Make Legacy Fortran Code Fly on GPUs in 5 Days or Less

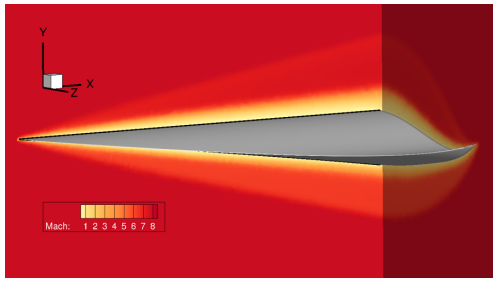
Alice Koniges* and High Flyers Team

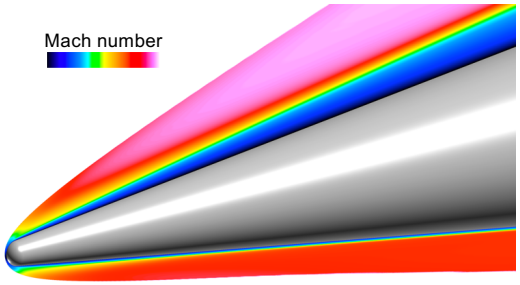
***University of Hawaii, koniges@hawaii.edu**

Summary of GPU Hackathon Experience (July 15-19, 2019) Oakland, CA

and follow-on optimizations

(Co-sponsored by NERSC and OLCF)





High Flyers Team and other involved people

- Alice Koniges (Team Leader): University of Hawaii/MHPCC
- Joel Bretheim: Formerly Naval Research Laboratory, Now PET
- David Eder: University of Hawaii/MHPCC
- Gabriele Jost: NASA
- Christos Kavouklis: Lawrence Livermore National Laboratory
- Ioannis Nompelis: University of Minnesota
- Chris Stone: Formerly PETTT, Now Computational-Science

- Chris Daley (Mentor): NERSC/Berkeley Lab
- Fazlay Rabbi (Mentor/Graduate Student): NERSC/Berkeley Lab
- Angela Chen: NVIDIA
- Max Katz: NVIDIA
- Kevin Gott (Hackathon Organizer): NERSC/Berkeley Lab

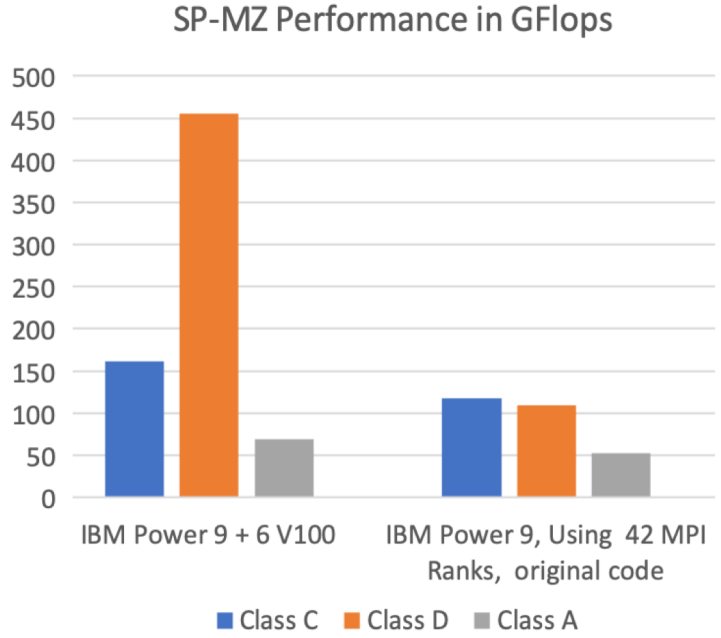
High Flyers Team members and mentors

Summary of hackathon experience

- Studied Multi-zone NAS Parallel Benchmark SP-MZ written in FORTRAN with OpenMP
- SP-MZ is a mini-application that mimics the performance of CFD applications
- SP-MZ uses diagonalized Beam-Warming alternating direction implicit (ADI) algorithm
- Explicit right-hand-side (RHS) vector with finite difference and the inversion of the scalar pentadiagonal (SP) matrices along each grid line in all three directional sweeps
- Modifying the OpenMP directives, we were able to offload the computations to GPUs relatively quickly (first day)
- Majority of time spent was on performance optimization
- Worked on IBM system at Oak Ridge and a Cray system at Berkeley both with V100s
- We discovered differences in the Cray and IBM compliers
- Performance of GPUs relative to CPUs is a strong function of the zonal dimensions

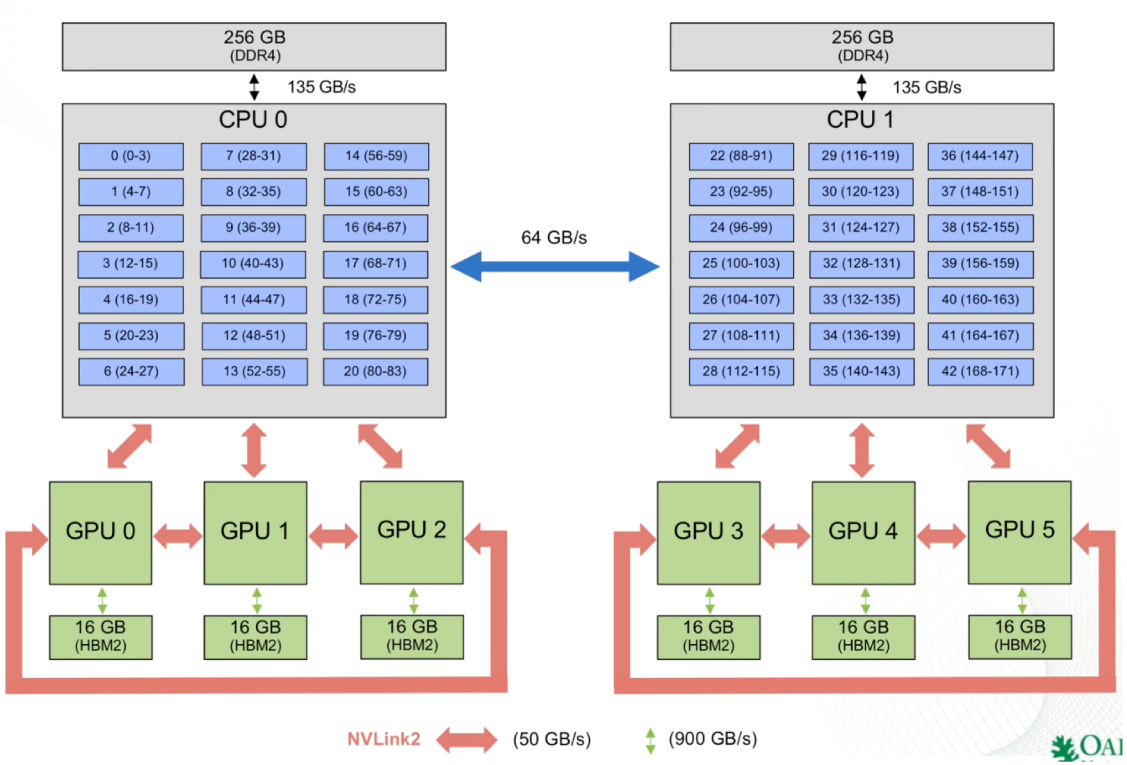
Results Summary: SP-MZ at Hackathon

- **Benchmark description**
  - SP-MZ - Scalar Penta-diagonal solver with even-size zones within a problem class
  - Employs distributed memory parallelism via MPI for inter-zone communication and shared multi-core parallelization within the zones
  - Open source code with parallel patterns representative of many CFD and Aerospace codes
  - FORTRAN Source with CPU-enabled OpenMP => mimics many existing applications relevant to DoDHPC Mod programs



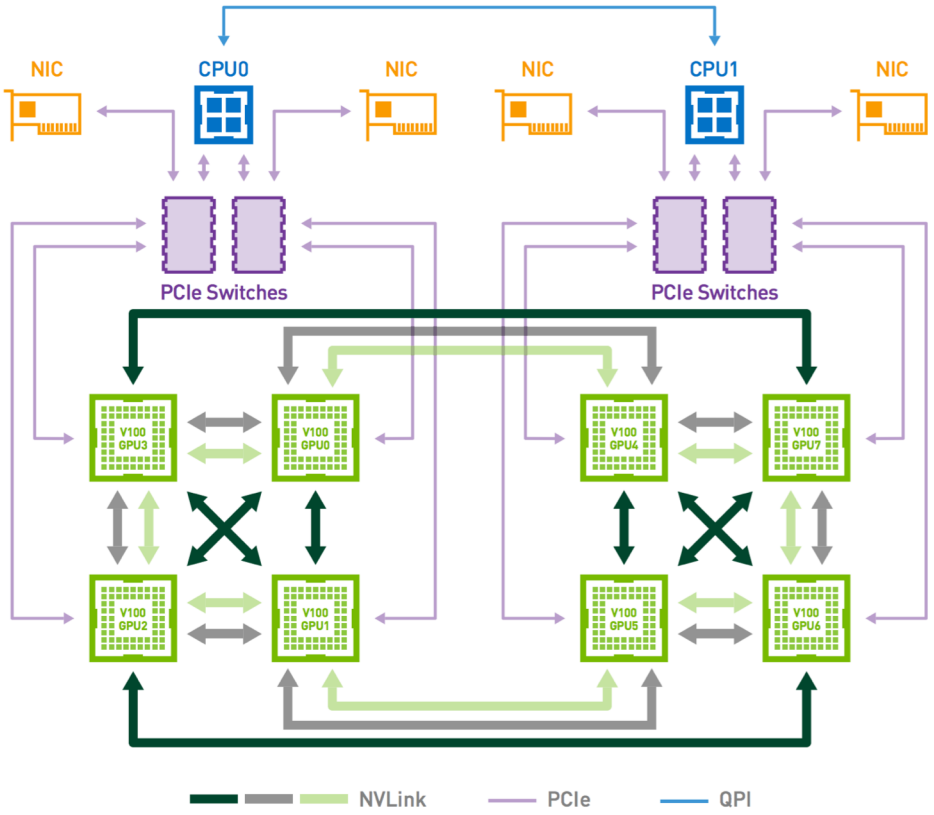
- **From CPU-only OpenMP to OpenMP 4.5**
  - Decorate existing OpenMP directives with off-load extensions
    - Accelerator kernels within the zone
  - Re-organize some data structures to allow stride 1 memory access
  - Avoid using arrays private to a thread and rather use shared arrays with higher dimensionality
  - Use pre-allocated scratch space rather than dynamic allocation
  - Exploit limited implicit zone-level parallelism by enabling asynchronous kernel execution via OpenMP tasking

IBM system at Oak Ridge (Ascent/Summit)



- **Each node contains two IBM POWER9 processors and six NVIDIA V100 GPUs.**
- **Each POWER9 processor is connected via dual NVLINK bricks, each capable of a 25GB/s transfer rate in each direction.**
- **The POWER9 processor is built around IBM's SIMD Multi-Core (SMC).**
- **The processor provides 22 SMCs with separate 32kB L1 data and instruction caches.**
- **Pairs of SMCs share a 512kB L2 cache and a 10MB L3 cache.**

Cray system at Berkeley (Cori-GPU)



- Each node contains 8 NVIDIA V100 GPUs connected to each other in a 'hybrid cube-mesh' topology.
- Each GPU is connected directly to 4 others with NVLINK.
- All GPUs are connected to the Skylake CPUs and the Infiniband network interface cards (NICs) via PCIe 3.0.

SP-MZ supports host-side distributed and shared-memory parallelism with MPI and OpenMP

- Version 3.4 has 3,515 total LOC ignoring comment lines and blank lines.
- Counting only LOC associated with the implicit integration (i.e., the core CFD solver), the mini-app has only 1,809 LOC.
- The small size of the mini-app allows an end-to-end refactoring and several iterations on offloading strategies.
- For a mesh with N^3 points, the ADI scheme requires that N^2 SP line matrices (with N rows) be solved in each of the sweeps.
- Fast assembly and inversion of the SP matrices is key to achieving high performance in the mini-app and the real CFD applications.

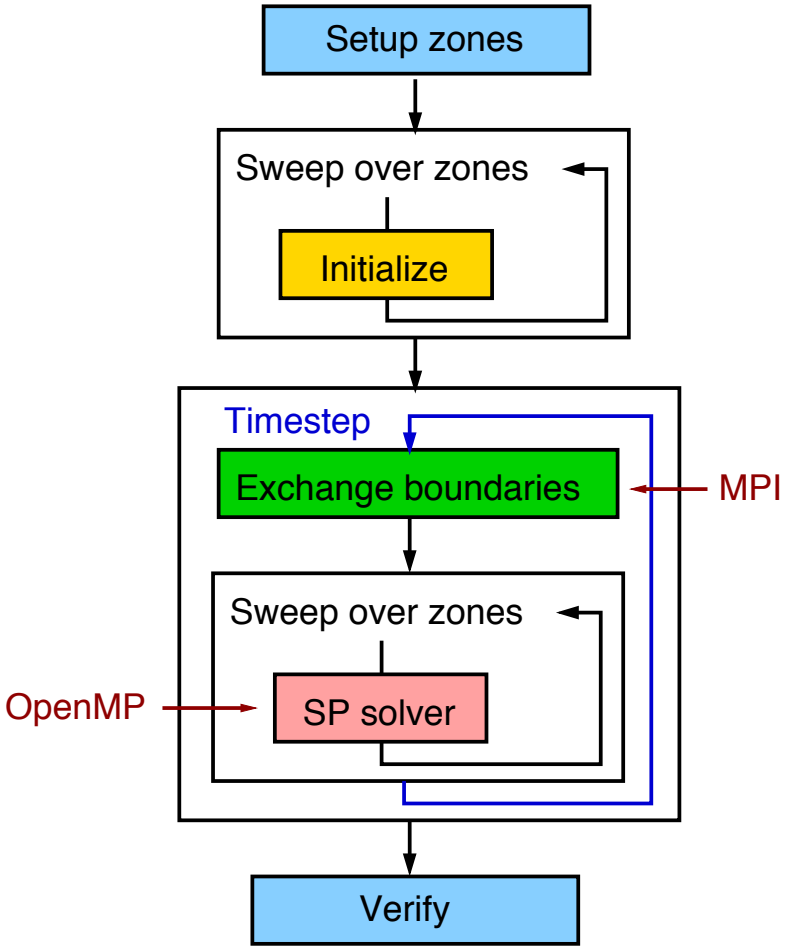
The week long hackathon had a total of 11 teams

- Each team gave daily updates, which allowed sharing of lessons learned
- Each team had a private Slack channel to communicate and upload results, and a group Slack channel addressed global concerns
- Our team used git and bitbucket to document code changes
- Also used Google drive to share documentation, PDFs, PPTs, etc.
- Each team was assigned one or two mentors and there were additional computer scientists from vendors that provided tool instruction and optimization techniques

Background on OpenMP

- Compiler directives, runtime library routines, and environment variables
- OpenMP 5.0 released at SC18
- Accelerator (GPU) programming support since OpenMP 4.0 : Identify kernels for offload, specify parallelism and manage data transfer between host and device
- GPU code builds on existing OpenMP code, makes strides towards portability
- FORTRAN OpenMP supported by many compilers such as gfortran, Cray ftn and IBM xlf. Full 5.0 support in progress
- OpenMP also takes a vendor neutral approach as the standard is written by a consortium of HPC compiler teams, universities and research laboratories

Execution flow of the SP-MZ Benchmark



CPU version (pink) – GPU version (green)

```
        implicit none

        integer nx, nxmax, ny, nz
-       double precision rhs(5,0:nxmax-1,0:ny-1,0:nz-1),
-     $                  u  (5,0:nxmax-1,0:ny-1,0:nz-1)
+       double precision rhs(0:nxmax-1,0:ny-1,0:nz-1,5),
+     $                  u  (0:nxmax-1,0:ny-1,0:nz-1,5)

        integer i,j,k,m

-!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(m,i,j,k)
-!$OMP&  SCHEDULE(STATIC) COLLAPSE(2)
+!$OMP TARGET TEAMS DISTRIBUTE
+!$OMP&    PRIVATE( m,i,j,k )
        do k = 1, nz-2
           do j = 1, ny-2
+!$OMP        PARALLEL DO PRIVATE(i,m)
              do i = 1, nx-2
                 do m = 1, 5
-                   u(m,i,j,k) = u(m,i,j,k) + rhs(m,i,j,k)
+                   u(i,j,k,m) = u(i,j,k,m) + rhs(i,j,k,m)
                 end do
              end do
+!$OMP        END PARALLEL DO
           end do
        end do
-!$OMP END PARALLEL DO
+!$OMP END TARGET TEAMS DISTRIBUTE

        return
```

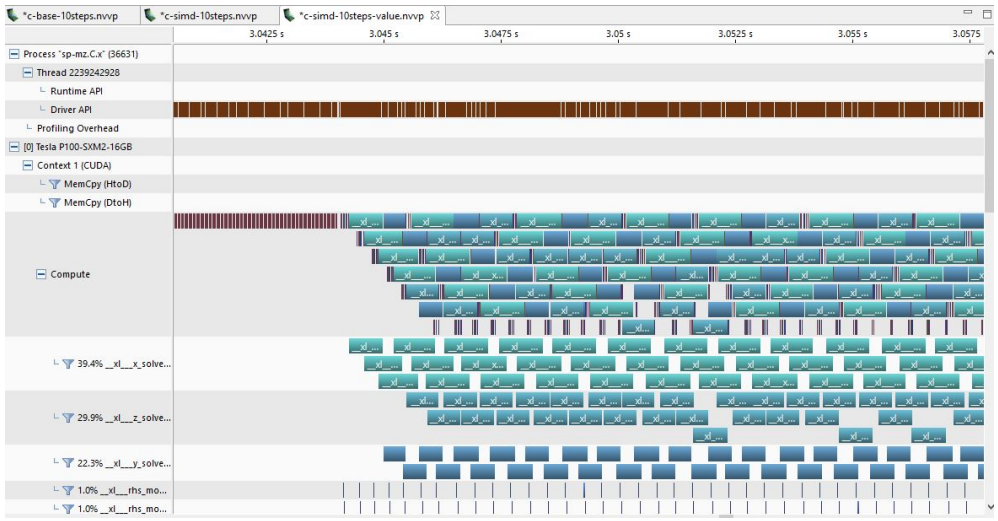
Implementation optimized for CPUs

```
!$OMP PARALLEL DO DEFAULT(SHARED)
!$OMP&  PRIVATE(fac2,m,fac1,i2,i1,ru1,i,j,k)
!$OMP&  SCHEDULE(STATIC) COLLAPSE(2)
 do k = 1, nz-2
   do j = 1, ny-2
     call lhsinit(lhs, lhsp, lhsm, nx-1)
     ....   !--- operations localized at "i"
     do i = 0, nx-3    !--- Thomas alg. forward elim.
       i1 = i + 1
       i2 = i + 2
       fac1      = 1.d0/lhs(3,i)
       lhs(4,i)  = fac1*lhs(4,i)
       lhs(5,i)  = fac1*lhs(5,i)
       do m = 1, 3
         rhs(m,i,j,k) = fac1*rhs(m,i,j,k)
       end do
       lhs(3,i1) = lhs(3,i1) -
                   lhs(2,i1)*lhs(4,i)
       lhs(4,i1) = lhs(4,i1) -
                   lhs(2,i1)*lhs(5,i)
       do m = 1, 3
         rhs(m,i1,j,k) = rhs(m,i1,j,k) -
                   lhs(2,i1)*rhs(m,i,j,k)
       end do
       ....
       lhsm(4,i1) = lhsm(4,i1) -
                    lhsm(2,i1)*lhsm(5,i)
       ....
     end do
     ....
   end do
 end do
```

Implementation optimized for GPUs

```
!$OMP  TARGET TEAMS DISTRIBUTE
!$OMP&  PARALLEL DO SIMD COLLAPSE(2)
!$OMP&  NOWAIT DEPEND( inout: rhs )
!$OMP&    MAP( ALLOC: lhs4, lhsp4, lhsm4, rhst )
!$OMP&    PRIVATE(fac2,m,fac1,i2,i1,ru1,i)
!$OMP&    PRIVATE( cv_im1, cv_ip1 )
!$OMP&    PRIVATE( rhon_ip1, rhon_im1, rhon_i )
 do k = 1, nz-2
   do j = 1, ny-2
     lhs4 (j,1:5,   0,k) = 0.0d0
     lhs4 (j,1:5,nx-1,k) = 0.0d0
     ....   !--- operations localized at "i"
     do i = 0, nx-3    !--- Thomas alg. forward elim.
       i1 = i  + 1
       i2 = i  + 2
       fac1         = 1.d0/lhs4(j,3,i,k)
       lhs4(j,4,i,k) = fac1*lhs4(j,4,i,k)
       lhs4(j,5,i,k) = fac1*lhs4(j,5,i,k)
       do m = 1, 3
         rhst(j,m,i,k) = fac1*rhst(j,m,i,k)
       end do
       lhs4(j,3,i1,k) = lhs4(j,3,i1,k) -
                  lhs4(j,2,i1,k)*lhs4(j,4,i,k)
       ....
     end do
     ....
   end do
 end do
```

Tools such as nvprof used for kernel execution timelines



Compilation using the Cray compiler

`ftn -openmp`

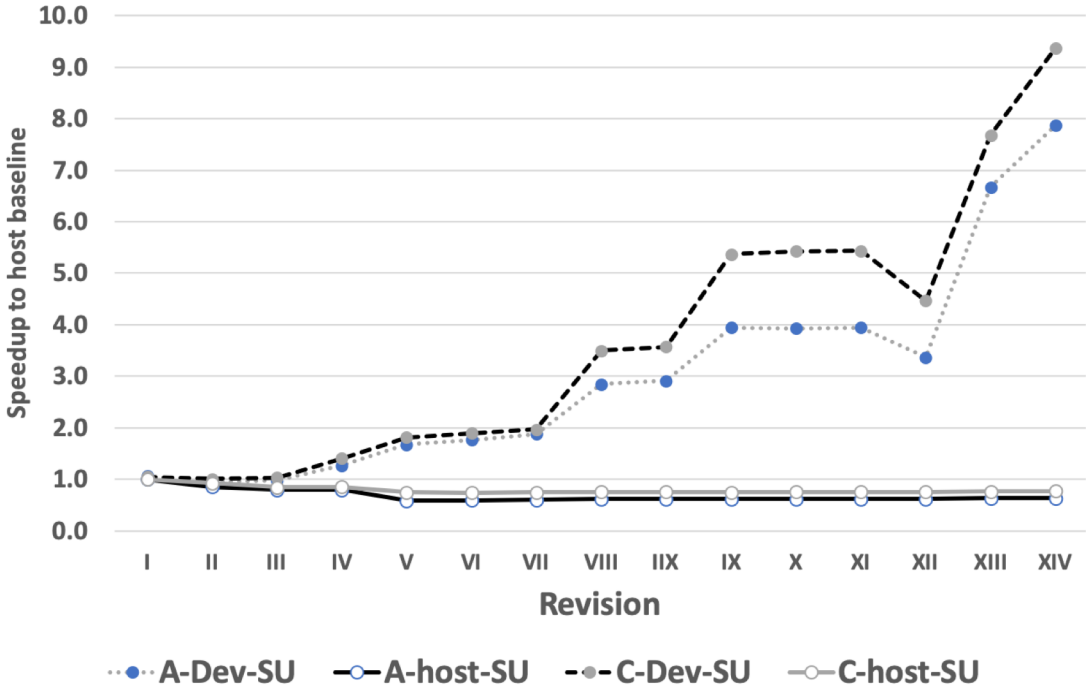
Compilation using the IBM compiler

`xlf -qsmp -qoffload`

- **IBM compiler introduced unnecessary memory copies of constants during offloading**
- **Cray compiler did not work with MPI implementation on Cori-GPU system**
- **We reported discovered problems with FORTRAN using OpenMP on CPU/GPU systems**

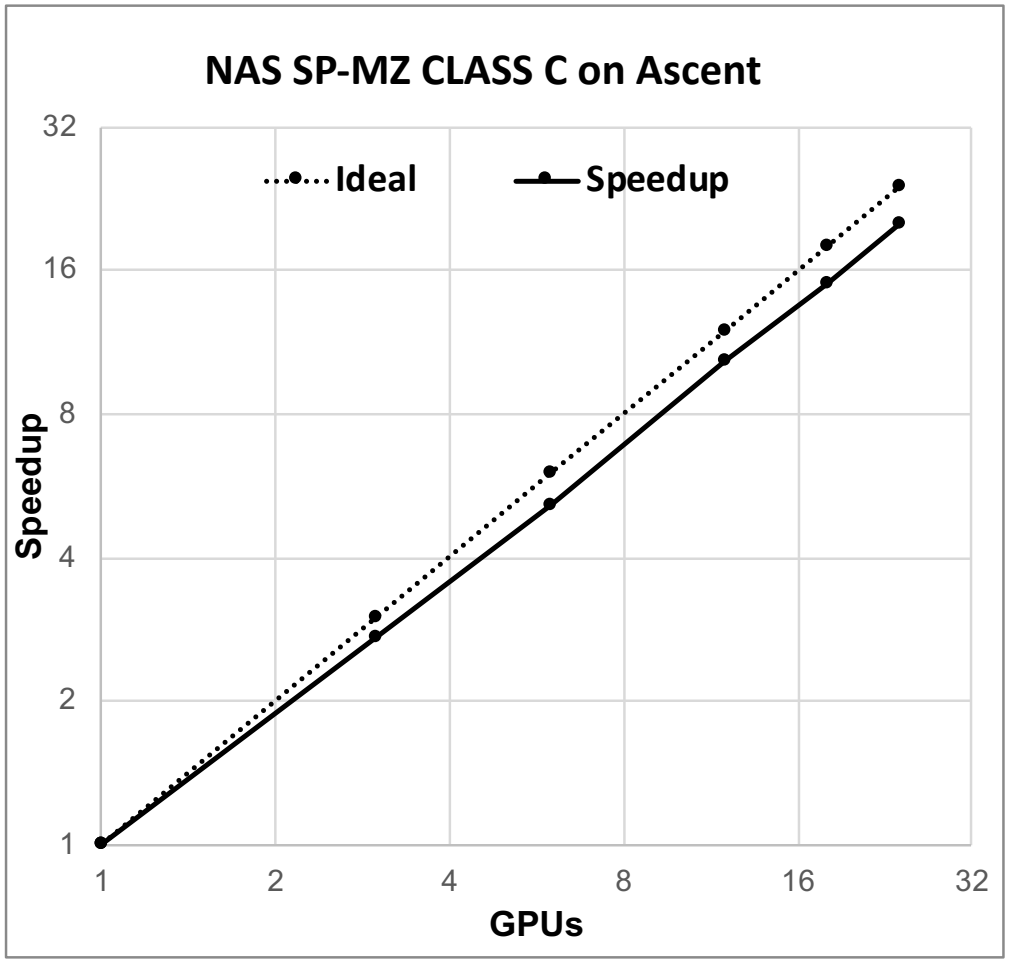
I. Baseline
II. Transposed arrays to support coalesced memory operations; replaced 1D with 4D scratch arrays.
III. Changed index ordering in x-solve to improve memory coalescing.
IV. Reduced or eliminated unnecessary host-to-device transfers.
V. Merged target, teams distribute, and parallel do directives. into one combined directive and additionally used loop collapsing.
VI. Enabled kernel task dependencies to allow asynchronous execution.
VII. Improved efficiency of ghost/rind data buffer fill kernels.
VIII. Enabled asynchronous rind fill kernels.
X. Added loop scalars to private clauses to avoid unnecessary host-to-device transfers before kernel execution.
X. Added CUDA-aware MPI option to bypass host transfers during MPI communication.
XI. Improved parallelism and occupancy of rind fill in/out routines.

Execution Incremental Performance on the Ascent (IBM) System



XII. Added SIMD to the PARALLEL DO ~ directive to improve performance with the implementation on the CRAY
XIII. Change argument passing from by-reference to by-value to allow the IBM OpenMP v4.5 compiler to avoid unnecessary host-to-device transfers.
XIV. Combined all rind/ghost cell fill in/out routines to execute in one kernel instead of one per zone. That is, changed parallelism from within each zone to all zones at once.

We also studied parallel scaling performance across nodes



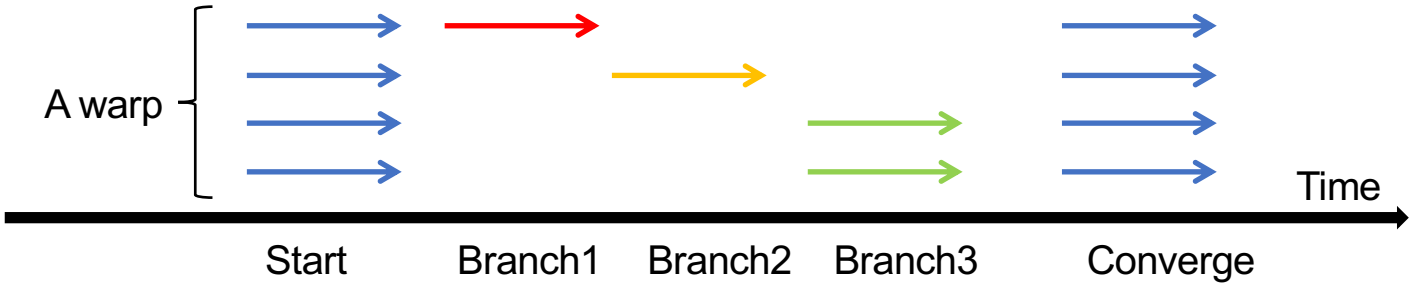
Optimizations include:

- Changes to the memory layout of the arrays
- Addition of extra array dimensions to eliminate need for threadprivate data
- Reordering of array dimensions
- Loop collapsing
- Replacing unnecessary OpenMP firstprivate clauses with private, in order to avoid unnecessary HtD data transfers
- Using OpenMP nowait to enable asynchronous execution to allow for overlap of kernel execution

Future Directions: Portability and GPU Optimization

- Compilers will accept the combined construct
  - **#pragma omp target teams distribute parallel for**
    - This *does not* generalize to all algorithms unfortunately, but the majority can be adapted.
    - The construct makes a lot of guarantees to the compiler and it is very easy to reason about for good performance.
- GPU optimization: understanding SIMD behavior (same for all languages)
  - Individual work-items of a warp start together at the same program address
  - Each work-item has its own instruction address counter and register state
    - Each work-item is free to branch and execute independently
    - Supports the SPMD pattern.
  - Branch behavior
    - Each branch will be executed serially
    - Work-items not following the current branch will be disabled

Adapted From SC19 Tutorial by Mattson, Stotzer, McIntosh-Smith : Programming Your GPU with OpenMP



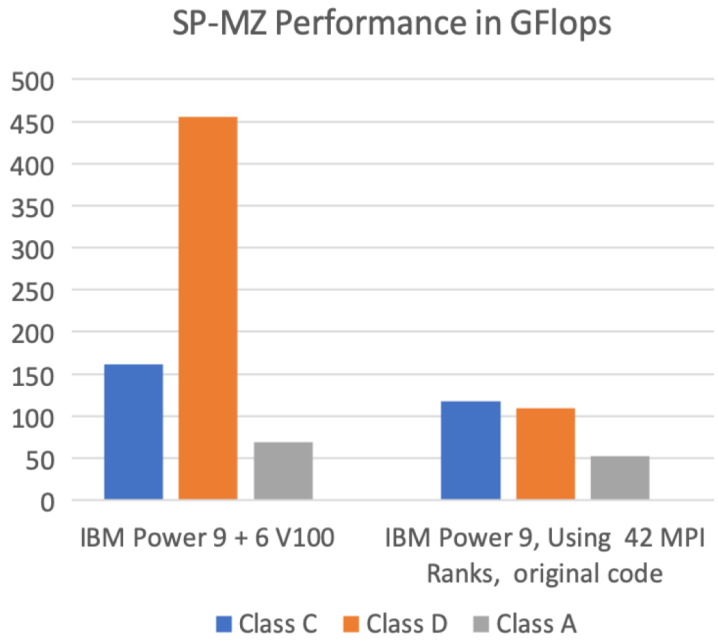
Code transformations of our initial implementation

- We manually inlined routines called within OpenMP target regions.
- We transposed some of the arrays to allow for stride one memory access to enables simultaneous loads and stores.
- We used the OpenMP COLLAPSE clause to collapse as many loops as possible.
- We found that OpenMP PRIVATE arrays per thread gave poor performance because of the large size of the arrays.
- Therefore, we made the array shared. This was accomplished by adding two extra dimensions.
- To exploit some of the available zone-level parallelism within the code, we enabled asynchronous kernel execution using deferred OpenMP target tasks with dependencies.
- This is why the directive in the code listing contains the NOWAIT and DEPEND clauses.

Further Optimizations

- We eliminated some small host-to-device data transfers by using the DECLARE TARGET construct
- Instead of a kernel for each rind copy in/out in “exch-bc” for each zone, we created a target region over the zone loop so all the copy in/out operations run within the same kernel
- We pre-allocated a large temporary array for all the lhs structures on all zones and passed them to the x, y and z solver routines
- We merged asynchronous kernels with communication optimizations
- We also combined the PARALLEL DO directives with TARGET TEAMS DISTRIBUTE since this enables the compiler to generate simpler code where all GPU threads execute the same computation on different data.

Hack-a-Thon Take-aways:



- **OpenMP target off-load is an excellent programming API for porting legacy FORTRAN codes to the GPU**
- **Changes to increase GPU performance sometimes results in slightly worse CPU performance, more comparisons required**
- **Exploring newly developed larger size classes more appropriate for real applications and modern architectures that should perform well on GPUs**
- **OpenMP 5.0 features allow for additional optimization on the GPUs. Compilers and tools are still immature for FORTRAN (many DoD applications)**
- **The hackathon experience provides for several person months of work in a condensed setting if experts are involved.**
  - **Post-hackathon work to make developed code “usable” and well-documented is time consuming**