Hybrid Fortran v0.93

Documentation

Michel Müller

michel@typhooncomputing.com

Tokyo Institute of Technology, GSIC
Typhoon Computing

December 15, 2014

Contents

List of Figures

List of Tables

Chapter 1

The Hybrid Fortran Framework

Hybrid Fortran is a directive based extension of the and a code transformation tool within the Fortran language. It is intended for enabling GPGPU acceleration of data parallel programs using a unified codebase. Performance Portability was a major design aspect of this framework - not only should it enable the accelerated target to achieve optimal performance, but the CPU target should keep performing as before the migration. In the backend it automatically creates CUDA Fortran code for GPU and OpenMP Fortran code for CPU, in both cases making use of data parallelism defined by the user through directives. Additionally, a GNU Make build system as well as an automatic test system is provided. Hybrid Fortran has been successfully used for speeding up the Physical Core of Japan's national next generation weather prediction model by a factor of 3.6x on Kepler K20x vs. 6 Core Westmere *while not loosing any CPU performance*.

This chapter will describe the functionality of the **Hybrid Fortran** framework from a user perspective. Chapter 2 guides through setup, portation, debugging and testing in a 'howto'-like fashion. Chapter 3 will go into implementation details for those who would like to adapt **Hybrid Fortran** to their own specific usecases.

1.1 Design Goals

This section gives an overview over the design goals of the **Hybrid Fortran** framework.

1. **Hybrid Fortran** is intended to enable performance portable parallel programming for HPC purposes.

2. **Hybrid Fortran** offers a unified codebase for both CPU and GPU execution.

3. The storage order is compile time defined since CPU and GPU implementations usually have different ideal storage orders.

4. The rules for creating the GPU code versions should have a low complexity, such that optimizations performed by the user lead to predictable results.

5. The framework enables GPU performance at the level of hand optimized CUDA Fortran while maintaining CPU performance as close as possible to the original CPU code.

6. The implementation details are abstracted from the rest of the system, such that other parallel programming frameworks can be supported in the future without changes in the user code.

7. **Hybrid Fortran** is optimized for two use cases in particular: Stencil access patterns with tight loops and Parallel vector access patterns with outside loops. **Hybrid Fortran** abstracts domain specifications of arrays such that the migration path from typical Fortran code for these two use cases becomes minimal.

1.2 Features

- Backends for OpenACC (GPU), CUDA Fortran (GPU) and OpenMP (CPU) parallelizations.
- Separate parallel regions for CPU and GPU at multiple points in the callgraph (if needed). This allows high performance for both CPU and GPU implementations.
- Compile-time defined storage order with different orders for CPU and GPU - all defined in one handy place, the `storage_order.F90` file. Multiple storage orders are supported through attributes in the Hybrid Fortran directives.
- Automatic compile time array reformatting - Hybrid Fortran reformats your data with respect to privatization and storage order at compile time. This means you can leave existing Fortran code as is, only the setup using the two Hybrid Fortran directives is needed.
- Temporary automatic arrays, module data and imported scalars within GPU kernels (aka subroutines containing a GPU '@parallelRegion') - this functionality is provided in addition to CUDA Fortran's device syntax features.
- Experimental Support for Pointers.
- Separate build directories for the automatically created CPU and GPU codes, showing you the created F90 files. Get a clear view of what happens in the back end without cluttering up your source directories.
- Use any x86 Fortran compiler for the CPU code (PGI and Intel Fortran have been tested).
- Highly human readable intermediate F90 source files. The callgraph, including subroutine names, remains the same as in the user code you specify. The code in the intermediate source files is auto indented for additional reading comfort.
- Macro support for your codebase - a separate preprocessor stage is applied even before the hybrid fortran preprocessor comes in, in order to facilitate the DRY principle.
- Automatic creation of your callgraph as a graphviz image, facilitating your code reasoning. Simply type 'make graphs' in the command line in your project directory.

- Automatic testing together with your builds - after an initial setup you can run validation tests as well as valgrind automatically after every build (or by running ‘make tests‘). As an example, say you’d like the preprocessor to tell you that there is a calculation error in array X at point i=10,j=5,k=3? If you set up everything accordingly, this is pretty much what Hybrid Fortran does for you. This speeds up development in large framework a lot, since automated testing means that you can check your work after each submodule or even kernel.

- Automatic ’printf’ based device debugging mode. Prints all input arrays, output arrays and temporary arrays at the end of each kernel run at a compile-time specified point in a nicely readable format. This output can then be manually validated against the CPU version (which should already produce correct results at that point). Please note: Since PGI’s CUDA Fortran does not yet support real device debugging, Hybrid Fortran cannot support that either at this point. However, since the code runs on CPU as well, the class of bugs that are affected by this restriction is rather small (since computational code can be validated on the CPU first) and the current debug mode has been proven to be sufficient for the time being.

- Automatic linking and installing of executables. Simply specify the executable names in the ‘MakesettingsGeneral‘ configuration file and use corresponding filenames for the main files (which can be placed anywhere in your source tree). The Hybrid Fortran build system will automatically generate the executables (each in CPU and GPU version) and install them in subdirectories of your test directory. The test directories are persistant, such that you can put your initialization files, validation scripts and performance test scripts there. All this happens simply through running ‘make; make install‘ in your project directory.

1.3 Hybrid Fortran Directives

The directives introduced with the **Hybrid Fortran** framework are the only additions that have been made to the Fortran 90 language in order to achieve the objectives stated in sec. 1.1. All other syntax elements in **Hybrid Fortran** are a strict subset of Fortran 90.

It is necessary to introduce the following denotations before introducing the new directives:

A domain in this context denotes a tuple containing a data dimension and its size. For example, if we have an array `a` declared within the range (1, NX) and looped over using the iterator `x`, we call this array to be **domain dependant** in domain `x`. For simplicity, we assume that iterators over the same data dimension and range are always named in a consistant way.

A parallel region is a code region that can be executed in parallel over a one or more domains.

The following section lists the two directives and their available options for later reference.

1.3.1 Parallel Region Directive

Listing 1.1 shows the parallel region directive. This directive is an abstraction of for-loops as well as CUDA kernels that allows the framework to define these structures at compile time. It is only allowed to be inserted in the implementation part of a subroutine.

```
1 @parallelRegion{ATTRIBUTE_NAME1(MEMBER1, MEMBER2, ...), ...}
2 ! code to be executed in parallel !
3 symbol1, symbol2, ...
4 @end parallelRegion
```

Listing 1.1: Parallel region directive syntax.

The following attributes are supported for this directive:

appliesTo Specify one or more of the following attribute members in order to set this parallel region to apply to either the CPU code version, the GPU version or both. Omitting this attribute has the same effect as specifying all supported architectures.

1. CPU
2. GPU

domName (Required) Specify one or more domain names over which the code can be executed in parallel. These domain names are being used as iterator names for the respective loops or CUDA kernels.

domSize (Required) Set of the domain dimensions in the same order as their respective domain names specified using the `domName` attribute. It is required that $|domName| = |domSize|$.

startAt Set the lower boundary for each domain at which to start computations. Omitting this attribute will set all boundaries to `1`. It is required that $|startAt| = |domName|$.

endAt Set the upper boundary for each domain at which to end computations. Omitting this attribute will set all boundaries to `domSize` for each domain. It is required that $|startAt| = |domName|$.

template Defines a postfix that is to be applied to different attributes that are loaded using the preprocessor. Currently this only affects CUDA Blocksizes: They are loaded using `CUDA_BLOCKSIZE_X_[Template-Name]` from the preprocessor (`storage_order.F90` is the most handy place to define them). The goal of this attribute is to to be able to hoist hardware dependent attributes outside of the code, so when a new architecture comes along, all you need to do is rewriting the template. This functionality will be extended in future Hybrid Fortran releases. Specify this template name without quotes.

Please note: CUDA Fortran differentiates between different subroutine types [1, p. 4]. Following its design goal of keeping a low complexity, **Hybrid Fortran** simply rewrites subroutines definitions to one of the CUDA subroutine types, depending on the subroutine's position relative to the parallel region (determined through metainformation about the entire visible source code). This introduces some restrictions for subroutines calling, containing or being called by GPU parallel regions (see also sec.1.4). For future reference these restricted subroutines are named in the following way (see figure 1.1):

1. Subroutines that call one or more subroutines containing a GPU parallel region are called "wrapper subroutines".
2. Subroutines that contain a GPU parallel region are called "kernel subroutines".
3. Subroutines that are called inside a GPU parallel region are called "inside kernel subroutines".
4. All other subroutines are called "host subroutines".

1.3.2 Domain Dependant Directive

Listing 1.2 shows the "domain dependant" directive. It is used to specify the involved symbols and their domain dependencies. This information then allows the framework to rewrite the symbol accesses and declarations for the GPU and CPU cases (see example in section 1.3.4). Please note the following for using this directive:

1. For symbols the framework only operates on local information available for each subroutine. As an example, whether a symbol has already been copied to the GPU is not being analyzed. For this reason the `present` flag has been introduced (see below).
2. Domain Dependant Directives need to be specified between the specification and the implementation part of a Fortran 90 subroutine.

```
1 @domainDependant{ATTRIBUTE_NAME1(MEMBER1, MEMBER2, ...), ...}
2 ! symbols that share the attributes !
3 ! defined above to be defined here, separated  !
4 ! by commas                                     !
5 ...
6 @end domainDependant
7
8 !Minimal Example:
9 @domainDependant{domName(x), domSize(NX)}
10 a, b, c
11 @end domainDependant
12 !-> Defines the three arrays a, b, c to be dependant in domain x.
```

Listing 1.2: Domain dependant directive syntax.

The following attributes are supported for this directive:

domName Set of all domain names in which the symbol needs to be privatized. This needs to be a superset of the domains that are being declared as the symbol's dimensions in the specification part of the current subroutine (except if the `autoDom` attribute flag is used, see below). More specifically, the domain names specified here must be the set of domains from the specification part plus the parallel domains (as specified using the parallel region directive, see section 1.3.1) for which privatization is needed.

domSize Set of the domain dimensions in the same order as their respective domain names specified using the `domName` attribute. It is required that $|domName| = |domSize|$.

accPP Preprocessor macro name that takes $|domSize|$ arguments and outputs them comma separated in the current storage order for symbol accesses. This macro must be defined in the file `storage_order.F90` (see section

3.1). In case the `autoDom` attribute is being used, the `accPP` specification is not necessary - `AT`, `AT4`, `AT5` (...) are assumed as defined storage order macro names, depending on the number of array dimensions.

domPP Preprocessor macro name that takes $|domSize|$ arguments and outputs them comma separated in the current storage order for the symbol declaration. This macro must be defined in the file `storage_order.F90` (see section 3.1). In case the `autoDom` attribute is being used, the `domPP` specification is not necessary - `DOM`, `DOM4`, `DOM5` (...) are assumed as defined storage order macro names, depending on the number of array dimensions. This preprocessor macro is usually identical to the one defined in `accPP`.

attribute Attribute flags for these symbols. Currently the following flags are supported:

present In case this flag is specified, the framework assumes array data to be already present on the device memory for GPU compilation and the data will not be transfered.

transferHere In case this flag is specified, all domain dependants with `intent` specified as `in`, `inout` or `out` will be transfered to- and from the device (according to the intent). This flag may not be specified together with the `present` flag (and it should not be necessary, since it has exactly opposite effects).

autoDom In case this flag is specified, the framework will use the array dimensions that have been declared using standard Fortran 90 syntax to determine the domains for each symbol. In case the parallel domains are ommitted from the Fortran specification (in order to allow different parallelization for CPU and GPU), they still need to be specified using `domName` and `domSize` for symbols that are to be privatized for each thread. The parallel domains will be inserted before any independent domains picked up through the declaration, depending on the subroutine's position towards the parallel region. In addition, using `autoDom` will by default enable standard `accPP` and `domPP` settings, if not specified otherwise. Using this flag then greatly simplifies the `@domainDependant` specification part, since the directive template (everything between the directive and the corresponding `@end domainDependant` statement) can be reused by symbols of different domains.

host In subroutines or modules that are not related to a parallel region in their callgraph, Hybrid Fortran would not know whether a symbol resides on the host or the device. This attribute makes this clear. It is mostly needed in setup code and for module arrays.

Please note: In case neither `present` nor `transferHere` is being used, Hybrid Fortran will automatically transfer all domain dependants with

`intent` specified as `in`, `inout` or `out` from and to the device according to their intent - *if and only if* the current subroutine is calling a kernel. In other words, the automatic behavior without specifying `present` or `transferHere` in host- or wrapper subroutines will only work if the domain dependants are only used within one wrapper subroutine. For real world callgraphs it is therefore usually necessary to use `present` and `transferHere` attributes in all host- and wrapper subroutines.

An important special case are scalar parameters: In case of a CUDA Fortran implementation, scalars must be passed by value in device functions. The framework must for that reason be aware of scalar symbols in kernel subroutines, such that their specification can be adjusted accordingly. For simplicity reasons, the `@domainDependant` directive has been reused for scalars and can be used in the following way:

```
1 @domainDependant{}
2 scalar1, scalar2, ...
3 @end domainDependant
```

Listing 1.3: Domain dependant directive syntax for scalars.

In **Hybrid Fortran** terms then, a scalar is a domain dependant without domains. You can however use `attribute(autoDom)` as well here. As a general best practice, you will have two domain dependant directives: One listing the symbols and arrays privatized in the parallel domains and one listing the non privatized symbols and arrays (i.e. they are not variant over these domains).

1.3.3 Backend Implementations

Now that the directives have been introduced, you might wonder what exactly Hybrid Fortran is doing with all this. Essentially, the code transformation framework consists of three parts:

1. Parser (Your Hybrid Fortran code including directives).
2. Callgraph Analysis (Based on the Parser, produces a callgraph coloured by the subroutines' position relative to the parallel regions for CPU and GPU case).
3. Backend Implementation (For each relevant code point, the Parser calls implementation hooks for the transformation).

This separation of concerns allows different implementations to be passed in and used, thus enabling portability to different hardware platforms as well as flexibility during development and testing. The implementation class can be

chosen using the `MakesettingsGeneral` file in your project's config folder (see also sec. 2.2). Table 1.1 gives you an overview over the capabilities of each implementation. Adding your own implementation class is as easy as implementing the class in the file `hf_processor/FortranImplementation.py`, using any of the provided classes as a base class and specifying your class name in `MakesettingsGeneral`. For a more detailed look at how to do this, please see also sec. 3.5.

1.3.4 Example

Let's look at the following example module that performs matrix element addition as well as multiplication. Please note, that the storage order inefficiencies are disregarded in the lst. 1.4. Also, please note that there are many more examples provided in soure form on http://github.com/muellermichel/Hybrid-Fortran/blog/master/examples/Overview.md.

```
1  module example
2  contains
3  subroutine wrapper(a, b, c)
4      real, intent(in), dimension(NX, NY, NZ) :: a, b
5      real, intent(out), dimension(NX, NY, NZ) :: c
6      integer(4) :: x, y
7      do y=1,NY
8        do x=1,NX
9          call add(a(x,y,:), b(x,y,:), c(x,y,:))
10         call mult(a(x,y,:), b(x,y,:), c(x,y,:))
11       end do
12     end do
13 end subroutine
14
15 subroutine add(a, b, c)
16     real, intent(in), dimension(NZ) :: a, b, c
17     integer :: z
18     do z=1,NZ
19         c(z) = a(z) + b(z)
20     end do
21 end subroutine
22
23 subroutine mult(a, b, d)
24     real, intent(in) :: a(NZ), b(NZ)
25     real, intent(out) :: d(NZ)
26     integer :: z
27
28     do z=1,NZ
29       d(z) = a(z) * b(z)
30     end do
31 end subroutine
32 end module example
```

Listing 1.4: CPU version of matrix element module

Capability/ Class name (*='Implementation')	`Fortran*`	`OpenMPFortran*`	`PGIOpenACC*`	`CUDAFortran*`	`DebugCUDAFortran*`	`DebugEmulated-CUDAFortran*`
Tested target platforms	Intel x86	Intel x86	Nvidia Kepler	Nvidia Kepler	Nvidia Kepler	CUDA emulation mode
Tested compilers	Intel, PGI	Intel, PGI	PGI	PGI	PGI	PGI
Goal	Basic sequential implementation	OpenMP CPU Parallelization	OpenACC GPU Parallelization	CUDA Fortran GPU Parallelization	Debugging CUDA Fortran by printing one data point for each array after each kernel returns	Debugging in device emulated mode (allowing `write` statements within kernels)
Parallel regions implementation	Loops	Loops w/ OpenMP directives	Loops w/ OpenACC directives	CUDA kernels	CUDA kernels	CUDA kernels
Base class	`object`	`Fortran*`	`Fortran*`	`Fortran*`	`CUDAFortran*`	`DebugCUDAFortran*`
Allows multiple parallel regions per subroutine	Yes	Yes	Yes	No	No	No

Table 1.1: Backend Implementations in Hybrid Fortran

Porting this to the GPU, one might want to move the loops over the `x` and `y` domains to the `add` and `mult` subroutines, in order to eliminate the need for inlining and optimize for register usage. The following listing shows how this is done in **Hybrid Fortran**. Figure 3.3 in chap. 3 shows the (programmatically created) callgraph of this module.

```
1  module example contains
2  subroutine wrapper(a, b, c, d)
3    real, dimension(NZ), intent(in) :: a, b
4    real, dimension(NZ), intent(out) :: c, d
5    @domainDependant{domName(x,y), domSize(NX,NY), attribute(autoDom)
         }
6    a, b, c
7    @end domainDependant
8    @parallelRegion{appliesTo(CPU), domName(x,y), domSize(NX, NY)}
9    call add(a, b, c)
10   call mult(a, b, d)
11   @end parallelRegion
12 end subroutine
13
14 subroutine add(a, b, c)
15   real, dimension(NZ), intent(in) :: a, b
16   real, dimension(NZ), intent(out) :: c
17   integer :: z
18   @domainDependant{domName(x,y), domSize(NX,NY), attribute(autoDom)
         }
19   a, b, c
20   @end domainDependant
21   @parallelRegion{appliesTo(GPU), domName(x,y), domSize(NX, NY)}
22   do z=1,NZ
23       c(z) = a(z) + b(z)
24   end do
25   @end parallelRegion
26 end subroutine
27
28 subroutine mult(a, b, d)
29     real, dimension(NZ), intent(in) :: a, b
30     real, dimension(NZ), intent(out) :: d
31     integer :: z
32
33     @domainDependant{domName(x,y), domSize(NX,NY), attribute(
           autoDom)}
34     a, b, d
35     @end domainDependant
36
37     @parallelRegion{appliesTo(GPU), domName(x,y), domSize(NX, NY)}
38     do z=1,NZ
39       d(z) = a(z) * b(z)
40     end do
41     @end parallelRegion
42 end subroutine
43 end module example
```

Listing 1.5: example of a Hybrid Fortran subroutine with a parallel region

This will be rewritten by the **Hybrid Fortran** framework into two versions. The CPU version will be look like the original (but with an optimized storage order and added OpenMP directives for the parallelization) while the GPU version, using the CUDA Fortran backend implementation, is shown below. Please note, that

1. the two implementations loop over the `xy` domains at different points, according to the `appliedTo` attribute defined in the `@parallelRegion` directives. See figure 3.3 in cha. 3 in order to get an overview.

2. the declarations and accessors for the arrays `a`, `b`, `c` and `d` did not need to be changed in the `add` and `mult` subroutines.

```
1  module example
2  contains
3    subroutine wrapper(a, b, c, d)
4      use cudafor
5      real, intent(in) :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY, NZ))
6      real ,device :: a_d(DOM(NX, NY, NZ))
7      real ,device :: b_d(DOM(NX, NY, NZ))
8      real, intent(out) :: c(DOM(NX, NY, NZ)), d(DOM(NX, NY, NZ))
9      real ,device :: c_d(DOM(NX, NY, NZ))
10     real ,device :: d_d(DOM(NX, NY, NZ))
11
12     type(dim3) :: cugrid, cublock
13     integer(4) :: cuerror
14     a_d(:,:,:) = a(:,:,:)
15     c_d(:,:,:) = 0
16     b_d(:,:,:) = b(:,:,:)
17     d_d(:,:,:) = 0
18
19     cugrid = dim3(NX / CUDA_BLOCKSIZE_X, NY / CUDA_BLOCKSIZE_Y, 1)
20     cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
21     call add <<< cugrid, cublock >>>(a_d(AT(:,:,:)), b_d(AT(:,:,:))
           , c_d(AT(:,:,:)))
22     ! **** error handling left away to improve readability *** !
23
24     cugrid = dim3(NX / CUDA_BLOCKSIZE_X, NY / CUDA_BLOCKSIZE_Y, 1)
25     cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
26     call mult <<< cugrid, cublock >>>(a_d(AT(:,:,:)), b_d(AT(:,:,:)
           ), d_d(AT(:,:,:)))
27     ! **** error handling left away to improve readability *** !
28
29     c(:,:,:) = c_d(:,:,:)
30     d(:,:,:) = d_d(:,:,:)
31   end subroutine
32
33   attributes(global)   subroutine add(a, b, c)
34     use cudafor
35     real, intent(in) ,device :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY,
           NZ))
36     real, intent(out) ,device :: c(DOM(NX, NY, NZ))
```

```
37      integer :: z
38      integer(4) :: x, y
39
40      x = (blockidx%x - 1) * blockDim%x + threadidx%x
41      y = (blockidx%y - 1) * blockDim%y + threadidx%y
42      do z=1,NZ
43        c(AT(x,y,z)) = a(AT(x,y,z)) + b(AT(x,y,z))
44      end do
45    end subroutine
46
47    attributes(global) subroutine mult(a, b, d)
48      use cudafor
49      real, intent(in) ,device :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY,
          NZ))
50      real, intent(out) ,device :: d(DOM(NX, NY, NZ))
51      integer :: z
52      integer(4) :: x, y
53
54      x = (blockidx%x - 1) * blockDim%x + threadidx%x
55      y = (blockidx%y - 1) * blockDim%y + threadidx%y
56      do z=1,NZ
57        d(AT(x,y,z)) = a(AT(x,y,z)) * b(AT(x,y,z))
58      end do
59    end subroutine
60  end module example
```

Listing 1.6: GPU version of the hybrid code shown above

1.4 Restrictions

The following restrictions will need to be applied to standard Fortran 90 syntax in order to make it compatible with the **Hybrid Fortran** framework in its current state. For the most part these restrictions are necessary in order to ensure CUDA Fortran compatibility. Other restrictions have been introduced in order to reduce the program complexity while still maintaining suitability for common physical packages.

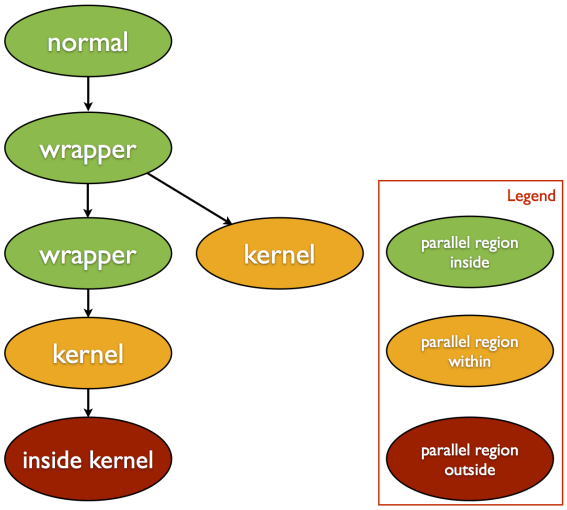

Figure 1.1: Callgraph showing subroutine types with restrictions for GPU compilation.

1. Hybrid Fortran has only been tested using free form Fortran 90 and Fortran 2003 syntax.

2. Your free form files will need the f90 or F90 file endings for the Hybrid Fortran build system to pick them up (this is also recommended by Intel if you want to use their compiler).

3. Support for pointers in kernel callers is still experimental. See the `diffusion3D` example for an example code on how to use pointers together with Hybrid Fortran. All pointers that are to touch the device need a specified intent and their domain / dimension setup needs to be specified within a domainDependant directive.

4. Hybrid Fortran currently supports data parallel programming for multicore CPU and GPU. In order to use reduce functions, it is recommended to use BLAS/CUBLAS (see poisson2d solver example).

5. Currently no line continuations are supported for the attribute definitions of directives.

6. Because of OpenACC / CUDA Fortran restrictions, kernel- and inside kernel subroutines may not

   (a) contain symbols declared with the `DATA` or `SAVE` attribute.

   (b) contain multiple parallel regions.

   (c) be recursive.

   (d) call other kernel subroutines.

   (e) contain the `recursive`, `pure` and `elemental` keywords.

   (f) contain inline array initialisations. Static data initialisations are ideally being done outside the parallel loop, typically in `init` subroutines for each module that are always executed on the CPU. **Hybrid Fortran** directives are not needed for code parts that are in no case to be executed on the GPU, however the compile time storage order needs to be be respected by using the same storage order macros as specified in the domPP and accPP attributes for the involved arrays. Alternatively you can specify `@domainDependant` directives in these code parts, so HF will take over the storage order handling.

7. Inside kernel subroutines called by kernel subroutines must reside in the same Fortran module as their caller.

8. All symbols that are declared as domain dependant using `@domainDependant` directives must be of `integer`, `real`, `character` or `logical` type (however any byte length within the Fortran 90/2003 specification is allowed). Hybrid Fortran doesn't currently have support for derived types containing data arrays that need to be copied onto the device. You can use derived types in host-only code however.

9. Arrays that are declared as domain dependant using `@domainDependant` directives may not appear in declaration lines with mixed domain dependance. Example:

```
1  ..
2  real(8), dimension(nz) :: a, b
3  real(8), dimension(nz) :: c
4  ..
5  @domainDependant {domName(x), domSize(nx)}
6  a, b
7  @end domainDependant
8
9  @domainDependant {domName(y), domSize(ny)}
10 c
11 @end domainDependant
```

```
12  ..
```

Listing 1.7: This is ok.

```
1  ..
2  real(8) dimension(nz) :: a, b, c
3  ..
4  @domainDependant {domName(x), domSize(nx)}
5  a, b
6  @end domainDependant
7
8  @domainDependant {domName(y), domSize(ny)}
9  c
10 @end domainDependant
11 ..
```

Listing 1.8: This is not ok.

10. The regular Fortran 90 declarations of any symbols declared as domain dependant may not contain line continuations.

11. Use statements in kernel- and inside kernel subroutines may not contain line continuations.

12. All source files (h90[1], H90[2], f90, F90) need to have distinctive filenames since they will be copied into flat source directories by the build system.

13. Subroutines in h90 and H90 files need distinctive names for the entire source tree.

14. Only subroutines are supported together with **Hybrid Fortran** directives, e.g. functions are not supported.

15. Preprocessor directives that affect the Hybrid Fortran preprocessing (such as code macros) must be expandable from definitions within the same H90 file. Use the H90 file suffix (instead of h90) in case you want to use macros in your code.

16. If you use local module scalars inside a kernel subroutine, the wrapper subroutine must reside in the same module.

17. Module scalars, when used in a kernel subroutine, will loose their constant characteristic on GPU. They therefore can't be used where a constant is required, such as in a `case` statement. (They do work as a dimension specifier for automatic arrays however.)

[1] h90 is the file extension used for Hybrid Fortran source files.

[2] H90 is the file extension used for Hybrid Fortran source files that contain same-file macro expansions.

18. I/O statements such as `read` or `write` and `STOP` statements are not possible inside GPU parallel regions, except for emulated mode.

In general, since

1. GPU execution currently requires subroutine calls to be inlined and
2. the number of registers per GPU Streaming Multiprocessor is limited

it is best to split deep callgraphs and large computations into multiple smaller kernels (i.e. `@parallelDomain{ appliedTo(GPU), ..}`).

1.5 Device Data Handling

The goal of device data handling is to hide and abstract code that is only necessary for the CUDA execution path. **Hybrid Fortran** works similarly to OpenACC in that respect. For all non-scalars that are marked as domain dependant using the `@domainDependant` directive, the following rules apply with respect to device data in case of GPU compilation:

1. If the current subroutine contains calls to kernel subroutines and the domain dependant symbol is declared using the `intent(in)` or `intent(inout)` statement, a device version of the symbol will be allocated and its content will be copied to that device array at the beginning of the subroutine.
2. If the current subroutine contains calls to kernel subroutines and the domain dependant symbol is declared using the `intent(out)` or `intent(inout)` statement, a device version of the symbol will be allocated and set to zero and the device array's content will be copied to the original array at the end of the subroutine.
3. If the domain dependant symbol is local for this subroutine, it will be allocated as a device symbol and its content will be set to zero at the beginning of the subroutine.
4. In case the domain dependant directive contains an `attribute(present)` statement, no data will be copied and the original symbol will be declared as a device symbol.
5. In case the domain dependant directive contains an `attribute(transferHere)` statement, the data will always copied over to the device, following the specified intents (as in (1), (2)).

1.5.1 Module Data

Using the following guidelines, Hybrid Fortran is able to manage your imported module data to be copied to and from the device.

1. After the specification part of your module, add `@domainDependant` directives and include all arrays that need to be touched by your parallel regions.
2. In case your module data arrays are allocatable (which is probable in case your problem dimensions are runtime defined), you cannot use `attribute(autoDom)`. Instead, use the `domName` and `domSize` attributes to specify the domain setup. You may use runtime defined scalar variables (e.g. `nx`, `i`, ...) within this specification as long as these variables are always defined within the

parallel regions that use these arrays. These runtime variables do **not** need to be imported into the declaring module.

3. Specify `attribute(host)` for all these arrays in the module specification.

4. In the module data consuming kernel subroutine, simply import the data with `use` statements (please note that Hybrid Fortran cannot parse multi-line `use` statements at this point). And specify them also inside the corresponding `domainDependant` directive. You may use `attribute(autoDom)` there, since Hybrid Fortran will use the domain information that you provide in the module specification. The data handling at this point follows the same rules as locally declared arrays (see above) - so make sure that you use `attribute(present)` and `attribute(transferHere)` if you have multiple kernel subroutines that touch the same data.

1.6 Feature Comparison between Hybrid Fortran and OpenACC

The following table gives an overview over the differences between OpenACC and the **Hybrid Fortran** framework.

Feature	OpenACC	Hybrid Fortran 90	Comments
Enables close to fully optimized Fortran code for GPU execution		✓	CUDA Fortran implementation available, which has equal or better performance than OpenACC in all cases known to us
Enables close to fully optimized Fortran code for CPU execution		✓	Storage order abstraction as well as allowing both coarse grained as well as fine grained parallelization leads to this result.
Automatic device data copying	✓	✓	
Allows adjusted looping patterns for CPU and GPU execution		✓	
Allows changing the looping patterns with minimal adjustments in user code		✓	
Handles compile time defined storage order		✓	
Allows to adapt for other technologies without changing the user code (e.g. switching to OpenCL)		✓	Details, see section 3.5
Allows arbitrary access patterns in parallel domains	✓	✓	
Allows multiple parallel regions per subroutine	✓	(✓)	HF: Only OpenACC backend.
Generated GPU code remains easily human readable		✓	OpenACC compiles to CUDA C (PGI), introduces new functions for device kernels. Hybrid Fortran can translate to CUDA Fortran, code remains easily readable.
Allows debugging of device data	✓	✓	
Framework Sourcecode available		✓	

Table 1.2: Feature Comparison OpenACC vs. Hybrid Fortran

Chapter 2

Usage of the Hybrid Fortran Framework

This chapter is intended to give the informations necessary for installing and using the **Hybrid Fortran** framework.

2.1 Framework Dependencies

Hybrid Fortran requires the following software components:

1. A compiler for either OpenACC Fortran (Cray or PGI (only PGI tested so far)) or CUDA Fortran (PGI).
2. An x86 Fortran compiler.
3. Python v2.6.x, Python v2.7.x or compatible.
4. GNU Make 3.81.
5. A POSIX compatible operating system.
6. (optional) `valgrind` is recommended if you would like to use the test system shipped with this framework (accessible through `make tests`).
7. (optional) Allinea DDT if you need parallel debugging on the device.
8. (optional) For the graphical callgraph representation using `make graphs`: "pydot" python library[1] as well as the "Graphviz" program package[2].
9. (optional) `NetCDF4-Python` and `numpy` in case you'd like to use Hybrid Fortran's automated testing together with NetCDF Output.

[1] http://code.google.com/p/pydot/
[2] http://www.graphviz.org/Download..php

2.2 User Defined Components

The following files displayed in figure 3.1 are defined by the user:

h90(H90) Fortran sources A source directory that contains Hybrid Fortran files (h90/H90 extension). It may also contain files with f90 or F90 extensions. The source directory is by default located at `path-to-project/source/*` (this can be changed in the `path-to-project/config/MakesettingsGeneral` file).

Makefile Used to define module dependencies. The Makefile is by default located at `path-to-project/buildtools/Makefile`. Note: All source files are being copied into flat source folders before being compiled - the build system is therefore source directory structure agnostic, i.e. files can be placed into arbitrary subdirectores below the source directory.

MakesettingsCPU CPU compiler settings are specified in `MakesettingsCPU`, located at `path-to-project/buildtools/`.

MakesettingsGPU GPU compiler settings are specified in `MakesettingsGPU`, located at `path-to-project/buildtools/`.

MakesettingsGeneral Common settings, such as executable file names, the choice Hybrid Fortran preprocessor implementation class, or files and folders being excluded from compilation. With the helper comments these settings should be self explanatory.

storage_order.F90 This fortran file contains fortran preprocessor statements in order to define the storage order for both CPU and GPU implementation. It can be placed anywhere in the source directory or its subdirectories (see above).

2.3 Build Interface

make builds both cpu and gpu versions of the codebase situated in `path-to-project/source/*`.

make build_cpu builds the cpu version of the codebase situated in `path-to-project/source/*`.

make build_gpu builds the gpu version of the codebase situated in `path-to-project/source/*`.

make install builds both cpu and gpu versions of the codebase situated in `path-to-project/source/*` and installs the executables into the test folder defined in `path-to-project/buildtools/MakesettingsGeneral`.

make install_cpu Like `make install`, but only for the cpu version.

make install_gpu Like `make install`, but only for the gpu version.

make clean Removes the build directories as well as cpu executables from the test folders.

make clean_cpu Like `make clean`, but only for the cpu version.

make clean_gpu Like `make clean`, but only for the gpu version.

make tests Executes `make install` and runs the automatic tests (see sec. 2.4) for all executables.

make tests_cpu Like `make tests`, but only for the cpu version.

make tests_gpu Like `make tests`, but only for the gpu version.

make TARGETS DEBUG=1 builds TARGETS in debug mode (use any of the targets defined above). Uses the `DebugCUDAFortranImplementation` in case of GPU compilation (by default), which prints predefined data points for every kernel parameter after every kernel execution. See also the flag `DEBUG_MODE` in `MakesettingsGeneral` which allows to use the debug mode by default.

make TARGETS VERBOSE=1 builds TARGETS with more detailed output.

make graphs creates the graphical callgraph representations in the `path-to-project/build/callgraphs/` directory.

2.4 Test Interface

Hybrid Fortran comes with an automated test system that - once set up - is intended to find all errors in your code, each time a build completes. This includes

- Initialization errors (symbols / arrays being read without initializing them first), using `valgrind`.
- Memory handling errors (in case you forget to deallocate arrays), using `valgrind`.
- Computational errors, using reference data. Currently supported for this validation is output in `NetCDF` format as well as standard Fortran `.DAT` files. For NetCDF, up to five data dimensions per variable are supported, for .DAT files it is up to three dimensions.

2.4.1 Integration

If you would like to integrate the provided system, please do the following:

1. Choose between the following options for your output.
   (a) Have your program output in NetCDF format. In that case you will need `numpy` as well as `NetCDF-Python` installed as additional dependencies. It is recommended to install these dependencies into a `virtualenv` such that you can use a different compilers for the post processing as for the program itself (you will need NetCDF compiled in both versions). Compiling the entire post processing stack with PGI compilers (all of python, numpy) is not recommended. You can then use the `SOURCE_THIS_BEFORE_TESTING` and `SOURCE_THIS_AFTER_TESTING` options in `MakesettingsGeneral` to specify the activate and deactivate scripts for your virtualenv. In case of NetCDF, the Hybrid Fortran test system will automatically pick up any variable in your output and test it against the reference.
   (b) Add calls to the `helper_functions` module procedures `write1DToFile`, `write2DToFile` and `write3DToFile` to your program to be tested in order to write your data to the `.dat` files in the `test\your-executable\out` folder. The `helper_functions` module is part of the Hybrid Fortran libraries and it is always included in Hybrid Fortran builds - in fact it gets copied into your build directories for consistancy reasons. You may choose any filename for the `.dat` files, the `allAccuracy.py` script will find them automatically in the `test\your-executable\out` folder.
2. Compile (`make; make install` in project directory) and run your program to make sure the files are being created. Make sure that they actually contain data, for example by checking the file size.
3. Repeat steps 1, 2 in your reference source code in case you go for the .DAT-files option.
4. Compress the reference data created by your reference program into `./test/your-executable/ref.tar.gz`. See also the description of `runTest.sh` in conjunction with the `validation` command below to understand the correct format.
5. During development of your hybrid codebase, set the flags `TEST_WITH_EVERY_BUILD` and `DEBUG_MODE` to `true` in the file `MakesettingsGeneral`. This ensures that your tests will run with every build of Hybrid Fortran.

2.4.2 Interface

The following files are part of the sample test interface provided with **Hybrid Fortran**. They are located in the framework's binary directory. In order to set up the test system correctly, what's relevant for you is the information provided for the files `runTest.sh` and `runTests.sh`. The other files are described here for completeness and in case you'd like to adapt the system for different use cases.

accuracy.py Compares one NetCDF - or Fortran 90 `.dat` file with a reference file. Endianness, number of bytes per floating point value can be specified for the `.dat` case using command line parameters. See `--help` for usage.

allAccuracy.sh Compares all Fortran 90 `.dat` files or NetCDF files according that match a filename pattern. By default, the pattern `./out/*.dat` is used - you can override this by defining `TEST_OUTPUT_FILE_PATTERN` in MakesettingsGeneral.

runTest.sh Executes a series of tests for one executable. In order to use this, please `cd` into the executable's test directory first. This script takes three mandatory and three optional command line arguments :

1. The path to the executable as seen from its working directory.
2. The architecture name for which the tests should be performed (currently either `cpu` or `gpu`).
3. The postfix of the command line argument specification file. These files with filename `testConfig_[postfix].txt` should be placed in the executable's test directory for this matter. Each line in these text files will be interpreted as follows: `arg_name1 arg_value1 arg_name2 arg_value2 ...`. All lines need to have the same number of command line arguments specified. The executables to be used with this test system are assumed to have a unix-style command line interface. This can easily be achieved using the `kracken` Fortran module, already provided with Hybrid Fortran (for its documentation, see [2]). As an example, the following command line argument specification file will call the executable once with arguments `-nx 1 -ny 2` and once with `-nx 2 -ny 3`:

```
1 nx 1 ny 2
2 nx 2 ny 3
```

Listing 2.1: A sample command line argument specification file

. By using different postfixes you can define any number of configuration files that can then be used together with `runTest.sh`. Please note that the following postfixes have a special meaning:

validation attempts to extract the reference data from the file `ref.tar.gz` (which is to be located inside the executable's test directory) and runs `allAccuracy.sh` with matching reference directories named using the schema `./ref_[arg_name1][arg_value1]_[...]/`. As an example, if you'd like to use the specification file as shown in lst. 2.1, you will need to provide a file `ref.tar.gz` that contains the following reference data directories: `ref_nx1_ny2` and `ref_nx2_ny3`. Use the command `tar -cvzf ref.tar.gz ref_*` to create this file once you have the reference data ready. In order to create

valgrind calls valgrind tests with these command line specifications. This should only be used for cpu executables that have been compiled using debug flags (`-g`).

4. (optional) The output file pattern for your executable. See also the setting `TEST_OUTPUT_FILE_PATTERN` in `config/MakesettingsGeneral`.
5. (optional) The path to a script that is to be sourced before running validation tests (see also step 1 in section 2.4.1).
6. (optional) The path to a script that is to be sourced after running validation tests.

runTests.sh This script `cd`'s into the test directories and runs `validation` as well as `valgrind` tests (for cpu executables when the debug argument is being passed to this script) for all specified executables. This assumes that you have specified command line argument specification files for those two test cases (see above). `runtests.sh` takes the following arguments:

1. A list of paths to executables.
2. The mode in which the executables should be run - currently `debug` or `production`.
3. The architecture name for which the tests should be performed (currently either `cpu` or `gpu`).
4. (optional) The output file pattern for your executable. See also the setting `TEST_OUTPUT_FILE_PATTERN` in `config/MakesettingsGeneral`.
5. (optional) The path to a script that is to be sourced before running validation tests (see also step 1 in section 2.4.1).
6. (optional) The path to a script that is to be sourced after running validation tests.

By setting the `TEST_WITH_EVERY_BUILD` flag to `true` in the file `config/MakesettingsGeneral`, every build will automatically run `runTests.sh` with the executable list used for compilation as well as the correct debug flag.

2.5 Getting Started

1. Clone http://github.com/muellermichel/Hybrid-Fortran to your computer used for development. Make sure your system meets the dependencies specified in section 2.1.

2. Set the environment variable `HF_DIR` to the location under which you have installed Hybrid Fortran.

3. `cd` into the Hybrid Fortran directory you've now installed on your computer.

4. Run `make example`. This creates a new project directory named `example`.

5. Run `cd example`.

6. Run `make; make install`. If everything worked you should now have a test subdirectory containing the example subdirectory containing two executables, one for CPU and one for GPU execution. Otherwise it is likely that some dependencies are missing, please reconsider section 2.1.

7. Run `./test/example/example_cpu; ./test/example/example_gpu`. This should execute and validate both versions.

8. Review the example source files located in `./source` and get a feel for the Hybrid Fortran directive syntax. Notice the `storage_order.F90` file which is used as a central point for specifying the data storage orders. Please refer to the documentation for details.

9. Review the preprocessed source files located in `./build/cpu/source` and `./build/gpu/source`. Notice the OpenMP and CUDA code that has been inserted into the example codebase. These files are important for debugging as well as when you want to do manual performance optimizations (but you should usually never change anything there, since it will get overwritten with the next preprocessor run).

10. Review the config files located in `./config`. The most important file for integrating your own codebase will be `./config/Makefile`. This file specifies the dependency tree for your source files. Please note that `vpath`'s are not necessary, the Hybrid Fortran build system will find your source files automatically, as long as you use the source directory specified in `./config/MakesettingsGeneral` as the root of your sources (i.e. you may place your sources in an arbitrarily deep subdirectory structure). The `MakesettingsCPU` and `MakesettingsGPU` are used to define the compilers and compiler flags. You may use any CPU compiler, however only `pgf90` is currently supported for CUDA compilation.

11. Run `make clean; make DEBUG=1; make install` in your example project directory. This replaces the previously compiled executables with debug mode executables.

12. The CPU version can be debugged with a compatible debugger.

13. Run `./test/example/example_gpu` and notice how this executable now prints debug information for every input and output at a specific data point after every kernel run. You can change the data point in `storage_order.F90`.

14. Rename the example project directory to your project name and start integrating your codebase. You can move it to any directory you'd like.

2.6 Migration to Hybrid Fortran with CUDA Fortran Backend

Assuming that the starting point is a Fortran 90 source code for CPU, use the following guidance in order to port your codebase to **Hybrid Fortran** with the CUDA Fortran Backend. Please note: If you have many tight parallel loops in the same subroutine, you may want to start with the OpenACC backend, since it allows multiple parallel regions per subroutine. You can at a later point still easily migrate to the CUDA Fortran backend by splitting up your kernels into their own subroutine, if you find the OpenACC performance to be poor.

1. Run `make example` in the Hybrid Fortran root directory.

2. Rename the new example project directory to your project name. You can also move it to any location you'd like.

3. Delete `source/example.h90` and copy in the sourcecode you'd like to hybridize using Hybrid Fortran. Please note that all loops that are to be run in parallel on CPU or GPU and their entire callgraph should be visible to the compiler in the source subdirectory. Your hybrid sources may be in an arbitrary subdirectory structure below the `source` directory and may consist of `f90` and `F90` files. The buildsystem will find all your sourcefiles recursively and copy them into the respective build directory in a flat hierarchy.

4. When it comes to integrating your hybrid sourcecode into a larger codebase, there are essentially two recommended options.

   (a) Move the entire codebase over into the Hybrid Fortran build system. This is recommended if your build dependencies are specified in (or can easily ported to) one central Makefile.

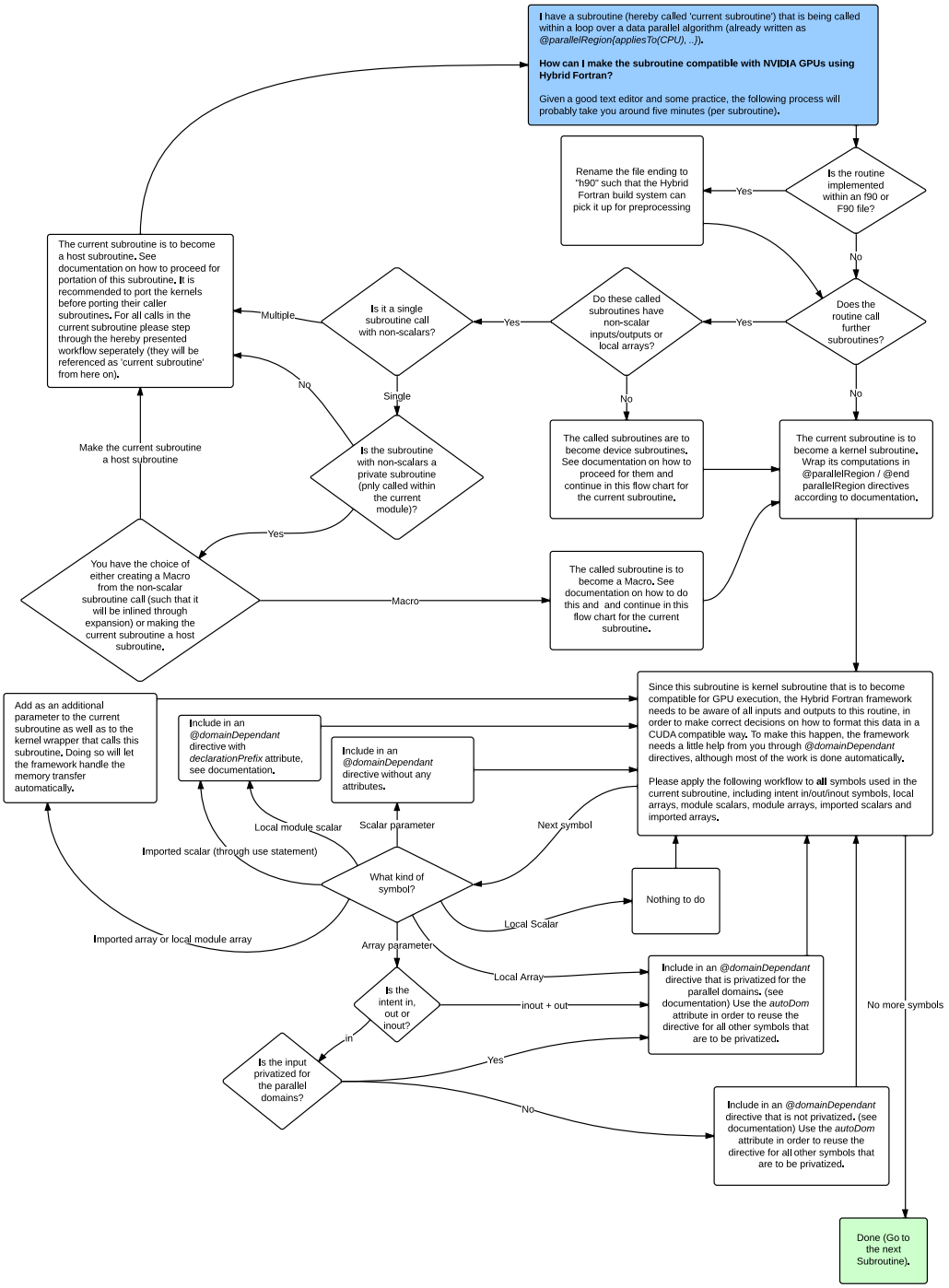

Figure 2.1: Best practices for a source code migration to Hybrid Fortran

(b) (new since version 0.9) Move one (or later more) module of your sourcebase over into the Hybrid Fortran build system. In this case it is recommended to create a separate source directory for hybrid sources, move your code there and link its path using the `SRC_DIR_COMMON` setting in `MakesettingsGeneral`. Then, specify your other framework source directories using the setting `FRAMEWORK_DIRS` and specify your previous main Makefile using the setting `FRAMEWORK_MAKEFILE`. You should use the Makefile coming with Hybrid Fortran in the root project folder in order to interface with your build (no changes should be required other than the settings you specify in `MakesettingsGeneral`. This will create separate build directories for the CPU and GPU case, copy in your `FRAMEWORK_DIRS`, generate and compile the hybrid sources and then call `FRAMEWORK_MAKEFILE` within the build directory. You will also need the settings `FRAMEWORK_EXECUTABLE_PATHS` and `FRAMEWORK_INSTALLED_EXECUTABLE_PATHS` in order to tell Hybrid Fortran which executables are being created by your build system and where you'd like them installed afterwards for testing purposes. You will need to adapt your previous build system for having moved your hybrid sources.

5. In case you went with option 4a, adapt your filenames containing main routines. The filename of your file(s) containing main routines need to correspond to the executable names you have specified in the `EXECUTABLES` variable in `MakesettingsGeneral`. E.g. if you want executables named `production_exec`, `test1_exec` and `test2_exec`, the corresponding main files need to be named `production_exec.f90`, `test1_exec.f90` and `test2_exec.f90`.

6. Adjust `config/MakesettingsGeneral` - the configuration options should be self explanatory.

7. Adjust `config/Makefile` by adding your hybrid source dependencies (and deleting the example). If your previous build system was already built on `GNU Make` it should be possible to cut and paste your dependency definitions without change. If you would like to exclude some files or folders in your source directory from compilation (in order to integrate modules one by one), you can do this by editing the `EXCEPTIONS` variable in `MakesettingsGeneral`.

8. Adjust the `FFLAGS` and `LDFLAGS` variables in `config/MakesettingsGPU` and `config/MakesettingsCPU` to reflect the compiler and linker options that are needed to compile your codebase. Please note that currently only CUDA Fortran with the Portland Group compiler `pgf90` is supported and tested as the GPU implementation.

9. Adjust the `source/storage_order.F90` file according to the comments you find there. This defines storage order macros and a few other variables

that Hybrid Fortran will use at the preprocessor stage. Make sure that the storage order matches a scheme that is performant on the respective target architecture - as the example shows, the `GPU` variable can be used to differentiate between GPU and CPU architecture. For the GPU case, the first dimension should be the one that gets mapped to X on the thread-blocks, in order for memory accesses to be coalesced. This is in turn the first dimension you specify in the `domName` and `domSize` attributes in your `@parallelRegion` directives.

10. Run `make; make install` and run your program in order to test whether the integration of your sourcecode into the **Hybrid Fortran** build system has been successful. It should create the cpu and gpu executable versions of your program in the test directory, however the gpu version will not run on the GPU yet, since no directives have been defined so far.

11. Integrate a test system. You can use the test scripts that have been provided with this framework (see sec. 2.4) or use any other test system.

12. Define the parallel regions that are to be accelerated by CPU[3] using a `@parallelRegion{appliesTo(CPU), ...}` directive. See sec. 1.3.1 for details. Rename all files that (a) contain such regions or (b) contain subroutines that are part of the call hierarchy inside those regions from `*.f90` or `*.F90` to `*.h90`.

13. Make sure your program still compiles and runs correctly on CPU by executing `make clean; make install_cpu; [run your tests]`.

14. (optional, you will need graphviz and pydot) Run `make graphs`. You should now have a graphical representation of your call hierarchy as "seen" from your parallel regions upwards and downwards in the call tree.

15. Define the subprocedures within that call hierarchy that are to be ported as GPU kernels. Fig. 2.1 is designed to help you with that process. Kernel subprocedures should have the following properties:

    (a) They only call subprocedures from the same module.

    (b) They only call one more level of subprocedures (rule of thumb).

    (c) The set of these subprocedures is self enclosed for all data with dependencies in your parallel domains, i.e. your data is only directly read or written to inside these to-be kernels and their callees. If this is not the case it will be necessary to restructure your codebase, e.g. put pre- and postprocessing tasks that are defined inline within higher level subprocedures into subprocedures and put them into your set of kernels.

[3] For example "do" loops that are already executed on multicore CPU using OpenMP statements.

16. Make sure that callgraphs within GPU kernel subprocedures as well as kernel subprocedure callers are all defined in h90/H90 files (rename from f90/F90 where necessary).

17. Make sure that at least the kernel callers have all the array arguments defined using `intent` statements (`in, out, inout`). It is recommended that all arguments in all your subroutines are defined this way, since the compiler will pick up some logical errors for you with this information. This includes pointer types as well.

18. Analyse for all kernels which data structures they require and which of those structures are dependant on your parallel domain. Define `@domainDependant` directives for those data structures within all kernels, kernel subprocedures and all intermediate subprocedures between your kernels and your CPU parallel region. See sec. 1.3 for details. In kernel subroutines and inside kernel subroutines you will need to declare local scalars and imported data as well. See sec. 1.3.2 for details. See also fig. 2.1.

19. If your kernels use arrays from the local module or imported arrays, pass them to the kernel subroutine using parameters. Use appropriate `@domainDependant` directives. See sec. 1.3.2 for details.

20. Imported arrays that are used in kernel subroutines and inside kernel subroutines also need to be input parameters of the kernel caller itself for appropriate automatic device data handling. That is, they need to be passed down to the kernels on two levels. Use appropriate `@domainDependant` directives.

21. Wrap the implementation sections of all your kernels with `@parallelRegion{appliesTo(GPU), ...}` / `@end parallelRegion` directives. See sec. 1.3.1 for details.

22. Test and debug on CPU by executing `make clean;make build_cpu DEBUG=1` with automatic tests enabled. You can use any compatible x86 debugger here, such as PGI debugger, Intel debugger, Totalview or Allinea DDT. Please note that the debuggers will show you the line numbers of the respective `F90` files, not your `h90` or `H90` files. The `F90` files have been kept as readable as possible however, and they can all be inspected in `./build/cpu/source`. Fig. 2.2 is designed to help you with this process.

23. Switch to GPU tests debug on GPU using `make clean;make build_gpu DEBUG=1; make install_gpu; [run your tests]`. Fig. 2.2 is designed to help you with this process. The print output will help you identify problems that only exist in the GPU version. Hint: Most of the bugs that persist after having a correct CPU implementation are

usually related to data dimensionalities specified using `@domainDependant` directives. You should be able to trace back the error by comparing the printed output with the CPU version opened in a separate debugger window and thus identify the kernels responsible for the errors. Once you've found the offending kernels, make sure the data is correctly formatted in the specification part of the `F90` file in `./build/gpu/source`. If the printed output does not show the error (whose index point should be recognized in your validation / test system you've integrated before), you can change the debug data indexes in `storage_order.F90`.

24. Congratulations, you have just completed a CPU/GPU hybrid portation using **Hybrid Fortran**.

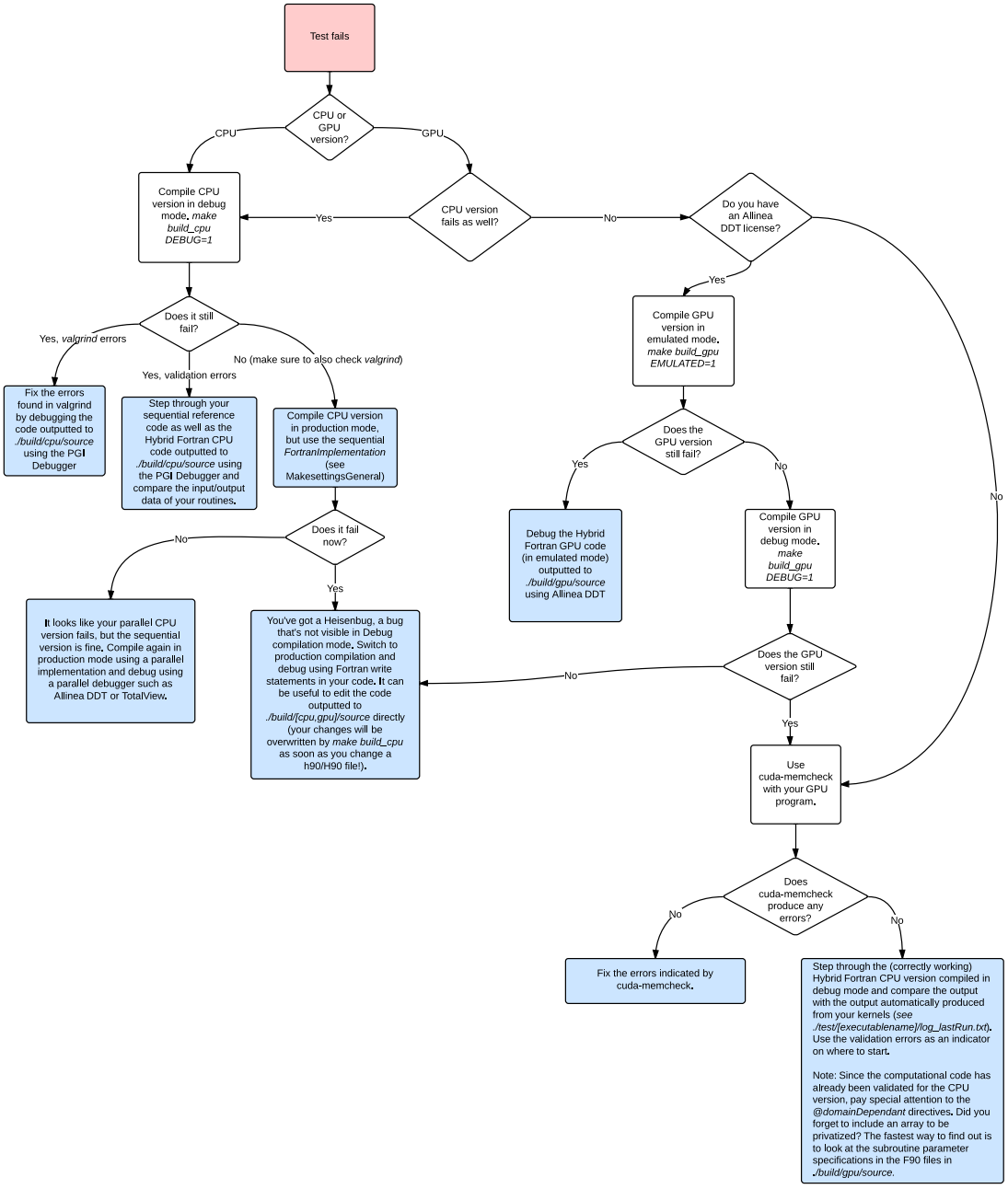

Figure 2.2: Best practices for debugging Hybrid Fortran code

Chapter 3

Framework Implementation

In this chapter the design of the **Hybrid Fortran** framework is presented from the implementation perspective. It will discuss the architecture that has been introduced for implementing the **Hybrid Fortran** framework, in order to achieve the goals and behaviour outlined in sec. 1.1 and sec. 1.3. If you'd like to adjust the framework for your own purposes, this chapter is a good place to start.

3.1 Overview and Build Workflow

The **Hybrid Fortran** build system involves the following components (depicted in fig. 3.1):

project-dir/Makefile offers a convenient interface to the build system. Please refer to appendix 2.3 for the usage of this build interface. It performs the following operations (assuming a clean rebuild):

1. Creates a build directory containing subdirectories for CPU and GPU builds.
2. Copies all `f90` and `F90` source files (pure Fortran 90 sources without **Hybrid Fortran** directives) into the CPU and GPU build directories using a flat file hierarchy.
3. Creates the callgraph `xml` file as well as the colored CPU and GPU callgraph versions in the callgraph subdirectory within the build directory.
4. Creates the graphical callgraph representations in the callgraph directory using **Graphviz** libraries.
5. Converts each `h90` source file into `F90` source files, using different implementations and callgraph colorings for the CPU and GPU case. The `F90` files are created in their respective build subdirectories (CPU or GPU).

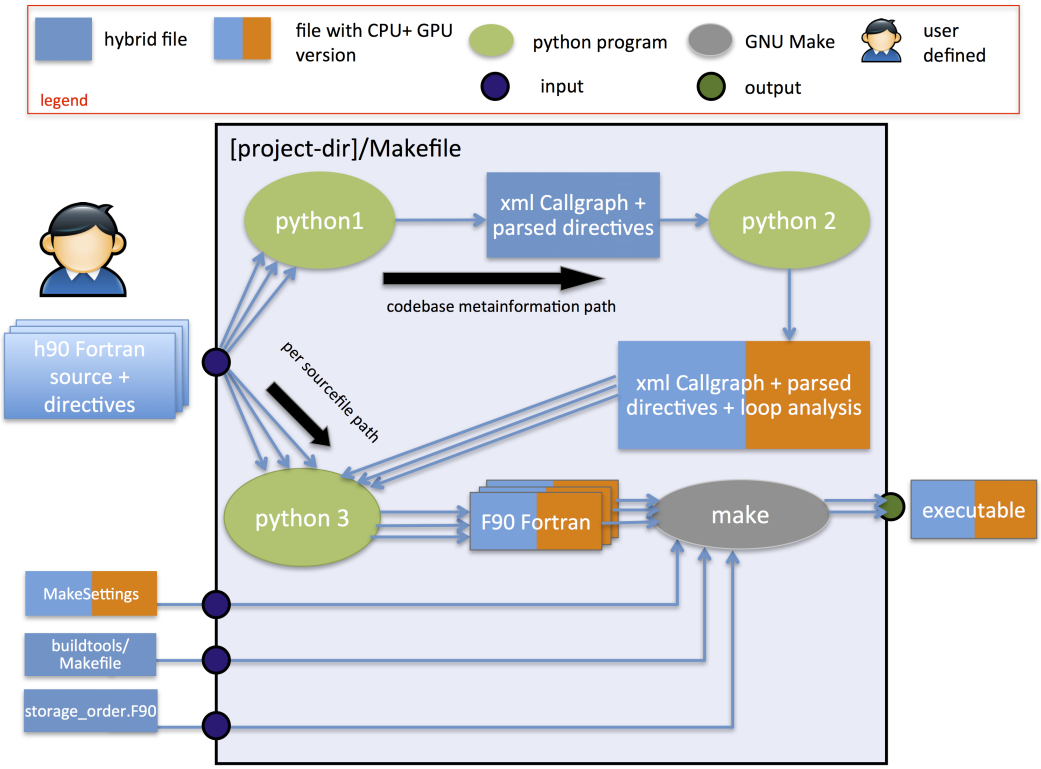

Figure 3.1: **Hybrid Fortran** Components and Information Flow.

6. Copies the `project-dir/buildtools/Makefile` into the CPU and GPU source directories.
7. Copies either `project-dir/buildtools/MakesettingsCPU` and `project-dir/buildtools/MakesettingsGPU` into the respective build subdirectory.
8. Executes `make` within the build subdirectories.
9. Installs the resulting executables into the test directory, using `cpu` or `gpu` as a postfix in the executable filename.

project-dir/buildtools/Makefile defines the dependencies between the Fortran 90 and **Hybrid Fortran** sources.

project-dir/buildtools/MakesettingsCPU defines the compiler name, compiler flags and linker flags for the CPU case.

project-dir/buildtools/MakesettingsGPU defines the compiler name, compiler flags and linker flags for the GPU case.

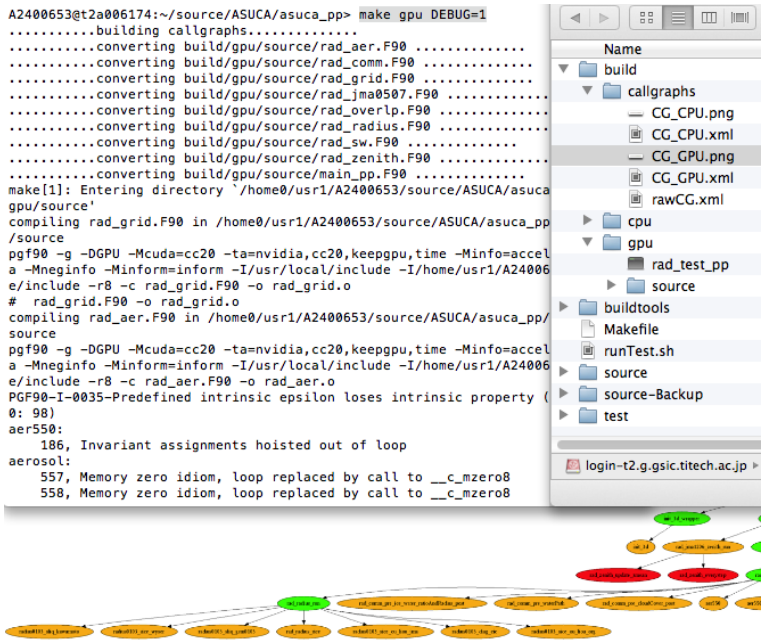

Figure 3.2: Screenshot of the **Hybrid Fortran** build system in action.

3.2 Python Build Scripts

The following python command line interface programs are part of the **Hybrid Fortran** build system:

annotatedCallGraphFromH90SourceDir.py goes through all h90 files in a given source directory and builds an xml file containing meta information about that source tree. The extracted meta information includes the call-graph visible from h90 files as well as a parsed version of the **Hybrid Fortran** directives inserted by the user. Figure 3.1 depicts this program in node `python 1`.

loopAnalysisWithAnnotatedCallGraph.py takes the meta information xml file from the previous script as its input and analyses the positioning of the user defined parallel regions relative to all subprocedures. Depending on its input arguments it performs this analysis for either the CPU or GPU version of the program in order for the framework to support compile time defined positioning of loops (kernel regions in the CUDA implementation). Figure 3.1 depicts this program in node `python 2`.

generateF90fromH90AndAnalyzedCallGraph.py takes one h90 source file as well as the analyzed meta information xml file as its inputs. It goes through the source file line by line and rewrites it in order to create compatible versions for CPU and GPU. The following operations are most essential to this module:

- Rewriting of parallel region definitions to conventional loops for the CPU or CUDA Fortran kernels for the GPU.
- Mutation of declarations and accesses of domain dependant arrays according to their position relative to the currently active parallel region.
- Insertion of statements to copy array data to and from the device in the GPU case.

Parallel domain dependant Figure 3.1 depicts this program in node `python 3`.

graphVizGraphWithAnalyzedCallGraph.py This program has been created in order to make debugging easier and to give the user an overview over the codebase and the involved parallel regions. It creates a graphical representation of the call graph from the analyzed meta information. The nodes in these call graphs are colored according to their relative position to the parallel regions. Figure 3.3 shows a sample of such a programmatically created call graph representation.

3.3 User Defined Files

The following files, depicted figure 3.1, are defined by the user:

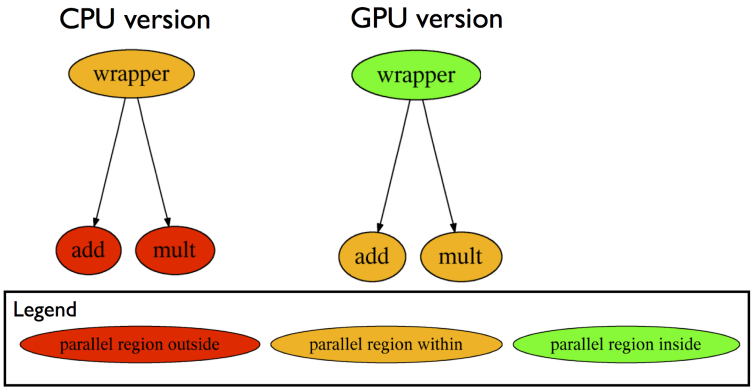

Figure 3.3: Condensed version of a simple callgraph, programmatically created by graphVizGraphWithAnalyzedCallGraph.py.

h90 Fortran sources A source directory that contains **Hybrid Fortran** files (h90 extension). It may also contain files with f90 or F90 extensions. The source directory is by default located at `path-to-project/source/*`.

Makefile Used to define module dependencies. The Makefile is by default located at `path-to-project/buildtools/Makefile`. Note: All source files are being copied into flat source folders before being compiled - the build system is therefore agnostic to the source directory structure implemented by the framework user.

storage_order.F90 This fortran file contains fortran preprocessor statements in order to define the storage order for both CPU and GPU implementation. This file is located at

`path-to-project/source/`
`hybrid_fortran_commons/storage_order.F90.`

3.4 Class Hierarchy

Figure 3.4 shows the classes created to implement the functionality described in section 3.2. For a full list of the implementation classes, please see also sec. 1.3.3.

H90Parser parses **Hybrid Fortran** (h90) files. The parser uses a mixture of state machine and regular expression design patterns. More specifically: Each line is matched against a set of regular expressions. The set of regular expressions being used is determined by a state machine and the outcomes of the regular expression matches in turn determine the state transitions.

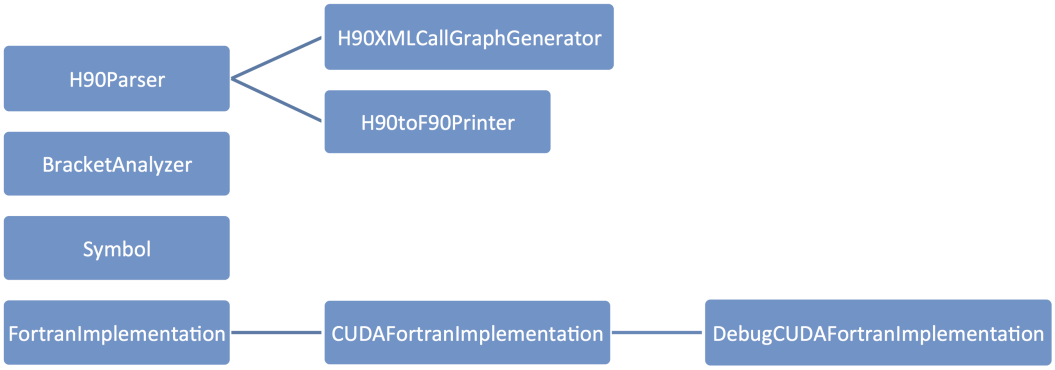

Figure 3.4: **Hybrid Fortran** Python Class Hierarchy.

See section 3.6 for a more detailed look at the **Hybrid Fortran** parser implementation.

H90XMLCallGraphGenerator (subclass of `H90Parser`) adds routine and call nodes to a new or existing call graph xml document. This functionality is used by `annotatedCallGraphFromH90SourceDir.py` as described in section 3.2.

H90toF90Printer (subclass of `H90Parser`) prints a fortran 90 file in F90 format (including preprocessor statements) to POSIX standard output. The configuration of this class includes

1. a **Hybrid Fortran** file as its main input (inherited from the parent class).
2. an xml callgraph including parsed **Hybrid Fortran** directives and the positions of parallel regions relative ot the routine nodes.
3. a FortranImplementation object which determines the parallel implementation.

`generateF90fromH90AndAnalyzedCallGraph.py` uses this functionality as described in section 3.2.

BracketAnalyzer is used to determine whether a Fortran line ends with an open bracket.

Symbol Stores array dimensions determined at the time of declaraton for later use and includes functionality to print adapted declaration and access statements.

FortranImplementation provides the concrete syntax for a standard Fortran 90 implementation of the **Hybrid Fortran** program.

CUDAFortranImplementation (subclass of `FortranImplementation`) provides the syntax for a CUDA Fortran implementation, thus handling

- the conversion of parallel region directives into CUDA kernels,
- the conversion of subroutines called by kernels into device subroutines,
- the copying of data from and to the device,
- the synchronization of threads after CUDA kernels have finished executing (asynchronous execution of kernels is currently not supported) and
- error handling.

DebugCUDAFortranImplementation (subclass of `CUDAFortranImplementation`) extends the CUDA Fortran implementation to include print statements to POSIX standard error output for all kernel parameters at a user defined data point after the execution of the kernel. This functionality enables debugging of device code since barebone CUDA Fortran currently does not offer printing or debugging for code executed on the GPU. (There is an emulation mode available which runs CUDA Fortran programs on the CPU, however it has been found to diverge too much from the device version).

3.5 Switching Implementations

Figure 3.5 shows the the most important class member functions of `FortranImplementation` classes and their role with respect to the example shown earlier in section 1.3.4. Each of these methods takes context information objects (for example a set of symbols that are referenced on this line, or a parallel region template containing the information users have passed with the directives) and returns strings that will be inserted at the indicated places into Fortran 90 files by the `H90toF90Printer` class. Introducing a new underlying technology such as OpenCL (for GPU implementations) or OpenMP (for CPU implementations) is as simple as writing a new `FortranImplementation` subclass containing these functions.


```
1  module example
2  contains
3    subroutine wrapper(a, b, c, d)
4      use cudafor
5      real, intent(in) :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY, NZ))
6      real ,device :: a_d(DOM(NX, NY, NZ))
7      real ,device :: b_d(DOM(NX, NY, NZ))
8      real, intent(out) :: c(DOM(NX, NY, NZ)), d(DOM(NX, NY, NZ))
9      real ,device :: c_d(DOM(NX, NY, NZ))
10     real ,device :: d_d(DOM(NX, NY, NZ))
11
12     type(dim3) :: cugrid, cublock
13     integer(4) :: cuerror
14     a_d(:,:,:) = a(:,:,:)
15     c_d(:,:,:) = 0
16     b_d(:,:,:) = b(:,:,:)
17     d_d(:,:,:) = 0
18
19     cugrid = dim3(NX / CUDA_BLOCKSIZE_X, NY / CUDA_BLOCKSIZE_Y, 1)
20     cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
21     call add <<< cugrid, cublock >>>(a_d(AT(:,:,:)), b_d(AT(:,:,:))
           , c_d(AT(:,:,:)))
22     cuerror = cudaThreadSynchronize()
23     if(cuerror .NE. cudaSuccess) then
24       write(6, *) 'CUDA error: ', cudaGetErrorString(cuerror)
25       stop 1
26     end if
27
28     cugrid = dim3(NX / CUDA_BLOCKSIZE_X, NY / CUDA_BLOCKSIZE_Y, 1)
29     cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
30     call mult <<< cugrid, cublock >>>(a_d(AT(:,:,:)), b_d(AT(:,:,:)
           ), d_d(AT(:,:,:)))
31     cuerror = cudaThreadSynchronize()
32     if(cuerror .NE. cudaSuccess) then
33       write(6, *) 'CUDA error: ', cudaGetErrorString(cuerror)
34       stop 1
35     end if
36
37     c(:,:,:) = c_d(:,:,:)
38     d(:,:,:) = d_d(:,:,:)
39   end subroutine

41   attributes(global) subroutine add(a, b, c)
42     use cudafor
43     real, intent(in) ,device :: a(DOM(NX, NY, NZ)), b(DOM(NX, NY,
           NZ))
44     real, intent(out) ,device :: c(DOM(NX, NY, NZ))
45     integer :: z
46     integer(4) :: x, y
47
48     x = (blockidx%x - 1) * blockDim%x + threadidx%x
49     y = (blockidx%y - 1) * blockDim%y + threadidx%y
50     do z=1,NZ
51       c(AT(x,y,z)) = a(AT(x,y,z)) + b(AT(x,y,z))
52     end do
53   end subroutine
```

Figure 3.5: Class member functions of "FortranImplementation" classes (Example shown with CUDAFortranImplementation).

3.6 Hybrid Fortran Parser

In order to interpret the directives introduced in cha 1 in the right context, it was necessary to create the parser program outlined in this section. This parser is used by the `annotatedCallGraphFromH90SourceDir` and `generateF90fromH90AndAnalyzedCallGraph.py` python scripts (as described in sec. 3.2) through subclasses.

This section gives a more detailed view of that parser. Figure 3.6 shows the state machine pattern that has been used for the parser implementation.

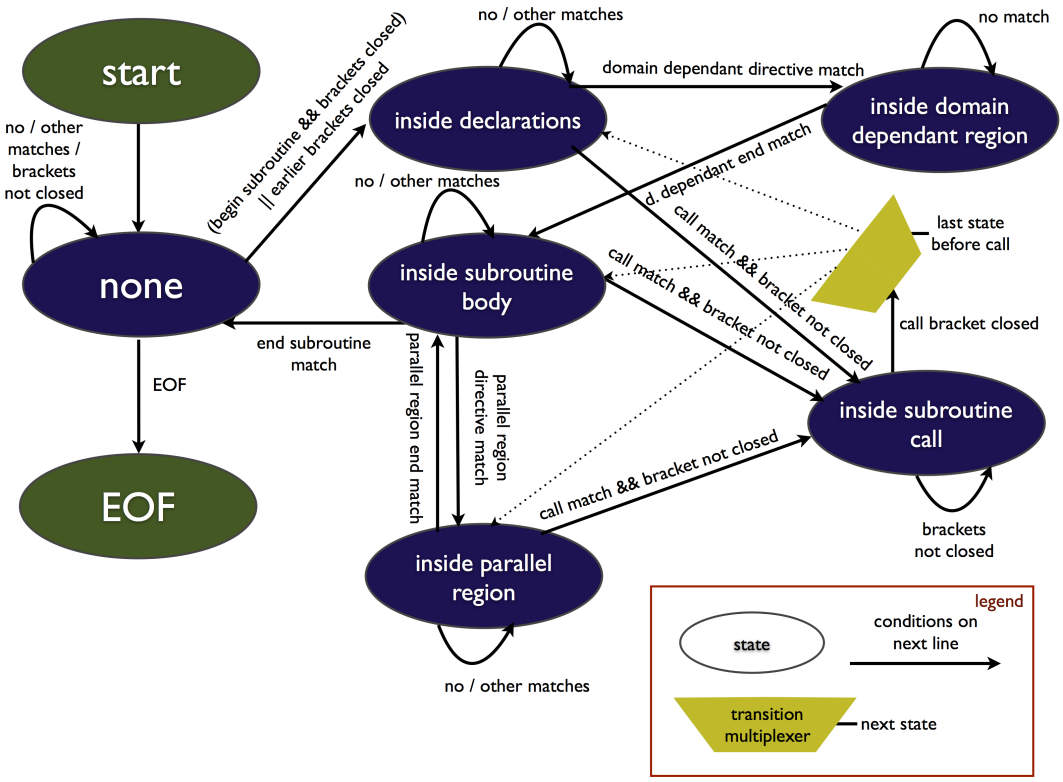

Figure 3.6: H90 Parser State Machine.

The state machine design pattern being used here resembles that of a Mealy machine. However, two changes have been applied to the Mealy machine properties:

1. The output is being detached from the machine. In other words the `H90Parser` class does not produce any output itself. Its subclasses (as described in sec. 3.4) are responsible for that task. This allows large parts of the parser code to be reused for both python programs dealing with h90 source files as described in sec. 3.2

2. A multiplexer is introduced as an additional element in order to reduce the number of states (which matches the way code is being reused in the actual implementation).

Bibliography

[1] The Portland Group, *CUDA Fortran Programming Guide and Reference*, 2012.

[2] J. S. Urban, "Kracken(3f): The ultimate fortran command line argument cracker." http://home.comcast.net/~urbanjost/CLONE/KRACKEN/krackenhelp.html, 2013. [Online; accessed June 24, 2014].