NESUS cost IC1305
Network for Sustainable Ultrascale Computing
Second NESUS Workshop • September 2015 • Vol. I, No. 1

NekBone with Optimized OpenACC directives

Jing Gong, Stefano Markidis, Michael Schliephake, Erwin Laure
PDC Center for High Performance Computing,
KTH Royal Institute of Technology, Sweden, and
the Swedish e-Science Research Centre (SeRC)
(gongjing,markidis,michs,erwinl)@pdc.kth.se

Luis Cebamanos
The University of Edinburgh, UK
l.cebamanos@epcc.ed.ac.uk

Alistair Hart
Cray Exascale Research Initiative Europe, UK
ahart@cray.com

Misun Min, Paul Fischer
Argonne National Laboratory, U.S.A
mmin,fischer@mcs.anl.gov

Abstract

Accelerators and, in particular, Graphics Processing Units (GPUs) have emerged as promising computing technologies which may be suitable for the future Exascale systems. Here, we present performance results of NekBone, a benchmark of the Nek5000 code, implemented with optimized OpenACC directives and GPUDirect communications. Nek5000 is a computational fluid dynamics code based on the spectral element method used for the simulation of incompressible flow. Results of an optimized NekBone version lead to 78 Gflops performance on a single node. In addition, a performance result of 609 Tflops has been reached on 16,384 *GPUs of the Titan supercomputer at Oak Ridge National Laboratory.*

Keywords NekBone/Nek5000, OpenACC, Spectral element method, GPUDirect

I. Introduction

There is a long history to employ GPUs to accelerate Computational Fluid Dynamic (CFD) codes [1, 2, 3]. However, most implementations use the Nvidia parallel programming and computing platform; CUDA. This means that developers need to rewrite their original applications in order to obtain a substantial performance improvement [4].

OpenACC [5] is a directive-based HPC parallel programming model, using host-directed execution with an attached accelerator device. In addition, GPUDirect for communication enables a direct path for data exchange between GPUs bypassing CPU host memory. In the paper, we extend the initial results [6, 7, 8] on porting Nek5000 to GPU systems, to enhance and optimize the performance on massively parallel hybrid CPU/GPU systems. In this implementation, the large-scale parallelism is handled by MPI, while OpenACC deal with the fine-grained parallelism of matrix-matrix multiplication.

The paper is organized as follows. In Section II we give an overview of the Nek5000 and NekBone code. In Section III we discuss in details regarding

optimized matrix-matrix multiplications and gather-scatter operators. The performance results are provided in Section IV. Finally, we summarize the results and further works.

II. Nek5000 and its NekBone benchmark

Nek5000 [9] is an open-source code for simulating incompressible flows using MPI for parallel communication. The code is widely used in a broad range of applications. The Nek5000 discretization scheme is based on the spectral-element method [10, 11]. In this approach, the incompressible Navier-Stokes equations are discretized in space by using high-order weighted residual techniques employing tensor-product polynomial bases.

In Nek5000, the derivatives in physical space can be calculated using the chain rule [12],

$$\frac{\partial \mathbf{U}}{\partial x_l} = \sum_{m=1}^{3} \frac{\partial \mathbf{U}}{\partial r_m} \frac{\partial r_m}{\partial x_l}. \tag{1}$$

Typically, this equation (1) is evaluated on the GLL points of N. In the case of three-dimensional with number of elements E, it creates an additional $9n$ memory references and $36n$ operations, where $n = E \cdot N^3$ is the total number of gridpoints. The tensor-product-based operator evaluation can be implemented as matrix-matrix products. This implementation of (1) makes possible to port the most time-consuming parts of the code into a GPU-accelerated system.

NekBone [13] is configured with the basic structure and user interface of the extensive Nek5000 software. NekBone solves a standard Poisson equation using the spectral element method with an iterative conjugate gradient solver and exposes the principal computational kernel to reveal the essential elements of the algorithmic-architectural coupling that is pertinent to Nek5000. Consequently, the results from investigating the performance and profiling of NekBone can be directly applied to Nek5000.

III. Optimized OpenACC implementations

III.1 Matrix-Matrix Multiplications

The matrix-matrix multiplications are performed through the direct operator evaluation based on Equation (1). Algorithm 1 shows the pseudo-code of the `local_grad_acc` subroutine which computes the derivatives of U using CCE compiler.

Algorithm 1 CCE version for the final optimized derivative operations.

```
local_grad_acc
 !$ACC DATA PRESENT(w,u,gxyz,ur,us,ut,wk,dxm1,dxtm1)
 !$ACC PARALLEL LOOP COLLAPSE(4) GANG WORKER VECTOR
 !$ACC& VECTOR_LENGTH(128)  PRIVATE(wr,ws,wt)
      do e = 1,nelt
         do k=1,nz1
         do j=1,ny1
         do i=1,nx1
            wr = 0
            ws = 0
            wt = 0
 !$ACC LOOP SEQ
            do l=1,nx1    ! serial loop, no reduction needed
               wr = wr + dxm1(i,l)*u(l,j,k,e)
               ws = ws + dxm1(j,l)*u(i,l,k,e)
               wt = wt + dxm1(k,l)*u(i,j,l,e)
            enddo
            ur(i,j,k,e) = gxyz(i,j,k,1,e)*wr
     $                  + gxyz(i,j,k,2,e)*ws
     $                  + gxyz(i,j,k,3,e)*wt
         enddo
         enddo
         enddo
      enddo
!$ACC END PARALLEL LOOP
...
```

Algorithm 2 illustrates the use of OpenACC directives with PGI compiler. Here, the OpenACC directives `KERNELS` and `LOOP VECTOR` are used for an optimized performances with the PGI compiler.

The other optimized implementation evaluated in this paper is to call CUDA device functions from within OpenACC kernels. This can be done using the OpenACC directive `!$acc host_data use_device`. The OpenACC construct `host_data` indicates the address of device data available on the host, then the arrays are listed in the `use_device` clause within the `host_data` region. The compiler will generate code to use the device copy of the arrays, instead of the host copy. The interface implemented between OpenACC and CUDA functions is provided below.

```
!$acc host_data use_device(w,u,ur,us,ut,gxyz,dxm1,dxtm1)
   if (nx1.eq.8) then
     call ax_cuf8<<<nelt,dim3(nx1,ny1,nz1)>>>(w,u,
  $                ur,us,ut,gxyz,dxm1,dxtm1)
...
   else if (nx1.eq.14) then
     call ax_cuf14<<<nelt,dim3(nx1,ny1,nz1/2)>>>(w,u,
  $                ur,us,ut,gxyz,dxm1,dxtm1)
   else
     call ax_cuf16<<<nelt,dim3(nx1,ny1,nz1/4)>>>(w,u,
```

Algorithm 2 PGI version for the final optimized derivative operations.

```
local_grad_acc
 !$ACC DATA PRESENT(w,u,gxyz,ur,us,ut,wk,dxm1,dxtm1)
 !$ACC KERNELS
 !$ACC& GANG
      do e = 1,nelt
 !$ACC& LOOP VECTOR(NZ1)
         do k=1,nz1
 !$ACC& LOOP VECTOR(NY1)
         do j=1,ny1
 !$ACC& LOOP VECTOR(NX1)
         do i=1,nx1
            wr = 0
            ws = 0
            wt = 0
 !$ACC LOOP SEQ
            do l=1,nx1    ! serial loop, no reduction needed
               wr = wr + dxm1(i,l)*u(l,j,k,e)
               ws = ws + dxm1(j,l)*u(i,l,k,e)
               wt = wt + dxm1(k,l)*u(i,j,l,e)
            enddo
            ur(i,j,k,e) = gxyz(i,j,k,1,e)*wr
     $                  + gxyz(i,j,k,2,e)*ws
     $                  + gxyz(i,j,k,3,e)*wt
         enddo
         enddo
         enddo
      enddo
!$ACC END KERNELS
...
```

```
  $                  ur,us,ut,gxyz,dxm1,dxtm1)
   endif
   istat = cudaDeviceSynchronize()
!$acc end host_data
```

The utilization of shared memory in GPUs is very important for writing optimized CUDA code since access to shared memory is much faster than to global memory. Shared memory is allocated per thread block, therefore all threads in a block have access to the same shared memory. On NVIDIA Kepler GPUs with compute capability 3.x, shared memory has 32 banks, with each bank having a bandwidth of 64-bits per clock cycle. Kepler GPUs are configurable where either successive 32-bit words or 64-bit words are assigned to successive banks. This is particularly important for cases with higher polynomial degree. Considering that our matrix-matrix operations use three temporary arrays of size N^3 a total of 64KB shared memory is required for the case $N = 14$ with double precision. An example of such implementation in CUDA FORTRAN for polynomial degree 14 can be seen in Algorithm 3.

Algorithm 3 CUDA FORTRAN version for the final tuned derivative operations

```
subroutine local_grad_cuf14
    real, intent(out) :: w(lx1,ly1,lz1,lelt)
    real, intent(in)  :: u(lx1,ly1,lz1,lelt)

    real gxyz(lx1,ly1,lz1,2*ldim,lelt)

    real, intent(in) :: dxm1(lx1,lx1)
    real, intent(in) :: dxtm1(lx1,lx1)

    real rtmp,stmp,ttmp,wijk1e,wijk2e
    real, shared :: shdxm1(lx1,lx1)
    real, shared :: shdxtm1(lx1,lx1)
    real, shared :: shur(lx1,ly1,lz1)
    real, shared :: shus(lx1,ly1,lz1)
    real, shared :: shut(lx1,ly1,lz1)
    integer e,i,j,k,l

    e = blockIdx%x
    k = threadIdx%z
    j = threadIdx%y
    i = threadIdx%x

     if (k.eq.1) then
        shdxm1(i,j) = dxm1(i,j)
        shdxtm1(i,j) = dxtm1(i,j)
     end if
     call syncthreads()

     rtmp = 0.0
     stmp = 0.0
     ttmp = 0.0
     do l=1,lx1
       rtmp = rtmp+shdxm1(i,l)*u(l,j,k,e)
       stmp = stmp+shdxm1(j,l)*u(i,l,k,e)
       ttmp = ttmp+shdxm1(k,l)*u(i,j,l,e)
     enddo
         shur(i,j,k) = gxyz(i,j,k,1,e)*rtmp
    $                + gxyz(i,j,k,2,e)*stmp
    $                + gxyz(i,j,k,3,e)*ttmp
     rtmp = 0.0
     stmp = 0.0
     ttmp = 0.0
     do l=1,lx1
       rtmp = rtmp+shdxm1(i,l)*u(l,j,k+7,e)
       stmp = stmp+shdxm1(j,l)*u(i,l,k+7,e)
       ttmp = ttmp+shdxm1(k+7,l)*u(i,j,l,e)
     enddo
         shur(i,j,k+7) = gxyz(i,j,k+7,1,e)*rtmp
    $                  + gxyz(i,j,k+7,2,e)*stmp
    $                  + gxyz(i,j,k+7,3,e)*ttmp

     call syncthreads()
...
```

III.2 GPUDirect Gather-Scatter operator

The Gather-Scatter operator is implemented by `gs_op` routine in NekBone. Notice that we have already split the `gs_op` routine with local gather and scatter operations on GPUs in [7]. In the implementation only the non-local data need to be transferred between GPU and CPU to conduct MPI communication. The non-local data is exchanged with standard MPI subroutines `MPI_Irecv()`, `MPI_Isend()`, `MPI_Waitall()` with combination of `MPI_Waitall`. The modified `gs_op` operator with local gather and scatter is described in Algorithm 4.

Algorithm 4 Modified Gather-scatter operator (adapted from [7]).

```
unew_l = u_l
! u_g = Q u_l Local Gather on GPU
 !$ACC PARALLEL LOOP
 u_g = 0
 do i = 1, nl
    li = lgl(1,i)
    gi = lgl(2,i)
    u_g(gi) = u_g(gi)+u_l(li)
 enddo

gs_op(u_g,1,1,0) ! MPI communication between CPUs

! u_l = Q^T u_g Local Scatter on GPU
!$ACC PARALLEL LOOP
do i = 1, nl
   li = lgl(1,i)
   gi = lgl(2,i)
   unew_l(li) = u_g(gi)
enddo
```

In [8] a new version of Gather-scatter operator with GPUDirect is developed. The new version accelerates all four parts of the gather-scatter routine: local-gather, global-scatter, global-gather, and local-scatter, whereas previous versions only accelerated the local parts. Accelerating the global loops allows us to use the GPUDirect pragmas, as the buffers are prepared on the GPU. The local-gather and local-scatter loops above are included into the tuned `fgs_fields_acc`. In a similar manner, the global-scatter and global-gather loops are also accelerated. In this new version, data communication between CPUs is not necessary since the GPU would efficiently perform the local additions and does not need any information from other nodes.

IV. Performance Results

IV.1 Systems and compiler environments

We performed our simulations on few supercomputing systems. *Titan*, a Cray XK7 system at the Oak Ridge Leadership Computing Facility (OLCF), consists of 18,688 AMD Opteron 6274 16-core CPUs and 18,688 Nvidia Tesla K20X GPU computing accelerator with 6GB of GDDR5 memory each. Titan has a hybrid architecture with peak performance of 27 Pflops and 40 PB of Lustre storage. The pair of nodes shares a Gemini high-speed interconnect router in a 3D torus topology. *Curie* is a PRACE Tier-0 system, installed at CEA in France. The Curie system has total 144 hybrid nodes. Each hybrid node has two 4-cores Westmere-EP@2.67GHz CPUs and two Nvidia M2090. The compute nodes are connected through a QDR InfiniBand network and the topology of this InfiniBand network is a full fat tree. *Raven* system at Cray has 8 XK7 compute nodes with 8 NVIDIA Telsa K20 GPUs. A compute node of Raven has one Opteron processor with a total of 16 processor cores and 2 NUMA nodes with a total of 16 GB of DDR3-1600 main memory. The Opteron processors run at 2.1 GHz. Each GPU has 6 GB of GDDR5 memory.

Raven and Titan support GPU-Direct with both Cray CCE and PGI compilers. The Curie system supports only the PGI compiler without GPU-Direct features.

IV.2 Single GPU Performance tests

We optimized our GPU enabled code with CCE and PGI compilers for the compute intensive matrix-matrix multiplications routines, as discussed in a previous section using Algorithms 1–3. Figures 1–4 show the single GPU performance tests on different platforms. For all cases, the performance critically depends on the computational workload on the GPU. The more calculations are completed on the GPU, the performance is higher. the performance increases as the number of elements (E) increases and the performance significantly increases with the polynomial order (N). In addition, different compilers and versions also affect the performance of NekBone.

On the Curie hybrid nodes the performance increases around 5-10% with the optimized OpenACC directives compared to the original case, see Figures 1 and 2. The maximum performance achieved is 43.6 Gflops with elements $E = 4096$ and polynomial order

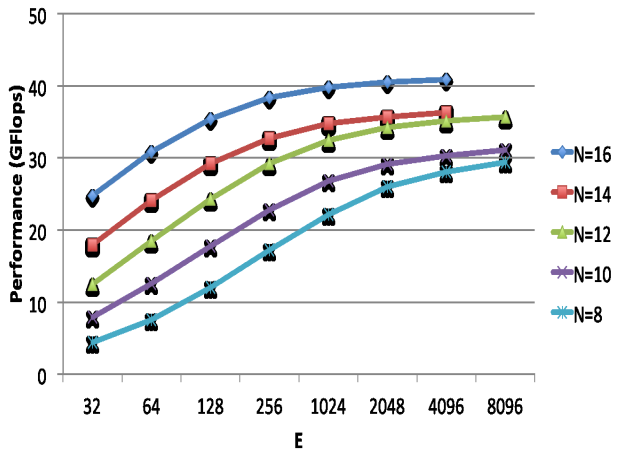

Figure 1: The performance with the total number of grid points $n = E \cdot N^3$ for $E = 32, 64, \ldots 8096$ and $N = 8, 10, 12, 14, 16$ on single node of Curie using PGI compilers. Original OpenACC directives.

$N = 16$.

Figure 4 shows the performance on Titan with Cray CCE, PGI, and PGI CUDA FORTRAN compilers. The CUDA FORTRAN code is around 10% faster than the OpenACC code. However, it is also important to highlight the little effort required to port an application like NekBone to GPU systems using OpenACC compared to the CUDA porting process. Furthermore, the small number of additional lines of code required to port an application to OpenACC is not comparable with the addition of CUDA kernels code. Such is the case that is necessary to rewrite the CUDA code for each polynomial order (N) case.

IV.3 Multi GPU Performance tests

From Figures 1 and 2, even without MPI communication we can identify that the performance of the matrix-matrix multiplication kernels highly depends on the order of polynomial (N) and number of elements (E). Larger values of N give better performance. This effect could be cause due to the amount of work per thread (which is proportional to N) is greater, which either leads to better kernel efficiency or assists to offset the latency cost of launching kernels. Also, the MPI communication overlaps the less workload of GPUs. Consequently, the degradation performance with the increase of the number of GPUs for the strong scalability is expected. The performance of OpenACC version is quite different from the original MPI ver-

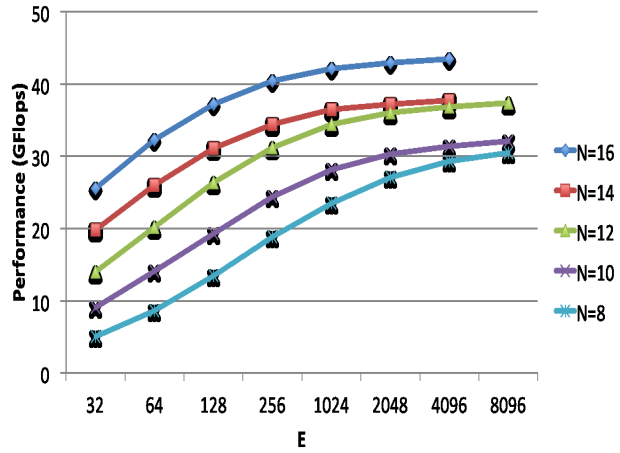

Figure 2: The performance with the total number of grid points $n = E \cdot N^3$ for $E = 32, 64, \ldots 8096$ and $N = 8, 10, 12, 14, 16$ on single node of Curie using PGI compilers. Optimized OpenACC directives.

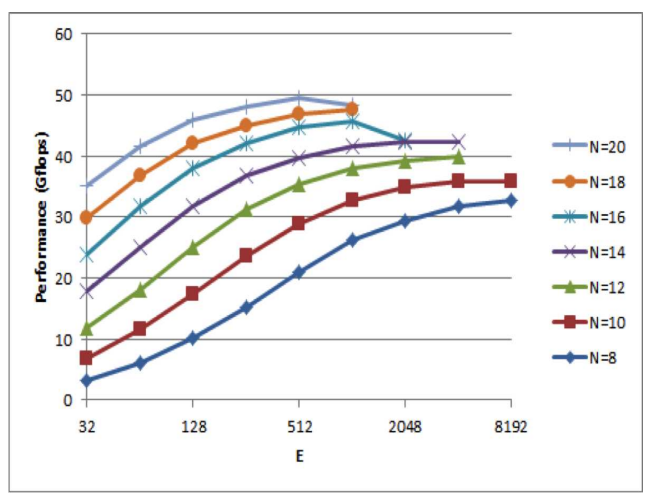

Figure 3: The performance with the total number of grid points $n = E \cdot N^3$ for $E = 32, 64, \ldots 8096$ and $N = 8, 10, 12, 14, 16$ on single node of Raven using CCE compilers.

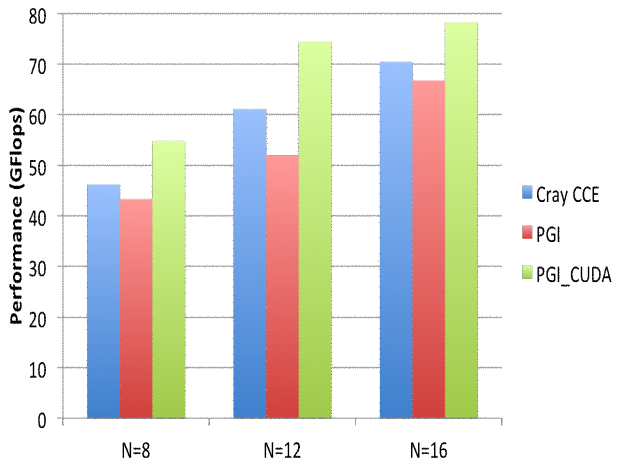

Figure 4: The performance with the total number of grid points $n = E \cdot N^3$ for $E = 512$ and $N = 8, 12, 16$ on single node of Titan using CCE and PGI compilers.

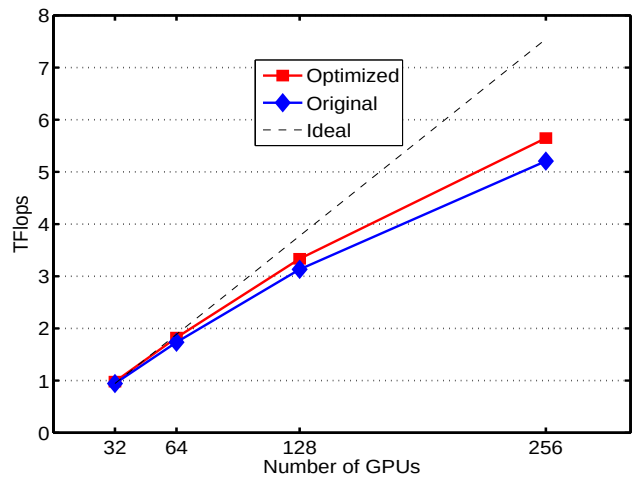

Figure 5: Strong scaling results with total number points are $n = 1024 \cdot 16^3 \cdot 32$ on Curie using PGI compilers.

sion where parallel efficiency 0.6 has been measured for strong scalability between 32768 and 1048576 MPI ranks [14].

Figure 5 shows the NekBone strong scaling performance, measured in Tflops, with up to 256 GPUs as a solid line, while the black dashed line represents the ideal strong scaling on Curie. The parallel efficiency with 256 GPUs was 69% compared when using 32 GPUs. In order to get better performance we should use as many elements per node as we can fit into GPU memory (6GB for the Tesla M2090 card).

Each Curie hybrid node has two sockets and each GPU is bound to a socket. By the default both processes are running on only one GPU. As a result, the `CUDA_VISIBLE_DEVICES` variable should be set to 0 or 1 depending on the rank of the process. In addition, it is necessary to bind the processes to the GPU to make sure they run on the same socket that hosts the desired GPU. However the binding of the processes still slows down the application since both processes share some resources. For instance, I/O operations are slower or the MPI communications may behave differently. Figures 6 and 7 show the weak scaling results on Curie where we can see how the curve representing the optimized version is growing apart as more GPUs are used in the simulation.

For the test on Titan, 1024 elements and 16th-order polynomials were used for a total of $4,194,304$ points per GPU. Figure 8 shows the NekBone weak scaling performance, measured in TFlops, with up to $16,384$

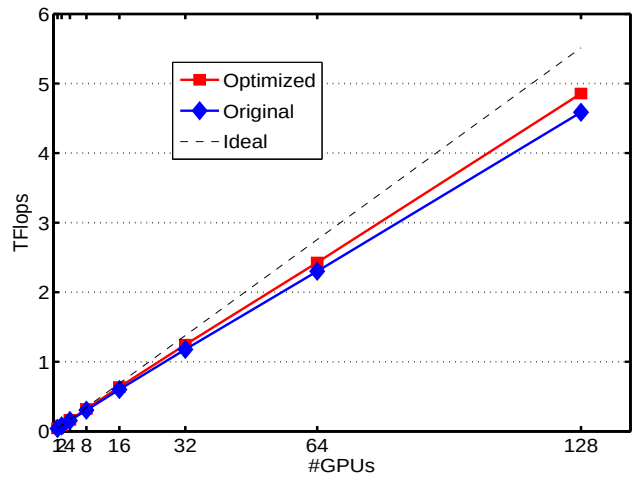

Figure 6: Weak scaling results with $n = 1024 \cdot 16^3$ per GPU on Curie using PGI compilers.

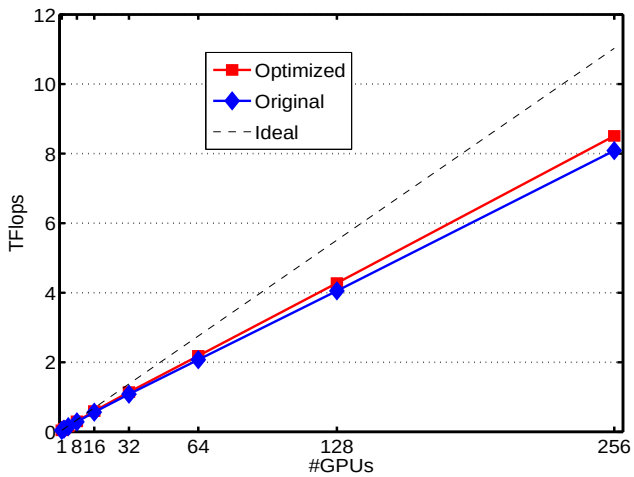

Figure 7: Weak scaling results with $n = 1024 \cdot 16^3$ per GPU on Curie using PGI compilers.

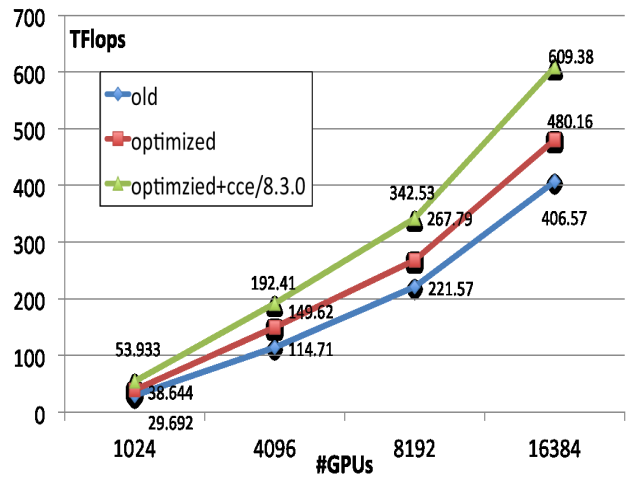

Figure 8: Weak scaling performance with $n = 1024 \cdot 16^3$ per GPU on Titan using CCE compilers.

GPUs. The parallel efficiency on 16,384 GPUs was 52.8% compared with single GPU and the maximum performance obtained is 609.8 Tflops with optimized OpenACC code. This good scaling results is achieved by using the proper construction of the global communication and the code simplicity.

IV.4 GPUDirect tests

The Cray performance analysis tool `CrayPat` is used to conduct the profiling analysis for the data communication. The tests are conducted on the Raven system with 8 GPUs and the number of grid points is $n = 1024 \cdot 16^3$ per GPU. The total time on the MPI communication is 2.52 sec with the modified gather-scatter Algorithm 4, see Profiling by Function Group Table using CrayPat below.

```
 Table 1:  Profile by Function Group and Function

  Time% |       Time |     Imb. |  Imb. |    Calls |Group
        |            |     Time | Time% |          | Function
 100.0% | 11.864466 |       -- |    -- | 34739.0 |Total
|-----------------------------------------------------------------
|  94.7% | 11.235358 |       -- |    -- | 30332.2 |USER
||----------------------------------------------------------------
||  19.9% |  2.366802 | 0.000701 |  0.0% |    200.0 |dssum_acc_.ACC_
...
||   1.3% |  0.152252 | 0.000371 |  0.3% |    200.0 |dssum_acc_.ACC_
...
```

With the Algorithm developed in [8], the total time is reduced to 2.36 sec. This can be seen in the next Table.

```
Table 1:  Profile by Function Group and Function

  Time% |       Time |     Imb. |  Imb. |    Calls |Group
        |            |     Time | Time% |          | Function
 100.0% | 11.806398 |       -- |    -- | 38739.0 |Total
|-----------------------------------------------------------------
|  94.8% | 11.193856 |       -- |    -- | 34332.2 |USER
||----------------------------------------------------------------
||  20.0% |  2.357107 | 0.000715 |  0.0% |    200.0 |fgs_fields_acc
...
```

V. Conclusions

A hybrid Nek5000 version was created to exploit the processing power of multi-GPU systems by using OpenACC compiler directives. This work focused on advance GPU optimizing and tuning of the most time-consuming parts of Nek5000, namely the matrix-matrix multiplication operations and the preconditioned linear solve operation. Furthermore, the gather-scatter kernel used with MPI operations has been redesigned in order to decrease the amount of data transferred between the host and the accelerator. The speed-up achieved using OpenACC directives is 1.30 with a 16th order polynomial on 16,384 GPUs of the Cray XK7 supercomputer when compared to 16,384 full CPU nodes having 262,144 CPU cores in total.

We have been able to compare performance results of NekBone versions running with CUDA FORTRAN and OpenACC. Although OpenACC has proven to be a simpler solution for porting applications to GPUs, our results demonstrate that CUDA is still more efficient and that in OpenACC there is still room for improvement.

With a multi-GPU setup, the gather-scatter operator and the associated MPI communication can be improved. The original gather-scatter operator was split

into two parts. First a local gather on the GPU is performed, followed by the transfer of the boundary values at the interfaces of the domain. Then the boundary values need to be copied to a local CPU memory, communicated via network to the memory of another CPU, and then transferred back a memory of a remote GPU to finally carry out a local scatter on the GPU. This approach allows a considerable reduction in the amount of data to be moved from the GPU and CPU memory and vice versa.

In spite of the reduction in the amount of data transferred, the additional transfers between the host and accelerator have an effect on the achievable performance. In the future, we will employ the techniques such as overlapping of GPU kernels with host-accelerator memory transfers to further increase the performance of the OpenACC version of Nek5000.

Acknowledgments

This work is partially supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS) and the Swedish e-Science Research Center (SeRC). We acknowledge PRACE for awarding us access to resource CURIE based in France at CEA as well as the computing time on the Raven system at Cray and the Titan supercomputer at Oak Ridge National Laboratory. We would also like to thank Brent Leback for the CUDA FORTRAN code used in the paper.

References

[1] J. H. Chen, A. Adhere, B. De Supinski, M. DeVries, E. Hawkes, S. Klasky, W. Liao, K. Ma, J. Mellor-Crummey, N. Podhorszki, et al., "Terascale direct numerical simulations of turbulent combustion using S3d", *Computational Science & Discovery* vol. 2, no. 1, 2009.

[2] D. C. Jespersen, "Acceleration of a CFD code with a GPU", *Scientific Programming*, vol. 18, no. 3-4, pp. 193-201, 2010

[3] C. K. Aidun and J. R. Clausen, "Lattice Boltzmann method for complex flows", *Annual Review of Fluid Mechanics*, vol. 42, pp. 439-472, 2010

[4] K. Niemeyer and C. Sung, "Recent progress and challenges in exploiting graphics processors in computational fluid dynamics", *The Journal of Supercomputing*, vol. 67, no. 2, pp. 528-564, 2014.

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

[6] J. Gong, S. Markidis, M. Schliephake, E. Laure, D. Henningson, P. Schlatter, A. Peplinski, A. Hart, J. Doleschal, D. Henty, and P. Fischer, Nek5000 with OpenACC, in *Solving Software Challenges for Exascale, the International Conference on Exascale Applications and Software, EASC 2014 Stockholm, Sweden, April 20-23, 2014*, Stefano Markidis, Erwin Laure (Eds.), Springer LNCS8759, 2015.

[7] S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, and P. Fischer, "OpenACC acceleration of the Nek5000 spectral element code", *International Journal of High Performance Computing Applications*, vol. 29, pp. 311-319, 2015.

[8] M. Otten, J. Gong, A. Mametjanov, A. Vose, J. Levesque, P. Fischer, and M. Min "An MPI/OpenACC Implementation of a High Order Electromagneticcs Solver with GPUDirect Communication", accepted in *International Journal of High Performance Computing Applications*.

[9] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, Nek5000 web page, Web page: http://nek5000.mcs. anl. gov.

[10] A. T. Patera, "A spectral element method for fluid dynamics: laminar flow in a channel expansion", *Journal of Computational Physics*, vol. 54, No. 3, pp. 68-488, 1984

[11] H. M. Tufo and P. F. Fischer, "Terascale spectral element algorithms and implementations", in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, ACM, 1999, p. 68.

[12] M. Deville, P. Fischer, and E. Mund, *High-order methods for incompressible fluid flow*, Cambridge University Press, 2002.

[13] NekBone: Proxy-Apps for Thermal Hydraulics, `https://cesar.mcs.anl.gov/content/software/thermal_hydraulics`

[14] Nek5000 strong scaling tests to over one million processes. `http://nek5000.mcs.anl.gov/index.php/Scaling`