Tuning MPAS-A on SPR-HBM

Vamsi Sripathi, Andrey Ovsyannikov, Ruchira Sasanka

6/27/2023

intel

MPAS

- Model for Prediction Across Scales (MPAS) is a collaborative project for developing atmosphere, ocean and other earth-system simulation components for use in climate, regional climate and weather studies.
- Primary development partners are LANL (Los Alamos National Laboratory) and NCAR (National Center for Atmospheric Research)

- In this work, the "atmosphere" component (MPAS-A) version 7.3 with 120km resolution problem set was benchmarked on SPR-HBM
- SPR-HBM (B2 stepping) configured as SNC4, 1LM/HBM-only (ortce-sprh4)
- Using IFORT + Pure MPI (112 MPI ranks pinned to 112-cores (56c/socket))

https://mpas-dev.github.io/

MPAS
Model for Prediction Across Scales

MPAS Home

Overview
MPAS-Atmosphere
MPAS-Albany Land Ice
MPAS-Ocean
MPAS-Seaice
Data Assimilation
Publications
Presentations

Download
MPAS-Atmosphere download
MPAS-Albany Land Ice download
MPAS-Ocean download
MPAS-Seaice download

Resources
License Information
Wiki
Bug Tracker
Mailing Lists
MPAS Developers Guide
MPAS Mesh Specification Document

MPAS Atmosphere

Overview

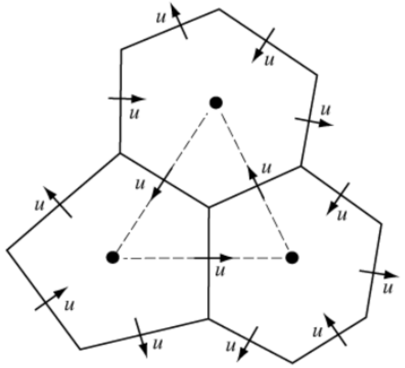
The atmospheric component of MPAS, as with all MPAS components, uses an unstructured centroidal Voronoi mesh (grid, or tessellation) and C-grid staggering of the state variables as the basis for the horizontal discretization in the fluid-flow solver. The unstructured variable resolution meshes can be generated having smoothly-varying mesh transitions (see the figure to the right); we believe that this capability will ameliorate many issues associated with the traditional mesh refinement strategy of one-way and two-way grid nesting where the transitions are abrupt. Using the flexibility of the MPAS meshes, we are working towards applications in high-resolution numerical weather prediction (NWP) and regional climate, in addition to global uniform-resolution NWP and climate applications.


A variable resolution MPAS Voronoi mesh

The MPAS atmosphere consists of an atmospheric fluid-flow solver (the *dynamical core*) and a subset of the Advanced Research WRF (ARW) model atmospheric physics. Work is underway to port the MPAS atmospheric dynamical core to the Community Atmosphere Model (CAM) in the Community Earth Systems Model (CESM), which will provide coupling between MPAS Ocean and MPAS Atmosphere and coupling to the CAM physics and other components of the CESM system. Work is also progressing on porting the National Centers for Environmental Prediction (NCEP) Global Forecast System (GFS) atmospheric physics to MPAS.

Dynamical Core

The MPAS atmospheric dynamical core solves the fully compressible nonhydrostatic equations of motion. The horizontal Voronoi mesh, depicted to the right, uses a C-grid staggering of the state variables; the horizontal velocity *u* is defined as the normal velocity on Voronoi cell faces while the other state variables are defined at the cell centers. The dual of the Voronoi mesh is the triangular mesh shown in dashed lines in the figure. The variable resolution meshes a predominantly comprised of hexagons, but pentagons and septagons are occasionally present. The primary advances associated with the C-grid-staggered Voronoi mesh can be found in Thuburn et al JCP (2009) and Ringler et al JCP (2010).



A description of the compressible nonhydrostatic atmospheric solver can be found in Skamarock et al MWR (2012). The fully compressible nonhydrostatic equations are cast in terms of a geometric-height vertical coordinate, and the solver makes use of a split-explicit time integration scheme that is described in Klemp et al MWR (2007). The time-integration scheme employs a 3rd-order Runge-Kutta method, and large time step, for the meteorologically significant modes and a forward-backward method with smaller time steps for the acoustic modes (See Wicker and Skamarock MWR 2002). The numerical schemes used in the atmospheric component of MPAS are very similar to those employed in the Advanced Research WRF model

C-grid staggered variables on the horizontal Voronoi mesh. Normal velocities are defined on the cell faces and all other scalar variables are defined at the cell centers. Vertical

Executive Summary

- Software optimizations deliver speed-ups of
  - 1.23x on SPR-HBM
  - 1.14x on SPR+DDR5 (2S Xeon 8480+, 56c/socket)
  - 1.15x on ICX+DDR4 (2S 8360Y, 36c/socket)
- HW + Optimized SW speed-ups
  - SPR + HBM / SPR + DDR5 = 1.9x (baseline: 1.75x)
  - SPR + HBM / ICX + DDR4 = 3.28x (baseline: 3.06x)
  - SPR + DDR5 / ICX + DDR4 = 1.73x (baseline: 1.75x)
- Next steps - contact MPAS developers and discuss plans for upstreaming the code changes

Problem size	Performance on SPR-HBM (time, lower is better)				Speedup	
	Baseline		Tuned			
	IFORT	IFX	IFORT	IFX	IFORT	IFX
120km	39.03	44.73	31.3	37.47	1.25	1.19
60km	296.89	345.5	254.48	301.19	1.17	1.15
30km	1241.9	1399.78	1063.61	1228.73	1.17	1.14

MPAS-A Hotlist

	timer_name	total	calls	min	max	avg	pct_tot	pct_par	par_eff
1	total time	43.90016	1	43.89877	43.90016	43.89951	100.00	0.00	1.00
2	initialize	5.43465	1	5.43401	5.43465	5.43433	12.38	12.38	1.00
2	time_integration	36.49409	360	0.09566	0.10758	0.10099	83.13	83.13	1.00
3	atm_rk_integration_setup	0.30344	360	0.00061	0.00141	0.00078	0.69	0.83	0.93
3	atm_compute_moist_coefficients	0.10724	360	0.00022	0.00056	0.00028	0.24	0.29	0.95
3	physics_get_tend	0.11091	360	0.00021	0.00100	0.00028	0.25	0.30	0.89
3	atm_compute_vert_imp_coefs	0.49004	1080	0.00030	0.00222	0.00040	1.12	1.34	0.89
3	atm_compute_dyn_tend	9.85093	3240	0.00167	0.00612	0.00279	22.44	26.99	0.92
3	small_step_prep	1.32537	3240	0.00032	0.00102	0.00038	3.02	3.63	0.92
3	atm_advance_acoustic_step	3.42694	4320	0.00047	0.00151	0.00067	7.81	9.39	0.85
3	atm_divergence_damping_3d	1.09264	4320	0.00016	0.00095	0.00022	2.49	2.99	0.87
3	atm_recover_large_step_variables	4.80361	3240	0.00100	0.00222	0.00139	10.94	13.16	0.94
3	atm_compute_solve_diagnostics	4.86276	3240	0.00094	0.00239	0.00130	11.08	13.32	0.87
3	atm_rk_dynamics_substep_finish	1.15744	1080	0.00041	0.00182	0.00090	2.64	3.17	0.84
3	atm_advance_scalars	1.85324	720	0.00181	0.00415	0.00243	4.22	5.08	0.94
3	atm_advance_scalars_mono	0.74803	360	0.00149	0.00353	0.00186	1.70	2.05	0.89

- MPAS has support for native profiling framework, already has hooks around hot compute functions
  - But does not include communication functions, the #2 most time-consuming component
  - You cannot tune what you don’t measure – so, added timers around halo-exchange routines
- Overall, this is memory bandwidth bound application (VTune shows about 450 GB/s on 1-socket SPR-HBM)
  - Mostly read-heavy traffic with several address streams

Tuning Experiments Prolog

- Source code changes
  - Compute functions: All the hot functions are in one single Fortran source file and one single module (7000 loc)
  - Makes it hard to experiment with Compiler options, reading optimization reports and make code changes
  - For easier prototyping, split each targeted hot function to individual .F file and use FPP for conditional compilation
  - This also enables to fully rewrite the Fortran source in “C” Compiler intrinsics for performance experiments
- Faster compilation
  - A full build of MPAS-A takes 20 mins
  - Compiling of modified source components leads to build-time of less than 2 mins
  - Enables faster code<->run feedback loop

MPAS-A Results

SPR-HBM

Function Name	Baseline	Optimized	Speed-up
atm_compute_dyn_tend	9.80	9.00	1.09
halo_comms	6.65	4.35	1.53
atm_compute_solve_diagnostics	4.92	4.01	1.23
atm_recover_large_step_variables	4.85	4.11	1.18
atm_advance_acoustic_step	3.38	3.17	1.07
atm_advance_scalars	1.87	0.57	3.30
small_step_prep	1.32	1.19	1.11
atm_rk_dynamics_substep_finish	1.17	0.69	1.68
atm_divergence_damping_3d	1.09	0.95	1.15
atm_advance_scalars_mono	0.76	0.70	1.09
atm_compute_vert_imp_coefs	0.51	0.53	0.95
atm_rk_integration_setup	0.31	0.39	0.81
physics_get_tend	0.11	0.14	0.80
atm_compute_moist_coefficients	0.11	0.10	1.02
Total (time_integration)	36.83	29.90	1.23

Function	Extent of modifications
(green)	No source code changes. All gains from Compiler flags
(yellow)	Moderate source code changes. Compiler Pragmas, assists to compiler to generate better code
(red)	Heavy source code changes. Fusing loops to facilitate non-temporal stores, AVX512 intrinsics

- Optimizations deliver speed-up of 1.23x on SPR-HBM

Halo_comms

- MPAS-A needs to exchange data among MPI ranks between computations
- The data arrays are 2 or 3D double precision values
- The data needs to be packed/unpacked from a n-D array to 1-D array before/after the communication
- The exchange is done using MPI Isend/Irecv pairs
- MPI Derived Datatype
  - Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.
  - MPI_Type_hvector() - Creates a vector (strided) datatype with offset in bytes
  - Does anyone have positive experience with usage of MPI Derived data-types over explicit pack/unpack mechanism?

```

      type (field2DReal), pointer :: theta_m_field
      type (field3DReal), pointer :: scalars_field
      type (field2DReal), pointer :: pressure_p_field
      type (field2DReal), pointer :: rtheta_p_field
      type (field2DReal), pointer :: rtheta_pp_field
      type (field2DReal), pointer :: tend_u_field
      type (field2DReal), pointer :: u_field
      type (field2DReal), pointer :: w_field
      type (field2DReal), pointer :: rw_p_field
      type (field2DReal), pointer :: ru_p_field
      type (field2DReal), pointer :: rho_pp_field
      type (field2DReal), pointer :: pv_edge_field
      type (field2DReal), pointer :: rho_edge_field
      type (field2DReal), pointer :: exner_field

```

```
            ! w
            call mpas_pool_get_subpool(domain % blocklist % structs, 'state', state)
            call mpas_pool_get_field(state, 'w', w_field, 2)
            call mpas_dmpar_exch_halo_field(w_field)

            ! pv_edge
            call mpas_pool_get_subpool(domain % blocklist % structs, 'diag', diag)
            call mpas_pool_get_field(diag, 'pv_edge', pv_edge_field)
            call mpas_dmpar_exch_halo_field(pv_edge_field)

            ! rho_edge
            call mpas_pool_get_field(diag, 'rho_edge', rho_edge_field)
            call mpas_dmpar_exch_halo_field(rho_edge_field)

            ! scalars
            if (config_scalar_advection .and. (.not. config_split_dynamics_transport) ) then
               call mpas_pool_get_field(state, 'scalars', scalars_field, 2)
               call mpas_dmpar_exch_halo_field(scalars_field)
            end if
```

Halo_comms: Packing/Unpacking

Pack + Isend()

```fortran
        commListPtr => sendList
        do while(associated(commListPtr))
          allocate(commListPtr % rbuffer(commListPtr % nList))
          nullify(commListPtr % ibuffer)
          bufferOffset = 0
          do iHalo = 1, nHaloLayers
            nAdded = 0
            fieldCursor => field
            do while(associated(fieldCursor))
              exchListPtr => fieldCursor % sendList % halos(haloLayers(iHalo)) % exchList
              do while(associated(exchListPtr))
                if(exchListPtr % endPointID == commListPtr % procID) then
                  do  i = 1, exchListPtr % nList
#if defined (AWE_DMPAR)
                    src_idx = exchListPtr % srcList(i)
                    dst_idx = (exchListPtr % destList(i)-1) * fieldCursor % dimSizes(1) + bufferOffset
!DIR$ vector always nontemporal(commListPtr%rbuffer)
!DIR$ IVDEP
                    do j = 1, fieldCursor % dimSizes(1)
                      commListPtr % rbuffer(dst_idx + j) = fieldCursor % array(j, src_idx)
                      nAdded = nAdded + 1
                    end do
#else
                    do j = 1, fieldCursor % dimSizes(1)
                      commListPtr % rbuffer((exchListPtr % destList(i)-1) * fieldCursor % dimSizes(1) + j + bufferOffset) = &
                      fieldCursor % array(j, exchListPtr % srcList(i))
                      nAdded = nAdded + 1
                    end do
#endif
                  end do
                end if

                exchListPtr => exchListPtr % next
              end do

              fieldCursor => fieldCursor % next
            end do
            bufferOffset = bufferOffset + nAdded
          end do

          call MPI_Isend(commListPtr % rbuffer, commListPtr % nList, MPI_REALKIND, commListPtr % procID, &
                         dminfo % my_proc_id, dminfo % comm, commListPtr % reqID, mpi_ierr)
          commListPtr => commListPtr % next
```

Tuned: the `#if defined (AWE_DMPAR)` branch. Baseline: the `#else` branch.

Unpack + Irecv()

```fortran
        commListPtr => recvList
        do while(associated(commListPtr))
          call MPI_Wait(commListPtr % reqID, MPI_STATUS_IGNORE, mpi_ierr)
          bufferOffset = 0
          do iHalo = 1, nHaloLayers
            nAdded = 0
            fieldCursor => field
            do while(associated(fieldCursor))
              exchListPtr => fieldCursor % recvList % halos(haloLayers(iHalo)) % exchList
              do while(associated(exchListPtr))
                if(exchListPtr % endPointID == commListPtr % procID) then
#if defined (AWE_DMPAR)
#if 0

                  call awe_unpack(exchListPtr % nList, fieldCursor % dimSizes(1), bufferOffset, &
                                  exchListPtr % destList, exchListPtr % srcList, &
                                  fieldCursor % array, commListPtr % rbuffer, dminfo % my_proc_id)
#else
                  do i = 1, exchListPtr % nList
!DIR$ vector always nontemporal(fieldCursor%array)
!DIR$ IVDEP
                    do j = 1, fieldCursor % dimSizes(1)
                      fieldCursor % array(j, exchListPtr % destList(i)) = &
                      commListPtr % rbuffer((exchListPtr % srcList(i)-1)*fieldCursor % dimSizeS(1) + j + bufferOffset)
                    end do
                  end do
#endif
#else
                  do i = 1, exchListPtr % nList
                    do j = 1, fieldCursor % dimSizes(1)
                      fieldCursor % array(j, exchListPtr % destList(i)) = &
                      commListPtr % rbuffer((exchListPtr % srcList(i)-1)*fieldCursor % dimSizeS(1) + j + bufferOffset)
                    end do
                  end do
#endif
                  nAdded = max(nAdded, maxval(exchListPtr % srcList) * fieldCursor % dimSizes(1))
                end if
                exchListPtr => exchListPtr % next
              end do

              fieldCursor => fieldCursor % next
            end do
            bufferOffset = bufferOffset + nAdded
          end do
          commListPtr => commListPtr % next
        end do
```

Tuned: the `#else` branch of `#if 0` (inside `#if defined (AWE_DMPAR)`). Baseline: the outer `#else` branch.

- Use IVDEP to tell the Compiler that there are no loop carried dependencies
- Use non-temporal stores through Pragmas (to avoid adverse impact on other parts of code)
- Having a complex ptr calculation in store address trips-off the Compiler, nudge it by storing the constant part of the store address outside the loop

- Applied to both 2D and 3D routines
- 1.53x speed-up for this code-block on SPR-HBM

atm_compute_solve_diagnostics

- The baseline version of this routine is about 200 loc containing 12 loop blocks
- In the tuned version, 12 loops are collapsed into 4 blocks to facilitate the generation of non-temporal stores and to minimize redundant loads
- The 4 loop blocks are packaged into C functions and either use C + Compiler Pragams or C + AVX512 Compiler intrinsics
- As part of perf. tuning experiments, the 4 loop blocks were written in C. These tunings can be incorporated back into Fortran subroutines (through Compiler Pragams) for easier acceptance into upstream
- These deliver a speed-up of 1.23x

```fortran
      ke_fact = 1.0 - .375

      reconstruct_v = .true.
      reconstruct_v_int = 1
      if(present(rk_step)) then
        if(rk_step /= 3) then
          reconstruct_v = .false.
          reconstruct_v_int = 0
        end if
      end if

      call atm_compute_solve_diagnostics_awe_l1(nVertLevels, edgeStart, edgeEnd, &
                                                cellsOnEdge, dcEdge, dvEdge, &
                                                h_edge, h, &
                                                ke_edge, u)

      call atm_compute_solve_diagnostics_awe_l2(nVertLevels, vertexStart, vertexEnd, &
                                                edgesonVertex, edgesonVertex_sign, dcEdge, &
                                                invAreaTriangle, &
                                                vorticity, u, &
                                                pv_vertex, fVertex, &
                                                ke_vertex, ke_edge)

      call atm_compute_solve_diagnostics_awe_l3(nVertLevels, cellStart, cellEnd, nEdgesOnCell, &
                                                edgesOnCell, edgesOnCell_sign, dvEdge, &
                                                divergence, u, &
                                                ke, ke_edge, &
                                                invAreaCell, ke_fact, verticesOnCell, &
                                                kiteForCell, ke_vertex, kiteAreasOnVertex, &
                                                pv_cell, pv_vertex, maxEdges)

      call atm_compute_solve_diagnostics_awe_l4(reconstruct_v_int, config_apvm_upwinding, &
                                                nVertLevels, edgeStart, edgeEnd, &
                                                nEdgesOnEdge, edgesOnEdge, v, weightsOnEdge, u, &
                                                dt, invDvEdge, invDcEdge, &
                                                pv_edge, pv_vertex, verticesOnEdge, &
                                                gradPVt, gradPVn, pv_cell, cellsOnEdge, &
                                                maxEdges2)
```

atm_compute_solve_diagnostics: sample loop blocks

Loop Block-1

```c
#if defined (USE_INTRINSICS)
    for (k=0; k<(nVertLevels/8)*8; k+=8) {
      zmm_0 = _mm512_loadu_pd(&p_h[(cell11*nVertLevels) + k]);
      zmm_1 = _mm512_loadu_pd(&p_h[(cell12*nVertLevels) + k]);
      zmm_0 = _mm512_add_pd(zmm_0, zmm_1);
      zmm_0 = _mm512_mul_pd(zmm_sf, zmm_0);
      _MM512_STORE(&p_h_edge[(iEdge*nVertLevels) + k], zmm_0);

      zmm_2 = _mm512_loadu_pd(&p_h[(cell21*nVertLevels) + k]);
      zmm_3 = _mm512_loadu_pd(&p_h[(cell22*nVertLevels) + k]);
      zmm_2 = _mm512_add_pd(zmm_2, zmm_3);
      zmm_2 = _mm512_mul_pd(zmm_sf, zmm_2);
      _MM512_STORE(&p_h_edge[((iEdge+1)*nVertLevels) + k], zmm_2);

#if UNROLL > 2
      zmm_0 = _mm512_loadu_pd(&p_h[(cell31*nVertLevels) + k]);
      zmm_1 = _mm512_loadu_pd(&p_h[(cell32*nVertLevels) + k]);
      zmm_0 = _mm512_add_pd(zmm_0, zmm_1);
      zmm_0 = _mm512_mul_pd(zmm_sf, zmm_0);
      _MM512_STORE(&p_h_edge[((iEdge+2)*nVertLevels) + k], zmm_0);

      zmm_2 = _mm512_loadu_pd(&p_h[(cell41*nVertLevels) + k]);
      zmm_3 = _mm512_loadu_pd(&p_h[(cell42*nVertLevels) + k]);
      zmm_2 = _mm512_add_pd(zmm_2, zmm_3);
      zmm_2 = _mm512_mul_pd(zmm_sf, zmm_2);
      _MM512_STORE(&p_h_edge[((iEdge+3)*nVertLevels) + k], zmm_2);
#endif
    }

    zmm_efac1 = _mm512_set1_pd(efac1);
    zmm_efac2 = _mm512_set1_pd(efac2);
#if UNROLL > 2
    zmm_efac3 = _mm512_set1_pd(efac3);
    zmm_efac4 = _mm512_set1_pd(efac4);
#endif

#pragma nofusion
    for (k=0; k<(nVertLevels/8)*8; k+=8) {
      zmm_0 = _mm512_loadu_pd(&p_u[(iEdge*nVertLevels) + k]);
      zmm_0 = _mm512_mul_pd(zmm_0, zmm_0);
      zmm_0 = _mm512_mul_pd(zmm_efac1, zmm_0);
      _MM512_STORE(&p_ke_edge[(iEdge*nVertLevels) + k], zmm_0);
```

Loop Block-2

```c
#if defined (USE_INTRINSICS)
    zmm_s11 = _mm512_set1_pd(s11);
    zmm_s12 = _mm512_set1_pd(s12);
    zmm_s13 = _mm512_set1_pd(s13);
    zmm_r11 = _mm512_set1_pd(r11);
    zmm_r12 = _mm512_set1_pd(r12);

    zmm_s21 = _mm512_set1_pd(s21);
    zmm_s22 = _mm512_set1_pd(s22);
    zmm_s23 = _mm512_set1_pd(s23);
    zmm_r21 = _mm512_set1_pd(r21);
    zmm_r22 = _mm512_set1_pd(r22);

    zmm_fVertex_1 = _mm512_set1_pd(p_fVertex[iVertex]);
    zmm_fVertex_2 = _mm512_set1_pd(p_fVertex[iVertex+1]);

#pragma nofusion
    for (k=0; k<(nVertLevels/8)*8; k+=8) {
      zmm_0 = _mm512_loadu_pd(&p_u[(iEdge11*nVertLevels) + k]);
      zmm_0 = _mm512_mul_pd(zmm_s11, zmm_0);
      zmm_1 = _mm512_loadu_pd(&p_u[(iEdge12*nVertLevels) + k]);
      zmm_1 = _mm512_mul_pd(zmm_s12, zmm_1);
      zmm_2 = _mm512_loadu_pd(&p_u[(iEdge13*nVertLevels) + k]);
      zmm_2 = _mm512_mul_pd(zmm_s13, zmm_2);
      zmm_0 = _mm512_add_pd(zmm_0, zmm_1);
      zmm_0 = _mm512_add_pd(zmm_0, zmm_2);
      zmm_0 = _mm512_mul_pd(zmm_r11, zmm_0);
#ifdef FUSE_LOOPS
      zmm_6 = _mm512_add_pd(zmm_fVertex_1, zmm_0);
      _MM512_STORE(&p_pv_vertex[(iVertex*nVertLevels) + k], zmm_6);
#endif
      _MM512_STORE(&p_vorticity[(iVertex*nVertLevels) + k], zmm_0);

      zmm_3 = _mm512_loadu_pd(&p_u[(iEdge21*nVertLevels) + k]);
      zmm_3 = _mm512_mul_pd(zmm_s21, zmm_3);
      zmm_4 = _mm512_loadu_pd(&p_u[(iEdge22*nVertLevels) + k]);
      zmm_4 = _mm512_mul_pd(zmm_s22, zmm_4);
      zmm_5 = _mm512_loadu_pd(&p_u[(iEdge23*nVertLevels) + k]);
      zmm_5 = _mm512_mul_pd(zmm_s23, zmm_5);
      zmm_3 = _mm512_add_pd(zmm_3, zmm_4);
      zmm_3 = _mm512_add_pd(zmm_3, zmm_5);
      zmm_3 = _mm512_mul_pd(zmm_r21, zmm_3);
#ifdef FUSE_LOOPS
      zmm_7 = _mm512_add_pd(zmm_fVertex_2, zmm_3);
      _MM512_STORE(&p_pv_vertex[((iVertex+1)*nVertLevels) + k], zmm_7);
#endif
      _MM512_STORE(&p_vorticity[((iVertex+1)*nVertLevels) + k], zmm_3);
    }
```

Loop Block-3

```c
#if defined (USE_INTRINSICS)
      zmm_r1 = _mm512_set1_pd(r1);
      zmm_r2 = _mm512_set1_pd(r2);

      for (k=0; k<nVertLevels; k+=8) {
        zmm_pv1 = _mm512_loadu_pd(&p_pv_vertex[(v1*nVertLevels) + k]);
        zmm_pv2 = _mm512_loadu_pd(&p_pv_vertex[(v2*nVertLevels) + k]);
        zmm_c1  = _mm512_loadu_pd(&p_pv_cell[(c1*nVertLevels) + k]);
        zmm_c2  = _mm512_loadu_pd(&p_pv_cell[(c2*nVertLevels) + k]);

        zmm_pve = _mm512_mul_pd(zmm_r0, _mm512_add_pd(zmm_pv1, zmm_pv2));
        zmm_pvt = _mm512_mul_pd(zmm_r1, _mm512_sub_pd(zmm_pv2, zmm_pv1));
        zmm_pvn = _mm512_mul_pd(zmm_r2, _mm512_sub_pd(zmm_c2, zmm_c1));

        zmm_v1 = _mm512_loadu_pd(&p_v[(iEdge*nVertLevels) + k]);
        zmm_u1 = _mm512_loadu_pd(&p_u[(iEdge*nVertLevels) + k]);

        zmm_t2  = _mm512_mul_pd(zmm_v1, zmm_pvt);
        zmm_t2  = _mm512_fmadd_pd(zmm_u1, zmm_pvn, zmm_t2);
        zmm_pve = _mm512_fnmadd_pd(zmm_r, zmm_t2, zmm_pve);

        _mm512_stream_pd(&p_pv_edge[(iEdge*nVertLevels) + k], zmm_pve);
        _mm512_stream_pd(&p_gradPVt[(iEdge*nVertLevels) + k], zmm_pvt);
        _mm512_stream_pd(&p_gradPVn[(iEdge*nVertLevels) + k], zmm_pvn);

#if defined (PREFETCH)
#if 1
        _mm_prefetch((char *) &p_v[((iEdge+32)*nVertLevels) + k], _MM_HINT_T0);
        _mm_prefetch((char *) &p_u[((iEdge+32)*nVertLevels) + k], _MM_HINT_T0);
#else
        _mm_prefetch((char *) &p_v[((iEdge+__pf_dist)*nVertLevels) + k], _MM_HINT_T1);
        _mm_prefetch((char *) &p_u[((iEdge+__pf_dist)*nVertLevels) + k], _MM_HINT_T1);
#endif
#endif
      }
#else
#pragma vector always aligned nontemporal(p_pv_edge, p_gradPVt, p_gradPVn)
      for (k=0; k<nVertLevels; k++) {
        p_pv_edge[(iEdge*nVertLevels) + k] = 0.5 * (p_pv_vertex[(v1*nVertLevels) + k] +
                                                     p_pv_vertex[(v2*nVertLevels) + k]);
        p_gradPVt[(iEdge*nVertLevels) + k] =  r1 * (p_pv_vertex[(v2*nVertLevels) + k] -
                                                     p_pv_vertex[(v1*nVertLevels) + k]);
        p_gradPVn[(iEdge*nVertLevels) + k] =  r2 * (p_pv_cell[(c2*nVertLevels) + k] -
                                                     p_pv_cell[(c1*nVertLevels) + k]);

        p_pv_edge[(iEdge*nVertLevels) + k] = p_pv_edge[(iEdge*nVertLevels) + k] - r *
                                             (p_v[(iEdge*nVertLevels) + k] *
                                              p_gradPVt[(iEdge*nVertLevels) + k] +
```

atm_recover_large_step_variables

```
      do iCell=cellStart,cellEnd

!DIR$ vector always aligned nontemporal(rho_p)
!DIR$ IVDEP
         do k = 1, nVertLevels
            rho_p(k,iCell) = rho_p_save(k,iCell) + rho_pp(k,iCell)

            rho_zz(k,iCell) = rho_p(k,iCell) + rho_base(k,iCell)
         end do

         w(1,iCell) = 0.0

!DIR$ vector nontemporal(rw)
!DIR$ IVDEP
         do k = 2, nVertLevels
            wwAvg(k,iCell) = rw_save(k,iCell) + (wwAvg(k,iCell) * invNs)
            rw(k,iCell) = rw_save(k,iCell) + rw_p(k,iCell)

           ! pick up part of diagnosed w from omega - divide by density later
            w(k,iCell) = rw(k,iCell)/(fzm(k)*zz(k,iCell)+fzp(k)*zz(k-1,iCell))

         end do

         w(nVertLevels+1,iCell) = 0.0

         if (rk_step == 3) then
!DIR$ vector always aligned nontemporal(theta_m, rtheta_p, exner, pressure_p)
!DIR$ IVDEP
            do k = 1, nVertLevels
               rtheta_p(k,iCell) = rtheta_p_save(k,iCell) + rtheta_pp(k,iCell) &
                                   - dt * rho_zz(k,iCell) * rt_diabatic_tend(k,iCell)
               theta_m(k,iCell) = (rtheta_p(k,iCell) + rtheta_base(k,iCell))/rho_zz(k,iCell)
               exner(k,iCell) = (zz(k,iCell)*(rgas/p0)*(rtheta_p(k,iCell)+rtheta_base(k,iCell)))**rcv
               ! pressure_p is perturbation pressure
               pressure_p(k,iCell) = zz(k,iCell) * rgas * (exner(k,iCell)*rtheta_p(k,iCell)+rtheta_base(k,iCell)  &
                                                         * (exner(k,iCell)-exner_base(k,iCell)))
            end do
         else
!DIR$ vector always aligned nontemporal(theta_m, rtheta_p)
!DIR$ IVDEP
            do k = 1, nVertLevels
               rtheta_p(k,iCell) = rtheta_p_save(k,iCell) + rtheta_pp(k,iCell)
               theta_m(k,iCell) = (rtheta_p(k,iCell) + rtheta_base(k,iCell))/rho_zz(k,iCell)
            end do
         end if

      end do
```

rho_p is stored in register, so it's safe to generate non-temporal stores

wwAvg needs to be read and then updated, so using non-temporal stores would be detrimental to perf.

- Using non-temporal stores through Compiler flag is almost always a bad idea unless if you are dealing with STREAM benchmark like kernels
- Selectively apply non-temporal stores by identifying store buffers that need not be read in calculations
- Intermediate results are stored in registers, so it's safe to force non-temporal stores even if the store buffer is read by statements in the loop body
- Improves performance by 1.18x

Check for Compiler generated strided refs

Baseline

```fortran
      if (dynamics_substep < dynamics_split) then

         ru_save(:,edgeStart:edgeEnd) = ru(:,edgeStart:edgeEnd)
         rw_save(:,cellStart:cellEnd) = rw(:,cellStart:cellEnd)
         rtheta_p_save(:,cellStart:cellEnd) = rtheta_p(:,cellStart:cellEnd)
         rho_p_save(:,cellStart:cellEnd) = rho_p(:,cellStart:cellEnd)

         u_1(:,edgeStart:edgeEnd) = u_2(:,edgeStart:edgeEnd)
         w_1(:,cellStart:cellEnd) = w_2(:,cellStart:cellEnd)
         theta_m_1(:,cellStart:cellEnd) = theta_m_2(:,cellStart:cellEnd)
         rho_zz_1(:,cellStart:cellEnd) = rho_zz_2(:,cellStart:cellEnd)

      end if

      if (dynamics_substep == 1) then
         ruAvg_split(:,edgeStart:edgeEnd) = ruAvg(:,edgeStart:edgeEnd)
         wwAvg_split(:,cellStart:cellEnd) = wwAvg(:,cellStart:cellEnd)
      else
         ruAvg_split(:,edgeStart:edgeEnd) = ruAvg(:,edgeStart:edgeEnd)+ruAvg_split(:,edgeStart:edgeEnd)
         wwAvg_split(:,cellStart:cellEnd) = wwAvg(:,cellStart:cellEnd)+wwAvg_split(:,cellStart:cellEnd)
      end if

      if (dynamics_substep == dynamics_split) then
         ruAvg(:,edgeStart:edgeEnd) = ruAvg_split(:,edgeStart:edgeEnd) * inv_dynamics_split
         wwAvg(:,cellStart:cellEnd) = wwAvg_split(:,cellStart:cellEnd) * inv_dynamics_split
         rho_zz_1(:,cellStart:cellEnd) = rho_zz_old_split(:,cellStart:cellEnd)
      end if
#endif
```

```
LOOP BEGIN at mpas_atm_time_integration.F(3770,10)
   remark #15389: vectorization support: reference at (3770:10) has unaligned access
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15329: vectorization support: non-unit strided store was emulated for the variable <WWAVG(:,:)>, stride is unknown to compiler
   remark #15305: vectorization support: vector length 2
   remark #15309: vectorization support: normalized vectorization overhead 1.667
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15453: unmasked strided stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 4
   remark #15477: vector cost: 3.000
   remark #15478: estimated potential speedup: 1.290
   remark #15488: --- end vector cost summary ---
LOOP END
```

Tuned

```fortran
#ifdef AWE_ATM_RK_DYNAMICS_SUBSTEP_FINISH
      if (dynamics_substep < dynamics_split) then
        do j=edgeStart, edgeEnd
         ru_save(:,j) = ru(:,j)
         u_1(:,j)     = u_2(:,j)
        end do

        do j=cellStart, cellEnd
         rw_save(:,j)       = rw(:,j)
         rtheta_p_save(:,j) = rtheta_p(:,j)
         rho_p_save(:,j)    = rho_p(:,j)

         w_1(:,j) = w_2(:,j)
         theta_m_1(:,j) = theta_m_2(:,j)
         rho_zz_1(:,j)  = rho_zz_2(:,j)
        end do
      end if

      if (dynamics_substep == 1) then
        do j=edgeStart, edgeEnd
         ruAvg_split(:,j) = ruAvg(:,j)
        end do

        do j=cellStart, cellEnd
         wwAvg_split(:,j) = wwAvg(:,j)
        end do
      else
        do j=edgeStart, edgeEnd
         ruAvg_split(:,j) = ruAvg(:,j) + ruAvg_split(:,j)
        end do
        do j=cellStart, cellEnd
         wwAvg_split(:,j) = wwAvg(:,j) + wwAvg_split(:,j)
        end do
      end if

      if (dynamics_substep == dynamics_split) then
        do j=edgeStart, edgeEnd
         ruAvg(:,j) = ruAvg_split(:,j) * inv_dynamics_split
        end do

        do j=cellStart, cellEnd
         wwAvg(:,j)    = wwAvg_split(:,j) * inv_dynamics_split
         rho_zz_1(:,j) = rho_zz_old_split(:,j)
        end do
      end if
#else
```

```
LOOP BEGIN at mpas_atm_time_integration.F(3741,10)
   remark #25399: memcopy generated
   remark #15542: loop was not vectorized: inner loop was already vectorized
```

- Nudge the Compiler to use memcpy() over multi-versioned strided loads/stores
- Either use “CONTIGOUS” keyword in pointer declarations (or) -assume contiguous_assumed_shape -assume contiguous_pointer

Misc

- Compiler is your friend - read its optimization reports of hot functions. Even better, look at generated ASM (as the Compiler code-gen can also have perf. bugs)
- Know your loop bounds and guide the Compiler in targeting code optimizations (unrolling, unroll-jam)
- Align your arrays on 64-byte boundary (IFORT: -align array64byte)
- Important to pad the leading dimensions of multi-dimensional arrays to 64-byte boundary as well
- Surprised that MPAS does not use a memory manager, it would have provided a handy mechanism to tweak memory alignment in a single wrapper function
- Use prefetches with care – understand which of the mem. references are coming from DRAM vs cache hierarchy

intel®

Backup

Compiler Flags

- AWE_FFLAGS = -qopt-zmm-usage=high \
  -align array64byte \
  -assume contiguous_assumed_shape \
  -assume contiguous_pointer

- AWE_CPP_FLAGS = -DAWE_DMPAR \
  -DAWE_ATM_ADVANCE_SCALARS \
  -DAWE_ATM_COMPUTE_SOLVE_DIAGNOSTICS \
  -DAWE_ATM_RECOVER_LARGE_STEP_VARIABLES \
  -DAWE_ATM_RK_DYNAMICS_SUBSTEP_FINISH \
  -UAWE_SMALL_STEP_PREP \
  -UAWE_ATM_ADVANCE_ACOUSTIC_STEP \
  -UAWE_ATM_COMPUTE_DYN_TEND