QCD Library for GPU Cluster with Proprietary Interconnect for GPU Direct Communication

Norihisa Fujita[1], Hisafumi Fujii[1], Toshihiro Hanawa[2], Yuetsu Kodama[3], Taisuke Boku[1,3], Yoshinobu Kuramashi[3], and Mike Clark[4]

[1] Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Japan
[2] Information Technology Center, The University of Tokyo
[3] Center for Computational Sciences, University of Tsukuba, Tsukuba, Japan
[4] NVIDIA Corporation

Abstract. QUDA is a Lattice QCD library that can use NVIDIA's Graphics Processing Unit (GPU) accelerators, and is widely used as a framework for Lattice QCD applications. In this paper, we apply our novel proprietary interconnect network called the Tightly Coupled Accelerators (TCA) architecture, to inter-node GPU communication in QUDA. The TCA architecture was developed for low-latency inter-node communication among accelerators connected through the PCI Express (PCIe) bus on PC clusters. It enables direct memory copy between accelerators, such as GPUs, over nodes in the same manner as an intra-node PCIe transaction. We assess the performance of TCA on QUDA by a high-density GPU cluster HA-PACS/TCA, which is a proof-of-concept testbed for TCA architecture. The results show that our interconnection network system, which effects a stronger scaling than ordinary InfiniBand solutions on PC clusters with GPUs, significantly reduces communication latency. The execution time for Conjugate Gradient (CG) iteration shows that the TCA implementation is 2.14 times faster than peer-to-peer MPI implementation and 1.96 times faster than MPI remote-memory access (RMA) implementation, where InfiniBand QDRx2 rail network is used in both cases.

1 Introduction

In recent years, research related to General-Purpose computing on Graphics Processing Units (GPGPU) has focused on High Performance Computing (HPC). Many GPGPU clusters are in the TOP500 list of the most powerful computer systems [1]. In general, GPUs use Peripheral Component Interconnect Express (PCIe) [2] buses in order to communicate with CPUs and other GPUs in the same computation node. However, the transfer bandwidth of a PCIe bus is at most several tens part of the memory bandwidth of a GPU. Thus, the bandwidth of PCIe buses becomes a bottleneck for parallel GPU applications. Moreover, since GPUs cannot communicate directly with GPUs in other nodes, the CPU has to assist data exchange between GPUs from different nodes. As the data is thus relayed through the CPU, inter-node GPU communication latency becomes larger than inter-node CPU communication latency.

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 251–262, 2014.
© Springer International Publishing Switzerland 2014

In past research, we developed an interconnection network system based on the Tightly Coupled Accelerator (TCA) architecture that enables inter-node GPU communication. The TCA architecture is based on the concept of expanding the PCIe link to inter-node communication between accelerators over the multiple nodes. In this network system, all devices connected through PCIe can theoretically communicate directly across nodes. In this paper, we apply our TCA technology to QUDA which is widely used as a framework for Lattice QCD applications with NVIDIA GPU accelerators.

2 QUDA

QUDA is an open-source Lattice Quantum Chrono-Dynamics (LQCD) framework library developed by Mike Clark et al. that supports NVIDIA GPU accelerators [3,4]. Roughly speaking, there are two kinds of computations in QUDA: stencil computations in the Dirac operator, and computations to solve linear equations using Krylov solvers. QUDA supports multiple Krylov solvers, such as the Conjugate Gradient (CG) method and the BiConjugate Gradient Stabilized (BiCGSTAB) method, which are chosen according to the matrix type that needs to be solved.

QUDA supports a single GPU environment as well as a multi-node and multi-GPU environment [5]. It can use the MPI [6] and Lattice QCD Message Passing (QMP) [7] as a communication API. We will add TCA support to QUDA as an extension of the MPI. As we detail in the next section, the TCA architecture is currently not self-contained. Because of this, we employ MPI for the general preparation phase of the computation, which does not affect overall performance, and TCA for critical communication in the body of the main loop of the code.

3 Tightly Coupled Accelerators (TCA) Architecture and PEACH2

Tightly Coupled Accelerator (TCA) architecture enables inter-node GPU communication using PCIe technology as a communication link. The PCI Express Adaptive Communication Hub ver. 2 (PEACH2) is a proof-of-concept implementation of the TCA architecture developed at the Center for Computational Sciences in the University of Tsukuba [8][9]. It is an interface board for low-latency inter-node communication among accelerators.

3.1 Overview of TCA Architecture

TCA uses PCIe as an inter-node communication channel. PCIe is a serial bus interface that is widely used to connect most peripheral devices, including extension boards such as GPUs, Ethernet boards, and InfiniBand HCAs in PCs. While PCIe is commonly used for intra-node interconnection, it is possible to extend it to inter-node interconnection using PCIe external cables [10] under

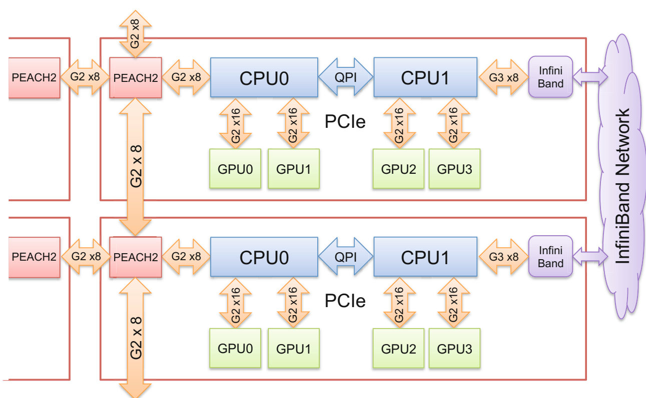

Fig. 1. TCA connection in the HA-PACS/TCA cluster

the following condition. PCIe is a packet-based network that consists of one root complex (RC) and a few end points (EPs) in a single-address space of the network. RC is usually a CPU and EPs are extension boards. Since only one RC is allowed in a PCIe network, we cannot assign multiple nodes to a PCIe network without special routing mechanisms. The PEACH2 chip acts as a router to effect inter-node communication, hence solving this problem to allow multiple RCs in a network system.

Figure 1 shows a typical computation node with the PEACH2 board and multiple GPUs. An Intel Xeon E5 (SandyBridge-EP or IvyBridge-EP architecture) processor is assumed to be the host CPU because it supports the local switching of PCIe on the chip in order to handle multiple PCIe devices that are directly attached to the CPU. We implement this configuration on the Highly Accelerated Parallel Advanced system for Computational Sciences with the TCA feature (HA-PACS/TCA) GPU cluster, which is a proof-of-concept testbed for the TCA architecture. The PEACH2 board is connected to CPU0 and the InfiniBand HCA is connected to CPU1, where both network interfaces require eight lanes of PCIe respectively. Since all CPUs and GPUs share the same PCIe address space, PEACH2 can access all of their memory. However, we assume to use only two GPUs (GPU0 and GPU1) connected to the same CPU with the PEACH2 board because PCIe communication throughput degrades drastically over Intel's QuickPath Interconnect (QPI). This problem of PCIe communication over QPI is a well-known fact and does not limit the use of the TCA concept in general. For example, we can connect four or more GPUs to PEACH2 without creating a bottleneck if an appropriate PCIe switch, such as PLX, is used instead of the internal PCIe switch on an Intel Xeon E5 CPU.

PEACH2 has a DMA Controller (DMAC) with four DMA channels. *DMA descriptors* are prepared to invoke PEACH2's DMAC. We can prepare multiple DMA descriptors for various communication patterns. It contains information regarding the source memory region and the destination region. PEACH2's DMAC

has a chaining DMA function. We first create the required DMA descriptors and chain them as a linked list. Once we invoke the top of the DMA descriptor in a DMA chain, following communication transactions are invoked automatically and continuously with the chained descriptors.

3.2 GPUDirect RDMA

PEACH2 uses GPUDirect Remote DMA (GDR) [11] to access GPU memory, which allows the mapping of GPU address space to PCIe address space. CUDA 5.0 or later and NVIDIA Kepler or later generation of GPUs are required to use GDR. Other devices that share the same PCIe address space can logically access GPU memory in the same PCIe address space. By this mechanism, GPU-to-GPU memory copying can be handled as PCIe-to-PCIe device memory copying.

4 Implementation of QUDA on TCA Architecture

We focus on accelerating communication during stencil computations in QUDA by using the TCA architecture. The original QUDA uses send-receive semantics, i.e., a point-to-point protocol, in both the MPI and Lattice QCD Message Passing (QMP) communication schemes. On the other hand, since TCA supports Remote Memory Access (RMA) Write and Read (emulated by Proxy Write) in one-sided communication, we need to change the communication structure of QUDA from its original point-to-point protocol.

4.1 MPI-3 Remote Memory Access

TCA supports only one-sided write operations, such as the `MPI_Put` function. Therefore, we first rewrite QUDA's communication protocol using MPI-3 Remote Memory Access (RMA) before rewriting it using TCA. We consider that two-phase porting (MPI point-to-point $\rightarrow$ MPI RMA $\rightarrow$ TCA) is more reasonable than direct porting from MPI point-to-point to TCA because new RMA-version of code can be applied for any MPI-3 ready GPU clusters, which is expected to enhance the performance even without TCA. In the performance comparison, we compare the original point-to-point MPI, MPI-RMA and TCA implementations.

We implement MPI RMA support and TCA support based on the method of the QUDA communicator, which consists of multiple communicators in its abstraction layer. We call our extended method the "QUDA RMA method." Figure 2 shows the basic code flow of the RMA method. We describe the details of these functions in the following subsections.

4.2 Message Handle

QUDA has a `MsgHandle` object that represents a communication entity. In its implementation, the MPI point-to-point protocol contains an `MPI_Request` created

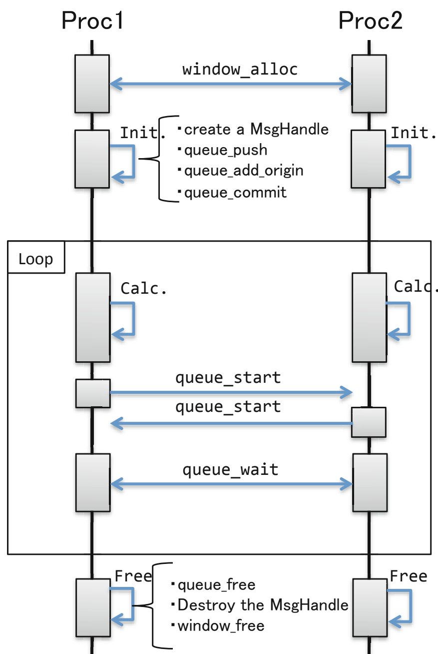

Fig. 2. Basic code flow of the RMA method

by persistent APIs, such as the `MPI_Send_init` function, to preserve communication properties. `MPI_Request` contains all information required by `MPI_Send` and `MPI_Recv` such as pointers, data types, data lengths, etc. We extend `MsgHandle` to support RMA communication for MPI RMA and TCA. We describe the details of the extension for the two implementations in Section 5.

4.3 Memory Window Object

We introduce an `RmaWindow` object that represents a memory region for RMA. Since QUDA uses only GPU memory for RMA communication, an `RmaWindow` does not contain CPU memory. The source and destination memory regions specified using the RMA method must belong to an `RmaWindow`.

4.4 RMA Operation Queue

We introduce an `RmaQueue` object used for RMA operation management. An `RmaQueue` object has two kinds of information: that associated with `MsgHandle`, and the origin rank of the communication process.

We can use the following functions in the method:

Alloc
: creates a new queue associated with a specified `RmaWindow`.

Free
: destroys a queue.

Start
: starts RMA operations with `MsgHandle`s in the queue.

Wait
: waits for RMA operations issued by Start to complete.

Push
: adds an RMA operation to the queue.

Commit
: tells the queue that all RMA operations have been added (Pushed) and the relevant queue is ready.

Add Origin
: adds an origin process to wait.

Clear
: removes all operations from the queue.

The *Add Origin* operation is used to wait until RMA write operations on remote processes are completed. The RMA method should know which rank of process would write to its memory to wait these operations before the next calculation begins. The *Push* and *Commit* operations are too expensive to perform one after the other. Thus, we have to reuse the same queue as much as possible. Since the *Start* and the *Wait* operations do not modify the contents of the queue, we can reuse the same `RmaQueue` if we do not need to modify the communication pattern. This condition is satisfied by typical scientific computations, including QUDA's communication protocol.

4.5 Rewriting Point-to-Point Communication

We rewrite point-to-point communication to RMA communication in QUDA. We only use RMA write operations, replacing send operations with write operations and removing receive operations. This is required to attain high performance in TCA implementation, where RMA Write is the fundamental communication operation.

In order to hide communication latency in MPI point-to-point implementation, MPI communications and subsequent calculations are pipelined and overlapped using the `MPI_Isend` and `MPI_Irecv` functions. To implement overlapping, we need a fine-grained synchronization method. However, since MPI RMA synchronization methods are not fine grained, the *Wait* queue operation waits until all RMA operations are completed. We intend to improve the performance of RMA synchronization area in future research.

QUDA supports domain decomposition in the dimensions x, y, z, and t. The communication pattern (message address, length and communicating partners) of each dimension is not changed over the loop iteration. Therefore, we can apply the DMA chaining mechanism of PEACH2 to this communications repeatedly.

5 RMA Implementation

5.1 MPI RMA Implementation

In MPI point-to-point implementation, a `MsgHandle` object contains an `MPI_Request` and `MPI_Datatype` attributes, and the `RmaWindow` object contains an `MPI_Win` object. Since MPI does not provide a persistent API for RMA APIs, such as the `MPI_Send_init` function, a `MsgHandle` contains all parameters that will be passed to the `MPI_Put` function in MPI RMA implementation. We can create an `MPI_Request` through the `MPI_Rput` function. However, we cannot use the `MPI_Rput` function because the request begins automatically. In MPI RMA implementation, the `Commit` queue operation does nothing. There is no space for optimization in this phase.

The *Wait* queue operation uses the `MPI_Win_post`, `MPI_Win_start`, `MPI_Win_complete`, and `MPI_Win_wait` functions to wait for RMA operations. The `MPI_Win_post` function starts an exposure RMA epoch and the `MPI_Win_complete` function closes the epoch. These functions control RMA access from specified processes. We only wait to complete the RMA operations that are related to a process, namely RMA write operations initiated by the process and operations that will write to the process. The `Add Origin` operation creates an `MPI_Group` to be passed to the first argument of the `MPI_Win_post` function.

5.2 TCA Implementation

The TCA environment is developed on the CUDA Toolkit [12] provided by NVIDIA. Therefore, GPU manipulations, such as memory management, data transfer, and kernel launch, are conducted in the same manner as when TCA is not used. TCA uses the PCIe address space to specify memory location. However, pointers from the PCIe address space are generally not the same as CPU/GPU pointers used in applications. Thus, we use the *memory handle* model API to avoid using the inconvenient PCIe address space. We create a handle of GPU memory allocated through the CUDA API (e.g., `cudaMalloc`). It is called a `tcaHandle` and contains the PCIe address of the memory region and the node id.

In TCA implementation, a `MsgHandle` object contains all parameters to pass to the TCA APIs, as in the MPI RMA implementation. In the MPI RMA implementation, we can use `MPI_Win` to communicate with all processes in an MPI version of communicator. However, since a `tcaHandle` object can be used with a node determined at its time of creation, we have 16 handles for all nodes in it.

In contrast to MPI implementation, the `Commit` queue operation is important for TCA implementation. We create DMA descriptors and a DMA chain in the operation. To achieve the best performance on TCA, all communication should be in the form of a series of DMA chains. Transactions in a DMA chain are begun using the `tcaStartDMADesc` function in the stencil communication phase.

We wait for RMA operations in the queue to be completed in the `Wait` queue operation. In addition to waiting for these, the RMA method knows the ranks

Table 1. The node specifications of HA-PACS/TCA

CPU	Intel(R) Xeon(R) CPU E5-2680 v2 2.80 GHz × 2
CPU Memory	64 GB / CPU
GPU	NVIDIA Tesla K20X × 4
GPU Memory	6 GB / GPU
IB HBA	Mellanox Connect-X3 Dual-port QDR
PEACH2	1 board / node
OS	CentOS 6.4
CUDA Toolkit	version 6.0
GPU Driver	version 311.67
MPI	MVAPICH2-GDR 2.0b built for CUDA 6.0

of processes that will write to the target process using the `Add Origin` queue operation, so that we wait for all RMA operations issued to the process by other processes in the `Wait` queue operation.

6 Performance Evaluation

In this section, we compare the performance of three implementations: MPI point-to-point (the original QUDA implementation), MPI RMA, and TCA.

6.1 Machine Environment

For performance evaluation, We use the HA-PACS/TCA GPU cluster, which was developed in the Center for Computational Sciences at the University of Tsukuba. The node specifications are listed in Table 1 and the node construction is shown in Figure 1. There are two CPUs in a node; a PEACH2 board is attached to CPU0 and an InfiniBand HCA is attached to CPU1. We use CUDA Toolkit 6.0 and MVAPICH2-GDR 2.0b as an MPI implementation.

PEACH2 and InfiniBand do not access GPU memory over QPI because of performance degradation. Therefore, PEACH2 only directly accesses the memories of GPU0 and GPU1, whereas InfiniBand only directly accesses the memories of GPU2 and GPU3, as shown in Figure 1. Since QUDA supports only one GPU per process, we measured and compared single-GPU performances in this study. We use GPU0 to measure the performance of TCA and GPU2 for that of InfiniBand.

6.2 Performance Measurement

We use the `invert_test` test program provided by QUDA for performance measurement. It solves a linear equation using a CG solver and shows its performance in terms of GFLOPS and the number of iterations.

X, Y, Z, and T denote the mesh size in each dimension x, y, z, and t, respectively, on each process. n_X, n_Y, n_Z, and n_T denote the number of processes of

each dimension. $N_X,\ N_Y,\ N_Z$, and N_T denote the mesh size of each dimension in total, where $(N_X,\ N_Y,\ N_Z,\ N_T) = (X \times n_X,\ Y \times n_y,\ Z \times n_Z,\ T \times n_T)$.

We measured two kinds of mesh size parameters, $(N_X,\ N_Y,\ N_Z,\ N_T) = (8,\ 8,\ 8,\ 8)$ named "Small Model" and $(N_X,\ N_Y,\ N_Z,\ N_T) = (16,\ 16,\ 16,\ 16)$ named "Large Model", and various node sizes to assess the strong scaling. Since current PEACH2 implementation supports up to 16 nodes in a network, we made our measurement using 16 nodes at most. Figure 3 shows the performance using MPI point-to-point, MPI-RMA, and TCA for the Small Model, whereas Figure 4 shows the performance of each of the three networks for the Large Model. These performance are measured on Rank 0. With the same number of computation nodes (processes), the process granularity of the $(8, 8, 8, 8)$ problem is $\frac{1}{16}$ of the $(16, 16, 16, 16)$ problem because it is a four-dimensional domain decomposition problem. Since the number of iterations of the CG solver may differ for each configuration, the time magnitudes in Figures 3 and 4 are normalized by the number of iteration. The message size in the Small Model is $2 \times \frac{24\text{KB}}{n}$, where n is the number of nodes in a dimension. The message size in the Large Model is $2 \times \frac{192\text{KB}}{n}$. Since there are two neighboring processes in each dimension, two messages are issued for each dimension if the dimension is divided into processes.

In case of the Small Model, the TCA implementation is faster than both MPI implementations in all configurations. In the case of $(n_x,\ n_y) = (2,\ 2)$, the TCA implementation is 2.14 times faster than the MPI point-to-point implementation and approximately 1.96 times faster than the MPI RMA implementation. The calculation times do not scale well in this problem because the mesh size is too small and the high cost of launching CUDA kernels renders them a bottleneck. The MPI RMA implementation is slightly faster than the MPI point-to-point implementation in all configurations.

For the Large Model, since the peak bandwidth of TCA is lower than that of InfiniBand [1], TCA is slower than the MPI implementations for some configurations. The TCA implementation is slower than both MPI implementations in cases involving two nodes, is almost eve with the MPI peer-to-peer implementation in cases involving four nodes, and is faster than the MPI peer-to-peer implementation in cases involving eight and 16 nodes. In the case of $(n_x,\ n_y) = (4,\ 4)$, the TCA implementation is 1.14 times faster than the MPI peer-to-peer implementation. Since the MPI RMA implementation does not support overlapping between calculation and communication at present, as described in the previous section, the MPI RMA implementation is slower than the MPI peer-to-peer implementation in some configurations. In contrast to cases involving large numbers of nodes, the MPI point-to-point implementation and the MPI RMA implementation exhibit almost identical performance in cases involving small numbers of nodes.

[1] Due to FPGA technology used in PEACH2, all PCIe ports are limited as PCIe gen.2 x8 lanes while InfiniBand dual port QDA HCA can use PCIe gen.3 x8 lanes. Therefore, TCA achieves much shorter latency than InfiniBand but the bandwidth becomes lower beyond certain message size.

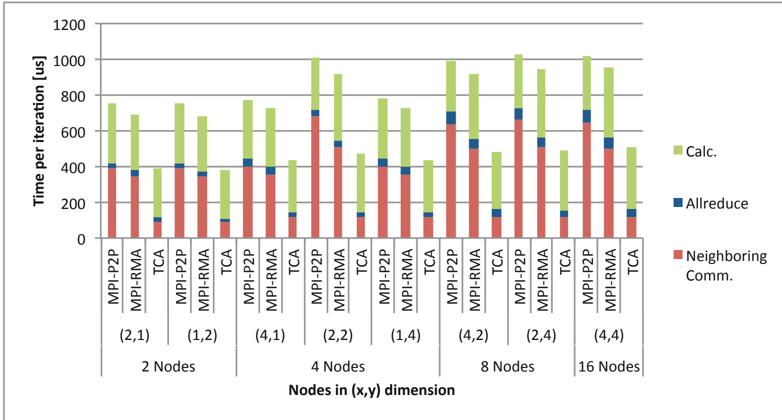

Fig. 3. Performance in the case of the Small Model

The Small Model where $(n_x,\ n_y) = (2,\ 2)$ is the configuration most improved by TCA in both measurements. In communication time comparisons in particular, the TCA implementation is 5.90 times faster than the MPI peer-to-peer implementation in the Small Model when $(n_x,\ n_y) = (2,\ 4)$, and is 4.54 times faster than the MPI RMA implementation in the Small Model when $(n_x,\ n_y) = (2,\ 4)$.

In the comparison between communication times in Figure 3 and Figure 4, and between the message sizes of the two problem sizes, we see that TCA is useful for communication involving small messages. A message in the Small Model is less than or equal to 24 KB in all configurations. The message size in the Large Model when $(n_x,\ n_y) = (4,\ 4)$, which is the configuration most improved in the Large Model by TCA, is 48 KB for the x and y dimensions. However, since the bandwidth of the TCA for large messages is lower than that of InfiniBand [8], TCA performance in the Large Model with two nodes is unsatisfactory. We think that the application can be accelerated by adopting a hybrid approach by switching the channel of a message depending on the message size and its destination GPU. We plan to continue work on improving the performance of TCA application. In general, in a strong configuration, it is important for applications to transfer small messages quickly because the message size in applications decreases as the number of nodes increases.

7 Related Work

APEnet+[13] is a field-programmable gate array (FPGA)-based 3D torus network architecture that, like TCA, supports direct GPU communication [14] but uses their original protocol.

Mellanox's InfiniBand Host Bus Adapter (HBA) supports GPUDirect RDMA for direct GPU memory access [15]. We can use GPU pointers with the Verbs

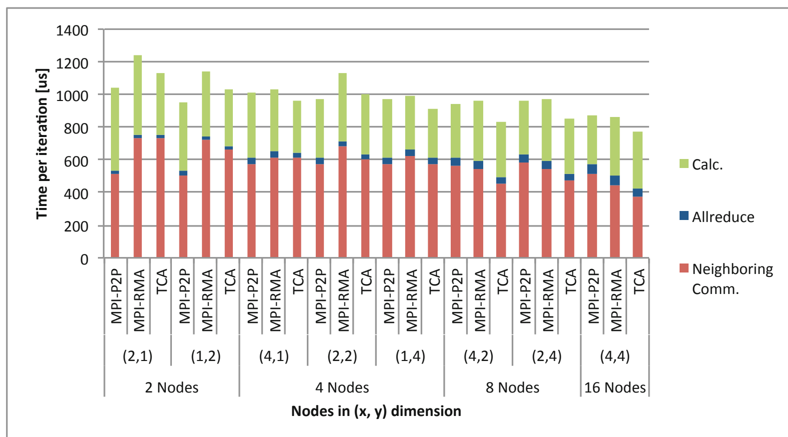

Fig. 4. Performance in the case of the Large Model

API if the environment supports RDMA. Some MPI implementations [16,17] support the use of InfiniBand HBA with GPUDirect RDMA. By using a GDR-supported MPI, we can pass GPU pointers to a few MPI APIs. For example, we can pass a GPU pointer to the first argument of the `MPI_Send` function.

These technologies are similar to TCA but do not use the PCIe protocol. Since TCA does not need to convert PCIe into another protocol, it has an advantage over rival technologies in terms of communication latency.

8 Conclusion

In this paper, we ported the open-source Lattice QCD framework library QUDA, which supports NVIDIA GPU accelerators, to our novel interconnection TCA. TCA is an interconnection concept to expand the PCIe channel to inter-node communication. It enables us to access GPU memory across the network.

TCA only supports one-sided write operations, such as the `MPI_Put` function. Therefore, we first extended QUDA's communication abstraction layer to support RMA, and then implemented TCA API support and MPI RMA support on it.

In the performance comparison among the MPI point-to-point implementation, the MPI RMA implementation, and the TCA implementations, the execution time of CG iteration shows that the TCA implementation is 2.14 times faster than the MPI peer-to-peer implementation and 1.96 times faster than the MPI RMA implementation in case of the Small Model, where $(n_x, n_y) = (2, 2)$.

Acknowledgments. This research was partially supported by the Japan Science and Technology Agency's (JST) CREST program "Research and Development of Unified Environment on Accelerated Computing and Interconnection for

Post-Petascale Era." The authors would like to thank the Center for Computational Sciences, University of Tsukuba, for allowing them to use the HA-PACS system as part of the Interdisciplinary Collaborative Research Program.

References

1. Top500 Supercomputer Sites, http://top500.org/
2. PGI-SIG. PCI Express Base Specification, Rev. 3.0 (2010)
3. Clark, M.A., Babich, R., Barros, K., Brower, R.C., Rebbi, C.: Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. Comput. Phys. Commun. 181, 1517–1528 (2010)
4. QUDA - A Library for QCD on GPUs, http://lattice.github.io/quda/
5. Babich, R., Clark, M.A., Joo, B., Shi, G., Brower, R.C., Gottlieb, S.: Scaling lattice QCD beyond 100 GPUs. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (2011)
6. Message Passing Interface (MPI) Forum Home Page, http://www.mpi-forum.org/
7. Lattice QCD Message Passing (QMP), http://usqcd.jlab.org/usqcd-docs/qmp/
8. Hanawa, T., Kodama, Y., Boku, T., Sato, M.: Interconnect for Tightly Coupled Accelerators Architecture. In: IEEE 21st Annual Symposium on High-Performance Interconnects (HOT Interconnects 21), pp. 79–82 (2013)
9. Hanawa, T., Kodama, Y., Boku, T., Sato, M.: Tightly Coupled Accelerators Architecture for Minimizing Communication Latency among Accelerators. In: The Third International Workshop on Accelerators and Hybrid Exascale Systems, AsHES (2013)
10. PGI-SIG. PCI Express External Cabling Specification, Rev. 1.0 (2007)
11. NVIDIA GPUDirect — NVIDIA Developer Zone, https://developer.nvidia.com/gpudirect.
12. CUDA Toolkit, https://developer.nvidia.com/cuda-toolkit
13. Ammendola, R., Biagioni, A., Frezza, O., Lo, F.: APEnet+: high bandwidth 3D torus direct network for petaflops scale commodity clusters. J. Phys. Conf (2011)
14. Ammendola, R., Biagioni, A., Frezza, O., Lo, F.: APEnet+: a 3D Torus network optimized for GPU-based HPC Systems. J. Phys. Conf. (2012)
15. Mellanox Products: Mellanox OFED GPUDirect RDMA Beta, http://www.mellanox.com/page/products_dyn?product_family=116.
16. MVAPICH2, http://mvapich.cse.ohio-state.edu/overview/mvapich2/.
17. OpenMPI, http://www.open-mpi.org/.