Performance Analysis of Lattice QCD on GPUs in APGAS Programming Model (Unrefereed Workshop Manuscript)

Koichi Shirahata[1,a)] Jun Doi[2,b)] Mikio Takeuchi[2,c)]

Abstract: The APGAS programming model abstracts deep memory hierarchy such as distributed memory and GPU device memory by a global view of data and asynchronous operations on massively parallel computing environments. However, how much GPUs accelerate applications using the APGAS model remains unclear. In order to understand the effectiveness of using GPUs in the APGAS model, we give a comparative performance analysis of the APGAS model in X10 on GPUs with a standard massage passing model using lattice QCD. Our experimental results on TSUBAME2.5 show that our X10 CUDA implementation on 32 GPUs exhibits 19.4x speedup over X10 C++ on multi-core CPUs, and comparative performance with MPI CUDA in weak scaling. The results indicate that the APGAS programming model on GPUs scales well and accelerates the lattice QCD application significantly.

1. Introduction

It is expected that the first exascale supercomputer will be deployed within the next 10 years, but the programming model that allows us easy development and high performance is still unknown. Recent supercomputers deploy manycore accelerators such as GPUs in order to accelerate a wide range of applications. The Asynchronous Partitioned Global Address Space (APGAS) programming model abstracts deep memory hierarchy such as distributed memory and GPU device memory through the combination of a global view of data and asynchronous operations. The APGAS model offers a flexible way for a wide range of applications to express many patterns of concurrency, communication, and control for computing on massively parallel computing environments. The APGAS model is a possible programming model for computing on exascale supercomputers since the APGAS model can utilize multiple nodes as well as multiple GPUs with high productivity.

Although the APGAS model can utilize multiple GPUs, how much GPUs accelerate applications using APGAS model remains unclear. While the APGAS model allows us highly productive programming for massively parallel computing, the abstraction of deep memory hierarchy may limit performance since the memory abstraction limits domains of performance tuning. Moreover, when using multiple GPUs, the scalability of multiple GPUs in the APGAS model is also an open problem.

In order to address the problems, we give a comparative analysis of the APGAS model in X10 with a standard message passing model, by using lattice Quantum Chromodynamics (QCD) as an example, which is one of the most challenging applications for supercomputers. We further analyze the performance of lattice QCD in X10 on multiple GPUs. We firstly implement a CPU-based lattice QCD in X10 by fully porting a sequential CPU implementation in C into X10. Then we extend the X10 implementation into a multi-GPU-based implementation by implementing CUDA kernels and partitioning four-dimensional grid into multiple places in order to handle memories on multiple nodes and GPU device memory, where the place indicates a part of memory that corresponds to a host memory or a device memory on a compute node. We further apply several optimizations including data layout optimization for coalesced memory access on GPUs and communication overlapping using asynchronous memory copy functions in X10.

Our experimental results on TSUBAME2.5 show that our X10 implementation on multiple GPUs outperforms a X10 implementation on multi-core CPUs in both strong and weak scalabilities using multiple nodes. The strong scalability evaluation shows our X10 implementation on 16 GPUs exhibits 8.28x speedup, and the weak scalability evaluation shows our X10 implementation on 32 GPUs exhibits 19.4x speedup over X10 on multi-core CPUs. We also show that X10 CUDA exhibits comparative scalability with MPI CUDA in weak scaling. The results indicate that the APGAS programming model on GPUs scales well and accelerates the lattice QCD application significantly.

Here we describe a summary of our contributions:

- We describe an implementation of lattice QCD in X10 CUDA.
- We give a detailed performance analysis of X10 on GPUs.
- We reveal that X10 CUDA achieves significant speedup

[1] Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[2] IBM Research - Tokyo, Toyosu, Koto, Tokyo, 135–8511, Japan
[a)] koichi-s@matsulab.is.titech.ac.jp
[b)] doichan@jp.ibm.com
[c)] mtake@jp.ibm.com

from CPU-based implementations, and exhibits comparative performance with MPI CUDA in weak scaling.

2. Background

We explain APGAS programming model with X10 programming language. Then introduction of GPU computing is described. After that we introduce lattice QCD application, which is used as a benchmark application for large-scale computing environments.

2.1 APGAS Programming Model and X10

Partitioned Global Address Space (PGAS) model [1] is a programming model which virtualizes distributed memory as a global address space where a object can be placed over multiple locations on the distributed memory. There are several PGAS programming languages such as Co-Array Fortran [2] and Unified Parallel C [3].

APGAS (Asynchronous PGAS) programming model [4] is a PGAS model which enables dynamic task creation under programmer control. APGAS programming model mainly consists of two parts: *places* and *activities*. A *place* is simply coherent portion of the address space; a collection of data together with the activities that operate on that data. Places have a important property that they are not required to be single-threaded. That is, multiple activities may be active simultaneously in a single place. An `async` is the denotational mechanism to express activities that perform computation in a place. An activity is launched at a given place and stays at that place for its lifetime.

X10 [5], [6], [7] is a language which implements APGAS programming model. In X10, PGAS memories are called places, where each place is allocated on a process. Programmer controls places by moving to other place by using `at` statement. A new activity, which is allocated on the same place, is created dynamically by using `async` statement. There are language specific limitations on how activities reference remote data. In X10, an activity cannot access locations at remote places. If it desires to operate on remote locations it has access to, it must launch a new activity at that place. Activities may be used not just to run computations at a remote place but also to specify data-transfers such as array copies from an array at a place p to an array at a place q. `asyncCopy` conducts a place-to-place asynchronous data transfer. Activities in a place can be spawned locally or remotely. To control their execution, `finish` statement is introduced; a synchronization construct that allows a parent computation to wait for the completion of all its children activities. `finish` captures the very powerful notion of distributed termination detection [8].

2.2 GPU Computing in APGAS Programming Model

X10 also supports using accelerators such as GPU and FPGA. Cunningham et al. implements CUDA supports for X10 [9]. **Fig. 1** shows an overview of APGAS programming model with accelerators. As the same way as the APGAS model without accelerators, the APGAS model provides a global view of memory among multiple memories by using *places* and *activities* and the main activity can create another activity by `async` statement and can move to another place by `at` statement with the destination

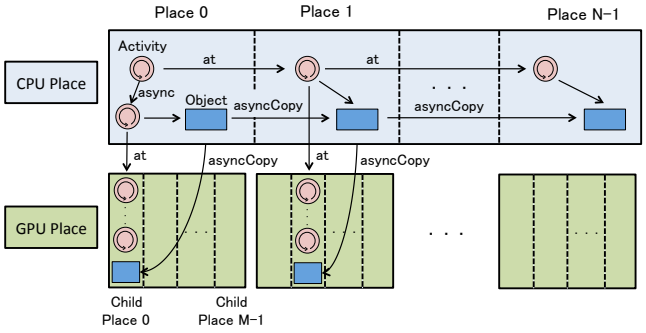

Fig. 1 APGAS programming model with accelerators.

place. When using accelerators, an accelerator is regarded as a child place, where a child place is bound to a place on CPUs. By moving to a child place by `at` statement, an activity can move to an accelerator. Then we can gain the massive parallelism of many core accelerators by creating as many numbers of activities as the number of threads the application requires. After that, we can execute operations on the accelerator by giving operations on the activities created on the accelerator. We can also copy objects on a place on a host memory to a place on a device memory by simply using `asyncCopy` statement in the same way as a host-to-host copy. During a copy between a place on a host memory and a place on a device memory, we can overlap computations on accelerators.

When using GPUs in X10, we can write X10 codes on GPUs similar to native CUDA codes. X10 CUDA supports most of the operations in CUDA such as allocating memory on GPUs, copying memory between CPU and GPU, invoking threads, blocks, and shared memory, barrier synchronization.

2.3 Lattice QCD Application

Lattice QCD [10] is a common technique to simulate a field theory of quantum chromodynamics (QCD) theory of quarks and gluons on 4D lattice consisting of 3D space and 1D time. The quark fields are placed on the sites of 4D lattice and the gauge fields are placed as the links of the lattice sites to represent the effect of the gluons as the transporters of the quark fields. The simulation of lattice QCD uses a finite difference method to solve the interactions.

Lattice QCD computation mainly consists of Monte-Carlo simulations on 4D lattice. The computation is dominated by solving a system of linear equations of matrix-vector multiplication using iterative methods, such as conjugate gradient (CG) method. The most computation and communication intensive procedure in lattice QCD is solving the following equation for Dirac matrix:

$$M(U)x = b \tag{1}$$

where M is the discretized Dirac operator which is a sparse matrix whose elements are a function of a background field U, and b and x are the source and solution vectors respectively. Wilson-Dirac operator is used to calculate the physical exchange between 4D lattice sites through the effects of the gluon fields , by multiplying spinor and gauge matrix on 8 neighbors of x, y, z, t dimensions with positive or negative signs. This problem accounts for the majority of operations in lattice QCD.

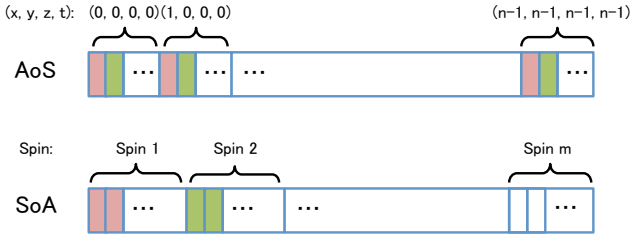

Fig. 2 Two data layouts: (Top) Array of Structure (AoS), and (Bottom) Structure of Array (SoA). The two different colors in elements indicate colors of spinors.

2.4 Implementation of Lattice QCD in X10 C++

In our earlier work, we implement a CPU-based lattice QCD in X10 by fully porting a sequential CPU-based implementation [11] in C++ [12] into X10 [13]. We applied multi-threading by using `async` statement to invoke multiple threads. Based on the single node implementation, we extended it to multiple nodes by using multiple places.

We also applied several optimizations including multi-dimensional communication overlapping among multiple places. As for the communication overlapping, the X10 C++ implementation overlaps between boundary exchanges and bulk computations. The communications are overlapped by using the `asyncCopy` method of the `Rail` class. The `asyncCopy` method create a new activity and transfers data asynchronously on the activity. While the boundary data is being transferred from the activity, the main activity continues bulk computation. After calling the bulk computation, the main activity waits the completion of the boundary exchange by using `finish` statement and barrier synchronization. We also applied other optimizations such as hybrid parallelization of invoking multiple threads and multiple places per node. Our previous experimental results showed that our X10 C++ implementation scales well at least up to 256 places.

3. Implementation of Lattice QCD in X10 CUDA

We describe how we extend the CPU-based X10 implementation which is introduced in section 2.4 into a multi-GPU-based X10 implementation.

3.1 Basic Idea

We extend a CPU-based lattice QCD in X10 into X10 CUDA. We port lattice QCD in X10 C++ into the one in X10 CUDA, by porting whole solvers into CUDA kernels in X10 and partitioning four-dimensional grid into multiple places in order to handle memories on multiple nodes and GPU device memory. As for porting X10 C++ into X10 CUDA, we port whole computation into CUDA kernels in X10, in order to avoid waste memory copy overheads between CPU and GPU except boundary data transfers. We basically port each compute kernel into CUDA kernel, including operations in the Wilson-Dirac operator and the other BLAS level 1 operations such as dot product and reduce operations. We also implement boundary data transfer between multiple GPUs. We extend our lattice QCD in X10 C++ for GPUs. We add memory copy operations from GPU to CPU before memory copy operations between CPU to CPU, and after that we also add memory copy operations from CPU to GPU. We apply overlapping technique between bulk computations on GPUs, memory copy operations between GPU and CPU as well as copy operations between CPU and CPU.

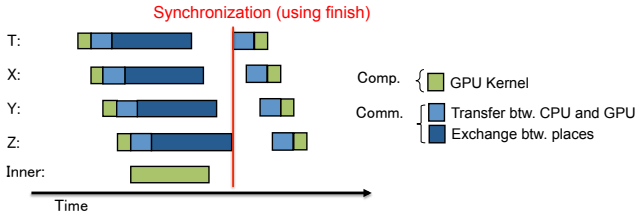

Fig. 3 Implementation of Wilson-Dirac operation of lattice QCD in X10 CUDA.

We apply several optimizations into lattice QCD in X10 CUDA, including data layout optimization for coalesced memory access on GPUs and communication overlapping using asynchronous memory copy functions in X10. The detail of the overlapping technique as well as other optimizations are explained in section 3.2 and section 3.3.

3.2 Data Layout Optimization

The first optimization for X10 CUDA is data layout optimization. **Fig. 2** shows two data layouts of spinors (elements of quarks). The top bar indicates AoS (Array of Structure) and SoA (Structure of Array) layouts. In our CPU-based lattice QCD, the data layout is based on AoS, which is suitable for running on multi-core CPU. However, the AoS data layout is not suitable in the GPU case, since GPU is suitable for coalesced memory access while the AoS data layout is non-contiguous data. In order to enable coalesced memory access, we translate from AoS to SoA, which is consisting of contiguous data.

3.3 Communication Overlapping in X10 CUDA

The second optimization is communication optimization. We apply two communication optimizations; multi-dimensional partitioning and communication overlapping. We apply the multi-dimensional partitioning in the similar way as our CPU-based lattice QCD in X10. In the GPU case, the difference from the CPU case is that we overlap memory copy operations between GPU and CPU in addition to memory copy operations between CPU and CPU. The overview of our overlapping technique is described in **Fig. 3**. X-axis indicates time and y-axis indicates the four dimensions of the lattice. Green bars indicate GPU kernels, light blue bars indicate memory copy between CPU and GPU, and dark blue bars indicate data transfer between places on CPUs. The red vertical line indicates the synchronization point by using the `finish` statement on a place where a place waits completion of all activities (i.e. eight activities in each dimension with positive and negative signs in this case) invoked by the place.

Fig. 4 shows the pseudo code of an implementation of Wilson-Dirac operator in X10 CUDA. The implementation applies Put-wise data transfer in the same way as the X10 C++ implementation [13]. We extend the X10 C++ implementation to X10 CUDA by adding copy operations between CPU and GPU and implementing CUDA kernels in X10. **Fig. 5** shows the copy method

```
def Dopr_Put(dv : CUDAWilsonVectorField,
             du : CUDASU3MatrixField,
             dw : CUDAWilsonVectorField,
             cks : Double,
             bx : CUDALatticeComm) {
  // make sure all places finished prev. operation
  Team.WORLD.barrier();
  finish {
    // compute in T plus direction
    val iTP = bx.neighbors()(bx.TP);
    // move to remote place to recv. boundary data
    at (Place(iTP)) async {
      MakeTPBndKernel(bx.dSendBuf(bx.TP), dw);
       // copy from remote device to local host
       // via remote host
      bx.SendBndDevicetoRemoteHost(bx.TP);
    }
    // compute in T minus direction
    val iTM = bx.neighbors()(bx.TM);
    at (Place(iTM)) async {
      MakeTMBndKernel(bx.dSendBuf(bx.TM), dw, du);
      bx.SendBndDevicetoRemoteHost(bx.TM);
    }
    ...  // compute in the other directions
    dv.Copy(dv, dw);
    // bulk computation
    MultKernel(dv, du, dw, -cks);
  } // wait copy to local host and local comp.
  // copy from local host to local device
  bx.RecvBndHostToDevice();
  //set boundary part
  SetTPBndKernel(bx.dRecvBuf(bx.TP), dv, du, -cks);
  SetTMBndKernel(bx.dRecvBuf(bx.TP), dv, -cks);
  ...
}
```

Fig. 4 Pseudo source code of Wilson-Dirac operator using GPUs

```
def SendBndDevicetoRemoteHost(dir : Long) {
  val size = hSendBuf(dir).size;
  // copy from device to host
  finish {
    Rail.asyncCopy(dSendBuf(dir), 0,
                   hSendBuf(dir), 0, Size(dir));
  }
  // copy from host to remote host
  Rail.asyncCopy(hSendBuf(dir), 0,
                 hRemoteRecvBuf(dir), 0, Size(dir));
}

def RecvBndHostToDevice() {
  // copy from host to device
  finish {
    Rail.asyncCopy(hRecvBuf(TP), 0,
                   dRecvBuf(TP), 0, Size(TP));
    Rail.asyncCopy(hRecvBuf(TM), 0,
                   dRecvBuf(TM), 0, Size(TM));
    ... // copy in the other directions
}}
```

Fig. 5 Pseudo source code of data transfer between CPU and GPU

which copies data from a device to remote host through a host which has the device. The method first copies data from a device to a host using the `asyncCopy` method. Then the data on the host is copied to remote host using `asyncCopy` again.

A drawback of this implementation is that the domain of communication overlapping is limited by `finish`-based synchronization. We can further overlap in the case of MPI, since MPI has a feature to call one-to-one synchronization by using `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait`. We also tried one-to-one synchronization in X10 by avoiding `finish` synchronization by using callback function, which is called when the copy using the `uncountedCopy` method is complete. However, we observe poor performance and deadlock when using multiple places in this implementation. As for further communication optimization in CUDA, we can further overlap communication and computation by using multiple CUDA streams. However, we could not apply this optimization, since the current version of X10 statically allocate a single CUDA stream for each computation and communication and does not provide a feature to dynamically invoke multiple CUDA streams.

4. Evaluation

In order to understand the effectiveness of using GPUs in X10, we evaluate the performance of lattice QCD on GPUs in X10. The objective of the evaluation is to understand how much GPUs can accelerate applications in the APGAS programming model. We compare the performance of lattice QCD in X10 CUDA with in CPU-based implementations in X10 and MPI C as well as with a GPU-based MPI implementation. Also, we conduct scalability study on multiple GPUs to see the efficiency of using multiple GPUs in the APGAS programming model. We conduct both weak and strong scalability studies, as well as detailed analysis including performance breakdown and comparison of productivity with MPI CUDA.

We use the TSUBAME2.5 supercomputer [14] located at Tokyo Insitute of Technology for the performance experiments. TSUBAME2.5 mainly consists of 1408 compute nodes, each of which has 2 sockets of Intel Xeon X5670 (Westmere EP, 2.93GHz, 6 cores) CPU, 54GB of DDR3 main memory, 3 devices of NVIDIA Tesla K20X GPU with 6GB of discrete GDDR5 memory connected to PCI-Express 2.0 × 16 buses, and 2 cards of QDR InfiniBand HBA (40Gbps each) connected to the dual rail interconnect network with full bisection fat tree, and runs on SUSE Linux Enterprise 11 SP3. We use up to 32 compute nodes of TSUBAME2.5 in the experiments. We use X10 version 2.4.3.2, CUDA version 6.0, GCC version 4.3.4 and OpenMPI version 1.6.5.

As for experimental configurations, We use one place per node, one GPU per place in X10 CUDA, and 12 threads per place in X10 C++ and in MPI C. Note that here we use one GPU per node, since we observe using multiple child places on a node causes deadlock in the current version of X10. We measure average iteration time of one convergence of the CG solver. We observe one convergence typically includes 300 to 400 iterations. As for lattice QCD in X10 CUDA, we measured two precisions; single precision (X10 CUDA SP) and double precision (X10 CUDA DP). Note that we use one dimensional partitioning in the strong scalability study while we use four dimensional partitioning in the weak scalability study, since we observe one dimensional partitioning exhibits better strong scalability than multi-dimensional partitioning while we observe good weak scalability using four dimensional partitioning.

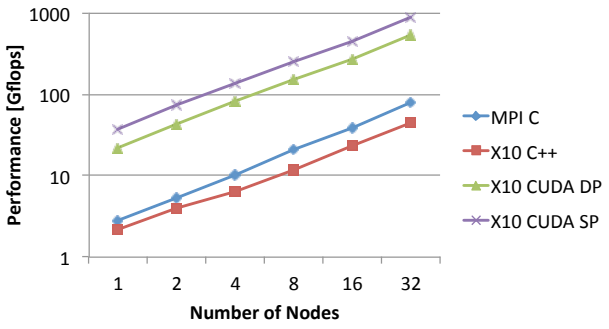

Fig. 6 Performance of weak scaling using the problem size (x, y, z, t) = (24, 24, 24, 96) per place on TSUBAME2.5.

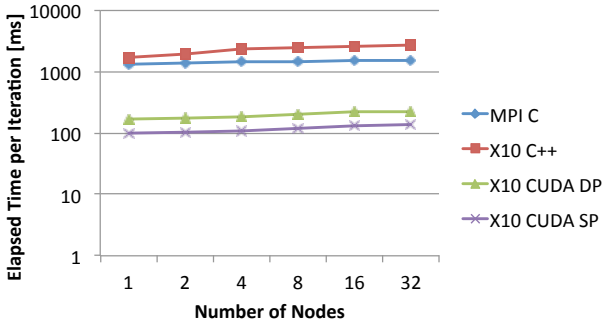

Fig. 7 Elapsed time of weak scaling using the problem size (x, y, z, t) = (24, 24, 24, 96) per place on TSUBAME2.5.

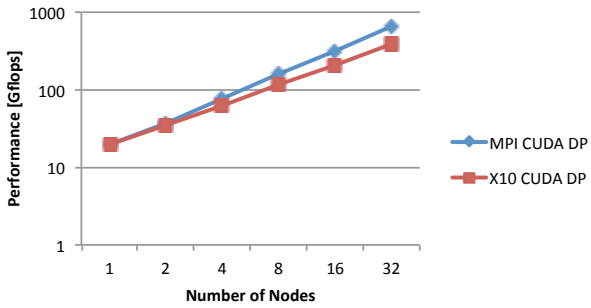

Fig. 8 Performance of weak scaling using the problem size (x, y, z, t) = (24, 24, 24, 24) per place on TSUBAME-KFC.

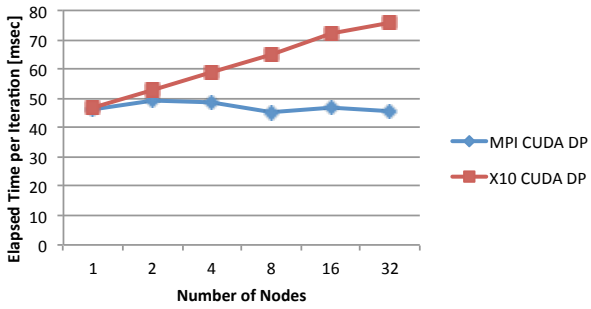

Fig. 9 Elapsed time of weak scaling using the problem size (x, y, z, t) = (24, 24, 24, 24) per place on TSUBAME-KFC.

4.1 Weak Scaling

We conduct weak scalability study in X10 CUDA. Firstly we compare weak scaling in X10 CUDA with CPU-based implementations using MPI or X10 on up to 32 nodes of TSUBAME2.5. We use the problem size (x, y, z, t) = (24, 24, 24, 96) per place by increasing problem size in proportional to the total number of places to use. **Fig. 6** shows the results of performance, where x-axis indicates the number of places and y-axis indicates average performance of one GC iteration in Gigaflops. **Fig. 7** shows the results of elapsed time, where x-axis indicates the number of places and y-axis indicates average elapsed time of one CG iteration in milliseconds. The results show that lattice QCD in X10 CUDA outperforms both X10 C++ and MPI C. X10 CUDA performs 19.4x and 11.0x faster than X10 C++ and MPI C respectively on 32 GPUs. The results also show that X10 CUDA exhibits good scalability at least up to 32 GPUs. This behavior indicates that X10 CUDA does not incur significant communicational penalty when the amount of computation is sufficient for hiding communication.

We also conduct comparative weak scaling experiments of our X10 CUDA implementation with a MPI CUDA implementation on up to 32 nodes of TSUBAME-KFC. Each compute of TSUBAME-KFC consists of 2 sockets of Intel Xeon E5-2620 v2 (Ivy Bridge EP, 2.10GHz, 6 cores) CPU, 64GB of DDR3 main memory, 4 devices of NVIDIA Tesla K20X GPU with 6GB of discrete GDDR5 memory connected to PCI-Express 2.0 × 16 buses, and 1 card of FDR InfiniBand HBA (56Gbps) connected to a single rail interconnect network, and runs on CentOS release 6.4. We use Open MPI 1.7.2 with GNU GCC 4.4.7 for the MPI implementation, and CUDA driver 5.5 and CUDA runtime 5.5 for the GPU implementation. We use X10 2.5.1 for experiments on TSUBAME-KFC. We use the problem size (x, y, z, t) = (24, 24, 24, 24) per place by increasing problem size in proportional to the total number of places to use. **Fig. 8** shows the results of performance, where x-axis indicates the number of places and y-axis indicates average performance of one GC iteration in Gigaflops. **Fig. 9** shows the results of elapsed time, where x-axis indicates the number of places and y-axis indicates average elapsed time of one CG iteration in milliseconds. The results show that lattice QCD in X10 CUDA exhibits similar weak scaling with MPI CUDA in Fig. 8. The results also show that elapsed time in X10 CUDA increases as the number of nodes increases in Fig. 9. The result of X10 CUDA on 32 nodes takes 1.63x longer time compared with that on 1 node. This elapsed time increase indicates X10 CUDA suffers significant overhead using multiple nodes.

4.2 Strong Scaling

We also conduct strong scalability study in X10 CUDA. We compare the performance in X10 CUDA with X10 C++ and MPI C using the problem size (x, y, z, t) = (24, 24, 24, 96), whose data size is the largest size to fit on one Tesla K20X GPU.

Fig. 10 shows the results of performance, where x-axis indicates the number of places and y-axis indicates average performance of one CG iteration in Gigaflops. The results show that lattice QCD in X10 CUDA outperforms both X10 C++ and MPI C. X10 CUDA is 4.57x and 8.28x faster than MPI C and X10 C++ respectively on 16 GPUs. A reason for this speedup is that computational kernels of the Wilson-Dirac operation and the BLAS level 1 operations are highly accelerated by using GPUs. As for multi-GPU scalability, X10 CUDA scales well on small number

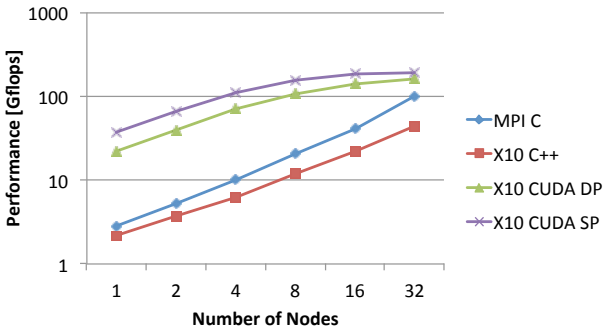

Fig. 10 Performance of strong scaling using the problem size (x, y, z, t) = (24, 24, 24, 96) on TSUBAME2.5.

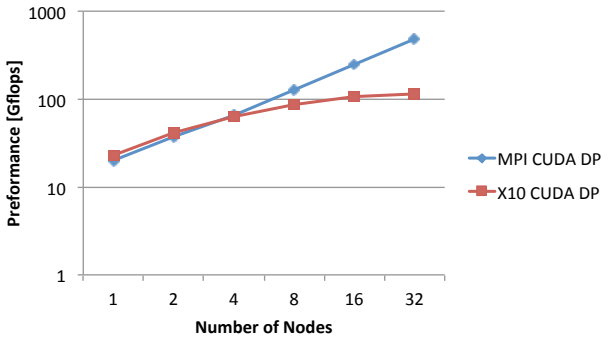

Fig. 11 Performance of strong scaling using the problem size (x, y, z, t) = (24, 24, 24, 96) on TSUBAME-KFC.

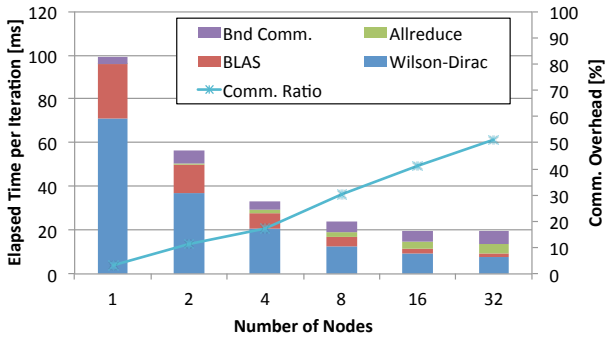

Fig. 12 Breakdown of strong scaling in X10 CUDA using the problem size (x, y, z, t) = (24, 24, 24, 96) on TSUBAME2.5.

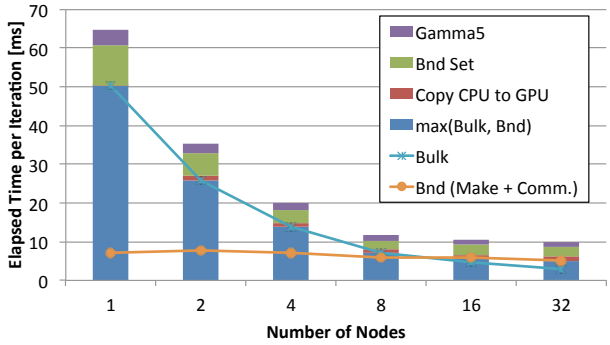

Fig. 13 Breakdown of Wilson-Dirac strong scaling in X10 CUDA using the problem size (x, y, z, t) = (24, 24, 24, 96) on TSUBAME2.5.

of GPUs. However, the results also show that the scalability of X10 CUDA gets smaller as the number of places increases. A possible reason for this degradation is that communicational time increases and computational time decreases since problem size on a GPU decreases as the number of places increases.

We also compare strong scalability with MPI CUDA on TSUBAME-KFC. **Fig. 11** shows the comparative results. X10 CUDA exhibits comparative performance up to 4 nodes, while exhibits performance degradation from 8 nodes. The results indicate that the X10 CUDA implementation suffers heavy overheads derived from X10 CUDA. We consider two possibilities of overheads: communication overhead and overhead of X10 CUDA on one GPU. As for the communication overhead, ratio of communication cost increases while computational cost decreases when using larger number of GPUs, since problem size for each GPU decreases while the number of neighbor GPUs to transfer boundary data increases. Besides, the X10 CUDA implementation includes less overlapping of boundary data exchange and inner kernel computation of Wilson-Dirac operation compared with the MPI-based implementation. As for the overhead of X10 CUDA on one GPU, overhead of X10 CUDA may increase due to short computation time on smaller problem size when increasing the number of nodes. We further investigate the two possible overheads in the following of this section and section 4.3.

We investigate **Fig. 12** shows performance breakdown in X10 CUDA using the same problem size in single precision, where x-axis indicates the number of places and y-axis indicates average elapsed time of each part. We divide total elapsed time of lattice QCD into four parts; computational time of Wilson-Dirac operation, computational time of BLAS level 1 operations, communicational time of boundary exchange between GPU and GPU (including communications from GPU to CPU, from CPU to CPU, and from CPU to GPU), and communicational time of the MPI Allreduce operation. The figure also indicates the ratio of communication overhead over total time of lattice QCD computation. The results show that both computational parts of Wilson-Dirac and BLAS scales well. However, the results also show that communicational overhead increases significantly when increasing the number of places, in both boundary exchange and MPI Allreduce.

Fig. 13 shows further breakdown of the Wilson-Dirac operation, where x-axis indicates the number of places and y-axis indicates average elapsed time of each part in Wilson-Dirac. The blue bars indicate longer elapsed time between bulk computation and the sum of boundary creation and boundary data transfer. Actually elapsed time is the longer time between them since these bulk and boundary operations are synchronized by `finish` statement as illustrated in Fig. 3. The light blue and yellow lines indicates elapsed time of the bulk and boundary operations each other. The figure shows that these two lines cross over when the number of places is increased from 8 to 16. This result indicates that communication becomes dominant when using more than 16 places. This crossover results in a cause of the limit of strong scaling. Possible ways to improve the scalability include applying one-to-one synchronization in X10 or improve communication and synchronization operations in the X10 runtime. As for MPI Allreduce, we simply use a collective API in X10 called Team, so decreasing the overhead of Team collective API itself is

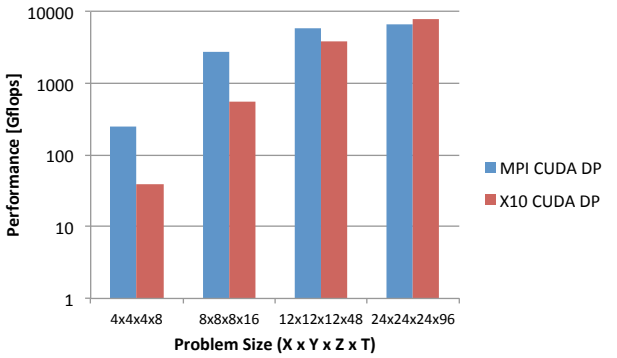

Fig. 14 Performance of data size scaling on a single node of TSUBAME-KFC.

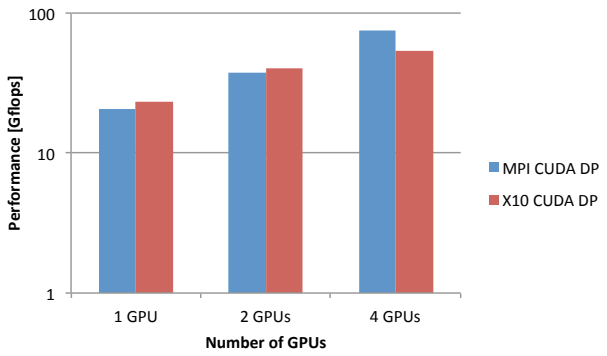

Fig. 15 Performance of multi-GPU scaling using the problem size (x, y, z, t) = (24, 24, 24, 96) on a single node of TSUBAME-KFC.

required to improve strong scalability.

4.3 Performance on Different Problem Sizes

In order to investigate overheads of the strong scaling performance of X10 CUDA, we conduct performance experiments on different problem sizes on a single node of TSUBAME-KFC. **Fig. 14** shows comparative performance on four different problem sizes: (x, y, z, t) = (4, 4, 4, 8), (8, 8, 8, 16), (12, 12, 12, 48), and (24, 24, 24, 96). The results show that X10 CUDA exhibits similar performance with MPI CUDA on larger problem sizes. However, X10 CUDA exhibits significant performance degradation when we use smaller problem sizes; X10 CUDA exhibits 6.02x slower performance than MPI CUDA on problem size (4, 4, 4, 8). These results indicate that the X10 CUDA implementation suffers additional overhead derives from the X10 CUDA runtime. We consider this additional overhead is a possible cause of the limit of strong scaling in X10 CUDA.

4.4 Multi-GPU Scaling

We also conduct experiments of multi-GPU scalability on a single node of TSUBAME-KFC. We use problem size (x, y, z, t) = (24, 24, 24, 96) and use up to 4 GPUs on a single node of TSUBAME-KFC. When we use multiple GPUs, we invoke multiple processes such that the number of processes is equal to the number of GPUs to use, and assign one GPU per process. **Fig. 15** shows the results of strong scaling performance on multiple GPUs on a single node. The results indicate the X10 CUDA implementation performs comparatively with the MPI CUDA implementation on up to 2 GPUs. Although the X10 CUDA implementation also improves performance using 4 GPUs, MPI CUDA performs 1.39x faster than X10 CUDA. We consider this performance gap derives from the fact that X10 CUDA suffers additional overhead on smaller problem size as we see in section 4.3.

4.5 Comparison of Productivity

We also conduct comparative analysis of productivity in terms of the number of lines of code and compilation time of source code between MPI CUDA and X10 CUDA. Firstly, **Table 1** shows the number of lines of code of the MPI CUDA and X10 CUDA implementations. The results show that the X10 CUDA implementation contains 4.10x and 1.92x larger number of lines in the Wilson-Dirac operation and in total respectively compared with MPI CUDA. This significant increase of lines of code in X10 CUDA derives from the fact that currently X10 CUDA does not support calling CUDA device functions inside CUDA global functions. These limitations result in multiple copies of inlined operations on each direction of operations such as Wilson-Dirac operation.

Secondly, **Table 2** shows the compilation time of the MPI CUDA and X10 CUDA implementations. The results show that the X10 CUDA implementation takes 11.3x longer compilation time than the MPI CUDA implementation.

4.6 Discussion

We see advantages and disadvantages of X10 CUDA from the performance experiments. In terms of the advantages, we could extend our X10 code into X10 CUDA in a straightforward manner; simply porting computation kernels into CUDA and adding memory copy operations between CPU and GPU. We see that our X10 CUDA implementation exhibits speedup over X10 C++ and MPI C implementations. We also see good scalability of X10 CUDA on multiple GPUs. On the other hand, we also see drawbacks of X10 CUDA, in both the productivity and performance perspectives. In terms of productivity, we observe some limitations including that we could not call CUDA methods inside a CUDA place, which caused the increase of the lines of code. In terms of performance, the current version of X10 CUDA has some limitations preventing performance tuning. Currently X10 CUDA does not support creating CUDA streams, which enables GPUs overlap multiple CUDA kernels and multiple memory copies between CPU and GPU. We also see that the strong scalability in X10 CUDA is limited when increasing the number of GPUs. The implementation of communication overlapping based on `finish` statement has overhead of waiting all dimensions at the same time, while MPI can wait each dimension separately. We consider the reason why this scalability limitation is revealed is that the computational kernels are highly accelerated by using GPUs.

5. Related Work

There has been a lot of efforts on high performance large-scale lattice QCD implementations. Doi et al. work on a peta-scale lattice QCD implementation [15] on Blue Gene/Q supercomputer [16]. The implementation fully optimizes overlapping com-

Table 1 Comparison of the number of lines of code of Wilson-Dirac operation and the total.

	MPI CUDA	X10 CUDA
Wilson-Dirac	1590	6512
Total	4667	8942

munication by computation. The implementation also applies node-mapping optimizations for fully utilizing network topology on Blue Gene/Q. In our earlier work, we compare the scalability of our lattice QCD implementation in X10 with in MPI [13]. Our implementation in MPI also applies overlapping technique in the same way as this work.

There are several efforts on accelerating lattice QCD using many core accelerators such as GPUs and Xeon Phi. Clark et al. implemented lattice QCD in CUDA called QUDA and applied optimizations [17]. Babich et al. extended lattice QCD in CUDA onto multiple GPUs [18]. Their implementation invokes multiple CUDA streams so that multiple kernels and memory copies between CPU and GPU can be overlapped. We also apply similar optimizations as their work including data ordering and overlapping of computation and communication. However, our X10 implementation has less overlapping domains since we see that currently X10 could not invoke multiple CUDA streams. Joó el al. optimized lattice QCD for Intel Xeon Phi [19].

There is also work about PGAS language extension for multi-node GPU clusters [20]. They extended the XcalableMP PGAS language [21] for GPU and demonstrated their N-body implementation scales well. However, they did not compare their performance with neither their CPU-based PGAS implementation nor a MPI-based implementation. We compare the performance of X10 CUDA with MPI C and X10 C++ and revealed that X10 CUDA accelerates the CPU-based implementations significantly.

6. Conclusions

We give a comparative analysis of X10 on GPUs with X10 on CPUs and a standard message passing model, by using lattice Quantum Chromodynamics (QCD) as an example, which is one of the most challenging applications for supercomputers. We implement lattice QCD in X10 CUDA and apply several optimizations including data layout optimization for coalesced memory access on GPUs and communication overlapping using asynchronous memory copy functions in X10. Our experimental results on TSUBAME2.5 show that our X10 implementation on multiple GPUs outperforms a MPI implementation on multi-core CPUs using multiple nodes. The weak scalability evaluation also shows our X10 implementation on 32 GPUs exhibits 19.4x speedup over X10 on multi-core CPUs. The results also show that X10 CUDA exhibits comparative performance with MPI CUDA in weak scaling. The results indicate that the APGAS programming model on GPUs scales well and accelerates the lattice QCD application significantly.

Future work includes improving the performance of lattice QCD in X10 CUDA; tuning single GPU performance in X10 CUDA and improving strong scalability of multiple GPUs. We also plan to conduct experiments on large number of GPUs.

Acknowledgments This work was partially supported by JSPS KAKENHI Grant Number 26011503.

Table 2 Compilation of compilation time of MPI CUDA and X10 CUDA implementations in seconds.

	MPI CUDA	X10 CUDA
Compilation Time [sec]	15.19	171.50

References

[1] Almasi, G.: PGAS (Partitioned Global Address Space) Languages, *Encyclopedia of Parallel Computing*, Springer, pp. 1539–1545 (2011).
[2] Numrich, R. W. and Reid, J.: Co-Array Fortran for parallel programming, *ACM Sigplan Fortran Forum*, Vol. 17, No. 2, ACM, pp. 1–31 (1998).
[3] El-Ghazawi, T. and Smith, L.: UPC: unified parallel C, *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, p. 27 (2006).
[4] Saraswat, V., Almasi, G., Bikshandi, G., Cascaval, C., Cunningham, D., Grove, D., Kodali, S., Peshansky, I. and Tardieu, O.: The Asynchronous Partitioned Global Address Space Model, *The First Workshop on Advances in Message Passing (co-located with PLDI 2010)* (2010).
[5] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. and Grove, D.: X10 Language Specification Version 2.4 (2014).
[6] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Praun, C. and Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing., *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)* (2005).
[7] IBM Research: X10: Performance and Productivity at Scale., `http://x10-lang.org/`.
[8] Saraswat, V. A., Kambadur, P., Kodali, S., Grove, D. and Krishnamoorthy, S.: Lifeline-based global load balancing, *ACM SIGPLAN Notices*, Vol. 46, No. 8, ACM, pp. 201–212 (2011).
[9] Cunningham, D., Bordawekar, R. and Saraswat, V.: GPU programming in a high level language: compiling X10 to CUDA, *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, ACM, p. 8 (2011).
[10] Gupta, R.: Introduction to Lattice QCD (1997).
[11] Ueda, S., Aoki, S., Aoyama, T., Kanaya, K., Matsufuru, H., Motoki, S., Namekawa, Y., Nemura, H., Taniguchi, Y. and Ukita, N.: Bridge++: an object-oriented C++ code for lattice simulations, *31st International Symposium on Lattice Field Theory - LATTICE 2013* (2013).
[12] Stroustrup, B.: The C++ Programming Language, Fourth Edition (2013).
[13] Shirahata, K., Doi, J. and Takeuchi, M.: Performance Analysis of Lattice QCD Application with APGAS Programming Model, *The 2014 X10 Workshop (X10'14) co-located with PLDI'14* (2014).
[14] Matsuoka, S.: The TSUBAME2.5 Evolution, *Tsubame e-Science Journal*, Vol. 10, pp. 2 – 8 (online), available from ⟨http://www.gsic.titech.ac.jp/en/TSUBAME_ESJ⟩ (2013).
[15] Doi, J.: Peta-scale lattice quantum chromodynamics on a Blue Gene/Q supercomputer, *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, IEEE, pp. 1–10 (2012).
[16] IBM Systems and Technology Group: IBM Blue Gene/Q, `http://www.ibm.com/systems/technicalcomputing/solutions/bluegene/`.
[17] Clark, M. A., Babich, R., Barros, K., Brower, R. C. and Rebbi, C.: Solving Lattice QCD systems of equations using mixed precision solvers on GPUs, *Computer Physics Communications*, Vol. 181, No. 9, pp. 1517–1528 (2010).
[18] Babich, R., Clark, M. A., Joó, B., Shi, G., Brower, R. C. and Gottlieb, S.: Scaling Lattice QCD Beyond 100 GPUs, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 70:1–70:11 (2011).
[19] Joó, B., Kalamkar, D. D., Vaidyanathan, K., Smelyanskiy, M., Pamnany, K., Lee, V. W., Dubey, P. and III, W. W.: Lattice QCD on IntelR Xeon Phi TM Coprocessors, *Lecture Notes on Computer Science*, Vol. 7905, pp. 40–54 (2013).
[20] Lee, J., Tran, M. T., Odajima, T., Boku, T. and Sato, M.: An extension of XcalableMP PGAS lanaguage for multi-node GPU clusters, *Euro-Par 2011: Parallel Processing Workshops*, Springer, pp. 429–439 (2012).
[21] Lee, J. and Sato, M.: Implementation and performance evaluation of XcalableMP: A parallel programming language for distributed memory systems, *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, IEEE, pp. 413–420 (2010).