Getting Fortran onto GPUs

Jeff Hammond and Jeff Larkin
NVIDIA HPC Group
Outline

• Where are we now?
  • DO CONCURRENT
  • Data parallel intrinsics
  • Directives
  • CUDA support

• Where do we want to go?
  • Fetching atomics in DO CONCURRENT
  • Asynchrony and task parallelism
# Programming the nvidia platform WITH FORTRAN

**CPU, GPU, and Network**

## ACCELERATED STANDARD MODELS

<table>
<thead>
<tr>
<th>ISO Fortran</th>
<th>OpenACC</th>
<th>OpenMP</th>
</tr>
</thead>
</table>
| **do** concurrent *(j=1:order, & i=1:order)*  
  B(i,j) = A(j,i)  
  **enddo**  
  B = transpose(A)  
| !$acc parallel loop tile(32,32)  
  do j=1,order  
  do i=1,order  
  B(i,j) = A(j,i)  
  enddo  
| !$omp target teams distribute & parallel do simd & collapse(2)  
  do j=1,order  
  do i=1,order  
  B(i,j) = A(j,i)  
  enddo  
| **ACCELERATION LIBRARIES**

<table>
<thead>
<tr>
<th>CUDA Runtime</th>
<th>CUBLAS</th>
<th>CUTENSOR</th>
<th>CUSOLVER</th>
<th>...</th>
<th>NVSHMEM</th>
</tr>
</thead>
</table>

## PLATFORM SPECIALIZATION

<table>
<thead>
<tr>
<th>CUDA Fortran</th>
</tr>
</thead>
</table>
| BIDX = blockIdx%x-1  
  BIDY = blockIdx%y-1  
  TIDX = threadIdx%x  
  TIDY = threadIdx%y  
| !$omp target teams distribute & parallel do simd & collapse(2)  
  do j=1,order  
  do i=1,order  
  B(i,j) = A(j,i)  
  enddo  
| !$omp target teams loop & collapse(2)  
  do j=1,order  
  do i=1,order  
  B(i,j) = A(j,i)  
  enddo  
| !$acc end kernels  
| !$omp target teams distribute & parallel do simd & collapse(2)  
  do j=1,order  
  do i=1,order  
  B(i,j) = A(j,i)  
  enddo  
| !$acc end kernels  

HPC PROGRAMMING IN ISO FORTRAN

ISO is the place for portable concurrency and parallelism

**Fortran 2018**

- **Fortran Array Math Intrinsics**
  - NVFORTRAN 20.5
  - Accelerated matmul, reshape, spread, ...

- **DO CONCURRENT**
  - NVFORTRAN 20.11
  - Auto-offload & multi-core

- **Co-Arrays**
  - Not currently available
  - Accelerated co-array images

**Fortran 202x**

- **DO CONCURRENT Reductions**
  - NVFORTRAN 21.11
  - REDUCE subclause added
  - Support for +, *, MIN, MAX, IAND, IOR, IECR.
  - Support for .AND., .OR., .EQV., .NEQV on LOGICAL values

Preview support available now in NVFORTRAN
MiniWeather

Standard Language Parallelism in Climate/Weather Applications

Mini-App written in C++ and Fortran that simulates weather-like fluid flows using Finite Volume and Runge-Kutta methods.

Existing parallelization in MPI, OpenMP, OpenACC, ...

Included in the SPEChpc benchmark suite*

Open-source and commonly-used in training events.

https://github.com/mrnorman/miniWeather/

```c++
// Sample code snippet from MiniWeather

do concurrent (ll=1:NUM_VARS, k=1:nz, i=1:nx)
    local(x,z,x0,z0,xrad,zrad,amp,dist,wpert)
    if (data_spec_int == DATA_SPEC_GRAVITY_WAVES) then
        x = (i_beg-1 + i-0.5_rp) * dx
        z = (k_beg-1 + k-0.5_rp) * dz
        x0 = xlen/8
        z0 = 1000
        xrad = 500
        zrad = 500
        amp = 0.01_rp
        dist = sqrt(((x-x0)/xrad)**2 + ((z-z0)/zrad)**2 )
        * pi / 2._rp
        wpert = amp * cos(dist)**2
    else
        wpert = 0._rp
    endif
    tend(i,k,ID_WMOM) = tend(i,k,ID_WMOM)
    + wpert*hy_dens_cell(k)
endif
dstate_out(i,k,ll)=state_init(i,k,ll)
+ dt * tend(i,k,ll)
enddo
```

*SPECchpc is a trademark of The Standard Performance Evaluation Corporation
POT3D: Do Concurrent + Limited OpenACC

POT3D is a Fortran application for approximating solar coronal magnetic fields.

Included in the SPEChpc benchmark suite*

Existing parallelization in MPI & OpenACC

Optimized the DO CONCURRENT version by using OpenACC solely for data motion and atomics

https://github.com/predsci/POT3D

Data courtesy of Predictive Science Inc.

* SPEChpc is a trademark of The Standard Performance Evaluation Corporation
GAMESS
Computational Chemistry with Fortran Do Concurrent

- GAMESS is a popular Quantum Chemistry application.
- More than 40 years of development in Fortran and C
- MPI + OpenMP baseline code
- Hartree-Fock rewritten in Do Concurrent

* Courtesy of Melisa Alkan, Iowa State University. Not yet published.
ACCELERATED PROGRAMMING IN ISO FORTRAN

NVFORTRAN Accelerates Fortran Intrinsics with cuTENSOR Backend

---

**MATMUL FP64 matrix multiply**

```fortran
real(8), dimension(ni,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(ni,nj) :: c
...
!$acc enter data copyin(a,b,c) create(d)
do nt = 1, ntimes
  !$acc kernels
do  j = 1, nj
  do  i = 1, ni
    d(i,j) = c(i,j)
    do  k = 1, nk
      d(i,j) = d(i,j) + a(i,k) * b(k,j)
    end do
  end do
!$acc end kernels
end do
!$acc exit data copyout(d)
```

**Naive Inline**

```fortran
real(8), dimension(ni,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(ni,nj) :: c
...
do nt = 1, ntimes
  d = c + matmul(a,b)
end do
```

---

**Inline FP64 matrix multiply**

---

---

**Utilizes A100 Tensor Cores automatically**

---

---
HPC PROGRAMMING IN ISO FORTRAN

Examples of Patterns Accelerated in NVFORTRAN

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=[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 = abs(reshape(a,shape=[64,2,16,16,64],order=[5,2,3,4,1]))
c = matmul(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 = 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
Refactoring Fortran Loops

1. Identify an important loop nest that can be run in parallel.

!Compute fluxes in the x-direction for each cell
do k = 1 , nz+1
  do i = 1 , nx
    !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,k-hs-1+s,ll)
      enddo
      !Fourth-order-accurate interpolation of the state
    enddo

    !Compute density, u-wind, w-wind, potential
    !temperature, and pressure (r,u,w,t,p respectively)
    r = vals(ID_DENS) + hy_dens_int(k)
    u = vals(ID_UMOM) / r
    w = vals(ID_WMOM) / r
    t = ( vals(ID_RHOT) + hy_dens_theta_int(k) ) / r
    p = C0*(r*t)**gamma - hy_pressure_int(k)
    ...
  enddo
enddo
Refactoring Fortran Loops

1. Identify an important loop nest that can be run in parallel.

2. Replace existing loops with do concurrent loops
   
   Note: Multiple loop iteration variables can be used in the same do concurrent loop, if they are all legal to parallelize.

```fortran
!Compute fluxes in the x-direction for each cell
do concurrent (k=1:nz, i=1:nx+1)
  !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,k-hs-1+s,ll)
    enddo
    !Fourth-order-accurate interpolation of the state
  enddo
  !Compute density, u-wind, w-wind, potential
  !Temperature, and pressure (r,u,w,t,p respectively)
  r = vals(ID_DENS) + hy_dens_int(k)
  u = vals(ID_UMOM) / r
  w = vals(ID_WMOM) / r
  t = ( vals(ID_RHOT) + hy_dens_theta_int(k) ) / r
  p = C0*(r*t)**gamma - hy_pressure_int(k)
...  
enddo
```
Refactoring Fortran Loops

1. Identify an important loop nest that can be run in parallel.

2. Replace existing loops with do concurrent loops
   Note: Multiple loop iteration variables can be used in the same do concurrent loop, if they are all legal to parallelize

3. Add local clause for variables that must be privatized for correctness.

!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,k-hs-1+s,ll)
      enddo
      !Fourth-order-accurate interpolation of the state
   enddo

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

...
Refactoring Fortran Loops

1. Identify an important loop nest that can be run in parallel.
2. Replace existing loops with do concurrent loops
   Note: Multiple loop iteration variables can be used in the same do concurrent loop, if they are all legal to parallelize
3. Add local clause for variables that must be privatized for correctness.
4. Recompile with -stdpar and test for correctness.
   Note 1: Only refactor one loop nest at a time to ensure errors aren’t introduced, such as forgetting to localize a variable.
   Note 2: Performance may get worse at first due to increased memory migration.

!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,k-hs-1+s,ll)
    enddo
    !Fourth-order-accurate interpolation of the state
  enddo
  !Compute density, u-wind, w-wind, potential
  !temperature, and pressure (r,u,w,t,p respectively)
  r = vals(ID_DENS) + hy_dens_int(k)
  u = vals(ID_UMOM) / r
  w = vals(ID_WMOM) / r
  t = ( vals(ID_RHOT) + hy_dens_theta_int(k) ) / r
  p = C0*(r*t)**gamma - hy_pressure_int(k)
  ...
Refactoring Fortran Loops

1. Identify an important loop nest that can be run in parallel.
2. Replace existing loops with do concurrent loops
   Note: Multiple loop iteration variables can be used in the same do concurrent loop, if they are all legal to parallelize
3. Add local clause for variables that must be privatized for correctness.
4. Recompile with -stdpar and test for correctness.
   Note 1: Only refactor one loop nest at a time to ensure errors aren’t introduced, such as forgetting to localize a variable.
   Note 2: Performance may get worse at first due to increased memory migration.
5. Increase the number of concurrent loops to run more work in parallel and reduce memory migration on GPU.

!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,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
    s = 1 , sten_size
    stencil(s) = state(i,k-hs-ll+1,s,1)
  enddo
  !Fourth-order-accurate interpolation of the state
enddo
!Compute density, u-wind, w-wind, potential !temperature, and pressure (r,u,w,t,p respectively)
  r = vals(ID_DENS) + hy_dens_int(k)
  u = vals(ID_UMOM) / r
  w = vals(ID_WMOM) / r
  t = ( vals(ID_RHOT) + hy_dens_theta_int(k) ) / r
  p = C0*(r**gamma - hy_pressure_int(k))
  ...
enddo

!Code from MiniWeather mini-app, trimmed for space.

...
Other Examples

- Bristol BabelStream
  - Modern Fortran implementation of BabelStream.
  - Preprint available upon request.
- NWChem TCE CCSD(T) kernels
  - 6D = 4D x 4D tensor contractions from quantum chemistry with different memory access patterns.
  - https://github.com/jeffhammond/nwchem-tce-triples-kernels
- Parallel Research Kernels (PRK)
  - Shows simple patterns implemented in 50+ different programming languages x models, including Fortran StdPar, OpenACC, OpenMP, etc.
  - https://github.com/ParRes/Kernels/
- GPU Gearbox
  - Based on PRK codes
Fortran Future Features?
Parallelism in Fortran 2018

! fine-grain parallelism

! explicit
do concurrent (i=1:n)
    Z(i) = X(i) + Y(i)
end do

! implicit
MATMUL
TRANSPOSE
RESHAPE
...

! coarse-grain parallelism

np = num_images()
n_local = n / np

! X, Y, Z are coarrays
do i=1,n_local
    Z(i) = X(i) + Y(i)
end do
sync all
Do Concurrent Locality Specifiers and Atomics (2023)

! Scalar reduction – probably implemented with privatization

do concurrent (i=1:n) reduce(Z:+)
   Z = X(i) / Y(i)
end do

! Array reduction – probably implemented with atomics

do concurrent (i=1:n) reduce(Z:+)
   Z(i) = Z(i) + decision(i)
end do
What if we need to use the result?

! Inserting into an array

offset = 1

do concurrent (i=1:n) shared(X,offset) local(s,stuff)
  stuff = ..
  s = size(stuff)
  !$omp/acc atomic capture
  j = offset
  offset = offset + size
  !$omp/acc end atomic capture
  X(j:j+size) = stuff
end do
What if we need to use the result?

! Inserting into an array
offset = 1
do concurrent (i=1:n) shared(X,offset) local(s,stuff)
    stuff = ..
    s = size(stuff)
    call atomic_fetch_add(offset, size, j) ! coarrays
    X(j:j+size) = stuff
end do
What if we need to use the result?

! Inserting into an array
offset = 1

do concurrent (i=1:n) shared(X) local(s,stuff) fetched(offset:+)
  stuff = ..
  s = size(stuff)
  j = offset = offset + s ! Syntax to be determined later
  X(j:j+size) = stuff
end do
Asynchrony
Motivation for Asynchrony 1

! sequential
call my_input(X,Y)

! parallel
do concurrent (i=1:n)
    Z(i) = X(i) + Y(i)
end do

! sequential
call my_output(Z)
Motivation for Asynchrony 1

! sequential
call my_input(X,Y)

! parallel
do concurrent (i=1:n)
   
   Z(i) = X(i) + Y(i)

end do

! sequential
call my_unrelated(A)
Motivation for Asynchrony 1

! sequential on CPU
call my_input(X,Y)

! parallel on GPU
do concurrent (i=1:n)
    Z(i) = X(i) + Y(i)
end do

! sequential on CPU
call my_unrelated(A)
Motivation for Asynchrony 1

! sequential on CPU
call my_input(X,Y)

! parallel on GPU w/ async
do concurrent (i=1:n)
   Z(i) = X(i) + Y(i)
end do

! sequential on CPU w/ async
call my_unrelated(A)
Motivation for Asynchrony 2 (synthetic)

call sub1(IN=A,OUT=B)
call sub2(IN=C,OUT=D)
call sub3(IN=E,OUT=F)
call sub4(IN=B,IN=D,OUT=G)
call sub5(IN=F,IN=G,OUT=H)
! 5 steps require only 3 phases

Fortran compilers may be able to prove these procedures are independent but it is often impossible to prove that executing them in parallel is profitable.
Motivation for Asynchrony 2 (realistic)

Figure 3. The directed acyclic graph (DAG) representing data dependencies within one formulation of the CCSD method. The vertex labels are not important.

https://dl.acm.org/doi/10.1145/2425676.2425687
https://pubs.acs.org/doi/abs/10.1021/ct100584w
Both of the popular directive-based models for parallel computing support asynchronous tasks in a range of operations.

OpenACC supports async and wait, with an implicit/default queue (stream) as well as explicit/numbered queues, and the ability to create dependency chains between operations, similar to CUDA streams.

OpenMP supports tasks with dependencies (and without). The syntax for dependencies is finer granularity - based on data references rather than queues - and the implementation may end up using a global queue as a result.

There are merits to both approaches, so the Fortran community will have to think about what form should be standardized.
These are examples of different things. Please don’t try to compare them.

**Prior Art in OpenMP and OpenACC**

```plaintext
do i=1,n
    !$acc parallel loop async(i)
    do j=1,m
        ...
    enddo
enddo

do i=1,n
    !$acc parallel loop async(i)
    do j=1,m
        ...
    enddo
enddo

!$omp parallel
!$omp master
do j=1,n
    do i=1,m
        !$omp task
        !$omp& depend(in:grid(i-1)) &
        !$omp& depend(out:grid(j))
        ...
        !$omp end task
    enddo
enddo

!$acc wait
 e.g. https://github.com/ParRes/Kernels/blob/default/FORTRAN/p2p-tasks-openmp.F90
```
Example

program main
    use numerot
    real :: A(100), B(100), C(100)
    real :: RA, RB, RC
    A = 1;  B = 1;  C = 1
    RA = yksi(A)
    RB = kaksi(B)
    RC = kolme(C)
    print*, RA+RB+RC
end program main

module numerot
    contains
    pure real function yksi(X)
        real, intent(in) :: X(100)
        !real, intent(out) :: R
        yksi = norm2(X)
    end function yksi
    pure real function kaksi(X)
        real, intent(in) :: X(100)
        kaksi = 2*norm2(X)
    end function kaksi
    pure real function kolme(X)
        real, intent(in) :: X(100)
        kolme = 3*norm2(X)
    end function kolme
end module numerot

https://github.com/jeffhammond/blog/tree/main/CODE
A coarray implementation?

Coarrays are designed to support distributed memory, hence are based on image-private data.

There is limited opportunity for shared-memory optimizations in such codes, as direct inter-image copies will be required.

One of the common motivations for task-based models is dynamic load-balancing, but coarrays provide no mechanism for doing this, so users will have to write their own, which they always do poorly.

program main
  use numerot
  real :: A(100) ! each image has one
  real :: R
  A = 1
  if (num_images().ne.3) STOP
  if (this_image().eq.1) R = yksi(A)
  if (this_image().eq.2) R = kaksi(A)
  if (this_image().eq.3) R = kolme(A)
  sync all
  call co_sum(R)
  if (this_image().eq.1) print*,R
end program main
A do concurrent implementation?

This implementation only supports independent tasks, and is likely completely useless when the implementation uses SIMD lanes or GPU threads for DO CONCURRENT (DC).

As with coarrays, the if (...eq...) is not scalable to more general examples. Do we want arrays of functions?

Both the coarray and DC are also tedious and error prone, which is a good justification for adding new language features.
What might Fortran tasks look like?

```fortran
do i=1,n
    task block async(i)
    do j=1,m
        ...
    enddo
end task block
endo

task sync all
```

The block mechanism is used for scoping.

Prepending task implies this block scope is also a task, which can execute asynchronously until synchronized.

Important questions:
- Is everything (e.g. I/O) allowed to be in a task?
- How do tasks interact with shared state?
What might Fortran tasks look like?

```fortran
do i=1,n
    task block async(i)
        type :: private
        do j=1,m
            ...
        enddo
    end task block
enddo

task sync all
```

The block mechanism is used for scoping.

Prepending task implies this block scope is also a task.

It is essential to be able to have task-private state, which is already covered by the block feature.
What might Fortran tasks look like?

```fortran
real :: x
do i=1,n
    task block async(i) shared(x)
    type :: private
    do j=1,m
        ...
    enddo
end task block
enddo

task sync all

We also want to be able to describe the intent of data outside of the task, so we could reuse locality specifiers from DO CONCURRENT.

Locality specifiers already match OpenMP syntax, and a related feature in Fortran, so they are likely to be intuitive to Fortran programmers.

Task reductions are supported by OpenMP now, but the concept is tricky.

Atomics would be nice but that’s a big bag of worms.
What might Fortran tasks look like?

```fortran
real :: x

do i=1,n
   task call foo(i,x)
enddo

task wait

do i=1,n
   task call foo(i,x) async(mod(i,2))
enddo

task sync 0
...
task sync 1
```

Calling subroutines as tasks is useful, but they should be pure in order to have reasonable behavior.

The right syntax for this is not obvious, but we can solve that later.
Summary

Fortran has two great ways to write parallel code, but needs a third.

Shared-memory task parallelism is implemented in OpenMP, OpenACC, and in models associated with languages that aren’t Fortran.

Task parallelism allows users to solve new types of problems and make better use of existing parallel features, especially DO CONCURRENT (e.g. when executing on GPUs).

Fortran tasks make new things possible and obviate the need for tedious and error prone implementations. They also reduce the need for non-standard extensions like OpenMP and OpenACC.

Please do not let whatever you don’t like about my syntax to get in the way 😊
J3/WG5 papers targeting Fortran 2026


https://j3-fortran.org/doc/year/22/22-169.pdf Fortran asynchronous tasks