GPU PROGRAMMING WITH CUDA

An Introduction to CUDA Fortran

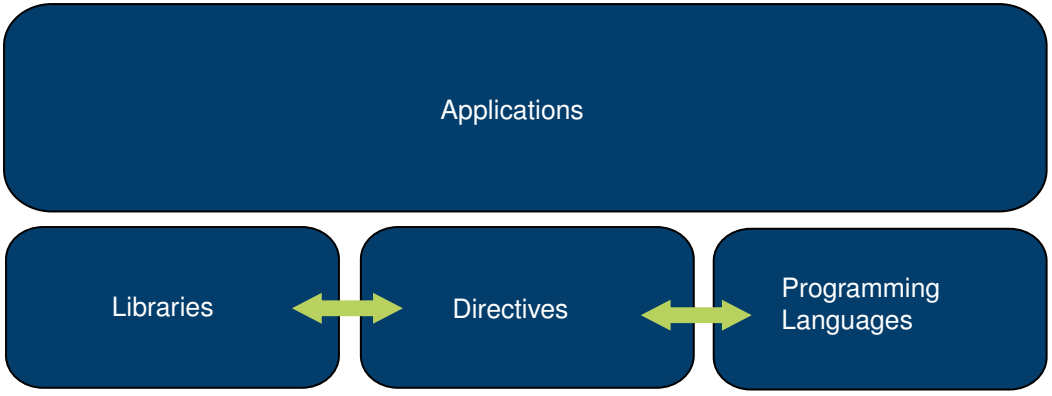
June 22, 2023 | Kaveh Haghighi Mood | JSC

JÜLICH
Forschungszentrum

OVERVIEW

- Introduction
- CUDA Fortran basics
- Kernel loop directives (`CUF` kernels)
- Useful libraries
- Drop-in Fortran array intrinsics acceleration with CuTENSOR
- CUDA Fortran Limitations
- ISO standard Fortran + GPUs

ACCELERATION POSSIBILITIES



WHY CUDA FORTRAN?

- GPU support in native Fortran language
- Libraries and directive-based programming models are not flexible enough
- Not so difficult!
- Interoperable with OpenACC and standard language parallelization
- Similar to CUDA C
- CUDA Libraries

FORTRAN VS CUDA FORTRAN

Fortran

```
program testVecAdd
use mathOps
implicit none

 integer, parameter :: N = 40000
 real :: a(N)

 a = 10.0
 call vecAdd(a,1.0)
 print*,"max_diff=", maxval(a-11.0)

end program testVecAdd
```

```
module mathOps
contains

 subroutine vecAdd(a,b)
 implicit none

  real :: a(:)
  real :: b
  integer :: i, n

  n = size(a)
  do i=1,n
    a(i)=a(i)+b
  enddo

 end subroutine vecAdd
end module mathOps
```

FORTRAN VS CUDA FORTRAN

CUDA Fortran

```
program testVecAdd
use mathOps
use cudafor
implicit none

 integer, parameter :: N = 40000
 real :: a(N)
 real,device :: a_d(N)
 integer tBlock, grid

 a = 10.0
 a_d = a
 tBlock = 256
 grid = ceiling(real(N)/tBlock)
 call vecAdd<<<grid,tBlock>>>(a_d,1.0)
 a = a_d
 print*,"max_diff=", maxval(a-11.0)

end program testVecAdd
```

```
module mathOps
contains
 attributes(global) subroutine vecAdd(a,b)
  implicit none

   real :: a(:)
   real,value :: b
   integer :: i, n

   n = size(a)
   i= blockDim%x*(blockIdx%x-1)+threadIdx%x
   if (i=<n) then
     a(i)=a(i)+b
   endif

 end subroutine vecAdd
end module mathOps
```

CUDA FORTRAN BASICS

Data management

- Fortran enabled for CUDA
  - `device` attribute $\longrightarrow$ declare variables in the device memory

    ```
    real,device :: a_d(N)
    ```

  - Standard Fortran array assignment $\longrightarrow$ data copies between host and device + sync

    ```
    a = a_d
    ```

  - Standard Fortran `allocate` and `deallocate` $\longrightarrow$ for both host and device allocations

    ```
    allocate (a(N),a_d(M), b(M,N))
    ```

  - `managed` attribute $\longrightarrow$ declare unified memory arrays

    ```
    real,managed :: a(:)
    ```

- Memory copy functions (cudaMemcpy, cudaMemcpy2D,...) are also available
- Scalars $\longrightarrow$ CUDA runtime responsibility, if passed by value

  ```
  real,value :: b
  ```

CUDA FORTRAN BASICS

Kernel launch

- Fortran enabled for CUDA
  - triple chevron notation:

    ```
    call kernel<<<grid,block[,bytes][,streamid]>>>(arg1,arg2,...)
    ```
  - `attributes(global)` ⟶ mark kernel subroutines
  - `use cudafor` ⟶ CUDA Fortran types (`blockDim%x, blockIdx%x`)
- Similar to CUDA C loops are replaced with bound checks
- Launch parameters can be extended to two and three dimensions with `dim3` derived type:

```
type(dim3) :: gridDim, blockDim

blockDim = dim3(32,32,1)
gridDim = dim3(ceiling(real(NN)/tBlock%x), ceiling(real(NM)/tBlock%y), 1)
call calcKernel<<<gridDim,blockDim>>>(A_dev,Anew_dev)
```

CUDA FORTRAN BASICS

From a subroutine to a CUDA Kernel

Consider a simple subroutine:

```
module mathOps
         contains

         subroutine vecAdd(a,b)
         implicit none

          real :: a(:)
          real :: b
          integer :: i, n

          n = size(a)
          do i=1,n
            a(i)=a(i)+b
          enddo
         end subroutine vecAdd
end module mathOps
```

CUDA FORTRAN BASICS

From a subroutine to a CUDA Kernel

- Step 1: identify the loop and its index

```
do i=1,n
```

- Step 2: replace loop boundaries with if statement

```
if (i=<n) then
   a(i)=a(i)+b
endif
```

- Step 3: calculate the loop index using CUDA Fortran variables

```
i= blockDim%x*(blockIdx%x-1)+threadIdx%x
```

- Step 4: add `global` attribute

```
attributes(global) subroutine vecAdd(a,b)
```

TASK1

The first CUDA Fortran program

In this exercise, we'll scale a vector (array) of single-precision numbers by a scalar.

- Navigate to:

  `07-CUDA_Fortran/exercises/tasks/scale_vector`

- Look at `Instructions.ipynb` for instructions
- Call `source setup.sh` to load the modules of this task into your environment

IMPORTANT NOTES

- `use cudafor` is necessary to use CUDA Fortran types
- The Fortran array notation should be used for simple data transfers not complicated calculations
- Only one device array is allowed on the right hand side. Following statement is not legal: $A = C\_dev + B\_dev$
- CUDA Fortran source code should have .cuf or .CUF extension or you can add "-cuda" to compiler flags

TASK2

Jacobi solver with explicit kernel

- Navigate to:

  `07-CUDA_Fortran/exercises/tasks/jacobi-explicit`

- Look at `Instructions.ipynb` for instructions
- Call `source setup.sh` to load the modules of this task into your environment

ROUTINE QUALIFIERS

In CUDA Fortran you can specify the type of memory you want to use for your data with variable qualifiers.

- default or `attributes(host)` $\longrightarrow$ allocated in the host main memory
- `attributes(global)` $\longrightarrow$ kernel subroutine
- `attributes(device)` $\longrightarrow$ called from a kernel or another device routine
- `attributes(grid_global)` $\longrightarrow$ threads within the grid group are guaranteed to be co-resident. Allows grid sync operations (on cc70 or better)

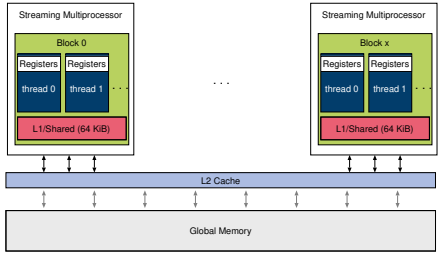
Some notes:

- Device routines should not have variable with `save` attribute
- Device routines should not contain any host routine

VARIABLE QUALIFIERS

In CUDA Fortran you can specify the type of memory you want to use for your data with variable qualifiers.

- default $\longrightarrow$ allocated in the host main memory
- `device` $\longrightarrow$ device global memory
- `managed` $\longrightarrow$ managed memory
- `constant` $\longrightarrow$ device constant memory space, read only for device subprograms
- `shared` $\longrightarrow$ shared memory, only be declared in a kernels or device routines
- `pinned` $\longrightarrow$ allocated in host pagelocked memory



JÜLICH Forschungszentrum

ROUTINE QUALIFIERS

In CUDA Fortran you can specify the type of memory you want to use for your data with variable qualifiers.

- default or `attributes(host)` $\longrightarrow$ allocated in the host main memory
- `attributes(global)` $\longrightarrow$ kernel subroutine
- `attributes(device)` $\longrightarrow$ called from a kernel or another device routine
- `attributes(grid_global)` $\longrightarrow$ threads within the grid group are guaranteed to be co-resident. Allows grid sync operations (on cc70 or better)

Some notes:

- Device routines should not have variable with `save` attribute
- Device routines should not contain any host routine

CUDA STREAMS IN CUDA FORTRAN

Recap from CUDA cource part 1

- Cuda Streams are work queues to express concurrency between different tasks, e.g.
  - Host to device memory copies
  - Device to host memory copies
  - Kernel execution

- To overlap different tasks just launch them in different streams
  - All tasks launched into the same stream are executed in order
  - Tasks launched into different streams might execute concurrently (depending on available resoucres: copy engines, compute resources)
- Kernel launches are always asynchronous
- The default (NULL) stream is used
- The default (NULL) stream waits for work in all other streams which do not have the cudaStreamNonBlocking flag set

CUDA STREAMS

How to use

- Create / destroy a stream

```
integer(kind=cuda_stream_kind) :: stream
integer                        :: istat

istat = cudaStreamCreate(stream2)
istat = cudaStreamDestroy( stream2 )
```

- Launch

```
call kernel <<<gridSize ,blockSize ,0, stream2 >>>(b_d)
istat = cudaMemcpyAsync(a_d , a, nElements, stream1)
```

- Synchronize

```
istat = cudaStreamSynchronize ( stream )
```

CUDA EVENTS IN CUDA FORTRAN

Recap from CUDA cource part 1

Cuda Events are synchronization markers that can be used to:

- Time asynchronous tasks in streams
- Allow fine grained synchronization within a stream
- Allow inter stream synchronization, e.g. let a stream wait for an event in another stream

CUDA EVENTS

How to use

- Create an event

```
type (cudaEvent) :: startEvent, stopEvent

istat = cudaEventCreate(startEvent)
```

- Record

```
istat = cudaEventRecord(startEvent,stream)
```

- Query

```
istat = cudaEventQuery ( event )
```

- Synchronize

```
istat = cudaEventSynchronize ( event )
```

TASK3

CUDA events, Pinned Host Memory and CUDA Streams

- Navigate to:

  `07-CUDA_Fortran/exercises/tasks/streams_and_events`

- Look at `Instructions.ipynb` for instructions
- Call `source setup.sh` to load the modules of this task into your environment

CUF KERNELS

- To many loops? Reductions? Writing kernels is difficult?
- Compiler can write kernels for you, using `!$CUF` directive:

```
!$cuf kernel do[(n)] <<< grid, block, stream=streamid >>>
  do i=1,N
   do j=1,M
    do k=1,P
      ...
    enddo
   enddo
  enddo
```

CUF KERNELS

- Compiler can choose launch parameters, if "$*$" is used
- The `n` parameters after do, denotes the minimum debt of nested loops
- DO loops must have invariant loop limits
- GOTO or EXIT statements are not allowed
- Array syntax are not allowed
- Kernel launch is asynchronous

CUF KERNELS

Explicit Reductions

Since version 21.7 explicit reductions are possible for CUF kernels:

```
!$cuf kernel do <<< *, *>>> reduce(+:value)
 do i=1,N
      ...
```

- Both the `reduce` and `reduction` keywords are accepted
- Supported reductions for integers: +, *, max, min, iand, ior, and ieor
- Supported reductions for real: +, *, max, min
- Supported reductions for complex: +

TASK4

Jacobi solver with kernel loop directives

- Navigate to:

  `07-CUDA_Fortran/exercises/tasks/jacobi-cuf`

- Look at `Instructions.ipynb` for instructions
- Call `source setup.sh` to load the modules of this task into your environment
- Compare the results with the explicit kernel version

USEFUL LIBRARIES INCLUDED IN NVHPC SDK

Why Libraries?

Programming GPUs is easy: No need to reinvent the wheel, use libraries!

USEFUL LIBRARIES INCLUDED IN NVHPC SDK

Why Libraries?

Programming GPUs is easy: No need to reinvent the wheel, use libraries!

- Delivers good performance

USEFUL LIBRARIES INCLUDED IN NVHPC SDK

Why Libraries?

Programming GPUs is easy: No need to reinvent the wheel, use libraries!

- Delivers good performance
- Optimizations for different architectures $\longrightarrow$ Not your responsibility

USEFUL LIBRARIES INCLUDED IN NVHPC SDK

Why Libraries?

Programming GPUs is easy: No need to reinvent the wheel, use libraries!

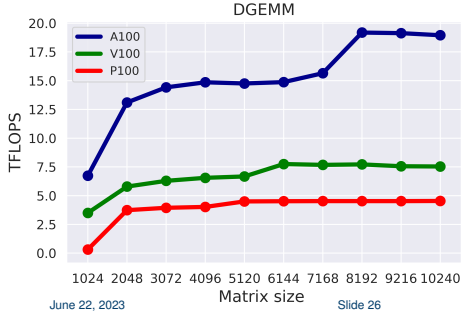
- Delivers good performance
- Optimizations for different architectures $\longrightarrow$ Not your responsibility
- Testing and bug fixing $\longrightarrow$ Not your responsibility

LIBRARIES

cuBLAS

GPU-accelerated implementation of the basic linear algebra subroutines:

- Complete 152 BLAS routines
- Multi-GPU support
- Supports CUDA streams
- Uses Tensor cores if available
- cuSPARSE provides similar functionality for sparse matrices



JÜLICH Forschungszentrum

LIBRARIES

cuBLAS

How to use:
Two interfaces are available `cublas` legacy interface is similar to BLAS `cublas_v2` have more options for example to use streams.

1. Create a handle (only for v2):

```
type(cublasHandle) :: handle
cublasCreate(handle)
```

2. Copy the data to device.
3. Call cuBLAS:

```
cublasSaxpy(n, a, x, incx, y, incy) ! v1
saxpy(n, a, x, incx, y, incy)       ! v1
cublasSaxpy(h, n, a, x, incx, y, incy) v2
```

4. Copy the data back to host:
5. Clean up:

```
cublasDestroy(handle)
```

TASK5

cuBLAS

Matrix-matrix multiplication:

- Location of code: 07-CUDA_Fortran/exercises/tasks/cuBLAS
- Check cuBLAS documentation for details on cublasDgemm()
- For the Fortran interface check URL
- Look at Instructions.ipynb Notebook for instructions
- implement call to double-precision GEMM of cuBLAS with v1 interface

LIBRARIES

cuRAND

CUDA pseudorandom generator:

- Host and device random number generator
- Nine type of generators include Mersenne Twister, Xorshift and MRG
- Supports Uniform, Poisson and Normal distributions

How to use:

1. Create a new generator:

```
type(curandGenerator) :: g
istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_DEFAULT)
```

2. Set the generator options (seed, offset, ...):

```
istat = curandSetPseudoRandomGeneratorSeed(g,1648195)
```

3. Generate random numbers:

```
istat = curandGenerateUniformDouble(g, x_d, sampleSize)
```

4. Clean up:

```
istat = curandDestroyGenerator(g)
```

TASK6

Monte Carlo integration with cuRAND

In this exercise, we'll use cuRAND to generate random numbers for Monte Carlo integration:

$$I = \int_a^b f(x)dx = \lim_{N\to\infty} \frac{b-a}{N} \sum_{i=1}^{N} f(x_i). \tag{1}$$

Crude MC integration algorithm:

1. Draw the sample out of the uniform distribution
2. Calculate function values with the sample
3. Estimate the results $I_{es} = \frac{b-a}{N} \sum_{i=1}^{N} f(x_i)$

TASK6

Monte Carlo integration with cuRAND

- Navigate to:

  `07-CUDA_Fortran/exercises/tasks/cuRAND`

- Look at `Instructions.ipynb` for instructions
- See the cuRAND documentation for further information: URL
- For the Fortran interface check URL
- Call `source setup.sh` to load the modules of this task into your environment

LIBRARIES

cuFFT

GPU-accelerated implementation of FFT:

- 1D, 2D, 3D transforms
- Half, single and double precision, complex and real data types support
- Supports CUDA streams
- Batch execution
- FFTW like API
- Multi GPU support

LIBRARIES

cuFFT

How to use:

1. Create a plan:

```
ierr = cufftPlan1D(plan, L, CUFFT_D2Z,1)
```

2. Copy the data to device
3. Call cuFFT:

```
ierr = cufftExecD2Z(plan, S, ST)
```

4. Copy the data back to host
5. Clean up:

```
cufftDestroy(plan)
```

TASK7

cuFFT

Use cuFFT to find the components of a signal:

- Location of code: 07-CUDA_Fortran/exercises/tasks/cuFFT
- Check cuFFT documentation for details on `cufftPlan1d` and `cufftExecD2Z`
- For the Fortran interface check URL
- Look at Instructions.ipynb notebook for instructions
- implement call to cuFFT to perform a real to complex Fourier transformation
- use Instructions.ipynb notebook to visualize results

LIBRARIES

cuSOLVER

Collection of dense and sparse direct linear solvers and Eigen solvers:

- Similar to LAPACK
- Both dence and sparse solvers
- Multi GPU support
- Tensor core support

LIBRARIES

cuTENSOR

Tensor Linear Algebra on GPUs:

- Direct Tensor Contraction
- Reduction
- Elementwise Operations
- Mixed precision support
- Multi GPU support
- High level interfaces (cutensorex) for Fortran intrinsic array functions (see next section)

FORTRAN ARRAY INTRINSICS WITH CUTENSOR

- `nvfortran` compiler can map Fortran intrinsic to to CuTENSOR
- Close to zero efforts acceleration for intrinsic functions like `matmul`, `transpose`, `reshape` functions!
- Just add `use cutensorex` and recompile with `-cudalib=cutensor` !

TASK8

Fortran array intrinsics using Tensor Cores

- Navigate to:

  `07-CUDA_Fortran/exercises/tasks/matmul-cutensor`

- Look at `Instructions.ipynb` for instructions
- Call `source setup.sh` to load the modules of this task into your environment
- Compare the calculation time with and without the cuTENSOR

TASK8

Fortran array intrinsics using Tensor Cores

Results on JUWELS Booster (gflops):

Size	8192	16384
Naive CUDA shared mem implementation	1945	2205
cuTENSOREX	16083	16435

CUDA FORTRAN LIMITATIONS

CUDA Fortran is the most mature programming model for accelerating Fortran codes. Still, there are some limitations:

- Not portable! You have to use Nvidia GPUs
- Supported only by Nvidia HPC SDK (formerly known as PGI) and IBM XL Fortran compilers
- For some CUDA libraries, you have to write interfaces
- Small community

ISO STANDARD FORTRAN + GPUS!

- Non-standard libraries, directives or language extensions are not attractive enough?
- Standard portable acceleration is possible now!
- Fortran 2008 `DO CONCURRENT` supported by *nvfortran*:

```
subroutine vecAdd(a,b)
implicit none

real :: a(:)
real :: b
integer :: i, n

n = size(a)
do i=1,n
   a(i)=a(i)+b
enddo

end subroutine vecAdd
```

```
subroutine vecAdd(a,b)
implicit none

real :: a(:)
real :: b
integer :: i, n

n = size(a)
do concurrent (i = 1: n)
  a(i)=a(i)+b
enddo

end subroutine vecAdd
```

ISO STANDARD FORTRAN ON GPUS!

- Correctness? $\longrightarrow$ You are responsible
- Data transfer? $\longrightarrow$ Compiler and runtime env
- Additional `-stdpar` compilation flag is necessary
  - `-stdpar=multicore` $\longrightarrow$ compiles for CPU
  - `-stdpar=gpu,multicore` $\longrightarrow$ compiles for GPU or CPU

ISO STANDARD FORTRAN ON GPUS!

- Nested loop example:

```
do i = 1, n
 do j =1,m
  C(i,j)=a(i)+b(j)
 enddo
enddo
```

```
do concurrent (i = 1: n, j=1: m)
  C(i,j)=a(i)+b(j)
enddo
```

- Data privatization:

```
DO CONCURRENT (...) [locality-spec]
```

`locality-spec` options:

```
local(list)
local_init(list)
share(list)
```

ISO STANDARD FORTRAN TASK

Jacobi solver with do concurrent

- Navigate to:

  `07-CUDA_Fortran/exercises/tasks/jacobi-std`

- Look at `Instructions.ipynb` for instructions
- Call `source setup.sh` to load the modules of this task into your environment
- Compare the results with the explicit and CUF kernel versions

CONCLUSION

- CUDA Fortran is a mature and powerful programming model for accelerating Fortran codes.
- The language semantics is close to standard Fortran, making it easy to use for Fortran developers.
- Development tools and libraries in Nvidia HPC SDK make the porting process more straightforward.
- The compiler can write efficient accelerated routines using `CUF` kernels.
- ISO standard Fortran can be executed efficiently on GPU using `nvfortran` compiler.

RESOURCES

- CUDA Fortran for Scientists and Engineers by Ruetsch and Fatica 2013
- CUDA Fortran Porting Guide
- CUDA Fortran Programming Guide and Reference
- Examples:
  `NVHPC-INSTALLDIR/arch/version/examples`

RESOURCES

- CUDA Fortran for Scientists and Engineers by Ruetsch and Fatica 2013
- CUDA Fortran Porting Guide
- CUDA Fortran Programming Guide and Reference
- Examples:
  `NVHPC-INSTALLDIR/arch/version/examples`

Thank you for your attention!