Mixed-Language Programming with Fortran and Data Parallel C++

Christoph Bauinger (Technical Consulting Engineer Intern)

James Tullos (Technical Consulting Engineer)

intel®

Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation. Learn more at intel.com or from the OEM or retailer.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804. https://software.intel.com/en-us/articles/optimization-notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Goals

- Offload computationally intensive tasks in existing Fortran code using DPC++
- Use as much of the existing Fortran code as possible and use DPC++ only for offload
- Minimize changes to the code
- Establish BKMs, patterns and designs useful for DPC++ code migration
- Quantify and compare performance (DPC++, OMP)

Code Analysis – The Driver

Bottleneck → Offload

```fortran
  call write_state_stats("Input State", log_file_unit)

  ! Get the start time
  call system_clock(count_start, count_rate)

#ifdef ENABLE_GPTL
  if (do_profile == 1) then
     ret = gptlstart('kernel')
  end if
#endif

  ! Run the kernel
  !$OMP parallel do schedule(runtime)
  do k=1, npz
     call c_sw(sw_corner, se_corner, nw_corner, ne_corner,              &
               rarea, rarea_c, sin_sg, cos_sg, sina_v, cosa_v,           &
               sina_u, cosa_u, fC, rdxc, rdyc, dx, dy, dxc, dyc,        &
               cosa_s, rsin_u, rsin_v, rsin2, dxa, dya,                 &
               delpc(isd,jsd,k), delp(isd,jsd,k), ptc(isd,jsd,k),       &
               pt(isd,jsd,k), u(isd,jsd,k), v(isd,jsd,k),               &
               w(isd,jsd,k), uc(isd,jsd,k), vc(isd,jsd,k),              &
               ua(isd,jsd,k), va(isd,jsd,k), wc(isd,jsd,k),             &
               ut(isd,jsd,k), vt(isd,jsd,k), divg_d(isd,jsd,k), dt2)
  enddo

#ifdef ENABLE_GPTL
  if (do_profile == 1) then
     ret = gptlstop('kernel')
  end if
#endif

  ! Get the stop time
  call system_clock(count_end, count_rate)

  ! Write the output state statistics to the log file
  call write_state_stats("Output State", log_file_unit)
```

Driver – in Fortran

```fortran

  !------------------------------------------------------------------
  ! c_sw
  !
  ! Top-level routine for c_sw kernel
  !------------------------------------------------------------------
  subroutine c_sw(sw_corner, se_corner, nw_corner, ne_corner,       &
                  rarea, rarea_c, sin_sg, cos_sg, sina_v, cosa_v,   &
                  sina_u, cosa_u, fC, rdxc, rdyc, dx, dy, dxc, dyc, &
                  cosa_s, rsin_u, rsin_v, rsin2, dxa, dya, delpc,   &
                  delp, ptc, pt, u, v, w, uc, vc, ua, va, wc, ut,   &
                  vt, divg_d, dt2)

    logical, intent(   in) :: sw_corner, se_corner, ne_corner, nw_corner
    real,    intent(   in), dimension(isd:ied,  jsd:jed    ) :: rarea
    real,    intent(   in), dimension(isd:ied+1,jsd:jed+1  ) :: rarea_c
    real,    intent(   in), dimension(isd:ied,  jsd:jed,  9) :: sin_sg, cos_sg
    real,    intent(   in), dimension(isd:ied,  jsd:jed+1  ) :: sina_v, cosa_v
    real,    intent(   in), dimension(isd:ied+1,jsd:jed    ) :: sina_u, cosa_u
    real,    intent(   in), dimension(isd:ied+1,jsd:jed+1  ) :: fC
    real,    intent(   in), dimension(isd:ied+1,jsd:jed    ) :: rdxc, dy, dxc
    real,    intent(   in), dimension(isd:ied,  jsd:jed+1  ) :: rdyc, dx, dyc
    real,    intent(   in), dimension(isd:ied,  jsd:jed    ) :: cosa_s
    real,    intent(   in), dimension(isd:ied+1,jsd:jed    ) :: rsin_u
    real,    intent(   in), dimension(isd:ied,  jsd:jed+1  ) :: rsin_v
    real,    intent(   in), dimension(isd:ied,  jsd:jed    ) :: rsin2, dxa, dya
    real,    intent(inout), dimension(isd:ied,  jsd:jed+1  ) :: u, vc
    real,    intent(inout), dimension(isd:ied+1,jsd:jed    ) :: v, uc
    real,    intent(inout), dimension(isd:ied,  jsd:jed    ) :: delp, pt, ua
    real,    intent(inout), dimension(isd:ied,  jsd:jed    ) :: va, ut, vt, w
    real,    intent(  out), dimension(isd:ied,  jsd:jed    ) :: delpc, ptc, wc
    real,    intent(  out), dimension(isd:ied+1,jsd:jed+ 1 ) :: divg_d
    real,    intent(   in)                                   :: dt2

    ! Local:
    real, dimension(is-1:ie+1, js-1:je+1) :: vort, ke
    real, dimension(is-1:ie+2, js-1:je+1) :: fx, fx1, fx2
    real, dimension(is-1:ie+1, js-1:je+2) :: fy, fy1, fy2
    real                                  :: dt4
    integer                               :: i, j
    integer                               :: iep1, jep1
    integer                               :: ret
```

Kernel Parameters: 2- and 3-dimensional arrays

Code Analysis – Kernel c_sw

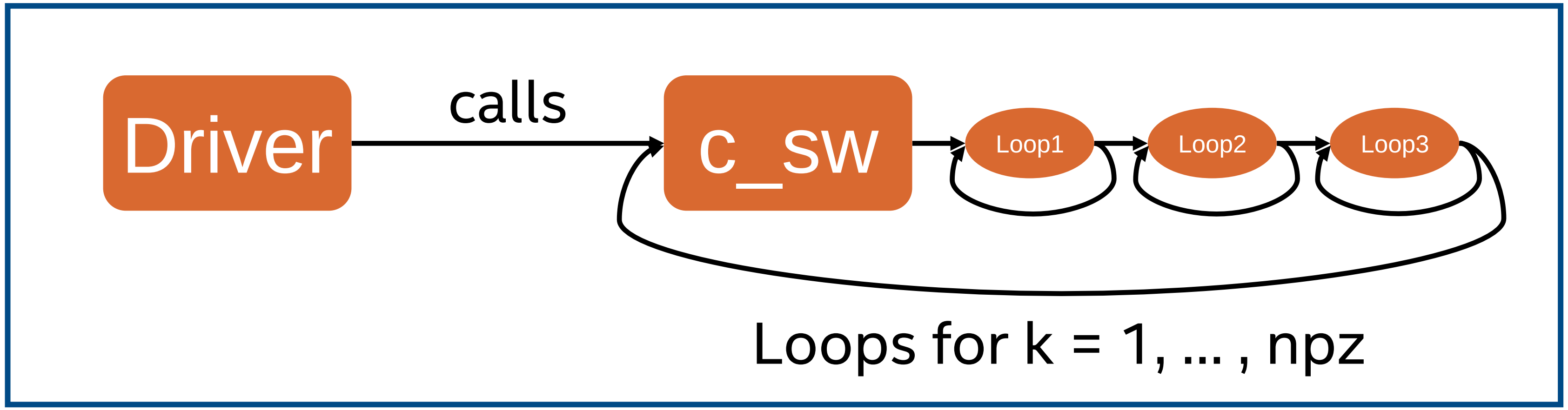
- Kernel consists of **function calls** and nested **loops**
- ~800 lines of c_sw kernel Fortran code

```fortran

    iep1 = ie + 1
    jep1 = je + 1

    call d2a2c_vect(sw_corner, se_corner, ne_corner, nw_corner,           &
                    sin_sg, cosa_u, cosa_v, cosa_s, rsin_u, rsin_v, rsin2, &
                    dxa, dya, u, v, ua, va, uc, vc, ut, vt)

    if ( nord > 0 ) then
      call divergence_corner(sw_corner, se_corner, ne_corner, nw_corner, &
                             rarea_c, sin_sg, cos_sg,                    &
                             dxc, dyc, u, v, ua, va, divg_d)
    endif

    do j = js-1, jep1
      do i = is-1, iep1+1
        if (ut(i, j) > 0.) then
              ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i-1, j, 3)
        else
              ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i, j, 1)
        end if
      enddo
    enddo
    do j = js-1, je+2
      do i = is-1, iep1
        if (vt(i, j) > 0.) then
          vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j-1, 4)
        else
          vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j, 2)
          end if
      enddo
    enddo

```

Piece of the kernel code

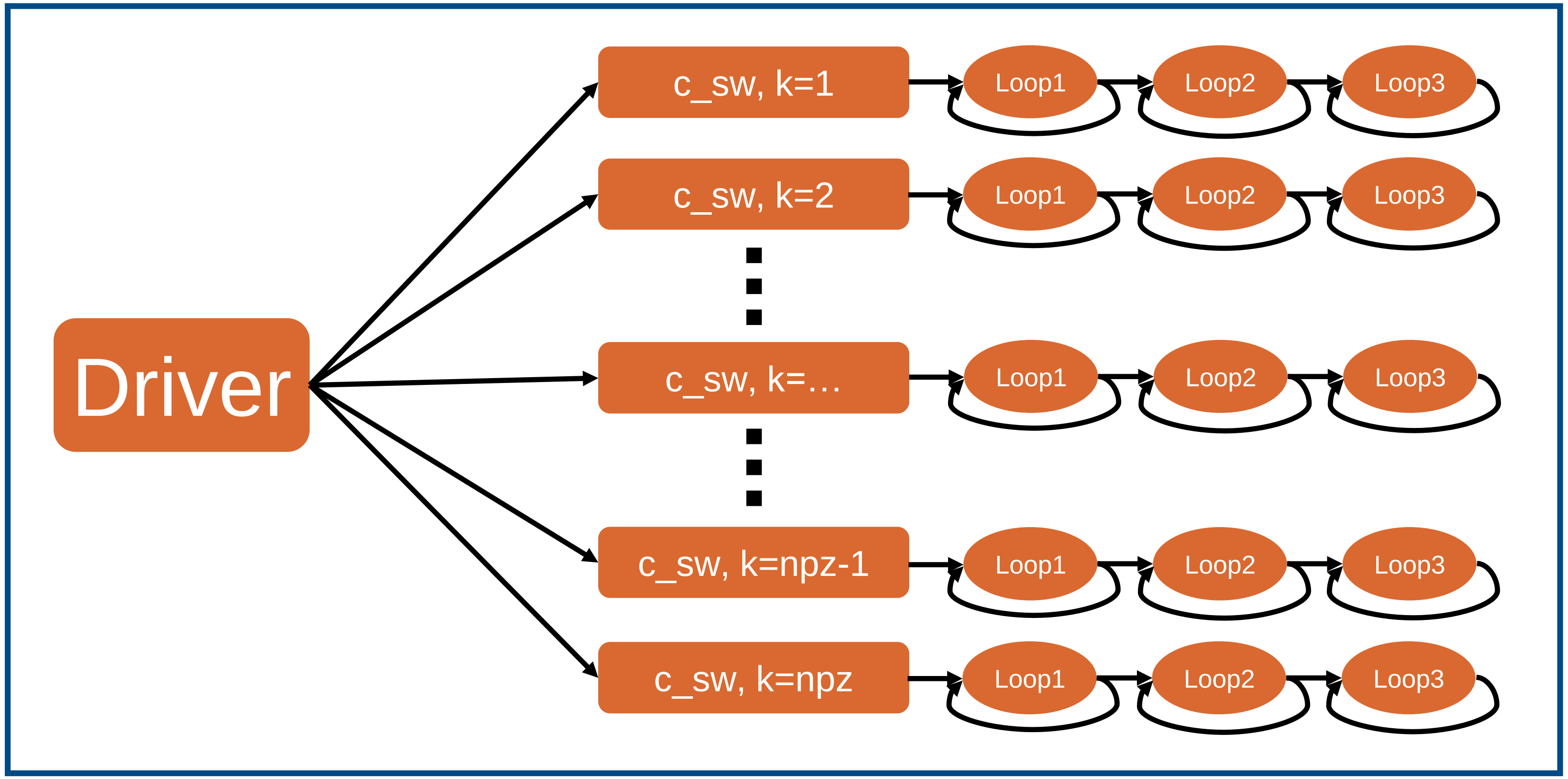
Code Flow Chart

Serial

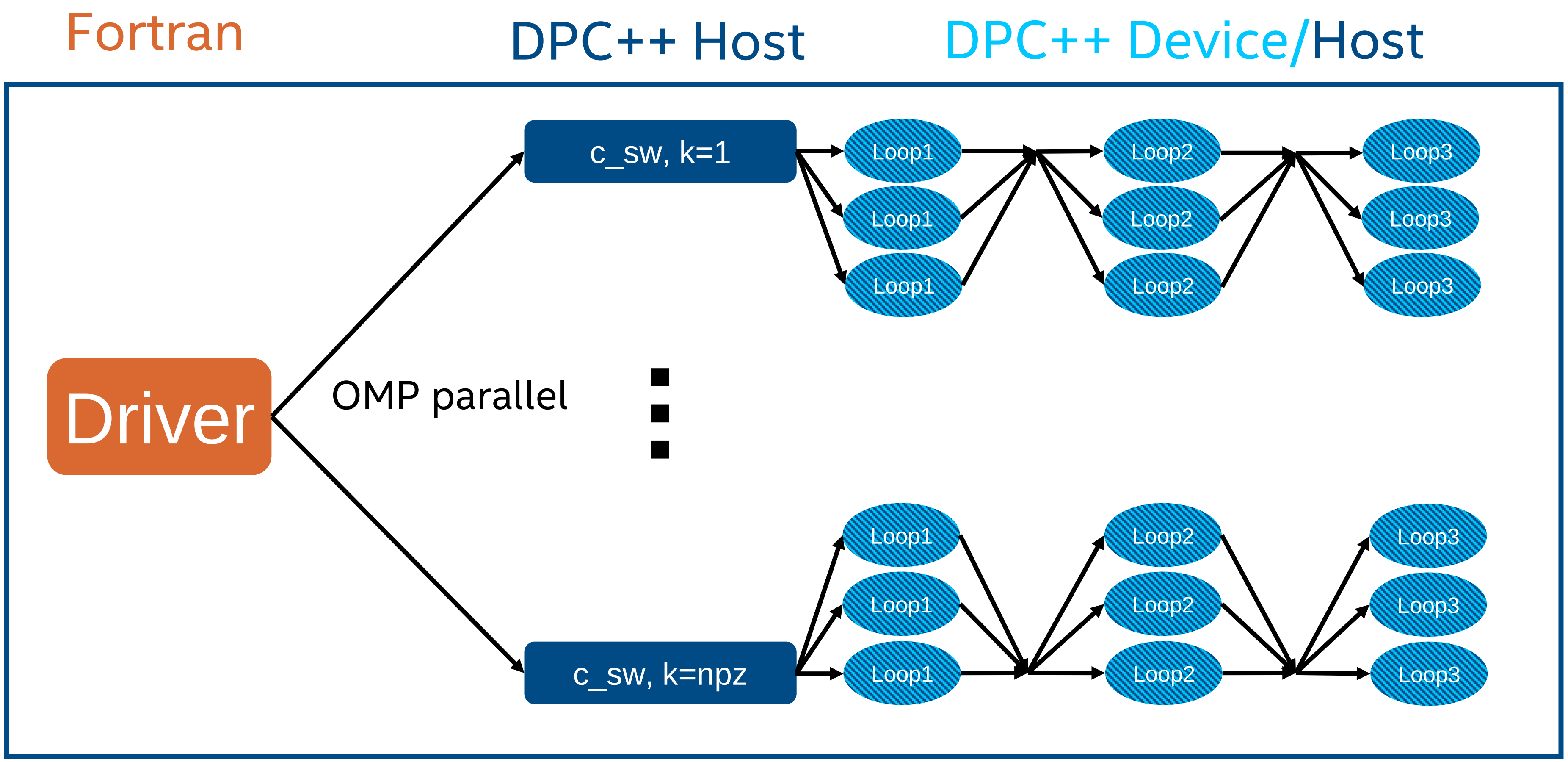


Code Flow Chart

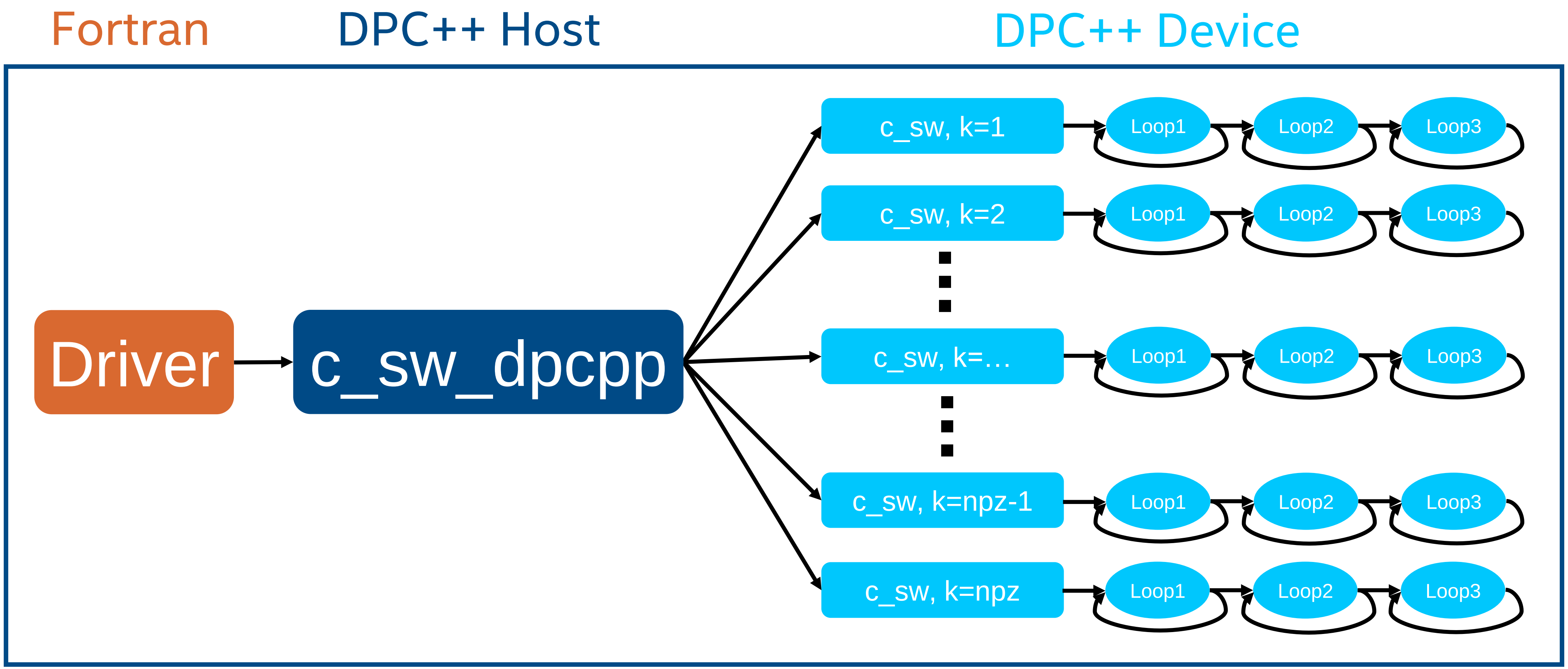
OMP parallel



Possible Offload - Approach ‘Loop’



Possible Offload - Approach ‘Outer’



Design Challenges

- Non-zero start of Fortran arrays
- Non-zero start of loops
- Fortran column-major vs C++ row-major
- Access operator in Fortran differs from C++
- No memory allocations in device code
- Many arrays as parameters
- Solution: Two step migration to C++ and then DPC++, use Approach 'Outer'

```fortran

  !-------------------------------------------------------------------
  ! c_sw
  !
  ! Top-level routine for c_sw kernel
  !-------------------------------------------------------------------
  subroutine c_sw(sw_corner, se_corner, nw_corner, ne_corner,       &
                  rarea, rarea_c, sin_sg, cos_sg, sina_v, cosa_v,   &
                  sina_u, cosa_u, fC, rdxc, rdyc, dx, dy, dxc, dyc, &
                  cosa_s, rsin_u, rsin_v, rsin2, dxa, dya, delpc,   &
                  delp, ptc, pt, u, v, w, uc, vc, ua, va, wc, ut,   &
                  vt, divg_d, dt2)

    logical, intent(   in) :: sw_corner, se_corner, ne_corner, nw_corner
    real,    intent(   in), dimension(isd:ied,   jsd:jed     ) :: rarea
    real,    intent(   in), dimension(isd:ied+1,jsd:jed+1  ) :: rarea_c
    real,    intent(   in), dimension(isd:ied,   jsd:jed,  9) :: sin_sg, cos_sg
    real,    intent(   in), dimension(isd:ied,   jsd:jed+1  ) :: sina_v, cosa_v
    real,    intent(   in), dimension(isd:ied+1,jsd:jed     ) :: sina_u, cosa_u
    real,    intent(   in), dimension(isd:ied+1,jsd:jed+1  ) :: fC
    real,    intent(   in), dimension(isd:ied+1,jsd:jed     ) :: rdxc, dy, dxc
    real,    intent(   in), dimension(isd:ied,   jsd:jed+1  ) :: rdyc, dx, dyc
    real,    intent(   in), dimension(isd:ied,   jsd:jed     ) :: cosa_s
    real,    intent(   in), dimension(isd:ied+1,jsd:jed     ) :: rsin_u
    real,    intent(   in), dimension(isd:ied,   jsd:jed+1  ) :: rsin_v
    real,    intent(   in), dimension(isd:ied,   jsd:jed     ) :: rsin2, dxa, dya
    real,    intent(inout), dimension(isd:ied,   jsd:jed+1  ) :: u, vc
    real,    intent(inout), dimension(isd:ied+1,jsd:jed     ) :: v, uc
    real,    intent(inout), dimension(isd:ied,   jsd:jed     ) :: delp, pt, ua
    real,    intent(inout), dimension(isd:ied,   jsd:jed     ) :: va, ut, vt, w
    real,    intent(  out), dimension(isd:ied,   jsd:jed     ) :: delpc, ptc, wc
    real,    intent(  out), dimension(isd:ied+1,jsd:jed+ 1 ) :: divg_d
    real,    intent(   in)                                    :: dt2

    ! Local:
    real, dimension(is-1:ie+1, js-1:je+1) :: vort, ke
    real, dimension(is-1:ie+2, js-1:je+1) :: fx, fx1, fx2
    real, dimension(is-1:ie+1, js-1:je+2) :: fy, fy1, fy2
    real                                  :: dt4
    integer                               :: i, j
    integer                               :: iep1, jep1
    integer                               :: ret
```

Design Challenges

- Non-zero start of Fortran arrays
- Non-zero start of loops
- Fortran column-major vs C++ row-major
- Access operator in Fortran differs from C++
- No memory allocations in device code
- Many arrays as parameters
- Solution: Two step migration to C++ and then DPC++, use Approach ‘Outer’

c_sw code

```fortran
    iep1 = ie + 1
    jep1 = je + 1

    call d2a2c_vect(sw_corner, se_corner, ne_corner, nw_corner,          &
                    sin_sg, cosa_u, cosa_v, cosa_s, rsin_u, rsin_v, rsin2, &
                    dxa, dya, u, v, ua, va, uc, vc, ut, vt)

    if ( nord > 0 ) then
      call divergence_corner(sw_corner, se_corner, ne_corner, nw_corner, &
                             rarea_c, sin_sg, cos_sg,                    &
                             dxc, dyc, u, v, ua, va, divg_d)
    endif

    do j = js-1, jep1
      do i = is-1, iep1+1
        if (ut(i, j) > 0.) then
              ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i-1, j, 3)
        else
              ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i, j, 1)
        end if
      enddo
    enddo
    do j = js-1, je+2
      do i = is-1, iep1
        if (vt(i, j) > 0.) then
          vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j-1, 4)
        else
          vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j, 2)
          end if
      enddo
    enddo
```

Design Challenges

- Non-zero start of Fortran arrays
- Non-zero start of loops
- Fortran column-major vs C++ row-major
- Access operator in Fortran differs from C++
- No memory allocations in device code
- Many arrays as parameters
- Solution: Two step migration to C++ and then DPC++, use Approach ‘Outer’

```fortran

    iep1 = ie + 1
    jep1 = je + 1

    call d2a2c_vect(sw_corner, se_corner, ne_corner, nw_corner,           &
                    sin_sg, cosa_u, cosa_v, cosa_s, rsin_u, rsin_v, rsin2, &
                    dxa, dya, u, v, ua, va, uc, vc, ut, vt)

    if ( nord > 0 ) then
      call divergence_corner(sw_corner, se_corner, ne_corner, nw_corner, &
                             rarea_c, sin_sg, cos_sg,                    &
                             dxc, dyc, u, v, ua, va, divg_d)
    endif

    do j = js-1, jep1
      do i = is-1, iep1+1
        if (ut(i, j) > 0.) then
              ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i-1, j, 3)
        else
              ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i, j, 1)
        end if
      enddo
    enddo
    do j = js-1, je+2
      do i = is-1, iep1
        if (vt(i, j) > 0.) then
          vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j-1, 4)
        else
          vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j, 2)
          end if
      enddo
    enddo

```

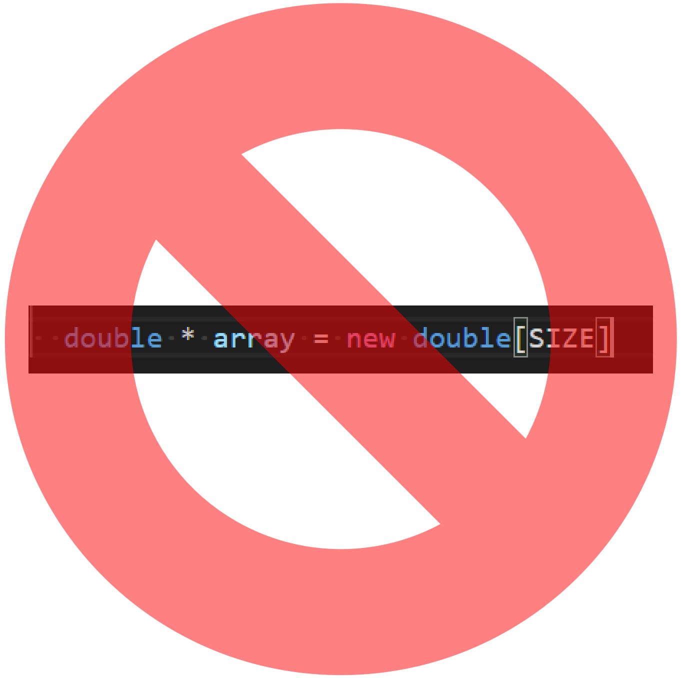
Design Challenges

- Non-zero start of Fortran arrays
- Non-zero start of loops
- Fortran column-major vs C++ row-major
- Access operator in Fortran differs from C++
- No memory allocations in device code
- Many arrays as parameters
- Solution: Two step migration to C++ and then DPC++, use Approach ‘Outer’

```
iep1 = ie + 1
jep1 = je + 1

call d2a2c_vect(sw_corner, se_corner, ne_corner, nw_corner,            &
                sin_sg, cosa_u, cosa_v, cosa_s, rsin_u, rsin_v, rsin2, &
                dxa, dya, u, v, ua, va, uc, vc, ut, vt)

if ( nord > 0 ) then
  call divergence_corner(sw_corner, se_corner, ne_corner, nw_corner, &
                         rarea_c, sin_sg, cos_sg,                    &
                         dxc, dyc, u, v, ua, va, divg_d)
endif

do j = js-1, jep1
  do i = is-1, iep1+1
    if (ut(i, j) > 0.) then
       ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i-1, j, 3)
    else
       ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i, j, 1)
    end if
  enddo
enddo
do j = js-1, je+2
  do i = is-1, iep1
    if (vt(i, j) > 0.) then
      vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j-1, 4)
    else
      vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j, 2)
    end if
  enddo
enddo
```

Design Challenges

- Non-zero start of Fortran arrays
- Non-zero start of loops
- Fortran column-major vs C++ row-major
- Access operator in Fortran differs from C++
- No memory allocations in device code
- Many arrays as parameters
- Solution: Two step migration to C++ and then DPC++, use Approach ‘Outer’



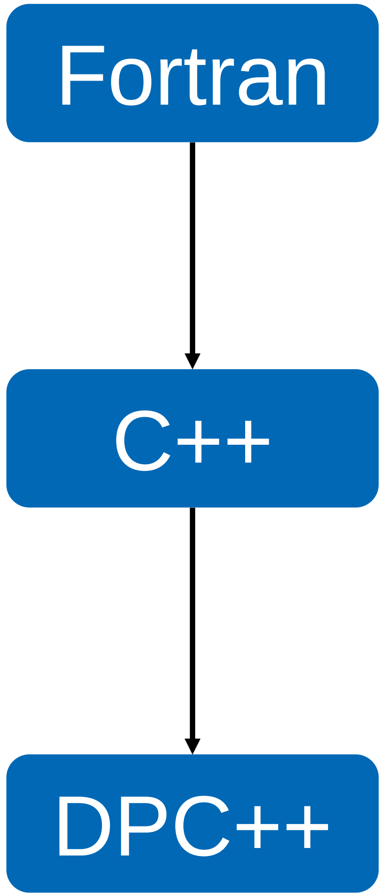
Design Challenges

- Non-zero start of Fortran arrays
- Non-zero start of loops
- Fortran column-major vs C++ row-major
- Access operator in Fortran differs from C++
- No memory allocations in device code
- Many arrays as parameters
- Solution: Two step migration to C++ and then DPC++, use Approach ‘Outer’

```
  call write_state_stats("Input State", log_file_unit)

  ! Get the start time
  call system_clock(count_start, count_rate)

#ifdef ENABLE_GPTL
  if (do_profile == 1) then
    ret = gptlstart('kernel')
  end if
#endif

  ! Run the kernel
  !$OMP parallel do schedule(runtime)
  do k=1, npz
    call c_sw(sw_corner, se_corner, nw_corner, ne_corner,                  &
              rarea, rarea_c, sin_sg, cos_sg, sina_v, cosa_v,              &
              sina_u, cosa_u, fC, rdxc, rdyc, dx, dy, dxc, dyc,            &
              cosa_s, rsin_u, rsin_v, rsin2, dxa, dya,                     &
              delpc(isd,jsd,k), delp(isd,jsd,k), ptc(isd,jsd,k),           &
              pt(isd,jsd,k), u(isd,jsd,k), v(isd,jsd,k),                   &
              w(isd,jsd,k), uc(isd,jsd,k), vc(isd,jsd,k),                  &
              ua(isd,jsd,k), va(isd,jsd,k), wc(isd,jsd,k),                 &
              ut(isd,jsd,k), vt(isd,jsd,k), divg_d(isd,jsd,k), dt2)
  enddo

#ifdef ENABLE_GPTL
  if (do_profile == 1) then
    ret = gptlstop('kernel')
  end if
#endif

  ! Get the stop time
  call system_clock(count_end, count_rate)

  ! Write the output state statistics to the log file
  call write_state_stats("Output State", log_file_unit)
```

Design Challenges

- Non-zero start of Fortran arrays
- Non-zero start of loops
- Fortran column-major vs C++ row-major
- Access operator in Fortran differs from C++
- No memory allocations in device code
- Many arrays as parameters
- Solution: Two step migration to C++ and then DPC++, use Approach ‘Outer’



Challenges Fortran - C++

- Main Issue: Array Indexing
  - Unit stride is inner index in Fortran vs outer in C++
  - Non-zero indexing in Fortran possible (and used in code)

- Solution: Class OffsetArray (1D, 2D, 3D) wrapping dynamic array emulating Fortran style arrays (see Figure)

```cpp
template <class T>
class OffsetArray
{
    const int istart;
    const int iend;
    T * const values;

  public:

    OffsetArray(int istart_in, int iend_in) :
      istart(std::min(istart_in, iend_in)),
      iend(std::max(istart_in, iend_in)),
      values(new T[iend-istart+1])
    {
    }

    ~OffsetArray() {
        delete [] values;
    }

    T &operator() (int i) {
      const int index=i-istart;
      return values[index];
    }

};


/////USAGE/////
int main()
{
    OffsetArray<double> u(5, 10);
    u(5) = 1.0;
}
```

Fortran - C++ Code Comparison

Fortran

```fortran
call d2a2c_vect(sw_corner, se_corner, ne_corner, nw_corner,            &
                sin_sg, cosa_u, cosa_v, cosa_s, rsin_u, rsin_v, rsin2, &
                dxa, dya, u, v, ua, va, uc, vc, ut, vt)

if ( nord > 0 ) then
  call divergence_corner(sw_corner, se_corner, ne_corner, nw_corner, &
                         rarea_c, sin_sg, cos_sg,                    &
                         dxc, dyc, u, v, ua, va, divg_d)
endif

do j = js-1, jep1
  do i = is-1, iep1+1
    if (ut(i, j) > 0.) then
          ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i-1, j, 3)
    else
          ut(i, j) = dt2 * ut(i, j) * dy(i, j) * sin_sg(i, j, 1)
    end if
  enddo
enddo
do j = js-1, je+2
  do i = is-1, iep1
    if (vt(i, j) > 0.) then
      vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j-1, 4)
    else
      vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j, 2)
      end if
  enddo
enddo
```

C++

```cpp
d2a2c_vect(sw_corner, se_corner, ne_corner, nw_corner, sin_sg, cosa_u,
    cosa_v, cosa_s, rsin_u, rsin_v, rsin2, dxa, dya, u, v, ua, va, uc, vc,
    ut, vt);

if (nord > 0)
{
  divergence_corner(sw_corner, se_corner, ne_corner, nw_corner,
                    rarea_c, sin_sg, cos_sg, dxc, dyc, u, v, ua, va, divg_d);
}

for (j=js-1; j<=jep1; j++)
{
  for (i=is-1; i<=iep1+1; i++)
  {
    if (ut(i,j)>0.0)
      ut(i,j)=dt2*ut(i,j)*dy(i,j)*sin_sg(i-1,j,3);
    else
      ut(i,j)=dt2*ut(i,j)*dy(i,j)*sin_sg(i,j,1);
  }
}

for (j=js-1; j<=je+2; j++)
{
  for (i=is-1; i<=iep1; i++)
  {
    if (vt(i,j)>0.0)
      vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j-1, 4);
    else
      vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j, 2);
  }
}
```

Challenges C++ - DPC++

- Main Issues:
  - OffsetArray class cannot be used in device code (as it is)
  - DPC++ parallel_for indices start at 0 (‘loop indexing starting at 0’)

- Solutions:
  - Naively: Simple offsets based on loop and array start index
  - Better: Adjusting the OffsetArray class and use Approach ‘Outer’

C++ - DPC++ Code Comparison: Naively

C++

```cpp
d2a2c_vect(sw_corner, se_corner, ne_corner, nw_corner, sin_sg, cosa_u,
        cosa_v, cosa_s, rsin_u, rsin_v, rsin2, dxa, dya, u, v, ua, va, uc, vc,
        ut, vt);

if (nord > 0)
{
  divergence_corner(sw_corner, se_corner, ne_corner, nw_corner,
                    rarea_c, sin_sg, cos_sg, dxc, dyc, u, v, ua, va, divg_d);
}

for (j=js-1; j<=jep1; j++)
{
  for (i=is-1; i<=iep1+1; i++)
  {
    if (ut(i,j)>0.0)
      ut(i,j)=dt2*ut(i,j)*dy(i,j)*sin_sg(i-1,j,3);
    else
      ut(i,j)=dt2*ut(i,j)*dy(i,j)*sin_sg(i,j,1);
  }
}

for (j=js-1; j<=je+2; j++)
{
  for (i=is-1; i<=iep1; i++)
  {
    if (vt(i,j)>0.0)
      vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j-1, 4);
    else
      vt(i, j) = dt2 * vt(i, j) * dx(i, j) * sin_sg(i, j, 2);
  }
}
```

DPC++

```cpp
d2a2c_vect();

if (nord > 0)
  divergence_corner();


Q_.submit([&](handler &h) {
  accessor abut{but, h, read_write};
  accessor abdy{bdy, h, read_only};
  accessor absin_sg{bsin_sg, h, read_only};
  const int i_off = -isd + is-1;
  const int j_off = -jsd + js-1;
  const int k_off = -1;
  h.parallel_for(range<2>{static_cast<size_t>(jep1-(js-1)+1), static_cast<size_t>(iep1+1-(is-1)+1)}, [=](id<2> idx) {
    const int i = idx[1] + i_off;
    const int j = idx[0] + j_off;
    if (abut[j][i]>0.0)
      abut[j][i]=ldt2*abut[j][i]*abdy[j][i]*absin_sg[3+k_off][j][i-1];
    else
      abut[j][i]=ldt2*abut[j][i]*abdy[j][i]*absin_sg[1+k_off][j][i];
  });
});

Q_.submit([&](handler &h) {
  accessor abvt{bvt, h, read_write};
  accessor abdx{bdx, h, read_only};
  accessor absin_sg{bsin_sg, h, read_only};
  const int i_off = -isd + is-1;
  const int j_off = -jsd + js-1;
  const int k_off = -1;
  h.parallel_for(range<2>{static_cast<size_t>(je+2-(js-1)+1), static_cast<size_t>(iep1-(is-1)+1)}, [=](id<2> idx) {
    const int i = idx[1] + i_off;
    const int j = idx[0] + j_off;
    if (abvt[j][i]>0.0)
      abvt[j][i]=ldt2*abvt[j][i]*abdx[j][i]*absin_sg[4+k_off][j-1][i];
    else
      abvt[j][i]=ldt2*abvt[j][i]*abdx[j][i]*absin_sg[2+k_off][j][i];
  });
});
```

} DPC++ Accessors

} Offsets due to array and loop

For loop range

C++ - DPC++ Code Comparison: Naively

- Tedious migration
- Hard-to-read code
- Need to rewrite everything carefully
- Debugging is a horror
- Focus on Approach 'Outer'

Adjusting OffsetArray

- Challenges when implicitly copying objects from host to device
  - Trivial destructor required
  - Const access operator necessary (copy to device is by const value)
  - No memory allocations in device code possible; Host needs to handle memory

```cpp
template <class T>
class OffsetArray
{
    const int istart;
    const int iend;
    T * const values;

  public:
    OffsetArray() = delete;

    OffsetArray(int istart_in, int iend_in, T * values_in) :
      istart(istart_in),
      iend(istart_in),
      values(values_in)
    {
    }

    T &operator()(int i) const
    {
      return values[i-istart];
    }
};
```

Adjusting OffsetArray

- Challenges when implicitly copying objects from host to device
  - Trivial destructor required
  - Const access operator necessary (copy to device is by const value)
  - No memory allocations in device code possible; Host needs to handle memory

```
template <class T>
class OffsetArray
{
    const int istart;
    const int iend;
    T * const values;

  public:
    OffsetArray() = delete;

    OffsetArray(int istart_in, int iend_in, T * values_in) :
        istart(istart_in),
        iend(istart_in),
        values(values_in)
    {
    }

    T &operator()(int i) const
    {
        return values[i-istart];
    }
};
```

Adjusting the OffsetArray Class for Approach 2

- Challenges when implicitly copying objects from host to device
  - Trivial destructor required
  - Const access operator necessary (copy to device is by const value)
  - No memory allocations in device code possible; Host needs to handle memory

```cpp
template <class T>
class OffsetArray
{
    const int istart;
    const int iend;
    T * const values;

  public:
    OffsetArray() = delete;

    OffsetArray(int istart_in, int iend_in, T * values_in) :
        istart(istart_in),
        iend(istart_in),
        values(values_in)
    {
    }

    T &operator()(int i) const
    {
        return values[i-istart];
    }
};
```

Usage of the OffsetArray

- Allocate device memory explicitly
- Construct OffsetArray with it
- Copy memory back to host after device code execution
- Free device memory explicitly after device code execution
- Minimize allocations and copies between host and device
- OffsetArray can be used on host and device

```cpp
template <class T>
class OffsetArray
{
    const int istart;
    const int iend;
    T * const values;

  public:
    OffsetArray() = delete;

    OffsetArray(int istart_in, int iend_in, T * values_in) :
        istart(istart_in),
        iend(istart_in),
        values(values_in)
    {
    }

    T &operator()(int i) const
    {
        return values[i-istart];
    }
};
```

Variations of the OffsetArray

- Can use accessors to buffers instead of USM (T * const)
- Shared memory (malloc_shared) is also a possibility → can be used on host and device with implicit copying → would work well for Approach ‘Loop’
- A possible extension would be a factory pattern for the generation of the OffsetArray

```cpp
template <class T>
class OffsetArray
{
    const int istart;
    const int iend;
    T * const values;

  public:
    OffsetArray() = delete;

    OffsetArray(int istart_in, int iend_in, T * values_in) :
        istart(istart_in),
        iend(istart_in),
        values(values_in)
    {
    }

    T &operator()(int i) const
    {
        return values[i-istart];
    }
};
```

Putting Everything Together

Fortran Driver

```fortran
! Run the kernel
call c_sw_dpcpp(isd, ied, jsd, jed, is, ie, js, je, nord, npx, npy,  &
                npz, do_profile, dt2,                                &
                sw_corner, se_corner, nw_corner, ne_corner,          &
                rarea, rarea_c, sin_sg, cos_sg, sina_v, cosa_v,      &
                sina_u, cosa_u, fC, rdxc, rdyc, dx, dy, dxc, dyc,    &
                cosa_s, rsin_u, rsin_v, rsin2, dxa, dya,             &
                delpc, delp, ptc, pt, u, v, w, uc, vc, ua, va, wc,   &
                ut, vt, divg_d)

! !$OMP parallel do schedule(runtime)
! do k=1, npz
!    call c_sw(sw_corner, se_corner, nw_corner, ne_corner,    &
!              delpc(isd,jsd,k), delp(isd,jsd,k), ptc(isd,jsd,k),    &
!              pt(isd,jsd,k), u(isd,jsd,k), v(isd,jsd,k),            &
!              w(isd,jsd,k), uc(isd,jsd,k), vc(isd,jsd,k),           &
!              ua(isd,jsd,k), va(isd,jsd,k), wc(isd,jsd,k),          &
!              ut(isd,jsd,k), vt(isd,jsd,k), divg_d(isd,jsd,k), dt2)
! enddo
```

DPC++ Device Code = C++ Code

```cpp
d2a2c_vect(u, v, ua, va, uc, vc, ut, vt, utmp, vtmp);


if (nord > 0)
{
  divergence_corner(u, v, ua, va, divg_d, uf, vf);
}

for (j=js-1; j<=jep1; j++)
{
  for (i=is-1; i<=iep1+1; i++)
  {
    if (ut(i,j)>0.0)
      ut(i,j)=dt2*ut(i,j)*dy(i,j)*sin_sg(i-1,j,3);
    else
      ut(i,j)=dt2*ut(i,j)*dy(i,j)*sin_sg(i,j,1);
  }
}


for (j=js-1; j<=je+2; j++)
{
  for (i=is-1; i<=iep1; i++)
  {
    if (vt(i,j)>0.0)
      vt(i,j) = dt2 * vt(i,j) * dx(i, j) * sin_sg(i, j-1, 4);
    else
      vt(i,j) = dt2 * vt(i,j) * dx(i, j) * sin_sg(i, j, 2);
  }
}
```

Fortran Driver

```fortran

#ifdef ENABLE_GPTL
  if (do_profile == 1) then
      ret = gptlstart('kernel')
  end if
#endif

  ! Run the kernel
  call c_sw_dpcpp(isd, ied, jsd, jed, is, ie, js, je, nord, npx, npy,  &
              npz, do_profile, dt2,                                     &
              sw_corner, se_corner, nw_corner, ne_corner,               &
              rarea, rarea_c, sin_sg, cos_sg, sina_v, cosa_v,           &
              sina_u, cosa_u, fC, rdxc, rdyc, dx, dy, dxc, dyc,         &
              cosa_s, rsin_u, rsin_v, rsin2, dxa, dya,                  &
              delpc, delp, ptc, pt, u, v, w, uc, vc, ua, va, wc,        &
              ut, vt, divg_d)

  ! !$OMP parallel do schedule(runtime)
  ! do k=1, npz
  !    call c_sw(sw_corner, se_corner, nw_corner, ne_corner,    &
  !              delpc(isd,jsd,k), delp(isd,jsd,k), ptc(isd,jsd,k),    &
  !              pt(isd,jsd,k), u(isd,jsd,k), v(isd,jsd,k),            &
  !              w(isd,jsd,k), uc(isd,jsd,k), vc(isd,jsd,k),           &
  !              ua(isd,jsd,k), va(isd,jsd,k), wc(isd,jsd,k),          &
  !              ut(isd,jsd,k), vt(isd,jsd,k), divg_d(isd,jsd,k), dt2)
  ! enddo

```

→

New DPC++ c_sw_dpcpp

```cpp
Q.submit([&](handler &h) {
  CSW Foo(isd, ied, jsd, jed, is, ie, js, je, nord, npx, npy, npz, do_profile,
      dt2, sw_corner, se_corner, nw_corner, ne_corner, rarea, rarea_c, sin_sg,
      cos_sg, sina_v, cosa_v, sina_u, cosa_u, fC, rdxc, rdyc, dx, dy, dxc, dyc,
      cosa_s, rsin_u, rsin_v, rsin2, dxa, dya);

  h.parallel_for(npz, [=](auto k) {
    Offset2DArray<double, true> delpc(isd, ied, jsd, jed, pdelpc+k*isize*jsize);
    Offset2DArray<double, true> delp(isd, ied, jsd, jed, pdelp+k*isize*jsize);
    Offset2DArray<double, true> ptc(isd, ied, jsd, jed, pptc+k*isize*jsize);
    Offset2DArray<double, true> pt(isd, ied, jsd, jed, ppt+k*isize*jsize);
    Offset2DArray<double, true> u(isd, ied, jsd, jed+1, pu+k*isize*jsizep1);
    Offset2DArray<double, true> v(isd, ied+1, jsd, jed, pv+k*isizep1*jsize);
    Offset2DArray<double, true> w(isd, ied, jsd, jed, pw+k*isize*jsize);
    Offset2DArray<double, true> uc(isd, ied+1, jsd, jed, puc+k*isizep1*jsize);
    Offset2DArray<double, true> vc(isd, ied, jsd, jed+1, pvc+k*isize*jsizep1);
    Offset2DArray<double, true> ua(isd, ied, jsd, jed, pua+k*isize*jsize);
    Offset2DArray<double, true> va(isd, ied, jsd, jed, pva+k*isize*jsize);
    Offset2DArray<double, true> wc(isd, ied, jsd, jed, pwc+k*isize*jsize);
    Offset2DArray<double, true> ut(isd, ied, jsd, jed, put+k*isize*jsize);
    Offset2DArray<double, true> vt(isd, ied, jsd, jed, pvt+k*isize*jsize);
    Offset2DArray<double, true> divg_d(isd, ied+1, jsd, jed+1, pdivg_d+k*isizep1*jsizep1);

    Offset2DArray<double, true> vort(is-1, ie+1, js-1, je+1, pvort+k*bufisize*bufjsize);
    Offset2DArray<double, true> ke(is-1, ie+1, js-1, je+1, pke+k*bufisize*bufjsize);
    Offset2DArray<double, true> fx(is-1, ie+2, js-1, je+1, pfx+k*bufisizep1*bufjsize);
    Offset2DArray<double, true> fx1(is-1, ie+2, js-1, je+1, pfx1+k*bufisizep1*bufjsize);
    Offset2DArray<double, true> fx2(is-1, ie+2, js-1, je+1, pfx2+k*bufisizep1*bufjsize);
    Offset2DArray<double, true> fy(is-1, ie+1, js-1, je+2, pfy+k*bufisize*bufjsizep1);
    Offset2DArray<double, true> fy1(is-1, ie+1, js-1, je+2, pfy1+k*bufisize*bufjsizep1);
    Offset2DArray<double, true> fy2(is-1, ie+1, js-1, je+2, pfy2+k*bufisize*bufjsizep1);
    Offset2DArray<double, true> utmp(isd, ied, jsd, jed, putmp+k*isize*jsize);
    Offset2DArray<double, true> vtmp(isd, ied, jsd, jed, pvtmp+k*isize*jsize);
    Offset2DArray<double, true> uf(is-2, ie+2, js-1, je+2, puf+k*(bufisize+2)*bufjsizep1);
    Offset2DArray<double, true> vf(is-1, ie+2, js-2, je+2, pvf+k*bufisizep1*(bufjsize+2));

    Foo.c_sw(delpc, delp, ptc, pt, u, v, w, uc, vc, ua, va, wc, ut, vt, divg_d,
      vort, ke, fx, fx1, fx2, fy, fy1, fy2, utmp, vtmp, uf, vf);
  });
}).wait();

```

→

DPC++ Device Code

```cpp
d2a2c_vect(u, v, ua, va, uc, vc, ut, vt, utmp, vtmp);


if (nord > 0)
{
  divergence_corner(u, v, ua, va, divg_d, uf, vf);
}

for (j=js-1; j<=jep1; j++)
{
  for (i=is-1; i<=iep1+1; i++)
  {
    if (ut(i,j)>0.0)
      ut(i,j)=dt2*ut(i,j)*dy(i,j)*sin_sg(i-1,j,3);
    else
      ut(i,j)=dt2*ut(i,j)*dy(i,j)*sin_sg(i,j,1);
  }
}


for (j=js-1; j<=je+2; j++)
{
  for (i=is-1; i<=iep1; i++)
  {
    if (vt(i,j)>0.0)
      vt(i,j) = dt2 * vt(i,j) * dx(i, j) * sin_sg(i, j-1, 4);
    else
      vt(i,j) = dt2 * vt(i,j) * dx(i, j) * sin_sg(i, j, 2);
  }
}

```

Compilation

- export FC=ifx
- export CXX=dpcpp
- Link for_main.o
- Use linker flag –lifcore
- Use dpcpp for linking

```cmake
cmake_minimum_required( VERSION 3.10 )

project( c_sw LANGUAGES Fortran C CXX)

list( APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake )
set( CMAKE_DIRECTORY_LABELS ${PROJECT_NAME} )
set( CMAKE_EXE_LINKER_FLAGS "-g -lifcore" )

enable_testing()

include(${PROJECT_NAME}_compiler_flags)

find_package( OpenMP COMPONENTS C Fortran )
find_package( NetCDF REQUIRED COMPONENTS C Fortran )

add_subdirectory(src)
add_subdirectory(test)

set_target_properties( ${PROJECT_NAME} PROPERTIES LINKER_LANGUAGE CXX )
```

CMakeLists.txt

```cmake
SET(OBJS
  /global/panfs01/admin/opt/intel/oneAPI/2021.1.2/compiler/2021.1.2/linux/compiler/lib/intel64_lin/for_main.o
)

add_executable( ${PROJECT_NAME} ${OBJS} ${c_sw_src_files} )
```

src/CMakeLists.txt

Compilation

- export FC=ifx
- export CXX=dpcpp
- Link for_main.o
- Use linker flag –lifcore
- Use dpcpp for linking

```
cmake_minimum_required( VERSION 3.10 )

project( c_sw LANGUAGES Fortran C CXX)

list( APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake )
set( CMAKE_DIRECTORY_LABELS ${PROJECT_NAME} )
set( CMAKE_EXE_LINKER_FLAGS "-g -lifcore" )

enable_testing()

include(${PROJECT_NAME}_compiler_flags)

find_package( OpenMP COMPONENTS C Fortran )
find_package( NetCDF REQUIRED COMPONENTS C Fortran )

add_subdirectory(src)
add_subdirectory(test)

set_target_properties( ${PROJECT_NAME} PROPERTIES LINKER_LANGUAGE CXX )
```

CMakeLists.txt

```
SET(OBJS
  /global/panfs01/admin/opt/intel/oneAPI/2021.1.2/compiler/2021.1.2/linux/compiler/lib/intel64_lin/for_main.o
)

add_executable( ${PROJECT_NAME} ${OBJS} ${c_sw_src_files} )
```

src/CMakeLists.txt

Compilation

- export FC=ifx
- export CXX=dpcpp
- Link for_main.o
- Use linker flag –lifcore
- Use dpcpp for linking

```
cmake_minimum_required( VERSION 3.10 )

project( c_sw LANGUAGES Fortran C CXX)

list( APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake )
set( CMAKE_DIRECTORY_LABELS ${PROJECT_NAME} )
set( CMAKE_EXE_LINKER_FLAGS "-g -lifcore" )

enable_testing()

include(${PROJECT_NAME}_compiler_flags)

find_package( OpenMP COMPONENTS C Fortran )
find_package( NetCDF REQUIRED COMPONENTS C Fortran )

add_subdirectory(src)
add_subdirectory(test)

set_target_properties( ${PROJECT_NAME} PROPERTIES LINKER_LANGUAGE CXX )
```

CMakeLists.txt

```
SET(OBJS
  /global/panfs01/admin/opt/intel/oneAPI/2021.1.2/compiler/2021.1.2/linux/compiler/lib/intel64_lin/for_main.o
)

add_executable( ${PROJECT_NAME} ${OBJS} ${c_sw_src_files} )
```

src/CMakeLists.txt

Compilation

- export FC=ifx
- export CXX=dpcpp
- Link for_main.o
- Use linker flag –lifcore
- Use dpcpp for linking

```
cmake_minimum_required( VERSION 3.10 )

project( c_sw LANGUAGES Fortran C CXX)

list( APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake )
set( CMAKE_DIRECTORY_LABELS ${PROJECT_NAME} )
set( CMAKE_EXE_LINKER_FLAGS "-g -lifcore" )

enable_testing()

include(${PROJECT_NAME}_compiler_flags)

find_package( OpenMP COMPONENTS C Fortran )
find_package( NetCDF REQUIRED COMPONENTS C Fortran )

add_subdirectory(src)
add_subdirectory(test)

set_target_properties( ${PROJECT_NAME} PROPERTIES LINKER_LANGUAGE CXX )
```

CMakeLists.txt

```
SET(OBJS
  /global/panfs01/admin/opt/intel/oneAPI/2021.1.2/compiler/2021.1.2/linux/compiler/lib/intel64_lin/for_main.o
)

add_executable( ${PROJECT_NAME} ${OBJS} ${c_sw_src_files} )
```

src/CMakeLists.txt

OMP Offload – Approach ‘Loop’

Fortran OMP Offload

```fortran
!$omp target parallel do private(i,j) map(to:js,je,is,ie,npy) &
!$omp map(to: va,dya,sin_sg,vtmp,u,cosa_v,rsin_v) &
!$omp map(tofrom: vt,vc)
do j = js-1, je+2
  if ( j == 1 ) then
    do i = is-1, ie+1
      vt(i, j) = edge_interpolate4(va(i, -1:2), dya(i, -1:2))
      if ( vt(i, j) < 0. ) then
        vc(i, j) = vt(i, j) * sin_sg(i, j-1, 4)
      else
        vc(i, j) = vt(i, j) * sin_sg(i, j, 2)
      end if
    enddo
  elseif ( j == 0 .or. j == (npy-1) ) then
    do i = is-1, ie+1
      vc(i, j) = c1 * vtmp(i, j-2) + c2 * vtmp(i, j-1) + c3 * vtmp(i, j)
      vt(i, j) = (vc(i, j) - u(i, j) * cosa_v(i, j)) * rsin_v(i, j)
    enddo
  elseif ( j == 2 .or. j == (npy+1) ) then
    do i = is-1, ie+1
      vc(i, j) = c1 * vtmp(i, j+1) + c2 * vtmp(i, j) + c3 * vtmp(i, j-1)
      vt(i, j) = (vc(i, j) - u(i, j) * cosa_v(i, j)) * rsin_v(i, j)
    enddo
```

DPC++ Offload

```cpp
Q.parallel_for(static_cast<size_t>((je+2)-(js-1)+1),
  [=](id<1> idx) {
  const int j = idx[0] + js-1;
  int i;

  if ( j == 1 )
  {
    for (i = is-1; i<= ie+1; i++)
    {
      //vt(i, j) = edge_interpolate4(va(i, -1:2), dya(i, -1:2))
      double va_interp[4];
      double dya_interp[4];
      for (int k=0;k<=3;k++)
      {
        va_interp[k]=va(i,k-1);
        dya_interp[k]=dya(i,k-1);
      }

      vt(i,j)=edge_interpolate4(va_interp, dya_interp);
      if ( vt(i, j) < 0. )
        vc(i, j) = vt(i, j) * sin_sg(i, j-1, 4);
      else
        vc(i, j) = vt(i, j) * sin_sg(i, j, 2);
    }
  }
```

OMP Offload – Approach ‘Outer’

Fortran OMP Offload

```fortran
  ! Run the kernel
  !$omp target map(to:sw_corner,se_corner,ne_corner,nw_corner,rarea) &
    !$omp map(to:rarea_c,sin_sg,cos_sg,sina_v,cosa_v,sina_u,cosa_u,fC) &
    !$omp map(to:rdxc,dy,dxc,rdyc,dx,dyc,cosa_s,rsin_u,rsin_v,rsin2) &
    !$omp map(to:dxa,dya,dt2) map(from:delpc,ptc,wc,divg_d) &
    !$omp map(tofrom:u,vc,v,uc,delp,pt,ua,va,ut,vt,w)
  !$omp parallel do
  do k=1, npz
     call c_sw(sw_corner, se_corner, nw_corner, ne_corner,           &
               rarea, rarea_c, sin_sg, cos_sg, sina_v, cosa_v,       &
               sina_u, cosa_u, fC, rdxc, rdyc, dx, dy, dxc, dyc,     &
               cosa_s, rsin_u, rsin_v, rsin2, dxa, dya,              &
               delpc(isd,jsd,k), delp(isd,jsd,k), ptc(isd,jsd,k),    &
               pt(isd,jsd,k), u(isd,jsd,k), v(isd,jsd,k),            &
               w(isd,jsd,k), uc(isd,jsd,k), vc(isd,jsd,k),           &
               ua(isd,jsd,k), va(isd,jsd,k), wc(isd,jsd,k),          &
               ut(isd,jsd,k), vt(isd,jsd,k), divg_d(isd,jsd,k), dt2)
  enddo
  !$omp end target
```

DPC++ Offload

```fortran
  ! Run the kernel
  call c_sw_dpcpp(isd, ied, jsd, jed, is, ie, js, je, nord, npx, npy,  &
                  npz, do_profile, dt2,                               &
                  sw_corner, se_corner, nw_corner, ne_corner,         &
                  rarea, rarea_c, sin_sg, cos_sg, sina_v, cosa_v,     &
                  sina_u, cosa_u, fC, rdxc, rdyc, dx, dy, dxc, dyc,   &
                  cosa_s, rsin_u, rsin_v, rsin2, dxa, dya,            &
                  delpc, delp, ptc, pt, u, v, w, uc, vc, ua, va, wc,  &
                  ut, vt, divg_d)

  ! !$OMP parallel do schedule(runtime)
  ! do k=1, npz
  !    call c_sw(sw_corner, se_corner, nw_corner, ne_corner,    &
  !              delpc(isd,jsd,k), delp(isd,jsd,k), ptc(isd,jsd,k),   &
  !              pt(isd,jsd,k), u(isd,jsd,k), v(isd,jsd,k),           &
  !              w(isd,jsd,k), uc(isd,jsd,k), vc(isd,jsd,k),          &
  !              ua(isd,jsd,k), va(isd,jsd,k), wc(isd,jsd,k),         &
  !              ut(isd,jsd,k), vt(isd,jsd,k), divg_d(isd,jsd,k), dt2)
  ! enddo
```

Performance Comparison

Testcase	No Offload (+OMP)		Approach 'Outer'			Approach 'Loop'		
	Fortran	C++	OMP	DPC++ GPU	DPC++ CPU	OMP	DPC++ GPU	DPC++ CPU
Default	0.003 s	0.005 s	7.95 s	1.094 s	2.378 s	4.32 s	3.175 s	0.225 s

- Test performed on Intel® DevCloud
- Running on a small test
- Timings exclude memory movement and show pure kernel time
- DPC++ is much slower than Fortran. Why? → 'just in time' compilation

Testing Date: Performance results are **based on testing by Intel as of June 17, 2021** and may not reflect all publicly available security updates.

Configuration Details and Workload Setup: Intel® Xeon® E-2176G Processor with Intel® UHD Graphics P630 on Intel® DevCloud. Ubuntu 20.04.2 LTS, Intel® oneAPI Base Toolkit 2021.2, Intel® oneAPI HPC Toolkit 2021.2. SENA c_sw kernel at https://github.com/NOAA-GSL/SENA-c_sw.

Ahead of time compilation

- Device kernel code is compiled 'just in time' (JIT) at runtime
- A lot of device code → Takes a lot of time
- If hardware is known a priori: Pre-compile (ahead of time, AOT) device code
- Negligible for kernels that are called multiple times, or longer running kernels

CMakeLists.txt

```
cmake_minimum_required( VERSION 3.10 )

project( c_sw LANGUAGES Fortran C CXX)

list( APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake )
set( CMAKE_DIRECTORY_LABELS ${PROJECT_NAME} )
#set( CMAKE_EXE_LINKER_FLAGS "-g -lifcore" )
set( CMAKE_EXE_LINKER_FLAGS "-g -lifcore -fsycl-targets=spir64_x86_64-unknown-unknown-sycldevice \
   -Xs \"-march=sse4.2\"" )
#set( CMAKE_EXE_LINKER_FLAGS "-g -lifcore -fsycl-targets=spir64_gen-unknown-unknown-sycldevice \
#    -Xs \"-device gen9\"" )


enable_testing()

include(${PROJECT_NAME}_compiler_flags)

find_package( OpenMP COMPONENTS C Fortran )
find_package( NetCDF REQUIRED COMPONENTS C Fortran )

add_subdirectory(src)
add_subdirectory(test)

set_target_properties( ${PROJECT_NAME} PROPERTIES LINKER_LANGUAGE CXX )
```

```
add_executable( ${PROJECT_NAME} ${OBJS} ${c_sw_src_files} )

#target_compile_options(${PROJECT_NAME} PUBLIC $<$<COMPILE_LANGUAGE:CXX>:-v
#  -fsycl-targets=spir64_gen-unknown-unknown-sycldevice>)
target_compile_options(${PROJECT_NAME} PUBLIC $<$<COMPILE_LANGUAGE:CXX>:-v
    -fsycl-targets=spir64_x86_64-unknown-unknown-sycldevice>)
```

src/CMakeLists.txt

Performance Comparison

Testcase	No Offload (+OMP)		Approach 'Outer'			Approach 'Loop'		
	Fortran	C++	OMP*	DPC++ GPU	DPC++ CPU	OMP*	DPC++ GPU	DPC++ CPU
Default	0.003 s	0.005 s	7.95 s	1.094 s	2.378 s	4.32 s	3.175 s	0.225 s
AOT	0.003 s	0.005 s	7.95 s	0.094 s	0.042 s	4.32 s	3.049 s	0.079 s

- Gains from AOT more prominent if there is more device code
- DPC++ outperforms OMP
- Offload still slower due to low GPU usage and the device used
- *Known issue with AOT, OMP offload and Gen9 graphics

Testing Date: Performance results are **based on testing by Intel as of June 17, 2021** and may not reflect all publicly available security updates.

Configuration Details and Workload Setup: Intel® Xeon® E-2176G Processor with Intel® UHD Graphics P630 on Intel® DevCloud. Ubuntu 20.04.2 LTS, Intel® oneAPI Base Toolkit 2021.2, Intel® oneAPI HPC Toolkit 2021.2. SENA c_sw kernel at https://github.com/NOAA-GSL/SENA-c_sw.

Performance Comparison

Testcase	No Offload (+OMP)		Approach 'Outer'			Approach 'Loop'		
	Fortran	C++	OMP	DPC++ GPU	DPC++ CPU	OMP	DPC++ GPU	DPC++ CPU
Default	0.003 s	0.005 s	7.95 s	1.094 s	2.378 s	4.32 s	3.175 s	0.225 s
AOT	0.003 s	0.005 s	7.95 s	0.094 s	0.042 s	4.32 s	3.049 s	0.079 s
Large Case	4.239 s	3.95 s	11.66 s	16.33 s	3.94 s	60 s	7.73 s	9.29 s

- 'Large Case' test is ~330 times the size of 'Default' test
- AOT is also used in these tests (except OMP)

Testing Date: Performance results are **based on testing by Intel as of June 17, 2021** and may not reflect all publicly available security updates.

Configuration Details and Workload Setup: Intel® Xeon® E-2176G Processor with Intel® UHD Graphics P630 on Intel® DevCloud. Ubuntu 20.04.2 LTS, Intel® oneAPI Base Toolkit 2021.2, Intel® oneAPI HPC Toolkit 2021.2. SENA c_sw kernel at https://github.com/NOAA-GSL/SENA-c_sw.

Summary

- Fortran – DPC++ migration possible and feasible
- Still requires Fortran – C++ migration for kernel code
- Leveraging C++ in kernel code and implicit data movement between host and device allows neat code and easy migration
- Performance in such a first approach might not be ideal
- One of the core strengths of DPC++ is C++

Acknowledgement

The NOAA HPCC Software Engineering for Novel Architectures (SENA) program and the participating organizations: the Geophysical Fluid Dynamics Laboratory, the Global Systems Laboratory and the Environmental Modeling Center.

Resources

- NOAA c_sw kernel - https://github.com/NOAA-GSL/SENA-c_sw
- Intel® oneAPI – https://software.intel.com/oneapi
- Intel® DevCloud for oneAPI – https://devcloud.intel.com/oneapi

intel®