Directive-based Programming Models for Scientific Applications - A Comparison

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

***Abstract*—Accelerators have been considered a viable way by many scientific and technical programmers to program and accelerate huge scientific applications. Accelerators such as GPUs have immense potential in terms of high compute capacity but programming these devices is a challenge. CUDA, OpenCL and other vendor-specific models are definitely a way to go, but these are low-level models that demand excellent programming skills; moreover, they are time consuming to write and debug. In order to simplify GPU programming several directive-based programming models have already been proposed. In this paper, we evaluate and compare several directive-based models such as PGI, HMPP and OpenACC models involving four scientific applications. From our experimental analysis, we conclude that efficient implementations of high-level directive-based models plus user guided optimizations can actually reach the performance obtained via a hand written CUDA code. For example a computer tomography-based algorithm ported to GPUs using a directive-based approach showed that the performance achieved is about 90% to that of CUDA version of the code.**

***Keywords*-Accelerator (GPUs); Directive-based Programming Models, Optimization Strategies**

I. Introduction

Computer architecture relying on heterogeneous systems has gained significant importance over the last decade. Accelerators attached to the host (CPU) could vary from GPUs, DSPs and FPGAs. Such new heterogeneous architecture can leverage the power of these accelerators while preserving the capabilities of CPU. While heterogeneous architectures help in increasing the computational power significantly, they also pose potential challenges for programmers before the capabilities of these new architectures could be exploited. The CUDA programming model [1] of NVIDIA and Brook+ (implementation of Brook [7]) of AMD release the power of GPUs from graphical domain, making it also applicable in general purpose computing. Both CUDA and Brook+, however, are low-level programming models that non-expert programmers find difficult to use. The programmer needs to thoroughly understand the underlying architecture and specify every detail that is required by the programming model. A number of high-level directive-based programming models have emerged recently from both vendors and academia. Recently, several directive-based models for GPUs have been proposed, they are HMPP[2], PGI[12], hiCUDA[13], and a fairly newer model OpenACC[3]. HMPP/PGI is to CUDA[1] what OpenMP[5] is to pthreads[22]. OpenACC is working towards establishing a programming standard for parallel computing developed by Cray, CAPS, NVIDIA and PGI. These directive-based models offer a high-level of abstraction such that the programmer does not need to be aware of low-level details of the architectures. Although these directive-based programming models may ease programmability, it is still a challenge to achieve the speedup that a hand-written CUDA code could achieve.

The main motivation behind our research is to study the feasibility and applicability of these directive-based models when they are applied to scientific applications consisting of varied characteristics. These models allow programming without the need to explicitly manage the data transfer between CPU and the GPU, device start-up and shut-down to name a few. We will explore three directive-based programming models, HMPP, PGI and OpenACC and assess the models using different scientific applications. We will compare and contrast the performance achieved by these directives to that of the corresponding hand-written CUDA version of the applications. Using the novel OpenACC model, the programmer does not write several versions of the code for GPUs, unlike for HMPP and PGI models which makes OpenACC highly attractive. However as OpenACC has been just recently proposed, its benefits has not been systematically evaluated yet. This paper is one of the first to evaluate the OpenACC model.

The main contributions of this paper include:

- Assess and evaluate three directive-based programming models, HMPP, PGI and OpenACC with respect to their run-time performance, program complexity, and ease of use.
- Compare the performance obtained of the three programming models with that of the native CUDA and the sequential version.
- The OpenACC model is very attractive as it standardizes programming. To the best of our knowledge, this is one of the very few papers that systematically evaluates the models. This paper is one of the first to compare OpenACC with other programming models.

978-0-7695-4956-9/13 $26.00 © 2013 IEEE
DOI 10.1109/SC.Companion.2012.373

The organization of this paper is as follows: Section II highlights some of the related work in this area, Section III provides comparative analysis of the three chosen high-level directive-based programming models. In Section IV, we will discuss details about how we have used these models to port four scientific applications to a heterogeneous platform that has NVIDIA's GPU cards attached. Section V concludes our paper.

We have explored two vendor compilers supporting the OpenACC model. To maintain confidentiality we will be referring to the 2 vendor compilers in this paper as OpenACC_Compiler_A and OpenACC_Compiler_B.

II. Related Work

Currently CUDA and OpenCL [4] are two major low-level GPU programming models available. CUDA can only be applied to NVIDIA GPUs, while OpenCL tries to solve the portability issue and thus it supports different accelerator architectures from different vendors. *hi*CUDA [13] is a high-level directive-based language that was designed to ease the GPU programming. *hi*CUDA provides a computation model and a data model. The computation model allows the programmer to specify a code region to be executed on GPU and to specify how the parallelism map to NVIDIA GPU architecture. The data model allows programmers to control when to allocate and free GPU memory and the data movement between CPU and GPU. It is also able to achieve all CUDA optimizations such as the use of constant and shared memory. In *hi*CUDA, every directive is similar to a corresponding CUDA runtime library function, moreover the programmer is expected to manage all data transfer, including the memory allocation and de-allocation, explicit data movement, loop scheduling and the use of cache. CUDA-lite [25] can automatically apply memory coalescing in global memory via annotations to optimize memory access pattern. This model is still not high-level enough since the programmer still needs to write the separate CUDA kernel functions. Another GPGPU compiler Guru [17] is also designed to convert the directive annotated C code into the code that contains appropriate OpenGL[20] and Cg APIs[19]. This language only supports one directive and two clauses to parallelize loops, and comes with too many restrictions. For example, only two dimensional array is supported and no pointer is allowed inside a parallelized loop. Mint [26] can translate traditional C source to optimized CUDA C using a small set of directives. The difference between this model and other models is that Mint focuses on stencil methods, so most of its optimizations are stencil based.

OpenMP-to-CUDA framework [16] is a research-based direction to extend OpenMP for GPUs, this framework can translate an OpenMP application into CUDA code automatically. It 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. This framework is extended to OpenMPC [15] with a new set of directives and environment variables. Ohshima et al. [23] discusses another framework called OMPCUDA. Here, the original code is divided into CPU portions and GPU portions, Omni compiler is used, GPU portions are further written to independent functions and thread launch functions. All shared variables are transferred to GPU but the dynamic variables (array and struct) and pointers seemed to be complex to implement. Simple block distribution is used for 'for loop' to distribute a block of iterations to each thread in the GPU. So far it has no data transfer optimization strategy and can only translate simple programs.

The case studies that we have considered for our experimental analysis are Feldkamp-Davis-Kress (FDK) algorithm [11], Heat conduction application and a Clustering algorithm. Let us discuss some of these case studies that have been ported to GPUs using CUDA and other approaches. Nesterets et al. [21] parallelized the most time consuming step back-projection operation in FDK algorithm with CUDA and achieved up to two orders of magnitude speedup in the back-projection itself and more than an order of magnitude speedup in the whole CT application. We extended this work by exploring directive-based accelerator programming models HMPP, PGI and OpenACC and compared their performance with the hand written CUDA code.

Ayar et al. [6] applied OpenMP and MPI programming models to solve 2D heat equation and CUDA model to solve 3D heat equation, and visualized the results obtained from heat equation for various data. Chen et al. [9] parallelized CLEVER clustering algorithm with both OpenMP and CUDA, and achieved linear scalability in multi-core system. We have used both 2D heat equation and CLEVER algorithm in our study and parallelized them using different directive-based accelerator models.

III. Overview of directive-based programming models

In this section we provide details about the three directive-based models that are being evaluated in this paper. HMPP is a directive-based programming model used to build parallel applications running on manycore systems. It is a source-to-source compiler that can translate directive-associated functions or code portions into CUDA or OpenCL kernels. In HMPP, the two most important concepts are "*codelet*" and "*callsite*" [2]. The "*codelet*" concept represents the function that will be offloaded to the accelerator, and "*callsite*" is the place to call the "*codelet*". It is the programmer's responsibility to annotate the code by identifying the codelets and inform the compiler about the codelets and where to call the same. In the steps of compilation, the annotated code is parsed by the HMPP preprocessor to extract the codelets and to translate the directives into runtime calls. The preprocessed code is then compiled and linked to HMPP runtime with a general-purpose host compiler. If the accelerator is not found or not available, the program execution can fall back to the original sequential version. HMPP also supports the "*region*" directive which only offloads part of a function into accelerator and the "*region*" is a merge of *codelet/callsite* directives. The main issue with programming accelerators is the data transfer between the accelerator and the host. HMPP offers many data transfer policies as part of the optimization strategies. The user can manually control the data transfer, i.e. transfer the data every time the codelet is called or transfer the data only during the first time when the codelet is called. It can also be decided by the compiler automatically.

HMPP also provides a set of directives to improve the performance by enhancing the code generation. In the codelet, the user can put the read only data into constant memory, preload the frequently used data into shared memory, or explicitly specify the grid size in NVIDIA architecture. If the loop is so complex that the compiler is not able to parse, the user can give some hints to the compiler that all iterations in the loop are independent. HMPP also support multi-GPUs programming by using "*parallel*" directive.

PGI accelerator programming model contains a set of directives, runtime library routines and environment variables [12]. The directives include data directives, compute directives and loop directives. The compute directive specifies a portion of the program to be offloaded to the accelerator. There is an implicit data region surrounding the compute region, which means data will be transferred from the host to the accelerator before the compute region and be transferred back from the accelerator to the host at the exit of compute region. Data directives allow the programmer to manually control, i.e. where to transfer the data other than the boundaries of compute region. The loop directives enable the programmer to control how to map loop parallelism in a fine-grained manner. The user can add these directives incrementally so that the original code structure is preserved. The compiler maps loop parallelism onto the hardware parallelism using the *planner* [28]. PGI optimizes the data transfer by "*data region*" directive and its clauses and be able to remove unnecessary data copies. Using the loop scheduling directive, the user can add the data in the highest level of the data cache by using "*cache*" clause and this helps in improving the data access speed.

OpenACC [3] is an emerging GPU-based programming model that is working towards establishing a standard in directive-based accelerator programming. The OpenACC model is based on the PGI model, hence the former inherits most of the concepts from the latter. However some of the differences are: Unlike PGI's single "*region*" compute directive, OpenACC offers two types of compute directives "*parallel*" and "*kernels*". The directive "*kernels*" is similar to PGI's "*region*" that surrounds the loops to execute on the accelerator device. With the "*parallel*" directive, however, if there is any loop inside the following code block and the user does not specify any loop scheduling technique, all the threads will execute the full loop. OpenACC supports three levels parallelism: gang, worker and vector, while PGI only defines two levels of parallelism: parallel and vector. Both OpenACC and PGI models allow the compute region to use the "*async*" clause to execute asynchronously with the host computation and the user can synchronize these asynchronous activities with the "*wait*" directive. They also have a similar set of runtime library routines, including getting the total number of accelerators available, getting & setting the device type and number, checking & synchronizing the asynchronous activities and starting up & shutting down the accelerator. Unlike PGI, OpenACC can allocate and free a part of accelerator memory using *acc_malloc()*and *acc_free()* functions.

The NVIDIA GPU architecture consists of several streaming multiprocessors (SMs). Each SM contains multiple scalar processors (SPs, also referred to as cores), control units, and memory which is shared among all SPs. An SP contains integer, floating point, logic, branching, and move and compare units. Each thread is executed by an SP. The code is executed in a group of threads which is called a warp. All threads in a warp execute the same instruction at the same time. The threads mapped to an SM is called a thread block. A possible mapping from OpenACC execution model to NVIDIA GPU hardware is as follows: gangs map to thread blocks, workers map to warps inside each block, and vector maps to all threads inside a warp. When a "*kernels*" directive appears, blocks of threads will be created and the whole workload will be distributed to these threads. Several gangs may execute on a single SM if the user specified gang number is greater than the available SMs in the hardware. A single gang does not span SMs and will stay in only one SM until finished. If any conditional branching occurs inside a warp, the warp serially executes each branch path where some threads are disabled for conditional operations.

IV. Porting Applications on GPUs: Experimental Analysis

In this section, we evaluate three directive-based programming models using four scientific applications. The experimental platform is a server machine that is a multicore system consisting of two NVIDIA C2075 GPUs. Configuration details are shown in Table I. We use the most recent versions for all the compilers being discussed in this paper. We use GCC 4.4.7 for all the sequential versions of the programs as well as for the HMPP host compiler. We use -O3 as the compilation flag for optimization purposes. As part of the evaluation process, we highlight several features of the programming models, that is best suited for the characteristics of an application. We compare the performances achieved by each of the models with that of the sequential and CUDA versions of the code. We consider wall-clock time as the evaluation measurement.

Table I: Specification of experiment machine

Item	Description
Architecture	Intel Xeon x86_64
CPU socket	2
Core(s) per socket	8
CPU frequency	2.27GHz
L1, L2 and L3 cache	32KB, 256KB and 8192KB
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

A. 2D Heat Conduction

The formula to represent 2D heat conduction is explained in[24] and is 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. At the beginning, there is a grid that has boundary points with initial temperature and the inner points that need to update their temperature. Then 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 128*128 to 4096*4096. Figure 1 contains the code for temperature updating kernel and and steps to call this kernel in the main program.

It is possible to parallelize the 2D heat conduction using the most basic directives from OpenACC, HMPP and PGI. For example, we use the OpenACC model and insert "*#pragma acc kernels loop independent*" before the nested loop inside the temperature updating kernel. The main point to note with this algorithm is performance. By profiling the basic implementation, we found that the data is transferred back and forth in every main iteration step. The cost of data transfer is so expensive that the parallelized code is even slower than the original native version. The challenging task is executing the pointer swapping operation. In iteration *i*, *temp1* is the input and *temp2* stores the output data. Before proceeding to iteration $i+1$, these two pointers are swapped so that *temp1* holds the output data in iteration *i* while *temp2* will wait to store the output data in iteration $i+1$. An intermediate pointer *temp* is needed to swap these two pointers. Since *temp* resides on the host while *temp1* and *temp2* reside on the accelerator, they cannot be swapped directly. The key is how to swap these pointers inside the accelerator internally so that unnecessary data transfer is removed.

While using the OpenACC model, we use the *deviceptr* data clause to specify *temp* as a device pointer, i.e, the pointer will always remain on the accelerator side. To avoid transferring the data during each step, *data* directive needs to be added so that the data is transferred only before and after the main loops. After all the iterations are completed we need to transfer data in *temp1* instead of *temp2* from GPU to CPU since the pointers of *temp1* and *temp2* have been swapped earlier. Inside the temperature updating kernel, the nested loop is collapsed because every iteration is independent. Using HMPP, we have considered copying the input and output data just once by setting the data transfer policy of *temp_in* and *temp_out* to *manual*. To ensure that the correct data is being accessed, we used HMPP's data mirroring directive so that we refer to arguments *temp1* and *temp2* with their host addresses. Data mirroring requires data mirrors to be declared and allocated before being used. HMPP could collapse the nested loop in the kernel by using "*gridify(j,i)*" code generation directive. Figure 1 and Figure 2 show the code snippet using both HMPP and OpenACC model.

Note that this application is sensitive to floating point operations. We found that the precision of the floating point values of the final output temperature on the GPU is different to the values on the CPU. This is due to Fused Multiply-Add (FMA) [27], here the computation rn(X * Y + Z) occurs in a single step and rounded once. Without FMA, rn(rn(X * Y) + Z) is composed of two steps and rounded twice. So their results would be slightly different. In 2D heat conduction, this kind of numerical difference will propagate with more iterations. To disable FMA, we used the options "–nvcc-options -fmad=false" in HMPP and "-ta=nofma" in PGI. When we used these compilation flags, the results of both HMPP and PGI appeared to be exactly the same as that of the CPU. Figure 3 shows performance results when different programming models are applied on the application.

Let us consider the baseline (sequential) speedup to be 1 as shown in the Figure 3. We see that for almost all the grid sizes, HMPP and PGI models perform as close as 80% to that of the CUDA version. The CUDA code has been considered from [24]. OpenACC_Compiler_A and OpenACC_Compiler_B perform approximately 80% and 70% respectively to that of the CUDA version. For the smallest grid size 256*256 considered, we see that neither the directive-based approaches nor the CUDA version was able to perform better than the sequential version. This is due to the fact that GPU is only suitable for massive

```
#pragma hmpp heat codelet, target=CUDA &
#pragma hmpp & , args[temp_in].io=in &
#pragma hmpp & , args[temp_out].io=inout &
#pragma hmpp & , args[temp_in,temp_out].mirror &
#pragma hmpp & , args[temp_in, temp_out].transfer=manual
void step_kernel(int ni, int nj, float tfac,
                 float *temp_in, float *temp_out) {

    // loop over all points in domain (except boundary)
    #pragma hmppcg gridify(j,i)
    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 hmpp heat allocate, data["temp1"], size={ni*nj}
 #pragma hmpp heat advancedload, data["temp1"]

 #pragma hmpp heat allocate, data["temp2"], size={ni*nj}
 #pragma hmpp heat advancedload, data["temp2"]

 // main iteration loop
 for (istep=0; istep < nstep; istep++) {
     #pragma hmpp heat callsite
     step_kernel(ni, nj, tfac, temp1, temp2);
     // swap the temp pointers
     temp = temp1;
     temp1 = temp2;
     temp2 = temp;
    }

 #pragma hmpp heat delegatedstore, data[temp1]
 #pragma hmpp heat release
```

Figure 1: HMPP Implementation of 2D Heat Conduction

computation purposes. For example, we see that for the grid size 4096*4096, almost all of the directive-based approaches and CUDA model seem to achieve more than 100% to that of the sequential version.

B. FDK Algorithm

Computed Tomography (CT) has been widely used in medical industry to produce tomographic images of specific areas of the body. It uses reconstruction technique that reconstructs an image of the original 3D-object from a large series of two dimensional X-ray images. As a set of rays pass through an object around a single axis of rotation, the produced projection data is captured by an array of detectors, from which a Filtered Back-Projection method based on the Fourier Slice Theorem is typically used to reconstruct the original object. Among various filtered back-projection algorithms, the FDK algorithm is mathematically

```
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 copyin(ni, nj, tfac)\
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 2: OpenACC Impementation of 2D Heat Conduction

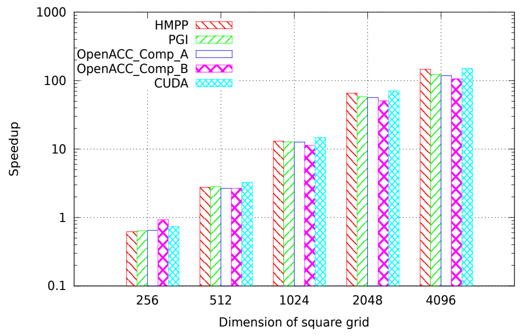

Figure 3: 2D Heat Conduction Speedup

straightforward and easy to implement. It is important that the acquired reconstruction effect is good, the goal of this work is to speed up the reconstruction using directive-based programming models.

Algorithm 1 shows the pseudo-code of the FDK algorithm, this is comprised of three main steps: Weighting - calculate the projected data; Filtering - filter the weighted projection by multiplying their Fourier transforms; Back-projection - back-project the filtered projection over the

3D reconstruction mesh. The reconstruction algorithm is computationally intensive and it has biquadratic complexity ($O(N^4)$), where N is the number of detector pixels in one dimension. The most time-consuming step of this algorithm is back-projection which takes more than 95% of the whole algorithm. So we will concentrate on parallelizing the back-projection step.

Algorithm 1: Pseudo-code of FDK algorithm

```
Initialization;
foreach 2D image in detected images do
    foreach pixel in image do
        Pre-weight and ramp-filter the projection;
    end
end
foreach 2D image in detected images do
    foreach voxel in 3D reconstruct volume do
        Calculate projected coordinate;
        Sum the contribution to the object from all
        tilted fan beams;
    end
end
```

We follow the approach from [14] for implementation purposes. The back-projection has four loops. The three outermost loops will loop over each dimension of the output 3D object, and the innermost loop will access each of 2D detected image slices. First the code is restructured so that the three outermost loops are tightly nested and then we can apply *collapse* clause from PGI and OpenACC and use *gridify* clause from HMPP. The innermost loop is sequentially executed by every thread. All detected images are transferred from CPU to GPU by using the *copyin* clause, and the output 3D object (actually many 2D image slices) are copied from GPU to CPU using the *copyout* clause. In HMPP, the input/output property of detected images is set as *in* and output object is set as *out*. To evaluate our implementations, we use the 3D Shepp-Logan head phantom data which has 300 detected images and the resolution of each image is 200*200. The algorithm produces 200*200*200 reconstructed cube. Note that this algorithm also has the same issue as 2D heat conduction i.e. the algorithm is sensitive to floating point operations, so we disabled the FMA using the compilation flags. This does not mean that the results using FMA are incorrect, but just that we explore different implementation technqiues while maintaining the same computation strategy, such that the results are consistent; we will also be able to make a fair comparison of results in this case. Figure 4 shows the speedup for different accelerator programming models compared to that of the sequential version. A point to note is that the interval between the points in the Y-axis is small, their performances are actually close to each other. The performance of all directive-based models (HMPP, PGI and OpenACC) are close to 90% of the CUDA code, that was written by us from scratch.

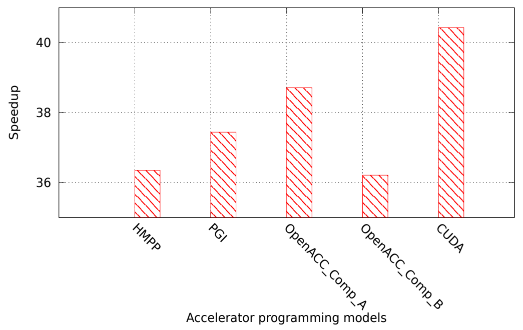

Figure 4: FDK Speedup with Different Models

C. CLEVER Clustering

In this section, we will parallelize a clustering algorithm called CLEVER (CLustEring using representatiVEs and Randomized hill climbing) [10]. CLEVER is a prototype-based clustering algorithm that seeks for clusters maximizing a plug-in fitness function. Prototype-based clustering algorithms construct clusters by seeking an 'optimal' set of representatives-one for each cluster; clusters are then created by assigning objects in the dataset to the closet cluster representatives. Like K-means [18], it forms clusters by assigning the objects to a cluster with the closest representative. CLEVER uses a randomized hill climbing to seek a good clustering, i.e. it samples *p* solutions in the neighborhood of the current solution and continues this process until no better solutions can be found. Algorithm 2 shows the pseudo-code of CLEVER program code. It starts with randomly selecting *k'* representatives from the dataset *O* where *k'* is provided by the user. CLEVER samples *p* solutions in the neighborhood of the current solution and chooses the solution s with the maximum fitness value for *q(s)* as the new current solution provided there is an improvement in the fitness value. New neighboring solutions of the current solution are created by three operators: "Insert" which inserts a new representative into current solution; "Delete" which deletes an existing representative from current solution, and "Replace" which replaces an existing representative with a non-representative. Each operator is selected at a certain probability and the representatives to be manipulated are chosen at random. To prevent premature convergence, CLEVER will resample $p * q$ more solutions in the neighborhood before terminating, where q is the resampling rate. The description of CLEVER parameters are as follows [8]:

1) *k'*: initial number of clusters
2) *neighborhood-size*: maximum numbers of operators applied to generate a solution in the neighborhood
3) *p*: sampling rate, number of samples that is randomly selected from the neighborhood

4) q: resampling rate. If the algorithm fails to improve fitness with p and then $2*p$ solutions, then sampling size in the neighborhood would be increased by factor $q-2$
5) $imax$: maximum number of iterations in the algorithm

Algorithm 2: Pseudo-code of CLEVER algorithm

```
Input: Dataset O, k', neighborhood-size, p, q,imax
Output: Clustering X, fitness q(X), rewards for clusters
        in X
Current solution ← randomly selecting k'
representatives from O ;
while iterations ≤ imax do
    Create neighbors of the current solution randomly
    using the given neighborhood definition, and
    calculate their respective fitness;
    if The best neighbor improved fitness then
        Current solution ← best neighbor;
    else
        Neighborhood of current solution is re-sampled
        by generating more neighbors;
        if re-sampling leads to better solution then
            Current solution ← best solution found by
            re-sampling;
        else
            Terminate returning the current solution;
        end
    end
end
```

We profile the CLEVER algorithm using GNU profiler before parallelizing the same. Statistical information gathered shows that the most time-consuming portion of the algorithm is the function that assigns objects to the closest representative which computes and compares a lot of distances. The original code is written in C++, this needed to be converted to C so that the algorithm could be supported by PGI, HMPP and OpenACC model. (A point to note is that HMPP recognizes C++ code to an extent).

Since the data structure of the dataset is user-defined and the pointer operation is quite complicated, the accelerator region cannot be parsed by the compiler. Hence, the code is restructured so that it is relatively easier to be parsed by the compiler. The directives also give hints to the compiler such that all the iterations in the loop are independent. Since the whole dataset is read-only, it is transferred to accelerator before the kernel is called. We can achieve this using both PGI and OpenACC model by using *copyin()* clause of data construct. We use the *region* directive of HMPP model and set the data transfer policy of the dataset as *atfirstcall* so that it is copied from the host to the accelerator only once. This will enable the dataset to stay in the global memory of accelerator even if the kernel is called many times.

Table II: L10Ovals Dataset Characteristics

Item	Description
Data size	335,900 objects
Attributes	<x, y, class label>
Distance Function	Euclidean Distance
Plug-in Fitness Function	Purity: Percentage of objects belonging to the majority class of the cluster

Table III: Earthquake Dataset Characteristics

Item	Description
Data size	330,561 objects
Attributes	<latitude, longitude, depth >
Distance Function	Euclidean Distance
Plug-in Fitness Function	High Variance: Measures how far the objects in in the cluster are spread out with respect to earthquake depth

We evaluated the three directive-based models on two datasets called L10Ovals(Large 10Ovals) and Earthquake. The characteristics of these two datasets are shown in Table II and Table III, respectively. The L10Ovals dataset has natural clusters representing L10Ovals containing 335,900 objects.

Earthquake dataset contains 330,561 earthquakes which are characterised by latitude, longitude and depth of the earthquake. The goal is to find clusters where the variance of the earthquake depth is high; that is where shallow earthquakes are co-located with deep earthquakes.

Figure 5 shows the speedup of how the four different models react to these two datasets. OpenACC_Compiler_A required a very long time to execute this algorithm, hence we do not include the speedup in the graph.The reason may be that the model is yet to provide an effective support to deal with pointer operations.

For L10Ovals dataset HMPP showed a speedup of 4.63x, which is very close to that of the CUDA version, 5.04x. We have considered the CUDA version of the algorithm from [8]. We noticed that among the OpenACC models, OpenACC_Compiler_B showed a speedup of 1.58x, performing poorer to CUDA. For Earthquake dataset, the speedup achieved by HMPP, PGI, OpenACC_Compiler_B were 29.99x, 29.65x and 29.32x respectively, these are almost the same as that of CUDA, which showed a speedup of 30.33x. Although L10Ovals and Earthquake have almost the same number of objects in their datasets we still see a significant difference in performance. This is primarily due to the characteristics of each of these datasets. L10Ovals has well defined and separated clusters therefore converges more quickly. The L10Ovals clustering task takes 21 iterations whereas the earthquake dataset clustering takes 216 iterations. Moreover the number of clusters searched in the earthquake clustering experiment is much higher than the number of clusters in L10Ovals experiment making it more time consuming to assign objects to clusters.

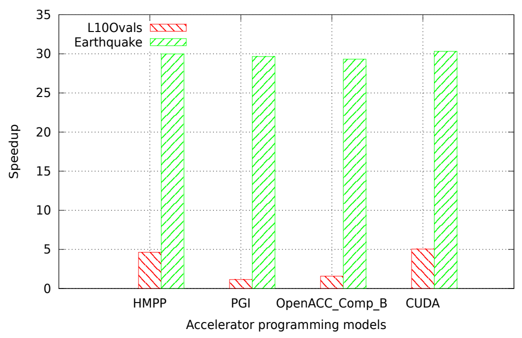

Figure 5: CLEVER Speedup with Different Models

Table IV: Time(in sec) Consumed by Serial, CUDA, HMPP, PGI and OpenACC Versions of the Code, Only for Most Time-consuming Dataset

Applications	Serial	CUDA	HMPP	PGI	OpenACC	
					A	B
2D Heat	8922.81	59.13	60.78	72.74	75.65	84.76
FDK	363.50	8.99	10.40	9.71	9.39	10.04
CLEVER	116.15	23.04	25.08	101.51	-	73.31

Table IV shows the execution time consumed in seconds by the different directive-based models for three case studies. Results for the application 2D Heat Conduction shows that the time consumed by directive-based models were significantly lower than the time consumed by the serial version of the code. However the execution time for the CUDA code still appears to be the best. We see that HMPP model performs better than that of PGI and also that of OpenACC models. It could be due to slightly better optimization strategies offered by HMPP compiler implementations. Among the two OpenACC models, OpenACC_Compiler_A seems to perform better than OpenACC_Compiler_B. With respect to FDK algorithm, we notice that almost all the models perform similar to each other and infact close to that of the CUDA code. CLEVER application shows that HMPP model performs the best compared to all other models. As mentioned earlier, we suspect that PGI compiler cannot handle pointer operations very efficiently yet. OpenACC_Compiler_A model may also have issues with the implementation of pointer operations. Hence the execution time seems to be long.

To summarize, directive-based models have indeed shown promising results compared to that of the CUDA version. In-depth research analysis of these models would lead to better performance. An important point to note is that the OpenACC model is still being constructed and the technical details may require fine tuning before we could actually make a deeper comparative analysis.

V. Conclusion and Future work

This paper evaluates some of the prominent directive-based GPU programming models for four applications with different characteristics. We compare each of these models and tabulate the performances achieved by these models. We see that the performance is highly dependent on the application characteristics. The high-level models also provide a high-level abstraction by hiding most of the low-level complexities of the GPU platform. This makes programming easier leading to programmer productivity. We noticed that all the directive-based models performed much better than that of the serial versions of the applications being evaluated. We also observed that these models demonstrated performance results close enough to that of the CUDA version of the applications. However, we had to write almost different versions of the code before we could use a particular programming model, especially while using HMPP and PGI. OpenACC solved this portability issue by providing a single standard to be used to program GPUs. As part of the future work, we would like to perform extensive research and propose solutions to enable OpenACC provide support for multiple GPUs. We will be also exploring suitable solutions to overcome the several limitations of the directive-based models that we came across while evaluating the different case studies.

Acknowledgment

We would like to thank PGI, CAPS and OpenACC for their unparallel support to help us understand their software infrastructure. We would also like to thank 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] OpenACC Standard Home. http://www.openacc-standard.org.

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

[5] The OpenMP API Specification for Parallel Programming. http://openmp.org/wp/.

[6] M. Zahid Ayar. Parallel Computations for Simulating Heat Conduction. world-comp.org/p2011/CSC8087.pdf.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[8] P. Charoenrattanaruk and C.F. Eick. Design and Evaluation of a High Performance Computing Framework for the CLEVER Clustering Algorithm. Master's thesis, University of Houston, December 2011.

[9] C.S. Chen, N. Shaikh, P. Charoenrattanaruk, C.F. Eick, R. Nouhad, and E. Gabriel. Design and Evaluation of a Parallel Execution Framework for the CLEVER Clustering Algorithm. pages 73–80, 2012.

[10] C.F. Eick, R. Parmar, W. Ding, T.F. Stepinski, and J. Nicot. Finding Regional Co-location Patterns for Sets of Continuous Variables in Spatial Datasets. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, GIS '08, pages 30:1–30:10, New York, NY, USA, 2008. ACM.

[11] L.A. Feldkamp, L.C. Davis, and J.W. Kress. Practical cone-beam algorithm. *JOSA A*, 1(6):612–619, 1984.

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

[13] 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.

[14] A. Kak and M. Slaney. *Principles of Computerized Tomographic Imaging*. Society for Industrial and Applied Mathematics, 2001.

[15] 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.

[16] S. Lee, S.J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[17] Y. Lin and P. Chen. Compiler Support for General-purpose Computation on GPUs. *J. Supercomput.*, 50(1):78–97, October 2009.

[18] J. MacQueen et al. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, page 14. California, USA, 1967.

[19] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM, 2003.

[20] J. Neider, T. Davis, and M. Woo. *OpenGL. Programming guide*. Addison-Wesley, 1997.

[21] Y.I. Nesterets and TE Gureyev. High-performance Tomographic Reconstruction Using Graphics Processing Units. In *Proc. 18th World IMACS/MODSIM Congress Modeling and Simulation (MODSIM08)*, 2009.

[22] B. Nichols, D. Buttlar, and J.P. Farrell. *Pthreads programming*. O'Reilly Media, Incorporated, 1996.

[23] S. Ohshima, S. Hirasawa, and H. Honda. OMPCUDA: OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler. *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, pages 161–173, 2010.

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

[25] 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.

[26] D. Unat, X. Cai, and S.B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[27] N. Whitehead and A. Fit-Florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. *rn (A+ B)*, 21:1–1874919424, 2011.

[28] M. Wolfe. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.