AMD

GPUFORT

A source-to-source translator for Fortran accelerator dialects

AMD

Motivation

- Porting to new accelerator architectures costs time and money
  - Especially if a new programming language needs to be adopted
- Developers that can do such porting are difficult to find
- However, more often than not it is the only way to achieve good performance on novel architectures
- GPUFORT is a tool that aims to reduce the costs of porting Fortran applications from one accelerator language to the other

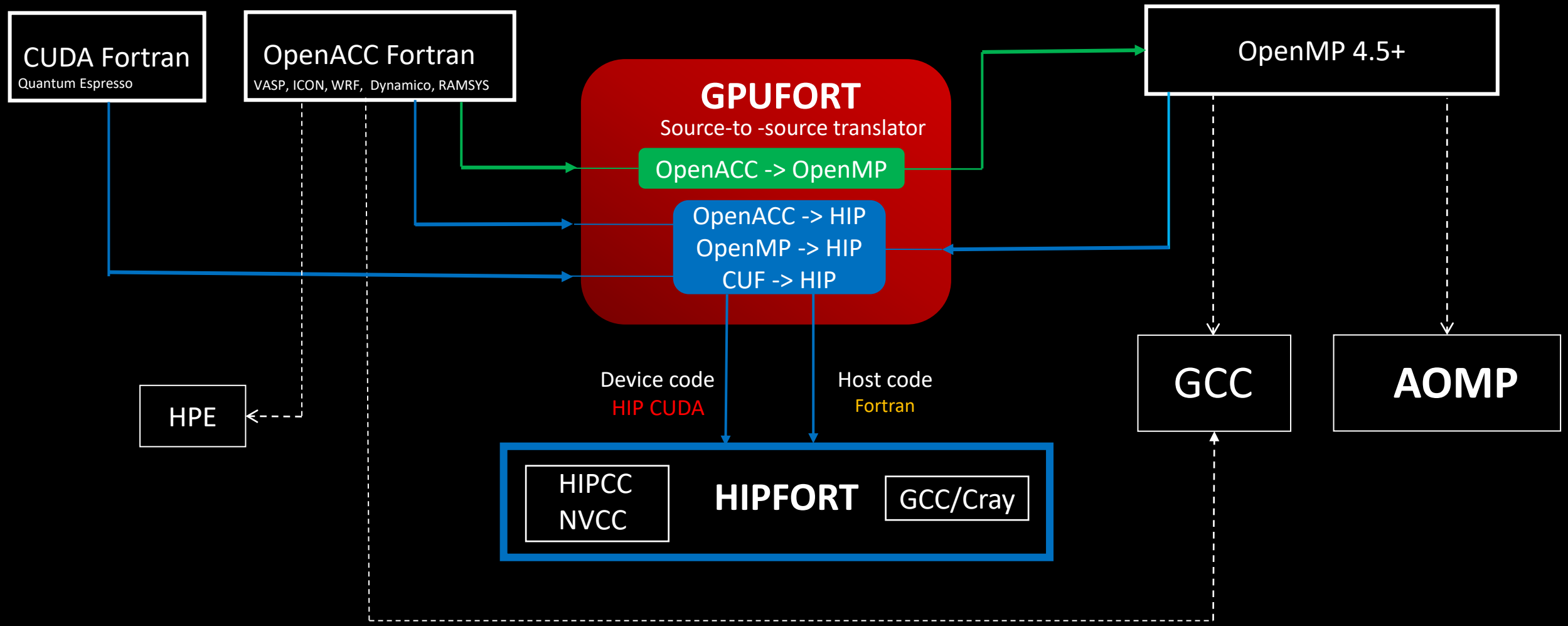
AMD

Outline

- **GPUFORT Goals and Design**
- CUDA Fortran & OpenACC Examples
- Components
- Current Limitations
- Success Story
- Summary & Next Steps

AMD

GPUFORT



AMD

Remarks

- All translation paths we support/aim to support result in code that compiles (at least for) AMD and NVIDIA devices
- Many of the applications we classify as OpenACC applications use “vendor-specific standard extensions” such as *acc_get_cuda_stream*; e.g. to use CUDA Fortran math modules
  - A “standard OpenACC compiler without vendor-specific extensions” cannot compile these applications

AMD

Goals of the project

- Develop a translator / productivity tool that aids HPC developers in translating a whole codebase from one Fortran accelerator dialect to the other
- Make the tool's actions transparent and allow manual intervention, customization, & code generation output optimization
- Anticipate + support debugging & benchmarking needs of HPC developers
- Make the tool as lightweight as possible:
  - no complicated build process/long list of dependencies/heavy UI

AMD

Design decisions

- Specifically for Fortran (might extend it to C)
  - Many popular HPC codes are written in CUDA Fortran **and/or** OpenACC Fortran
- Written in Python 3
  - Very expressive, cross platform & no compilation-latency
  - Not (that) slow and will be 2x-5x faster [G.v.Rossum]
  - Third-party packages for fast IO, code generation, parsing
- Allow customizations and output styling
  - It is tuneable and scriptable (many options, config files are in Python 3)
  - And again: It is all Python 3
- Ease usage, debugging, and developing:
  - Generate domain-specific debug code
  - Lots of logging with levels and filters
  - Allow inspection of intermediate output (symbol tables, kernel code)
  - Allow adoption/extension of toolchain

AMD

Outline

- GPUFORT Goals and Design
- **CUDA Fortran & OpenACC Examples**
- Components
- Current Limitations
- Success Story
- Summary & Next Steps

AMD

OpenACC Example (1/2)

```
program main
  integer, parameter :: N = 1000
  integer :: i
  integer(4) :: x(N), y(N),
y_exact(N)

  do i = 1, N
    y_exact(i) = 3
  end do

  !$acc data create(x(1:N),y(1:N))

  !$acc parallel loop
  do i = 1, N
    x(i) = 1
    y(i) = 2
  end do                                  <1/2>
```

```
  !$acc parallel loop
  do i = 1, N
    y(i) = x(i) + y(i)
  end do

  !$acc end data

  do i = 1, N
    if ( y_exact(i) .ne.&
          y(i) ) ERROR STOP "GPU and CPU result do not match"
  end do
  print *, "PASSED"
end program                               <2/2>
```

AMD

OpenACC Example (2/2)

```
program main
  integer, parameter :: N = 1000
  integer :: i
  integer(4) :: x(N), y(N), y_exact(N)

  do i = 1, N
    y_exact(i) = 3
  end do

#ifdef __GPUFORT
  !$omp target data map(alloc:x(1:N),y(1:N))

  !$omp target teams distribute parallel do
  do i = 1, N
    x(i) = 1
    y(i) = 2
  end do

  !$omp target teams distribute parallel do
  do i = 1, N
    y(i) = x(i) + y(i)
  end do
  !$omp end target data
```

<1/2>

```
#else
  !$acc data create(x(1:N),y(1:N))

  !$acc parallel loop
  do i = 1, N
    x(i) = 1
    y(i) = 2
  end do

  !$acc parallel loop
  do i = 1, N
    y(i) = x(i) + y(i)
  end do
  !$acc end data
#endif

  do i = 1, N
    if ( y_exact(i) .ne.&
         y(i) ) ERROR STOP "GPU and CPU result do not match"
  end do

  print *, "PASSED"
end program
```

<2/2>

- *gpufort –w <file> –E omp*

  -w: #ifdef __GPUFORT

- **OpenMP generated**
- **OpenACC original**

AMD

CUDA Fortran Example (1/3): Input File

```
program main
 use cudafor
 implicit none
 integer, parameter :: N = 40000
 real :: x(N), y(N), a
 real, device, allocatable :: x_d(:)
 real, allocatable :: y_d(:)
 type(dim3) :: grid, tBlock
 integer :: I
 !
 attributes(device) :: y_d
 tBlock = dim3(256,1,1)
 grid = dim3(ceiling(&
   real(N)/tBlock%x),1,1)
```

<1/2>

```
 allocate(x_d(N),y_d(N))

  x = 1.0; y = 2.0; a = 2.0
 x_d = x
 y_d = y
#define xi x_d(i)
 !$cuf kernel do(1) <<<grid, tBlock>>>
 do i=1,size(y_d,1)
  y_d(i) = y_d(i) + a*xi
 end do
 y = y_d
 deallocate(x_d,y_d)
  write(*,*) 'Max error: ', maxval(abs(y-4.0))
end program main
```

<2/2>

- CUDA Fortran module
- device variables
- device malloc/free
- host<->device copies
- offloaded loop

[similar to: nvidia.com/blog/easy-introduction-cuda-fortran/]

AMD

CUDA Fortran Example (2/3): S2S Translation

```
#ifdef __GPUFORT
#include "vector-add.f90-fort2hip.f08"
#endif
program main
#ifdef __GPUFORT
  use main_fort2hip
  use hipfort
  use hipfort_check
#else
  use cudafor
#endif
  implicit none
  integer, parameter :: N = 40000
  real :: x(N), y(N), a
#ifdef __GPUFORT
  real,pointer,dimension(:) :: x_d
  real,pointer,dimension(:) :: y_d
#else
  real, device, allocatable :: x_d(:)
  real, allocatable :: y_d(:)
#endif
  type(dim3) :: grid, tBlock
  integer :: i

#ifndef __GPUFORT
  attributes(device) :: y_d
#endif                                  <1/3>
```

```
  tBlock = dim3(256,1,1)
  grid = dim3(ceiling(real(N)/tBlock%x),1,1)

#ifdef __GPUFORT
  call hipCheck(hipMalloc(x_d, N))
  call hipCheck(hipMalloc(y_d, N))
#else
  allocate(x_d(N),y_d(N))
#endif

  x = 1.0; y = 2.0; a = 2.0
#ifdef __GPUFORT
  call hipCheck(hipMemcpy(x_d, x, hipMemcpyHostToDevice))
  call hipCheck(hipMemcpy(y_d, y,hipMemcpyHostToDevice))
#else
  x_d = x
  y_d = y
#endif

#define xi x_d(i)                       <2/3>
```

```
#ifdef __GPUFORT
  ! extracted to HIP C++ file
  call launch_main_26_e28c45(grid,tBlock,0,c_null_ptr,c_loc(y_d),&
     size(y_d,1),lbound(y_d,1),a,c_loc(x_d),size(x_d,1),lbound(x_d,1))

  call hipCheck(hipMemcpy(y, y_d, hipMemcpyDeviceToHost))

  call hipCheck(hipFree(x_d))
  call hipCheck(hipFree(y_d))
#else
  !$cuf kernel do(1) <<<grid, tBlock>>>
  do i=1,size(y_d,1)
    y_d(i) = y_d(i) + a*xi
  end do

  y = y_d

  deallocate(x_d,y_d)
#endif
  write(*,*) 'Max error: ', maxval(abs(y-4.0))
end program main                        <3/3>
```

- ***gpufort –w –E hip vector-add.f90*** (#ifdef's via ***–w*** *are* optional)
- All Fortran files can be compiled *hipfc* (hipFORT) , *gfortran*, or *cray*
- HIP C++ kernel implementation must be compiled via *hipcc*

- CUDA Fortran module
- device variables
- device malloc/free
- host<->device copies
- offloaded loop

AMD

CUDA Fortran Example (3/3): HIP C++ kernel

```
#include “gpufort.h
//…
// BEGIN main_26_e28c45
/*
  HIP C++ implementation of the function/loop body of:

   !$cuf kernel do(1) <<<grid, tBlock>>>
   do i=1,size(y_d,1)
     y_d(i) = y_d(i) + a*xi
   end do
*/
__global__ void  main_26_e28c45(
   float * __restrict__ y_d,
   const int y_d_n1,
   const int y_d_lb1,
   float a,
   float * __restrict__ x_d,
   const int x_d_n1,
   const int x_d_lb1) {
 #undef _idx_y_d
 #define _idx_y_d(a) ((a-(y_d_lb1)))
 #undef _idx_x_d
 #define _idx_x_d(a) ((a-(x_d_lb1)))
 int i = 1 + (1)*(threadIdx.x + blockIdx.x * blockDim.x);
 if (loop_cond(i,y_d_n1,1)) {
  y_d[_idx_y_d(i)]=(y_d[_idx_y_d(i)]+a*x_d[_idx_x_d(i)]);
 }
}
```

```
extern "C" void launch_main_26_e28c45(
   dim3* grid,
   dim3* block,
   const int sharedmem,
   hipStream_t stream,
   float * __restrict__ y_d,
   const int y_d_n1,
   const int y_d_lb1,
   float a,
   float * __restrict__ x_d,
   const int x_d_n1,
   const int x_d_lb1) {

 // launch kernel
 hipLaunchKernelGGL((main_26_e28c45), *grid, *block, sharedmem, stream,
         y_d,y_d_n1,y_d_lb1,a,x_d,x_d_n1,x_d_lb1);
}
```

- Generates per loop nest: kernel & variety of kernel launchers
- Kernel name consists of name of parent module/program/procedure, line number and a hash code
  - Hash code encodes non-whitespace characters in Fortran original; can be used to detect significant code changes

AMD

Outline

- GPUFORT Goals and Design
- CUDA Fortran & OpenACC Examples
- **Components**
- Current Limitations
- Success Story
- Summary & Next Steps

AMD

Requirements/tasks for a translator for …

CUDA Fortran

✓CUDA-like runtime and math libraries (https://github.com/ROCmSoftwarePlatform/hipfort)

- Detect & translate CUDA Fortran intrinsics & lang extensions to hipFORT routines and standard Fortran types
- Detect, analyze & translate:
  - CUDA Fortran accelerator routines
  - CUDA Fortran *!$cuf* directives

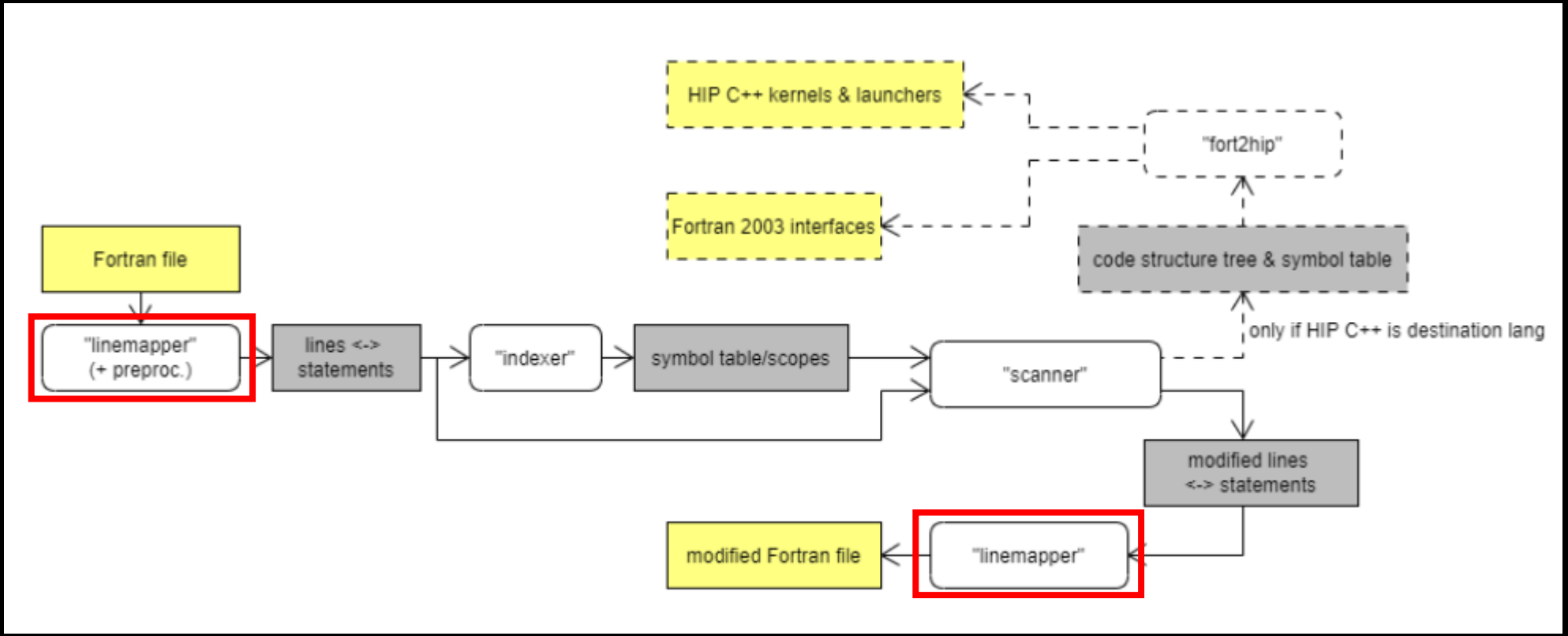
OpenACC Fortran

✓**(only Fortran + HIP C++)**
A runtime for managing array/ derived type mappings, async queues and exposing OpenACC runtime API (GNU LIBGOMP)

- Detect & translate OpenACC directives that result in runtime calls (*!$acc update, !$acc data, …)*
- Detect, analyze & translate:
  - OpenACC accelerator routines
  - OpenACC !$acc parallel and !$acc kernels construct

AMD

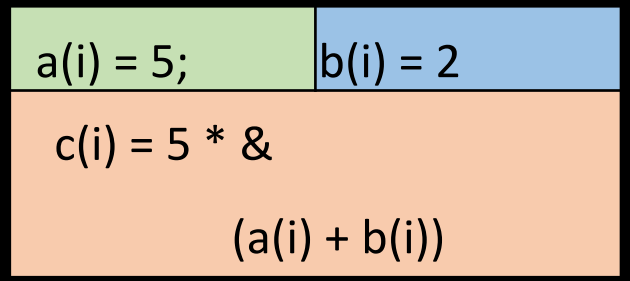
Architecture



AMD

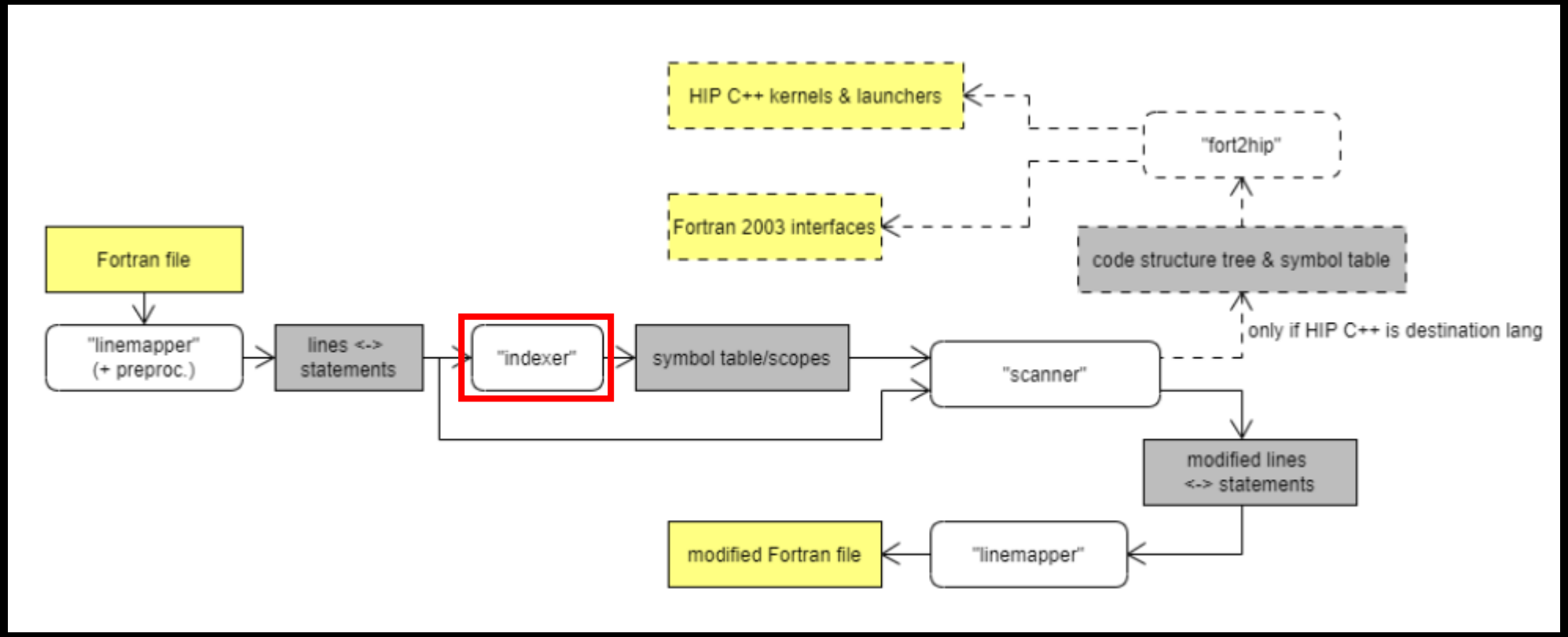
Components: “linemapper”

- Single lines can contain multiple statements in Fortran (**green** and **blue**) or spread over multiple lines (**orange**)
- “linemapper” maps code lines <-> statements
  - Maps also allow to append/prepend epilog/prolog
- Has C-style preprocessor to filter lines & evaluate macros
- Other toolchain components use/modify linemaps
  - Code generation works with the preprocessed statements
  - S2S translation modifies the maps and flags them
- “linemapper” then modifies file according to flagged maps



AMD

Architecture



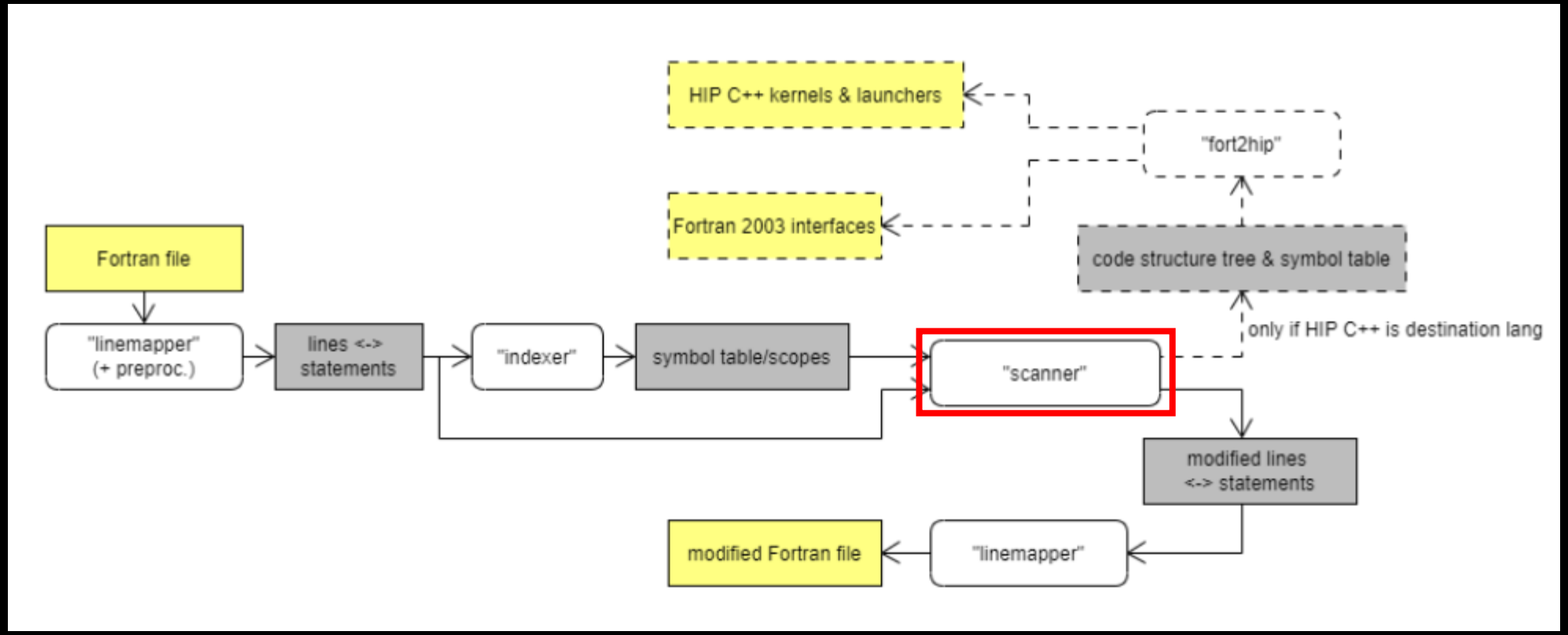
AMD

Components: “indexer”

- Creates GPUFORT *mod* file (json) per module/program/subprogram:
  - Collects info on modules, programs, procedures, derived types & variables
  - No dependency on particular compiler and their mod file format
- Can create scopes for modules, programs, procedures
  - Resolves module and parameter dependencies
- Variable records contain info on type & kind, rank, name, … plus
  - CUDA Fortran qualifiers such as *device*, *managed*, *pinned*, …
  - If variables are subject to an OpenACC *declare* directive
- Subprogram records contain arguments, result name, … plus
  - CUDA Fortran procedure modifiers such as *device/host/host,device/global*
  - If subprogram is subject to an OpenACC *routine* directive

AMD

Architecture



AMD

Components: “scanner”

```
integer  parameter :: N = 10
integer :: i
integer(4) :: x(N), y(N), y_exact(N)

y_exact(:) = 3
!$acc data copy(x(1:N),y(1:N))
!$acc parallel loop
do i = 1, N
  x(i) = 1
  y(i) = 2
end do

!$acc parallel
!$acc loop
do i = 1, N
  y(i) = x(i) + y(i)
end do
!$acc end parallel
!$acc end data
```

- Root
  - STProgram
    - STDeclaration
    - STDeclaration
    - STDeclaration
    - STAccDirective
    - STLoopKernel
    - STLoopKernel
    - STAccDirective

- Iterates statements and puts interesting blocks into a tree
- Avoids full parses of statements during tree creation (cheap) and records only statements meaningful for offloading / compute (reduced complexity)
- Calls “translator” backend on identified blocks in source modification / code generation phase for in-depth analysis and transformation of marked code blocks

AMD

Parsing

- Strategy is to use fast standard string comparison and *re* module; e.g. for marking/filtering in “scanner” and “indexer” ...
  - *re.split((<tokens>),text,flags)* for tokenizing
  - *re.compile/re.sub* for simple pattern matching
- … and more sophisticated parsing tools for complicated expressions such as arithmetic & logical expression, assignments, function calls, …
  - Tool of choice is *pyparsing*
- **Remark**: The frontend is not the hard part, the backend is!

AMD

On pyparsing (1): Syntax trees

```
# !/usr/bin/env python3
import pyparsing as pp
# grammar
rvalue = pp.pyparsing_common.identifier
op     = pp.Literal("+")
expr   = rvalue + op + rvalue
# test
print(expr.parseString("a + b"))
# output:  ['a','+','b']
```

- *pyparsing* is a quick way to construct and execute grammars directly in python [github.com/pyparsing/pyparsing]; e.g. used by Red Hat Ansible
- Usage of *parse actions* allows you to generate an abstract syntax tree from a string expression
  - AST structure is as you define in *__init__*
- *pyparsing* comes with some great helpers …

AMD

On pyparsing (1): Syntax trees

```
# !/usr/bin/env python3
import pyparsing as pp
# grammar
rvalue = pp.pyparsing_common.identifier
op     = pp.Literal("+")
expr   = rvalue + op + rvalue
class Rvalue():
  def __init__(self,tokens):
    self._value = tokens
class Op():
  def __init__(self,tokens):
    self._op = tokens
rvalue.setParseAction(Rvalue)
op.setParseAction(Op)
print(expr.parseString("a + b"))
# output: [__main__.RValue object ..., __main__.Op object ...,
__main__.RValue object ...]
```

- *pyparsing* is a quick way to construct and execute grammars directly in python [github.com/pyparsing/pyparsing]; e.g. used by Red Hat Ansible
- Usage of *parse actions* allows you to generate an abstract syntax tree from a string expression
  - AST structure is as you define in *__init__*
- *pyparsing* comes with some great helpers …

AMD

On pyparsing (2): Shortcuts

pyparsing.InfixNotation(…)

```
import pyparsing as pyp

number    = pyp.pyparsing_common.number
identifier = pyp.pyparsing_common.identifier
rval       = identifier | number

expr = pyp.infixNotation(rval, [\
   (pyp.oneOf('* /'), 2, pyp.opAssoc.LEFT),\
   (pyp.oneOf('+ -'), 2, pyp.opAssoc.LEFT),\
   (pyp.oneOf('-'), 1, pyp.opAssoc.LEFT),\
]) + pyp.stringEnd()

print(expr.parseString("-5 + 3*(a+1.0)") )
```

pyparsing.Forward()

```
import pyparsing as pyp

identifier = pyp.pyparsing_common.identifier
derived_type_rvalue = pyp.Forward()

derived_type_elem = identifier + pyp.Literal(“%”) +\
    derived_type_rvalue )
derived_type_rvalue <<= derived_type_elem | identifier

print(derived_type_elem.parseString(“a%b") )
print(derived_type_elem.parseString(“a%b%c") )
print(derived_type_elem.parseString(“a%b%c%d“) )
…
```

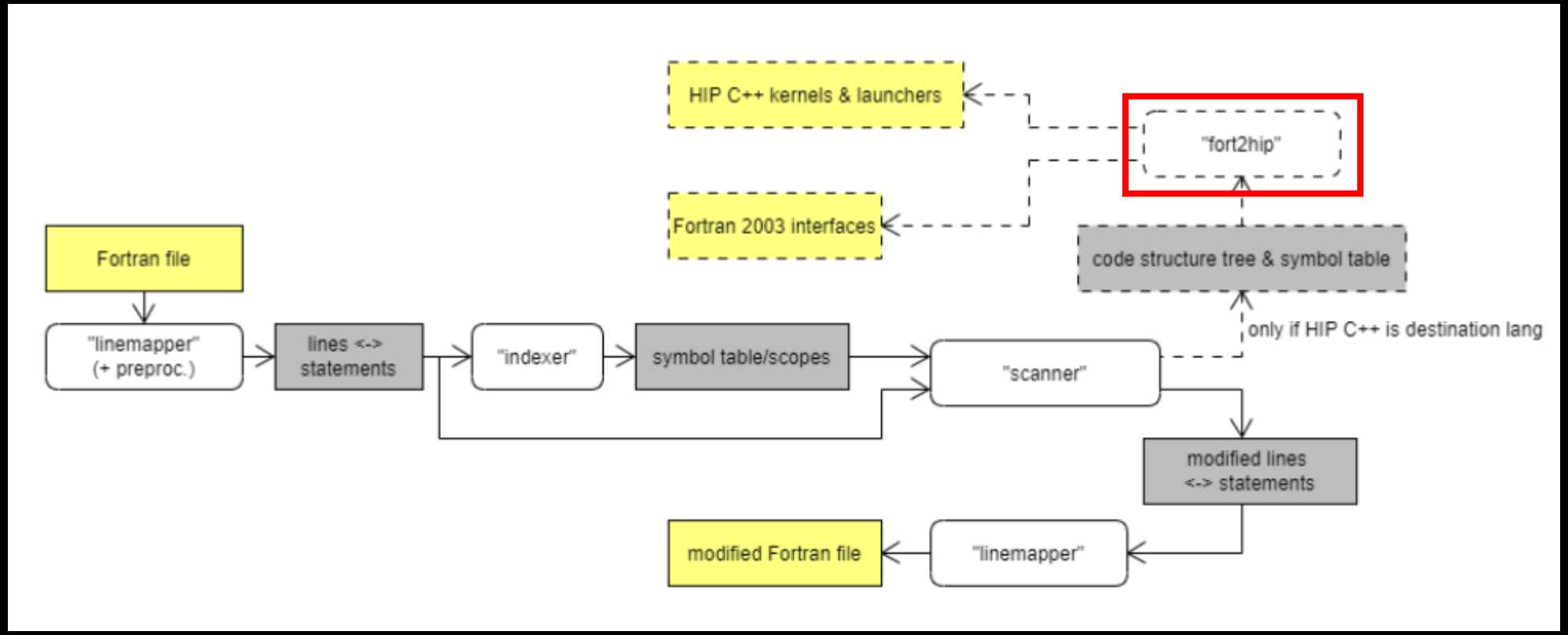
AMD

On pyparsing (3): Lessons learned

- Don’t overuse pyparsing!
  - String/list comparisons and regular expressions can be much faster!
  - pyparsing error messages are difficult to understand:
    - parse parts of flow statements manually
- Don’t express details via grammar!
  - Check details with string comparisons or regular expressions after the initial parsing
    - Bad: *expr = pyp.Literal(“parameter”) | pyp.Literal(“device”) | …*
    - Faster:
      - grammar: *expr = pyp.Regex(r”[a-z]+”)*
      - check in parse action __init__: *if tokens[0] in [“parameter”,”device”,…]*
- Run *pyparsing.ParserElement.enablePackrat()* at the top of your grammar to enable caching of partial results when dealing with nested expressions
  - Makes parsing often much faster!

AMD

Components: “fort2hip”



AMD

Components: “fort2hip”

- Given symbol table and “scanner tree”, “fort2hip” generates HIP C++ kernels, launchers and interfaces to the Fortran code
- Code generation is simplified thanks to *Jinja*
  - “***Jinja*** *is a fast, expressive, extensible templating engine. Special placeholders in the template allow writing code similar to Python syntax.*” [palletsprojects.com/p/jinja/]
  - Widely used by web frameworks
  - Takes Python JSON objects as input
- “fort2hip” relies on HIP C++ header file providing overloaded device math functions macros to facilitate type conversions, complex variable construction, max/min with arbitrary argument list, … (also generated with *Jinja*)

AMD

On Jinja2: Example from hipFORT

```
{% for tuple in datatypes %}{% for dims in dimensions %}{% if dims > 0 %}
{% set size = 'size(dest)*' %}
{% set rank = ',dimension(' + ':,'*(dims-1) + ':)' %}
{% else %}
{% set size = '' %}
{% set rank = '' %}
{% endif %}
{% for size_t in sizeTypes %}
{% set name = 'hipMemcpyAsync_' + tuple[0] + '_' + dims|string  + '_' + size_t -%}
   function {{name}}(dest,src,length,myKind,stream)
    use iso_c_binding
    use hipfort_enums
    use hipfort_types
    implicit none
    {{tuple[2]}},target{{ rank }},intent(inout) :: dest
    {{tuple[2]}},target{{ rank }},intent(in)    :: src
    integer({{size_t}}),intent(in) :: length
    integer(kind(hipMemcpyHostToHost)) :: myKind
    type(c_ptr) :: stream
    integer(kind(hipSuccess)) :: {{name}}
   !
    {{name}} = hipMemcpyAsync_b(c_loc(dest),c_loc(src),length*{{tuple[1]}}_8,myKind,stream)
   end function
{% endfor %}
```

- *Jinja* is also great for generating overloaded interfaces for C libraries (such as the ROCm)
- Example on the left is a (simplified) snippet that generates interfaces for *hipMemcpyAsync* in hipFORT
- Input is a python dict, e.g.:

```
context = {
    “datatypes”: [
          [“r4”,“4”,”real(4)”],
          [“r8”,“8”,”real(8)”],
          ... ],
      “dimensions”: range(0,7+1)
 }
```

AMD

Additional features

- GPUFORT can be configured via CLI (see: *gpufort –help*) or Python 3 config file (see: *gpufort –print-config-defaults*)
- Selected CLI options (also accessible via config file):
  - Extract original loop nests and put them into separate “CPU kernel launcher” routine:
    - *--emit-cpu-impl*
    - Downloads GPU data and then runs original loop on CPU
    - Can be plugged into original application and then gradually replaced by manually tested GPU implementation
    - Great for creating unit tests (in Fortran or C++)
  - Generate debug code into kernel launcher routines (GPU and CPU):
    - *--emit-debug-code*
    - Output kernel (launcher) argument values
    - Download and print GPU array elements, array norms, min and max
    - Great to gather (application specific) input data for unit tests

AMD

Preview: Interoperable arrays and structs (I)

```
! non-interoperable types
type :: node_t
  real :: val
end type

type :: mesh_t
  real :: a
  type(node_t),allocatable  :: x(:)
  real,pointer,dimension(:) :: y
end type
```

- Derived types without the *bind(c)* qualifier are in general non-interoperable with C/C++ codes
  - as the Fortran compiler might decide to reorder members or add some padding bytes, …

AMD

Preview: Interoperable arrays and structs (I)

```
! non-interoperable types
type, bind(c) :: node_t ! works!
  real :: val     !
end type

type :: mesh_t ! why not here?
  real :: a
  type(node_t),allocatable  :: x(:)
  real,pointer,dimension(:) :: y
end type
```

- Derived types without the *bind(c)* qualifier are in general non-interoperable with C/C++ codes
  - as the Fortran compiler might decide to reorder members or add some padding bytes, …
- One can force the same data layout as in a C struct by adding the *bind(c) keyword*

AMD

Preview: Interoperable arrays and structs (I)

```
! non-interoperable types
type :: node_t
  real :: val
end type

type :: mesh_t
  real :: a
  type(node_t),allocatable  :: x(:)
  real,pointer,dimension(:) :: y
end type
```

- Derived types without the *bind(c)* qualifier are in general non-interoperable with C/C++ codes
  - as the Fortran compiler might decide to reorder members or add some padding bytes, …
- One can force the same data layout as in a C struct by adding the *bind(c) keyword*
- **Problem: Does not work with pointer or allocable members**

AMD

Preview: Interoperable arrays and structs (II)

```
! original non-interoperable types
type :: node_t
  real :: val
end type

type :: mesh_t
  real :: a
  type(node_t),allocatable  :: x(:)
  real,pointer,dimension(:) :: y
end type
```

```
! interoperable types
type,bind(c) :: node_t_interop
  real(c_float) :: val
end type

type,bind(c) :: mesh_t_interop
  real(c_float)          :: a
  type(gpufort_array1) :: x
  type(gpufort_array1) :: y
end type
```

- Same data layout due to custom array type (*gpufort_array*)
- GPUFORT arrays are **interoperable** derived types that model arrays of a certain rank
- They can wrap existing host and/or device data or allocate host and/or data itself
- On the HIP C++ side, it is equipped with functions that allow to write code that looks very similar to Fortran arithmetic expressions
- It can be used to synchronize data between host and device buffers or to print out device data and norms

AMD

Preview: Interoperable arrays and structs (II)

```
! original non-interoperable types
type :: node_t
  real :: val
end type

type :: mesh_t
  real :: a
  type(node_t),allocatable  :: x(:)
  real,pointer,dimension(:) :: y
end type
```

```
! interoperable types
type,bind(c) :: node_t_interop
  real(c_float) :: val
end type

type,bind(c) :: mesh_t_interop
  real(c_float)          :: a
  type(gpufort_array1) :: x
  type(gpufort_array1) :: y
end type
```

```
// C++ structs with same layout as Fortran interoperable types
struct node_t { float val; };
struct mesh_t {
   float a;
   gpufort::array1<node_t> x;
   gpufort::array1<float> y; };
```

AMD

Preview: Interoperable arrays and structs (III)

- Create interoperable derived type from non-interoperable type
- Scalar values are straightforward
- Arrays of basic datatypes can be mapped via overloaded *gpufort_array_init* function

```
! Copy scalar
mesh_interop%a = mesh_orig%a

! Init & copy basic datatype array
call hipCheck(gpufort_array_init(mesh_interop%y,mesh_orig%y,&
    alloc_mode=gpufort_array_wrap_host_alloc_device,&
    sync_mode=gpufort_array_sync_copyin))
```

AMD

Preview: Interoperable arrays and structs (IV)

- Derived types are more complicated to map as the host pointer cannot be wrapped
- Instead, a new array of interoperable types must be created
- Scalar derived type members must be copied from the original types and members that are arrays must be wrapped as shown on the previous slides

AMD

Preview: Interoperable arrays and structs (V)

```
type(node_t_interop)          :: node_t_interop_dummy
type(node_t_interop),pointer  :: mesh_t_interop_x(:)

! 1. Allocate interop. derived type array on host and device
call hipCheck(gpufort_array_init(mesh_interop%x,&
  int(c_sizeof(node_t_interop_dummy),c_int),&
  shape(mesh_orig%x),&
  lbounds=lbound(mesh_orig%x),&
  alloc_mode=gpufort_array_alloc_pinned_host_alloc_device))
! 2. Get host pointer to interop type array
call c_f_pointer(mesh_interop%x%data%data_host,&
  mesh_t_interop_x,shape=shape(mesh_orig%x))
! 3. Copy from non-interop. type to interop. type
mesh_t_interop_x(:)%val = mesh_orig%x(:)%val
! 4. Synchronize device array
call hipCheck(gpufort_array_copy_to_device(mesh_interop%x))
```

AMD

Preview: … GPUFORT Array Fortran API (VI)

- **gpufort_array_init(bytes_per_element,hostrptr,devptr,[sizes,bounds],alloc_mode,sync_mode)**
  - Init the array; different variants available
  - allocate (pinned) host data if requested (*alloc_mode*)
  - allocate/wrap device data if requested (*alloc_mode*)
  - Copy host <-> device at init / detruction if requested (*sync_mode*)
- **gpufort_array_wrap(bytes_per_element,hostptr,devptr,[sizes,bounds,…])**
  - Specialized init routine that always wraps and doesn't synchronize at destruction / init
- **gpufort_array_copy_data to_(host|device|buffer)(array,…)**
- **gpufort_array_hostptr(array,hostptr)**
  - Get a Fortran pointer type that allows you to modify the host data
- **gpufort_array_devptr(array,devptr)**
  - Get a Fortran pointer to the device data. You cannot modify it in the Fortran code but get pointers to subarrays
- **gpufort_array_destroy(array)**
  - **Deallocates all memory allocated at init; runs *sync_mode* operations before destruction**
- … and more; typically also an async variant is available for routines involving data movement

AMD

Preview: Interoperable arrays and structs (VI)

```
!$cuf kernel do(1) <<<grid, tBlock>>>
do i=1,size(mesh%y,1)
  mesh%y(i) = mesh%y(i) + mesh%a*mesh%x(i)%val
end do                                          ! FORTRAN
```

```
__global__ void  vecadd_kernel(mesh_t mesh) {
  int i = 1 + (1)*(threadIdx.x + blockIdx.x * blockDim.x);
  if (loop_cond(i,mesh.y.size(1),1)) {
    mesh.y(i)= mesh.y(i) + mesh.a*mesh.x(i).val;
  }
}                                          // HIP C++ kernel
```

- Body of kernel looks quite similar to original Fortran code; no index macros or C array access operator []

AMD

Preview: Interoperable arrays and structs (VII)

```
extern "C" void launch_vecadd_kernel_(
    const dim3& grid, const dim3& block,
    const int& sharedmem, hipStream_t& stream,
    mesh_t& mesh
) {
  hipLaunchKernelGGL((vecadd_kernel), grid, block,
      sharedmem, stream, mesh);
}
```

- Fortran passes all arguments by **reference** by default
  - if you do not create an explicit ISO_C_BINDING interface and add the *value* qualifier to the arguments
- Then, the C-side callee will need to dereference pointers or use pointer access operators.
- Often, programmers make mistakes here
- Fortunately, HIP C++ is a C++ language. So we can just use **references (&)** on the C++ side
  - No need for explicit **value arguments on the** Fortran side and/or pointer expressions on the C++ side
  - Arguably no need for an explicit Fortran interface.

AMD

Preview: Interoperable arrays and structs (VIII)

```
__global__ void  main_26_e28c45(
   float * __restrict__ y_d,
   const int y_d_n1,
   const int y_d_lb1,
   float a,
   float * __restrict__ x_d,
   const int x_d_n1,
   const int x_d_lb1) {
 #undef _idx_y_d
 #define _idx_y_d(a) ((a-(y_d_lb1)))
 #undef _idx_x_d
 #define _idx_x_d(a) ((a-(x_d_lb1)))
 int i = 1 + (1)*(threadIdx.x + blockIdx.x * blockDim.x);
 if (loop_cond(i,y_d_n1,1)) {
  y_d[_idx_y_d(i)]=(y_d[_idx_y_d(i)]+a*x_d[_idx_x_d(i)]);
 }
}
```

```
extern "C" void launch_main_26_e28c45(
   dim3* grid,
   dim3* block,
   const int sharedmem,
   hipStream_t stream,
   float * __restrict__ y_d,
   const int y_d_n1,
   const int y_d_lb1,
   float a,
   float * __restrict__ x_d,
   const int x_d_n1,
   const int x_d_lb1) {

 // launch kernel
 hipLaunchKernelGGL((main_26_e28c45), *grid, *block, sharedmem, stream,
         y_d,y_d_n1,y_d_lb1,a,x_d,x_d_n1,x_d_lb1);
}
```

- Original (old) GPUFORT output for comparison (similar kernel)
  - Long parameter lists
  - Indexing via [] operator and index macros
  - Pointer operations in the launcher body (**grid, *block*)

AMD

Preview: Interoperable arrays and structs (IX)

- **Summary:**
  - GPUFORT now provides interoperable array types (develop-acc branch) for AMD & NVIDIA devices (open-source; based on HIP)
  - Overloaded operator() allows Fortran-like syntax in C++ kernels
  - GPUFORT will use them to map derived types to the GPU (which require some OpenACC & CUDA Fortran codes)
    - … and to simplify and reduce the length of the kernel argument list
    - … and to simplify kernel code generation: Function call and array access look the same in C++ using these datatypes
  - GPUFORT arrays itself might aid to gradually move a Fortran code base to CUDA/HIP C++/(HIP-)SYCL