Hybrid Programming using OpenSHMEM and OpenACC

Matthew Baker[1], Swaroop Pophale[3], Jean-Charles Vasnier[4], Haoqiang Jin[2], and Oscar Hernandez[1]

[1] Oak Ridge National Laboratory, Oak Ridge, Tennessee, 37840, USA
bakermb@ornl.gov,oscar@ornl.gov
[2] NASA Ames, Moffet Field, California USA,
haoqiang.jin@nasa.gov
[3] University of Houston, Houston, Texas 77004, USA,
spophale@cs.uh.edu
[4] CAPS entreprise, France,
jvasnier@caps-entreprise.com

Abstract. With high performance systems exploiting multicore and accelerator-based architectures on a distributed shared memory system, heterogenous hybrid programming models are the natural choice to exploit all the hardware made available on these systems. Previous efforts looking into hybrid models have primarily focused on using OpenMP directives (for shared memory programming) with MPI (for inter-node programming on a cluster), using OpenMP to spawn threads on a node and communication libraries like MPI to communicate across nodes. As accelerators get added into the mix, and there is better hardware support for PGAS languages/APIs, this means that new and unexplored heterogenous hybrid models will be needed to effectively leverage the new hardware. In this paper we explore the use of OpenACC directives to program GPUs and the use of OpenSHMEM, a PGAS library for one-sided communication between nodes. We use the NAS-BT Multi-zone benchmark that was converted to use the OpenSHMEM library API for network communication between nodes and OpenACC to exploit accelerators that are present within a node. We evaluate the performance of the benchmark and discuss our experiences during the development of the OpenSHMEM+OpenACC hybrid program.

1 Introduction

New HPC systems are increasingly turning to accelerators to increase compute power while mitigating the rising cost of power [1]. For example, four of the top ten super computers use GPUs as their main devices to perform the majority of the computations. Oak Ridge National Laboratories Titan [2], a DOE leadership class machine, makes extensive use of GPUs, using one Nvidia Kepler GPU per node. Without a major breakthrough in technology, the future of the fastest super computers will consist of clusters with multiple cores and attached to specialized devices for accelerators, interconnects and I/O. Modern cluster nodes

have many different types of hardware that need to be exploited efficiently to make the maximum use of the provided resources. Current nodes have multiple sockets with attached memory, each socket has a CPU with multiple cores. On top of this the node usually has an attached accelerator, currently the most common one is the GPU. In addition the nodes are all connected with network hardware that provides better support for PGAS languages/libraries to allow them to communicate. This means that each node has three different major components that need to be programmed for. Each of these components has its own programming model with its own challenges. When these models are used together for hybrid programming models, new challenges arise, specially when one of the models deals with heterogeneous programming. In this paper we explore a new hybrid programming model OpenSHMEM+OpenACC.

OpenSHMEM is the result of standardizing several shmem libraries [3]. It is a one-sided communication library where individual processes do one-sided puts and gets, as compared to MPI that does synchronized send/receive between pairs of processes. This allows for data to be sent without having to wait on remote nodes to do communication. OpenACC is the result of standardizing compiler directives for accelerator programming sanctioned by the OpenACC organization. It allows for an OpenMP-like programming with support for incremental parallelism. This includes the ability to incrementally add directives to a code to program for GPUs, rather than having to do a considerable amount of code restructure to just start using an accelerator (like with the OpenCL standard). We use the NASA Advanced Supercomputing (NAS) Block Tri-diagonal (BT) Multizone benchmark [4] to evaluate our results. This benchmark is structured so that there are multiple zones that can be solved independently with the boundary values of each zone exchanged on each iteration, making it well suited to experimentation with heterogeneous hybrid programming model.

This paper is organized into 5 sections. In Section 2 we discuss the other research done on hybrid programming models and provide background information on the BT-MZ benchmark used, the OpenACC directives, and the OpenSHMEM library in Section 3 . In Section 4 we discuss the implementation details of how OpenSHMEM and OpenACC are used together. The results are presented in Section 5. We discuss the platforms used and the timings from running BT-MZ on those platforms. In Section 6 we interpret the results we obtain and discuss the future paths for exploration in this hybrid programming.

2 Related Work

Hybrid models that explore shared-memory and distributed-memory programming have been researched over the last few decades. The idea is to exploit the strengths of the different models, including the in-node efficiency, memory savings, accelerator programming, and the scalability characteristics within a distributed memory system. The shared and distributed programming models models have been evolving separately and an attempt to unify them resulted in the creation of new languages and models such as the HPCS and PGAS lan-

guages (X10, Chapel, Fortress, UPC, etc). In terms of heterogenous programming, there have been attempts to explore the use of message passing libraries and accelerator languages and APIs (i.e. CUDA, OpenCL). Recently, a high-level approach to program accelerators has been released, called OpenACC that improves portability across accelerators. Some applications have successfully used the model of MPI/OpenACC, MPI/OpenCL, MPI/CUDA. Very little work has been done to explore OpenSHMEM with accelerator programming models.

3 Background

In this section we introduce the different models that we used to experiment with heterogenous hybrid programming.

3.1 OpenSHMEM

The OpenSHMEM library is a PGAS library that provides a library API for programmers using the Single Program Multiple Data (SPMD) programming paradigm for programs written in C, C++ and Fortran. The OpenSHMEM Specification [5] provides the definition, functionality and expected behavior of these powerful library calls that are meant for communicating and processing data. The SGI SHMEM library specification and implementation motivated OpenSHMEM Specification 1.0 which was finalized by the OpenSHMEM community in early 2012. OpenSHMEM is an evolving open standard for all SHMEM library implementations and we expect many useful changes to the API and library in the near future to be able to cater to the growing and ever changing high performance computing environment. The current library specification provides API for *one-sided* reads and writes, remote atomic memory operations, broadcasts, reductions, collects, distributed locking, and collective and point-to-point synchronization and ordering primitives. OpenSHMEM *put/get* calls provide excellent opportunities for hiding communication latency by overlapping communication with computation when the underlying hardware supports true one-sided remote direct memory access (RDMA). Along with performance, application programmers require both portability and productivity and the OpenSHMEM library facilitates this by providing a standard and simple API.

3.2 OpenACC

Directives comprise a mechanism that allows a program to "direct" (or "hint") the compiler as to what it should do about a region of code that is often in proximity to the directive's occurrence in the program. In the C/C++ and Fortran languages directives appear as lines prefixed with `#pragma` *id* or `!$`*id*, respectively, where *`id`* signifies the directives API that these directives belong to. In the case of Fortran, for instance, OpenMP and HPF use `omp` and `hpf` respectively. It is the directive's API specification that designates what is legal – when put this way, directive APIs can be thought of as separate languages meant to annotate C & Fortran sources.

Using directives to program accelerators is not new. Data and executable code placement on National Semiconductors' NAPA1000 reprogrammable chips is steered from pragma-annotated C [6], while intelligent memory operations can be offloaded onto a FlexRAM array by using the the CFlex pragmas for the C language [7]. IBM's Cell BE is programmable with the Cell Superscalar (CellSs) [8]. The accessibility to GPUs saw the emergence of OpenHMPP [9] and hiCUDA [10], while for the more recent Intel Many Integrated Cores (MIC), it was shown how work offloading can be achieved by OpenMP offloading (`#pragma offload target(mic)` combined with OpenMP directives) [11].

The OpenACC specification comprises a programming model supported by a directives-based API that was put together by a consortium of four, namely CAPS enterprise, Cray Inc, the Portland Group Inc (PGI) and NVIDIA. The specification was put together to provide a prototype implementation to speedup the OpenMP accelerator directive process, which constantly gets merged in the OpenMP accelerator model [12]. The OpenACC defines a host and accelerator programming model where accelerated regions are used to define which parts of the program can be (1) offloaded to an accelerator and/or (2) data regions describe the data motion between the accelerator and the CPU, where some host variables become available on the device. (`acc_data`).

The data regions address concerns with the (mostly) disjoint CPU and accelerator memory spaces and the lifetime of data objects. Similar to OpenMP where the programmer tags objects as `private`, `shared`, `lastprivate`, OpenACC offers a number of clauses. Some of the supported clauses are the following: (1) the `copyin` clause for objects that are to copied over to the accelerator at the begining of the region, (2) the `copyout` and then copied back to the CPU at the end of the region, (3) the `present` clause for checking if an object is already available on the accelerator (in order to avoid redundant or outdated copies) and (4) `copy` that combines `copyin` and `copyout`.

In recognition of existing accelerators' processing element (PE) topology, OpenACC orchestrates the work to be assigned to the accelerator's PE in a hierarchical fashion: there are *gangs* of *workers* where each worker performs a *vector* operation. The actual mapping is both target and compiler specific. In the case of NVIDIA GPUs, for instance, the number of gangs corresponds to the number of CUDA threadblocks, the number of workers determines the size of the warps or the Y dimension of the threadblock while the vector suggests the SIMD length [13]. To designate work for offloading onto the accelerator, OpenACC makes available the `parallel` directive, which will, essentially, launch work on the device according to a compiler-selected or user-selected configuration of gangs, workers and vectors.

In addition to the `parallel` directive, OpenACC offers the `kernel` and `loop` directives. Given a region of sequential statements, where statements may be loops or less complex statements, the `kernel` directive instructs the compiler to organize the statements into *kernels* and execute them sequentially on the accelerator. An implementation is free to modify the organization of gangs, workers and vectors between launches. The reader may assume that a mapping to CUDA would suggest that the kernels have been queued up for serial launching over a

CUDA stream. The `loop` directive is meant to map loop nests onto the gang, worker and vector hierarchy.

3.3 Hybrid programming with OpenSHMEM and OpenACC

Using the OpenSHMEM and OpenACC programming models together is a new type of hybrid model. The low latency characteristic of OpenSHMEM combined with accelerator programming of OpenACC makes it an attractive model to explore. However, there are limitations from both APIs that makes it hard for them to interoperate. For example, the accelerator memory is not part of the *symmetric memory* required and used by the OpenSHMEM library. Also, the current OpenSHMEM 1.0 specification is not thread safe which limits the use of OpenSHMEM library API to outside of OpenACC regions.

3.4 BT Multizone Benchmark

The Block Tri-diagonal (BT) benchmark is part of the NPB benchmark suite. The benchmark simulates a CFD application that solves 3-dimensional compressible Navier-Stokes equations using Alternating Direction Implicit (ADI) to find the finite difference solution to the problem by solving three sets of uncoupled systems of equations in x, y and z directions. These equations are block tridiagonal with a 5x5 block size. The multi-zone version of the benchmark a logically rectangular discretization mesh is divided into a two-dimensional horizontal tiling of three-dimensional zones [4] and the aspect ratios are changed from the original NPB to avoid pathologically shaped zones. In the reference BT-MZ implementation (MPI+OpenMP) a MPI process executes the initialization step and after initial setup and synchronization of all processes the benchmark loops over the computation kernels. Communication between processes occurs after the computations are completed and the *root* process verifies the results obtained for the problem size class chosen. The number of zones grows with the problem size and the ratio of the largest zone over the smallest zone is about 20. The zones span a signifiant range and can be modulated by using different classes of input data.

4 Implementation

The OpenSHMEM+OpenACC version of the BT-MZ benchmark is structured into five distinct steps (similar to the reference BT-MZ implementation). The first setup is to set up the zones, then all the zones are initialized. These two steps are done independently on each node and without accelerators. The next step is the boundary exchange and OpenSHMEM is used to communicate between nodes as they exchange boundary values. The next step is the BT solver, where using OpenACC directives computation is offloaded to the accelerator. These two steps that consist of the boundary exchange and the solver, are looped over for a fixed number of iterations. After this a verification step is performed to

ensure that the solver produced the correct result. Conversion of the benchmark from using MPI+OpenMP to OpenSHMEM+OpenACC is a two step process where we first replace all MPI calls be equivalent OpenSHMEM calls to get an intermediate OpenSHMEM+OpenMP BT-MZ version. This is used as the starting point for OpenACC related modifications.

4.1 Inter process communication using OpenSHMEM

We use the OpenSHMEM communication library to communicate between processes. It effectively accomplishes the same role as Message Passing Interface (MPI) did in the reference implementation of the NPB-MZ hybrid parallel benchmark. The difference emanates from the fact that all communication in OpenSHMEM is *one-sided*, thus not requiring the participation of the target process. In the OpenSHMEM+OpenACC hybrid benchmark OpenSHMEM communicates data related to overlap regions of *zones*, and OpenACC parallelizes loops within each zone.

The BT-MZ benchmark has distinct communication and computation phases. There is no communication during the solving stage (*x-solve, y-solve and z-solve*). While porting the BT-MZ benchmark to use OpenSHMEM certain design choices have to be made which include decisions regarding the program variables that need to be *symmetric*, the choice of communication primitives (*put*

```
if (iodd == 0) {
MPI_Isend(&qbc_ou[qoffset],m_size,
          dp_type,ip,tag+myid,
          comm_setup,&requests[nr]);
MPI_Irecv(&qbc_in[qoffset],m_size,
          dp_type,ip,tag+ip,
          comm_setup,
          &requests[nr+1]);
}
else {
MPI_Irecv(&qbc_in[qoffset],m_size,
          dp_type, ip, tag+ip,
          comm_setup,
          &requests[nr]);
MPI_Isend(&qbc_ou[qoffset],m_size,
          dp_type, ip,tag+myid,
          comm_setup,
          &requests[nr+1]);
 }
```

Listing 1.1: MPI buffer exchange in reference BT-MZ *exch qbc* routine.

```
1  shmem_putmem(&dest_qoffset,
2               &qoffset,
3               sizeof(idx_t),ip);
4  shmem_fence();
5  shmem_long_put(&done, &x,1,ip);
6  shmem_quiet();
7  shmem_wait(&done, 0);
8  shmem_double_put(
9               &qbc_in[dest_qoffset],
10              &qbc_ou[qoffset],
11              m_size, ip);
12 shmem_quiet();
```

Listing 1.2: OpenSHMEM buffer exchange in *exch qbc* routine.

vs *get*) and synchronization points. Since OpenSHMEM does not have matching sends and receives (refer Listing 1.1) this exchange has to be ordered with an extra communication to indicate the correct offset location (Listing 1.2, line 1) and point to point synchronization (Listing 1.2, lines 4, 6, 7) that guarantees that the correct data has been communicated. Listing 1.2 shows how the exchange is effected using OpenSHMEM communication and synchronization calls. Moreover there is a significant benefit in using *put* as (unlike *get*) it returns as soon as the

buffer is available for reuse [5]. The final verification stage performs a reduction of solutions over all processes and computes residues from over all zones.

4.2 Targeting Hybrid Architectures with OpenACC directives

4.2.1 Introducing the OpenACC directives We started with the OpenSHMEM and OpenMP version of BT-MZ. We first find the OpenMP pragmas, which indicates code kernels that are already parallel and replaced them with OpenACC pragmas. We started by replacing OpenMP pragma (refer Listing 1.3) with OpenACC *#pragma acc kernels* around the main computational loops of *x-solve, y-solve, z-solve compute_rhs*, and *add* (refer Listing 1.4). This pragma indicates to the compiler that it should generate accelerated kernels for each loop nest. Only with this approach we do not expect to see performance gain since for every kernel the runtime will transfer the data back and forth to the accelerator and at this stage the kernels are not optimized at all. Unlike OpenMP where a subroutine called in a OpenMP parallel context is the same machine code as the host core, OpenACC regions are compiled to CUDA code and then compile by the Nvidia CUDA compiler for the GPU. This means that the compiler must either know that a function will be called in OpenACC or the function must be inlined. Because of this the subroutines lhs_init, matvec_sub, matmul_sub, binvcrhs and bincrhs have to be manually copied to the same source file so they be inlined by the OpenACC compiler. The new OpenACC 2.0 solves this problem using the routine directive. [5]

```
#pragma omp parallel for
for (k = 1; k <= nz-2; k++) {
  for (j = 1; j <= ny-2; j++) {
    for (i = 1; i <= nx-2; i++) {
      for (m = 0; m < 5; m++) {
        u(m,i,j,k) = u(m,i,j,k)
             + rhs(m,i,j,k);
      }
    }
  }
}
```

Listing 1.3: Original OpenMP pragmas in *add* routine.

```
#pragma acc kernels
for (k = 1; k <= nz-2; k++) {
  for (j = 1; j <= ny-2; j++) {
    for (i = 1; i <= nx-2; i++) {
      for (m = 0; m < 5; m++) {
        u(m,i,j,k) = u(m,i,j,k)
             + rhs(m,i,j,k);
      }
    }
  }
}
```

Listing 1.4: New OpenACC pragma *add* routine.

4.2.2 Split loop nests After inlining these subroutines the loops in the OpenACC regions were very large with high memory utilization. By experience, we know that splitting a huge loop nest into smaller ones will allow both OpenACC compiler, capsmc, and NVCC, to generate more optimized code for the GPU. This has advantages as it allows the compiler to find more parallelism to exploit, reduce register pressure and the device shared memory footprint. It does not negatively affect performance since maximum data is kept on the GPU between the different calls to the OpenACC kernels, prefetch to local caches and memories, which improves performance.

[5] This directive will be available in the CAPS OpenACC compiler later this year.

After splitting the solver loop nests into smaller ones, these loop nests are not anymore parallel. In order to enable the parallelization on the two outer loops, two additional dimensions were added to the fjac, njac, and lhs arrays (refer Listing 1.5 and Listing 1.6). We had to redeclare those arrays because the dimensions depend on the external loop levels. In the x-solve file (similarly for y-solve and z-solve), the external loop levels are based on the nz and ny sizes (respectively on, nz and nx, ny and nx). This enables the compiler to parallelize this loop nest by removing the dependencies between the different iterations of the outer loop accessing these arrays. While this puts additional pressure on the GPU memory, it allows us to reestablish the parallelism lost by splitting the loop nest.

```
for (k = 1; k <= nz-2; k++) {
  for (j = 1; j <= ny-2; j++) {
    for (i = 1; i <= nx-2; i++) {
...
      fjac[i][0][1] = -(u(1,i,j,k) * tmp2 *
          u(1,i,j,k))
          + c2 * qs(i,j,k);
      fjac[i][1][1] = ( 2.e0 - c2 )
          * ( u(1,i,j,k) / u(0,i,j,k) );
      fjac[i][2][1] = - c2 * ( u(2,i,j,k) * tmp1 );
      fjac[i][3][1] = - c2 * ( u(3,i,j,k) * tmp1 );
      fjac[i][4][1] = c2;
...
    }
  }
}
```

Listing 1.5: Original arrays in *x_solve* routine.

```
double fjacX[5][5][PROBLEM_SIZE+1][ny][nz];
#pragma acc kernels loop independent present(up[0:size5],rhsp[0:size5])
for (k = 1; k <= nz-2; k++) {
  for (j = 1; j <= ny-2; j++) {
    for (i = 1; i <= nx-2; i++) {
...
      fjacX[i][1][0][j][k] = -(u(1,i,j,k) * temp2 * u(1,i,j,k))
           + c2 * qs(i,j,k);
      fjacX[i][1][1][j][k] = ( 2.0 - c2 ) * ( u(1,i,j,k) / u(0,i,j,k) );
      fjacX[i][1][2][j][k] = - c2 * ( u(2,i,j,k) * temp1 );
      fjacX[i][1][3][j][k] = - c2 * ( u(3,i,j,k) * temp1 );
      fjacX[i][1][4][j][k] = c2;
...
    }
  }
}
```

Listing 1.6: Expanded arrays and OpenACC pragma *x_sovle* routine.

4.2.3 Reducing Data transfers To reduce the data transfer between the different OpenACC kernels we make the data reside on the GPU. We allocate the data on the GPU with the pragma *#pragma acc enter data create* for all the data at the beginning of the BT-MZ benchmark. The *#pragma acc data present* pragma is used to indicate to the kernels that the data already resides on the

accelerator when the kernels block arrives. The *#pragma acc update host* and *#pragma acc update device* pragmas are used to manually update the data on the host or device after the data on the other side is modified.
To allocate the memory for the different matrices, we used the OpenACC enter data directive from OpenACC 2.0 was already available in the CAPS OpenACC compiler. As shown in Listing 1.7, we allocate all the zones for each matrices. We cannot do only one allocation per matrix because the OpenACC's present table maps the addresses between the hosts and device pointers using the address of the first element. In our case, the matrix is a double linked vector and these addresses will in not available in the solver functions.

```
        u = (double *)shmalloc(sizeof(double)*PROC_MAX_SIZE5);
        for (iz = 0; iz < proc_num_zones; iz++) {
          zone = proc_zone_id[iz];
          size=nxmax[zone]*ny[zone]*nz[zone]*5;
          up=&u[start5[iz]];

#pragma acc enter data create(up[0:size], ...)

          initialize(&u[start5[iz]], ...);
...
        }
```

Listing 1.7: Allocates in *main* routine to create data on GPU.

In the solver functions, to indicate the data is already available on the device, we used the *#pragma acc data* directive with the present clause. This allows the runtime to know this data is already on the device and ready to be used. On this data directive we also specify to the runtime to allocate the locals lhs, fjac and njac buffer, if not already done, using the pcreate clause. The Listing 1.8 illustrates this.

```
#pragma acc data present(up[0:size5],...) pcreate(lhsX,fjacX,njacX)
{
  #pragma acc kernels loop independent
 for (i = 0; i <= isize; i++)
...
} //end data
```

Listing 1.8: Data clauses in *x_solve* routine.

Two operations compose the timestep loop, the exchange boundaries function call and the solver function calls. At every timestep, we need to update U matrix on the host with the values from the device before updating the boundary values. Then perform the OpenSHMEM communications to update the other PEs and afterwards, we update the U matrix from the host to the device. We use the *#pragma acc update device—host* directive around the OpenSHMEM calls

(Refer to Listing 1.9).

```
#pragma acc update host(u[0:size])
...
//OpenSHMEM communications
...
#pragma acc update device(u[0:size])
```

Listing 1.9: Update clauses in *exch_qbc* routine.

4.2.4 Improve kernels performance After improving the cumulative time taken for transfers by reducing their occurrences we focus on optimizing the kernels. There are many ways to improve the performance of these kernels. The optimization we applied are the following: increased the threads to execute the different kernels, take care of the coalescing, unrolled some of the loops and pre-accessing some data.
To improve the global performance of the kernels, a first step consist in indicating the compiler that it can parallelize on more loop levels. To do so, the use of the *#pragma acc loop independent* indicates that the user knows for sure this particular loop level is parallel and can be executed by multiple OpenACC threads. As a result, the kernel in itself is executed by more number of threads and each thread does less work.
A well-known performance issue on the Nvidia GPU is the non-contiguous global memory accesses, also known as un-coalesced accesses (refer to Listing 1.10). The goal of this optimization is to allow contiguous accelerator threads in the thread grid to work on contiguous data in the memory. This way the memory controller can reduce the number of memory loads and stores to the data. When a thread is accessing data in the GPU memory, the memory controller will load the memory segment that contains this particular data. So if contiguous threads are accessing contiguous data at the same time, the memory controller will load the needed memory segment for all this threads only once. To do so, we will ensure that the inner parallelized loop level corresponds to the most contiguous dimension of the main arrays of each loop nest (refer to Listing 1.11).

```
#pragma acc kernels loop independent present(up[0:size5],rhsp[0:size5])
for (k = 1; k <= nz-2; k++) {
  #pragma acc loop independent
  for (j = 1; j <= ny-2; j++) {
  #pragma acc loop independent
    for (i = 1; i <= nx-2; i++) {
...
      fjacX[1][0][i][j][k] = -(u(1,i,j,k) * temp2 * u(1,i,j,k))
           + c2 * qs(i,j,k);
      fjacX[1][1][i][j][k] = ( 2.0 - c2 ) * ( u(1,i,j,k) / u(0,i,j,k) );
      fjacX[1][2][i][j][k] = - c2 * ( u(2,i,j,k) * temp1 );
      fjacX[1][3][i][j][k] = - c2 * ( u(3,i,j,k) * temp1 );
      fjacX[1][4][i][j][k] = c2;
...
    }
  }
}
```

Listing 1.10: Un-coalesced array accesses in *x_solve* routine.

```
#pragma acc kernels loop independent present(up[0:size5],rhsp[0:size5])
for (i = 1; i <= nx-2; i++) {
  #pragma acc loop independent
  for (j = 1; j <= ny-2; j++) {
    #pragma acc loop independent
    for (k = 1; k <= nz-2; k++) {
...
      fjacX[1][0][i][j][k] = -(u(1,i,j,k) * temp2 * u(1,i,j,k))
           + c2 * qs(i,j,k);
      fjacX[1][1][i][j][k] = ( 2.0 - c2 ) * ( u(1,i,j,k) / u(0,i,j,k) );
      fjacX[1][2][i][j][k] = - c2 * ( u(2,i,j,k) * temp1 );
      fjacX[1][3][i][j][k] = - c2 * ( u(3,i,j,k) * temp1 );
      fjacX[1][4][i][j][k] = c2;
...
    }
  }
}
```

Listing 1.11: Coalesced array access in *x_sovle* routine.

Unrolling is a well-known technique to increase the global performance of kernels. It allows to increase the amount of work per thread and takes advantage of data reuse.
Finally, in order to help the CAPS OpenACC compiler get better performance, in some kernels, we pre-loaded some of the values in temporary variables (refer to Listing 1.12). At runtime, the data will be preloaded in a register which has a very low latency access latency (few memory cycles) compare to an access to gloabl memory (400-700 memory cycles). Otherwise, it accesses the same data in global memory multiple times resulting in higher latencies.

```
double tmprhs0, tmprhs1, tmprhs2, tmprhs3, tmprhs4;
  tmprhs0 = rhs(0,i-1,j,k);
  tmprhs1 = rhs(1,i-1,j,k);
  tmprhs2 = rhs(2,i-1,j,k);
  tmprhs3 = rhs(3,i-1,j,k);
  tmprhs4 = rhs(4,i-1,j,k);

  rhs(0,i,j,k) = rhs(0,i,j,k) - lhsX[0][0][AA][i][j][k]*tmprhs0
                    - lhsX[0][1][AA][i][j][k]*tmprhs1
                    - lhsX[0][2][AA][i][j][k]*tmprhs2
                    - lhsX[0][3][AA][i][j][k]*tmprhs3
                    - lhsX[0][4][AA][i][j][k]*tmprhs4;
```

Listing 1.12: Preloading temporary values *matvec_sub* routine.

5 Results

5.1 Platform

These tests were run on the Titan supercomputer, a Cray XK7 supercomputer [14]. Titan has 18,688 compute nodes equipped with 1 GPU per node. The nodes are connected with Cray's Gemini interconnect. For OpenSHMEM Titan has installed Cray's shmem implementation version 5.6.3. For OpenACC Titan has capsmc version 3.3.4. For this paper we used a beta version of the CAPS compiler,

version 3.3beta-r50937. Additionally the GNU compiler collection version 4.7.1, nvidia CUDA compiler version 5.5 and CUDA driver 5.0 were used.

5.2 Timing and Scalability

In the following figure you can see the different speedup. the Figure 1 shows the speed ups of the different configuration compared to the fully serial version 1 PE on 1 node of the class C. the Figure 2 shows the speed ups of the different configuration compared to the fully serial version 16 PEs on 16 nodes of the class D.

We focused on the execution of the OpenSHMEM version and the OpenSHMEM-OpenACC version of the BT-MZ benchmark. We choose to compare the class C and D. The class C is composed of 256 zones with zone sizes distributed from 13x8x28 to 57x38x28 elements and executes 200 iterations. The class D is composed of 1024 zones with zone sizes distributed from 22x16x34 to 98x73x34 elements and executes 250 iterations. We first run these tests against a serial version on a single node, then compare the speed up of using 8, 16, 32, 64, 128, and 256 nodes. We also compare the speed of using OpenACC as well, showing what benefits or deficits that are incurred for using GPU acceleration. We do this in a seperate graph for a class C run and a class D run.

In Figure 1 we can see that the OpenACC struggles to match the speedup from the pure OpenSHMEM version at first. Then after 64 nodes the performance of the pure OpenSHMEM version plateaus while the GPU version continues to see gains. The most likely explanation for this is that for class C after 64 nodes, OpenSHMEM is no longer able to extract additional parallelism. This may be a result of the simplistic nature of the port from MPI to OpenSHMEM. It may be possible to get additional performance gains by restructuring the communication patterns. OpenACC continues to see performance gains because distributing more zones across more PEs with more GPUs allows for fewer transfers of zones across the PCIe bus increasing the efficiency of the GPUs.

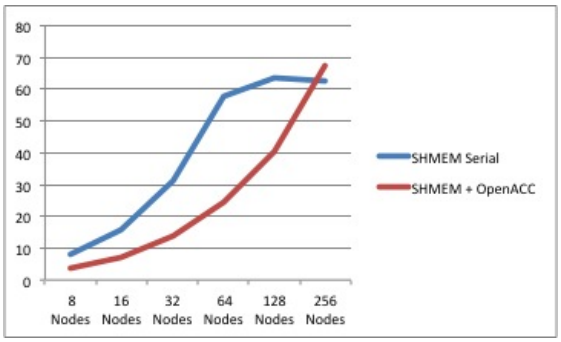

Fig. 1: Speed-up for the Class C of BT-MZ

In Figure 2 we can see that OpenACC and OpenSHMEM+OpenACC both continue to see the same amount of performance gain as we add more PEs. However, unlike in Class C where OpenSHMEM saw an early lead before dropping off in gains, both see the same amount of performance gain for the additional PEs. However, in this run we also see the version with OpenACC has a consistent performance advantage of 10%.

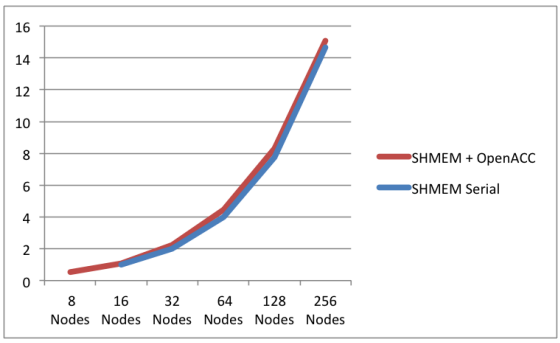

Fig. 2: Speed-up for the Class D pf BT-MZ

On the figure 3 we can see the percentage of the time spent in the exch_boundary function call for the OpenSHMEM-OpenACC version of the code. This function call is also where the updates to GPU memory occur, so in addition to the time spent in network communication it also encompasses the time spent transferring data between the GPU and the system memory. For the OpenSHMEM serial execution we observed up to 3% of time spent in exch_qbc for 256 PE. Concerning the OpenSHMEM-OpenACC version the time spent in exch_qbc goes up to 47%.

We also compared the performance of the Serial-C version [15] and the OpenACC version of the BT benchmark (non-MZ) for the class B and C. The B class computes on a matrix of 102x102x102 elements for 200 iterations. The C class computes on a matrix of 162x162x162 elements for 200 iterations. In the table 1, you can see the time of execution of the different class on 1 node of Titan. Here we can see that increasing the size of the data to compute on the accelerator allows us to get some good speedup compare to the serial execution. It uses the same algorithm with the same code implementation for the BT solver. This isolates the performance of OpenACC versus the performance of OpenSHMEM for this algorithm. We do this to demonstrate the performance gains of using a GPU implementation without OpenSHMEM communication.

6 Conclusions and Future Work

While the non multi-zone version of the BT benchmark showed great performance benefit, the Multizone version struggled to match serial performance.

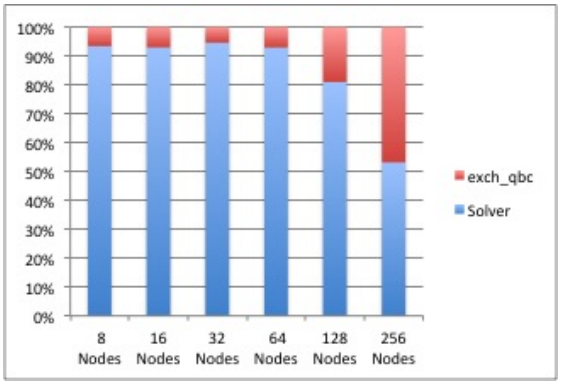

Fig. 3: Proportion of time spent in exch_qbc and the solver for the Class D Of BT-MZ

Class	B	C
serial	427.53	1575.67
ACC	99.71	460.74
speed-up	4.29	3.42

Table 1: Execution time of the NAS BT benchmark for Class B & C

This was largely because, as seen in the time spent exchanging data, because as the zones got smaller the benefits extracted from the GPU became smaller, since the transfer times overwhelmed the computational benefits. Using larger problem sizes should also result in larger gains in speed, since we can see increases in the ratio of transfer versus computation as the number of nodes was increased for the same problem size.

In our OpenSHMEM and OpenACC hybrid most of the performance problems came from the need to communicate memory stored on the GPU across the network. Network performance was good but since the GPUs had to send their data to main memory to communicate over the network the performance gains of having accelerators was hard to realize in a distributed environment.

Using an accelerator that uses host memory should eliminate these problems. The AMD APUs would not suffer from the performance degradation associated with transferring memory from the accelerator to the host system and thus should not have problems associated with transferring small zones to and from the independent accelerator memory. This would solve a large part of the growth seen in figure 3 since the boundary exchange includes memory transfer from the accelerator.

We can see in Table 1 that OpenACC saw a solid performance gain when run in serial with one zone, so it is safe to say that we are not seeing performance degradation because of the execution on the accelerator. In figure 3 we can see that the memory exchange, including updating system memory and transfers across the network, quickly grew in the dominance of the total run time. In

figure 1 we can see that the BT-MZ benchmark without OpenACC is initially faster, and with the evidence in table 1 and figure 3 it is reasonable to guess that most of this slowdown is the accelerator transfer. Because of this we believe it is reasonable to project that an integrated solution like an AMD APU will see performance closer to what we saw in figure 2.

Further work can include increasing the problem size, allowing the GPU to do more compute work for the zones between transfers. The ability to target the benchmark to fill the memory size of the GPU would expose the maximum benefits of the GPUs. It would also be interesting to see the results of exceeding the maximum size of the GPU memory to see how OpenACC would cope or suggest to future OpenACC specifications to support the swapping of memory from the GPU to the host.

One interesting avenue to explore for this purpose would be using the accelerator as the primary source of memory, as opposed to the system memory. In the current implementation, the benchmark still has to transfer memory to the system memory in order to communicate barrier conditions with OpenSHMEM. If something similar to the GPU direct with CUDA and MPI could be implemented for OpenSHMEM there would be no need to transfer this memory to the system memory. In fact, the host system could be made unnecessary in an extreme case. Further exploration of how OpenSHMEM and OpenACC can be utilized together represents a large challenge that also holds promise for excellent performance. The main hurdle to this remains an awareness of the GPU and how it works with memory transfers and it's impact on communication.

7 Acknowledgments

This work is supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at the Oak Ridge National Laboratory.

References

1. Top500: Top 500 supercomputer sites. http://www.top500.org/ (2013)
2. Bland, B.: Titan - early experience with the titan system at oak ridge national laboratory. In: Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. SCC '12, IEEE Computer Society (2012) 2189–2211
3. Poole, S., Hernandez, O., Kuehn, J., Shipman, G., Curtis, A., Feind, K.: Openshmem - toward a unified rma model. In Padua, D., ed.: Encyclopedia of Parallel Computing. Springer US (2011) 1379–1391
4. Jin, H., der Wijngaart, R.F.V.: Performance characteristics of the multi-zone nas parallel benchmarks. In: IPDPS, IEEE Computer Society (2004)
5. Org., O.: Openshmem specification (2011)
6. Gokhale, M., Stone, J.: Napa c: compiling for a hybrid risc/fpga architecture. In: FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on. (1998) 126 –135

7. Fraguela, B.B., Renau, J., Feautrier, P., Padua, D., Torrellas, J.: Programming the flexram parallel intelligent memory system. SIGPLAN Not. **38** (2003) 49–60
8. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Cellss: a programming model for the cell be architecture. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. SC '06, New York, NY, USA, ACM (2006)
9. OpenHMPP: OpenHMPP: Concepts & Directives (2012)
10. Han, T.D., Abdelrahman, T.S.: 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, New York, NY, USA, ACM (2009) 52–61
11. Koesterke, L., Boisseau, J., Cazes, J., Milfeld, K., Stanzione, D.: Early Experiences with the Intel Many Integrated Cores Accelerated Computing Technology. In: Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery. TG '11, New York, NY, USA, ACM (2011) 21:1–21:8
12. OpenACC: How does the openacc api relate to openmp api? (2013)
13. NVIDIA: OpenACC Directives for Accelerators. In: NVIDIA Developer Zone. (2012) `http://developer.download.nvidia.com/CUDA/training/OpenACC_1_0_intro_jan2012.pdf`.
14. Facility, O.R.L.C.: Introducing titan: Advancing the era of accelerated computing. http://www.olcf.ornl.gov/titan/ (2013)
15. Center for Manycore Programming, Seoul National University, K.: Snu npb suite site (2013)