Targeting Accelerator using OpenMP with GenASiS: A Simple and Effective Fortran Experience

Reuben Budiardja, Oak Ridge National Laboratory
The Application

**General Astrophysics Simulation System (GenASiS)**

- Designed for parallel, large scale simulations
  - weak-scale to ~100 thousands MPI processes
- Written entirely in modern Fortran (2003, 2008)
- Modular, object-oriented design, and extensible
- Multi-physics solvers:
  - (Magneto)-hydrodynamics (HLL, HLLC solvers)
  - Explicit 2nd order time-integration
  - Self-gravity, polytropic & nuclear EoS
  - Grey and spectral neutrino transport
- CPU only code with OpenMP for threading (prior to this work)
The Application

- Studied the role fluid instabilities --- convection and Standing Accretion Shock Instability (SASI) --- in supernova dynamics
- Discovered exponential magnetic field amplification by SASI in progenitor star → origin of neutron star magnetic fields
- Refactored to three major subdivisions: Basics, Mathematics, Physics → allowing unit testing, ad-hoc/standalone tests, mini-apps
Paths to Targeting Accelerators

• “Native” accelerator programming model (e.g. CUDA, HIP, … )
  – requires rewrite of all computational kernels
  – loss of Fortran semantics (multi-d arrays, pointer/array remapping)
  – requires interfacing with the rest of the (Fortran) code

• Extensions to Fortran (e.g. CUDA Fortran)
  – non standard extension to Fortran only supported by few compilers non-uniformly
  – cannot easily fall back to standard Fortran for host-only code

• OpenMP offload
  – standardized directives
  – retain Fortran semantics
  – increasing support by more compilers (with existing implementation on our current target platform, OLCF Summit)
Higher-level GenASiS Functionality

• StorageForm :
  – a class for data and metadata; the ‘heart’ of data storage facility in GenASiS
  – metadata includes units, variable names (for I/O, visualization)
  – used to group together a set of related physical variables (e.g. Fluid)
  – render more generic and simplified code for I/O, ghost exchange, prolongation & restriction (AMR mesh)

• Data :
  – StorageForm % Value ( nCells, nVariables )
  – use as, e.g. Pressure => StorageForm % Value ( :, 1 ), Density => StorageForm % Value ( :, 2 )

• Methods:
  – call S % Initialize ( ) ← allocate data on host
  – call S % AllocateDevice ( ) ← allocate and associate data on GPU
  – call S % Update{Device,Host} ( ) ← transfer data
Higher-level GenASiS Functionality (2)

- call StorageForm % Initialize &
  ( Shape = [6, 6], &
  VariableOption = [“Pressure”, “Density”, &
  “Energy”, ... ] )
real ( KDR ), dimension ( :, : ), pointer :: Scratch

type ( c_ptr ) :: D_Value

call AllocateDevice ( S % nValues * S % nVariables, D_Value )
call c_f_pointer ( D_Value, Scratch, [ S % nValues, S % nVariables ] )
do iV = 1, S % nVariables
    D_Value = c_loc ( Scratch ( :, iV ) )
    Variable => S % Value ( :, iV )
call AssociateHost ( D_Value, Variable )
end do
Lower-Level GenASiS Functionality

- Fortran wrappers to OpenMP APIs
  - call AllocateDevice(Value, D_Value) → omp_target_alloc()
  - call AssociateHost(D_Value, Value) → omp_target_associate_ptr()
  - call UpdateDevice(Value, D_Value), call UpdateHost(Value, D_Value) → omp_target_memcpy()

Value : Fortran array
D_Value : type(c_ptr), GPU address
Offloading Computational Kernel

```
1 subroutine AddKernel ( A, B, C )
2
3    real ( KDR ), dimension ( : ), intent ( in ) :: A, B
4    real ( KDR ), dimension ( : ), intent ( out ) :: C
5
6    integer ( KDI ) :: i
7
8    !$OMP target teams distribute parallel do schedule ( static, 1 )
9    do i = 1, size ( C )
10       C ( i ) = A ( i ) + B ( i )
11    end do
12    !$OMP end target teams distribute parallel do
13
14 end subroutine AddKernel
```

No implicit data transfer, no explicit map()
Example of Kernel with Pointer Remapping

real ( KDR ), dimension ( :, :, : ), pointer :: V, dV

V ( -1:nX+2, -1:nY+2, -1:nZ+2 ) => F % Value ( : , iV )
dV ( -1:nX+2 , -1:nY+2 , -1:nZ+2 ) => dF % Value ( : , iV )
call ComputeDifferences_X ( V, dV )
Example of Kernel with Pointer Remapping

```fortran
1 subroutine ComputeDifference_X ( V, dV )
2
3 real ( KDR ), dimension ( -1:, -1:, -1: ), &
4    intent ( in ) :: &
5    V
6 real ( KDR ), dimension ( -1:, -1:, -1: ), &
7    intent ( out ) :: &
8    dV
9
10 integer ( KDI ) :: i, j, k
11
!$OMP target teams distribute parallel do collapse (3) schedule (static, 1)
12 do k = 1, nZ
13    do j = 1, nY
14        do i = 0, nX + 2
15            dV ( i, j, k ) &
16                = V ( i, j, k ) &
17                - V ( i - 1, j, k )
18        end do
19    end do
20 end do
21 !$OMP end target teams distribute parallel do
22
23 end subroutine ComputeDifferences_X
```
Porting a Fluid Dynamics Application: RiemannProblem

Initial (left) and final (right) density of 1D and 3D RiemannProblem
Fluid Evolution on CPU

1: Call: Initialize ( )
2: Call: GhostExchange ( )
3: Set: Time = StartTime
4: while Time < FinishTime do
5:  Call: ComputeTimeStep ( ) \rightarrow TimeStep
6:  Set: FluidOld = FluidCurrent
7:
8:  Call: ComputeDifferences ( )
9:  Call: ComputeReconstruction ( )
10: Call: ComputeFluxes ( )
11: Call: ComputeUpdate ( TimeStep ) \rightarrow FluidUpdate
12: Set: FluidCurrent = FluidOld + FluidUpdate
13:
14: Call: GhostExchange ( )
15:
16: Call: ComputeDifferences ( )
17: Call: ComputeReconstruction ( )
18: Call: ComputeFluxes ( )
19: Call: ComputeUpdate ( TimeStep )
20: Set: FluidCurrent = 0.5 * (FluidOld + FluidCurrent + FluidUpdate)
21:
22: Call: GhostExchange ( )
23: end while
Fluid Evolution on GPU

1: **Host**: Call: Initialize ( )
2: **Call**: GhostExchange ( )
3: **Host**: Set: Time = StartTime
4: while Time < FinishTime do
5:    **Host**: Call: ComputeTimeStep ( ) \rightarrow TimeStep
6:    **TRANSFER**: Call: FluidCurrent \% UpdateDevice ( )
7:    **DEVICE**: Set: FluidOld = FluidCurrent
8:    
9:    **DEVICE**: Call: ComputeDifferences ( )
10:   **DEVICE**: Call: ComputeReconstruction ( )
11:   **DEVICE**: Call: ComputeFluxes ( )
12:   **DEVICE**: Call: ComputeUpdate ( TimeStep ) \rightarrow FluidUpdate
13:   **DEVICE**: Set: FluidCurrent = FluidOld + FluidUpdate
14:   
15:    **TRANSFER**: Call: FluidCurrent \% UpdateHost ( )
16:   **Host**: Call: GhostExchange ( )
17:    **TRANSFER**: Call: FluidCurrent \% UpdateDevice ( )
18:    
19:    **DEVICE**: Call: ComputeDifferences ( )
20:    **DEVICE**: Call: ComputeReconstruction ( )
21:    **DEVICE**: Call: ComputeFluxes ( )
22:    **DEVICE**: Call: ComputeUpdate ( TimeStep )
23:    **DEVICE**: Set: FluidCurrent = 0.5 * (FluidOld + FluidCurrent + FluidUpdate)
24:     
25:    **TRANSFER**: Call: FluidCurrent \% UpdateHost ( )
26:    **Call**: GhostExchange ( )
27: **end while**
Performance Results

**Summit Node**

(2) IBM Power9 + (6) NVIDIA Volta V100

---

**Performance Results**

"Proportional resource tests":

7 CPU cores vs. 1 GPU

6 RS per host,
1 MPI (+OpenMP) per RS

jsrun -r6 -c7 -g1 -a1 -bpacked:7
Results: Weak-Scaling 3D Riemann Problem

~12-15X speedup from 7 CPU threads to GPU
Results: GenASiS Basics RiemannProblem

Kernel timings for 50 cycles, 3D - 256$^3$ cells per GPU (lower is better/faster)

Timings for OpenMP on CPU with 7-threads are scaled down by 25X to fit this plot.
Performance Results: Kernel Speedups

![Bar chart showing kernel speedups.](chart.png)
Conclusion

• OpenMP provides a simple and effective path to port Fortran code to run on accelerators
  – more compilers are supporting OpenMP offload (XL, GCC, CCE, Intel, PGI, LLVM-based)

• Performance parity between OpenMP offload and vendor-specific accelerator programming model (e.g. CUDA) is achievable
  – OpenMP is more portable and can be more “natural” to the application
  – compilers need to do a good job of optimization
  – no need to rewrite kernels, simpler to port from multi-threading to GPU offload
  – can continue to exploit base language feature (Fortran)

• Code and paper: github.com/GenASiS arxiv.org/abs/1812.07977
openmp.org  OpenMP API specs, forum, reference guides, and more

link.openmp.org/sc20  Videos and PDFs of OpenMP SC’20 presentations