

GPUFORT

A source-to-source translator for Fortran accelerator dialects

Motivation

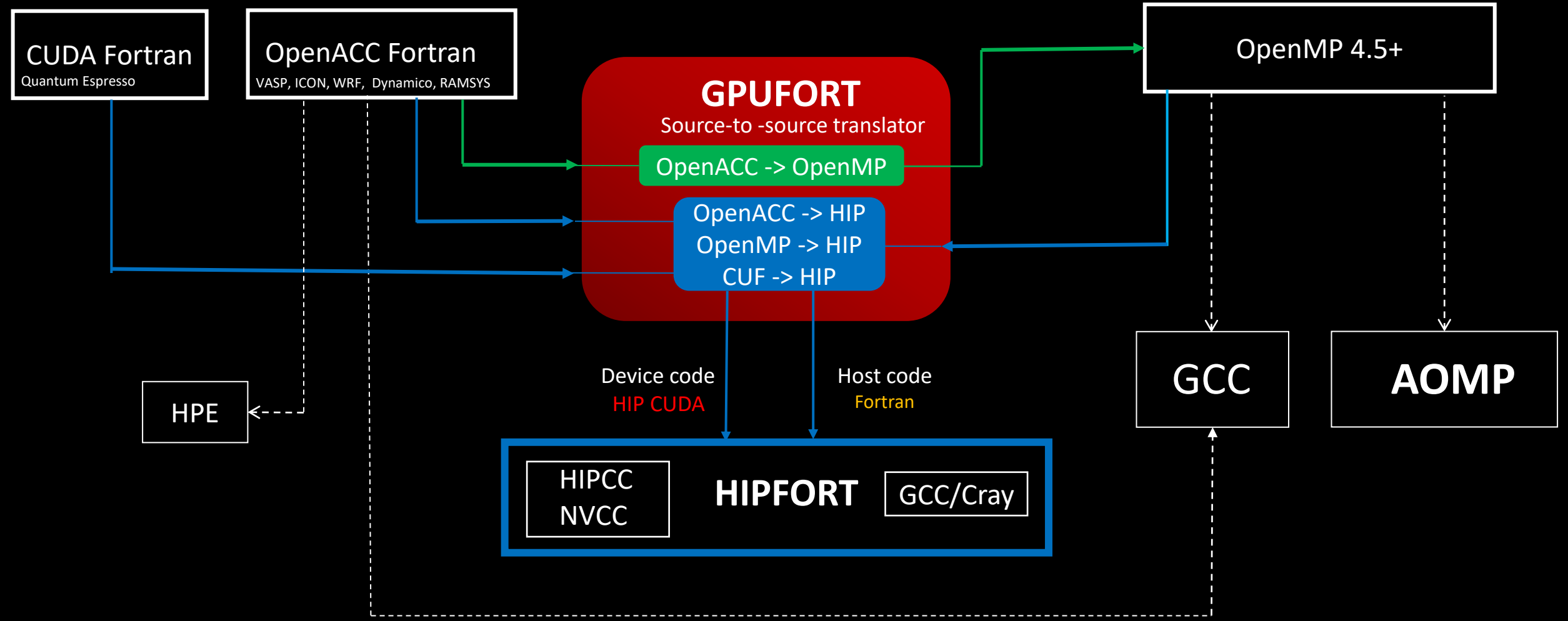
- Porting to new accelerator architectures costs time and money
 - Especially if a new programming language needs to be adopted
- Developers that can do such porting are difficult to find
- However, more often than not it is the only way to achieve good performance on novel architectures
- GPUFORT is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

Outline

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



GPUFORT



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

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

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

Outline

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

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>

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**

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 :: l
```

```
!
```

```
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/]

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**

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

Outline

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

Requirements/tasks for a translator for ...

CUDA Fortran

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

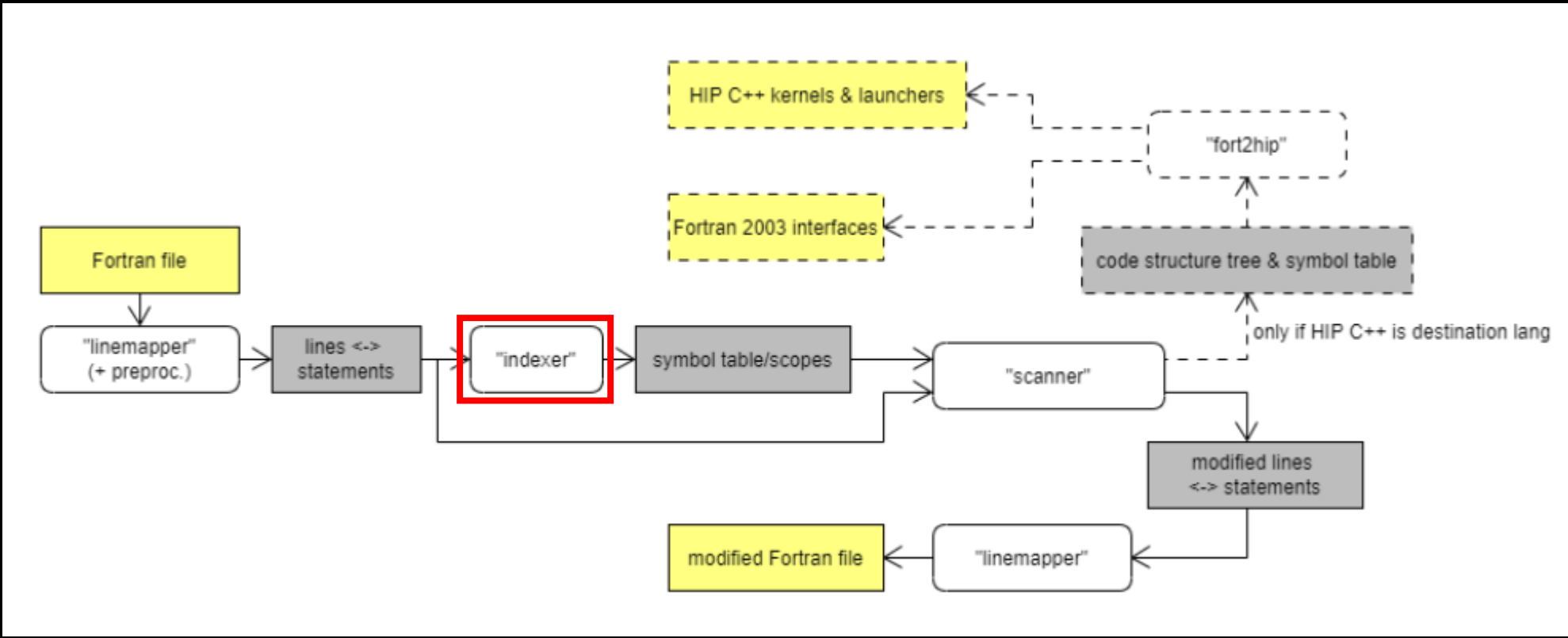
- Detect & translate **CUDA Fortran** intrinsics & lang extensions to hipFORT routines and standard Fortran types
- Detect, analyze & translate:
 - **CUDA Fortran** accelerator routines
 - **CUDA Fortran** *!\$cuf* directives

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

Architecture



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

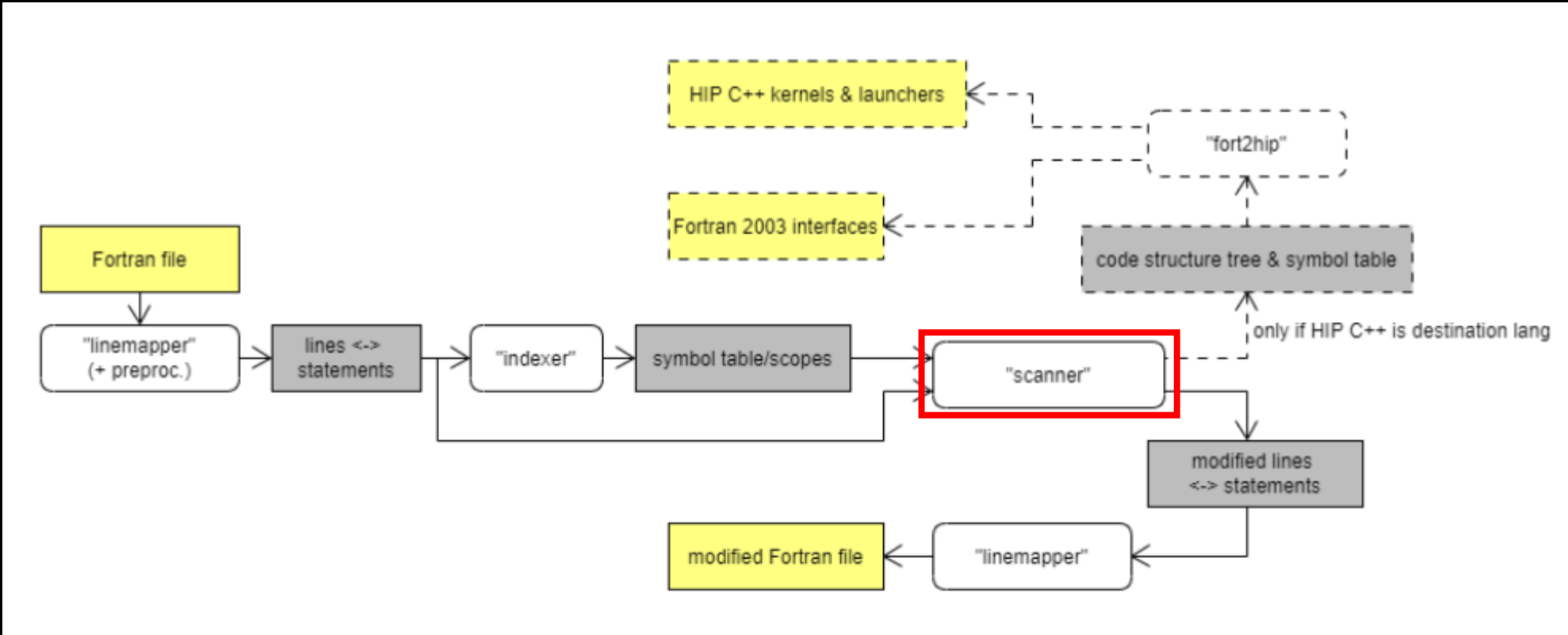
a(i) = 5;	b(i) = 2
c(i) = 5 * & (a(i) + b(i))	

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



Architecture



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

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!

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 ...

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 ...

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
```

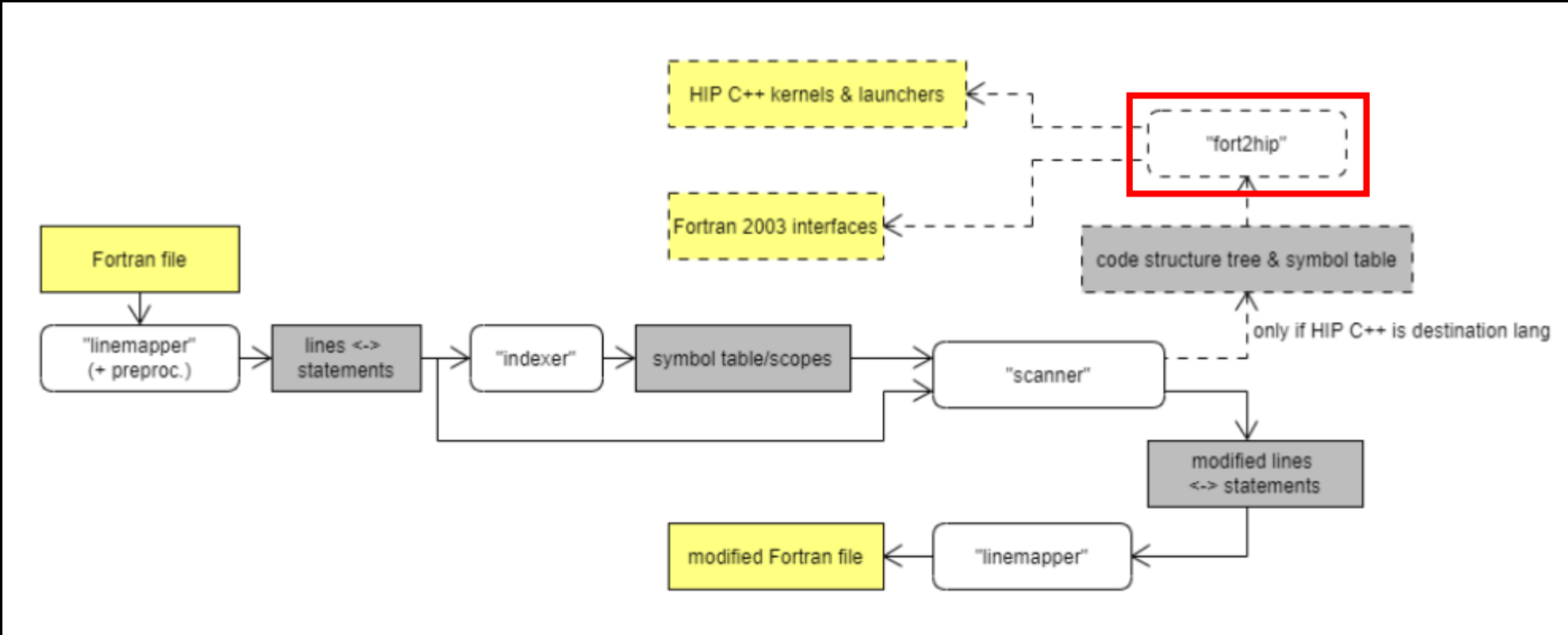
```
print(derived_type_elem.parseString("a%b") )
print(derived_type_elem.parseString("a%b%c") )
print(derived_type_elem.parseString("a%b%c%d") )
```

```
...
```


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!

Components: “fort2hip”



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*)

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)
}
```

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

Outline

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

Current Limitations

- **CUDA/OpenACC** Fortran to Fortran + **HIP C++**
 - GCC LIBGOMP runtime does not provide vendor-specific extension *acc_get_cuda_stream* for AMD backend as it is based on HSA internally
 - ⇒ Implement OpenACC runtime in HIP C++ that supports targeted apps (proof of concept exists)
 - As GPUFORTRAN does not know the memory layout of non-interoperable Fortran derived types, it cannot pass such variables to the **HIP C++** kernel launcher
 - ⇒ Generate intermediate interoperable type that can be mapped plus deep copy routines/code (proof of concept exists)
- **CUDA/OpenACC** Fortran to **OpenMP 4.5+ Fortran**
 - No mapping async queues to OpenMP *depend*
 - ⇒ Get user feedback via config file ({file,construct,line} -> depend(...))
 - Colon (":") matrix-vector operations
 - ⇒ Short Term: Unroll manually
 - ⇒ Long Term: Wait for *!\$omp parallel workshare*

Outline

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

Success Story: DYNAMICO

- During a bidding process, we were required to estimate the performance of a new architecture for the **OpenACC** application DYNAMICO
 - 93 offloaded loop nests
 - No async queue usage aside from default queue
 - Nearly no derived type access in offloaded loop nests
- GCC compiler did not deliver competitive results on our reference hardware
- With GPUFORT and a team of 4, we were able to translate the code within two weeks (Fortran + **HIP C++**)
- We achieved competitive performance on NVIDIA and AMD devices within the same time frame (vs NVIDIA OpenACC compiler)
 - GPUFORT version outperformed original NVIDIA SDK OpenACC build on A100

Outline

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

Summary

- We presented a python3 tool/-chain (GPUFORT) for analyzing & translating Fortran accelerator dialects (or custom DSLs)
 - Cross platform, tuneable, scriptable, no compile-latency, fast enough (O(1 sec))
 - Designed as translator but might develop into pre-compiler
 - Relies on well-tested and well-used python modules (pyparsing, jinja2, orjson)
- Potential applications are:
 - OpenACC Fortran to OpenMP Fortran conversion
 - CUDA Fortran to hipFORT (plus HIP C++ kernel conversion)
 - Manual optimization of bottleneck kernels in OpenACC/CUDA Fortran applications
 - Gradually porting a Fortran application to C++
 - Implementing domain specific languages for Fortran
 - ...
- Despite current limitations, team of 4 ported & tuned a real-world HPC application (OpenACC Fortran, 93 kernels) in less than two weeks and the result was competitive

Next Steps

- General:
 - Harden all components in collaboration with independent software vendors (in progress)
 - Make accessible to wider public as open-source tool (in progress)
 - Improve proof of concept conversion paths. For example:
 - **OpenACC** Fortran -> **OpenMP** Fortran
 - **CUDA Fortran** -> **OpenMP** Fortran
 - Implement additional conversion paths. For example:
 - **OpenMP 4.5+ Fortran** -> Fortran + **HIP C++**
 - Generate SYCL / DPC++ / ... output?
- Fortran to **HIP C++**:
 - Initial support of non-interoperable derived types (proof of concept exists)
 - Abandon LIBGOMP & create HIP-based offloading runtime that at least supports targeted applications (proof of concept exists)
 - Alternatively: Hop on another existing runtime that supports multiple vendors?