JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 32, 517-539 (2016)

An OpenMP Programming Toolkit for Hybrid CPU/GPU Clusters Based on Software Unified Memory*

HUNG-FU LI, TYNG-YEU LIANG+ AND YU-JIE LIN
Department of Electrical Engineering
National Kaohsiung University of Applied Sciences
Kaohsiung, 807 Taiwan
E-mail: lty@mail.ee.kuas.edu.tw+; {sunneo, jaredlin}@hpds.ee.kuas.edu.tw

Recently, hybrid CPU/GPU cluster has drawn much attention from the researchers of high performance computing because of amazing energy efficiency and adaptable resource exploitation. However, the programming of hybrid CPU/GPU clusters is very complex because it requires users to learn new programming interfaces such as CUDA and OpenCL, and combine them with MPI and OpenMP. To address this problem, we propose a novel OpenMP toolkit called HyCOMP (Hybrid Cluster OpenMP) for hybrid CPU/GPU clusters in this paper. This toolkit is developed based on a novel page-based distributed shared memory system called SUM (software unified memory) which is aimed at emulating a virtual shared memory space over distributed CPUs and GPUs. Compared to traditional page-based DSM systems, SUM can effectively prevent GPUs from performance degradation caused by the latency of handling an enormous number of page faults coming from host-to-device memory copies. Moreover, HyCOMP can automatically achieve load balance of heterogeneous processors. Consequently, HyCOMP dramatically reduces the programming complexity of hybrid CPU/GPU clusters while simultaneously maintains the execution performance of user programs.

Keywords: hybrid CPU/GPU cluster, OpenMP, distributed shared memory, page-based, pre-fetching, load balance

1. INTRODUCTION

Hybrid CPU/GPU cluster computing recently has drawn much attention from the researchers of high performance computing. Since GPU [1] is higher in core density and energy efficiency than CPU, hybrid CPU/GPU clusters can provide higher computational performance with less energy consumption than pure CPU clusters. On the other hand, this computing architecture provides users with a flexible way to exploit the computational power of resources. That is, they can adapt the types of processors used for different problems based on the properties of their programs. Because of these advantages, more and more researchers [2-5] make use of hybrid CPU/GPU clusters instead of pure CPU ones for resolving data-intensive problems such as high energy physics, earthquake and weather forecast, big data mining and so on. Reacting to this new trend, 53 supercomputing sites in Top 500 [6], and the first 15 systems of Green 500 [7] have been built based on hybrid CPU/GPU clusters.

However, the programming of hybrid CPU/GPU cluster computing is very complex.

Received January 29, 2015; revised May 21, 2015; accepted June 24, 2015.
Communicated by Cho-Li Wang.
+ Corresponding author.
* This work is supported by National Science Council of Taiwan under the research project numbered as NSC-99-2221-E-151-055-MY3.

Users not only have to learn GPU programming such as CUDA [8] or OpenCL [9] but also need to manually handle data communication among distributed CPUs and GPUs with MPI and the memory-copy primitives of the GPU programming interfaces. On the other hand, GPU has a deeply hierarchical memory architecture including register, local memory, shared memory, global memory, texture memory and constant memory. These memories are different in access cost, access right and sharing scope. Therefore, users must carefully allocate data on proper memory location according to sharing type and scope to minimize the cost of data accesses. Moreover, they should tune their programs with special programming skills such as coalesced I/O and multiple asynchronous streams to optimize the execution performance of GPU, and manually handle the problem of load balance when the computation power of processors is not identical. Obviously, most of users must spend much effort on overcoming these difficulties, and thereby they hesitate to adopt hybrid CPU/GPU clusters for resolving their problems.

Fortunately, many source-to-source OpenMP [10] compilers such as Cetus [11], OpenMPC [12], OMPi [13], and HMCSOT [14] have been proposed for minimizing the complexity of GPU programming. These compilers can translate OpenMP directives into CUDA or OpenCL codes and optimize the generated source codes based on program analysis or user hints for minimizing the cost of data accesses, the number of branch instructions and synchronization points and so on. However, users have to combine OpenMP with MPI for exploiting multiple distributed GPUs. Although hybrid OpenMP/MPI [15, 16] is easier than hybrid CUDA/MPI [17] or OpenCL/MPI [18], it is not as easy as OpenMP based on distributed shared memory. Nonetheless, it is a pity that all the proposed page-based DSM systems are dependent on page-fault handler. Consequently, user programs cannot effectively exploit the computational power of GPUs because the latency of handling page faults coming from host-to-device memory copies is very high compared to the computational speed of GPU.

As previously discussed, we propose a novel OpenMP toolkit called HyCOMP (Hybrid Cluster OpenMP) based on software unified memory (SUM) in this paper. Using this toolkit, users can develop applications on a hybrid CPU/GPU cluster with only OpenMP while they can select only CPUs/GPUs or both of CPUs and GPUs for executing their programs. HyCOMP can automatically maintain the data consistency of shared memory, and load balance of heterogeneous processors during the execution of user applications. On the other hand, SUM can automatically pre-fetch shared pages necessary for processors before the processors access these pages in parallel regions without page faults by the assistance of OpenMP compiler. As a result, it can effectively maintain the execution performance of processors especially for GPUs because of eliminating the latency of handling a large number of page faults. Moreover, SUM is able to keep track of the addresses of the shared variables read and written by GPUs. Consequently, it can significantly minimize the cost of maintaining data consistency in a hybrid CPU/GPU cluster due to reduction on the number of memory copy from host to device, and the latency of retrieving the data updates from the modified shared memory pages.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 and Section 4 introduce HyCOMP and SUM, respectively. Section 5 is the performance evaluation of HyCOMP. Finally, Section 6 gives conclusions for this study and our future work.

2. RELATED WORK

In recent years, many toolkits have been proposed to provide users with a uniform programming interface for heterogeneous computing. For example, HMPP (Hybrid Multicore Parallel Programming) [19] provides a set of directives for users to exploit computational power of CPUs, GPUs, and Cell processors while they are not necessary to face the programming problem of heterogeneous processors. Basically, the HMPP directives are classified into control and data management. The control directives include codelet and callsite. A codelet is used to define a function executed on one specific target (*e.g.* CPU, GPU or Cell) while a callsite is used to invoke a codelet which is offloaded to a given target for execution. On the other hand, the basic directives of data management of HMPP are *in*, *out* and *inout*. These directives are used to define each argument of a codelet as input, output or both of input and output. The target compiler of HMPP uses these directives to decide when and where it must generate the codes for data transfer between host and device. Although HMPP supports a uniform programming interface for heterogeneous clusters, it requires users to explicitly specify the input and output of the codelet. Consequently, the API of HMPP is not as transparent as OpenMP in data communication. Moreover, HMPP is implemented based on RPC. Consequently, target processors cannot directly communicate with each other. This increases not only programming complexity but also the cost of data communication of user programs.

StarPU [20] is a task programming library for heterogeneous computing architecture. It offers a unified task abstraction called *codelet* which can be executed by x86 CPU, NVIDIA GPU, OpenCL devices, and Cell processors. An application basically is implemented as a set of codelets with high-level description of data dependencies. StarPU can automatically dispatch the codelets of the application onto heterogeneous processors for concurrent execution, and handle task dependency and data transfer/replication between host memory and device memory within the target machine when the application tasks are executed. Consequently, users can put their attention on algorithm development but programming problems of heterogeneous computing with the support of StarPU. On the other hand, StarPU can optimize the task scheduling of heterogeneous architecture and avoid unnecessary data transfer and replication to increase the execution performance of user applications. However, as same as HMPP, the users of StarPU must supply different implementation of the same function for each processor architecture and explicitly declare data dependencies in the definition of codelets. In addition, they have to combine the StarPU interface with MPI for heterogeneous cluster computing. Consequently, StarPU is not as transparent as HMPP for internode data communication.

OpenACC [21] is a new programming standard released by CAPS, PGI and CRAY to simplify the programming of heterogeneous computing. As same as OpenMP, programmers can use OpenACC directives to specify which loops or code regions require parallel computing with multicore CPU or many-core GPU. In addition, it allows users to manage the data communication and work distribution between host and device, and map computation onto devices. Although OpenACC can greatly reduce the programming complexity of hybrid CPU/GPU clusters, it also requires users to explicitly specify the memory copies between the host and the device with the data clauses of copy, *copyin* and *copyout*. Consequently, it is weaker than OpenMP for the transparency of data

communication. In addition, users must combine OpenACC with MPI or distributed shared memory systems such as OpenSHMEM [22] and TreadMarks [23] to exploit the computation power of multiple GPUs distributed over networks. However, OpenSHMEM asks users to explicitly allocate shared memory and use put (write) and get (read) which are similar to MPI_put() and MPI_get() for one-way data communication between one process and another. In other words, users must handle data communication in their programs by themselves. By contrast, TreadMarks allows distributed processes to transparently access shared memory for data communication. If processes modify shared memory, it can automatically maintain the data consistency of updated shared memory. However, this DSM system relies on the mechanism of page fault and a lazy release protocol [24] for maintaining data consistency. Consequently, the execution performance of GPUs is seriously degraded by the latency of handling lager number of page faults, and fetching the updates of shared memory. Different to TreadMarks, ADSM [25] provides a programming model based on an asymmetric shared memory between CPU and GPU. With the support of ADSM, the global memory of GPU is automatically mapped into the memory space of host. Therefore, CPU can transparently read and write the data allocated in device memory. However, it also relies on page-fault mechanism for maintaining data consistency between host and device, and requires users to combine CUDA and MPI for exploiting the computational power of GPU clusters.

In addition to directive-oriented interfaces, some proposed toolkits are aimed at enabling CUDA to be a uniform programming interface of heterogeneous computing. For example, Ocelot [26] supports users to develop applications by CUDA SDK, and execute their CUDA applications by CPU or NVIDIA GPU. When user executes CUDA programs by CPU, Ocelot can dynamically translate the PTX codes of kernel functions into LLVM-IRs and then execute the created LLVM-IRs with CPU through LLVM. However, the Ocelot users have to combine CUDA with MPI and multithreading programming APIs to make use of a cluster of CPUs and GPUs, and handle the problem of load balance by themselves. By contrast, CHC [27] provides a transparent memory space for CPU and GPU within the same node to communicate with each other by means of memory access instead of memory copy. Whenever the execution of a kernel function is finished, it will automatically maintain the consistency of this shared memory space. In addition, it supports automatic load balancing by a workload distribution module and global scheduling queue. Obviously, CHC successfully provides users with a single GPU programming model on hybrid CPU/GPU computing architecture. Unfortunately, it also does not support a cluster of CPUs and GPUs.

Compared to the previous toolkits, the advantage of HyCOMP provides a uniform and familiar programming interface, *i.e.*, OpenMP for users to develop applications in heterogeneous clusters. Consequently, users need not learn any new programming interface and supply different versions of programs for each processor architecture and handle data communication by means of explicit declaration of data dependency or function calls. In other words, HyCOMP is more effective for reducing the programming complexity of heterogeneous clusters. On the other hand, SUM emulates a virtual unified memory over a hybrid CPU/GPU cluster. It effectively resolves the high-latency problem existing in TreadMarks for improving the performance of user programs executed by multiple distributed GPUs. Although CUDA 6.0 [28] also supports unified memory for data communication between host and device through hardware, users must explicitly

call cudaMallocManaged() and cudaDeviceSynchronize() in their programs to make a memory space shared between host and device and keep data consistency, respectively. On the other hand, the hardware of unified memory does not allow CPU and GPU to concurrently access the same shared variable, and supports only a single node but a cluster. Moreover, it is supported only by NVIDIA Kepler and Maxwell architecture, and special OS versions such as 64bit Windows 7+ and Linux Kernel 2.6.18+. In contrast, HyCOMP allows CPU and GPU to concurrently access the same shared variable, and transparently maintains data consistency over a hybrid CPU/GPU cluster without involving users. It does not require any special hardware support and OS version.

3. HYCOMP

HyCOMP is mainly composed of an OpenMP compiler and a runtime system. The OpenMP compiler of HyCOMP is responsible to translate OpenMP directives in users programs into the codes executable on target processors. Generally speaking, an OpenMP program can be divided into sequential regions and parallel regions. The sequential regions are executed only with CPU while the parallel regions can be executed with only CPU, only GPU or both of CPU and GPU. HyCOMP allows users to select the type of target processor such as Intel x86 CPU, NVIDIA and AMD GPU for the execution of each parallel region to chase the best execution performance. However, if the selected processor type is not available in the execution node, HyCOMP will select one from available processor types for executing the parallel region according to the default favor order because of fault tolerance consideration. Consequently, the OpenMP compiler translates parallel regions into three different versions of dynamic linking libraries (DLL), and individually optimizes the DLLs according to the properties of different processor architectures. On the other hand, the runtime system of HyCOMP consists of load balancer and SUM. Load balancer is responsible for dynamically balancing the load of execution processors. SUM aims at emulating a shared memory space for data communication between CPUs and GPUs distributed over networks. From the viewpoint of users, HyCOMP constructs a system image that is a standalone machine with multiple CPUs and GPUs and a unified memory between the host and the devices for executing their OpenMP programs.

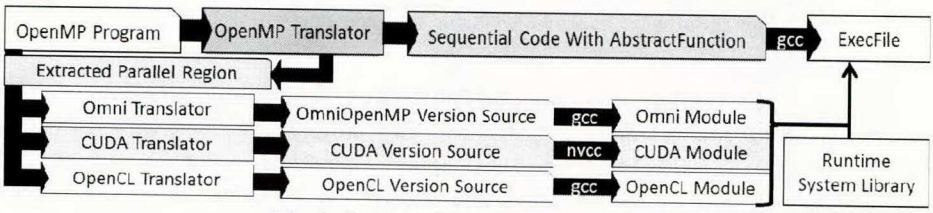


Fig. 1. Process of program compilation.

Basically, the compilation process of an OpenMP program is shown in Fig. 1. At first, the OpenMP compiler extracts the parallel regions that are annotated with parallel-for directives from the program and assigns a function name for each of the extracted parallel regions. Then, the three translators translate the parallel regions into three dif-

ferent versions of source codes for different processor architectures, respectively. Finally, the three different versions of source codes are compiled by gcc or nvcc to become three different dynamic linking libraries (DLLs). On the other hand, the user program is translated into another program called host program which consists of the sequential regions but replaces the parallel regions with abstract functions that play an adapter of the three DLLs. Finally, the host program is compiled by gcc and linked with the runtime libraries of HyCOMP to become an executable file.

After the compilation of a user program is completed, the stack of the executable file of the program is as showed in Fig. 2. Whenever the execution of a program process reaches a parallel region, the abstract function will be responsible to dynamically load one version of DLL based on the architecture of the execution processor selected by device directive, and invoke the function corresponding to the parallel region in the loaded DLL. It is worthy to say that when the selected processor is GPU, HyCOMP will properly determine the dimensions of thread grid and block for launching kernel functions by calling cudaOccupancyMaxPotentialBlockSize(). Because the execution performance of kernel functions is not only dependent on GPU organization but also application behavior, HyCOMP also provides an interface of setting gridDim and blockDim for users to tune the execution performance of kernel functions by themselves. On the other hand, the load balancer will distribute the problem data of the parallel region over different processors for load balance based on dynamically profiling the execution time and workload of the processors in the last parallel region. Moreover, SUM will make shared variables allocated on the virtual shared memory space be visible for all the execution processors, and will maintain the consistency of the shared variables written by CPUs and GPUs in the time interval between any two cumulative synchronization points or the entry point and the departure point of each parallel region.

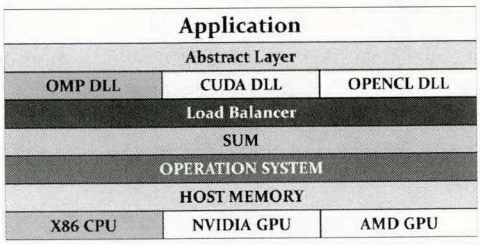


Fig. 2. Software stack of executable files generated by HyCOMP.

Basically, HyCOMP evolves from OMPICUDA [29] which was developed by our previous work to support compound OpenMP/MPI programming on hybrid CPU/GPU clusters. However, OMPICUDA does not support OpenMP-to-OpenCL translation. In addition, it requires users to manually handle internode data communication and load balance by calling the functions of MPI and OMPICUDA runtime libraries. Consequently, we implement an OpenMP-to-OpenCL translator in this paper for HyCOMP to support other processor architectures different from Intel x86 CPU and NVIDIA GPU. On the other hand, we develop a new DSM system, *i.e.*, SUM to emulate shared memory abstract over a hybrid CPU/GPU cluster for HyCOMP. Furthermore, we construct a load balancer for HyCOMP to take over the jobs of data distribution and load balance in order

for reducing the programming effort of users.

Although OpenCL is different from CUDA, the implementation of OpenMP-to-OpenCL translator basically is similar to that of OpenMP-to-CUDA translator. The detail of translating OpenMP directives into CUDA codes has been described in our previous work. Here we only give an example program as shown in Fig. 3 to briefly explain the translation process of OpenMP directives in HyCOMP. This program first allocates a shared array at the virtual unified memory by calling dsmMalloc() and then use a parallel-for directive to concurrently assign a value into the allocated array by multithreading. Finally, it releases the allocated array by calling dsmFree(). After this program is compiled by using the OpenMP compiler of HyCOMP, this program is translated into another program as shown in Fig. 4. The main function is translated into _ompc_main() while the for loop is replaced with an abstraction function. In addition, hycomp_set_global_dim() and hycomp_set_local_dim() are invoked to set the thread configuration for launching kernel function where the parameter, *i.e.*, DEVICE_PREFERENCE, is used for guiding the runtime system to use the default thread configuration. On the other hand, the abstraction function calls the functions of dllLoader() and dlsym() to load one version of DLLs based on the device directive in the user program, and look up the address of puck code that is an adapter responsible to handle the procedures necessary for executing a given parallel-for region on the target processor. Finally, it invokes the puck code in the loaded DLL to copy necessary shared data from the virtual unified memory to host's private memory or device's global memory, and then launches the kernel function of the parallel-for region to multicore CPU or many-core GPU for execution. Basically, the puck code is different from one processor architecture to another. To consider paper length, we describe only the CUDA version of the puck code and kernel function of the example program as follows.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include "DSM.h"
int main(int argc,char** argv){
  int i,*a,size=1000;
  if(argc>1)size=atoi(argv[1]);
  a=(int*)dsmMalloc(sizeof(int)*size);
  #pragma omp parallel for
  for(i=0; i<size; ++i)
        a[ i ] = i;
  dsmFree(a);
  return 0;
}
```

Fig. 3. Program example.

```
extern void abstractFunction(void **_params){
  typedef void (*Stub)(void**);
  extern struct Profile profile[3];
  static Stub stubCache[3];
  int type=hycomp_parse_conf(profile,"example");
  void* dllFile=dllLoader("PuckCode",type);
  if(!stubCache[type])
    stubCache[type]=(Stub)dlsym(dllFile,"PuckCode");
  hycomp_profile_record_start("PuckCode",&profile[type]);
  stubCache[type](_params);
  hycomp_profile_record_end(&profile[type]);
}
int _ompc_main(int argc, char **argv){
  int i,*a,size=1000;
  if(argc>1) size=atoi(argv[1]);
  a=dsmMalloc(sizeof(int)*size);
  { void *_params[2]={ &size,&a};
    hycomp_set_global_dim(DEVICE_PREFERENCE);
    hycomp_set_local_dim(DEVICE_PREFERENCE);
    abstractFunction(_params);
  }
  dsmFree(a);
  return 0;
}
```

Fig. 4. Translation of the example program.

As shown in Fig. 5, the first step of the puck code is to set up the number of threads used for executing the kernel function translated from the parallel-for loop by calling the _cudaSetThreadNumber() function. The second step is to copy the shared array from

```
extern void PuckCode(void **argvs){
  int *_pp_size=(int *)argvs[0],**_pp_a=(int **)argvs[1];
  dim3 blkdim,thdim; size_t *DiffVector,minaddr; Shares *shares=NULL;
  int ilb=0,iub=*_pp_size,numThread=hycomp_get_global_dim();
  int diffsz=sizeof(size_t)*1025;
  __cudaSetThreadNumber(numThread);    //-----------------step1
  hycomp_cuda_exchange_pointer(_pp_a); //-----------------step2
  cudaMalloc(&shares,sizeof(Shares)); //-----------------step3
  cudaMemcpy(&shares->_pp_size,_pp_size,sizeof(int),cudaMemcpyHostToDevice);
  cudaMemcpy(&shares->_pp_a,_pp_a,sizeof(int*,cudaMemcpyHostToDevice);
  allocateCUDAThread(&blkdim,&thdim,hycomp_get_local_dim(),numThread);
  hycomp_loadbalancer("PuckCode",&ilb,&iub,1,1); //------ step4
  cudaMalloc(&DiffVector,diffsz);  //------ step5
  hycomp_cuda_get_minaddr(&minaddr);
  //-----------------------------------------------------step6
  cudaKernel<<<blkdim,thdim,diffsz>>>(shares,ilb,iub,DiffVector,minaddr);
  hycomp_apply_mapping_update(); //-----------------------step7
  hycomp_apply_gpu_dsm_diff(DiffVector,1025);//-----------step8
  cudaFree(DiffVector); cudaFree(shares);
  cudaThreadSynchronize();
  hycomp_clear_segments();
}
```

Fig. 5. The CUDA version of puck code generated for example program.

host memory to device memory, and change the value of the pointer variable from the host memory address of the shared array to the device memory address by calling hycomp_cuda_exchange_pointer(). The third step is to allocate a data structure at device memory for storing the address of the shared array at device memory, and then set up the thread configuration for the kernel function by calling the allocatedCUDAThread() function. The fourth step is to call the hycomp_loadbalancer() function to invoke the load balancer of HyCOMP for loop partition. For achieve load balance, the load balancer first estimates the power factor of each processor with dividing the number of finished iterations by the used execution time in the previous parallel-for loop. Then, it calculates the amount of iterations assigned to each processor by Eq. (1) and sets up the smallest index (S_i) and the biggest index (B_i) of the iterations assigned to the processor based on Eqs. (2) and (3).

$$c_i = LP_i / \sum_{j=1}^{N} P_i, \tag{1}$$

where L is the total number of iterations in the *parallel-for* loop, N is the number of processes, P_i is the power factor of the ith processor and c_i is the amount of iterations assigned to the ith processor.

$$S_i = \sum_{j=1}^{i-1} c_j, \text{where } S_i \text{ is equal to zero} \tag{2}$$

$$B_i = \sum_{j=1}^{i} c_j \tag{3}$$

The load balancer will save these two iteration indexes into the two arguments, *i.e.*, ilb and iub of the hycomp_loadbalancer() function. If the execution processor is a GPU, the two parameters will be copied from host memory to device memory in order to distribute the assigned iterations to GPU threads for parallel execution.

After loop partition, the fifth step of puck code is to allocate a buffer called DiffVector in device memory for tracking the memory position of the data updated by GPU.

The sixth step is to launch the kernel function to GPU for execution and then wait until the kernel function is finished. The seventh step is to copy the shared array and DiffVector from the global memory of device to the private memory of host. The eighth step is to retrieve the updated data by comparing the shared array in the private memory of host and its copy in the shared memory of host, *i.e.*, the cache of the virtual unified memory, according to DiffVector and then flush the updated data to other execution nodes for maintaining the data consistency of virtual unified memory.

On the other hand, each GPU thread executes the same kernel function as shown in Fig. 6. First, it initializes DiffVector and copies it from the global memory of device to the shared memory of device in order to increase the speed of tracking write-access patterns. Second, it calculates the index of the first iteration which is needed to process according to the argument, ilb and its thread identifier number, and then shares the work of the parallel-for loop by an interleaving way with the other threads. Whenever it writes a shared variable, it will actively derive which shared page is updated based on the address of the written variable and then store this memory-access pattern into DiffVector. After the thread finishes the execution of the kernel function, DiffVector will be copied from the device shared memory to the device global memory for maintaining data consistency later.

```
static __global__ void
cudaKernel(Shares* shares,int ilb,int iub,size_t* DiffVector,size_t minaddr){
   size_t* sharedDiffVector=allocateShared<size_t>();
   int i,lb=ilb,ub=iub,step=1;
   /// 1. initialize DiffVector and copy it from global to onchip
   hycomp_cuda_device_write_diff_init(sharedDiffVector,DiffVector);
   sharedDiffVector[0]=minaddr;
   __syncthreads();
   /// 2. calculate idnex of first iteration
   lb += step*__cuda_get_thread_id();
   step*=__cuda_get_thread_size();
   for(i=lb;i<ub;i+=step){
   /// track the address of shared page in the on-chip memory.
      hycomp_cuda_device_write_diff(sharedDiffVector,&shares->_pp_a[i]);
         shares->_pp_a[i]=1;
   }
   /// copy DiffVector from the onchip one to global one.
   hycomp_cuda_device_write_diff_back(sharedDiffVector,DiffVector);
}
```

Fig. 6. The CUDA version of kernel function generated for example program.

4. SUM

SUM is a novel software distributed shared memory system [30-34] dedicated to emulating a shared memory space over distributed CPUs and GPUs as shown in Fig. 7. Basically, the memory space of a host is divided into two parts: private and shared. The shared memory space of a host and the global memory of a device can be regarded as a cache of the virtual unified memory.

For transparency and performance consideration, SUM adopts page as consistency granularity and uses the eager-release update protocol [35] to keep data coherence. For transparency, SUM is devoted to supporting different processes to use the same pointer variable for accessing the same shared variable located in the virtual shared memory

space. However, the memory space of GPU is separated from that of CPU. Consequently, GPU cannot directly use the same pointer variable to access the same shared variable as well as CPU. Moreover, the implementation of traditional page-based DSM systems usually relies on the memory-protection primitive such as mprotect() to adapt the access rights of local pages, and the system call, *i.e.*, signal() to catch page faults, define the fault handler, and keep track of written pages for data consistency maintenance, respectively. However, the GPU driver does not support these primitives or system calls.

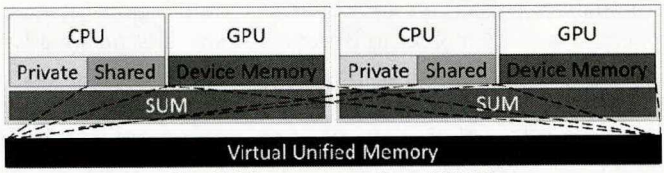


Fig. 7. Memory mapping of SUM.

As previously discussed, the implementation of SUM has two main challenges. One is transparently mapping between device memory and virtual unified memory. Another is efficiently tracking the memory addresses of shared variables written by GPU and making the data updates visible to other CPUs and GPUs. We overcome these problems by integrating the OpenMP compiler of HyCOMP with SUM. The implementation of SUM is described in detail as follows.

4.1 Initialization of SUM

SUM is aimed at constructing a global shared memory space to make the memory addresses of any shared data identical to different hosts. To achieve this goal, the SUM runtime system of each process in the same program negotiates with the others to decide the start address of the virtual unified memory space at first. Then, it allocates a big memory chunk with the same size to be the shared memory space, and creates a page table for managing the allocated shared memory space. Each entry of the page table is a data structure used to store the information of a given page. The page information includes *page owner*, *access right*, *copyset*, and the memory address of *twin copy* and so on. In addition, SUM registers the handlers of read and write faults to the operating systems of computers for catching and handling page faults generated by user programs.

4.2 Memory Allocation and Free

SUM provides two functions, *i.e.*, dsmMalloc(size_*t len*) and dsmFree(void* *ptr*) for allocating and releasing a segment of shared memory. It maintains a pointer variable called *dsmptr* to keep the start memory address for next memory allocation. This pointer variable initially points to the start address synchronized in program initialization. When a user program invokes dsmMalloc(), SUM will return the value of dsmptr to the user program, and will add the value of dsmptr by that of the *len* argument. Conversely, SUM will free the shared memory segment pointed by the *ptr* argument whenever a user program invokes dsmFree(). However, this function is aimed at freeing the physical memory

mapped to the shared memory space released by user programs but reusing the address space of the released shared memory segment in order to simplify memory space management. The reason for this design is that the cost of handling the problem of memory space fragmentation is very high. It is worthy to say that the root node of a computer cluster is regarded as the initial owner of newly allocated memory pages in SUM. Moreover, the access right of the root node to these pages is initially set as READ_ ONLY while the access rights of the other nodes to the pages are initially set as PROT_ NONE. For programming transparency, the OpenMP compiler of HyCOMP can automatically replace malloc() and free() by dsmMalloc() and dsmFree() when users includes the head file, *i.e.*, <hycom.h>.

4.3 Read/Write Fault Handler

An OpenMP program is basically composed of sequential regions and parallel regions. In HyCOMP, the sequential regions are executed only by CPU while parallel regions may be executed by both of CPU and GPU. For performance consideration, the OpenMP compiler analyzes the behavior of data accesses in parallel regions, and then creates codes in the new source programs for trigging SUM to pre-fetch the data pages necessary for the execution of parallel regions into host memory at run time. Consequently, read faults occur only when program processes access absent shared pages in the sequential regions of user programs. By contrast, write faults on shared data pages may happen in the sequential regions or parallel regions of user programs while they are created only by CPU but GPU. The process of handling the page faults of CPU in SUM basically is as same as that in the traditional DSM systems [36-38] with release consistency protocol.

4.4 Mapping Between Shared Memory Space and Device Memory Space

HyCOMP allows GPU to use the same pointer variables to access the shared data dynamically allocated at the shared memory space maintained by SUM. To achieve this goal, the runtime system of HyCOMP creates a mapping table between shared memory space and device memory space, as shown in Fig. 8. Each entry of the mapping table includes the start address (denoted as $\boldsymbol{a_i}$) and length (denoted as $\boldsymbol{L_i}$) of a shared memory segment and the start address (denoted as $\boldsymbol{cudaPtr_i}(\boldsymbol{CLObj_i})$ of the device memory segment mapped to the shared memory segment.

On the other hand, the OpenMP compiler scans out all the pointer variables used in each parallel region, and adds a block of codes into the puckcode function, which registers the shared memory segments of storing the shared data pointed by the pointer variables in the mapping table. Moreover, it adds a function, *i.e.*, hycomp_cudaexchange_ pointer() as shown in Fig. 5 for each of the pointer variables to translate the shared memory address pointed by the pointer variable into a corresponding device memory address before launching the kernel function for the parallel region. As shown in Fig.8, the first step is to look up the mapping table to find out which shared memory segment includes the memory address (denoted as **addr**) pointed by the pointer variable (denoted as **P)**. Assume the start address of the shared memory segment is **a3**. The second step is to check if the shared memory segment has been mapped to device memory. If it is true,

the process directly goes to the next step; otherwise, it goes to allocate a device memory segment of **L3** bytes and store the start address (denoted as **cu_addr3**) of allocated device memory into the mapping table first. The third step is to perform the memory copy from the shared memory segment to the allocated device memory segment. The fourth step is to modify the content of the pointer variable as the value of **cu_addr3+(addr-a3)**.

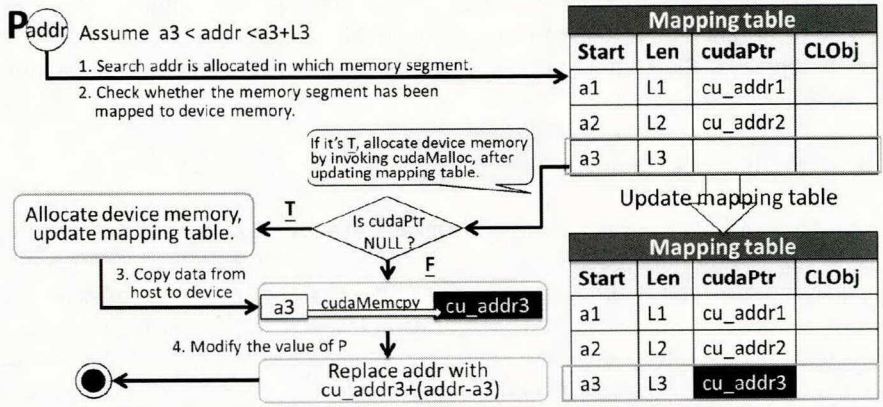


Fig. 8. Mapping table between shared memory space and device memory space.

4.5 Data-Consistency Maintenance

SUM adapts the eager-release update consistency protocol for maintaining data consistency. As a result, the modification of a process to shared memory is visible for other processes only when it arrives at a synchronization point such as barrier arrival, lock and unlock. For the programming model of OpenMP, data synchronization is performed at the points of thread forking and joining. Accordingly, the OpenMP compiler of HyCOMP generates a barrier-arrival function call in in front of the entrance and departure points of each parallel region in user programs for maintaining data consistency. Basically, SUM has two different mechanisms of data-consistency. One is for CPU and another is for GPU. The mechanism for CPU is as same as that used in Teamster [39]. Here we only describe the consistency mechanism for GPU as follows.

Basically, the OpenMP compiler uses hycomp_cuda_device_write_diff_init() as shown in Fig. 5 to allocate a DiffVector in the shared memory of GPU for each thread block and adds hycomp_cuda_device_write_diff() in the kernel function for keeping track of the write access pattern of the thread block in the DiffVector. Basically, each bit in a DiffVector represents if a shared page have been modified during the execution of the kernel function. If the bit value is 1, it means the page has been written. The function, *i.e.*, hycomp_cuda_devie_write_diff(), drives the identifiers of the updated pages from the memory addresses of the written variable in the shared array, and set the corresponding bit value in DiffVector to 1. Before leaving from the kernel function, the DiffVectors of the thread blocks are merged into a global DiffVector allocated at global memory by calling hycomp_cuda_device_write_diff_back(). After the execution kernel function is finished, a function named as hycomp_apply_gpu_dsm_diff() is invoked for the propagation of shared data modification made by GPU. Basically, this function copies the

modified pages from the device global memory to the host private memory according to the DiffVector, and makes a diff (*i.e.*, data updates) for each modified page by comparing its two copies at the host private memory and the host shared memory, respectively. Finally, it propagates the diffs of the modified pages to all the processes to update their shared memory space for data consistency. For the partial sharing applications, this data-consistency mechanism is not efficient enough because each process receives all the data updates from the others no matter if the local processor requires the data updates later no not. To attack this problem, SUM also supports tracking the read access patterns of GPUs in addition to write access patterns. Basically, the implementation of tracking read access patterns is as same as that of write access patterns. By contrast, SUM keep track of read faults to record the read access pattern of CPU. With read and write access patterns, SUM can derive the data sharing among different processes, and then decide if it is necessary to send the local data update on a given shared page to a given remote process. Furthermore, it can copy only the shared data pages which will be read or written by GPU into device memory when the data behavior of user programs is regular. However, tracking read access patterns of GPU will increase the execution time of kernel functions while it is necessary only for the applications of partial read-write sharing. Accordingly, HyCOMP allows users to decide if it is necessary to track the read access patterns of GPUs by using a compilation option in order to save unnecessary runtime cost.

It is worthy to say that the space of device shared memory usually is too small to store the DiffVector of the whole device global memory if the page size is set as 4K. To resolve this problem, HyCOMP increases the size of page granularity in GPU to 128KB in order to minimize the demand of device shared memory. As a result, each bit in a DiffVector is expanded as 32 bits when the DiffVector is copied from device memory to host memory because the page granularity of a page in CPU is 4K. This will result in that the overhead of unnecessary page comparison is increased. Fortunately, the benefit from using device shared memory for tracking the write access patterns of GPU usually is larger than the loss caused by unnecessary page comparison.

4.6 Prefetching Pages for Host-to-Device Memory Copy

As previously discussed, it is necessary to copy the data pages necessary for a kernel functions from host to device before launching the kernel function because the host memory space is different from the device memory space. However, the data pages have not always been cached in the local host memory before memory copy between host and device. If some of the data pages are absent in local host memory, each of the absent pages will cause the operating system to issue a page fault. The traditional page-based DSM system will handle the page faults to fetch the copies of the absent pages from other processes one by one. As a result, the execution performance of GPU will be seriously degraded due to the latency of handling the page faults caused by memory copies from host to device because the bandwidth of computer network is too small compared to the computation power of GPU. To overcome this problem, SUM supports a function called dsmPrefetch() to pre-fetch the absent data pages of a given shared memory segment before copying the shared memory segment from host memory to device memory. The execution flow of this function is simply described as follows. SUM divides the absent data

pages into a number of page groups according to their owners at first, and then acquires the absent data pages of the same page group from the same owner by a request message to minimize the number of requesting messages. After receiving the copies of all the absent pages, SUM sets the access rights of all the data pages in the shared memory segment as PROT_READ. After the pre-fetching process is finished, no read page fault will occur when memory copies from host to device happen.

5. PERFORMANCE EVALUATION

We have done five experiments for evaluating the performance of HyCOMP in this paper. Our experimental environment was built on a cluster of four computer servers. Each server has two Intel Xeon E5645 (12 cores) CPUs, 24GB RAM, and one NVIDIA GeForce 550Ti GPU (192 cores) with 2 GB VRAM. The network connecting these servers is 10Gbps SPF+ Ethernet. The operating system running on these servers is Ubuntu 12.04 64-bit edition with Linux 3.5.0-23-generic kernel. The applications used for this performance evaluation are matrix multiplication (MM), Successive of Relaxation (SOR) and N-body. The MM and Nbody applications have massive data computation while the memory demand of MM is much more than Nbody. In contrast, the SOR application is a data-intensive but computation-intensive problem. Our experimental results are discussed as follows.

5.1 Overhead of Tracking Data-Update Patterns in GPU

This experiment aims at estimating the costs of tracking data-update patterns in GPUs. To achieve this goal, we ran each of the applications with two different kernels at the same computer server. One of the kernels is applied with write-access tracking while another is not applied. We estimated the overhead of tracking data-update patterns by subtracting the execution time of the rear kernel from that of the front one. The estimation result is as shown in Fig. 9.

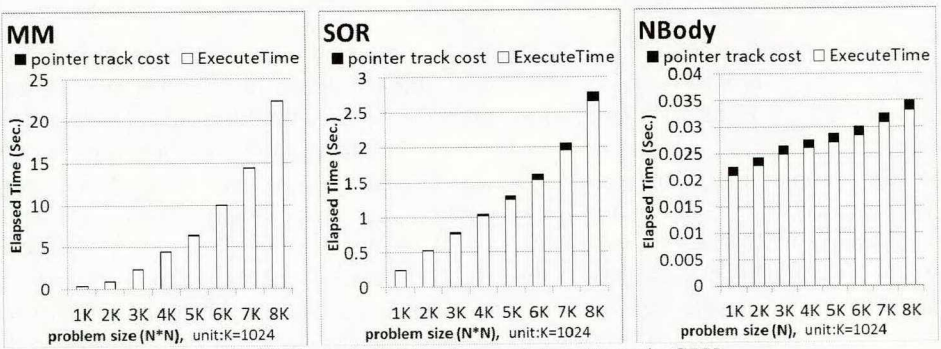


Fig. 9. Cost of tracking data-update patterns in GPU.

Obviously, the cost of tracking data-update patterns in GPU is negligible for the MM and NBody applications. However, it is significant for the SOR applications be-

cause the computation demand of the SOR kernel is relatively much less than that of the others. Fortunately, the impact of this overhead on the performance of the SOR application is reduced when the problem size is increased.

5.2 Effectiveness of the Optimization Mechanisms of SUM

We proposed three optimization mechanisms for increasing the efficiency of SUM as described in Section 4. The first is using DiffVector to avoid scanning the whole shared memory space for retrieving the data updates of GPUs. The second is pre-fetching data pages necessary for the execution of kernel functions without page fault. The third is to track read-access patterns for reducing the cost of host-to-device memory copy and data-consistency maintenance. We ran the test applications by using four GPUs for four different cases including no_opt, opt_1, opt_2 and opt_all to evaluate the effectiveness of the optimization mechanisms, respectively. The no_opt case means that SUM is not optimized by any of the first two mechanisms. The opt_1 or opt_2 case denotes that SUM is applied only with the first optimization mechanism or the second one, respectively. Finally, the opt_all case implies that SUM is applied with the first two mechanisms. The experimental result is depicted in Fig. 10.

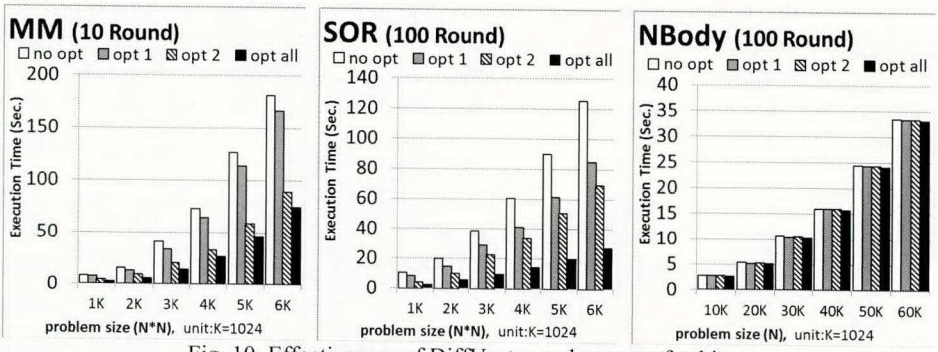


Fig. 10. Effectiveness of DiffVector and page prefetching.

We can find that scanning modified pages based on DiffVector is useful for the execution performance of the MM and SOR applications by comparing the opt_1 case with the no_opt. That is because it can save the time of scanning a large number of unwritten shared memory pages. However, it does not contribute a significant performance improvement for the NBody application because the size of matrixes used in Nbody is much smaller than that used in the previous two applications.

The same situation also happens between the performance comparison between the no_opt case and the opt_2 case. The second optimization mechanism contributes an amazing performance improvement for the MM and SOR applications. Because the data pages necessary for kernel functions has been pre-fetched into the local host memory, no page fault happens when they copy the pages from host to device for the execution of their kernel functions. Consequently, the two applications can save lots of time spent on host-to-device memory copies. However, page prefetching is not obviously useful for the performance of the NBody application because the number of data pages necessary for

the kernel function of NBody is much smaller than that for MM or SOR.

On the other hand, we evaluated the impact of applying read-access pattern tracking on the cost of host-to-device memory copy and data-consistency maintenance. The execution time of the test applications is divided into: Transfer (data-communication), KernelExec, MemcopyHtoD, and MemcopyDtoH. The experimental result is shown in Fig. 11. In the SOR application, data sharing only occurs in the boundaries of data partitions assigned to different processes. HyCOMP can copy only the shared data pages which will be accessed in the kernel function from host to device, and can maintain data consistency only for the shared data pages located at the partition boundary according to the read and write access patterns of GPUs. Consequently, it greatly reduces the cost of host-to-device memory copy and the communication cost of maintaining data consistency, and then significantly improves the performance of the SOR application.

Conversely, the performance of the MM application is not improved but degraded by tracking the read access patterns of GPUs. That is because the MM application reads shared data very frequently. Consequently, HyCOMP cannot compensate the cost of tracking read access patterns by the benefit from saving the cost of memory-copy and data-consistency communication. As to NBody, it is a full sharing application. All of the shared pages must be copies from host to device for the execution of the kernel function, and all the data updates made by any process must be flushed to all the others for data consistency. Tracking read access patterns only increases the execution time of the kernel function and contributes nothing to the cost reduction of memory copy and data-consistency communication. From the above discussion, it can be concluded that it is necessary only for the partial sharing applications such as SOR to track the read-access patterns of GPUs.

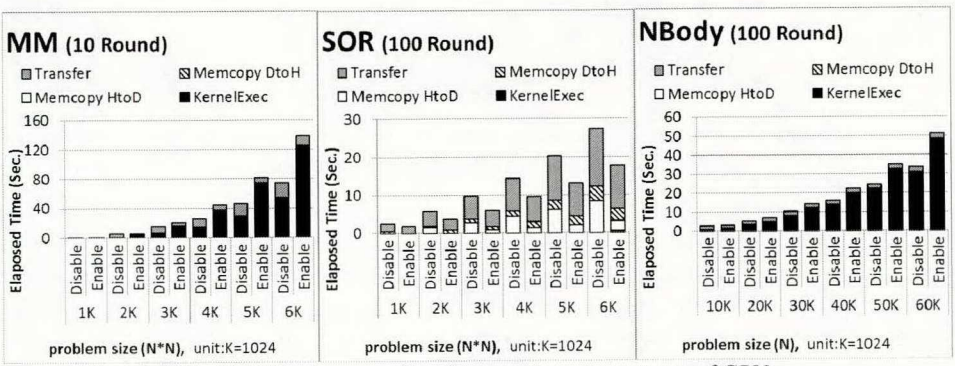


Fig. 11. Effectiveness of tracking read-access patterns of GPUs.

5.3 Performance Comparison Between HyCOMP and Hybrid MPI/CUDA

This experiment is dedicated to comparing the performance of HyCOMP with that of hybrid MPI/CUDA programming. To achieve this goal, we rewrote the test applications by hybrid MPI/CUDA programming. We used MPI_broadcast(), MPI_scatter() and MPI_gather() instead of MPI_send and MPI_receive for internode data communication for simplifying programming complexity. Consequently, the cost of data communication

in the MPI-version of the test applications is not always the minimal. In addition, we applied read-access pattern tracking only for the SOR application in this experiment according to the previous experimental result. Fig. 12 is the breakdown of the execution time of the test applications.

For the MM application, hybrid MPI/CUDA is better than HyCOMP because HyCOMP spends more communication cost on maintaining data consistency. Conversely, HyCOMP is more efficient than hybrid MPI/CUDA for the performance of the SOR and NBody application since HyCOMP spends less communication cost on data consistency. The main reason is that HyCOMP propagates data updates for data consistency only when the values of written data change while the MPI-version applications always gather the written data at the current iteration from all the program processes to the root one, and scatter the gathered data from the root process to the others for the next iteration no matter if the values of written data change or not. From the above discussion, HyCOMP is comparable to hybrid MPI/CUDA for the execution performance of the test applications while HyCOMP is much easier in programming than hybrid MPI/CUDA.

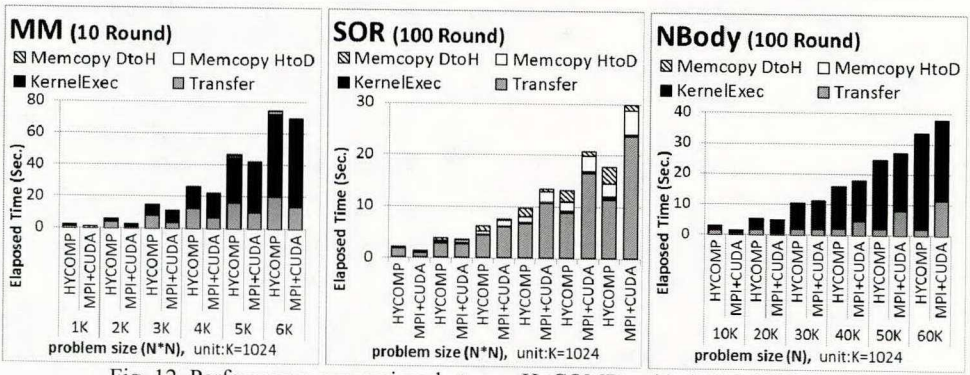


Fig. 12. Performance comparison between HyCOMP and hybrid MPI/CUDA.

5.4 Effectiveness of Adaptive Resource Allocation and Load Balancing

In this experiment, we created a program with two parallel-for regions. One parallel for region is to resolve the NBody problem while another is to resolve the SOR problem. We built a new test environment as shown in Table 1 for executing this complex application. The core number of Intel E5645 is 12. NVIDIA GTX550Ti and GTX750 have 192 and 640 CUDA cores while AMD R7-260 has 768 stream processing units. We ran this test program with four different resource configurations in order to show the impact of adaptive resource allocation. The first and the second are using only CPUs or GPUs of four servers for both of the two parallel-for regions, respectively. They are denoted as Nbody(cpu):SOR(cpu) and Nbody(gpu):SOR(gpu), respectively. The third denoted as Nbody(gpu):SOR(cpu) is to use only GPUs for the parallel-for region of NBody but only CPUs for that of SOR. Conversely, the fourth presented as Nbody(cpu):SOR(gpu) is to use only CPUs for the parallel-for region of NBody but only GPUs for that of SOR. Although the test application was executed by four different resource configurations, HyCOMP always executed this application with all the available core of used processors

and performed the mechanism of load balancing no matter which one of the four resource configurations was applied. Basically, GPU is much better than CPU for NBody while CPU is more suitable for SOR than GPU according to the previous experience. On the other hand, we ran the same program by using only GPUs within four servers without load balancing in order to evaluate the effectiveness of the load balancing mechanism of HyCOMP. The experimental result is shown in Fig. 13.

Table 1. Environment of the fifth experiment.

	Server 1	Server 2	Server 3	Server 4
CPU	E5645x2	E5645x2	E5645x2	E5640
GPU	NV GT 440	NV GTX550Ti	NV GTX750	AMD R7-260
RAM	24GB	24GB	24GB	4GB
Network	10G Fast Ethernet			

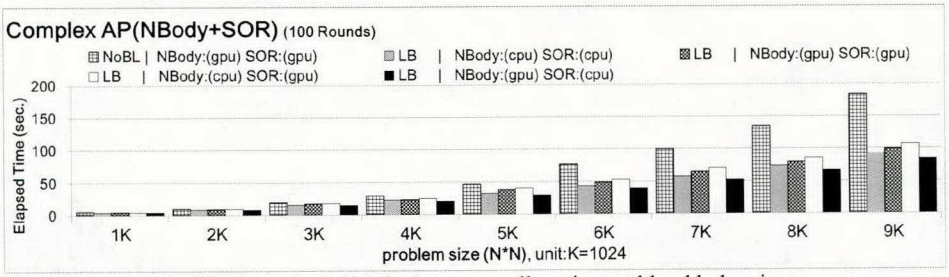


Fig. 13. Impact of adaptive resource allocation and load balancing.

At first, we can find that the best case for the test program is using only GPUs for NBody and only CPUs for SOR no matter what problem size is. The worst case is using only CPUs for NBody and only GPUs for SOR. When both of NBody and SOR use only GPUs, the computation time of the SOR kernel can reach to the minimal while the saved computation time cannot compensate the cost of memory copies between host and device. Therefore, the performance of this case is worse than that of the best case. By contrast, the computation time of the test program is the maximal when both of NBody and SOR use only CPUs although there is no memory copy between host and device. That is because the computational power of CPU is much smaller than that of GPU. However, it is better than the cases of using GPUs for the execution of SOR because the performance loss caused by memory copy between host and device is too high to be compensated by the saved computation time. As previously discussed, carefully adapting the types of processors used for executing different parallel regions in the same program indeed is necessary for the performance of user programs. On the other hand, the performance of executing this application by using only GPUs with load balancing is much better than that without load balancing. This result shows that HyCOMP really is effective for addressing the problem of load balance.

5.5 Effectiveness of Hybrid Processor Configuration

The goal of this experiment is to evaluate the effectiveness of simultaneously using

the CPUs and GPUs of execution nodes for executing the test applications. Since the previous experience had implied that the hybrid processor configuration is not efficient for the SOR application, we replaced it by a clustering application called K-means [40] in this experiment. In addition to hybrid processor configuration, we ran the test applications by using another two processor configurations, *i.e.*, only the GPUs of execution nodes, and only one CPU. Finally, we estimated the speedups of the hybrid and GPU-only configurations by comparing with the execution performance of using only one CPU. Our experimental result is as shown in Fig. 14. In this figure, N is the number of execution nodes. The symbols of gpu and hybrid denote using only GPUs, and both of CPUs and GPUs in execution nodes, respectively. On the other hand, we individually ran each of the test programs by one CPU or GPU, and estimated the execution speed ratio between CPU (*i.e.*, Intel E5645) and GPU (NVIDIA 550Ti) for each application. The estimated result is shown in Table 2, and it will be used to explain the experimental result of Fig. 14 later.

Table 2. Average ratio of execution speed between CPU and GPU.

	MM	NBody	K-Means
CPU:GPU	1 : 24.45	1 : 9.42	1 : 3.01

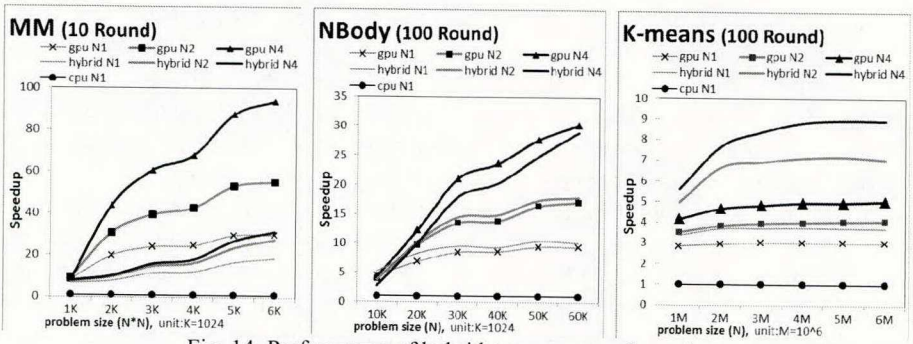


Fig. 14. Performance of hybrid processor configuration.

At first, the hybrid processor configuration is not better than the GPU-only one for the execution performance of the MM application no matter what problem size is and how many node is used. There are two reasons for this result. One is that GPU is much faster in computation speed than CPU for this application. Another is that the cost of data communication is too high to be compensated by the benefit from distributing data onto CPUs and GPUs for parallel computation. On the contrary, the hybrid processor configuration is greatly better than the GPU-only one for the execution performance of the K-means application because CPU is comparable with GPU in terms of execution speed for this application, and the cost of data communication in this application is less than that in the MM application. In the Nbody application, the number of shared data pages and the cost of data communication are much less than those in the MM applications. Consequently, simultaneously using CPUs and GPUs for parallel computation is useful for improving the execution performance of the Nbody application when the number of execution is 1 and 2. However, it becomes not better than using only GPUs for the performance of the Nbody application when the node number reaches to 4 because the in-

crease of data communication cost is larger than the reduction of computation cost. Nonetheless, it seems to be better than using only GPUs for the Nbody application again when the problem size increases.

As previously discussed, the hybrid processor configuration is useful for improving the execution performance of user applications when CPU is comparable with GPU in execution speed or the computation-to-communication ratio of the applications is large enough.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have successfully developed a novel OpenMP programming toolkit called HyCOMP for hybrid CPU/GPU cluster computing. Using this toolkit, programmers can use a uniform API, *i.e.*, OpenMP to develop their applications while they can exploit CPUs and GPUs distributed over networks for resolving the same problem. In addition, they can select different processor configurations such as CPU-only, GPU-only and hybrid CPU/GPU by using a simple device directive for executing different parallel regions in the same program according to the behavior of each parallel region. Consequently, HyCOMP successfully reduces the programming complexity of hybrid CPU/ GPU clusters further compared to the other toolkits proposed by related work, and provides a flexible way of resource exploitation for users to obtain the best program performance. On the other hand, this paper proposed a novel software distributed memory system called SUM for HyCOMP to emulate a virtual unified memory over a hybrid CPU/GPU cluster. With the support of SUM, all the CPUs and GPUs used for executing the same HyCOMP program can communicate with each other by implicit memory accesses instead of explicit message passing and memory copy. Our experiment results has shown that SUM indeed is more effective than traditional page-based DSM systems for the execution performance of GPUs because it can effectively resolve the problem of high latency caused by memory copies from host to device. Moreover, our performance evaluation shows that the adaptive resource allocation and load balance mechanisms in HyCOMP is effective for the performance of user applications.

Although HyCOMP has successfully reduced the programming complexity of hybrid CPU/GPU clusters and maintains an acceptable execution performance for user applications, it has some important issues necessary to be addressed. For example, our experimental results shows that the cost of maintaining data consistency still is a problem for the execution performance of user applications especially when the amount of shared data pages is large. It is necessary to further improve the protocol and mechanism of maintaining data consistency in HyCOMP. In addition, most of users have not enough knowledge to select the best processor configuration for their applications. Therefore, an effective method of automatic resource selection is essential to reduce user's programming burden and optimize the execution performance of user applications. We will address these issues for HyCOMP in the future.

REFERENCES

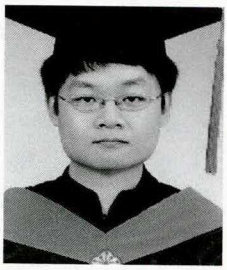
1. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, and A. E. Lefohn,

"A survey of general purpose computation on graphics hardware," *Computer Graph Forum*, Vol. 26, 2007 pp. 80-113.

2. S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, Vol. 36, 2010, pp. 232-240.
3. E. Niewiadomska-Szynkiewicza, M. Marksa, J. Janturab, and M. Podbielskib, "A hybrid CPU/GPU cluster for encryption and decryption of large amounts of data," *Journal of Telecommunications and Information Technology*, Vol. 3, 2012, pp. 32-39.
4. M. Wen, H. Su, W. Wei, N. Wu, X. Cai, and C. Zhang, "High efficient sedimentary basin simulations on hybrid CPU-GPU clusters," *Journal Cluster Computing*, Vol. 17, 2014, pp. 359-369.
5. S. Philip, B. Summa, V. Pascucci, and P.-T. Bremer, "Hybrid CPU-GPU solver for gradient domain processing of massive images," in *Proceedings of the 17th International Conference on Parallel and Distributed Systems*, 2011, pp. 244-251.
6. Top 500 list, http://www.top500.org/lists/2014/11/, 2015.
7. The Green500 list, http://www.green500.org 2015.
8. NVIDIA CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide, 2015.
9. J. E. Stone, D. Gohara, and G. Shi, "OpenCL, a parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, Vol. 12, 2010, pp. 66-73.
10. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka, "Design of OpenMP compiler for an SMP cluster," in *Proceedings of the 1st European Workshop on OpenMP*, 1999, pp. 32-39.
11. H. Bae, D. Mustafa, J.-W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. P. Midkiff, "The Cetus source-to-source compiler infrastructure: Overview and evaluation," *International Journal of Parallel Programming*, Vol. 41, 2013, pp. 753-767.
12. S. Lee and R. Eigenmann, "OpenMPC: extended OpenMP for efficient programming and tuning on GPUs," *International Journal of Computational Science and Engineering*, Vol. 8, 2013, pp. 4-20.
13. G. Noaje, C. Jaillet, and M. Krajecki, "Source-to-source code translator: OpenMP C to CUDA," in *Proceedings of IEEE 13th International Conference on High Performance Computing and Communications*, 2011, pp. 512-519.
14. HMCSOT: OpenMP to OpenCL source-to-source translator, http://www.openfoundry.org/of/projects/1117, 2015.
15. L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *Scientific Programming*, Vol. 9, 2011, pp. 83-98.
16. R. Rabenseifner, G. Hager, and G. Jost,"Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 427-436.
17. S. J. Pennycooka, S. D. Hammondb, S. A. Jarvisa, and G. R. Mudalige, "Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark," in *Proceedings of ACM SIGMETRICS Performance Evaluation Review – Special Issue on the 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, Vol. 38, 2011, pp. 23-29.
18. S. J. Pennycooka, S. D. Hammondb, S. A. Wrighta, J. A. Herdmanc, I. Millerc, and

S. A. Jarvisa, "An investigation of the performance portability of OpenCL," *Journal of Parallel and Distributed Computing*, Vol. 73, 2013, pp. 1439-1450.
19. F. Bodin and S. Bihan, "Heterogeneous multicore parallel programming for graphics processing units," *Scientific Programming*, Vol. 17, 2009, pp. 325-336.
20. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, Vol. 23, 2011, pp. 187-198.
21. OpenACC Home, http://www.openacc-standard.org/2015.
22. S. Pophale, T. Curtis, and B. Chapman, "Improving performance of OpenSHMEM reference library by portable PE mapping technique," in *Proceedings of the 27th International ACM Conference on Supercomputing*, 2013, pp. 485-486.
23. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared memory computing on networks of workstations," *Journal of Computer*, Vol. 29, 1999, pp. 18-28.
24. P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 13-21.
25. I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-M. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 347-358.
26. G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 353-364.
27. C. Lee, W. W. Ro, and J.-L. Gaudiot, "Cooperative heterogeneous computing for parallel processing on CPU/GPU hybrids," *Interaction between Compilers and Computer Architectures*, 2012, pp. 33-40.
28. CUDA6.0 Toolkit-NVIDIA Developer, https://developer.nvidia.com/cuda-toolkit-60, 2015.
29. H.-F. Li, T.-Y. Liang, and J.-Y. Chiu, "A compound OpenMP/MPI program development toolkit for hybrid CPU/GPU clusters," *The Journal of Supercomputing*, Vol. 66, 2013, pp. 381-405.
30. T.-Y. Liang, C.-Y. Wu, C.-K. Shieh, and J.-B. Chang, "A grid-enabled software distributed shared memory system on wide area network," *Future Generation Computer Systems*, Vol. 23. 2007, pp. 547-557.
31. K. Li, "IVY: A shared virtual memory system for parallel computing," in *Proceedings of International Conference on Parallel Processing*, 1988, pp. 94-101.
32. B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The midway distributed shared memory system," *Compcon Spring*, Digest of Papers, 1993, pp. 528-537.
33. R. Friedman, M. Goldin, A. Itzkovitz, A. Schuster, and Millipede, "Easy parallel programming in available distributed environments," *Software – Practice and Experience*, Vol. 27, 1997, pp. 929-965.
34. E. Speight and J. K. Bennett, "Brazos: A third generation DSM system," in *Proceedings of the USENIX Windows/NT Workshop*, 1997, pp. 95-106.
35. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy,

"Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 15-26.
36. M. Stumm and S. Zhou, "Algorithms implementing distributed shared memory," *IEEE Computer*, Vol. 23, 1989, pp. 54-64.
37. W. Hu, W. Shi, and Z. Tang, "JIAJIA: An SVM system based on a new cache coherence protocol," in *Proceedings of the High Performance Computing and Networking*, 1999, pp. 463-472.
38. J. B. Carter, "Design of the munin distributed shared memory system," *Journal of Parallel and Distributed Computing*, Vol. 29, 1995, pp. 219-227.
39. J.-B. Chang, C.-K. Shieh, and T.-Y. Liang, "A transparent distributed shared memory for clustered symmetric multiprocessors," *The Journal of Supercomputing*, Vol. 37, 2006, pp. 145-160.
40. T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient *k*-means clustering algorithm: Analysis and implementation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 24, 2002, pp. 881-892.



Hung-Fu Li (李洪福) currently is a Ph.D. student at the Department of Electrical Engineering in National Kaohsiung University of Applied Sciences (KUAS). He received his BS degree in Computer Science and Information Engineering from National Taitung University in 2009 and received his MS degree in Computer Science and Information Engineering from KUAS in 2011. His research interests are compiler, GPGPU computing and virtualisation.

Tyng-Yeu Liang (梁廷宇) currently is an Associate Professor at the Department of Electrical Engineering in National Kaohsiung University of Applied Sciences. He received his BS, MS and Ph.D. degrees in Electrical Engineering from National Cheng Kung University in 1992, 1994 and 2000, respectively. His research interests include distributed and parallel systems, and cluster/cloud/grid/mobile computing.

Yu-Jie Lin (林于傑) is currently a master student at the Department of Electrical Engineering in National Kaohsiung University of Applied Sciences (KUAS). He received his B.S. Degrees from KUAS in 2013. His research interest is in BigData, embedded system, parallel computing, general purpose GPU computing, and virtualization.