TYPHOON COMPUTING

RIKEN

Hybrid Fortran v0.8

Documentation

Michel Müller

michel@typhooncomputing.com

Typhoon Computing
Rikagaku Kenkyuujo (RIKEN)

April 1, 2013

Contents

List of Figures

List of Tables

Chapter 1

The Hybrid Fortran Framework

Hybrid Fortran is a directive based extension of the Fortran language. It is intended for enabling GPGPU acceleration of physical packages[1] while keeping x86 CPU compatibility and performance[2]. In the backend it automatically creates CUDA Fortran code for GPU and OpenMP Fortran code for CPU. Hybrid Fortran is currently used for porting Japan's national next generation weather prediction model to GPGPU[3].

This chapter will describe the functionality of the **Hybrid Fortran** framework from a user perspective, while chapter 3 will go into implementation details.

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 easy GPGPU portation of physical packages with a low amount of interprocedural communication, but high computational complexity and code complexity.

[1] Physical package here means code for high performance computations where the data access dependencies are orthogonal to the parallelized dimensions - as opposed to dynamical packages with general stencil dependencies.

[2] Between 20%-30% (10k codelines) of this weather model has already successfully been ported to GPGPU using Hybrid Fortran, showing a speedup of 3.5x with one Kepler card vs. 6 core Westmere. Fully integrated with the already ported dynamics we expect speedups in the region of 5x-7x per Kepler card. Considering that compute nodes can carry twice as many kepler cards as CPUs, this allows substantial cost and power savings and/or increased grid resolution.

[3] Ported code has so far shown a performance increase from anywhere between -5% and +25% on six core CPU.

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

3. The storage order must be 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. Since stencil computations in physical packages often do not have dependencies at offset positions in the parallelized dimensions, it is beneficial to the portation process to hide or abstract `IJ` dependencies of arrays.

8. Since the `IJ` dependencies are abstracted, it becomes possible to define the loops over the `IJ` domains at compile time. In other words, the framework is to be able to switch at compile time between outside loops (better suited for CPU caching) and inside loops (better suited for the GPU streaming computation model).

1.2 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 consistantly the same - a coding style that has been present within all the modules examined in the ASUCA Physical Process.

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.

1.2.1 Domain Dependant Directive

Listing 1.1 shows the "domain dependant" directive. They are 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 1.2.3).

Notes:

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 1.1: 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 1.2.2) 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).

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). Note: 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.

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 non parallel domains. Parallel domains (as specified using the parallel region directive, see section 1.2.2) still need to be defined using the *domName* and *domSize* attributes. In addition, using *autoDom* will by default enable standard *accPP* and *domPP*

settings, if not specified otherwise. This means the framework assumes DOM/AT (3D), $DOM4/AT4$ (4D), $DOM5/AT5$ (5D) (etc.) preprocessor directives to be specified for the compile time defined storage order (below three dimensional symbols don't get any directive added). Using this flag then greatly simplifies the @*domainDependant*| specification part, since the directive template (everything between the directive and the corresponding @*enddomainDependant* statement) can be reused by symbols of different domains.

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, 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 1.2: Domain dependant directive syntax for scalars.

In **Hybrid Fortran** terms then, a scalar is a domain dependant without domains.

1.2.2 Parallel Region Directive

Listing 1.3 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 1.3: 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.

1. CPU
2. GPU

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

1.2.3 Example

Let's look at the following example module that performs matrix element addition as well as multiplication. Please note, that the storage order in this example is defined at compile-time using the preprocessor macros `DOM` and `AT`. These macros simply reorder the arguments in order to reflect the optimal storage order for CPU and GPU case.

```
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       end do
11     end do
12 end subroutine
13
14 subroutine add(a, b, c)
15     real, intent(in), dimension(NZ) :: a, b, c
16     integer :: z
17     do z=1,NZ
18         c(z) = a(z) + b(z)
19     end do
20 end subroutine
21
22 subroutine mult(a, b, d)
23     real, intent(in) :: a(NZ), b(NZ)
24     real, intent(out) :: d(NZ)
25     integer :: z
26
27     do z=1,NZ
28       d(z) = a(z) * b(z)
29     end do
30 end subroutine
31 end module example
```

Listing 1.4: CPU version of matrix element module

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 module looks like 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,z), domSize(NX,NY,NZ), domPP(DOM),
           accPP(AT)}
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 abstracted storage order) while the GPU version 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

1.3 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.

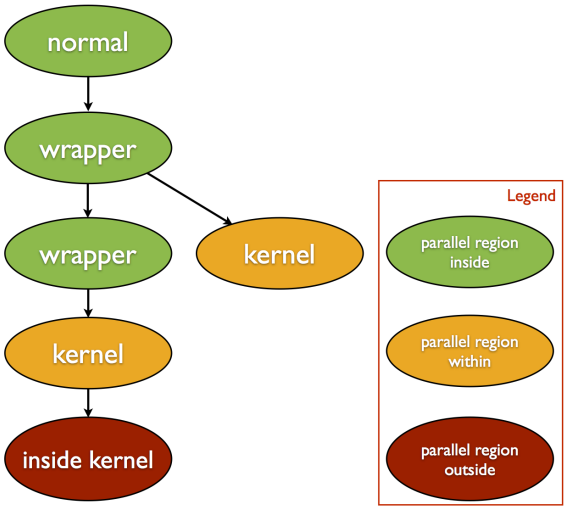

Figure 1.1: Callgraph showing subroutine types with restrictions for GPU compilation.

1. No data dependencies at offsets in I,J-directions (only vertical dependencies).

2. Hybrid Fortran has only been tested using Fortran 90 syntax and its GNU Make based build system only supports Fortran 90 files (f90 / F90). Since the Hybrid Fortran preprocessor only operates on subroutines (i.e. it is not affected by OOP specific syntax), this restriction can be lifted soon.

3. Currently no line continuations are supported for the attribute definitions of directives. Instead, directives are designed to only require a limited amount of attributes.

4. 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. For future reference these restricted subroutines are named in the following way (see figure 1.1):

(a) Subroutines that call one or more subroutines containing a GPU parallel region are called "wrapper subroutines".

(b) Subroutines that contain a GPU parallel region are called "kernel subroutines".

(c) Subroutines that are called inside a GPU parallel region are called "inside kernel subroutines".

Now that the names of those special subroutines are defined, it is possible to state the following restrictions. Because of 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.

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

6. All non scalar symbols with attributes added through `@domainDependant` directives may only be accessed and set inside parallel regions.

7. Arrays that are declared as domain dependant using `@domainDependant` directives must be of `integer` or `real` type (however any byte length within the Fortran 90 specification is allowed).

8. 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.

9. The regular Fortran 90 declarations of any symbols declared as domain dependant may not contain line continuations.
10. All source files (h90[4], H90, f90 and F90) need to have distinctive filenames since they will be copied into flat source directories by the build system.
11. Only subroutines are supported together with **Hybrid Fortran** directives, e.g. functions are not supported.
12. 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.
13. `@domainDependant` directives are required for all arrays in all subroutines called within parallel regions (the preprocessor operates only on local symbol information within each subroutine).

In general, since

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

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

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

1.4 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.

1.5 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		✓	
Enables close to fully optimized Fortran code for CPU execution		✓	
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	✓		Hybrid Fortran initially designed for ASUCA physical process access patterns (no offsets in I,J domains)
Allows multiple parallel regions per subroutine	✓		
Allows arbitrary CPU compilers	(✓)	✓	OpenACC: Only HMPP
Generated GPU syntax is a direct mapping of Fortran 90 user syntax		✓	OpenACC compiles to CUDA C (PGI), introduces new functions for device kernels. Hybrid Fortran translates to CUDA Fortran, syntax remains easily readable.
Allows debugging of device data		✓	
Framework Sourcecode available		✓	

Table 1.1: Feature Comparison OpenACC vs. Hybrid Fortran

Chapter 2

Usage of the Hybrid Fortran Framework

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

2.1 Framework Dependencies

Hybrid Fortran requires the following software components:

1. Any Fortran 90 CPU compiler.
2. For GPU compilation: PGI (`pgf90`) with CUDA Fortran support.
3. Python v2.6 or compatible.
4. GNU Make.
5. A POSIX compatible operating system.
6. (optional) For the graphical callgraph representation using `make graphs`: "pydot" python library[1] as well as the "Graphviz" program package[2].

2.2 User Defined Components

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

h90 Fortran sources A source directory that contains Hybrid Fortran files (h90 extension). It may also contain files with f90 or F90 extensions. The

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

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/`.

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 TARGETS DEBUG=1 builds TARGETS in debug mode (use any of the targets defined above). Uses the `DebugCUDAFortranImplementation` in case of GPU compilation, which prints predefined data points for every kernel parameter after every kernel execution.

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

2.4 Test Interface

The following files are part of the sample test interface provided with **Hybrid Fortran**:

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

allAccuracy.sh Compares all Fortran 90 `.dat` files in the `./out` directory with a specified reference directory.

runTest.sh Use the file name of your executalbe as a command line parameter. This assumes your executable takes one argument for the problem size.

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. `cd` into the Hybrid Fortran directory you've now installed on your computer. It should contain this README file as well as the GPL licence texts.
3. Run `make example`. This creates a new project directory named `example`.
4. Run `cd example`.
5. 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.
6. Run `./test/example/example_cpu; ./test/example/example_gpu`. This should execute and validate both versions.
7. 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.
8. 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).

9. 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.

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

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

12. 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`.

2.6 Migration to Hybrid Fortran

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**:

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

2. Rename the new example project directory to your project name.

3. Delete `source/example.h90` and copy in your sourcecode. Please note that all loops that are to be run in parallel and their entire callgraph should be visible to the compiler in the source subdirectory. Your sourcecode 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. 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`.

4. Adjust `buildtools/MakesettingsGeneral` - the configuration options should be self explanatory.

5. Adjust `buildtools/Makefile` by adding your dependencies (and deleting the example). If your previous build system was already built on `GNU Make` it should be possible to copy in 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`.

6. Adjust the `FFLAGS` and `LDFLAGS` variables in `buildtools/MakesettingsGPU` and `buildtools/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.

7. 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.

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

   (a) Add calls to the `helper_functions` module procedures `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\out` folder.

   (b) Compile (`make; make install` in project directory) and run your program again to make sure the files are being created. Make sure that they actually contain data, for example by checking the file size.

   (c) Repeat steps 8a, 8b in your reference source code.

   (d) Compress the reference data created by the last step into a `./test/your-executable/ref.tar.gz` and make sure it is referenced correctly within `runTest.sh`.

(e) Run `runTest.sh` with your executable and check whether the two versions match by checking the output.

(f) From now on you can run the `runTests.sh` skript every time you have modified and recompiled your program.

9. Define the parallel regions that are to be accelerated by CPU[3] using a `@parallelRegion{appliesTo(CPU), ...}` directive. See sec. 1.2.2 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`.

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

11. Replace the loops of your parallel region with

12. 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.

13. Define the subprocedures within that call hierarchy that are to be ported as GPU kernels. These 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 depencies 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.

14. 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. 1.2 for details. In kernel subroutines and inside kernel subroutines you will need to declare local scalars and imported data as well. See sec. 1.2.1 for details.

15. 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. 1.2.1 for details.

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

16. 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.

17. Wrap the implementation sections of all your kernels with `@parallelRegion{appliesTo(GPU), ...}` / `@end parallelRegion` directives. See sec. 1.2.2 for details.

18. Test and debug on CPU by executing `make clean;make build_cpu DEBUG=1; make install_cpu; [run your tests]`. 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`.

19. Switch to GPU tests debug on GPU using `make clean;make build_gpu DEBUG=1; make insta` 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`.

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

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. 1.2. 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).

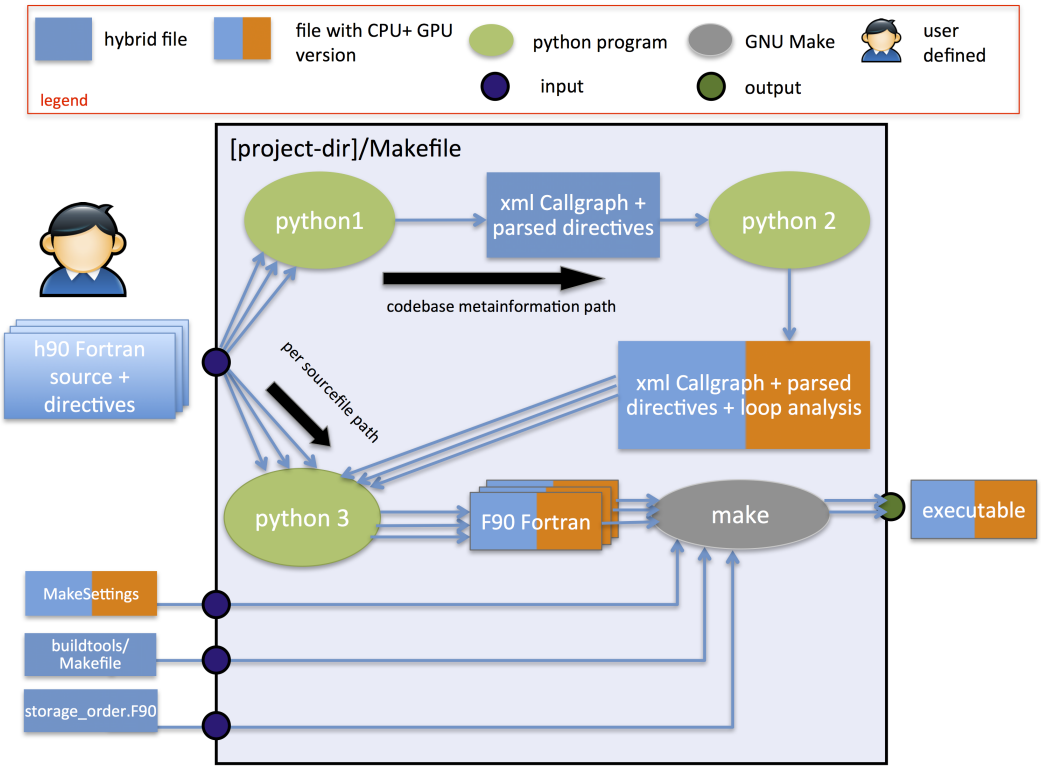

Figure 3.1: **Hybrid Fortran** Components and Information Flow.

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.

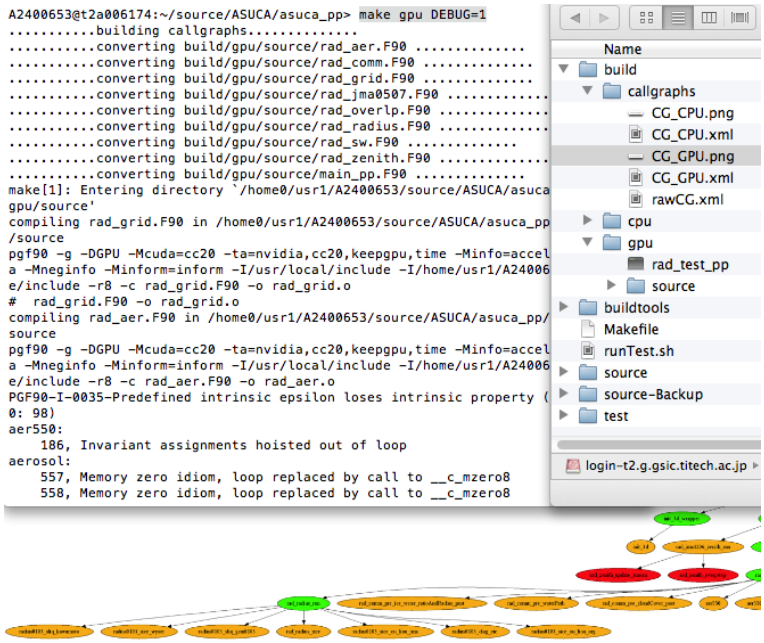

Figure 3.2: Screenshot of the **Hybrid Fortran** build system in action.

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:

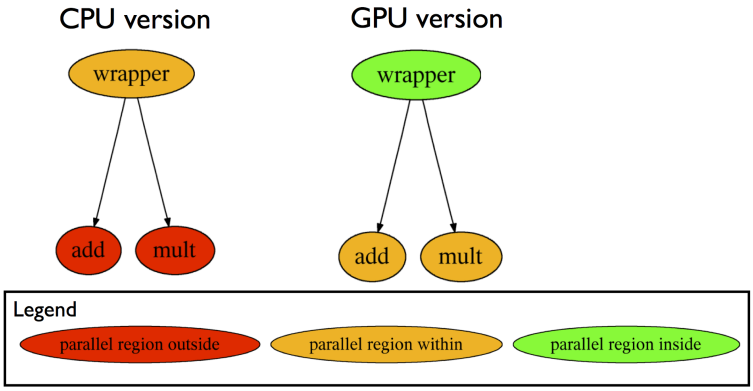

Figure 3.3: Condensed version of a simple callgraph, programmatically created by graphVizGraphWithAnalyzedCallGraph.py.

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.

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.

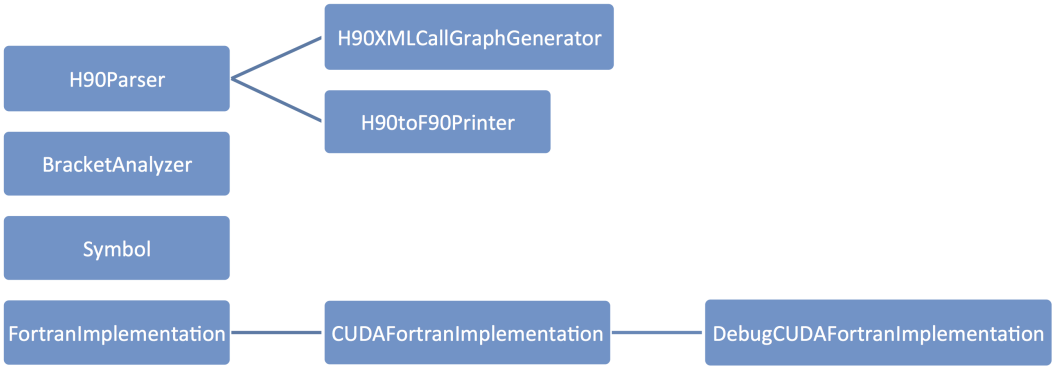

Figure 3.4: **Hybrid Fortran** Python Class Hierarchy.

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 1.2.3. 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.

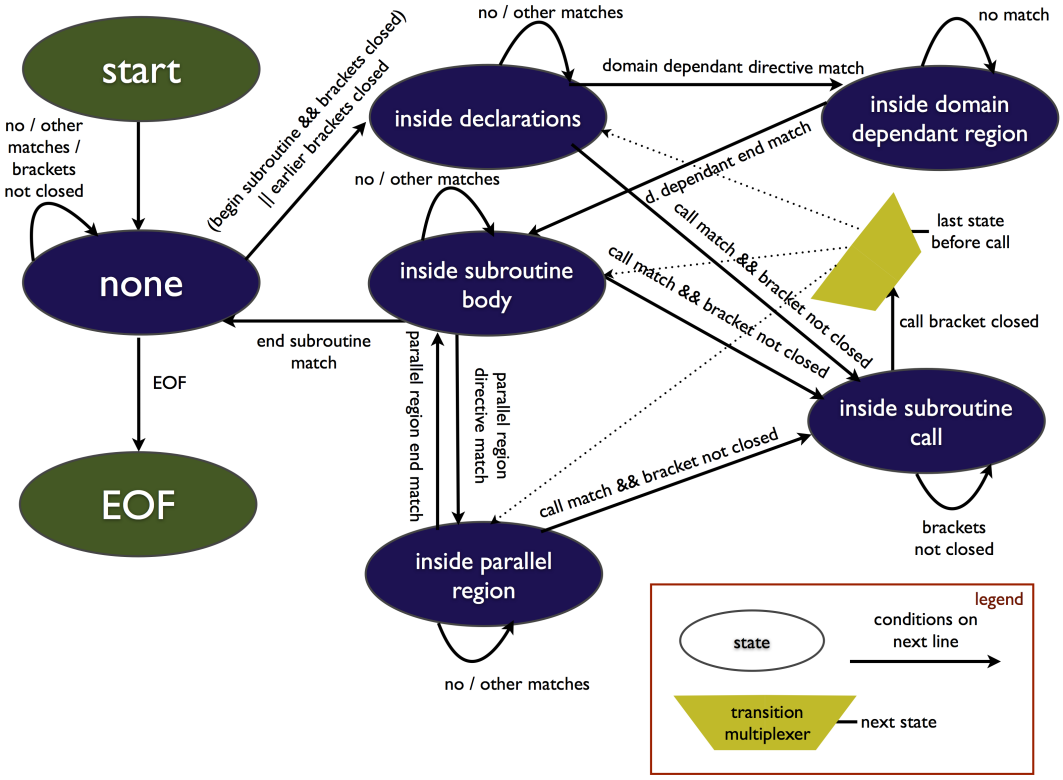

Figure 3.6: H90 Parser State Machine.

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.