An Extension of XcalableMP PGAS Lanaguage for Multi-node GPU Clusters

Jinpil Lee[1], Minh Tuan Tran[1], Tetsuya Odajima[1], Taisuke Boku[1,2], and Mitsuhisa Sato[1,2]

[1] Graduate School of Systems and Information Engineering, University of Tsukuba
[2] Center for Computational Sciences, University of Tsukuba

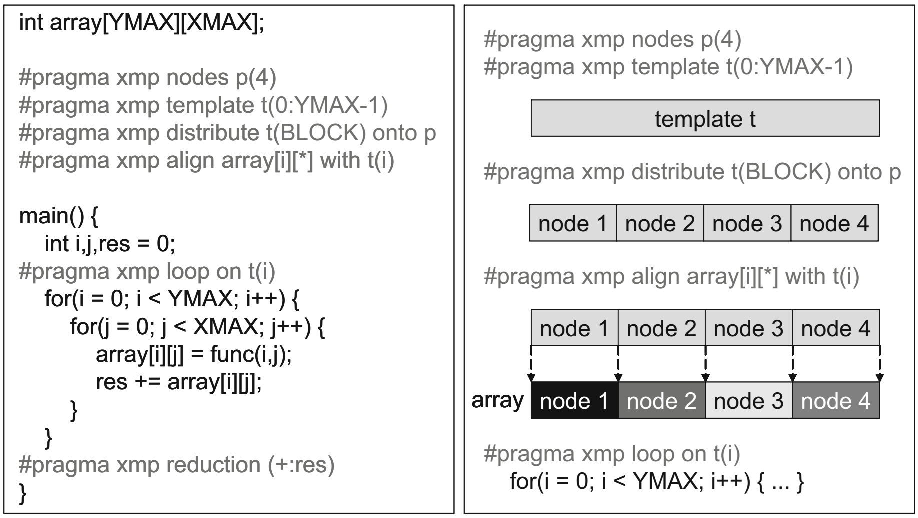
Abstract. A GPU is a promising device for further increasing computing performance in high performance computing field. Currently, many programming langauges are proposed for the GPU offloaded from the host, as well as CUDA. However, parallel programming with a multi-node GPU cluster, where each node has one or more GPUs, is a hard work. Users have to describe multi-level parallelism, both between nodes and within the GPU using MPI and a GPGPU language like CUDA. In this paper, we will propose a parallel programming language targeting multi-node GPU clusters. We extend XcalableMP, a parallel PGAS (Partitioned Global Address Space) programming language for PC clusters, to provide a productive parallel programming model for multi-node GPU clusters. Our performance evaluation with the N-body problem demonstrated that not only does our model achieve scalable performance, but it also increases productivity since it only requires small modifications to the serial code.

1 Introduction

GPGPU is becoming a popular research topic in High Performance Computing area. GPU vendors provide programming models for GPU computing. For example, NVIDIA provides CUDA, an extension of C, C++ and Fortran, which provides GPU data and threads management functions. Because CUDA only provides a primitive interface to control the GPU, GPU parallelization is often hard and time-consuming work. When using GPU clusters, the problem is getting worse because the user also needs to consider data distribution and inter-node communication using MPI, which also provides very primitive user APIs.

In this paper, we are proposing a parallel programming language called XcalableMP-ACC (XMP-ACC in short, ACC stands for ACCelerator). Following are the features of XMP-ACC.

- XMP-ACC is a GPGPU extension of XcalableMP[1] (XMP in short), a directive-based parallel programming language for PC clusters. It extends C language with new directives for GPU computing.
- XMP-ACC is targetting multi-node GPU clusters, where each node has one or more GPUs. So it can be used not only for not only a single GPU but also multiple GPU environment such as GPU clusters.



```
int array[YMAX][XMAX];

#pragma xmp nodes p(4)
#pragma xmp template t(0:YMAX-1)
#pragma xmp distribute t(BLOCK) onto p
#pragma xmp align array[i][*] with t(i)

main() {
   int i,j,res = 0;
#pragma xmp loop on t(i)
   for(i = 0; i < YMAX; i++) {
      for(j = 0; j < XMAX; j++) {
         array[i][j] = func(i,j);
         res += array[i][j];
      }
   }
#pragma xmp reduction (+:res)
}
```

#pragma xmp nodes p(4)
#pragma xmp template t(0:YMAX-1)
template t
#pragma xmp distribute t(BLOCK) onto p
node 1 node 2 node 3 node 4
#pragma xmp align array[i][*] with t(i)
node 1 node 2 node 3 node 4
array node 1 node 2 node 3 node 4
#pragma xmp loop on t(i)
for(i = 0; i < YMAX; i++) { ... }

Fig. 1. Data Parallelization in Global View Model

- XMP-ACC provides directives to describe typical processes for GPU computing like data allocation and loop parallelization on GPUs. XMP-ACC's directive-based programming model requires few modifications from a serial code. Users can exploit GPU performance with high productivity.
- Data distribution and inter-node communication for multi-node GPU environment is taken on by XMP directives in XMP-ACC. Users can do hybrid parallel programming on multi-node GPU clusters with little effort using XMP and XMP-ACC directives.

In section 2, we will give a brief overview of XcalableMP. In section 3, we will introduce new directives to describe GPGPU within the XcalableMP framework. Section 4 will show our implementation of the XMP-ACC compiler, and section 5 will shows the performance achieved by the compiler. We will show related work in section 6 and then, conclude the paper in section 7.

2 Overview of XcalableMP

Like OpenMP[2], XMP supports typical parallelization methods based on the data/task parallel paradigm under the **global view** model, and enables parallelizing the original sequential code with minimal modification using simple description. In this section, we will give a brief overview of XMP directives.

2.1 Execution Model

Like MPI, The basic execution model of XMP is the SPMD (Single Program Multiple Data) model. An XMP process begins its execution with a single thread

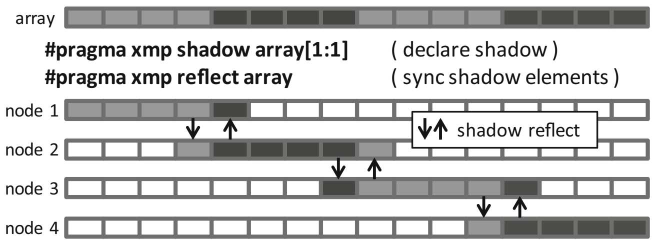


Fig. 2. Shadow Reflection

on each node, which is equivalent to a single-threaded MPI process. Because of its explicit parallelism design, memory access is always local, which means the compiler does not insert any automatic communications. To access the correct data during parallel execution, users should synchronize the local buffer using inter-node communication, which can be described by XMP directives.

2.2 XcalableMP Directives

The global view programming model provides a simple way to describe a parallel program starting from the sequential version: the user parallelizes it by adding directives incrementally. Because these directives can be treated as comments by sequential compilers of the base languages (C and Fortran), an XMP program derived from a sequential program can preserve the integrity of the original program when it is run sequentially.

Figure 1 shows a global view style code segment in XMP. The global view model shares major concepts with High Performance Fortran[3]. The programmer describes the data distribution of data shared among nodes by data distribution directives. The **node** directive declares a node set executing a XMP program, so the sample code would be executed on 4 nodes.

Data Distribution Using Templates. A template, a dummy array indicating data index space, is declared (via the **template** directive) and distributed onto nodes (via the **distribute** directive). In the sample code, a 1-dimensional template, t, is block distributed onto 4 nodes. Array distribution is declared by aligning the array to a template using the **align** directive. In the sample code, array $array[i]$ is aligned to template $t(i)$, that is, $array[i]$ will be allocated on the owner node of $t(i)$.

Work-Sharing. The **loop** directive splits up loop iterations among the executing nodes. The data accessed in a loop statement should be allocated in local memory, because communication is explicit in XMP, that is, work-sharing and data distribution should be done in the same way. A template can be used in the **loop** directive to specify the data allocation. In the sample code, template t is used for parallelizing the loop statement. Consequently, the local part of the distributed array would be processed on each node.

```
#pragma xmp align [i] with t(i) :: a, b
#pragma xmp shadow a[*]
int a[N], b[N];
void main(void) { . . .
  int i, x = 0;
#pragma xmp loop on t(i)
  for (i = 0; i < N; i++)
    a[i] = i;
#pragma xmp reflect a
#pragma xmp acc replicate (a, b)
{
```

```
#pragma xmp acc replicate_sync in (a)
#pragma xmp loop on t(i) acc
  for (i = 0; i < N; i++) {
    b[i] = x;
    for (int j = 0; j < N; j++) {
      b[i] += a[j];
    }
  }
#pragma xmp acc replicate_sync out (b)
} // #pragma xmp acc replicate
} // main()
```

Fig. 3. Sample Code of XcalableMP-ACC (sample.c)

Directives for Inter-node Communication. The XMP specification guarantees that communication takes place only when communication is explicitly specified. In the global view model, communication directives are used to synchronize and keep the data consistent among the executing nodes.

When an array is distributed, referencing the neighbor elements of the local block is a very typical access pattern that results in inter-node communication. To access the neighbor elements, we need to extend the local block because all memory access is local in XMP. We call the extended area a *shadow* of the array. Fig. 2 shows the shadow area of the array *array*. The **shadow** directive states that the size(the number of elements) of the shadow area on the array is 1 at both the lower and upper sides. A shadow is just a local memory buffer. To get the correct value of the neighbor elements, the data must be synchronized among the executing nodes. The **reflect** directive invokes inter-node communication, copying the original data to the shadow area. XMP also provides communication directives for barrier, reduction and broadcast communication which are commonly used functions in MPI.

3 Language Extension for Multi-node GPU

As shown in the previous section, users can easily write parallel programs using XMP directives. Our goal is to provide a productive and efficient parallel programming model for multi-node GPU clusters. In this section, we will introduce new directives to describe GPGPU within the XcalableMP framework. These directives are used to describe typical processes for GPU computing such as data allocation, data movement between the host memory and the GPU memory and loop parallelization on GPUs. Since XMP directives take on inter-node communication, XMP-ACC can be used not only for a single GPU but also multi-node GPU clusters.

Figure 3 shows some XMP-ACC sample code that calculates the sum of some of an array elements are on GPU. Lines including the keyword **acc** are the new directives/clauses added in XMP-ACC. In XMP-ACC, all actions involving GPUs occur only when the **acc** directives/clauses are used. Typical GPGPU actions including allocation/free data on GPU, data transfer between the host and

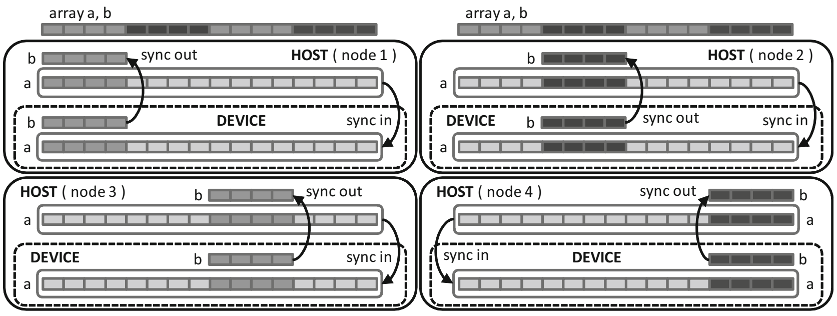


Fig. 4. Memory Image of Data in XcalableMP-ACC

the device and loop work sharing on GPU can be described using the **acc** directives/clauses. Note that there are no directives for inter-node communications (e.g. broadcast, reduction and shadow synchronization) for GPU. In XMP-ACC, the data should be copied to the host memory, then the data should be moved between the hosts using XMP directives (e.g. **bcast**, **reduction** and **reflect**), and then copied from the host to the device.

We assume that each XMP process uses only one GPU, which keeps the language model simple. If there are two or more GPUs in one node, users should assign an XMP process to each GPU (like flat MPI).

3.1 Data Declaration

GPUs have thier own seperate memory, and data should be allocated on the device before being processed. The **acc replicate** directive declares variables to be allocated on the GPU. The following is the syntax of the **acc replicate** directive.

#pragma xmp acc replicate (*list*)
compound-statement

When a variable is declared as **acc replicate**, a copy of the allocated local memory area is also allocated on each node's GPU. Fig. 4 shows the memory image of the arrays a and b declared in Fig. 3 (using 4 nodes). Array b is distributed among the nodes, so the distributed part of the array will be allocated on the GPU. And array a has shadow elements (in this case, full shadow is declared for array a) on both the host and the GPU.

The scope of the **acc replicate** variables is limited to the compound statement following the **acc replicate** directive. The replications will be allocated on the GPU when entering the compound statement and freed at the end of the statement. This helps users to use GPU memory more efficiently. It is also possible to describe the **acc replicate** directive in the global scope (like the **align** directive in Fig.3). Then the replications will be allocated on the GPU when the program starts and freed at the end of the program.

3.2 Data Transfer

Users can describe the data transfer between the host and the GPU using the **acc replicate_sync** directive. The following is the syntax of the **acc replicate_sync** directive and **acc** clause.

```
#pragma xmp acc replicate_sync clause
clause ::= in (list) | out (list)
```

The **acc replicate_sync** directive allows two clauses, **in** and **out**, which indicate the direction of the data transfer. When a process encounters a **acc replicate_sync** directive with the **in** clause, it copies the data from the host to the device. And with the **out** clause, it copies the data from the device to host. Currently, there is no way to indicate the range to be copied, so all of the variable's data is moved between the host and the device. Fig. 4 shows data transfer for arrays *a* and *b*. All the elements of the array *a* including the shadow elements are copied to the device.

3.3 Work Sharing

If every iteration in a loop statement can be processed independently, the loop statement can be parallelized not only among the nodes but also with GPUs. Therefore, the **loop** directive can be exetended to use GPUs. We introduced the **acc** clause for the **loop directive**. The following is the syntax of the **acc** clause for the **loop** directive.

```
#pragma xmp loop [(list)] on on-ref [reduction (op:list)] acc {clause}
loop-statement

clause ::= private (list) | firstprivate (list) | shared (list) |
num_threads (x[, y[, z]])
```

Variables listed in the **private**, **firstprivate** and the **shared** clause are declared as private variables on each thread. And the data of **firstprivate** variables is copied from the host to the device before the loop statement. The **private** and the **firstprivate** clause only allow scalar variables. Array variables should be replicated on the device using the **acc replicate** directive. The **shared** clause lists the variables allocated on the GPU. The data will be allocated and synchronized before loop execution. **acc replicate** variables are declared as **shared** variables by default. The **num_threads** clause is used to determine the thread block size. The default value is (16×16) when the clause is omitted.

4 Implementation of XcalableMP-ACC

Compiler Implementation. Our XMP-ACC compiler is based on the Omni XcalableMP Compiler[4]. Fig. 5 shows the compilation process. *sample.c* is written in the C language and XMP/XMP-ACC directives. The compiler creates two files from the source code. *sample.i* is a intermediate file including the host code (executed by the CPU). *sample.cu* includes the device code which is executed by the GPU. We are using CUDA as the GPGPU backend compiler for XMP-ACC. *sample.cu* is compiled by the CUDA compiler. Finally those object files are linked with the runtime library and produce a parallel program. When multiple XMP-ACC processes are assigned to a physical node, the runtime library will assign each XMP process to a GPU in a circular order.

Code Translation. When *sample.c* in Fig. 3 is compiled, it will produce the parallel codes, *sample.i* and *sample.cu* as shown in Fig. 5. *sample.i* shows the host code of the parallel program. _XMP_gpu_init/finalize_data_ALIGNED() initializes/finalizes the data region and the descriptor on the device. _XMP_gpu_sy nc() transfers the data between the device and the host.

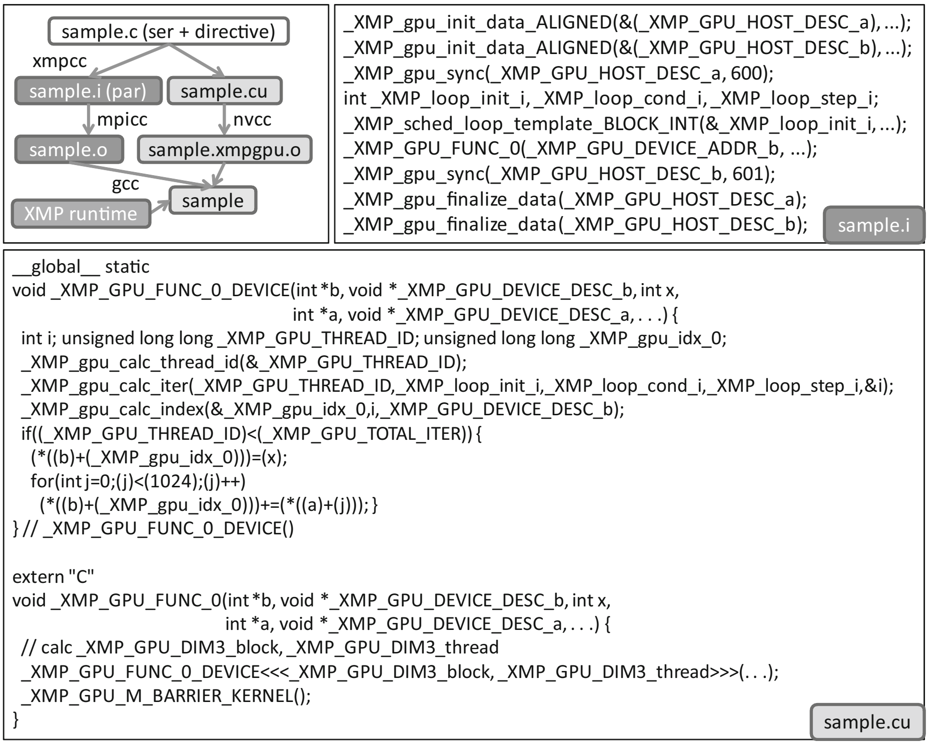
The loop statement parallelized by the **acc** clause is translated into a GPU kernel function in CUDA and added to *sample.cu*. The compiler replaces the loop statement with a GPU function call. The compiler analyzes the loop statement to create the function arguments. If a **acc replicate** array variable appears in the loop statement, the array address on the device will be added to the argument list. And the descriptor address also will be added for calculating parallel parameters. The scalar variables are added to the argument list unless they are described explicitly as thread-private or shared variables.

sample.cu includes the GPU kernel function and its wrapper function which is invoked by the host code. The wrapper function is called from *sample.i*. The thread block size is calculated before invoking the GPU kernel function. The number of threads the compiler creates is equal to the number of iterations allocated to the node. _XMP_GPU_M_BARRIER_KERNEL() waits for the GPU execution to end. Each GPU thread executes the kernel function, and calculates local array indices from its thread ID and the allocated iteration number.

5 Performance Evaluation

We evaluated our compiler using a benchmark that solves the n-body problem, which is often used for evaluating GPGPU performance. Fig. 6 shows our implementation of the XMP-ACC version of the n-body problem. *p_x*, *p_y* and *p_z* contain the x, y, z locations of the particles. The array has shadow elements because every element is needed to calculate the force on a particle. In each time step, the array data is updated. So the shadow elements should be also exchanged each time step. Because those arrays are stored in GPU memory, the data should be exchanged via host memory. The **acc replicate_sync out** directive is used to copy data from the device to the host. Then the **reflect** directive exchanges shadow elements in the host memory. Finally, the **acc replicate_sync in** directive copies the array data from the host to the device.

The force calculation is parallelized on the GPU. The **acc** clause directs the compiler to produce GPU code for the target loop statements. Note that we split the one loop into two seperate loops. This is because the update of *p_x*, *p_y* and



```
_XMP_gpu_init_data_ALIGNED(&(_XMP_GPU_HOST_DESC_a),...);
_XMP_gpu_init_data_ALIGNED(&(_XMP_GPU_HOST_DESC_b),...);
_XMP_gpu_sync(_XMP_GPU_HOST_DESC_a, 600);
int _XMP_loop_init_i, _XMP_loop_cond_i, _XMP_loop_step_i;
_XMP_sched_loop_template_BLOCK_INT(&_XMP_loop_init_i,...);
_XMP_GPU_FUNC_0(_XMP_GPU_DEVICE_ADDR_b, ...);
_XMP_gpu_sync(_XMP_GPU_HOST_DESC_b, 601);
_XMP_gpu_finalize_data(_XMP_GPU_HOST_DESC_a);
_XMP_gpu_finalize_data(_XMP_GPU_HOST_DESC_b);
```
sample.i

```
__global__ static
void _XMP_GPU_FUNC_0_DEVICE(int *b, void *_XMP_GPU_DEVICE_DESC_b, int x,
                            int *a, void *_XMP_GPU_DEVICE_DESC_a, . . .) {
  int i; unsigned long long _XMP_GPU_THREAD_ID; unsigned long long _XMP_gpu_idx_0;
  _XMP_gpu_calc_thread_id(&_XMP_GPU_THREAD_ID);
  _XMP_gpu_calc_iter(_XMP_GPU_THREAD_ID,_XMP_loop_init_i,_XMP_loop_cond_i,_XMP_loop_step_i,&i);
  _XMP_gpu_calc_index(&_XMP_gpu_idx_0,i,_XMP_GPU_DEVICE_DESC_b);
  if((_XMP_GPU_THREAD_ID)<(_XMP_GPU_TOTAL_ITER)) {
    (*((b)+(_XMP_gpu_idx_0)))=(x);
    for(int j=0;(j)<(1024);(j)++)
      (*((b)+(_XMP_gpu_idx_0)))+=(*((a)+(j))); }
} // _XMP_GPU_FUNC_0_DEVICE()

extern "C"
void _XMP_GPU_FUNC_0(int *b, void *_XMP_GPU_DEVICE_DESC_b, int x,
                     int *a, void *_XMP_GPU_DEVICE_DESC_a, . . .) {
  // calc _XMP_GPU_DIM3_block, _XMP_GPU_DIM3_thread
  _XMP_GPU_FUNC_0_DEVICE<<<_XMP_GPU_DIM3_block, _XMP_GPU_DIM3_thread>>>(. . .);
  _XMP_GPU_M_BARRIER_KERNEL();
}
```
sample.cu

Fig. 5. Compilation Process and Translated Codes

p_z should be done after all threads finish calculating the force on its allocated particle. Because each loop is translated into a GPU function, it is guaranteed that barrier synchronization takes place among all threads. In the future version, we need to implement inter-block barrier synchronization so we only have to use one loop.

Table 1 shows the node configuration of the GPU cluster. We evaluated the performance of 1,2 and 4 XMP processes using 4 physical nodes. Fig. 7 shows the performance of n-body. We compared the performance with the serial version of n-body (1 node, 1 thread, using CPU only). Since most of the execution time is spent on the force calculation which is embarrassingly parallel, XMP-ACC shows scalable performance up to 4 XMP processes. This is especially true as the data size increases since GPU calculation time dwarfs the data transfer and shadow reflection time, which leads to better performance when using GPUs. We added only 10 lines to write the XMP version of n-body (the serial version has 105 lines). Furthermore 4 directives and 2 **acc** clauses were added to write the XMP-ACC version. This shows that XMP-ACC provides a scalable and productive programming model for multi-node GPU clusters.

```
#pragma xmp align [i] with t(i) :: p_x, p_y, p_z, m, v_x, v_y, v_z
#pragma xmp shadow [*] :: p_x, p_y, p_z, m
#pragma xmp acc replicate (p_x, p_y, p_z, m, v_x, v_y, v_z)
{
#pragma xmp acc replicate_sync in (m, v_x, v_y, v_z)
  for (t = 0; t < TIME_STEP; t++) {
#pragma xmp reflect p_x, p_y, p_z
#pragma xmp acc replicate_sync in (p_x, p_y, p_z)
#pragma xmp loop on t(i) acc
    for (i = 0; i < N; i++) {
      double x_i, y_i, z_i, x_j, y_j, z_j, dx, dy, dz, r2, r, a;
      double acc_x = 0, acc_y = 0, acc_z = 0;
      x_i = p_x[i];    y_i = p_y[i];    z_i = p_z[i];
      for (int j = 0; j < N; j++)
        if (i != j) {
          x_j = p_x[j];       y_j = p_y[j];       z_j = p_z[j];
          dx = x_j - x_i;     dy = y_j - y_i;     dz = z_j - z_i;
          r2 = (dx * dx) + (dy * dy) + (dz * dz) + EPSILON;
          r = sqrt(r2);       a = G * m[j] / r2;
          acc_x += a*(x_j-x_i)/r;  acc_y += a*(y_j-y_i)/r;  acc_z += a*(z_j-z_i)/r; }
      v_x[i] += acc_x * DT;    v_y[i] += acc_y * DT;    v_z[i] += acc_z * DT; }
#pragma xmp loop on t(i) acc
    for (i = 0; i < N; i++) {
      p_x[i] += v_x[i] * DT;    p_y[i] += v_y[i] * DT;    p_z[i] += v_z[i] * DT; }
#pragma xmp acc replicate_sync out (p_x, p_y, p_z)
  } // for (t = 0; t < TIME_STEP; t++)
} // #pragma xmp acc replicate
```

Fig. 6. n-body Code

6 Related Work

OpenMPC[5] and OMPCUDA[6] are the GPGPU extensions for OpenMP. They produce CUDA code from OpenMP directives with few or no modifications. The OpenMP Architecture Review Board[2] itself is also considering a extension of OpenMP targetting many-core processors including GPUs and Digital Signal Processors. PGI Accelerator Compilers[7] provide original directives for GPU computing. Data allocation and loop parallelization can be described more explicitly than in OpenMPC and OMPCUDA which are based on the OpenMP specification. Those models make it easy to program with GPUs in a single node, but they do not work for GPU clusters or even for multiple GPUs in a single node. HMPP Workbench[8] provides directives to describe data transfer between the GPU and the CPU, launching GPU kernel functions (even asynchronously), etc. Because HMPP uses CUDA and OpenCL as a backend compiler, it works on multi-core CPUs and multiple GPUs in a single node. But HMPP is not considersing GPU clusters now. Yili Zheng et al. are working on a GPGPU extension of Unified Parallel C[9] targeting both single and multiple GPU environments.

Table 1. Node Configuration

CPU	AMD Opteron Processor 6134 × 2 (8 cores × 2 sockets)
Memory	DDR3-1333 2GB × 2 (4GB)
GPU	NVIDIA Tesla C2050 (GDDR5 3GB)
Network	InfiniBand (4X QDR)
OS	Linux kernel 2.6.18 x86_64
MPI	OpenMPI 1.4.2
GPU Backend	NVIDIA CUDA Toolkit v3.2

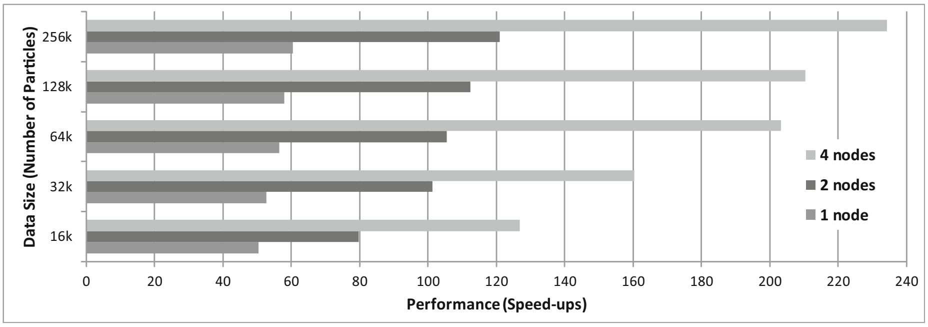


Fig. 7. Performance of n-body

They extended the communication library GASNET to handle one-sided communications for GPUs. It supports unified one-sided communication APIs for GPUs and CPUs. But significant modifications are needed to the serial code in order to parallelize the code.

7 Conclusion

In this paper, we have proposed XcalableMP-ACC, a language extension of XcalableMP for GPU computing. XMP-ACC targets multi-node GPU clusters which has one or more GPUs in each node and provides OpenMP-like directives which allows incremental parallelization from the serial code. XMP-ACC's new directives describe explicit data transfer and loop parallelization for GPU computing, and works naturally with the XMP model used for inter-node communication. Our performance evaluation with n-body problem shows that XMP-ACC achieved scalable performance with few modifications from the serial code.

We are currently improving both the language model and the implementation. We assumed that there is only one GPU per XMP-ACC process to keep the language model simple. However recent platforms feature many more CPU cores than GPUs and thus a variety of execution models exist according to the target applications (e.g. single process using multiple GPUs). So we need to extend the language specification to match various needs. When there are surplus CPU cores, CPU-GPU cooperative computing may be an attractive way to boost the performance. Efficient memory use is one of keys to achieving better performance in GPGPU. Our current implementation only uses the GPU's global memory, so we need to optimize the memory use, for example, using shared memory and coalesced memory access.

Acknowledgment. The specification of XcalableMP is being designed by the XcalableMP Specification Working Group which consists of members from academia, research labs and industries. This research is supported by "Seamless and Highly-productive Parallel Programming Environment for High-performance computing" project funded by Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

1. XcalableMP Official Website, `http://www.xcalablemp.org`
2. OpenMP.org, `http://openmp.org/wp`
3. Rice University. High Performance Fortran Forum, `http://hpff.rice.edu`
4. Lee, J., Sato, M.: Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In: 39th International Conference on Parallel Processing Workshops, pp. 413–420 (2010)
5. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11 (2010)
6. Ohshima, S., Hirasawa, S., Honda, H.: OMPCUDA: OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 161–173. Springer, Heidelberg (2010)
7. PGI Accelerator Compilers, `http://www.pgroup.com/resources/accel.htm`
8. HMPP Workbench, `http://www.caps-entreprise.com/hmpp.html`
9. Hargrove, P.H., Min, S.-J., Zheng, Y., Iancu, C., Yelick, K.: Extending Unified Parallel C for GPU Computing, `http://upc.lbl.gov/publications/UPC_with_GPU-SIAMPP10-Zheng.pdf`