Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 621730, 15 pages
http://dx.doi.org/10.1155/2015/621730

Research Article

Multi-GPU Support on Single Node Using Directive-Based Programming Model

Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran, and Barbara Chapman

Department of Computer Science, University of Houston, Houston, TX 77004, USA

Correspondence should be addressed to Rengan Xu; rxu6@uh.edu

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Xinmin Tian

Existing studies show that using single GPU can lead to obtaining significant performance gains. We should be able to achieve further performance speedup if we use more than one GPU. Heterogeneous processors consisting of multiple CPUs and GPUs offer immense potential and are often considered as a leading candidate for porting complex scientific applications. Unfortunately programming heterogeneous systems requires more effort than what is required for traditional multicore systems. Directive-based programming approaches are being widely adopted since they make it easy to use/port/maintain application code. OpenMP and OpenACC are two popular models used to port applications to accelerators. However, neither of the models provides support for multiple GPUs. A plausible solution is to use combination of OpenMP and OpenACC that forms a hybrid model; however, building this model has its own limitations due to lack of necessary compilers' support. Moreover, the model also lacks support for direct device-to-device communication. To overcome these limitations, an alternate strategy is to extend OpenACC by proposing and developing extensions that follow a task-based implementation for supporting multiple GPUs. We critically analyze the applicability of the hybrid model approach and evaluate the proposed strategy using several case studies and demonstrate their effectiveness.

1. Introduction

Heterogeneous architecture has gained great popularity over the past several years. These heterogeneous architectures are usually comprised of accelerators that are attached to the host CPUs, and such accelerators could include GPUs, DSPs, and FPGA. Although heterogeneous architectures help in increasing the computational power significantly, they also pose potential challenges to programmers before the capabilities of these new architectures could be well exploited. CUDA [1] and OpenCL [2] offer two different interfaces to program GPUs. But in order to perform effective programming using these interfaces, the programmers need to thoroughly understand the underlying architecture and the language/model. This affects productivity. To overcome these difficulties, a number of high-level directive-based programming models have been proposed that include HMPP [3], PGI [4], and OpenACC [5]. These models simply allow the programmers to insert directives and runtime calls into an application code, making partial or full Fortran and C/C++ code portable on accelerators. OpenACC is an emerging interface for parallel programmers to easily create and write simple code that executes on accelerators. In August 2013, the OpenACC standard committee released a second version of the API, OpenACC 2.0. Vendor companies, Cray and PGI, provide compiler support for OpenACC 2.0. CAPS, before they ran out of business, was also providing support for OpenACC 2.0. The model is aiming to port scientific applications to more than one GPU. Several large applications in the fields of geophysics, weather forecast require massive parallel computations and such applications could easily benefit from multiple GPUs. How can we create suitable software that could take advantage of multiple GPUs without losing performance portability? This remains a challenge. In a large cluster, multiple GPUs could reside within a single node or across multiple nodes. If multiple GPUs are used across nodes and each node has one GPU, then the CPU memory associated with the corresponding GPU is independent of the CPU memory associated with another GPU. This makes the communication between the GPUs easier since the CPU memories associated with those GPUs are distributed. However, multiple GPUs could also be in a single node. Since each node in a cluster is usually a shared memory system which has multiple processors, the same shared memory is associated with multiple GPUs. This makes the communication between these GPUs more difficult since they share the same CPU memory and thus the possible memory synchronization issue arises. This paper only focuses on multi-GPU within a single node.

In this paper, we develop strategies to exploit usage of multiple GPUs.

(i) Explore the feasibility of programming multi-GPU using the directive-based programming approaches.

(ii) Evaluate performance obtained by using the OpenMP and OpenACC hybrid model on a multi-GPU platform.

(iii) Propose extensions to OpenACC to support programming multiple accelerators within a single node.

We categorize our experimental analysis into three types: (a) port completely independent routines or kernels to multi-GPU, (b) divide one large workload into several independent subworkloads and then distribute each subworkload to one GPU. In these two cases, there is no communication between the GPUs, and (c) use workload in manner that requires different GPUs to communicate with each other.

The organization of this paper is as follows. Section 2 highlights related work in this area; Section 3 provides an overview of OpenMP and OpenACC directive-based programming models. In Section 4, we will discuss our strategies to develop the hybrid model using OpenMP and OpenACC-based hybrid model and port three scientific applications to multi-GPU within single node with NVIDIA's GPU cards attached. In Section 5 we propose newer extensions to the OpenACC programming model addressing limitations of the hybrid model. Section 6 provides the conclusion of our work.

2. Related Work

Existing research on directive-based programming approach for accelerators focuses on mapping applications to only single GPU. Liao et al. [6] provided accelerator support using OpenMP 4.0 in ROSE compiler. Tian et al. [7] and Reyes et al. [8] developed open source OpenACC compilers, OpenUH, and accULL, respectively. Both their approaches use source-to-source technique to translate OpenACC program to either CUDA program or OpenCL program. Besides compiler development, there is also extensive research [9–11] on the porting applications using OpenACC. These works, however, all utilize only a single GPU, primarily because both OpenACC and OpenMP do not yet provide support for multiple accelerators.

A single programming model may be insufficient to provide support for multiple accelerator devices; however, the user can apply hybrid model to achieve this goal. Hart et al. [12] used Coarray Fortran (CAF) and OpenACC and Levesque et al. [13] used MPI and OpenACC to program multiple GPUs in a GPU cluster. The inter-GPU communication in these works is managed by the Distributed Shared Memory (DSM) model CAF or distributed memory model MPI. Unlike these works, our work uses OpenMP and OpenACC hybrid model. We also develop an approach that extends the OpenACC model to support multiple devices in a single cluster node.

Other works that target multiple devices include OmpSs [14] that allows the user to use their own unique directives in an application so that the program can run on multiple GPUs on either the shared memory system or distributed memory system. StarPU [15] is a runtime library that schedules tasks to multiple accelerators. However, the drawbacks of these approaches are that both OmpSs and StarPU require the user to manually write the kernel that is to be offloaded to the accelerators. Moreover, their approach is not part of any standard thus limiting the usability.

To the best of our knowledge, the only other programming model that supports multiple accelerators without the need to manually write accelerator kernel files is HMPP [3], a directive-based approach. However, HMPP directives are quite challenging in terms of their usability and porting applications to accelerators and even complicated when the applications are large and complex enough.

In this paper, we have adopted a task-based concept by proposing extensions to the OpenACC model to support multiple accelerators. This work is based upon our previous work in [16]. The new work compared to our previous work is the model extension part. Related work that uses tasking concept for GPUs includes Chatterjee et al. [17] who designed a runtime system that can schedule tasks into different Stream Multiprocessors (SMs) in one device. In their system, at a specific time, the device can only execute the same number of thread blocks as the number of SMs (13 in Kepler 20c), thus limiting the performance. This is because their system is designed for tackling load balancing issues among all SMs primarily for irregular applications. Extensions to the OpenACC model were proposed by Komoda et al. [18] to support multiple GPUs. They proposed directives for the user to specify memory access pattern for each data in a computing region and the compiler identifies the workload partition. Typically, it is quite complicated if it is the user that identifies the memory access pattern for all data, especially when a data is multidimensional or accesses multiple index variables. In our extensions to the OpenACC model we allow the user to partition the workload, thus further simplifying the application porting, and make the multi-GPU support general enough to cover most application cases.

3. Overview of OpenACC and OpenMP

OpenMP is a high-level directive-based programming model for shared memory multicore platforms. The model consists of a set of directives, runtime library routines, and environment variables. The user just needs to simply insert the directives into the existing sequential code, with minor changes or no changes to the code. OpenMP adopts the fork-join model. The model begins with an initial main thread, then a team of threads will be forked when the program encounters a parallel construct, and all other threads will join the main thread at the end of the parallel construct. In the parallel region, each thread has its own private variable and does the work on its own piece of data. The communication between different threads is performed by shared variables. In the case of a data race condition, different threads will update the shared variable atomically. Starting from 3.0, OpenMP introduced *task* concept [19] that can effectively express and solve the irregular parallelism problems such as unbounded loops and recursive algorithms. To make the task implementation efficient, the runtime needs to consider the task creation, task scheduling, task switching, task synchronization, and so forth. OpenMP 4.0 released in 2013 includes support for accelerators [20].

OpenACC [5], similar to OpenMP, is a high-level programming model that is being extensively used to port applications to accelerators. OpenACC also provides a set of directives, runtime routines, and environment variables. OpenACC supports three-level parallelism: coarse grain parallelism "gang," fine grain parallelism "worker," and vector parallelism "vector." While mapping the OpenACC three levels of parallelism to the low level CUDA programming model, all of PGI, CAPS [21], and OpenUH [22] map each gang to a thread block, workers to the Y-dimension of a thread block and vector to the X-dimension of a thread block. This guarantees fair performance comparison when using these compilers since they use the same parallelism mapping strategy. The execution model assumes that the main program runs on the host, while the compute-intensive regions of the main program are offloaded to the attached accelerator. In the memory model, usually the accelerator and the host CPU consist of separate memory address spaces; as a result, data being transferred back and forth is an important challenge to address. To satisfy different data optimization purposes, OpenACC provides different types of data transfer clauses in 1.0 specification and possible runtime routines in the 2.0 document. To fully utilize the CPU resource and remove potential data transfer bottleneck, OpenACC also allows asynchronous data transfer and asynchronous computation with the CPU code. Also the model offers an *update* directive that can be used within a data region to synchronize data between the host and the device memory.

4. Multi-GPU Support with OpenMP and OpenACC Hybrid Model

In this section, we will discuss strategies to explore programming multi-GPU using OpenMP and OpenACC hybrid model within a single node. We will evaluate our strategies using three scientific applications. We study the impact of our approach by comparing and analyzing the performances achieved by the hybrid model (multi-GPU implementation) against that of a single GPU implementation.

The experimental platform is a server machine that is a multicore system consisting of two NVIDIA Kepler 20Xm GPUs. The system itself has Intel Xeon x86_64 CPU with 24 cores (12 × 2 sockets), 2.5 GHz frequency, and 62 GB main memory. Each GPU has 5 GB global memory. We use CAPS OpenACC for S3D and PGI OpenACC for matrix multiplication and 2D heat equation. PGI compiler is not used for S3D since it cannot compile the code successfully. The 2D heat equation program compiled by CAPS compiler is extremely long so we do not show the result here. CAPS compiler does compile the matrix multiplication program but we leave the performance comparison with other compilers later and here we only verify the feasibility of hybrid programming model. We use GCC 4.4.7 as CAPS host compiler for all C programs. For a Fortran program, we use PGI and CAPS (pgfortran as the host compiler of CAPS) to compile the OpenACC code. We use the latest versions of CAPS and PGI compilers, 3.4.1 and 14.2, respectively. CUDA 5.5 is used for our experiments. The CAPS compiler performs source-to-source translation of directives inserted code into CUDA code and then calls *nvcc* to compile the generated CUDA code. The flags passed to CAPS compiler are "–nvcc-options -Xptxas=-v,-arch,sm_35,-fmad=false," and the flags passed to PGI compiler are "-O3 -mp -ta=nvidia,cc35,nofma." We consider wall-clock time as the evaluation measurement. We ran all experiments for five times and then averaged the performance results. In the forthcoming subsections, we will discuss both single and multi-GPU implementations for the S3D thermodynamics application kernel, matrix multiplication, and 2D heat equation.

OpenMP is fairly easy to use, since all that the programmer needs to do is to insert OpenMP directives in the appropriate places and, if necessary, make minor modifications to the code. The general idea of an OpenMP and OpenACC hybrid model, as shown in Figure 1, is that we need to manually divide the problem among OpenMP threads and then associate each thread with a particular GPU. The easy case is when the work in each GPU is independent of each other and no communication among different GPUs is involved. But there may be cases where the GPUs will have to communicate with each other and this will involve the CPUs too. Different GPUs transfer their data to their corresponding host threads, these threads then communicate or exchange their data via shared variable, and finally the threads transfer the new data back to their associated GPUs. With the GPU direct technique [23], it is also possible to transfer data between different GPUs directly without going through the host. This has not been plausible in OpenMP and OpenACC hybrid model so far, but in Section 5 we will propose some extensions to the OpenACC programming model to accommodate this feature.

4.1. S3D Thermodynamics Kernel. S3D [24] is a flow solver that performs direct numerical simulation of turbulent combustion. S3D solves fully compressible Navier-Stokes, total energy, species, and mass conservation equations coupled with detailed chemistry. Apart from the governing equations, there are additional constitutive relations, such as the ideal gas equation of state, models for chemical reaction rates, molecular transport, and thermodynamic properties. These relations and detailed chemical properties have been implemented as kernels or libraries suitable for GPU computing. Some research on S3D has been done in [13, 25], but the code they used is not accessible for us. For the experimental purpose of our work, we only chose two separate and independent kernels of the large S3D application, discussed in detail in [26].

```
!$acc data copyout(c(1:np), h(1:np)) copyin(T(1:np),...)
do m = 1, MR
  call calc_mixcp(np, nslvs, T, midtemp, ... , c)
  call calc_mixenth(np, nslvs, T, midtemp, ... , h)
end do
!$acc end data
```

Algorithm 1: S3D thermodynamics kernel in single GPU.

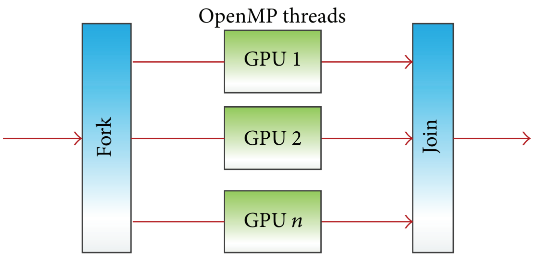


Figure 1: A multi-GPU solution using the hybrid OpenMP and OpenACC model. Each OpenMP thread is associated with one GPU.

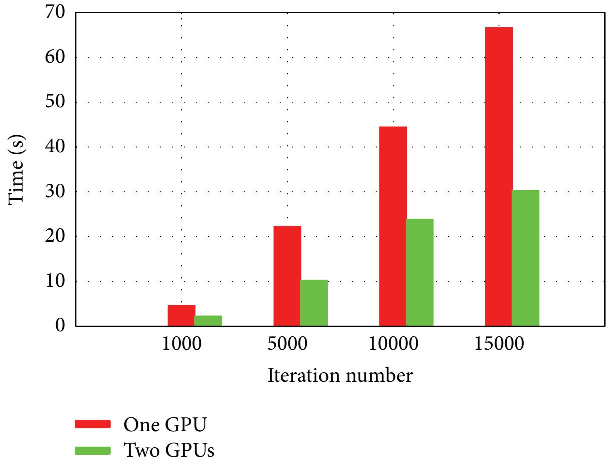


Figure 2: Performance comparison of S3D.

We observed that the two kernels of S3D have similar code structures and their input data are common. Algorithm 1 shows a small code snippet of a single GPU implementation. Both the kernels, *calc_mixcp and calc_mixenth*, are surrounded by a main loop with *MR* iterations. Each kernel produces its own output result, but their results are the same as that of the previous iteration. The two kernels can be executed in the same accelerator sequentially while sharing the common input data, which will stay on the GPU during the whole execution time. Alternatively, they can be also executed on different GPUs simultaneously since they are totally independent kernels.

In order to use multi-GPU, we distribute the kernels to two OpenMP threads and associate each thread with one GPU. Since we have only two kernels, it is not necessary to use *omp for*; instead we use *omp sections* so that each kernel is located in one section. Each thread needs to set the device number using the runtime *acc_set_device_num (int devicenum, acc_device_t devicetype)*. Note that the device number starts from 1 in OpenACC, or the runtime behavior would be implementation-defined if the *devicenum* were to start from 0. To avoid setting the device number in each iteration and make the two kernels work independently, we apply loop fission and split the original loop into two loops. Finally we replicate the common data on both the GPUs. The code snippet in Algorithm 2 shows the implementation for multi-GPU. Although it is a multi-GPU implementation, the implementation in each kernel is still the same as that of a single GPU implementation. Figure 2 shows the performance results of using single GPU and two GPUs. It is observed that two GPUs almost always take approximately half the time taken for a single GPU. This illustrates the performance advantage of using multiple GPUs over single GPU.

4.2. Matrix Multiplication. With S3D, we had distributed different kernels of one application to multiple GPUs. An alternate type of a case study would be where the workload of only one kernel is distributed to multiple GPUs, especially if the workload is very large. We will use square matrix multiplication as an illustration to explore this case study. We chose this application since this kernel is extensively used in numerous scientific applications. This kernel does not comprise complicated data movements and can be parallelized by simply distributing work to different thread. We also noticed a large computation to data movement ratio.

Typically matrix multiplication takes matrix A and matrix B as input and produces matrix C as the output. When multiple GPUs are used, we will use the same amount of threads as the number of GPUs on the host. For instance, if the system has 2 GPUs, then we will launch 2 host threads. Then we partition matrix A in block row-wise which means that each thread will obtain partial rows of matrix A. Every thread needs to read the whole matrix B and produce the corresponding partial result of matrix C. After partitioning the matrix, we use OpenACC to execute the computation of each partitioned segment on one GPU.

Algorithm 3 shows a code snippet for the multi-GPU implementation for matrix multiplication. Here we assume that the number of threads could be evenly divided by the square matrix row size. Since the outer two loops are totally independent, we distribute the i loop into all gangs and

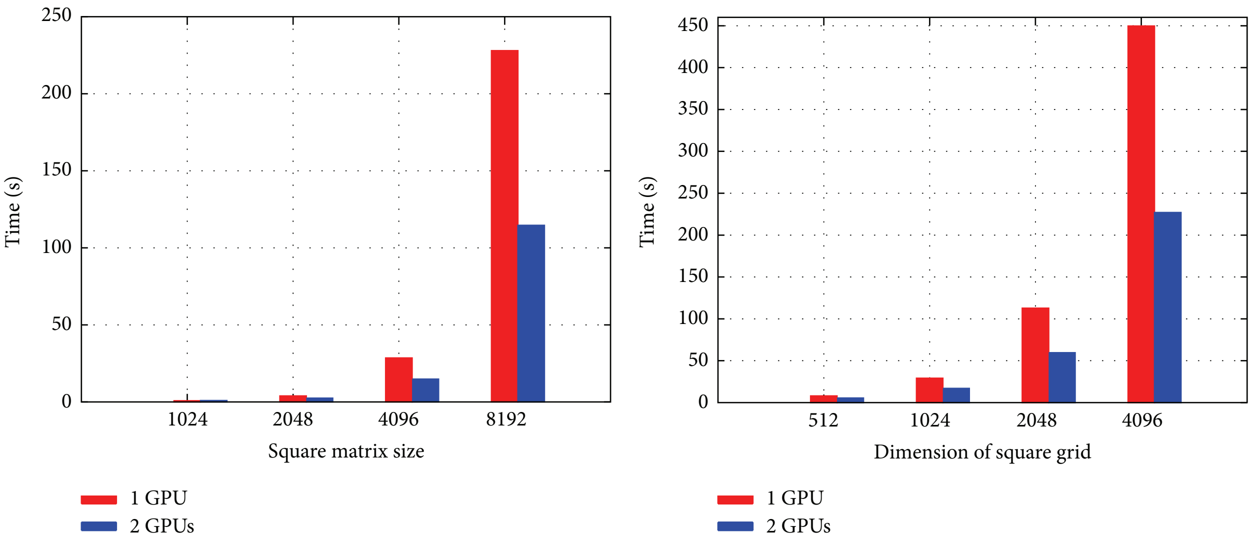
```
call omp_set_num_threads(2)
!$omp parallel private(m)
!$omp sections
!$omp section
call acc_set_device_num(1, acc_device_not_host)
!$acc data copyout(c(1:np)) copyin(T(1:np),...)
do m = 1, MR
  call calc_mixcp(np, nslvs, T, ... , c)
end do
!$acc end data
!$omp section
call acc_set_device_num(2, acc_device_not_host)
!$acc data copyout(h(1:np)) copyin(T(1:np),...)
do m = 1, MR
  call calc_mixenth(np, nslvs, T, ... , h)
end do
!$acc end data
!$omp end sections
!$omp end parallel
```

Algorithm 2: S3D thermodynamics kernel in multi-GPU using hybrid model.

```
omp_set_num_threads(threads);
#pragma omp parallel
{
    int i, j, k;
    int id, blocks, start, end;
    id = omp_get_thread_num();
    blocks = n/threads;
    start = id*blocks;
    end = (id+1)*blocks;
    acc_set_device_num(id+1, acc_device_not_host);
    #pragma acc data copyin(A[start*n:blocks*n])\
                     copyin(B[0:n*n])\
                     copyout(C[start*n:blocks*n])
    {
      #pragma acc parallel num_gangs(32) vector_length(32)
       {
       #pragma acc loop gang
       for(i=start; i<end; i++){
           #pragma acc loop vector
           for(j=0; j<n; j++){
             float c = 0.0f;
             for(k=0; k<n; k++)
                 c += A[i*n+k] * B[k*n+j];
             C[i*n+j] = c;
           }
       }
    }
}
```

Algorithm 3: A multi-GPU implementation of MM using hybrid model.

the j loop into all vector threads of one gang. We have used only 2 GPUs for this experiment; however, more than 2 GPUs can be easily used as long as they are available in the platform. In this implementation, we assume that the number of GPUs can be evenly divided by the number of threads. We use different workload size for our experiments. The matrix dimension ranges from 1024 to 8192. Figure 3(a) shows the performance comparison while using one and two GPUs. For all data size except 1024, the execution time with 2 GPUs is almost half of that with only 1 GPU. For



(a) Matrix multiplication

(b) 2D heat equation

Figure 3: Performance comparison using hybrid model.

1024 ∗ 1024 as the matrix size, we barely see any benefit using multiple GPUs. This is possibly due to the overhead incurred due to the creation of host threads and GPU context setup. Moreover, the computation is not large enough for two GPUs. When the problem size is more than 1024, the multi-GPU implementation shows a significant increase in performance. In these cases, the computation is so intensive that the aforementioned overheads are being ignored.

4.3. 2D Heat Equation. We notice that, in the previous two cases, the kernel on one GPU is completely independent of the kernel on the other GPU. Now we will explore a case where there is communication between different GPUs. One such interesting application is 2D heat equation. The formula to represent 2D heat equation is explained in [27] and is given as follows:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right), \quad (1)$$

where T is temperature, t is time, α is the thermal diffusivity, and x and y are points in a grid. To solve this problem, one possible finite difference approximation is

$$\frac{\Delta T}{\Delta t} = \alpha \left[\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2} \right], \quad (2)$$

where ΔT is the temperature change over time Δt and i, j are indexes in a grid. In this application, there is a grid that has boundary points and inner points. Boundary points have an initial temperature and the temperature of the inner points is also updated. Each inner point updates its temperature by using the previous temperature of its neighboring points and itself. The operation that updates temperature for all inner points in a grid needs to last long enough. This implies that many iterations are needed before arriving at the final stable temperatures. In our program, the number of iterations is 20,000, and we increase the grid size gradually from 512 ∗ 512 to 4096 ∗ 4096. Our prior experience working on single GPU implementation of 2D heat equation is discussed in [11]. Algorithm 4 shows the code snippet for the single GPU implementation. Inside the kernel that updates the temperature, we distribute the outer loop into all gangs and the inner loop into all vector threads inside each gang. Since the final output will be stored in *temp1* after pointer swapping, we just need to transfer this data out to host.

Let us discuss the case where the application uses two GPUs. Algorithm 5 shows the program in detail. In this implementation, *ni* and *nj* are X and Y dimensions of the grid (it does not include boundary), respectively. As shown in Figure 4, we partitioned the grid into two parts along Y dimension and run each part on one GPU. Before the computation, the initial temperature is stored in *temp1_h*, and after updating the temperature, the new temperature is stored in *temp2_h*. Then we swap the pointer so that in the next iteration the input of the kernel points to the current new temperature. Since updating each data point needs its neighboring points from the previous iteration, two GPUs need to exchange the halo data in every iteration. The halo data refers to the data that needs to be exchanged by different GPUs. So far by simply using high-level directives or runtime libraries, data cannot be exchanged directly between different GPUs and the only workaround is to first transfer the data from one device to the host and then from the host to another device. In 2D heat equation, different devices need to exchange the halo data; therefore, the halo data updating would go through the CPU. Because different GPUs use different parts of the data in the grid, we do not have to

```
void step_kernel{...}
{
  #pragma acc parallel present(temp_in[0:ni*nj], temp_out[0:ni*nj]) \
                         num_gangs(32) vector_length(32)
  {
     // loop over all points in domain (except boundary)
     #pragma acc loop gang
     for (j=1; j < nj-1; j++) {
         #pragma acc loop vector
         for (i=1; i < ni-1; i++) {
             // find indices into linear memory
             // for central point and neighbours
             i00 = I2D(ni, i, j);
             im10 = I2D(ni, i-1, j);
             ip10 = I2D(ni, i+1, j);
             i0m1 = I2D(ni, i, j-1);
             i0p1 = I2D(ni, i, j+1);
     // evaluate derivatives
    d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
    d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];
     // update temperatures
    temp_out[i00] = temp_in[i00]+tfac*(d2tdx2 + d2tdy2);
         }
     }
  }
}
#pragma acc data copy(temp1[0:ni*nj]) \
                 copyin(temp2[0:ni*nj])
{
     for (istep=0; istep < nstep; istep++) {
         step_kernel(ni, nj, tfac, temp1, temp2);
         // swap the temp pointers
         temp = temp1;
         temp1 = temp2;
         temp2 = temp;
     }
}
```

ALGORITHM 4: Single GPU implementation of 2D heat equation.

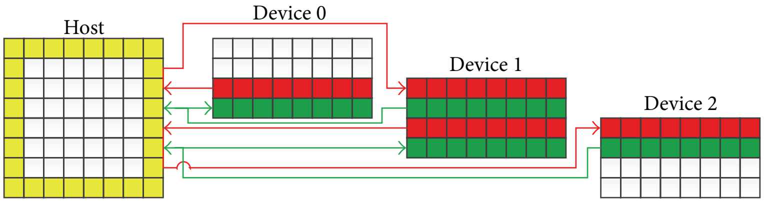


FIGURE 4: Multi-GPU implementation strategy for 2D heat equation using the hybrid model. Consider that there are 3 GPUs (Devices 0, 1, and 2). The grid in the left has 6 rows (excluding boundaries, i.e., the top and the bottom rows). By splitting the 6 rows into 3 parts, each GPU is expected to compute only 2 rows. However, the computation for a data point requires the value of the neighboring points (top, bottom, left, and right data points); hence, simply considering 2 rows of the grid for 1 GPU is not enough. For GPU Device 0, the last row added already has the left, top, and right data points but lacks data points from the bottom; hence, the bottom row needs to be added, leading to 3 rows in total. For GPU Device 1, the first and the second rows do not have data points from the top and the bottom, respectively, hence requiring an addition of the top and bottom rows. This leads to 4 rows in total. For GPU Device 2, the first row does not have data points from the top and requires the addition of the top row. This leads to 3 rows in total. Another point to note is that values in the rows added need to be updated from other GPUs as indicated by the arrows.

```
omp_set_num_threads(NUM_THREADS);
rows = nj/NUM_THREADS;
LDA = ni + 2;
// main iteration loop
#pragma omp parallel private(istep)
{
  float *temp1, *temp2, *temp_tmp;
  int tid = omp_get_thread_num();
  acc_set_device_num(tid+1, acc_device_not_host);
  temp1 = temp1_h + tid*rows*LDA;
  temp2 = temp2_h + tid*rows*LDA;
  #pragma acc data copyin(temp1[0:(rows+2)*LDA]) \
                   copyin(temp2[0:(rows+2)*LDA])
  {
    for(istep=0; istep < nstep; istep++){
      step_kernel(ni+2, rows+2, tfac, temp1, temp2);
      /* all devices (except the last one) update the lower halo to the host */
      if(tid != NUM_THREADS-1){
        #pragma acc update host(temp2[rows*LDA:LDA])
      }
      /* all devices (except the first one) update the upper halo to the host */
      if(tid != 0){
        #pragma acc update host(temp2[LDA:LDA])
      }
      /*  all host threads wait here to make sure halo data from all devices
         have been updated to the host */
      #pragma omp barrier
      /* update the upper halo to all devices (except the first one) */
      if(tid != 0){
        #pragma acc update device(temp2[0:LDA])
      }
      /* update the lower halo to all devices (except the last one) */
      if(tid != NUM_THREADS-1){
        #pragma acc update device(temp2[(rows+1)*LDA:LDA])
      }
      temp_tmp = temp1;
      temp1 = temp2;
      temp2 = temp_tmp;
    }
    /* update the final result to host */
    #pragma acc update host(temp1[LDA:row*LDA])
  }
}
```

Algorithm 5: Multi-GPU implementation with hybrid model of 2D heat equation.

allocate separate memory for these partial data; instead we just need to use private pointer to point to the different positions of the shared variables *temp1_h* and *temp2_h*. Let *tid* represent the id of a thread; then that thread points to the position $tid * rows * (ni + 2)$ of the grid (because it needs to include the halo region), and it needs to transfer $(rows + 2) * (ni + 2)$ data to the device where rows are equal to *nj*/*NUM_THREADS*. The kernel that updates the temperature in the multi-GPU implementation is exactly the same as the one in single GPU implementation.

Figure 3(b) shows the performance comparison of the different implementations, that is, single and multi-GPU implementations. While comparing the performances of multi-GPU with single GPU, we notice that there is a trivial performance difference when the problem size is small. However, there is a significant increase in performance using multi-GPU for larger grid sizes. With the grid size as 4096 * 4096, the speedup of using two GPUs is around 2x times faster than the single GPU implementation. This is because as the grid size increases, the computation also increases significantly, while the halo data exchange is still small enough. Thus, the ratio of the computation/communication becomes larger. Using multi-GPU can be quite advantageous to decompose the computation.

5. Multi-GPU Support with OpenACC Extension

We see that programming using multi-GPU using OpenMP and OpenACC hybrid model shows significant performance benefits in Section 4. However, there are some disadvantages too in this approach. First, the users need to learn two different programming languages which may impact the productivity. Second, in this approach the device-to-device communication happens via the host bringing more unnecessary data movement. Third, providing support for such hybrid model is a challenge for compilers. Compiler A provides support for OpenMP and Compiler B provides support for OpenACC; as a result, it is not straightforward for different compilers to interact with each other. For instance, Cray compiler does not allow OpenACC directives to appear inside OpenMP directives [28]; therefore, the examples in Algorithms 3 and 5 are not compilable by Cray compiler. Although CAPS compiler provides support for OpenACC, it still uses an OpenMP implementation from another host compiler, also a challenge to follow and maintain. Ideally, one programming model should provide support for multi-GPU. Unfortunately the existing OpenACC standard does not yet provide support for multiple accelerator programming. To solve these problems, we propose some extensions to the OpenACC standard in order to support multiple accelerator devices.

5.1. Proposed Directive Extensions. The goal is to help the compiler or runtime realize which device the host will communicate with, so that the host can issue the data movement and kernel launch request to the specific device. The new extensions are described as follows:

(1) `#pragma acc parallel/kernels deviceid` (*scalar-integer-expression*): this is to place the corresponding computing region into a specific device;

(2) `#pragma acc data deviceid` (*scalar-integer-expression*): this is the data directive extension for the structured data region;

(3) `#pragma acc enter/exit data deviceid` (*scalar-integer-expression*); this is the extension for unstructured data region;

(4) `#pragma acc wait device` (*scalar-integer-expression*): this is used to synchronize all activities in a particular device since by default the execution in each device is asynchronous when multiple devices are used;

(5) `#pragma acc update peer to` (*list*) `to_device` (*scalar-integer-expression*) `from` (*list*) `from_device` (*scalar-integer-expression*);

(6) void `acc_update_peer(void* dst, int to_device, const` void* `src, int from_device, size_t size)`.

The purpose of (5) and (6) is to enable device-to-device communication. This is particularly important when using multiple devices, since in some accelerators device can communicate directly with another device without going through the host. If the devices cannot communicate directly, the runtime library can choose to first transfer the data to a temporary buffer in the host and then transfer it from the host to another device. For example, in CUDA implementation, two devices can communicate directly only when they are connected to the same root I/O Hub. If this requirement is not satisfied, then the data transfer will go through the host. (Note that we believe these extensions will address direct device-to-device communication challenge; however, such direct communication also requires necessary support from the hardware. Our evaluation platform did not fulfill the hardware needs; hence, we have not evaluated the benefit of these extensions quantitatively yet and we will do so as part of the future work.)

5.2. Implementation Strategy. We implement the extensions discussed in Section 5.1 in our OpenUH compiler. Our implementation is based on the hybrid model of pthreads + CUDA. CUDA 4.0 and later versions simplify multi-GPU programming by using only one thread to manipulate multiple GPUs. However, in our OpenACC multi-GPU extension implementation, we use multiple pthreads to operate multiple GPUs and each thread is associated with one GPU. This is because the memory allocation and free operations are blocking operations. If a programmer uses data `copy/copyin/copyout` inside a loop, the compiler will generate the corresponding data memory allocation and transfer runtime APIs. Since the memory allocation is blocking operation, the execution in multiple GPUs cannot be parallel. CUDA code avoids this by allocating memory for all data first and then performs data transfer. In OpenACC, however, this is unavoidable because all runtime routines are generated by the compiler and the position of these routines cannot be randomly placed. Our solution is to create a set of threads and each thread manages the context of one GPU which is shown in Figure 5. This is a task-based implementation. Assume we have n GPUs attached to a CPU host, initially the host creates n threads, and each thread is associated with one GPU. Each thread creates an empty first-in first-out (FIFO) task queue which waits to be populated by the host main thread. Depending on the directive type and deviceid clause in the original OpenACC directive annotated program, the compiler generates the task enqueue request for the main thread. The task here means any command issued by the host and executed either on the host or on the device. For example, memory allocation, memory free, data transfer, kernel launch, and device-to-device communication, all of these, are different task types.

Algorithm 8 includes the definitions of the task structure and the thread controlling a GPU (refer it to GPU thread). The task is executed only by GPU thread. A task could be synchronous or asynchronous to the main thread. In the current implementation, most tasks are asynchronous except device memory allocation because the device address is passed from a temporary argument structure, so the GPU thread must wait for this to finish. Each GPU thread manages a GPU context, a task queue, and some data structures in order to communicate with the master thread. Essentially this is still the master/worker model and the GPU threads are

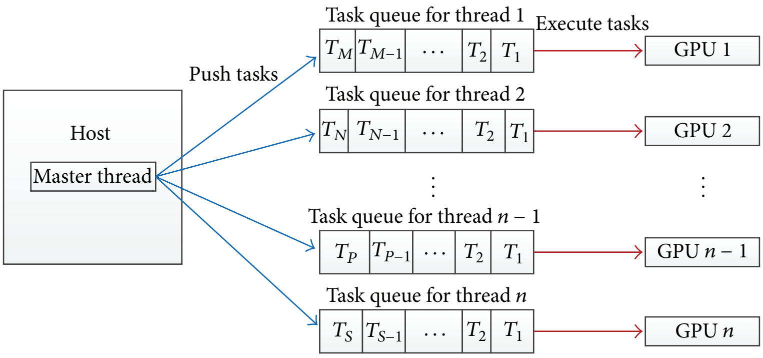

Task queue for thread 1
Execute tasks
T_M T_{M-1} ··· T_2 T_1
GPU 1
Push tasks
Task queue for thread 2
Host
T_N T_{N-1} ··· T_2 T_1
GPU 2
Master thread
Task queue for thread $n-1$
T_P T_{P-1} ··· T_2 T_1
GPU $n-1$
Task queue for thread n
T_S T_{S-1} ··· T_2 T_1
GPU n

Figure 5: Task-based multi-GPU implementation in OpenACC. T_i $(i = 1, 2, \ldots)$ means a specific task.

```
function WORKER_ROUTINE
    Create the context for the associated GPU
    pthread_mutex_lock(···)
    context_created++;
    while context_created! = num_devices do
        pthread_cond_wait(···)   ▷wait until all threads created their contexts
    end while
    pthread_mutex_unlock(···)
    if context_created == num_devices then
        pthread_cond_broadcast(···)
    end if
    Enable peer access among all devices
    while (1) do
        pthread_mutex_lock(&cur_thread → queue_lock)
        while cur_thread → queue_size == 0 do
            pthread_cond_wait(&cur_thread → queue_ready, &cur_thread → queue_lock)
            if cur_thread → destroyed then
                pthread_mutex_unlock(&cur_thread → queue_lock)
                Synchronize the GPU context   ▷the context is blocked until the device has completed all preceding requested tasks
                pthread_exit(NULL)
            end if
        end while
        cur_task = cur_thread → queue_head; ▷fetch the task from the queue head
        cur_thread → queue_size−−;
        if cur_thread → queue_size == 0 then
            cur_thread → queue_head = NULL;
            cur_thread → queue_tail = NULL;
        else
            cur_thread → queue_head = cur_task → next;
        end if
        pthread_mutex_unlock(&cur_thread → queue_lock);
        cur_task → routine((void∗)cur_task → args);          ▷execute the task
    end while
end function
```

Algorithm 6: The worker algorithm for multi-GPU programming in OpenACC.

workers. Algorithms 6 and 7 show a detailed implementation for the worker thread and master thread, respectively.

To enable device-to-device communication, we must enable such peer-to-peer access explicitly and this requires that all worker threads have created the GPU contexts. So each worker first creates the context for the associated GPU, and then it waits until all workers have created the GPU contexts. The worker that is the last one to create the context will broadcast to all worker threads so that they can start to enable the peer-to-peer access. The worker then enters

```
function ENQUEUE_TASK_XXXX
    Allocate memory and populate the task argument
    Allocate memory and populate the task
    pthread_mutex_lock(&cur_thread → queue_lock)
    if cur_thread → queue_size == 0 then        ▷push the task into the FIFO queue
        cur_thread → queue_head = cur_task;
        cur_thread → queue_tail = cur_task;
        pthread_cond_signal(&cur_thread → queue_ready); ▷signal the worker that the queue is not empty and the task is ready
    else
        cur_thread → queue_tail → next = cur_task;
        cur_thread → queue_tail = cur_task;
    end if
    cur_thread → queue_size++;
    pthread_mutex_unlock(&cur_thread → queue_lock);
    if cur_task → async == 0 then                    ▷if the task is synchronous
        pthread_mutex_lock(&cur_thread → queue_lock);
        while cur_task → work_done == 0 do  ▷wait until this task is done
            pthread_cond_wait(&cur_thread → work_done, &cur_thread → queue_lock);
        end while
        pthread_mutex_unlock(&cur_thread → queue_lock);
    end if
end function
```

Algorithm 7: The master algorithm for multi-GPU programming in OpenACC.

```
typedef struct _task_s
{
    int type; //task type (e.g. memory allocation and kernel launch, etc.)
    void* (*routine)(void*); // the task routine
    _work_args *args; // point to the task argument
    int work_done; // indicate whether the task is done
    int async; // whether the task is asynchronous
    struct _task_s *next; // next task in the task queue
} _task;
typedef struct
{
    int destroyed; // whether this thread is destroyed
    int queue_size; // the task queue size
    pthread_t thread; // the thread identity
    context_t *context; // the GPU context associated with this thread
    int context_id; // the GPU context id
    _task *queue_head; // head of the FIFO task queue
    _task *queue_tail; // tail of the FIFO task queue
    pthread_mutex_t queue_lock;
    pthread_cond_t queue_ready; // the task queue is not empty and ready
    pthread_cond_t work_done;
    pthread_cond_t queue_empty;
} _gpu_thread;
```

Algorithm 8

an infinite loop that waits for the incoming tasks. When the task is ready in the FIFO task queue, it will fetch the task from the queue head and execute that task. When there is no task available, the worker just goes to sleep to save CPU resource. Whenever a master pushes a task into the FIFO queue of a worker, it will signal to that worker that the queue is not empty and the task is ready. If the master notifies that the worker be destroyed, the worker will complete all pending tasks and then exit (see Algorithm 6).

5.3. Benchmark Example. In this section, we will discuss how to port some of the benchmarks discussed in Section 4 using the OpenACC extensions instead of using the hybrid model. The programs using the proposed directives are compiled

```
for(d=0; d<num_devices; d++)
{
    blocks = n/num_devices;
    start = id*blocks;
    end = (id+1)*blocks;
    #pragma acc data copyin(A[start*n:blocks*n])\
                     copyin(B[0:n*n])\
                     copyout(C[start*n:blocks*n])\
                     deviceid(d)
    {
      #pragma acc parallel deviceid(d)\
            num_gangs(32) vector_length(32)
       {
       #pragma acc loop gang
       for(i=start; i<end; i++){
          #pragma acc loop vector
          for(j=0; j<n; j++){
            float c = 0.0f;
            for(k=0; k<n; k++)
                c += A[i*n+k] * B[k*n+j];
            C[i*n+j] = c;
          }
       }
    }
}
for(d=0; d<num_devices; d++){
    #pragma acc wait device(d)
}
```

Algorithm 9: A multi-GPU implementation of MM using OpenACC extension.

by OpenUH compiler with "-fopenacc -nvcc,-arch=sm_35,-fmad=false" flag. We also compare the performance with that of the CUDA version. All CUDA codes are compiled using "-O3 -arch=sm_35 -fmad=false" flag.

Algorithm 9 shows a code snippet of multi-GPU implementation of matrix multiplication using the proposed OpenACC extensions. Using the proposed approach, the user still needs to partition the problem explicitly into different devices. This is because if there is any dependence between devices, it is difficult for the compiler to do such analysis and manage the data communication. For the totally independent loop, we may further automate the problem partition in compiler as part of the future work. Figure 6 shows the performance comparison using different models. We can see that the performance of manual CUDA version and OpenACC extension version is much better than that of the hybrid model. CAPS compiler seems to have not performed well at all using the hybrid model implementation. The performance of the proposed OpenACC extension version is the best and it is very close to the optimized CUDA code. There are several reasons for this. First, the loop translation mechanisms from OpenACC to CUDA in different compilers are different. Loop translation means the translation from OpenACC nested loop to CUDA parallel kernel. In the translation step, the OpenACC implementation in OpenUH compiler uses redundant execution model which has no synchronization between different OpenACC parallelism like gang and vector. However, PGI compiler uses another execution model which loads some scalar variables into shared memory in gang parallelism and then the vector threads fetch them from shared memory. The detailed comparison between these two loop translation mechanisms is explained in [29]. OpenUH compiler does not need to do those unnecessary shared memory store and load operations and therefore reduces those overhead. Second, we found that CAPS compiler uses similar loop translation mechanism as OpenUH. However, its performance is still worse than OpenUH compiler. The possible reason is that it has nonefficient runtime library implementation. Since CAPS itself does not provide OpenMP support, it needs complex runtime management to interact with the OpenMP runtime in other CPU compilers. This result demonstrates the effectiveness of our approach that not only simplifies the multi-GPU implementation but also maintains high performance.

We also port the 2D heat equation to the GPUs using the proposed OpenACC directive extension. In the code level, the user does not need to make the device-to-device communication go through the host anymore; instead the `update peer` directive can be used to reduce the code complexity and therefore improve the implementation productivity. Algorithm 10 shows the detailed multi-GPU implementation code using the OpenACC extension and Figure 7 explains this implementation graphically. Compared to Figure 4 that uses the hybrid model, it is obvious to see that the data

```
for(d=0; d<num_devices; d++){
    #pragma acc enter data copyin(temp1_h[d*rows*LDA:(rows+2)*LDA]) device(d)
    #pragma acc enter data copyin(temp2_h[d*rows*LDA:(rows+2)*LDA]) device(d)
}
for(istep=0; istep<nstep; istep++){
    for(d=0; d<num_devices; d++)
        step_kernel_(ni+2, rows+2, tfac, temp1_h+d*rows*LDA, temp2_h+d*rows*LDA)
    }
    /* wait to finish the kernel computation */
    for(d=0; d<num_devices; d++){
        #pragma acc wait device(d)
    }
    /* exchange halo data */
    for(d=0; d<num_devices; d++){
        if(d > 0){
          #pragma acc update peer to(temp2_h[d*rows*LDA:LDA]) to_device(d)
                              from(temp2_h[d*rows*LDA:LDA]) from_device(d-1)
        }
        if(d < num_devices - 1){
        #pragma acc update peer to(temp2_h[(d+1)*rows*LDA+LDA:LDA]) to_device(d)
                         from(temp2_h[(d+1)*rows*LDA+LDA:LDA]) from_device(d+1)
        }
    }
    /* swap pointer of in and out data */
    temp_tmp = temp1_h;
    temp1_h = temp2_h;
    temp2_h = temp_tmp;
}
for(d=0; d<num_devices; d++){
    #pragma acc exit data copyout(temp1_h[(d*rows+1)*LDA:rows*LDA]) deviceid(d)
}
for(d=0; d<num_devices; d++){
    #pragma acc wait device(d)
}
```

Algorithm 10: Multi-GPU implementation with OpenACC extension of 2D heat equation.

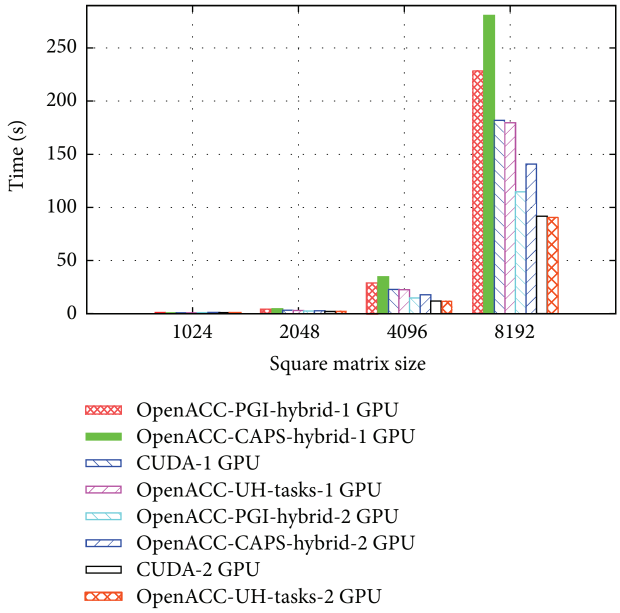


Figure 6: Performance comparison for MM using multiple models. PGI and CAPS compilers compile the hybrid model implementation.

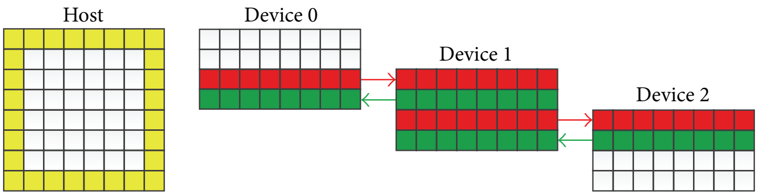


Figure 7: Multi-GPU implementation of 2D heat equation using OpenACC extension.

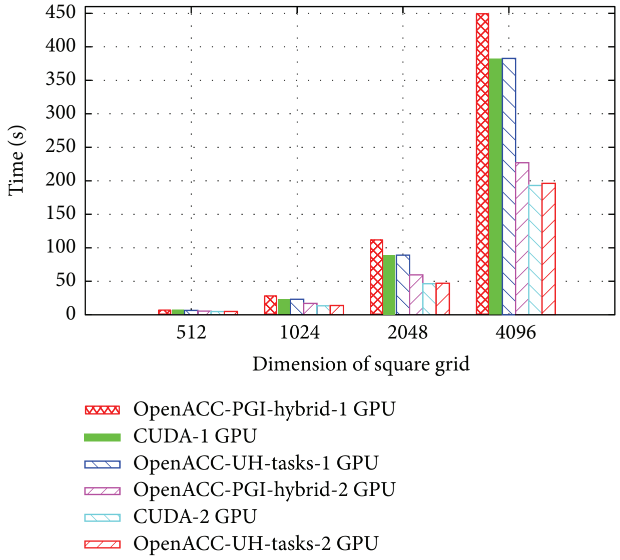


Figure 8: Performance comparison for 2D heat equation using multiple models. PGI compiler compiles the hybrid model implementation.

transfer between devices is much simpler now. Figure 8 shows the performance comparison using different models. The performance of the hybrid model version using CAPS compiler is not shown here because it is too slow, that is, around 5x slower than PGI's performance. When the grid size is 4096 ∗ 4096, the execution time of OpenACC version is around 60 seconds faster than the hybrid model and it is close to that of the optimized CUDA code. We notice that there is almost no performance loss with our proposed directive extension.

6. Conclusion and Future Work

This paper explores the programming strategies of multi-GPU within a single node using the hybrid model, OpenMP and OpenACC. We demonstrate the effectiveness of our approach by exploring three applications of different characteristics. In the first application where there are different kernels, each kernel is dispatched to one GPU. The second application has a large workload that is decomposed into multiple small subworkloads, after which each subworkload is scheduled on one GPU. Unlike the previous two applications that consist of totally independent workloads on different GPUs, the third application has some communication between different GPUs. We evaluated the hybrid model with these three applications on multi-GPU and noticed reasonable performance improvement. Based on the experience gathered in this process, we have proposed some extensions to OpenACC in order to support multi-GPU. By using the proposed directive extension, we can simplify the multi-GPU programming but still obtain much better performance compared to the hybrid model. Most importantly, the performance is close to that of the optimized manual CUDA code.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors would like to thank CAPS for helping them understand its software infrastructure. The authors would also like to thank Oscar Hernandez for his S3D code and Graham Pullan for his lecture notes on heat conduction that helped them understand the application better.

References

[1] CUDA, http://www.nvidia.com/object/cuda_home_new.html.

[2] OpenCL Standard, http://www.khronos.org/opencl.

[3] *HMPP Directives Reference Manual*, (HMPP Workbench 3.1), 2015, https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPP_Directives_ReferenceManual.pdf.

[4] The Portland Group, *PGI Accelerator Programming Model for Fortran and C*, Version 1.3, The Portland Group, 2010.

[5] OpenACC Standard Home, http://www.openacc-standard.org/.

[6] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the openMP accelerator model," in *OpenMP in the Era of Low Power Devices and Accelerators: Proceedings of the 9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, September 16–18, 2013*, vol. 8122 of *Lecture Notes in Computer Science*, pp. 84–98, Springer, Berlin, Germany, 2013.

[7] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman, "Compiling a high-level directive-based programming model for GPG-PUs," in *Languages and Compilers for Parallel Computing: 26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25–27, 2013, Revised Selected Papers*, pp. 105–120, Springer International Publishing, 2014.

[8] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, "accULL: an OpenACC implementation with CUDA and OpenCL support," in *Euro-Par 2012 Parallel Processing*, vol. 7484

of *Lecture Notes in Computer Science*, pp. 871–882, Springer, Berlin, Germany, 2012.

[9] S. Lee and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive exascale computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pp. 1–11, IEEE Computer Society Press, Salt Lake City, Utah, USA, November 2012.

[10] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC—first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*, vol. 7484 of *Lecture Notes in Computer Science*, pp. 859–870, Springer, Berlin, Germany, 2012.

[11] R. Xu, S. Chandrasekaran, B. Chapman, and C. F. Eick, "Directive-based programming models for scientific applications—a comparison," in *Proceedings of the SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC '12)*, pp. 1–9, IEEE, Salt Lake City, Utah, USA, November 2012.

[12] A. Hart, R. Ansaloni, and A. Gray, "Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 5–16, 2012.

[13] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pp. 1–11, IEEE Computer Society Press, Salt Lake City, Utah, USA, November 2012.

[14] J. Bueno, J. Planas, A. Duran et al., "Productive programming of GPU clusters with OmpSs," in *Proceedings of the IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS '12)*, pp. 557–568, IEEE, May 2012.

[15] 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, no. 2, pp. 187–198, 2011.

[16] R. Xu, S. Chandrasekaran, and B. Chapman, "Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model," in *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '13)*, pp. 1169–1176, IEEE, Cambridge, Mass, USA, May 2013.

[17] S. Chatterjee, M. Grossman, A. Sbîrlea, and V. Sarkar, "Dynamic task parallelism with a GPU work-stealing runtime system," in *Languages and Compilers for Parallel Computing*, vol. 7146 of *Lecture Notes in Computer Science*, pp. 203–217, Springer, Berlin, Germany, 2013.

[18] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama, "Integrating multi-GPU execution in an OpenACC compiler," in *Proceedings of the 42nd Annual International Conference on Parallel Processing (ICPP '13)*, pp. 260–269, IEEE, Lyon, France, October 2013.

[19] E. Ayguadé, N. Copty, A. Duran et al., "The design of OpenMP tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[20] Technical report on directives for attached accelerators, 2012, http://openmp.org/wp/openmp-specifications/.

[21] *CAPS OpenACC Parallism Mapping*, 2015, http://exxactcorp.com/index.php/software/prod_list/5.

[22] R. Xu, X. Tian, Y. Yan, S. Chandrasekaran, and B. Chapman, "Reduction operations in parallel loops for GPGPUs," in *Proceedings of the Programming Models and Applications on Multicores and Manycores (PMAM '14)*, pp. 10–20, ACM, New York, NY, USA, 2007.

[23] *NVIDIA Kepler GK110 Architecture Whitepaper*, 2014, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[24] J. H. Chen, A. Choudhary, B. de Supinski et al., "Terascale direct numerical simulations of turbulent combustion using S3D," *Computational Science & Discovery*, vol. 2, no. 1, Article ID 015001, 2009.

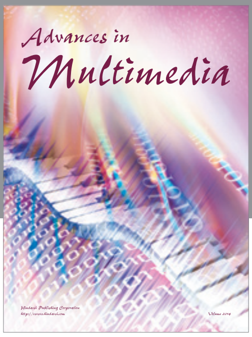
[25] K. Spafford, J. Meredith, J. Vetter, J. Chen, R. Grout, and R. Sankaran, "Accelerating S3D: a GPGPU case study," in *Euro-Par 2009—Parallel Processing Workshops*, vol. 6043 of *Lecture Notes in Computer Science*, pp. 122–131, Springer, Berlin, Germany, 2010.

[26] O. Hernandez, W. Ding, B. Chapman, C. Kartsaklis, R. Sankaran, and R. Graham, "Experiences with high-level programming directives for porting applications to GPUs," in *Facing the Multicore—Challenge II*, vol. 7174 of *Lecture Notes in Computer Science*, pp. 96–107, Springer, Berlin, Germany, 2012.

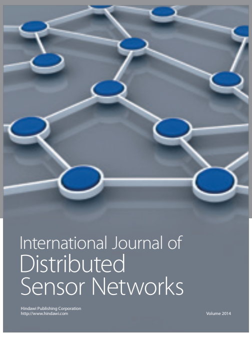
[27] G. Pullan, "Cambridge cuda course 25–27 May 2009," http://www.many-core.group.cam.ac.uk/archive/CUDAcourse09/.

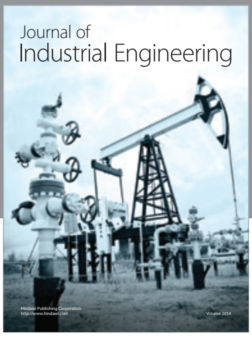
[28] *Cray C and C++ Reference Manual*, 2014, http://docs.cray.com/books/S-2179-81/S-2179-81.pdf.

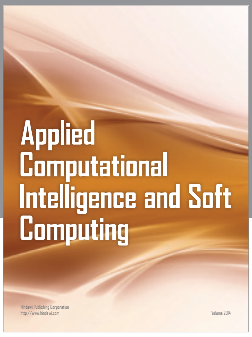
[29] R. Xu, M. Hugues, H. Calandra, S. Chandrasekaran, and B. Chapman, "Accelerating Kirchhoff migration on GPU using directives," in *Proceedings of the 1st Workshop on Accelerator Programming Using Directives (WACCPD '14)*, pp. 37–46, IEEE, 2014.

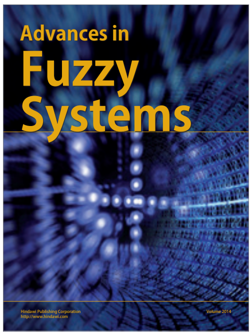

Advances in
Multimedia

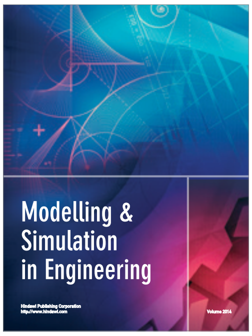
The Scientific World Journal

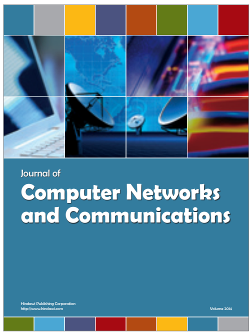

International Journal of
Distributed Sensor Networks

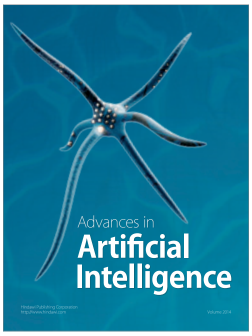

Journal of
Industrial Engineering

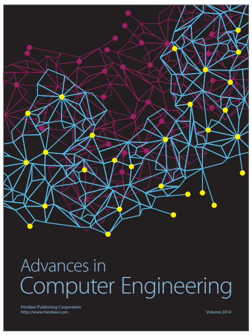

Applied Computational Intelligence and Soft Computing


Advances in
Fuzzy Systems

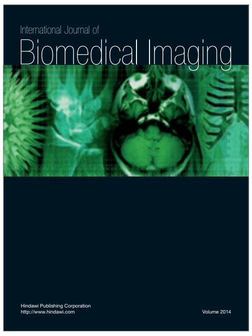

Modelling & Simulation in Engineering

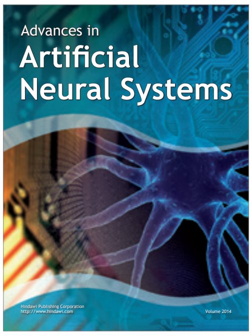

Journal of
Computer Networks and Communications

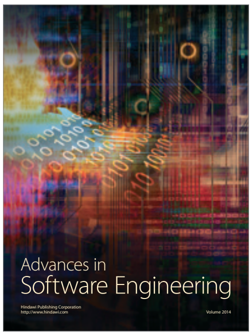

Advances in
Artificial Intelligence


Advances in
Computer Engineering

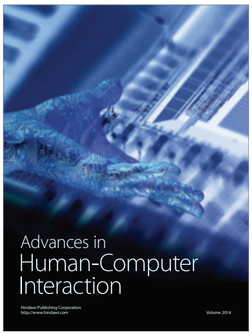
International Journal of
Computer Games Technology

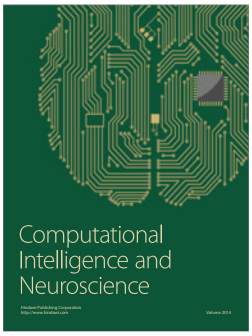

International Journal of
Biomedical Imaging


Advances in
Artificial Neural Systems


Advances in
Software Engineering

Journal of
Robotics


Advances in
Human-Computer Interaction


Computational Intelligence and Neuroscience

International Journal of
Reconfigurable Computing

Journal of
Electrical and Computer Engineering