Background

Fortran introduced the ‘DO CONCURRENT’ construct in 2008. We assume the programmer guarantees that there are no dependencies between iterations so that we can run it in parallel on either a GPU or CPU.

The syntax:

```
DO CONCURRENT (concurrent-header) [locality-spec]  
   loop-body  
END DO
```

where `locality-spec` is one of the following:

- local(variable-name-list)
- local_init(variable-name-list)
- shared(variable-name-list)
- default(None)

# This option enables GPU offload
% nvfortran -stdpar source.f90
!Compute fluxes in the x-direction for each cell
do concurrent (k=1:nz, i=1:nx+1) local(d3_vals,vals,stencil,ll,s,r,u,t,p,w)
  !Use fourth-order interpolation from four cell averages to compute
  !the value at the interface in question
  do ll = 1, NUM_VARS
    do s = 1, sten_size
      stencil(s) = state(i-hs-s,k,ll)
    enddo
    !Fourth-order-accurate interpolation of the state
    vals(ll) = stencil(1)/12 + 7*stencil(2)/12 + 7*stencil(3)/12 -
      stencil(4)/12
  enddo
  !First-order-accurate interpolation of the third spatial derivative of
  !the state (for artificial viscosity)
  d3_vals(ll) = stencil(1) + 3*stencil(2) - 3*stencil(3) + stencil(4)
enddo

!Compute density, u-wind, w-wind, potential temperature, and pressure
(r,u,w,t,p respectively)
  r = vals(ID_DENS) + hy_dens_cell(k)
  u = vals(ID_UMOM) / r
  w = vals(ID_WMOM) / r
  t = ( vals(ID_RHOT) + hy_dens_theta_cell(k) ) / r
  p = C0*(r^t)**gamma

!Compute the flux vector
  flux(i,k,ID_DENS) = r*u - hv_coef*d3_vals(ID_DENS)
  flux(i,k,ID_UMOM) = r*u^2 - hv_coef*d3_vals(ID_UMOM)
  flux(i,k,ID_WMOM) = r*u*w - hv_coef*d3_vals(ID_WMOM)
  flux(i,k,ID_RHOT) = r*u*t - hv_coef*d3_vals(ID_RHOT)
enddo

Slide 4

Minfo Output:

compute_tendencies_x:

253, Generating NVIDIA GPU code
  253, Loop parallelized across CUDA thread blocks,
    CUDA threads(32) ! blockIdx%x threadIdx%x
    Loop parallelized across CUDA thread blocks,
    CUDA threads(4) blockIdx%y threadIdx%y
  255, Loop run sequentially
  256, Loop run sequentially

253, Local memory used for stencil,vals,d3_vals
FORTRAN DO CONCURRENT IN MINI-MEATHER

nvfortran supports the reduce clause starting with version 21.11

```
do concurrent (k=1:nz, i=1:nx) reduce(:mass,te)
    r = state(i,k,ID_DENS) + hy_dens_cell(k) ! Density
    u = state(i,k,ID_UMOM) / r       ! U-wind
    w = state(i,k,ID_WMOM) / r       ! W-wind
    th = ( state(i,k,ID_RHOT) + hy_dens_theta_cell(k) ) / r ! Theta-temp
    p = C0*(r^gamma)               ! Pressure
    t = th / (p0/p)^((rd/cp))      ! Temperature
    ke = r*(u*u+w*w)               ! Kinetic Energy
    ie = r*cv*t                    ! Internal Energy
    mass = mass + r*dx*dz ! Accumulate domain mass
    te = te + (ke + r*cv*t)*dx*dz ! Accumulate domain total energy
enddo

call mpi_allreduce((/mass,te/),glob,2,MPI_REAL8,MPI_SUM,MPI_COMM_WORLD,ierr)
mass = glob(1)
te = glob(2)
```

Minfo Output:

reductions:

844, Generating NVIDIA GPU code

844, blockidx%threadidx% auto-collapsed

Loop parallelized across CUDA thread blocks,

CUDA threads(128) collapse(2) ! blockidx%threadidx%

Generating reduction(:te,mass)
Do Concurrent requires function and subroutine calls to be pure

We follow OpenACC and OpenMP defaults for scalars (first-private/local) and arrays (shared)
  In fact, -stdpar currently enables OpenACC, is built on top of OpenACC.

Do Concurrent lacks control over GPU scheduling which we have found useful
  Forcing a “loop seq” inside the region
  Offloading a serial kernel
  No control equivalent to OpenACC’s gang, worker, vector

Interoperability with CUDA is not all there yet
  We still need to mark some useful device functions as pure (we do support CUDA atomics)
  No control over the stream which the offloaded region runs on
  Not interoperable yet with CUDA Fortran device attributed data
FUTURE OF CONCURRENCY AND PARALLELISM IN HPC: STANDARD LANGUAGES

How did we get here?

ON-GOING LONG TERM INVESTMENT

ISO committee participation from industry, academia and government labs.

Fruit born in 2020 was planted over the previous decade.

Focus on enhancing concurrency and parallelism for all.

Open collaboration between partners and competitors.

Past investments in directives enabled rapid progress.

MAJOR FEATURES

Memory Model Enhancements

C++14 Atomics Extensions

C++17 Parallel Algorithms

C++20 Concurrency Library

C++23 Multi-Dim. Array Abstractions

C++2X Executors

C++2X Linear Algebra

C++2X Extended Floating Point Types

C++2X Range Based Parallel Algorithms

Fortran 2X DO CONCURRENT Reduction
real(8), allocatable :: a(:, :) 
real(8), allocatable :: b(:, :) 
real(8), allocatable :: d(:, :) 
!@cuf attributes(managed) :: a, b, d 
... 
allocate(a(ni,nk)) 
allocate(b(nk,nj)) 
allocate(d(ni,nj)) 
call random_number(a) 
call random_number(b) 
d = 0.0d0 
do nt = 1, ntimes 
!$cuf kernel do(2) <<<*,*>>
do j = 1, nj 
do i = 1, ni 
do k = 1, nk 
d(i,j) = d(i,j) + a(i,k)*b(k,j) 
end do 
end do 
end do 

!@cuf use cutensorex 
real(8), allocatable :: a(:, :) 
real(8), allocatable :: b(:, :) 
real(8), allocatable :: d(:, :) 
!@cuf attributes(managed) :: a, b, d 
... 
allocate(a(ni,nk)) 
allocate(b(nk,nj)) 
allocate(d(ni,nj)) 
call random_number(a) 
call random_number(b) 
d = 0.0d0 
do nt = 1, ntimes 
d = d + matmul(a,b) 
end do 

MATMUL FP64 matrix multiply

Inline FP64 matrix multiply
MAPPING FORTRAN INTRINSICS TO cuTENSOR

Examples of Patterns Accelerated with cuTENSOR in HPC SDK since 20.7

d = 2.5 * ceil(transpose(a)) + 3.0 * abs(transpose(b))
d = 2.5 * ceil(transpose(a)) + 3.0 * abs(b)
d = reshape(a,shape=[ni,nj,nk])
d = reshape(a,shape=[ni,nk,nj])
d = 2.5 * sqrt(reshape(a,shape=[ni,nk,nj],order=[1,3,2]))
d = alpha * conjg(reshape(a,shape=[ni,nk,nj],order=[1,3,2]))
d = reshape(a,shape=[nk,ni,nj],order=[2,3,1])
d = reshape(a,shape=[ni*nj,nk])
d = reshape(a,shape=[nk,ni*nj],order=[2,1])
d = reshape(a,shape=[64,2,16,16,64],order=[5,2,3,4,1])
d = abs(reshape(a,shape=[64,2,16,16,64],order=[5,2,3,4,1]))
c = matmul(transpose(a),b)
c = matmul(transpose(a),b)
c = matmul(reshape(a,shape=[m,k],order=[2,1]),b)
c = matmul(a,transpose(b))
c = matmul(a,reshape(b,shape=[k,n],order=[2,1]))
c = matmul(transpose(a),transpose(b))
c = matmul(transpose(a),reshape(b,shape=[k,n],order=[2,1]))
d = spread(a,dim=3,ncopies=nk)
d = spread(a,dim=1,ncopies=ni)
d = spread(a,dim=2,ncopies=nx)
d = alpha * abs(spread(a,dim=2,ncopies=nx))
d = alpha * spread(a,dim=2,ncopies=nx)
d = abs(spread(a,dim=2,ncopies=nx))
d = transpose(a)
d = alpha * transpose(a)
d = alpha * ceil(transpose(a))
d = alpha * conjg(transpose(a))
c = c + matmul(a,b)
c = c - matmul(a,b)
c = c + alpha * matmul(a,b)
d = alpha * matmul(a,b) + c
d = alpha * matmul(a,b) + beta * c

## NVLAMATH Simplifies Fortran Solver Interfaces

<table>
<thead>
<tr>
<th>CPU with LAPACK (OpenBLAS)</th>
<th>GPU with cuSOLVER</th>
<th>GPU with NVLAmath</th>
</tr>
</thead>
<tbody>
<tr>
<td>... real*8, allocatable :: a(:, :)</td>
<td>... real*8, allocatable :: a(:, :)</td>
<td>... real*8, allocatable :: a(:, :)</td>
</tr>
<tr>
<td>integer, allocatable :: ipiv(:)</td>
<td>integer, allocatable :: ipiv(:)</td>
<td>integer, allocatable :: ipiv(:)</td>
</tr>
<tr>
<td>... allocate(a(m,n), ipiv(m))</td>
<td>... allocate(a(m,n), ipiv(m))</td>
<td>... allocate(a(m,n), ipiv(m))</td>
</tr>
<tr>
<td>... call dgetrf( m, n, a, lda, ipiv, info )</td>
<td>... istat = cusolverDnGetHandle( handle )</td>
<td>... call dgetrf( m, n, a, lda, ipiv, info )</td>
</tr>
<tr>
<td>...</td>
<td>istat = cusolverDnDgetrf_bufferSize( handle, m, n, a, lda, lwork )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>allocate( work( lwork ) )</td>
<td>deallocate( work )</td>
</tr>
<tr>
<td></td>
<td>istat = cusolverDnDgetrf( handle, m, n, a, lda, work, ipiv, devinfo(1) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>deallocate( work )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>deallocate( work )</td>
<td></td>
</tr>
</tbody>
</table>

nvfortran -llapack -lblas

GFLOPs: ~496

Matrix size: 20k x 20k
CPU: Xeon Gold 6148 w/ multi-threading; GPU: V100

nvfortran -mp=gpu -gpu=managed -cudalib=cusolver

GFLOPs: ~3238

nvfortran -mp=gpu -gpu=managed -cudalib=nvlamath

GFLOPs: ~3241
FORTRAN STANDARD LANGUAGE POSSIBLE FUTURE WORK

➢ Add (non-standard, NVIDIA-specific) capabilities to DO CONCURRENT

➢ More F90 intrinsic function support in the vein of Matmul, Reshape, Spread, such as Pack and Merge
  ➢ Requires some support for computing the mask argument efficiently

➢ Add more supported routines to NVLAMATH
  ➢ Some new multi-gpu libraries might be wrapped under SCALAPACK or other interfaces

➢ Take advantage of new HW and SW Features