.

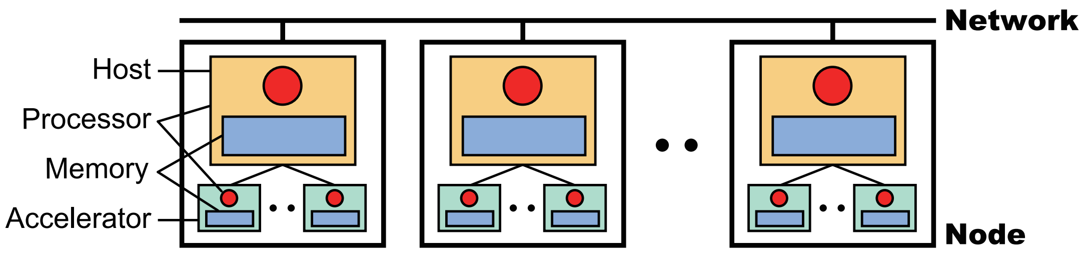

Fig. 1 Hardware Model

An execution unit is called **node** as with XcalableMP. Each **node** consists of a single host and multiple accelerators (such as GPUs and Intel MICs). Each host has a processor, which may have several cores, and own local memory. Each accelerator also has them. Each **node** is connected with each other via network. Each **node** can access its local memories directly and remote memories, that is, the memories of another **node** indirectly. In a host, the accelerator memory may be physically and/or virtually separate from the host memory as with the memory model of OpenACC. Thus, a host may not be able to read or write the accelerator memory directly.

1.2 Programming Model

XcalableACC is a directive-based language extension based on Fortran 90 and ISO C90 (ANSI C90). To develop applications on accelerated clusters with ease, XcalableACC extends XcalableACC and OpenACC independently as follow: (1) XcalableMP extensions are to facilitate cooperation between XcalableMP and OpenACC directives. (2) OpenACC extensions are to deal with multiple accelerators.

1.2.1 XcalableMP Extensions

In a program using the XcalableMP extensions, XcalableMP, OpenACC, and XcalableACC directives are used. Fig. 2 shows a concept of the XcalableMP extensions.

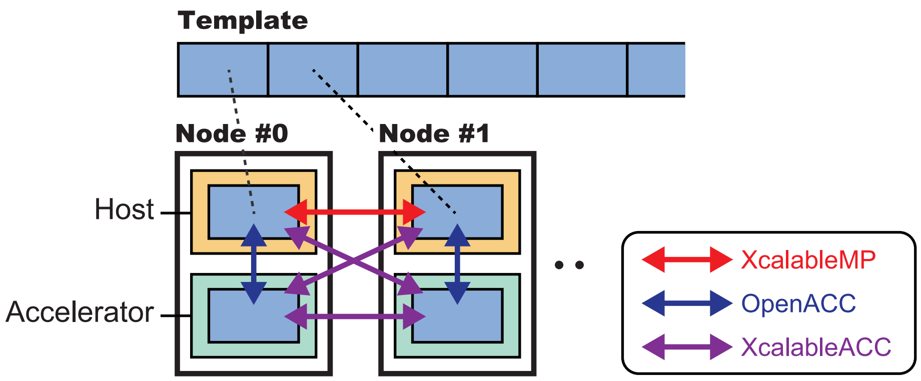

Fig. 2 Concept of XcalableMP Extensions

XcalableMP directives define a **template** and a **node set**. The **template** represents a global index space, which is distributed onto the **node set**. Moreover, XcalableMP directives declare **distributed arrays**, parallelize loop statements and transfer data among host memories according to the distributed **template**. OpenACC directives transfer the **distributed arrays** between host memory and accelerator memory on the same **node** and execute the loop statements parallelized by XcalableMP on accelerators in parallel. XcalableACC directives, which are XcalableMP communication directives with an `acc` clause, transfer data among accelerator memories and between accelerator memory and host memory on different **nodes**. Moreover, **coarray** features also transfer data on different nodes.

Note that the XcalableMP extensions are not a simple combination of XcalableMP and OpenACC. For example, if you represent communication of **distributed array** among accelerators shown in Fig. 2 by the combination of XcalableMP and OpenACC, you need to specify explicitly communication between host and accelerator by OpenACC and that between hosts by XcalableMP. Moreover, you need to calculate manually indices of the **distributed array** owned by each **node**. By contrast, XcalableACC directives can represent such communication among accelerators directly using global indices.

1.2.2 OpenACC Extensions

The OpenACC extension can represent offloading works and data to multiple-accelerators on a **node**. Fig. 3 shows a concept of the OpenACC extension.

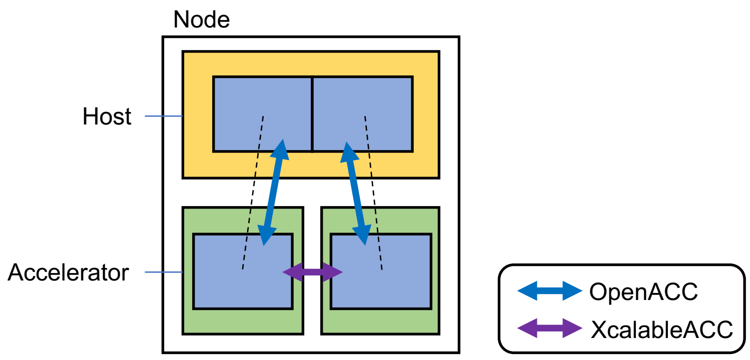

Fig. 3 Concept of OpenACC Extension

OpenACC extension directive defines a **device set**. The **device set** represents a set of devices on a **node**. Futher, OpenACC extension directives declare **distributed arrays** on the **device set** while maintaining the arrays on the host memory, and the directives distribute offloading loop statement and memory copy between host and device memories for the **distributed-arrays**. Moreover, OpenACC extension directives synchronizes devices among the **device set**. XcalableACC directives also transfer data between device memories on the **node**.

1.3 Execution Model

The execution model of XcalableACC is a combination of those of XcalableMP and OpenACC. While the execution model of a host CPU programming is based on that of XcalableMP, that of an accelerator programming is based on that of OpenACC. Unless otherwise specified, each **node** behaves exactly as specified in the XcalableMP specification[1] or the OpenACC specification[2].

An XcalableACC program execution is based on the SPMD model, where each **node** starts execution from the same main routine and keeps executing the same code independently (i.e. asynchronously), which is referred to as the replicated execution until it encounters an XcalableMP construct or an XcalableMP-extension construct. In particular, the XcalableMP-extension construct may allocate, deallocate, or transfer data on accelerators. An OpenACC construct or an OpenACC-extension construct may define **parallel regions**, such as work-sharing loops, and offloads it to accelerators under control of the host.

When a **node** encounters a loop construct targeted by a combination of XcalableMP `loop` and OpenACC `loop` directives, it executes the loop construct in parallel with other **accelerators**, so that each iteration of the loop construct is independently executed by the **accelerator** where a specified data element resides.

When a **node** encounters a XcalableACC synchronization or a XcalableACC communication directive, synchronization or communication occurs between it and other accelerators. That is, such **global constructs** are performed collectively by the **current executing nodes**. Note that neither synchronizations nor communications occur without these constructs specified.

1.4 Data Model

There are two classes of data in XcalableACC: **global data** and **local data** as with XcalableMP. Data declared in an XcalableACC program are local by default. Both **global data** and **local data** can exist on host memory and accelerator memory. About the data models of host memory and accelerator memory, refer to the OpenACC specification[2].

Global data are ones that are distributed onto the **executing node set** by the `align` directive. Each fragment of a **global data** is allocated in host memory of a **node** in the **executing node set**. OpenACC directives can transfer the fragment from host memory to accelerator memory.

Local data are all of the ones that are not global. They are replicated in the local memory of each of the **executing nodes**.

A **node** can access directly only **local data** and sections of **global data** that are allocated in its local memory. To access data in remote memory, explicit communication must be specified in such ways as the global communication constructs and the **coarray** assignments.

Particularly in XcalableACC Fortran, for common blocks that include any global variables, the ways how the storage sequence of them is defined and how the storage association of them is resolved are implementation-dependent.

1.5 Directive Format

This section describes the syntax and behavior of XcalableMP and OpenACC directives in XcalableACC. In this document, the following notation is used to describe the directives.

`xxx`	`type-face` characters are used to indicate literal type characters.
xxx...	If the line is followed by "...", then xxx can be repeated.
[xxx]	*xxx* is optional.
█	The syntax rule continues.
`[F]`	The following lines are effective only in XcalableACC Fortran.
`[C]`	The following lines are effective only in XcalableACC C.

In XcalableACC Fortran, XcalableMP and OpenACC directives are specified using special comments that are identified by unique sentinels `!$xmp` and `!$acc` respectively. the directives follow the rules for comment lines of either the Fortran free or fixed source form, depending on the source form of the surrounding program unit[1]. The directives are case-insensitive.

`[F]` `!$xmp` *directive-name clause*
`[F]` `!$acc` *directive-name clause*

In XcalableACC, XcalableMP and OpenACC directives are specified using the `#pragma` mechanism provided by the C standards. the directives are case-sensitive.

`[C]` `#pragma xmp` *directive-name clause*
`[C]` `#pragma acc` *directive-name clause*

1.6 Organization of This Document

The remainder of this document is structured as follows:

- Chapter 2: XcalableMP Extensions

[1] Consequently, the rules of comment lines that an XcalableMP directive follows is the same as the ones that an OpenMP directive follows.

2 XcalableACC Language

XcalableACC is roughly defined as a diagonal integration of XMP and OpenACC with some additional XACC extensions, where XMP directives are for specifying distributed-memory parallelism, OpenACC for offloading, and the extensions for other XACC-specific features.

The syntax and semantics of XMP and OpenACC directives appearing in XACC codes follow those in XMP and OpenACC, respectively, unless specified below.

2.1 Data Mapping

This chapter defines a behavior of mixing XcalableMP and OpenACC. Note that the existing OpenACC is not extended in the XcalableMP extensions. The XcalableMP extensions can represent (1) parallelization with keeping sequential code image using a combination of XcalableMP and OpenACC, and (2) communication among accelerator memories and between accelerator memory and host memory on different **nodes** using XcalableACC directives or **coarray** features.

When **distributed arrays** appear in OpenACC constructs, global indices in **distributed arrays** are used. The **distributed arrays** may appear in the `update`, `enter data`, `exit data`, `host_data`, `cache`, and `declare` directives, and the `data` clause accompanied by some of `deviceptr`, `present`, `copy`, `copyin`, `copyout`, `create`, and `delete` clauses. Data transfer of **distributed array** by OpenACC is performed on only **nodes** which have elements specified by the global indices.

Example

XcalableACC Fortran

```fortran
integer :: a(N), b(N)
!$xmp template t(N)
!$xmp nodes p(*)
!$xmp distribute t(block) onto p
!$xmp align a(i) with t(i)
!$xmp align b(i) with t(i)
...
!$acc enter data copyin(a(1:K))
!$acc data copy(b)
...
```

XcalableACC C

```c
int a[N], b[N];
#pragma xmp template t[N]
#pragma xmp nodes p[*]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i]
#pragma xmp align b[i] with t[i]
...
#pragma acc enter data copyin(a[0:K])
#pragma acc data copy(b)
{ ...
```

Fig. 4 Code example in XcalableMP extensions with `enter_data` directive

In lines 2-6 of Fig. 4, the directives declare the **distributed arrays** *a* and *b*. In line 8, the `enter data` directive transfers the certain range of the **distributed array** *a* from host memory to accelerator memory. Note that the range is represented by global indices. In line 9, the `data` directive transfers the whole **distributed array** *b* from host memory to accelerator memory.

2.2 Work Mapping

Description

In order to perform a loop statement on accelerators in **nodes** in parallel, XcalableMP `loop` directive and OpenACC `loop` directive are used. While XcalableMP `loop` directive performs a loop statement in **nodes** in parallel, OpenACC `loop` directive also performs the loop statement parallelized by the XcalableMP `loop` directive on accelerators in parallel. For ease of writing, the order of XcalableMP `loop` directive and OpenACC `loop` directive does not matter.

When `acc` clause appears in XcalableMP loop directive with `reduction` clause, the directive performs a reduction operation for a variable specified in the `reduction` clause on accelerator memory.

Restriction

- In OpenACC **compute region**, only XcalableMP `loop` directive without `reduction` clause can be inserted.
- In OpenACC **compute region**, targeted loop condition (lower bound, upper bound, and step of the loop) must remain unchanged.
- `acc` clause in XcalableMP loop directive can appear only when `reduction` clause appears there.

Example 1

In lines 2-6 of Fig. 5, the directives declare **distributed arrays** *a* and *b*. In line 8, the `parallel` directive with the `data` clause transfers the **distributed arrays** *a* and *b* from host memory to accelerator memory. Moreover, in lines 8-9, the `parallel` directive and XcalableMP `loop` directive perform the next loop statement on accelerators in **nodes** in parallel.

Example 2

In lines 2-5 of Fig. 6, the directives declare **distributed array** *a*. In line 7, the `parallel` directive with the `data` clause transfers the **distributed array** *a*

XcalableACC Fortran

```
   integer :: a(N), b(N)
   !$xmp template t(N)
   !$xmp nodes p(*)
   !$xmp distribute t(block) onto p
5  !$xmp align a(i) with t(i)
   !$xmp align b(i) with t(i)
   ...
   !$acc parallel loop copy(a, b)
   !$xmp loop on t(i)
10 do i=0, N
     b(i) = a(i)
   end do
   !$acc end parallel
```

XcalableACC C

```
   int a[N], b[N];
   #pragma xmp template t[N]
   #pragma xmp nodes p[*]
   #pragma xmp distribute t[block] onto p
5  #pragma xmp align a[i] with t[i]
   #pragma xmp align b[i] with t[i]
   ...
   #pragma acc parallel loop copy(a, b)
   #pragma xmp loop on t[i]
10 for(int i=0;i<N;i++){
     b[i] = a[i];
   }
```

Fig. 5 Code example in XcalableMP extensions with OpenACC loop construct

XcalableACC Fortran

```
   integer :: a(N), sum = 10
   !$xmp template t(N)
   !$xmp nodes p(*)
   !$xmp distribute t(block) onto p
5  !$xmp align a(i) with t(i)
   ...
   !$acc parallel loop copy(a, sum) reduction(+:sum)
   !$xmp loop on t(i) reduction(+:sum) acc
   do i=0, N
10   sum = sum + a(i)
   end do
   !$acc end parallel loop
```

XcalableACC C

```
   int a[N], sum = 10;
   #pragma xmp template t[N]
   #pragma xmp nodes p[*]
   #pragma xmp distribute t[block] onto p
5  #pragma xmp align a[i] with t[i]
   ...
   #pragma acc parallel loop copy(a, sum) reduction(+:sum)
   #pragma xmp loop on t[i] reduction(+:sum) acc
   for(int i=0;i<N;i++){
10   sum += a[i];
   }
```

Fig. 6 Code example in XcalableMP extensions with OpenACC loop construct with reduction clause

and variable **sum** from host memory to accelerator memory. Moreover, in lines 7-8, the `parallel` directive and XcalableMP `loop` directive perform the next loop statement on accelerators in **nodes** in parallel. When finishing the calculation of the loop statement, OpenACC `reduction` clause and XcalableMP `reduction` and `acc` clauses in lines 7-8 perform a reduction operation for the variable **sum** on accelerators in **nodes**.

2.3 Data Communication and Synchronization

2.3.1 `acc` clause

2.3.2 Coarray

2.4 Handling Multiple Accelerators

– devices Directive
  – on_device Clause
  – layout Clause
  – shadow Clause
  – barrier_device Construct

3 Omni XcalableACC Compiler

We also develop Omni XACC Compiler as a reference implementation for XACC. Omni XACC Compiler is a source-to-source compiler that translates the base language (C or Fortran) and XACC/XMP directives into runtime calls.

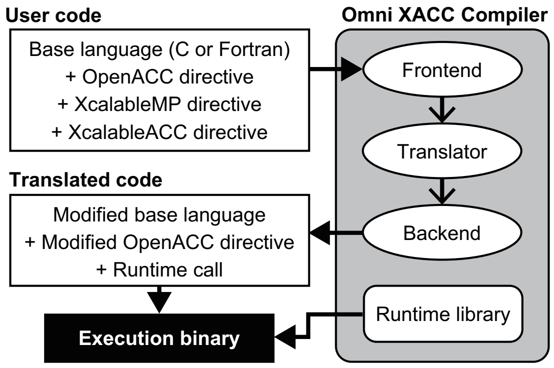

Fig. 7 Compile flow of Omni XcalableACC Compiler

Figure 7 shows the compile flow of Omni XACC Compiler. First, Omni XACC Compiler translates XACC/XMP directives present in the user code into runtime calls. Moreover, a part of the base language and OpenACC directives are modified if necessary. Next, the translated code is compiled by a native compiler to generate an object file. The native compiler must support OpenACC. Finally, the object file and the runtime are linked by the native compiler to generate an execution file.

Omni XACC Compiler dynamically allocates a reasonable memory for distributed arrays. According to the OpenACC specification, when using dynamic memory allocation, the start index and length of the array must be specified in the OpenACC data directive. Therefore, when using a distributed array in the OpenACC data directive, Omni XACC Compiler must insert the start index and length of the array. For example, Omni XACC Compiler translates **#pragma acc data copy(a)** into **#pragma acc data copy(a[start:length])** if an array **a** is a distributed array. The variables **start** and **length** are the start index and length of the distributed array on each node. Omni XACC Compiler also inserts functions to calculate these values before the OpenACC data directive. For example, lines 7 to 14 of Figure 8 are translated into Figure 9. In line 8 of Fig. 9, **_XMP_sched_loop_template_BLOCK()** calculates indexes of a loop statement assigned with each node, and these indexes are used in conditions of the loop statement. Moreover, a distributed array in the loop statement is translated into a pointer calculation. For example, **_XMP_M_GET_ADDR_E_1**() in Fig. 9 is an XMP macro to calculate a local index from a global index.

In order to transfer data between NVIDIA GPUs across compute nodes, we implement the following three methods for Omni XACC Compiler. (1) Using TCA/IB hybrid communication[10, 11]. (2) Using GPUDirect RDMA with CUDA-Aware

```
double a[N];
#pragma xmp template t[N]
#pragma xmp nodes p[4]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i]
...
#pragma acc data copy(a)
{
#pragma xmp loop on t[i]
#pragma acc parallel loop
  for(int i=0;i<N;i++){
    a[i] = ... ;
  }
}
```

Fig. 8 Example of XcalableACC code

```
long long _XACC_start_a = _XACC_get_array_start_index(_XMP_DESC_a, 0);
long long _XACC_length_a = _XACC_get_array_length(_XMP_DESC_a, 0, N−1);
#pragma acc data copy(_XMP_ADDR_a[_XACC_start_a:_XACC_length_a])
{
  int _XMP_loop_init_i;
  int _XMP_loop_cond_i;
  int _XMP_loop_step_i;
  _XMP_sched_loop_template_BLOCK(0, N, 1, &(_XMP_loop_init_i), &(
        _XMP_loop_cond_i), &(_XMP_loop_step_i), ...);
#pragma acc parallel loop
  for(int i = _XMP_loop_init_i; i < _XMP_loop_cond_i; i += _XMP_loop_step_i) {
    (*(_XMP_M_GET_ADDR_E_1(_XMP_ADDR_a, i))) = ... ;
  } // end for
} // end acc data
```

Fig. 9 Example of loop statement translation using Omni XcalableACC Compiler

MPI. (3) Using MPI and CUDA. Item (1) performs communication with the smallest latency, but a TCA system is required for the computing environment. Item (2) is superior in performance to Item (3), but Item (2) also requires specific software and hardware (e.g., MVAPICH2-GDR and Mellanox InfiniBand). Whereas Items (1) and (2) can realize direct communication between GPUs without the CPU, Item (3) copies the data from the accelerator memory to the host memory using CUDA and then transfers the data to other compute nodes using MPI. Therefore, although Item (3) has the lowest performance, it does not require specific software and hardware.

4 Performance of Lattice QCD Application

This section describes the evaluations of XACC performance and productivity for a lattice quantum chromodynamics (Lattice QCD) application.

4.1 Overview of Lattice QCD

The Lattice QCD is a discrete formulation of QCD that describes the strong interaction among “quarks” and “gluons.” While the quark is a species of elementary particles, the gluon is a particle that mediates the strong interaction. The Lattice QCD is formulated on a four-dimensional lattice (time: T and space:ZYX axes). We impose a periodic boundary condition in all the directions. The quark degree of freedom is represented as a field that has four components of “spin” and three components of “color,” namely a complex vector of $12 \times N_{site}$ components, where N_{site} is the number of lattice sites. The gluon is defined as a 3×3 complex matrix field on links (bonds between neighboring lattice sites). During a Lattice QCD simulation, one needs to solve many times a linear equation for the matrix that represents the interaction between the quark and gluon fields. This linear equation is the target of present work. The matrix acts on the quark vector and has nonzero components only for the neighboring sites, and thus sparse.

4.2 Implementation

We implemented a Lattice QCD code based on the existing Lattice QCD application Bridge++[4]. Since our code was implemented by extracting the main kernel of the Bridge++, it can be used as a mini-application to investigate its productivity and performance more easily than use of the original Bridge++.

```
S = B
R = B
X = B
sr = norm(S)
T = WD(U,X)
S = WD(U,T)
R = R − S
P = R
rr = norm(R)
rrp = rr
do{
  T = WD(U,P)
  V = WD(U,T)
  pap = dot(V,P)
  cr = rr/pap
  X = cr ∗ P + X
  R = −cr ∗ V + R
  rr = norm(R)
  bk = rr/rrp
  P = bk ∗ P + R
  rrp = rr
}while(rr/sr > 1.E−16)
```

Fig. 10 Lattice QCD pseudo code

Fig. 10 shows a pseudo code of the implementation, where the CG method is used to solve quark propagators. In Fig. 10, *WD()* is the Wilson-Dirac operator[5], *U* is a gluon, the other uppercase characters are quarks. The Wilson-Dirac operator is a main kernel in the Lattice QCD, which calculates how the quarks interact with each other under the influence of the gluon.

```
 1 typedef struct Quark {
 2   double v[4][3][2];
 3 } Quark_t;
 4 typedef struct Gluon {
 5   double v[3][3][2];
 6 } Gluon_t;
 7 Quark_t v[NT][NZ][NY][NX], tmp_v[NT][NZ][NY][NX];
 8 Gluon_t u[4][NT][NZ][NY][NX];
 9
10 #pragma xmp template t[NT][NZ]
11 #pragma xmp nodes n[NODES_T][NODES_Z]
12 #pragma xmp distribute t[block][block] onto n
13 #pragma xmp align v[i][j][*][*] with t[i][j]
14 #pragma xmp align tmp_v[i][j][*][*] with t[i][j]
15 #pragma xmp align u[*][i][j][*][*] with t[i][j]
16 #pragma xmp shadow v[1:1][1:1][0][0]
17 #pragma xmp shadow tmp_v[1:1][1:1][0][0]
18 #pragma xmp shadow u[0][1:1][1:1][0][0]
19 ...
20 int main(){
21 ...
22 #pragma acc enter data copyin(v, tmp_v, u)
```

Fig. 11 Declaration of distributed arrays for Lattice QCD

Fig. 11 shows how to declare distributed arrays of the quark and gluon. In lines 1-8, the quark and gluon structure arrays are declared. The last dimension "[2]" of both structures represents real and imaginary parts for a complex number. *NT*, *NZ*, *NY* and *NX* are the numbers of TZYX axis elements. In lines 10-18, distributed arrays are declared where the macro constant values *NODES_T* and *NODES_Z* indicate the number of `nodes` on the T and Z axes. Thus, the program is parallelized on T and Z axes. Note that an "*" in the **align** directive means that the dimension is not divided. In the **shadow** directive, halo regions are added to the arrays because each quark and gluon element is affected by its neighboring orthogonal elements. Note that "0" in the **shadow** directive means that no halo region exists. In line 22, the **enter data** directive transfers the distributed arrays from host memory to accelerator memory.

Fig. 12 shows how to call *WD()*. The **reflect** directives are inserted before *WD()* in order to update own halo region. In line 2, "1:0" in **width** clause means only the lower halo region is updated because only it is needed in *WD()*. The *u* is not updated before the second *WD()* function because values of *u* are not updated in

```
#pragma xmp reflect(v) width(/periodic/1:1,/periodic/1:1,0,0) orthogonal acc
#pragma xmp reflect(u) width(0,/periodic/1:0,/periodic/1:0,0,0) orthogonal acc
WD(tmp_v, u, v);
#pragma xmp reflect(tmp_v) width(/periodic/1:1,/periodic/1:1,0,0) orthogonal acc
WD(v, u, tmp_v);
```

Fig. 12 Calling Wilson-Dirac operator

WD(). Moreover, the **orthogonal** clause is added because diagonal updates of the arrays are not required in *WD()*.

```
 1  void WD(Quark_t v_out[NT][NZ][NY][NX], const Gluon_t u[4][NT][NZ][NY][NX],
        const Quark_t v[NT][NZ][NY][NX])
 2  {
 3  #pragma xmp align v_out[i][j][*][*] with t[i][j]
 4  #pragma xmp align u[*][i][j][*][*] with t[i][j]
 5  #pragma xmp align v[i][j][*][*] with t[i][j]
 6  #pragma xmp shadow v_out[1:1][1:1][0][0]
 7  #pragma xmp shadow u[0][1:1][1:1][0][0]
 8  #pragma xmp shadow v[1:1][1:1][0][0]
 9   ...
10  #pragma xmp loop (t,z) on t[t][z]
11  #pragma acc parallel loop collapse(4) present(v_out, u, v)
12   for(int t=0;t<NT;t++)
13    for(int z=0;z<NZ;z++)
14     for(int y=0;y<NY;y++)
15      for(int x=0;x<NX;x++){
```

Fig. 13 A portion of Wilson-Dirac operator

Fig. 13 shows a part of the Wilson-Dirac operator code. All arguments in *WD()* are distributed arrays. In XMP and XACC, distributed arrays which are used as arguments must be redeclared in function to pass their information to a compiler. Thus, the **align** and **shadow** directives are used in *WD()*. In line 10, the **loop** directive parallelizes the outer two loop statements. In line 11, the **parallel loop** directive parallelizes all loop statements. In the loop statements, a calculation needs neighboring and orthogonal elements. Note that while the *WD()* updates only the *v_out*, it only refers the *u* and *v*.

Fig. 14 shows L2 norm calculation code in the CG method. In line 8, the **reduction** clause performs a reduction operation for the variable a in each accelerator when finishing the next loop statement. The calculated variable a is located in both host memory and accelerator memory. However, at this point, all `nodes` have individual values of a. To obtain the total value of the variable a, the XMP **reduction** directive in line 18 also performs a reduction operation among `nodes`. Since the total value

```
double norm(const Quark_t v[NT][NZ][NY][NX])
{
#pragma xmp align v[i][j][*][*] with t[i][j]
#pragma xmp shadow v[1:1][1:1][0][0]
 double a = 0.0;

#pragma xmp loop (t,z) on t[t][z]
#pragma acc parallel loop collapse(7) present(v) reduction(+:a)
 for(int t=0;t<NT;t++)
  for(int z=0;z<NZ;z++)
   for(int y=0;y<NY;y++)
    for(int x=0;x<NX;x++)
     for(int i=0;i<4;i++)
      for(int j=0;j<3;j++)
       for(int k=0;k<2;k++)
        a += v[t][z][y][x].v[i][j][k]*v[t][z][y][x].v[i][j][k];

#pragma xmp reduction (+:a)
 return a;
}
```

Fig. 14 L2 norm calculation code

is used on only host after this function, the XMP **reduction** directive does not have **acc** clause.

5 Performance Evaluation

5.1 Result

Table 1 Evaluation environment

CPU	Intel Xeon-E5 2680v2 2.8 GHz x 2 Sockets
Memory	DDR3 1866MHz 59.7GB/s 128GB
GPU	NVIDIA Tesla K20X (GDDR5 250GB/s 6GB) x 4 GPUs
Network	InfiniBand Mellanox Connect-X3 Dual-port QDR 8GB/s
Software	Intel 16.0.2, CUDA 7.5.18, Omni OpenACC compiler 1.1 MVAPICH2 2.1

This section evaluates the performance level of XACC on the Lattice QCD code. For comparison purposes, those of MPI+CUDA and MPI+OpenACC are also evaluated. For performance evaluation, we use the HA-PACS/TCA system[6] the hardware specifications and software environments of which are shown in Table 1. Since each compute node has four GPUs, we assign four `nodes` per compute node and direct each `node` to deal with a single GPU. We use the Omni OpenACC compiler[7] as a backend compiler in the Omni XACC compiler. We execute the Lattice QCD codes with strong scaling in regions (32,32,32,32) as (*NT*,*NZ*,*NY*,*NX*). The Omni XACC compiler provides various types of data communication among accelerators[8]. We use the MPI+CUDA implementation type because it provides a balance of versatility and performance.

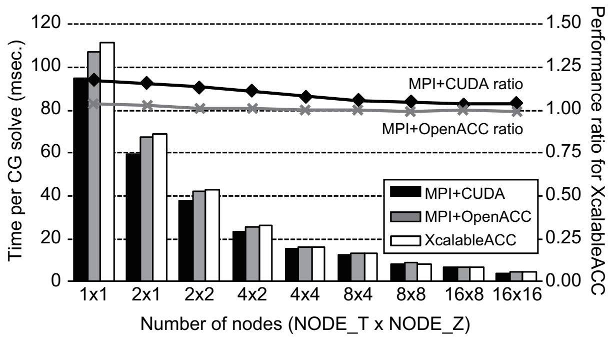

Fig. 15 Performance results

Fig. 15 shows the performance results that indicate the time required to solve one CG iteration as well as the performance ratio values that indicate the comparative performance of XACC and other languages. When the performance ratio value of a language is greater than 1.00, the performance result of the language is better than that of XACC. Fig. 15 shows that the performance ratio values of MPI+CUDA are between 1.04 and 1.18, and that those of MPI+OpenACC are between 0.99 and 1.04. Moreover, Fig. 15 also shows that the performance results of both MPI+CUDA and MPI+OpenACC become closer to those of XACC as the number of `nodes` increases.

5.2 Discussion

To examine the performance levels in detail, we measure the time required for the halo updating operation for two `nodes` and more. The halo updating operation consists of the communication and pack/unpack processes for non-contiguous regions in the XACC runtime.

While Fig. 16 describes communication time of the halo updating time of Fig 15, Fig. 17 describes pack/unpack time of it. Fig. 16 shows that the communication performance levels of all implementations are almost the same. However, Fig. 17 shows that the pack/unpack performance levels of MPI+CUDA are better than those of XACC, and that those of MPI+OpenACC are worse than those of XACC. The reason for the pack/unpack operation performance level difference is that the XACC

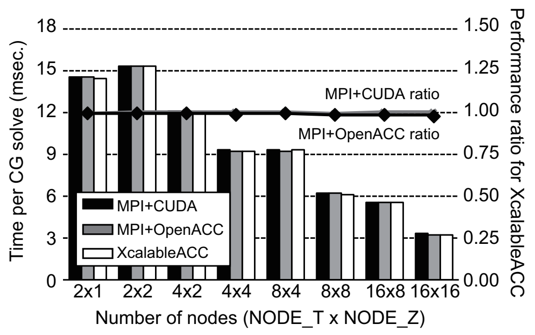

Fig. 16 Communication time

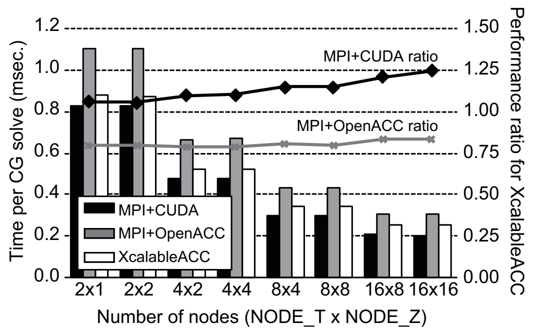

Fig. 17 Pack/unpack time

operation is implemented in CUDA at XACC runtime. Thus, some performance levels of XACC are better than those of MPI+OpenACC in Fig. 15. However, the performance levels of XACC in Fig. 17 is worse than those of MPI+CUDA because XACC requires the cost of XACC runtime calls.

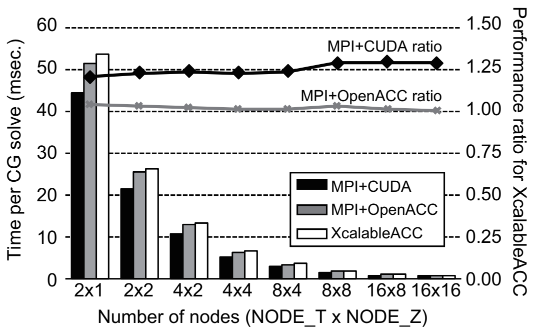

Fig. 18 Time excluding halo updating time

Fig. 18 shows the overall time excluding the halo updating time, where performance levels of MPI+CUDA are the best, and those of XACC are almost the same as those of MPI+OpenACC. The reason for the difference is due to how to use GPU threads. In the CUDA implementation, we assign loop iterations to GPU threads in a cyclic-manner manually. In contrast, in the OpenACC and XACC implementations, how to assign GPU threads is an implementation dependent of an OpenACC compiler. In the Omni OpenACC compiler, initially loop iterations are assigned to GPU threads by a gang (threadblock) in a block manner, and then are also assigned to them by a vector (thread) in a cyclic manner. With the **gang** clause with the **static** argument proposed in the OpenACC specification version 2.0, programmers can determine how to use GPU threads to some extent, but the Omni OpenACC compiler does not yet support it.

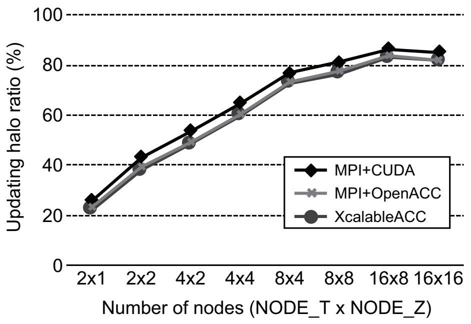

Fig. 19 Updating halo ratio

Fig. 19 shows the ratio of the halo updating time to overall time. As can be seen, as the number of `nodes` increases, the ratio increases as well. Therefore, when a large number of `nodes` are used, there is little difference in performance level of Fig. 15 among the three implementations. The reason why the ratio of MPI+CUDA is slightly larger than those of the others is that the time excluding the halo communication of MPI+CUDA in Fig. 16 is relatively small.

6 Productivity Improvement

6.1 Requirement for Productive Parallel Language

In Section 5, we developed three Lattice QCD codes using MPI+CUDA, MPI+OpenACC, and XACC. Fig. 20 shows our procedure for developing each code where we first develop the code for an accelerator from the serial code, and then extend it to handle an accelerated cluster.

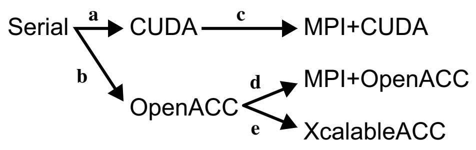

Fig. 20 Application development order on accelerated cluster

To parallelize the serial code for an accelerator using CUDA requires large code changes ("a" in Fig. 20), most of which are necessary to create new kernel functions and to make 1D arrays out of multi-dimensional arrays. By contrast, OpenACC accomplishes the same parallelization with just small code changes ("b"), because OpenACC's directive-based approach encourages reuse of an existing code. Besides, to parallelize the code for a distributed memory system, MPI also requires large changes ("c" and "d"), primarily to convert global indices into local indices. By contrast, XACC requires smaller code changes ("e") because XACC is also directive-based language as OpenACC.

In many cases, a parallel code for an accelerated cluster is based on an existing serial code. The code changes to the existing serial code are likely to trigger program bugs. Therefore, XACC is designed to reuse an existing code as possible.

6.2 Quantitative Evaluation by Delta Source Lines of Codes

Table 2 DSLOC of Lattice QCD implementations

	a	b	c	d	e	a+c	b+d	**b+e**
DSLOC	552	22	280	201	138	832	223	**160**
Add	137	20	185	140	134	322	160	**154**
Delete	73	0	0	0	0	73	0	**0**
Modify	342	2	95	61	4	437	63	**6**

As one of metrics for productivity, Delta Source Lines of Codes (DSLOC) is proposed[9]. The DSLOC indicates how the codes change from a corresponding implementation. The DSLOC is the sum of three components: how many lines are added, deleted and modified. When the DSLOC is small, the programming costs and the possibility of program bugs will be small as well. We use the DSLOC to count the amount of change required to implement an accelerated cluster code from a serial code.

Table 2 shows the DSLOC where lowercase characters correspond to Fig. 20. The DSLOC of XACC (b+e) is smaller than MPI+CUDA (a+c) and MPI+OpenACC (b+d). The difference between XACC and MPI+CUDA is 420.0%, and that between XACC and MPI+OpenACC is 39.4%.

6.3 Discussion

```
1 #pragma acc parallel loop collapse(4) present(v_out, u, v)
2   for(int t=0;t<NT;t++)
3     for(int z=0;z<NZ;z++)
4       for(int y=0;y<NY;y++)
5         for(int x=0;x<NX;x++){
6           int tt = (t + 1) % NT;
7           v_out[tt][z][y][x].v[0][0][0] = ... ;
```

Fig. 21 Code modification of *WD()* in OpenACC

```
1 #pragma xmp loop (t,z) on t[t][z]
2 #pragma acc parallel loop collapse(4) present(v_out, u, v)
3   for(int t=0;t<NT;t++)
4     for(int z=0;z<NZ;z++)
5       for(int y=0;y<NY;y++)
6         for(int x=0;x<NX;x++){
7           int tt = t + 1;
8           v_out[tt][z][y][x].v[0][0][0] = ... ;
```

Fig. 22 Code modification of *WD()* in XcalableACC

In "e" of Table 2, four lines for modification are required to implement the XACC code from the OpenACC code. Fig. 21 and 22 show the modification, which is a part of *WD()* of Fig. 13. A variable *tt* is used to be an index for halo region. The *tt* is modified from line 6 of Fig. 21 to line 7 of Fig. 22. In Fig. 21, when a value of a variable *t* is "*NT*-1", that of the variable *tt* becomes "0" which is the lower bound index of the first dimension of the array *v_out*. On the other hand, in Fig. 22, communication of the halo is performed before execution of *WD()* by the **reflect** directive shown in Fig. 12. Thus, the variable *tt* need only be incremented in Fig. 22. There are four such modifications in *WD()*. Note that XACC does not keep the semantics of the base code perfectly in this case in exchange for simplified parallelization. In addition, there are two lines for modification shown in "b" of Table 2. It is a very fine modification for OpenACC constraints, which keeps the semantics of the base code.

As basic information, we count the source lines of codes (SLOC) of each of the Lattice QCD implementations. Table 3 shows the SLOC excluding comments and blank lines, as well as the numbers of each directive and MPI functions included in their SLOC. For reader information, SLOC of the serial version Lattice QCD code is 842. Table 3 shows that the 122 XMP directives are used in the XACC implementation, many of which are declarations for function arguments. To reduce the XMP directives, we are planning to develop a new syntax that combines declarations with

Table 3 SLOC of Lattice QCD implementations

	MPI+CUDA	MPI+OpenACC	**XcalableACC**
SLOC	1091	1002	**996**
#XcalableMP	-	-	**122**
#OpenACC	-	26	**16**
#XcalableACC	-	-	**3**
#MPI function	39	39	**-**

```
void WD(Quark_t v_out[NT][NZ][NY][NX], const Gluon_t u[4][NT][NZ][NY][NX],
        const Quark_t v[NT][NZ][NY][NX])
{
#pragma xmp align [i][j][*][*] with t[i][j] shadow [1:1][1:1][0][0] :: v_out, v
#pragma xmp align [*][i][j][*][*] with t[i][j] shadow [0][1:1][1:1][0][0] :: u
```

Fig. 23 New directive combination syntax that applies to Fig. 13

the same attribute into one directive. Fig. 23 shows an example of the new syntax applied to the declarations in Fig. 13. Since the arrays *v_out* and *v* have the same attribute, they can be declared into a single XMP directive. Moreover, the **shadow** directive attribute is added to the **align** directive as its clause. When applying the new directive to XACC implementation, the number of XMP directives decreases from the 122 shown in Table 3 to 64, and the XACC DSLOC decreases from the 160 shown in Table 2 to 102.

References

1. XcalableMP Language Specification, http://xcalablemp.org/specification.html (2017).
2. The OpenACC Application Programming Interface, http://www.openacc.org (2015).
3. MPI: A Message-Passing Interface Standard, http://mpi-forum.org (2015).
4. Lattice QCD code Bridge++, http://bridge.kek.jp/Lattice-code/index_e.html.
5. Wilson, K. G., "Confinement of quarks" (1974).
6. HA-PACS, https://www.ccs.tsukuba.ac.jp/supercomputer/.
7. Akihiro Tabuchi et al, "A Source-to-Source OpenACC Compiler for CUDA", Euro-Par Workhops (2013)
8. Masahiro Nakao et al. "XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters", Proceedings of the First Workshop on Accelerator Programming Using Directives (2014)
9. Andrew I. Stone et al. "Evaluating Coarray Fortran with the CGPOP Miniapp", Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (2011)
10. Toshihiro Hanawa et al. "Tightly Coupled Accelerators Architecture for Minimizing Communication Latency among Accelerators," in IPDPSW ’13 Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, pp. 1030–1039 (2013).
11. Yuetsu Kodama et al. "PEACH2: FPGA based PCIe network device for Tightly Coupled Accelerators," in Fifth International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (2013).