Exploring Programming Multi-GPUs using OpenMP & OpenACC-based Hybrid Model

Rengan Xu, Sunita Chandrasekaran, Barbara Chapman
Dept. of Computer Science
University of Houston
Houston, USA
Email: {uhxrg, sunita, chapman}@cs.uh.edu

***Abstract*—Heterogeneous computing come with tremendous potential and is a leading candidate for scientific applications that are becoming more and more complex. Accelerators such as GPUs whose computing momentum is growing faster than ever offer application performance when compute intensive portions of an application are offloaded to them. It is quite evident that future computing architectures are moving towards hybrid systems consisting of multi-GPUs and multi-core CPUs. A variety of high-level languages and software tools can simplify programming these systems. Directive-based programming models are being embraced since they not only ease programming complex systems but also abstract low-level details from the programmer. We already know that OpenMP has been making programming CPUs easy and portable. Similarly, a directive-based programming model for accelerators is OpenACC that is gaining popularity since the directives play an important role in developing portable software for GPUs. A combination of OpenMP and OpenACC, a hybrid model, is a plausible solution to port scientific applications to heterogeneous architectures especially when there is more than one GPU on a single node to port an application to. However OpenACC meant for accelerators is yet to provide support for multi-GPUs. But using OpenMP we could conveniently exploit features such as *for* and *section* to distribute compute intensive kernels to more than one GPU. We demonstrate the effectiveness of this hybrid approach with some case studies in this paper.**

***Keywords*-Accelerators; GPU; OpenACC; OpenMP**

I. Introduction

Heterogeneous architecture has gained great popularity over the past several years. These heterogeneous architectures usually comprises of accelerators that are attached to the host CPUs, such accelerators could include GPUs, DSPs and FPGA. These heterogeneous architecture can leverage the power of these accelerators while preserving the capabilities of CPU. 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[3] 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. This affects productivity. To overcome these difficulties, a number of high-level directive-based programming models have been proposed that includes HMPP [2], PGI[7] and OpenACC. These models can be used by inserting directives and runtime calls into the existing source code, making partial or full Fortran and C/C++ code portable on accelerators. OpenACC Version 1.0 is available for use and Version 2.0 is being defined currently. OpenACC is an emerging interface for parallel programmers to easily write simple code that executes on GPU. The standard follows the OpenMP model. OpenACC is yet to provide support for porting scientific applications on more than one GPU. There are large applications especially in the fields of geophysics, weather forecast that requires massive parallel computation demanding usage of the several hardware resources available. Such applications could see several orders of performance speedup if multi-GPUs are used.

In this paper, we provide a high-level directive-based hybrid model (OpenACC & OpenMP) solution that allows the programmers to exploit the additional resources available by using multi-GPUs. The main contributions of this paper include:

- Explore the feasibility of programming multi-GPUs using the OpenACC programming model.
- Compare the performance obtained from OpenMP & OpenACC hybrid model with that of the a single GPU and multi-core platform.
- Propose extensions to OpenACC to support programming multiple accelerators within single node.

We categorize our experimental analysis into three types, first port completely independent routines or kernels to multi-GPUs, secondly divide one large workload into several independent sub-workloads and then distribute each sub-workload to one GPU. In these two cases, communication between the GPUs do not happen. The third type of analysis is similar to either the first or the second, but different GPUs will have to communicate with each other. Based on the results and performance achieved we will also propose extensions to the OpenACC model so that it can support the programming of multi-GPUs in a single node of a cluster.

The organization of this paper is as follows: Section II highlights related work in this area, Section III provides an overview of OpenMP and OpenACC directive-based programming models. In Section IV, we will discuss details about how to use OpenMP & OpenACC-based hybrid model to port three scientific applications to multi-GPUs within single node that has NVIDIA's GPU cards attached. We conclude the results of our work in Section VI.

II. Related Work

To overcome the obstacles of GPU programming, many directive-based high-level programming models have been proposed. *hi*CUDA [8] allows the user to have full control of data transfer management including the memory allocation and de-allocation, explicit data movement, loop scheduling and the use of cache. CUDA-lite [18] can automatically apply memory coalescing in global memory via annotations to optimize memory access pattern. OpenMPC [11] generates the GPU kernels from OpenMP parallel regions, and then applies various optimizations to optimize the memory access pattern in global memory and data transfer in different memory hierarchy. CAPS released its HMPP compiler which is a source-to-source compiler that can translate directive-associated functions or code portions into CUDA or OpenCL kernels. PGI accelerator programming model provides a set of directives to implicitly manage data, computation and loop mapping. All of these directive-based models have their unique syntax and features. OpenACC is an emerging model that is addressing both portability and productivity of GPU programming.

This section highlights some of the related work about programming GPUs using OpenACC. Hernandez et al. [10] used HMPP and PGI directives to port two programs S3D thermodynamics kernel and HOMME/SE application on single GPU and compared the performance achieved against CUDA and OpenMP implementations. We extended their work by porting S3D to multi-GPUs so that more powerful computing resources could be used. Hart et al. [9] described their OpenACC experiences by porting Himeno benchmark to run on the Cray XK6 hybrid supercomputer. They not only implemented the basic functionality of the benchmark but also tuned the performance. Especially, the asynchronous kernel execution and data transfer improved the performance by 5%-10%.

To compare the performance achieved by OpenACC to high-level models, research has been done to evaluate this model using different scientific applications. Lee et al. [12] evaluated the mainstream directive-based model by porting thirteen applications from various scientific domains into GPUs, and they also discussed in detail, the feature set and limitations of each of the models. [19] evaluated the programmability and productivity of two real-world applications, by using OpenACC, PGI Accelerator and OpenCL. Although in one of the applications, the performance of the OpenACC implementation was not comparable with that of the already-well-in-place OpenCL implementation, the authors discussed the promising approaches that OpenACC could take. The performance gap could be addressed with the addition of other adequate OpenACC directives. accULL [17] is the first open source OpenACC compiler that has already implemented some major directives and runtime calls of OpenACC. It used YaCF compiler framework [16] and a standalone runtime library Frangollo that is independent of any compiler. This implementation supports both CUDA and OpenCL platforms. accULL is yet to achieve performance as that of CUDA.

III. Overview of OpenMP and OpenACC Programming Models

OpenMP is an Application Programming Interface for multi-core platform shared memory programming, which 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 change or no change at all. 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 [5] which 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 and task synchronization, etc. OpenMP 4.0 targets to add support for accelerators [4].

OpenACC is an emerging model that is working towards establishing a standard in directive-based accelerator programming. It is a high-level programming model that can be used to port existing HPC applications on different types of accelerators without too much effort. Similar to OpenMP, it also provides directives, runtime routines and environment variables. OpenACC supports three level parallelism: coarse grain parallelism ”gang” (similar to blocks in CUDA), fine grain parallelism ”worker” (similar to groups of warps in CUDA) and vector parallelism ”vector” (similar to a warp in CUDA). 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, so the back and forth data transfer is an important issue. To satisfy different data optimization purposes, OpenACC provides different types of data transfer clauses in 1.0

specification and possible runtime routines are proposed in the 2.0 document. To fully use the CPU resource and remove the potential data transfer bottleneck, OpenACC also allows asynchronous data transfer and asynchronous computation with the CPU code, until all asynchronous activities are synchronized. Also the *update* directive can be used within a data region to synchronize data between the host and the device memory.

IV. Porting Applications to multi-GPUs: Experimental Analysis

In this section, we will discuss the strategies to explore programming multi-GPUs using the hybrid model within a single node of a cluster. We will support our strategies using three scientific applications. In order to evaluate the impact of our strategy, we compare the performance achieved by the multi-GPU implementation against the single GPU implementation. And to evaluate the OpenMP & OpenACC hybrid model, we further compare the performance against pure OpenMP implementation. The OpenMP performance is evaluated using 8 threads. The experimental platform is a server machine that is a multi-core system consisting of two NVIDIA C2075 GPUs. Configuration details are shown in Table I. We use CAPS HMPP compiler's implementation of OpenACC standard. For all C programs, GCC 4.4.7 is used to compile the parallelized OpenMP version; the same is also used for the HMPP host compiler. For a Fortran program, PGI compiler is used to compile the OpenMP code, and we use PGI and HMPP (pgfortran as the host compiler of HMPP) as the host compiler to compile the OpenACC code. We use the latest versions of HMPP and PGI compilers; 3.3.0 and 12.9 respectively. CUDA 5.0 is also used for our experiments. The HMPP compiler performs source-to-source translation of directives inserted code into cuda code, and then calls *nvcc* to compile the generated cuda code. We consider wall-clock time as the evaluation measurement. We will discuss both the single GPU and the multi-GPU implementations for the S3D Thermodynamics application kernel, matrix multiplication and 2D heat conduction.

Table I: Specification of experiment machine

Item	Description
Architecture	Intel Xeon x86_64
CPU Core(s)	16 (8 x 2 sockets)
CPU frequency	2.27GHz
Main memory	32GB
GPU Model	Tesla C2075
GPU cores	448
GPU clock rate	1.15GHz
GPU global & constant memory	5375MB & 64K
Shared memory per block	48KB

OpenMP is fairly easy to use, since all that the programmer needs to do is insert OpenMP directives at appropriate places and if necessary, make minor modifications to the code. The general idea of an OpenMP & OpenACC hybrid model is that we need to manually divide the problem among

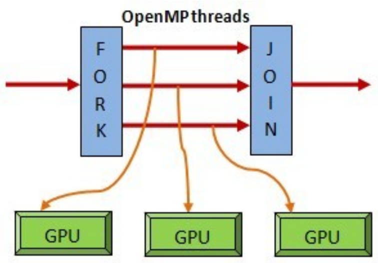

Figure 1: A Multi-GPU Solution Using the Hybrid OpenMP & OpenACC model

OpenMP threads, and then associate each thread with a particular GPU.

Figure 1 demonstrates the idea where each GPU implementation is performed using OpenACC. The best case scenario is when the work in each GPU is independent of the other and does not involve communication among the GPUs. But in some cases, these GPUs have to communicate with each other and the communication involves 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 technology, it is also possible to transfer the data between different GPUs directly without going through the host. But this depends on the runtime implementation.

A. S3D Thermodynamics Kernel

S3D [6] 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 that are suitable for GPU computing. In our work, we chose this thermodynamics kernel in [10] for evaluation purposes.

Figure 2 shows a portion of a single GPU implementation. Both the kernels 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. We perform multiple iterations to increase the workload so that we are able to observe the performance achieved by both the implementations; single and multi-GPU implementation. We have considered a portion of the large S3D application that for experimental purposes, this application has already been well discussed in [10]. The two kernels in this application have similar code structures and the input data is common. So 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. On the other hand, they can be executed on different GPUs simultaneously since they are totally independent kernels.

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

Figure 2: S3D Thermodynamics Kernel in Single GPU

To use multi-GPUs, 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 to split the original loop into two loops. Finally we replicate the same common data on both the GPUs. The code snippet in Figure 3 shows the implementation for multi-GPUs. Although it is a multi-GPU implementation, the implementation in each kernel is still as the same as that of a single GPU implementation. Figure 4 shows the speedup using single GPU and two GPUs against the execution time measured for 8 cores using OpenMP. It is observed that in all the cases, the speedup with two GPUs is always two times the speedup of a single GPU, and the single GPU speedup is more than 4x than that of an OpenMP code as shown in the figure. So this implies that the results for single GPU is much better than the OpenMP results. The speedup of the multi-GPU implementation is definitely better than a single GPU. A point to note here is that every iteration has the same amount of workload, hence we always notice that multi-GPU execution takes approximately half the time taken for a single GPU.

B. Matrix Multiplication

In the previous case, we distributed different kernels of one application to multi-GPUs. An alternate type of a case study would be where the workload of only one kernel is distributed to multi-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 because this kernel is extensively used in numerous scientific applications. This kernel does not comprise of complicated data movement activities and parallelization can happen by distributing work to different threads and we also noticed a high computation to data movement ratio. Matrix

```
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
```

Figure 3: S3D Thermodynamics Kernel in Multi-GPUs

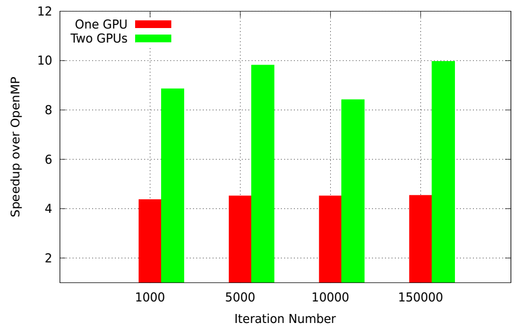

Figure 4: Performance Comparison of S3D

multiplication takes matrix A and matrix B as input, and produces matrix C as the output. When multi-GPUs are used, we will use the same amount of threads as the number of GPUs on the host. 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. Some OpenMP implementations use similar approach, but in our case we partition the matrix manually.

After partitioning the matrix, the computation of each partitioned segment is executed on one GPU using OpenACC Figure 5 shows the code snippet of 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. We also collapse the two outermost loops since their iterations are tightly nested and totally independent. We have only used 2 GPUs for this experiment, however more than 2 GPUs could be easily used as long as they are available in the platform and the number of GPUs could be evenly divided by the number of threads. Our experiments run with different workload size, in which the matrix dimension ranges from 1000 to 150000. Figure 6 shows the performance comparison while using one and two GPUs. For all the problem sizes, the single GPU performance is much better than the OpenMP code. When the square matrix size is 1000, we noticed that the speedup obtained by two GPUs was a little lesser than that of a single GPU, possibly due to the overhead incurred because of the host threads creation and GPU context setup. Moreover the computation is not large enough for two GPUs. When the problem size is more than 1000, the multi-GPU implementation shows a significant performance increase. In these cases, the computation is so intensive that the aforementioned overheads are ignored.

```
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 kernels loop collapse(2) private(j,k)
      for(i=start; i<end; i++)
         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;
         }
    }
}
```

Figure 5: A Multi-GPU Implementation of M*M

C. 2D Heat Conduction

We notice that in the previous two cases, the kernel on one GPU is completely independent from the kernel on another GPU. Now we will explore a case where the application has communication occurring between different GPUs. One such interesting application is 2D heat conduction. The formula to represent 2D heat conduction is explained in[15] and is

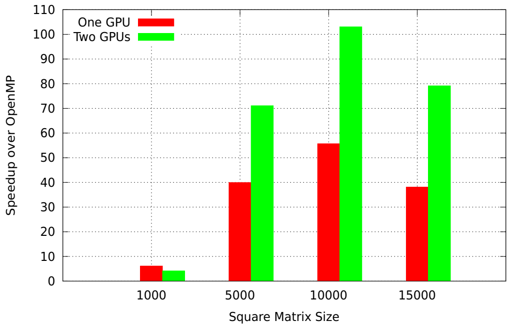

Figure 6: Performance Comparison of MM

given as follows:

$$\frac{\partial T}{\partial t} = \alpha(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2})$$

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:

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

where ΔT is the temperature change over time Δt and i, j are indices 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 are also updated. Each inner point updates its temperature by using the previous temperature of its neighboring points and itself. The temperature updating operation for all inner points in a grid needs to last long enough which means many iterations is required to get the final stable temperatures. In our program, the number of iterations is 20000, and we increase the grid size gradually from 512*512 to 4096*4096. We have prior experience working on the single GPU implementation [20], Figure 7 shows the code snippet for the single GPU implementation where we force the pointer swapping operation happen only on GPU side, by declaring the intermediate pointer *temp_tmp* as a device pointer. Inside the temperature updating kernel, we used the collapse optimization to increase the independent loop iteration space. Since the final output will be stored in *temp1* after pointer swapping, we just need to transfer this data out to host.

We will next discuss the application when it uses two GPUs. Figure 8 shows the code in detail. In this implementation, *ni* and *nj* are X and Y dimension of the grid, respectively. As shown in Figure 9, 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 temperature updating, 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. Because 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 is referred to the data that needs to be exchanged by different GPUs. So far there is no way to exchange the data between different GPUs directly using high-level directives or runtime library, 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 allocate separate memory for these partial data, instead we just need to use private pointer to point to the different position of the shared variable *temp1_h* and *temp2_h*. The first thread points to the start of the grid and the second thread points to the position $(nj/2 - 1) * ni$ of

```
void step_kernel{...}
{
  #pragma acc kernels \
  present(temp_in[0:ni*nj], temp_out[0:ni*nj])
  {
    // loop over all points in domain (except boundary)
    #pragma acc loop collapse(2) independent
    for (j=1; j < nj-1; j++) {
        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]) \
                 deviceptr(temp)
{
    for (istep=0; istep < nstep; istep++) {
        step_kernel(ni, nj, tfac, temp1, temp2);
        // swap the temp pointers
        temp = temp1;
        temp1 = temp2;
        temp2 = temp;
    }
}
```

Figure 7: Single GPU Implementation of 2D Heat Conduction

```
omp_set_num_threads(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*(nj/2-1)*ni;
  temp2 = temp2_h + tid*(nj/2-1)*ni;

  #pragma acc data copyin(temp1[0:(nj/2+1)*ni]) \
                   copyin(temp2[0:(nj/2+1)*ni]) \
                   deviceptr(temp_tmp)
  {
    for(istep=0; istep < nstep; istep++){
      step_kernel(ni, nj/2+1, tfac, temp1, temp2);
      temp_tmp = temp1;
      temp1 = temp2;
      temp2 = temp_tmp;

      if(tid == 0){
        #pragma acc update host(temp1[(nj/2-1)*ni:ni])
      } else{
        #pragma acc update host(temp1[ni:ni])
      }

      /*make sure another device has already
        updated the data into host*/
      #pragma omp barrier
      if(tid == 0){
        #pragma acc update device(temp1[(nj/2)*ni:ni])
      } else{
        #pragma acc update device(temp1[0:ni])
      }
    }
    /*update the final result to host*/
    #pragma acc update host(temp1[tid*ni:(nj/2)*ni])
  }
}
```

Figure 8: Multi-GPU Implementation-2D Heat Conduction

the grid (because it needs to include the halo region). Since both the parts need the halo data for calculation, they will transfer $(nj/2+1)*ni$ elements to the GPU. The temperature updating kernel in the multi-GPU implementation is exactly the same as the one in single GPU implementation.

Figure 10 shows the performance comparison of the different implementations, i.e. single and multi-GPU implementations. When the grid size is 512*512, both OpenACC implementations are a bit slower than OpenMP, but the speedup increase is noticeable for larger grid size. While comparing the performances of multi- GPUs to single GPU, we notice that there is no difference when the problem size is small. But the multi-GPU implementation demonstrated significant performance increase when the grid size is 4096*4096. This is because as the grid size increases, the computation also increases significantly, while the halo data exchange is still small enough. Thus the computation/communication ratio becomes larger. This is very advantageous when we use multi-GPUs to decompose the computation. In the future, we may also try the ghost zone optimization that could include redundant computation by communicating a larger halo and reducing the number of synchronization[14].

V. Proposed Directive

In this section, we propose an extension to OpenACC to support multi-GPUs based on our experimental analysis and evaluation of the three scientific applications. The next step would be to automate this strategy. In this section we propose the following directive-based extension to OpenACC so that the model could provide support for multi-GPUs:
#pragma acc multi_device [clause [[,] clause]...] new-line structured-block

where *clause* is one of the following:
devices [(scalar-integer-expression)]
if (condition)
async [(scalar-integer-expression)]
copy (list)
copyin (list)
copyout (list)
create (list)

The *devices* clause is to specify the number of devices to

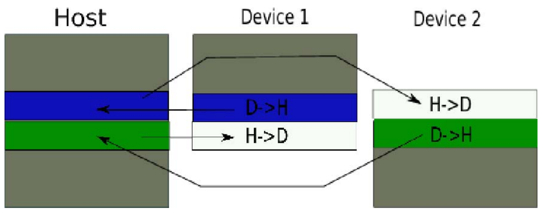

Figure 9: A Multi-GPU Implementation Strategy for 2D Heat Conduction. The left grid is split into two grids on the right. Note that each grid on the right has one halo size more than the half of the left grid. After each iteration, the blue halo data in device 1 and green halo data in device 2 would be transferred to their associated region in host grid, then the new data would be transferred to white halo region in both devices. This is a halo exchange step. Before halo is exchanged, a communication barrier between the host threads is needed to ensure that new halo data have already been on the host.

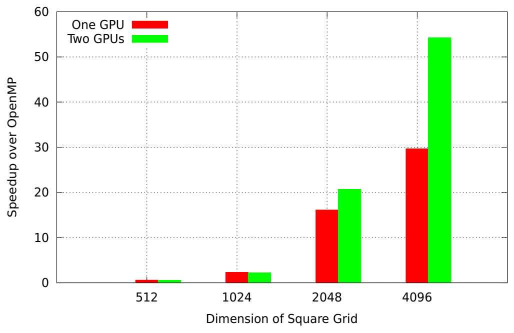

Figure 10: Performance Comparison of 2D Heat Conduction

execute the following code block. The *if* clause is optional; when there is no *if* clause, the compiler will generate the code to execute the following code on the host. When an *if* clause appears, the program will conditionally execute the following code block on multi-GPUs. If the *async* clause is not present, there is an implicit barrier at the end of *multi_device* region, and the host will wait until all kernels have completed execution. If *async* clause is present, then the following code will execute asynchronously with the host code. The *multi_device* construct also comes with some data clauses to control the data transfer between CPU and GPU.

In the code block followed by the *multi_device* construct, there might be one loop nest or multiple loop nests. These loops can also further use the *loop* directive of OpenACC to do some loop optimization. If multiple loop nests exist in the code block, then the compiler will do the dependence analysis to check whether there is dependence between these loops. If there is no dependence, the loops will be scheduled on multi-GPUs. Different scheduling strategies can be applied, for instance blocking scheduling and cyclic scheduling. If there is only one loop nest inside the code block, the compiler will split the loop iterations among the multi-GPUs. The split scheduling is usually block distribution because consecutive data should reside on the same GPU. The compiler front end will parse this directive and its clauses and convert them into runtime calls. We are currently exploring implementation of this proposed directive in our compiler OpenUH [13].

VI. Conclusion and Future Work

This papers explores the programming strategies of multi-GPUs within one node of a cluster using the hybrid model, OpenMP & 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 sub-workloads, after which each sub-workload 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-GPUs and noticed orders of performance improvement. Based on the experience gathered in this process, we have proposed some extensions to OpenACC in order to support multi-GPUs. As part of the future work, we will be performing extensive research and implement the proposed solution in an open source compiler for OpenACC to support multi-GPUs.

Acknowledgment

We would like to thank CAPS to help us understand its software infrastructure. We would also like to thank Oscar Hernandez for his S3D code, and Graham Pullan for his lecture notes on heat conduction that helped us understand the application better.

References

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

[2] HMPP Directives Reference Manual (HMPP Workbench 3.1).

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

[4] Technical report on directives for attached accelerators (november 2012). http://openmp.org/wp/openmp-specifications/.

[5] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.

[6] J.H. Chen, A. Choudhary, B. De Supinski, M. DeVries, ER Hawkes, S. Klasky, WK Liao, KL Ma, J. Mellor-Crummey, N. Podhorszki, et al. Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D. *Computational Science & Discovery*, 2(1):015001, 2009.

[7] The Portland Group. PGI Accelerator Programming Model for Fortran and C (v1.3), 2010.

[8] T. Han and T.S. Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM.

[9] A. Hart, R. Ansaloni, and A. Gray. Porting and Scaling OpenACC Applications on Massively-parallel, GPU-accelerated Supercomputers. *The European Physical Journal-Special Topics*, 210(1):5–16, 2012.

[10] O. Hernandez, W. Ding, B. Chapman, C. Kartsaklis, R. Sankaran, and R. Graham. Experiences with High-level Programming Directives for Porting Applications to GPUs. *Facing the Multicore-Challenge II*, pages 96–107, 2012.

[11] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society, 2010.

[12] S. Lee and J.S. Vetter. Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 23. IEEE Computer Society Press, 2012.

[13] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.

[14] J. Meng and K. Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*, pages 256–265. ACM, 2009.

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

[16] R. Reyes and F. de Sande. Optimization Strategies in Different CUDA Architectures Using llCoMP. *Microprocessors and Microsystems*, 36(2):78–87, 2012.

[17] R. Reyes, I. López-Rodríguez, J. Fumero, and F. de Sande. accULL: An OpenACC Implementation with CUDA and OpenCL Support. *Euro-Par 2012 Parallel Processing*, pages 871–882, 2012.

[18] S.Z. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu. CUDA-lite: Reducing GPU Programming Complexity. *Languages and Compilers for Parallel Computing*, pages 1–15, 2008.

[19] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC-First Experiences with Real-World Applications. *Euro-Par 2012 Parallel Processing*, pages 859–870, 2012.

[20] R. Xu, S. Chandrasekaran, and B. Chapman. Directive-based Programming Models for Scientific Applications - A Comprison. under publication, 2012.