Experiences with High-Level Programming Directives for Porting Applications to GPUs

Oscar Hernandez, Wei Ding, Barbara Chapman,
Christos Kartsaklis, Ramanan Sankaran, and Richard Graham

Computer Science and Mathematics Division,
National Center for Oak Ridge National Laboratory**
Dept. of Computer Science, University of Houston
{oscar,kartsaklisc,sankaranr,rlgraham}@ornl.gov,
{wding3,chapman}@cs.uh.edu

Abstract. HPC systems now exploit GPUs within their compute nodes to accelerate program performance. As a result, high-end application development has become extremely complex at the node level. In addition to restructuring the node code to exploit the cores and specialized devices, the programmer may need to choose a programming model such as OpenMP or CPU threads in conjunction with an accelerator programming model to share and manage the different node resources. This comes at a time when programmer productivity and the ability to produce portable code has been recognized as a major concern. In order to offset the high development cost of creating CUDA or OpenCL kernels, directives have been proposed for programming accelerator devices, but their implications are not well known. In this paper, we evaluate the state of the art accelerator directives to program several applications kernels, explore transformations to achieve good performance, and examine the expressivity and performance penalty of using high-level directives versus CUDA. We also compare our results to OpenMP implementations to understand the benefits of running the kernels in the accelerator versus CPU cores.

1 Introduction

Computational scientists performing research in a broad range of domains are demanding ever more powerful computing systems as they strive to solve some of the most pressing problems of today and prepare for those of tomorrow. To meet their needs, systems that provide significantly higher levels of peak performance than any currently installed platforms are already being procured. Much more

** This work was funded by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. This research used resources of the Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UTBattelle, LLC.

powerful computers are expected to be built and delivered during the coming decade. These future exascale systems may feature a variety of architectural innovations that offer new ways to address the inherent challenges of scale, power, reliability, cost, and packaging.

To attain the desired levels of computational performance while meeting operational constraints, hardware designers have begun to exploit recent developments in mainstream computer architectures. Following hard upon the heels of the industry-wide move from single to multi-core systems – designed to increase the overall computational performance without unduly increasing the amount of energy required to operate it – dominant mainstream computer architectures are now undergoing a second transition from *homogeneous* to *heterogeneous* designs that incorporate components designed for high-throughput, providing massive levels of parallelism. Emerging large-scale platforms will be heavily impacted by these technology shifts. For instance, the Tianhe-1A relies on graphic processing units (GPUs) to achieve 2.5 Petaflops of peak Performance. Nebulae and TSUBAME both use GPU technology and are in the current Top 10 supercomputers list. In addition, Oak Ridge National Laboratory (ORNL) has already announced that its next leadership class machine will be based on a heterogeneous multi-core system with GPUs.

Whereas accelerators on today's multi-core nodes are typically GPUs – massively parallel processors that can be programmed to perform a range of computations including, but not limited to, their original graphics domain – future systems are expected to have much higher core counts and density [1], and potentially will integrate a variety of special-purpose devices, or system on chips, that can provide the highest levels of performance on suitable code regions. Significant challenges face application developers as they learn to efficiently exploit recently installed architectures. The effectiveness of MPI-based applications has been reduced by the effects of multiple levels of architectural parallelism, limited amounts of memory per core and complex sharing of resources such as interconnects, caches and shared memories. The transition to heterogeneous node hardware will significantly exacerbate their difficulties, as they will need to explicitly detect and adapt code to run on the GPUs, potentially using yet another programming model. They will be required to deal with distinct and complex memory systems and the high cost of data transfer between them. As a result, GPU directive-based programming APIs are starting to emerge in an attempt to facilitate the porting process.

In this paper, we aim to identify the challenges involved in exploiting GPUs for non-graphics applications; to evaluate the programming effort and performance that can be obtained via the use of two existing, directive-based programming models, the HMPP and PGI accelerator directives, and compare them with CUDA and OpenMP. The accelerator directives used are a relatively new technology and we expect that the underlying implementation is capable of improvement. Nevertheless, we observe that even with the use of directives, a good deal of program reorganization may be required. The amount of effort depends not only upon the application program in question but also on the desired level of performance improvement.

The paper is organized as follows. The next section below discusses the state of the art with respect to heterogeneous computing. We compare two high-level directive-based programming models in Section 3. This is followed by the description of our efforts to adapt two existing programs, with different program structures and requirements, to a platform that includes GPUs in Section 4. In each case, we have used CUDA and two directive-based programming models to port the code to a multicore node that employs a GPU provided by NVIDIA. We then state our conclusions of our work in Section 5.

2 Programming Models for Heterogeneous Systems

Several vendors have provided programming interfaces for accelerators. Most have adapted the C programming language to fit the strict requirements of applications on their platform. GPUs were originally programmed using OpenGL. Domain-specific languages for graphics programming like GLSL (OpenGL Shading Language), HLSL (high level shader language), and Cg (C for graphics) from NVIDIA are also available. With their growing usefulness for compute-intensive functions in general-purpose applications, a number of programming interfaces (mostly based on C) have been provided to facilitate the development of application kernels for them. Rather than being fully fledged languages, most of them are based upon C. These include StreamIt [6], Sh [11], Brook [7], CUDA [13] and OpenCL. Thrust [14] is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). Compared with those programming languages, CUDA in particular has become popular for general-purpose programming on NVIDIA GPUs. The OpenCL [3] specification is the first standard programming API for accelerators released by Khronos [2].

Based heavily on CUDA, it is poised to become the first standard

Moreover, a variety of high level programming directives for accelerators are available or are undergoing development. CAPS HMPP [9], PGI accelerator directives [4] and HiCUDA [10] target CUDA and OpenCL [3]. RapidMind [5] defines C++ extensions that allow its users to describe how data in a C++ application should be mapped between GPUs, Cell processors and cores. However their approach requires redefinition of data types and the creation of kernels with a special syntax language. It is important to note that while these approaches provide portability at the language / directive level, the program optimizations and porting strategies required to apply them depend heavily on the target architecture and the application input set.

3 A Comparison of Directive-based Programming Approaches

CUDA (and OpenCL) require the application developer to carefully study all the salient details of the target architecture. The process of code adaptation and tuning may be lengthy, involving significant reorganization of code and data, and is moreover error-prone. Porting the resulting code to another GPU (e.g. a

successor model) may require non-trivial modification. High-level programming models have the potential to simplify the program creation and maintenance effort of the code. There have been few studies [12] that compare different vendors' GPU directives implementations. However, unlike ours, they do not focus on and the optimization process to achieve good performance, and the benefits of using the directives versus native CUDA or OpenMP.

In this section we provide an overview and compare two of the most popular directive based programming languages for GPUs: PGI Accelerator Directives and the HMPP Workbench.

3.1 Overview of The HMPP and PGI Accelerator Directives

HMPP is a directive-based programming interface for hybrid multicore parallel programming that aims to free the application developer from the need to code in a hardware-dependent manner. It is implemented by a source-to-source compiler designed to extract maximal data parallelism from C and FORTRAN kernels and translate them into NVIDIA CUDA or OpenCL. The main concepts of HMPP are the codelet and the callsite. A function that can be executed remotely on an accelerator is identified by the codelet directive; the callsite is the place the codelet (kernel) function call is launched. HMPP has both synchronous and asynchronous modes for the codelet remote procedure calls (RPCs). The asynchronous mode enables the overlapping of data transfers between the host and accelerators with other work. The programmer specifies targets for the execution of codelets. If the desired accelerator is present and available, it will run there. Otherwise the native host version is run.

PGI's Accelerator directives may be incrementally inserted into a code to designate a portion of C or Fortran code to be run on CUDA-enabled NVIDIA GPUs. They enable the application developer to specify regions for potential acceleration, to manage the necessary data transfers between the host and accelerator, and to initialize and shut down the accelerator. They further provide guidance to the implementation to help it perform data scoping, mapping of loops and transformations for performance. The directives assume that it is the host that handles the memory allocation on the device, initiates data transfers, sends the kernel to the device, waits for completion and transfers the results back from the device. The host is also responsible for queuing kernels for execution on the device.

The PGI directives include the kernel region declaration *#pragma acc* with *copyin*, *local* and *copyout* clauses to specify the input data, local data and output data of the kernel. PGI also supports the autoscoping of these data automatically. Directives *#pragma acc for parallel(M)* and *#pragma acc for vector(N)* are used to help the compiler identify parallel loops and how they should be mapped to the GPU.The compiler also attempts to associate a loop nest's iterations to grid and blocksizes that map to the GPU. The grid sizes will depend on the amount of work launched in the kernel while the threadblock size remains constant. In addition, PGI provides other clauses to optimize the kernels such as *cache* to load data to shared memory, loop *unroll* and *host* to run a loop in the CPU. PGI has

been developing new directives such as the *#pragma acc region* and *#pragma acc* declaration directive to scope variables that should be shared among kernels and reside in the GPU or CPU or both.

HMPP uses the concept of groups, *#pragma hmpp group <groupid >, target=CUDA*, to specify codelets that share the same data set and will run in the same accelerator. HMPP provides the codelet *#pragma hmpp <groupid ><codeletid >codelet* and callsite directive *#pragma hmpp <groupid ><codeletid >* to specify codelets and where to invoke them. In addition these directive have clauses to specify the input, and output data and their sizes in the format format *args[A1]=size* and *args[A1].io=in, args[An].io=inout*. HMPP provides the asynchronous advanced load and delegate store directive to control when to move data to and from the accelerator. The directive *#pragma hmppcg parallel* indicates that the following loop is parallel and can be mapped to the GPU, while *#pragma hmppcg noParallel* indicates that a loop must not be parallelized. HMPP allows the user to explicitly define the threadblock size of a loop nest by using the *#pragma hmppcg grid blocksize NxN* directive. The *#pragma hmpp <groupid >resident* specifies data that should be allocated in the GPU and that may be shared among codelets.

4 Adapting Programs for GPUs: Two Case Studies

In this section, we present our experimental results that consists of adapting two applications kernels to run on GPU based platform. For both kernels, we use the PGI and HMPP accelerator directive-based programming to accelerate the kernels and we describe the transformation we used to get good performance when comparing it against CUDA and OpenMP.

Our experiments were run on an NVIDIA Tesla C2070 GPU with 448 cores in 14 Streaming Multiprocessors with frequency of 1.15 GHz. The GPU has 6GB DDR5 global memory shared by all threads. The local memory is 64K in size, and can be split 16K/48K or 48K/16K between L1 cache and shared memory. Shared memory for each Streaming Multiprocessor is accessible only within a thread block. The Tesla C2070 is also equipped with an L2 cache that covers GPU global memory.

4.1 S3D Thermodynamics Kernel

S3D is a parallel combustion flow solver for the direct numerical simulation of turbulent combustion. S3D [8] solves fully compressible Navier-Stokes, total energy, species and mass conservation equations coupled with detailed chemistry. The governing equations are supplemented with 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 are implemented as kernels or community-standard libraries that are amenable to acceleration through GPU computing. For this work, we chose the thermodynamics kernel that evaluates the mixture-specific heat, enthalpy and Gibbs functions as a temperature polynomial.The coefficients of the

thermodynamic polynomials and their relevant temperature ranges are obtained from thermodynamic databases following the conventions used in the NASA Chemical Equilibrium code. The thermodynamic kernel with small variations is applicable across a wide range of reacting flow applications.

```
do i = 1, np
  enth(i) = 0.0
  do m = 1, nslvs
  if(temp(i)<midtemp(m)) then
    enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coefflow(6, m)+temp(i)*(&
                  . . .
            coefflow(5, m)*rp05)))) )
  else
    enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coeffhig(6, m)+temp(i)*(&
                  . . .
            coeffhig(5, m)*rp05)))) )
    end if
  end do
end do
```

(a)S3D Thermodynamics Serial

```
!$OMP parallel do private(i, m, enth)
do i = 1, np
  enth(i) = 0.0
  do m = 1, nslvs
    if(temp(i)<midtemp(m))then
     enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coefflow(6,m)+temp(i)* (&
                  .  .  .
            coefflow(5,m)*rp05))))))
    else
      enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coeffhig(6,m)+temp(i)*(&
                 .  .  .
            coeffhig(4,m)*rp04+temp(i)*(&
            coeffhig(5,m)*rp05))))))
    end if
  end do
end do
!$OMP end parallel do
```

(b)S3D Thermodynamics OpenMP

```
!$OMP parallel private(i,m,flag_hig,flag_low)
do m = 1, nslvs
  !$OMP do
  do i = 1, np
    if(temp(i)<midtemp(m)) then
         flag_low(i, m)=1
         flag_hig(i, m)=0
    else
         flag_low(i, m)=0
         flag_hig(i, m)=1
    endif
  enddo
!$OMP end do nowait
enddo

!$OMP do
do i = 1, np
enth(i) = 0.0
  do m = 1, nslvs
   enth(i)=flag_low(i,m)*(enth(i)+yspec(i,m)*&
           Rsp(m)*(coefflow(6,m)+ temp(i)*(&
                  . . .
           coefflow(5, m)*rp05)))) ))+&
           flag_hig(i,m)*(enth(i)+yspec(i,m)*&
           Rsp(m)*(coeffhig(6,m)+temp(i)*(&
                  . . .
           coeffhig(5, m)*rp05)))) ))
  end do
end do
!$OMP end do nowait
!$OMP end parallel
```

(c)Optimized S3D Thermodynamics with OpenMP

Fig. 1: S3D Thermodynamics Kernel Code snippet

Figure 1(a) shows the most time consuming portion of the serial kernel, where a double nested loop contains an *if* statement. The serial version takes about 22 seconds to execute in a CPU core. We decided to parallelize the outerloop with OpenMP as shown in Figure 1(b). We noticed that the inner loop was not being vectorized because of the *if* conditional. To further optimize the code, we hoisted the *if* conditional by precomputing the branch values in a separate loop that was also parallelized with OpenMP. As a result, we merge the if and else

```
!$acc data region copyin(temp,...),&
   copyout(enth)
do j = 1, MR
!$acc region
!$acc do parallel(np)
  do i = 1, np
    enth(i) = 0.0
    do m = 1, nslvs
      if(temp(i)<midtemp(m)) then
        enth(i)=enth(i)+yspec(i,m)*&
          Rsp(m)*(&
          ...
        coefflow(5, m)*rp05)))))
      else
        enth(i)=enth(i)+yspec(i,m)*&
          Rsp(m)*(&
          ...
        coeffhig(5, m)*rp05))))))
      end if
    end do
  end do
!$acc end region
!$acc region
!$acc do parallel(np)
  do i = 1, np
    cp(i) = 0.0
    do m = 1, nslvs
      ...
    end do
  end do
!$acc end region
end do
!$acc end data region
```

(a) PGI

```
!$hmpp <cudagroup> group, target=CUDA
!$hmpp <cudagroup> resident, args[Rsp].io=in
real,parameter::Rsp(1:nstts)=Ru/molwgt(1:nstts)
!$hmpp <cudagroup> resident, args[midtemp].io=in
real,parameter::midtemp(68)=(/ ... /)
!$hmpp <cudagroup> resident,args[coeffhig].io=in
real,parameter::coeffhig(7,68)=reshape(/.../)

subroutine calc_mixenth(np, ... ,cp)
implicit none
. . .
!$hmpp <cudagroup> allocate
!$hmpp <cudagroup> s3d_mixenth advancedload,&
       args[::Rsp; ...; ::coeffhig]
!$hmpp <cudagroup> s3d_mixenth callsite
call hmpp_kernel1(np, ... , coeffhig)
!$hmpp <cudagroup> s3d_mixcp callsite, &
       arg[::Rsp; ...].advancedload=true
call hmpp_kernel2(np, temp, ... , coeffhig)
!$hmpp <cudagroup> release
end subroutine calc_mixenth

!$hmpp <cudagroup> s3d_mixenth codelet, &
    args[np;...;yspec].io=in,args[enth].io=out
subroutine hmpp_kernel1(np,temp,...,coeffhig)
    ...
end subroutine hmpp_kernel1
!$hmpp <cudagroup> s3d_mixcp codelet, &
      args[np;...;yspec].io=in,args[cp].io=out
subroutine hmpp_kernel2(np,...,coeffhig)
    ...
end subroutine hmpp_kernel2
```

(b) HMPP

Fig. 2: S3D Thermodynamics Kernel Code snippet

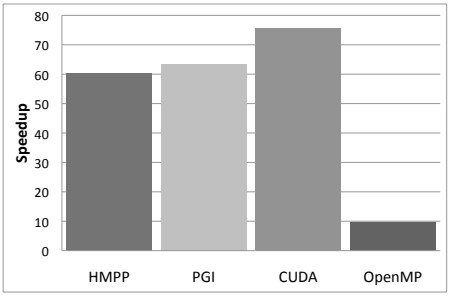
computations into single statements that were masked with the precomputed branch result. Figure 1(c) shows the transformation applied. By doing so, we were able to parallelize and vectorize the computational loop which yielded a good speed up of 3.8x when running the code on four cores. When running the code on twelve cores, the original OpenMP version in Figure 1(b) yielded the best performance of 9.5x because it does not have any shared memory contention on the masked branch variable.

Our first attempt to accelerate the code with PGI and HMPP directives, by inserting a *!$ acc region* directive and creating a HMPP codelet for the main computational loopnest yielded very little performance for PGI and HMPP directives (2x speedup for PGI and 1.2 speedup for HMPP). By using the CUDA Profiler from NVIDIA, we observed that most of the time was spent in the accelerated kernels on data transfer between the CPU and GPU. Since the kernel contains read-only arrays, we optimized the accelerated kernels by allocating and initializing the read only variables inside the GPU. To do so, we had to inline the main computational kernel loop (loop i) inside the procedure that was invoking it within its loop j. The PGI version of this transformation is shown in Figure 2(a), which uses a data region to define the data that resides in the

GPU. For HMPP, we used the *group* and *resident* directive to allocate data in the GPU and share data among the codelets of the same group. Figure 2(b), the HMPP implementation, where codelets *s3d_mixenth* and *s3d_mixcp* belong to the same group named cudagroup. Arrays *Rsp, midtemp, coeffhig* and *coefflow* are declared as resident variables, which makes them accessible to the two codelets defined in the HMPP group. In order to optimize the data transfers, we used the *advancedload* directive to initialize the read only data once before the first codelet *calc_mixenth* is executed. We also used the *advancedload* clause of the HMPP *callsite* directive to notify HMPP that the read only data is available in the GPU for the second codelet.

S3D Thermodynamics Timings (Seconds)	
SERIAL	21.926
HMPP	0.363
HMPP Kernel	0.3192948
HMPP Data Transfer	0.042834
PGI	0.346305
PGI Kernel	0.320225
PGI Data Transfer	0.02608
CUDA	0.29
CUDA Kernel	0.269265
CUDA Data Transfer	0.019952
OpenMP 12 Threads (best)	2.274

(a) S3D Thermodynamics Timing Table



(b) S3D Thermodynamics Speedup

Fig. 3: S3D Thermodynamics Kernel Experiment

When comparing the results from different parallelization and acceleration strategies, we found that the HMPP and PGI implementations produced 60x and 63x speedup, respectively. The native CUDA implementation produced a speedup of 76 times that amount, while the OpenMP version using twelve threads produced a speedup of 10, as shown in Figure 3(b). The timings of our experiments are shown in Figure 3(a). Our results show that by managing the data correctly we were able to produce good speedups with the PGI and HMPP accelerator directives, within 80% of the native CUDA performance.

4.2 HOMME/SE Application

The High-Order Multi-scale Modeling Environment application, HOMME, is one of the highly promising frameworks for integrating the atmospheric primitive equations in spherical geometry. HOMME applies a spectral element method to conserve both mass and energy using an isotropic hyper-viscosity term. To discretize horizontal dimension, it uses a cubed-sphere grid and in the radial direction a vertical dimension. The HOMME application consists of several hun-

dred Fortran 90 subroutines where the computations are spread evenly across them and whose relevance depends on the input problem.

```
  ...
do ie=nets, nete
  do q=1,qsize
    do k=1,nlev
      gradQ5d(:,:,k,q,1)=
      elem(ie)%state%v(:,:,1,k,n0)*&
      elem(ie)%state%Qdp(:,:,k,q,n0)

      gradQ5d(:,:,k,q,2)=
      elem(ie)%state%v(:,:,2,k,n0)* &
      elem(ie)%state%Qdp(:,:,k,q,n0)
    end do
  end do

  divdp4d(:,:,:,:) =
         divergence_sphere5d( &
         gradQ5d(:,:,:,:,:), &
         deriv, elem(ie))
    ...
end do
```

(a) Serial code

```
 ...
!$omp parallel private(ie,j,i,k,q,m,l,&
!$omp& gradQ5d,divdp4d,deriv)
 ...
!$omp do
  do ie=nets, nete
   do q=1,qsize
     do k=1,nlev
       gradQ5d(:,:,k,q,1)=
       elem(ie)%state%v(:,:,1,k,n0)*&
        elem(ie)%state%Qdp(:,:,k,q,n0)
        gradQ5d(:,:,k,q,2)=
        elem(ie)%state%v(:,:,2,k,n0)* &
        elem(ie)%state%Qdp(:,:,k,q,n0)
      end do
    end do

    divdp4d(:,:,:,:) =
           divergence_sphere5d( &
           gradQ5d(:,:,:,:,:), &
           deriv, elem(ie) )
   ...
  end do
!$omp enddo nowait
!$omp end parallel region
```

(b) OpenMP code

Fig. 4: The original and OpenMP divergence_sphere code version

For each of the spherical elements in the grid, HOMME maintains a global data structure that stores the state of the element, including velocity, temperature, pressure, divergence and geo-potential. Figure 4(a) shows a code fragment of the subroutine *compute_and_apply_rhs* which is one of the routines that computes the divergence for each of the cubed elements. The *ie* loop iterates over the spherical elements, the q loop over the advected physics, the k loop iterates over the vertical radial grid points and the j and i loops iterate over the horizontal plane grid points.

In HOMME, coarse grain parallelism is implemented via MPI by distributing the spherical elements across nodes, whereby one or more elements can be assigned to an MPI process (see *ie* loop). In our case we assumed that each node will be assigned twelve elements to provide enough work for all the cores for in-node optimization or acceleration (i.e one element per core). The in-node problem size used was: $ie = 12$, $qsize\_d = 101$, $nlev = 26$ and $nv,np = 4$. We optimized several versions of the kernel for OpenMP, PGI Accelerator directives, and HMPP. We then compared them against the original serial version and the CUDA implementation tuned by NVIDIA and ORNL.

For the OpenMP version, see Figure 4(b). We parallelized the *ie* loop with *OpenMP parallel do* to assign spherical elements to OpenMP threads and take

```
!$acc region
!$acc do parallel(nete)
  do ie=nets, nete
!$acc do parallel(qsize)
   do q=1,qsize
!$acc do vector(32)
   do k=1,nlev
!$acc do vector(nv)
    do j=1,nv
!$acc do vector(nv) private(dudx00,dvdy00i)
    do l=1,nv
      dudx00=0.0d0
      dvdy00i=0.0d0
     do i=1,nv
      dudx00 = dudx00 + Dvv(i,l  ) * &
      (metdet(i,j,ie)*(Dinv(1,1,i,j,ie)* &
      gradQ5da(i,j,k,q,1,ie) + &
      Dinv(1,2,i,j,ie)*gradQ5da(i,j,k,q,2,ie)))

      dvdy00i = dvdy00i + Dvv(i,j  ) * &
      (metdet(l,i,ie)*(Dinv(2,1,l,i,ie)* &
      gradQ5da(l,i,k,q,1,ie) + &
      Dinv(2,2,l,i,ie)*gradQ5da(l,i,k,q,2,ie)))
     end do
     divdp4da(l,j,k,q,ie)= &
              rmetdetp(l,j,ie) * &
             (rdx(ie))*dudx00+(rdy(ie))*dvdy00i
     end do
    end do
   end do
  end do
 enddo
!$acc end region
```

(a) PGI Accelerator Directives

```
!$hmpp <elements_group> divergence codelet
subroutine hmpp_divergence_sphere(rmetdetp,rdx,
 rdy,Dvv,metdet,Dinv,gradQ5da,divdp4dhmpp)
....
!$hmppcg parallel
!$hmppcg grid blocksize 32x4
  do k1 = 1, (nv*nv*nlev*qsize*(nete-nets+1))
   k2 = k1
   l = mod(k2, nv) + 1
   k2 = k2 / nv
   j =  mod(k2, nv) + 1
   k2 = k2 / nv
   k =  mod(k2, nlev) + 1
   k2 = k2 / nlev
   q =  mod(k2, qsize) + 1
   k2 = k2 / qsize
   ie = mod(k2, (nete-nets+1)) + nets
   k2 = k2 / (nete-nets+1)

   dudx00 =0.0d0
   dvdy00i=0.0d0
   do i=1,nv
    dudx00= dudx00+ Dvv(i,l)*(metdet(i,j,ie) *&
    (Dinv(1,1,i,j,ie)*gradQ5da(i,j,k,q,1,ie) +&
     Dinv(1,2,i,j,ie)*gradQ5da(i,j,k,q,2,ie)))

    dvdy00i =dvdy00i + Dvv(i,j)*(metdet(l,i,ie)*&
    (Dinv(2,1,l,i,ie)*gradQ5da(l,i,k,q,1,ie)+ &
    Dinv(2,2,l,i,ie)*gradQ5da(l,i,k,q,2,ie)))
   end do
   divdp4dhmpp(l,j,k,q,ie)= rmetdetp(l,j,ie) *
            ((rdx(ie))*dudx00+(rdy(ie))*dvdy00i)
  enddo
end subroutine hmpp_divergence_sphere
```

(b) HMPP Implementation

Fig. 5: The inlined and accelerated divergence_sphere code snippet

advantage of the node's shared memory. One of the challenges faced, when porting the code to OpenMP is to make sure memory access is consistent, by always accessing the same spherical element with the same thread including the data initialization loops. This improves locality by placing element's data in the core's local memory. We also must determine whether to privatize variables such as *gradQ*, a temporary variable that gathers data that is passed to the procedure or inline the procedure to avoid unnecessary data copies. For the inner loops we need to make sure loops get vectorized, if possible. When running the OpenMP kernel we noticed that using 4 threads gives the best performance with 510 milliseconds and a speedup of 2.67. The sequential version takes 1366 milliseconds.

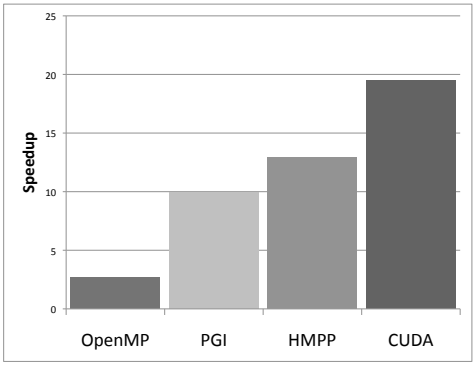
For the PGI and HMPP implementations of the kernel, it was necessary to inline the procedure *divergence_sphere* to map data parallel loops to a GPU and provide sufficient work. Also, this step is necessary if we want to accelerate the kernel at the *ie* loop, using the same approach adopted by the OpenMP implementation.

With the PGI accelerator directives, we needed to do some code restructuring to achieve good performance. We inlined the procedure *divergence_sphere* and inserted a *!$acc region* to accelerate the *ie* loop. This was necessary for the PGI

directives since it cannot handle function calls inside an accelerated region. The next step was to specify how to parallelize the the loop nest iterations across the GPU Symmetric Multi-processors (SM) and within the SMs efficiently. We use the parallel and vector clause to specify the vector size and grid size: in this case we specified a block size of $nv \times nv \times nv$. To avoid non-coalesced memory accesses we eliminated a temporary array gv and fused the inner loops. We also allocated and initialized all the data inside the GPU by using the PGI data region directive and copyout directive to obtain the results of the kerne. To achieve good performance, we allocated and initialized the twelve spectral elements state on the GPU. Figure 5(a) shows the implementation of the kernel using the PGI accelerator directives.

HOMME/SE Timings (Miliseconds)	
SERIAL	1366.37
HMPP Kernel	105.72
PGI Kernel	137.43
CUDA	70.00
OpenMP 4 Threads (best)	510.62

(a)HOMME/SE Timing Table



(b)HOMME/SE Divergence Sphere Speedup

Fig. 6: HOMME/SE Kernel Experiments

We used a similar code transformation to implement the kernel with HMPP directives; With HMPP we had to outline the ie loop to a separate procedure to create a *codelet*. We also had to transform the loops by collapsing the ie and q loops with the l and the e loop respectively to provide enough work for a two-dimensional thread block (At the time of writing, HMPP 2.5.0 only supported two dimensional thread blocks). Figure 5(b) shows the HMPP implementation of the kernel. The OpenMP version (with 4 threads) achieves a speedup of 2.67 (over the serial version). Without counting the data transfer time, the GPU implementations achieve a speed up of 9.9x for the (ACC) PGI directives, 12.92x for HMPP and 19.5x for the CUDA implementation

5 Conclusions

This paper explores GPU programming models and compares the use of two sets of accelerator directives in two real-world application kernel studies. We explain the challenges and limitations encountered and, based on the lessons

learned, reached initial conclusions on how to transform code to take advantage of the accelerator directive. In the HOMME/SE kernel, significant restructuring was needed to make sure the compilers generated the correct GPU scheduling (blocksize and grid size) and achieve a comparable performance to CUDA. We also compared the performance of running the codes on the GPU versus the CPU, and found that in all the cases the GPU yielded significantly better performance. In order to use the accelerator directives efficiently, it is necessary to perform code transformations to close the gap in performance to native CUDA implementations.

References

1. International Exascale Software Project Draft Report V 0.93. [Online]. http://www.exascale.org/iesp/MainPage.
2. Khronos Group. `http://www.khronos.org/`.
3. OpenCL 1.0 Specification. `http://www.khronos.org/opencl/`.
4. PGI Fortran & C Accelerator Compilers and Programming Model. `http://www.pgroup.com/lit/pgi_whitepaper_accpre.pdf`.
5. Rapidmind. `http://www.rapidmind.net/`.
6. Saman Amarasinghe, Michael I. Gordon, Michal Karczmarek, Jasper Lin, David Maze, Rodric M. Rabbah, and William Thies. Language and compiler design for streaming applications. *Int. J. Parallel Program.*, 33(2):261–278, 2005.
7. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.
8. J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science and Discovery*, 2(1):015001, 2009.
9. CAPS Enterprise. HMPP: A Hybrid Multicore Parallel Programming Platform. `http://www.caps-entreprise.com/en/documentation/caps_hmpp_product_brief.pdf`.
10. Tianyi David Han and Tarek S. Abdelrahman. /hi/cuda: a high-level directive-based language for gpu programming. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61, New York, NY, USA, 2009. ACM.
11. M. McCool and S. Toit. Metaprogramming GPUs with Sh. A K Peters, Ltd., 2004.
12. Richard Membarth, Frank Hannig, Jurgen Teich, Mario Korner, and Wieland Eckert. Frameworks for gpu accelerators: A comprehensive evaluation using 2d/3d image registration. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 78 –81, june 2011.
13. NVIDIA. CUDA. `http://www.nvidia.com/object/cuda_home.html`.
14. Google Code Project. Thrust: Thrust Quick Start Guide. `http://code.google.com/p/thrust/`.