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.

1 Introduction

This document defines the specification of XACC (XACC) which is an extension of XMP version 1.3[1] and OpenACC version 2.5[2]. XACC provides a parallel

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

programming model for accelerated clusters which are distributed memory systems equipped with accelerators.

In this document, terminologies of XMP 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 XACC is an accelerated cluster, a hardware model of which is shown in Fig. 1.

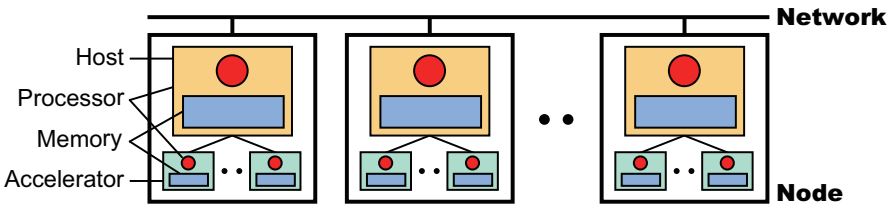

Fig. 1 Hardware Model

An execution unit is called **node** as with XMP. 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

XACC is a directive-based language extension based on Fortran 90 and ISO C90 (ANSI C90). To develop applications on accelerated clusters with ease, XACC extends XACC and OpenACC independently as follow: (1) XMP extensions are

to facilitate cooperation between XMP and OpenACC directives. (2) OpenACC extensions are to deal with multiple accelerators.

1.2.1 XMP Extensions

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

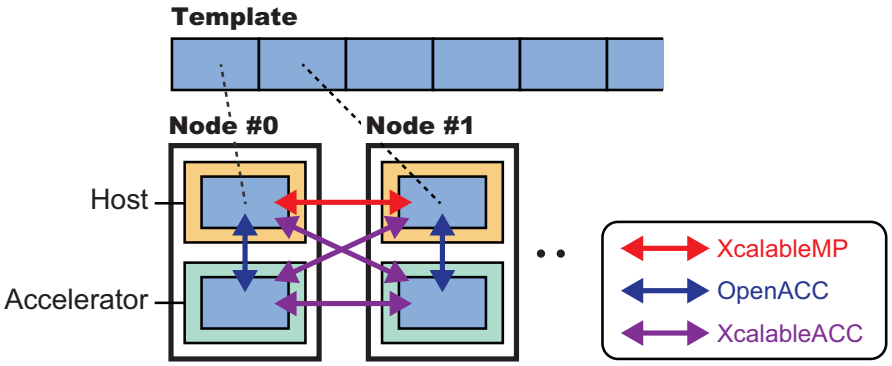

Fig. 2 Concept of XMP Extensions

XMP directives define a **template** and a **node set**. The **template** represents a global index space, which is distributed onto the **node set**. Moreover, XMP 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 XMP on accelerators in parallel. XACC directives, which are XMP 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 XMP extensions are not a simple combination of XMP and OpenACC. For example, if you represent communication of **distributed array** among accelerators shown in Fig. 2 by the combination of XMP and OpenACC, you need to specify explicitly communication between host and accelerator by OpenACC and that between hosts by XMP. Moreover, you need to calculate manually indices of the **distributed array** owned by each **node**. By contrast, XACC 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.

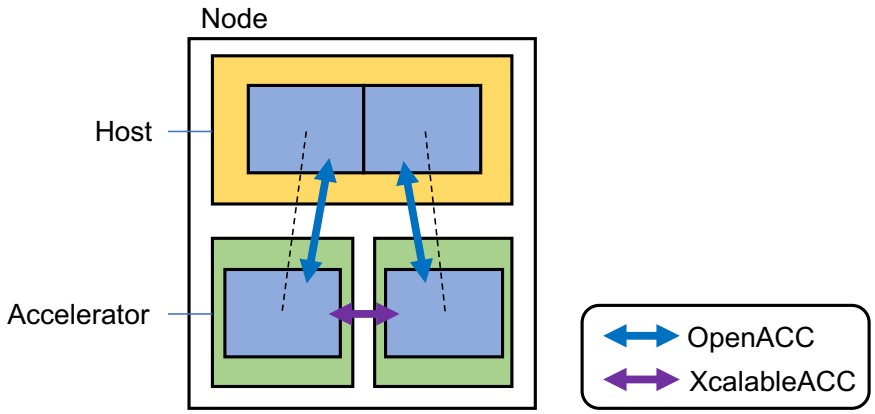

Fig. 3 Concept of OpenACC Extension

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**. XACC directives also transfer data between device memories on the **node**.

1.3 Execution Model

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

An XACC 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 XMP construct or an XMP-extension construct. In particular, the XMP-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 XMP `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 XACC synchronization or a XACC 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 XACC: **global data** and **local data** as with XMP. Data declared in an XACC 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.

2 XcalableACC Language

XACC 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

Global arrays distributed with XMP directives can be globaly-indexed in OpenACC constructs. **Global arrays** may appear in the `update`, `enter data`, `exit data`, `host_data`, `cache`, and `declare` directives; and the data clauses such as `deviceptr`, `present`, `copy`, `copyin`, `copyout`, `create`, and `delete`. When data transfer of a **global array** between host and accelerator memory is specified by a OpenACC directive, it is performed locally for the local section of the array within each node.

Example

In lines 2–6 of Fig. 4, the directives declare **global arrays** `a` and `b`. In line 8, the `enter data` directive transfers a section of `a` from host memory to accelerator memory. Note that `a` is globally-indexed. In line 9, the `data` directive transfers the whole of `b` from host memory to accelerator memory.

XcalableACC 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

```
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 XACC code with `enter_data` directive.

2.2 Work Mapping

In order to parallelize a loop statement among nodes and on accelerators, XMP `loop` directive and OpenACC `loop` directive are used. While an XMP `loop` directive parallelizes a loop statement among nodes, OpenACC `loop` directive further parallelizes the loop statement on accelerators within each node. For ease of writing, the nesting order of XMP `loop` directive and OpenACC `loop` directive does not matter.

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

Restriction

- In an OpenACC **compute region**, no XMPdirectives except for `loop` directive with no `reduction` clause is not allowed.
- In an OpenACC **compute region**, the parameter (i.e., the lower bound, upper bound, and step) of the target loop must remain unchanged.
- An `acc` clause can be specified in an XMP loop directive only when a `reduction` clause is also specified.

Example 1

XcalableACC 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 parallel loop copy(a, b)
!$xmp loop on t(i)
do i=0, N
  b(i) = a(i)
end do
!$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
#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]
for(int i=0;i<N;i++){
  b[i] = a[i];
}
```

Fig. 5 XACCcode with OpenACC `loop` construct.

In lines 2–6 of Fig. 5, the directives declare **global arrays** `a` and `b`. In line 8, the `copy` clause on the `parallel` directive transfers `a` and `b` from host memory to

accelerator memory. In lines 8–9, the `parallel` directive and XMP `loop` directive parallelize the following loop on an accelerator within a node and among nodes, respectively.

Example 2

XcalableACC Fortran

```
integer :: a(N), sum = 10
!$xmp template t(N)
!$xmp nodes p(*)
!$xmp distribute t(block) onto p
!$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
  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
#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++){
  sum += a[i];
}
```

Fig. 6 XACC code with OpenACC `loop` construct with `reduction` clause.

In lines 2–5 of Fig. 6, the directives declare a **global array** `a`. In line 7, the the `copy` clause on the `parallel` directive transfers `a` and a variable `sum` from host memory to accelerator memory. In lines 7–8, the `parallel` directive and XMP `loop` directive parallelize the following loop on an accelerator within a **node** and in among nodes, respectively. After finishing the calculation of the loop, the OpenACC `reduction` clause and the XMP `reduction` clause with `acc` in lines 7–8 perform a reduction operation for `sum` first on the accelerator within a node and then among all **nodes**.

2.3 Data Communication and Synchronization

When an `acc` clause is specified in an XMP's communication and synchronization directive, the directive works for the data on accelerator memory to transfer it.

The `acc` clause can be specified on the following XMP's communication and synchronization directives:

- `reflect`
- `gmove`
- `barrier`
- `reduction`
- `bcast`
- `wait_async`

Note that while a `gmove` directive with `acc` and **coarray** features can perform communication both between accelerators and between accelerator and host memory that may be on different **nodes**, other directives with `acc` can perform communication only between accelerators.

Example

In lines 2–5 of Fig. 7, the directives declare a **global array** `a`. In line 6, the `shadow` directive allocates the shadow areas of the `a`. In line 8, the `enter data` directive transfers `a` with the shadow areas from host memory to accelerator memory. In line 9, the `reflect` directive updates the shadow areas of the **distributed array** a on accelerator memory on all nodes.

XcalableACC Fortran

```
integer :: a(N)
!$xmp template t(N)
!$xmp nodes p(*)
!$xmp distribute t(block) onto p
!$xmp align a(i) with t(i)
!$xmp shadow a(1)
...
!$acc enter data copyin(a)
!$xmp reflect (a) acc
```

XcalableACC C

```
int a[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 shadow a[1]
...
#pragma acc enter data copyin(a)
#pragma xmp reflect (a) acc
```

Fig. 7 Code example in `reflect` construct

2.4 Coarrays

In XACC, programmers can specify one-sided communication (i.e., put and get operation) for data on accelerator memory using **coarray** features. A combination of **coarray** and the `host_data` construct enables one-sided communication between accelerators.

If **coarrays** appear in a `use_device` clause of an enclosing `host_data` construct, data on accelerator memory is selected as the target of the communication. The synchronization for **Coarray** operations on accelerators is similar to that in XMP.

Restriction

- **Coarrays** on accelerator memory can be declared only with the `declare` directive.
- No **coarray** syntax is allowed in the OpenACC **compute region**.

Example

In line 3 of Fig. 8, the `declare` directive declares a **coarray** `a` and an array `b` on accelerator memory. In lines 6–7, **node** 1 performs a put operation, where the the whole of `b` on the accelerator memory of **node** 1 is transferred to `a` on the accelerator memory of **node** 2. In lines 9–10, **node** 1 performs a get operation, where the whole of `a` on the accelerator memory of **node** 3 is transferred to `b` on the host memory of **node** 1. In line 13, the `sync all` statement in XACC/F or the `xmp_sync_all` function in XACC/C performs a barrier synchronizes among all **nodes** and guarantees the completion of ongoing coarray accesses.

XcalableACC Fortran

```
integer :: a(N)[*]
integer :: b(N)
!$acc declare create(a, b)
...
if(this_image() == 1) then
!$acc host_data use_device(a, b)
  a(:)[2] = b(:)

!$acc host_data use_device(a)
  b(:) = a(:)[3]
end if
...
sync all
```

XcalableACC C

```
int a[N]:[*];
int b[N];
#pragma acc declare create(a, b)
...
if(xmp_node_num() == 1){
#pragma acc host_data use_device(a, b)
  a[:]:[2] = b[:];

#pragma acc host_data use_device(a)
  b[:] = a[:]:[3];
}
...
xmp_sync_all(NULL);
```

Fig. 8 XACC code with coarray

2.5 Handling Multiple Accelerators

XACC also has a feature for handling multiple accelerators. This section provides a brief overveiw of this feature. Please refer to [**?**] for more detail.

2.5.1 `devices` Directive

The `devices` directive declares a **device array** that corresponds to a device set. This directive is analogous to the `nodes` directive for nodes in XMP.

Example

The following are examples of the devices declaration. The device array `d` corresponds to a set of entire default devices, and `e` is a subset of the predefined device array `nvidia`. The program must be executed by a node which is equipped with four or more NVIDIA accelerator devices.

XcalableACC Fortran

```
!$acc devices d(*)
!$acc devices e(2) = nvidia(3:4)
```

XcalableACC C

```
#pragma acc devices d[*]
#pragma acc devices e[2] = nvidia[2:2]
```

Fig. 9 XACC code with `devices` directive.

2.5.2 `on_device` Clause

The `on_device` clause in a directive specifies a device set that the directive targets.

The `on_device` clause may appear on `parallel`, `parallel loop`, `kernels`, `kernels loop`, `data`, `enter data`, `exit data`, `declare`, `update`, `wait`, and `barrier_device` directives.

The directive is applied to each device in the device set in parallel. If there is no `layout` clause, the all devices process the directive for same data or work redundantly.

2.5.3 `layout` Clause

The `layout` clause specifies data or work mapping on devices.

The `layout` clause may appear on `declare` directives and on `loop`, `parallel loop`, and `kernels loop` constructs. If the `layout` clause appears on a `declare` directive, it specifies the data mapping to the device set for arrays which are appeared

in data clauses on the directive. "*" represents that the dimension is not distributed, and `block` represents that the dimension is divided into contiguous blocks, which are distributed onto the device array.

Example

The following are examples of the `layout` clause. In line 2, the `devices` directive defines a device set `d`. In lines 3-4, the `declare` directive declares that an array `a` is distributed and allocated on `d`. In lines 6–9, the `kernels loop` directive distributes and offloads the following loops to `d`.

XcalableACC Fortran

```
integer :: a(N)
!$acc devices d(*)
!$acc declare create(a)
!$acc+layout((block)) on_device(d)
...
!$acc kernels loop layout(a(i))
do i = 1, N
  a(i) = i * 2
end do
```

XcalableACC C

```
int a[N];
#pragma acc devices d[*]
#pragma acc declare create(a) \
        layout([block]) on_device(d)
...
#pragma acc kernels loop layout(a[i])
for(int i = 0; i < N; i++){
  a[i] = i * 2;
}
```

Fig. 10 Xacc Code example with `layout` clause.

2.5.4 `shadow` Clause

The `shadow` clause in the `declare` directive specifies the width of the shadow area of arrays, which is used to communicate the neighbor element of the block of the arrays.

Example

The following are examples of the `shadow` clause. In line 2, the `devices` directive defines a device set `d`. In lines 3-5, the `declare` directive declares that an array `a` is distributed and allocated with shadow areas on the device set `d`. In lines 7–10, the `kernels loop` construct divides and offloads the loop to the device set `d`. In line 11, the `reflect` directive updates the shadow areas of the distributed array `a` on the device memory.

XcalableACC Fortran

```
integer :: a(N)
!$acc devices d(*)
!$acc declare create(a)
!$acc+layout((block))
!$acc+shadow((1:1)) on_device(d)
...
!$acc kernels loop layout(a(i))
do i = 1, N
  a(i) = i * 3
end do
!$acc reflect(a)
```

XcalableACC C

```
int a[N];
#pragma acc devices d[*]
#pragma acc declare create(a) \
        layout([block]) \
        shadow([1:1]) on_device(d)
...
#pragma acc kernels loop layout(a[i])
for(int i = 0; i < N; i++){
  a[i] = i * 3;
}
#pragma acc reflect(a)
```

Fig. 11 XACC code with `shadow` clause.

2.5.5 `barrier_device` Construct

The `barrier_device` construct specifies an explicit barrier among devices at the point which the construct appears.

The `barrier_device` construct blocks accelerator devices until all ongoing asynchronous operations on them are completed regardless of the host operations.

Example

The following are examples of the `barrier_devices` construct. In lines 1–2, the `devices` directives define device sets `d` and `e`. In lines 4–5, the first `barrier_device` construct performs a barrier operation for all devices, and the second one performs a barrier operation for devices in the device set `e`.

XcalableACC Fortran

```
!$acc devices d(*)
!$acc devices e(2) = d(1:2)
...
!$acc barrier_device
!$acc barrier_device on_device(e)
```

XcalableACC C

```
#pragma acc devices d[*]
#pragma acc devices e[2] = d[0:2]
...
#pragma acc barrier_device
#pragma acc barrier_device on_device(e)
```

Fig. 12 XACC Code with `barrier_device` construct.

3 Omni XcalableACC Compiler

We have developed the Omni XACC compiler as the reference implementation of XACC compilers.

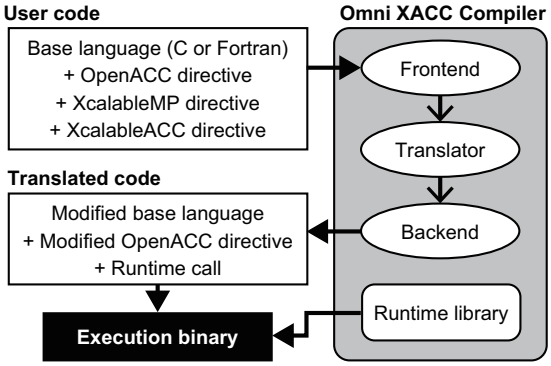

Fig. 13 Compile flow of Omni XcalableACC Compiler

Fig. 13 shows the compile flow of Omni XACC. First, Omni XACC accepts XACC source codes and translates them into those in the base languages with runtime calls. Next, the translated code is compiled by a native compiler, which supports OpenACC, to generate an object file. Finally, the object files and the runtime library are linked by the native compiler to generate an execution file.

In particular, for the data transfer between NVIDIA GPUs across nodes, we have implemented the following three methods in Omni XACC as follows:

(a) TCA/IB hybrid communication
(b) GPUDirect RDMA with CUDA-Aware MPI
(c) MPI and CUDA

Item (a) performs communication with the smallest latency, but it requires a computing environment equipped with the Tightly Copuled Accelerator (TCA) feature[**?**, **?**]. Item (b) is superior in performance to Item (c), but also requires specific software and hardware (e.g., MVAPICH2-GDR and Mellanox InfiniBand). Whereas (a) and (b) can realize direct communication between GPUs without the intervention of CPU, Item (c) cannot. It copies the data from accelerator memory to host memory using CUDA and then transfers the data to other compute nodes using MPI. Therefore, although its performacnce is the lowest, it requires neither specific software nor 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++.

```
1  S = B
2  R = B
3  X = B
4  sr = norm(S)
5  T = WD(U,X)
6  S = WD(U,T)
7  R = R − S
8  P = R
9  rr = norm(R)
10 rrp = rr
11 do{
10   T = WD(U,P)
11   V = WD(U,T)
12   pap = dot(V,P)
13   cr = rr/pap
14   X = cr ∗ P + X
15   R = −cr ∗ V + R
16   rr = norm(R)
17   bk = rr/rrp
18   P = bk ∗ P + R
19   rrp = rr
20 }while(rr/sr > 1.E−16)
```

Fig. 14 Lattice QCD pseudo code

Fig. 14 shows a pseudo code of the implementation, where the CG method is used to solve quark propagators. In Fig. 14, *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. 15 Declaration of distributed arrays for Lattice QCD

Fig. 15 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. 16 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

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

Fig. 16 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. 17 A portion of Wilson-Dirac operator

Fig. 17 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. 18 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

```
1  double norm(const Quark_t v[NT][NZ][NY][NX])
2  {
3  #pragma xmp align v[i][j][*][*] with t[i][j]
4  #pragma xmp shadow v[1:1][1:1][0][0]
5    double a = 0.0;
6
7  #pragma xmp loop (t,z) on t[t][z]
8  #pragma acc parallel loop collapse(7) present(v) reduction(+:a)
9    for(int t=0;t<NT;t++)
10     for(int z=0;z<NZ;z++)
11       for(int y=0;y<NY;y++)
12         for(int x=0;x<NX;x++)
13           for(int i=0;i<4;i++)
14             for(int j=0;j<3;j++)
15               for(int k=0;k<2;k++)
16                 a += v[t][z][y][x].v[i][j][k]*v[t][z][y][x].v[i][j][k];
17
18 #pragma xmp reduction (+:a)
19   return a;
20 }
```

Fig. 18 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.

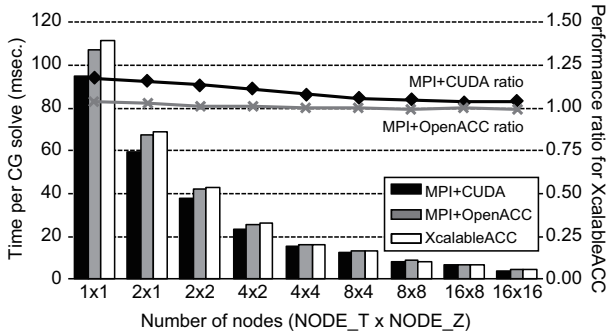

Fig. 19 Performance results

Fig. 19 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. 19 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. 19 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. 20 describes communication time of the halo updating time of Fig 19, Fig. 21 describes pack/unpack time of it. Fig. 20 shows that the communication performance levels of all implementations are almost the same. However, Fig. 21 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

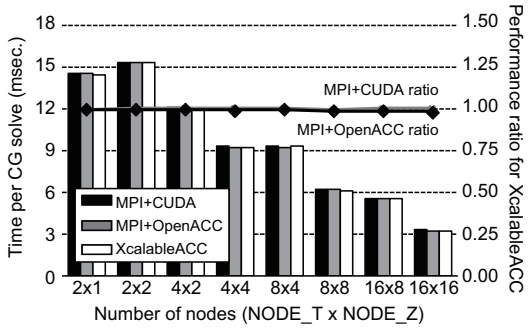

Fig. 20 Communication time

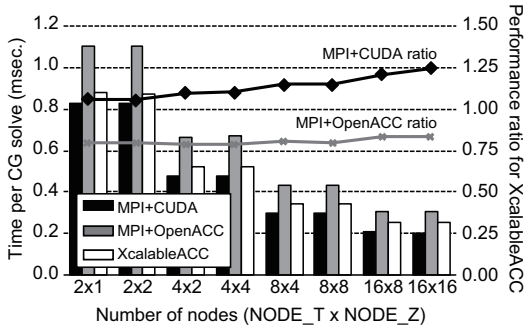

Fig. 21 Pack/unpack time

operation is implemented in CUDA at XACC runtime. Thus, some performance levels of XACC are better than those of MPI+OpenACC in Fig. 19. However, the performance levels of XACC in Fig. 21 is worse than those of MPI+CUDA because XACC requires the cost of XACC runtime calls.

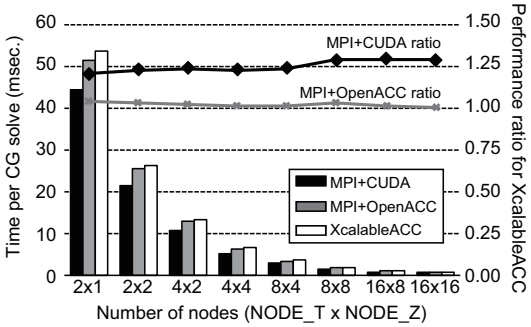

Fig. 22 Time excluding halo updating time

Fig. 22 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.

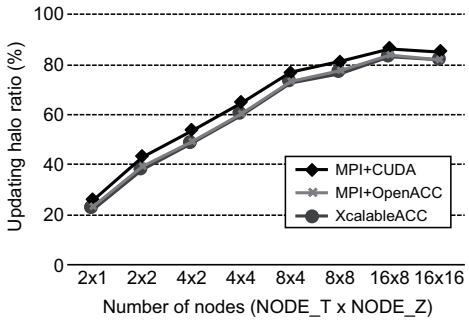

Fig. 23 Updating halo ratio

Fig. 23 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. 19 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. 20 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. 24 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.

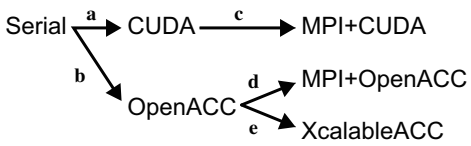

Fig. 24 Application development order on accelerated cluster

To parallelize the serial code for an accelerator using CUDA requires large code changes ("a" in Fig. 24), 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. 24. 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. 25 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. 26 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. 25 and 26 show the modification, which is a part of *WD()* of Fig. 17. A variable *tt* is used to be an index for halo region. The *tt* is modified from line 6 of Fig. 25 to line 7 of Fig. 26. In Fig. 25, 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. 26, communication of the halo is performed before execution of *WD()* by the **reflect** directive shown in Fig. 16. Thus, the variable *tt* need only be incremented in Fig. 26. 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	**-**

```
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 [i][j][*][*] with t[i][j] shadow [1:1][1:1][0][0] :: v_out, v
4 #pragma xmp align [*][i][j][*][*] with t[i][j] shadow [0][1:1][1:1][0][0] :: u
```

Fig. 27 New directive combination syntax that applies to Fig. 17

the same attribute into one directive. Fig. 27 shows an example of the new syntax applied to the declarations in Fig. 17. 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)