Just In Time Compilation for a High-Level DSL

Nathan Dunne

1604486

3rd Year Dissertation

Supervised by Gihan Mudalige

Department of Computer Science

University of Warwick

2019–20

Abstract

TODO

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Key Words

High Performance Computing, Unstructured Mesh, Just-In-Time Compilation

Contents

List of Figures

1 Introduction

In the field of High Performance Computing (HPC), computers with processing power tens or hundreds of times greater than conventially available machines are used to solve (or appoximate solutions to) problems that would otherwise take an unwarrantable amount of time. Such computers have been required for some time to make use of a large degree of parallelism in order to complete with reasonable runtime: dividing work into independant subsections which can be executed simultaneously.

Many paradigms for executing parallel workloads have emerged over time: including vector instructions (SIMD), many and multi-core CPUs, clusters of interconnected computers, and General Purpose Graphical Processing Units (GPGPUs). Hardware which was originally specilised for graphical shader calculations through it's very high number of processing units, allows carrying out the same operation across a very large amount of data in parallel. This harware has been adapted in GPGPUs to perform non-specific operations that would normally have been done by the CPU.

There is always space for further benefit to be gained, and even small gains in runtime can have large impact on workload that take hours or days to complete. This report details an investigation into applying a new optimisation to the CUDA code generation library subsection of OP2, and secondarily benchmarking what performance gain if any it is able to provide. The optimisation is named "Just-In-Time Compilation" for its similarities to a comparable process often performed by compilers when runtime efficiency is desired.

1.1 Background Work

The OP2 library is an Open Source Domain Specific Langauge (DSL) which provides a high level abstraction for describing physics problems which can be abstracted to an Unstructured Mesh. A large proportion of HPC workloads involve approximating Partial Differential Equations (PDEs) to simulate complex interactions in physics problems, for example the Navier-Stokes equations for computational fluid dynamics, prediciting weather patterns, or computational electro-magnetics. It is usually necessary to discretise such problems across some form of mesh, either structured (regular) or unstructured.

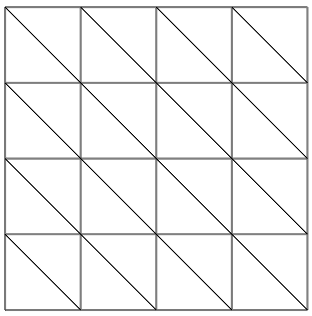


Figure 1: Tri-Structured Mesh

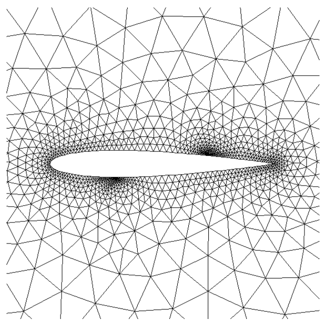


Figure 2: Airfoil Tri-Unstructed Mesh

Unstructed Meshes, such as Figure 2, use connectivity information to specify the mesh topology. The position of elements is highly arbritrary, unlike structured meshes where elements follow a regular pattern (Figure 1). A particular simulation might, for example, be approximating the velocity of a fluid in each cell based on the cells around it.

1.2 Motivations

The idea for this project was provided by my supervisor, Dr Gihan Mudalige - an Associate Proffesor in the University of Warwick Computer Science Department. It was pulled from the pool of uncompleted features for the OP2 project, and I selected it as it aligned with my interest in High Performance Computing, and similar experience with optimising exisiting codes.

Since OP2 is Open Source and freely available, the implementation I produce will become part of the library, allowing future contributors to build on my work. The project also allows me the opportunity to operate on a large codebase, where most university work is done largely within the confines of one's own code.

2 Research

2.1 OP2

The OP2 library is already able to generate optimised code for a number of platforms, including the increasingly popular NVidia CUDA for parallel programming on NVidia GPUs. The current implementation is compiled entirely ahead of time however, and is not able to optimise based on the input mesh.

The abstraction provided by OP2 allows scientists and engineers to focus on the description of the problem, and seperates the consideration for parallelism and efficiency into the back-end library and code generation. This is beneficial as it is unlikely that a single developer or team has the necessary expertise in both some niche area of physics with a non-trivial problem to be solved; and sufficient depth of knowledge in computer science to understand and utilise the latest generation of parallel hardware.

A further benefit is that existing solutions which make use of the OP2 library can be ported onto a new generation of hardware, by modifying only the OP2 backend library to support the new hardware, instead of every application individually. This portability can save both time and money in development if multiple different hardware platforms are desired to be used.

2.2 CUDA

2.3 Related Work

2.3.1 JIT

2.3.2 Similar Libraries

3 Specification

The implementation will require work in two main areas: The Python code generation script, and in the OP2 library itself which is implemented in both C and Fortran. Only the C library will be modified, due to developer familiarity. OP2 does also include code generation using MatLab, but Python is more preferable recently, due to its flexibility, and it's conveniant string manipulation capabilities.

The Python script will perform source-to-source translation. As input it will take the application files, which specify the structure of the program: declaring variables, and indicating where the loops parallelised by OP2 should be executed; and a kernel descriptor for each loop, which will describe the operation to be performed on each element of the set passed to the loop as a parameter. OP2 makes an important restriction that the order in which elements are processed must not affect the final result, to within the limits of finite precision floating-point arithmetic[7, p3]. This constraint allows the code generator freedom to not consider the ordering of iterations, and select an ordering based on performance.

From this a set of valid C files must be generated, to be compiled by a normal C compiler. In the case of this project the compiler will be the NVidia C Compiler (*nvcc*), as the code generated will be include CUDA.

It is important that the resulting executable compiled from the generated code produces outputs within some tolerance of the outputs generated by execting parallel loop iterations sequentially. Correctness is always a priority over performance for any compiler.

Furthermore, it will be ensured that the OP2 API is not be altered by any modifications to the library, to ensure that all exisiting programs using the API are able to seemless use the updated version.

3.1 Runtime Assertions

As discussed in Section 2.3.1 on Just-In-Time Compilation, the performance gain from this optimisation technique comes from making assertions at runtime which can only be made once the input is known. The application's input will be a mesh over which to operate, which includes a large amount of data, and opens up a number of runtime optimisations. The primary target for this project is "Constant Definition": turning values specified in the input as constants into `#define` directives for the C-Preprocessor. Other possible optimisations will be discussed in Section 7, Future Work.

3.2 System Model

Figure 3 describes the new workflow of the OP2 library, with the addition of Just-In-Time compilation. As before, code generation takes the application and loop files as input, and generates the Kernels and Modified Application Files. It also generates Optimised Kernels, where the code will only compile once the constants are known and defined by the pre-processor. These Kernel files are not used by the ahead of time compiler.

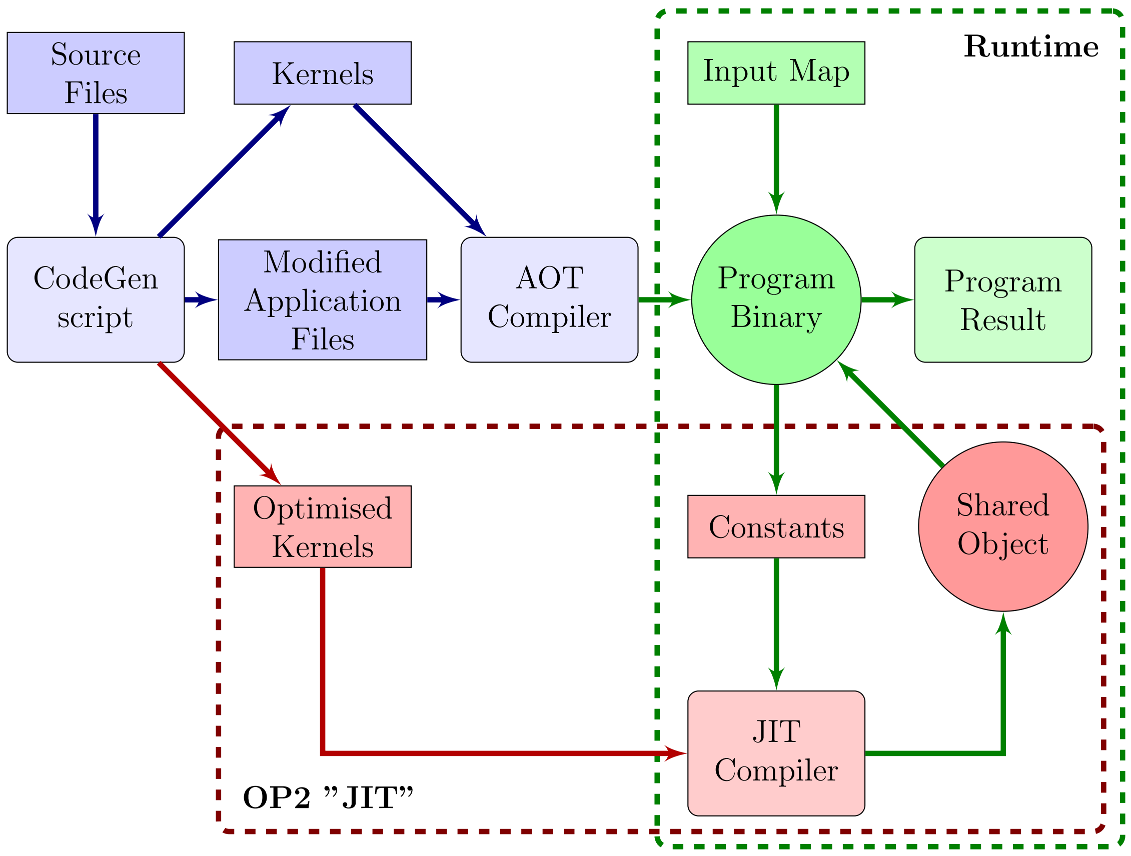


Figure 3: OP2 System Diagram with JIT Addition

The OP2 API function:

```
void op_decl_const(int dim, char *type, T *dat, char *name)
```

[6, p9]

Will be modified so that when called by the Program Binary it will generate a header file, where each of the constant values is added as a C `#define` directive, where previously it needed to copy these values to the GPU device's memory.

At runtime, the executable invokes the *nvcc* compiler again, to compile the Optimised Kernels which semantically include the constants header file, and link them into a Shared Object (Dynamically Loaded Library). This object is then loaded by the running executable, and the functions it provides are used instead of the unoptimised versions.

4 Implementation

The OP2 library is hosted open source on GitHub[4]. Instructions for obtaining the implementation completed for this report, and getting started with OP2 can be found in Appendix A.

The feature branch for this project, `feature/jit` was branched from `feature/lazy-execution` on 13th November 2019. The `lazy-execution` branch's last commit was in April 2018, and lagged behind the `master` branch somewhat. It was rebased onto `master` before any other changes were made.

This branch was created for developing a system to execute parallel loops when values are required rather than when called. This is done through an internal library function:

```
void op_enqueue_kernel(op_kernel_descriptor *desc)
```

op2/c/src/core/op_lazy.cpp [71-89]

Currently this function generates the constants header file, then executes the queued loop straight away. This process for calling parallel loops is used similarly throughout work done to enable Just-In-Time Compilation for CUDA, so that future efforts towards lazy execution can continue in the future on top of the JIT implementation.

4.1 Code Generation

The Python code generation script which forms the main body of the implementation can be found in: `translator/c/python/jit/op2_gen_cuda_jit.py`

Its entry point function is:

```
def op2_gen_cuda_jit(master, date, consts, kernels)
```

translator/c/python/jit/op2_gen_cuda_jit.py [102]

Which is called from `op2.py` in the parent directory - the same as the other code generation scripts, and its parameters are:

master: The name of the Application's master file

date: The exact date and time of code generation

consts: list of constants, with their type, dimension and name

kernels: list of kernel descriptors, where each kernel is a map containing many fields describing the kernel. The values may alter the way the code for that loop is generated.

The code generator first performs a quick check across all kernels to see if any use the Struct of Arrays feature [6, p13], or if all are using the default data layout. Then, it iterates over each kernel and generates both the Ahead-Of-Time (AOT) kernel file, and the Just-In-Time (JIT) kernel file simultaneously. A folder `cuda/` is created if it doesn't exist, and the files are generated with the following naming scheme:

- `AOT: cuda/[name]_kernel.cu`
- `JIT: cuda/[name]_kernel_rec.cu`

The AOT file is generated such that it doesn't just call the runtime compiler, but has the ability to execute without JIT as well. This is so that the JIT feature can be enabled or disabled by a compiler flag. A master kernels file Is also generated:

- `cuda/[application]_kernels.cu`

It contains shared functions, and include statements for each of the parallel loops' kernels.

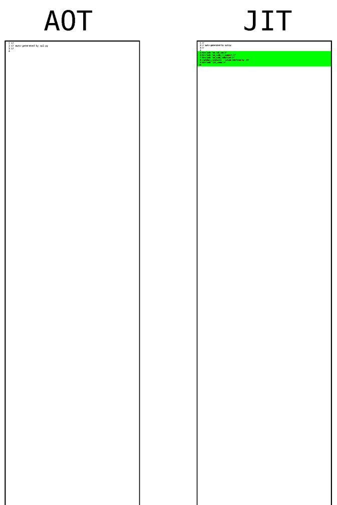
4.1.1 Kernel Files

As mentioned above, the code generator creates two files: AOT and JIT for each parallel loop. The following section details the functions generated, and examples in Figures 4-8 show the progression of each file for an example kernel. There is a summary on page 13 if this is not of interest.

1. JIT includes: The first part generated is simply the include directives required for the JIT compiled kernel. These are needed for JIT since they will be compiled individually, but aren't needed by the AOT kernel, as they will be included in the master kernels file:

```
#include ‘op_lib_cpp.h’
#include ‘op_cuda_rt_support.h’
```

Figure 4: JIT includes



```
#include 'op_cuda_reduction.h'

//global_constants
#include 'jit_const.h'
```

The `jit_const.h` file is also included, which will be generated at runtime (before the compiler is invoked) to contain a `#define` for all constants, to be processed by the preprocessor.

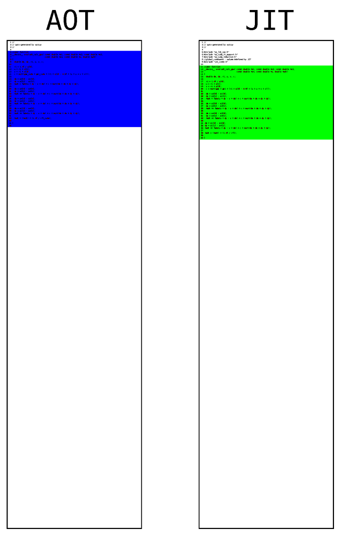
2. User Function: The User Function is the kernel operation specified by the user to be carried out on each iteration of the loop, so this function will run on the device (GPU) at least once for each set item. This function is given the signiture:

`__device__ void [name]_gpu( [args] )`

The `__device__` descriptor is used so that it will be compiled for the GPU, and can only be called from other device code. The kernel function found is checked to ensure it has the correct number of parameters, and, if requested, parameters are modified to utlise the struct of arrays layout.

The User Function is where any runtime assertions need to be made in order to benefit from the additional computation once inputs are known. In this report the assertion applied is using a `#define` for constants, so wherever the User Function references a constant it needs to be modifed.

Figure 5: User Function



AOT: In the Ahead-Of-Time kernel, only executed if JIT is not being used, the constant will need to read from the device's memory - having been copied there when it is defined constant. The copied version will have the identifier `[id]_cuda` to prevent a name collision, so all constant in the AOT kernel must be replaced with this pattern.

```
for nc in range(0,len(consts)):
  varname = consts[nc]['name']
  aot_user_function = re.sub('\\b' + varname + '\\b',
                             varname + '_cuda',
                             aot_user_function)
```

translator/c/python/jit/op2_gen_cuda_jit.py [905-907]

JIT: The JIT kernel is a little different: contants with a dimension of 1 (i.e. they contain only 1 value) can be left the unchanged, as the value will be defined under that same identifier. Multi-Value constants are slightly trickier - since values cannot be declared both `__constant__` and defined as external using `extern`[3, p126].

The eventual solution to this challenge was in two parts. For each index `N` of the constant array, a 1 dimensional constant would be defined with the name: `op_const_[id]_[N]`. All references to the constant where the index is a literal number can be replaced with the new identifier:

```
for nc in range(0,len(consts)):
  varname = consts[nc]['name']
  if consts[nc]['dim'] != 1:
    jit_user_function = re.sub('\\b' + varname + '\[([0-9]+)\]',
                               'op_const_' + varname + '_\g<1>',
                                jit_user_function)}
```

translator/c/python/jit/op2_gen_cuda_jit.py [931-934]

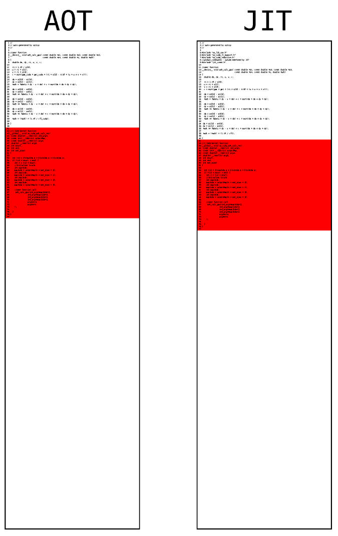
If the constant is accessed using a variable, expression, or anything other than a literal number, this system won't work however. In this case, an array is defined at the top of the function (only if required) with the identifier `op_const_[name]`, and the accesses are changed to match, so the access expression can remain and function as expected. This is only done where necessary, since allocating a new array can take time.

```
for nc in range(0,len(consts)):
  ...
  jit_user_function, numFound = re.subn('\\b' + varname + '\[',
                                        'op_const_' + varname + '[',
                                         jit_user_function)
  #At least one expression access
  if (numFound > 0):
    if CPP:
      #Line start
      codeline = '__constant__ ' + consts[nc]['type'][1:-1] +
                 ' op_const_' + varname + '[' +
                  consts[nc]['dim'] + '] = {'
      #Add each value to line
      for i in range(0,int(consts[nc]['dim'])):
          codeline += 'op_const_' + varname + '_' + str(i) + ', '
      codeline = codeline[:-2] + '};'

      jit_user_function = codeline+'\n\n'+jit_user_function
```

translator/c/python/jit/op2_gen_cuda_jit.py [931-944]

Figure 6: Kernel Function



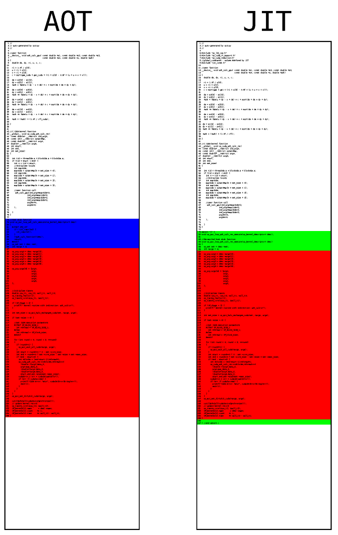
3. Kernel Function: From here onward, all code generated is based only on the kernel descriptor, and not the code that the user wrote for the body of the loop. The kernel function is the same in both files, and is executed on the GPU. It is declared `__global__` so that is exectuted on the device, but can be called from host (CPU) code:

```
__global__ void op_cuda_'+name+'( [args] )
```

The function arguments depend on whether any of the arguments are optional, and whether the loop uses indirection - accessing a set using an index which is the value in another set. OP2 enforces that the operands in the set operations are referenced through at most a single level of indirection [6, p4].

The function body also depends on whether there is indirection, as the indicies need to be retrieved from the inner map. A call is made to the user function generated above, then any reductions on arguments needs to be done. The supported reductions are: sum, maximum, and minimum[6, p11].

Figure 7: Host Function



4. Host Function: The purpose of the host function is to bridge the gap between the host and the device. It is CPU code, so runs on the host, but contains the CUDA call to the kernel function which will run on the GPU. While the function body is the same for both AOT and JIT: setting up arguments, timers, and block and thread sizes for the CUDA call; the function head differs, as shown in Figure 7.

AOT: In the Ahead-Of-Time kernel file, the C code generated for the head of the host function is as follows:

```
//Host stub function
void op_par_loop_[name]_execute(op_kernel_descriptor* desc)
{
  #ifdef OP2_JIT
    if (!jit_compiled) {
      jit_compile();
    }
    (*[name]_function)(desc);
    return;
  #endif

  op_set set = desc->set;
  int nargs = 6;
  ... //Identical Section
}
```

The function name is `op_par_loop_[name]_execute` because a pointer to this function will be queued by the lazy execution system mentioned previously in this Section, so this function actually executes the loop, whenever the lazy execution system should decide it needs to be executed. The decision of when to call the loop is outside the scope of this project, and currently a loop is simply called immediately after it is queued.

At the top of the function a decision is made as to whether JIT should be used, based on whether `OP2_JIT` has been defined. This allows JIT to be turned on and off through the used the compiler argument `-DOP2_JIT`. If JIT is enabled, then the compiler is invoked (if it hasn't been already), and the pointer to the newly compiler version of the function is executed instead.

If JIT is not enabled, this code will be ignored by the compiler, so the process will continue into the AOT host function, which causes it to stay whithin the AOT kernel file and never execute any code from the JIT file.

JIT: Contrasting this with the code generated for the JIT kernel file:

```
extern "C" {
void op_par_loop_[name]_rec_execute(op_kernel_descriptor* desc);

//Recompiled host stub function
void op_par_loop_[name]_rec_execute(op_kernel_descriptor* desc)
{
  op_set set = desc->set;
  int nargs = 6;
  ... //Identical Section
}

} //end extern c
```

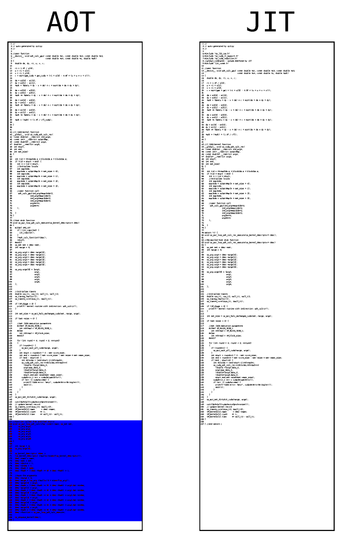
Firstly, since this function needs to be linked to the exisiting code as part of a dynamically loaded library, it is placed inside an `extern "C"` scope, to ensure C linkage, and prevent the compiler from "mangling" the name. Following that, the function, which is named `op_par_loop_[name]_rec_execute` ("rec" short for recompiled), will come to reside in the address of the `[name]_function` function pointer.

It will be executed after the runtime compiler has been invoked, as the replacement JIT-compiled host function, and make calls to the kernel and user functions in the same file as iteself, rather than those in the AOT file - allowing the optimisations made to be used.

5. Loop Function: The last section to be generated in the kernel files is the Loop Function, which is the entry point for the parallel loop:

```
op_par_loop_[name](char* name, op_set set, [args]... )
```

Figure 8: Loop Function



The application file will be modified by `op2.py` to contain an declaration for this function marked `extern`, to be linked againt this definition. Only the AOT kernel requires this, as previously mentioned the JIT host function acts as its entry point (Figure 8).

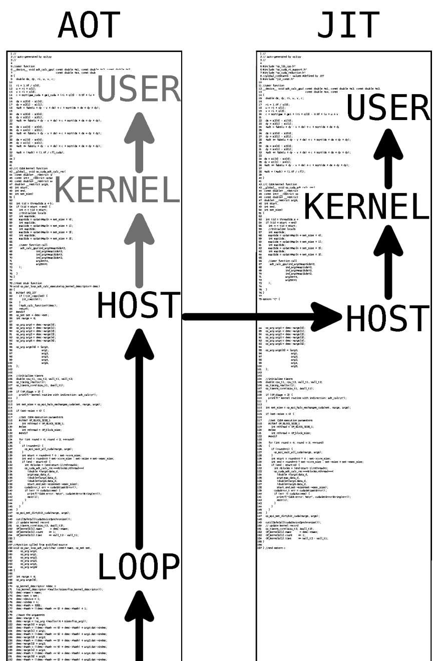
The purpose of this function is to generate the kernel descriptor, then make a call to:

```
void op_enqueue_kernel(op_kernel_descriptor *desc)
```

op2/c/src/core/op_lazy.cpp

[71-89]

As previously mentioned, the kernel descriptor and enqueue function were part of the work done to enable lazy execution in OP2, and not created as part of this project.



Summary:

To recap, the AOT and JIT kernel files are generated for each parallel loop, to be executed when that loop is invoked in the application file. Figure 9 has been included to make clear the data flow through the two files: starting in the Loop Function, which calls the AOT Host Function, where either the re-compiled version is invoked, or the original version is used if JIT is not enabled at compile time.

The `jit_compile()` function has not yet been defined, but this will be covered in the next section on the master kernels file.

4.1.2 Master Kernels File

The Master Kernels File: `cuda/[application]_kernels.cu` is the last file to be generated, once the kernels for each parallel loop have been completed. It ties up most of the remaining loose ends, as it contains shared functions for invoking the runtime compiler, and declaring constants. It also contains `#include` statements for each of the AOT kernel files, so that the application file can be linked against this file only at compile-time, and the linker will be able to find definitions for all the functions declared extern. The Makefile and compile process will be covered further in Section TODO.

At the top, the master kernels file includes the requried OP2 library files. It then defines a CUDA constant for each constant the user has defined, generated using the following python code:

```
for nc in range (0,len(consts)):
  if consts[nc]['dim']==1:
    # __constant__ [type] [name]_cuda;
    code('__constant__ ' + consts[nc]['type'][1:-1] + ' ' +
          consts[nc]['name'] + '_cuda;')
  else:
    if consts[nc]['dim'] > 0:
      num = str(consts[nc]['dim'])
    else:
      num = 'MAX_CONST_SIZE'

    # __constant__ [type] [name]_cuda[ [dim] ];
    code('__constant__ ' + consts[nc]['type'][1:-1] + ' ' +
          consts[nc]['name'] + '_cuda' + '['+num+'];')
```

translator/c/python/jit/op2_gen_cuda_jit.py [974-985]

Following this, the file contains definitions for two functions. The first is `op_decl_const_char`, which will be called from the application file to declare a constant identifier and value; and the second is `jit_compile` which will invoke the runtime compiler, load the generated DLL and create a function pointer for each re-compiled loop.

1. op_decl_const_char: This function is an OP2 library function which allows users to declare a value that will not change over the course of execution. It has the following signature, defined by the OP2 API[6, p9]:

```
void op_decl_const_char(int dim, char const *type, int size, char *dat, char const *name)
```

Two versions of the function are generated, one for AOT and one for JIT. The two functions are wrapped with pre-processor conditionals, so that only one of them will be visible to the compiler.

As before, `OP2_JIT` being defined is the test, so the JIT functionality can be enabled or disabled.

The AOT version of the function copies the value passed to it to the corresponding device constant using:

```
cudaMemcpyToSymbol(const void* symbol, const void* src, size_t count)
```

The copy default direction for this function is from host memory to device memory.

The JIT version instead invokes the internal library function:

```
void op_lazy_const(int dim, char const *type, int typeSize, char *data, char const *name)
```

op2/c/src/core/op_lazy.cpp [100-101]

Which maintains a de-duplicated list of constants, so that once they all have been declared the header file defining their values can be generated. As can be seen in the generated C code below, constants containing more than one value are declared as single values due to the issues with `extern __constant__` values described in Section 4.1.1 (2. User Function).

```
void op_decl_const_char(int dim, char const *type,
                        int size, char *dat,
                        char const *name)
{
  if (dim == 1) {
    op_lazy_const(dim, type, size, dat, name);
  }
  else {
    for (int d = 0; d < dim; ++d)
    {
      char name2[32];
      sprintf(name2, "op_const_%s_%d\0", name, d);
      op_lazy_const(1, type, size, dat+(d*size), name2);
    }
  }
}
```

generated by translator/c/python/jit/op2_gen_cuda_jit.py [1028-1046]

2. jit_compile: The other function generated is the `jit_compile` function, which is responsible for the actual recompilation of the JIT kernels, and making their functions available to the binary. It also uses the same timing library functions which gather data on the time spent in each parallel loop to determine how long the binary spends re-compiling, as this is important for performance measuring later.

The compiler arguments, library paths, and other required parameters are in this implementation handled by a make file which would need to be generated by the user. The contents of the makefile will be covered in the next section.

As can be seen below, the executable makes a system call to initiate a make command, and stores the result in a log file. If the compilation fails, an error message is printed, and the program exits early.

```
if (op_is_root()) {
  if (system("make -j [application]_cuda_rec &> jit_compile.log"))
  {
    // 0 indicated success
    printf("Error: JIT compile failed. \n
            - see jit_compile.log for details\n");
    exit(1);
  }
}
```

generated by translator/c/python/jit/op2_gen_cuda_jit.py [1071-1077]

It is expected that the make file will generate a shared object file named `cuda/airfoil_kernel_rec.so`. If this file does not exist the binary exits with an error, otherwise the recompiled function for each parallel loop is dynamically loaded using:

```
void *dlsym(void *restrict handle, const char *restrict name);
```

dlfcn.h

The function `op_par_loop_[name]_rec_execute` loaded, with the address stored in a void pointer with identifer `[name]_function`. We have seen this pointer before in Section 4.1.1 (4. Host Function).

Once this has been done for all loops, the wall clock time since the start of the `jit_compile` function is printed to the terminal.

4.2 Makefile

This implementation relies on GNU Make[9] to determine which compiler should be used, which parameters should be passed, and other options. There are a number of libraries required to build an OP2 binary, as covered in Appendix A, so only the recompilation target will be discussed here.

The binary expects there to be a Makefile in the directory it executes in, with a target: `[application]_cuda_rec` in order to work correctly. This is the target which will be compiled at runtime. As mentioned in the previous section, the result of making this target needs to be a a shared object file named `cuda/airfoil_kernel_rec.so`, which contains the recompiled loop functions.

The library object is produced by compiling each of the kernels individually, using the NVidia compiler `nvcc` from the NVidia CUDA Toolkit[2, 1] as the code contains CUDA, then linking them into a single object. It is necessary that the compiler flags include `--compiler-options -fPIC`. This passes a list of arguments to the underlying compiler, since nvcc only handles the CUDA code, and passes all host code compilation on to a C compiler. The argument to be passed down is `-fPIC`, to generate Position Independant Code, to allow the library function to execute correctly, regardless of the address at which it is loaded in memory.

4.2.1 Optional Functionality

By default, the JIT compilation functionality is enabled in the Makefile by setting the value of `$JIT` to `TRUE`. However, if the variable is set to anything else in the parameters of the make command, JIT will be disabled in the resulting executable. This is done with the following lines:

```
ifeq ($(JIT), TRUE)
        CCFLAGS    := $(CCFLAGS) -DOP2_JIT
        NVCCFLAGS  := $(NVCCFLAGS) -DOP2_JIT
        SUFFIX     := _jit
endif
```

Which adds a parameter to the C and CUDA compilers to define `OP2_JIT` for the preprocessor, and appends "_jit" to the name of the executable generated.

The target `cuda/airfoil_kernels_cu.o` is also declared PHONY, so that it is always recompiled even if the file already exists, otherwise this make flag would not function correctly, and a JIT enabled version of this file may be used when the user intended to recompile it with JIT disabled.

5 Testing

Throughout development, an example application was used to test code generation, and verify the results. The application has been used previously for validating generated OP2 code, as it makes use of all the key features, including having both direct and indirect loops. It is called *airfoil*, and is a computational fluid dynamics solver which models the air flow around the cross section of a aeroplane wing, using unstructured grid to discretise the space. A document detailing the airfoil code is available on the OP2 website [5].

5.1 Test Plan

Since the project is centered around code generation, the generated code must of course be valid - and compile without error. It is possible this could vary between compilers, so in this report results are primarily gathered using the Intel C/C++ Compilers, and the Intel MPI library. The Nvidia C Compiler `nvcc` is used to compile the CUDA device code sections, but it will refer all host code compilation to `icpc`.

Once the generated code compiles successfully, the most important result to achieve is that the compiler executable creates an output that is within tolerance of the expected value. Performance is still important - and the goal of this project is to investigate whether this technique does provide any performance benefit - but any perfomance increase that incurs unaccebtable deviation from the expected result is not a useful benefit. Section TODO on Benchmarking will cover the performance analysis.

With this in mind, the airfoil application code includes a test of the result after 1000 iterations against the expected outcome, and prints the percentage difference. A difference of less than 0.00001 is considered within tolerance due to the potential for minor floating point errors, and therefore a passing test.

The initial state for the test is a folder with the files listed in Figure 11a (p23). The main application file is `airfoil.cpp` which contains OP2 API calls, and the structure of the program. The 5 header files contain the user functions for the respective parallel loop with the same name, and `new_grid.h5` is the input data in the Heterogenous Data Format (HDF5 [8]) file format.

5.2 Test Results

5.2.1 Code Generation

To test the code generation, the python script `op2.py` is called in the directory, passing the main application file `airfoil.cpp` as an argument, as well as the string `JIT` to make sure the correct code generation scripts are called.

```
> python2 \$OP2_INSTALL_PATH/../translator/c/python/op2.py airfoil.cpp JIT
```

After running this command, the expected outcome is that a new file: `airfoil_op.cpp` is created in the directory, and a directory named `cuda/` will be created with eleven files in it: Two for each of the five parallel loops, as described in Section 4.1; and a single master kernels file named `airfoil_kernels.cu`.

This test is considered a pass if these files exist, as their contents is validated as correct by the next tests passing. A folder called `seq/` is also created by the translator script `translator/c/python/jit/op2_gen_seq_jit.py`, which was not completed as part of this project, but part of the `feature/lazy-execution`, the parent branch of `feature/jit`.

Figure 11b shows the folder after running the above command. The test is considered **PASSED**.

5.2.2 Ahead-of-Time Compilation

Compilation with both JIT enabled, and JIT disabled needs to be tested.

1. JIT Enabled:

Ahead of time compalation is considered a success if the compilation completes successful, without any errors. In the `airfoil_JIT` folder this is done using the Makefile and the `airfoil_cuda` target, and JIT is enabled in the Makefile by default, so the command to compile the JIT enabled version is simply:

```
> make airfoil_cuda
```

This target includes compiling all of the AOT kernel files into a single binary, then compiling the modified master application file `airfoil_op.cpp` and linking the two together to produce the executable, named `airfoil_cuda_jit`. The command executed by the Makefile is:

```
nvcc -gencode arch=compute_60,code=sm_60 -m64 -Xptxas=-v --use_fast_math -O3
     -lineinfo -DOP2_JIT -I/home/cs-dunn1/cs310/OP2-Common/op2//c/include
     -I/home/cs-dunn1/parlibs/phdf5/include -Icuda -I. -c
     -o cuda/airfoil_kernels_cu.o cuda/airfoil_kernels.cu
```

Some warnings are generated, but there are no compilation errors. Figure 11c shows the folder after running the above command. The test is considered **PASSED**.

2. JIT DISABLED:

To build the executable with JIT compilation disabled a parameter needs to be added to the make command:

```
> make airfoil_cuda JIT=FALSE
```

Which will prevent cause the compiler to ignore the call to `jit_compile()` in the Host Function, and instead continue using the AOT kernel file. The only difference in expected outcome from the previous test is that the executable will be named `airfoil_cuda`, without the "_jit" suffix.

Again some warnings are generated, but there are no compilation errors. The Figure is omitted due to similarity to Figure 11c . The test is considered **PASSED**.

5.2.3 Just-in-Time Compilation

Testing Just-In-Time Compilation requires only that when executed the binary does not exit early with an error. As described previously in Section 4.1.2 there exists a check for success in the code, and the terminal output of the compilation is dumped to a file named `jit_compile.log`.

The test can be considered successful if the executable prints the compilation duration to the console output, and confirmed as a success by checking the compiler log for errors.

```
> ./airfoil_cuda_jit
  ...
  JIT compiling op_par_loops
   Completed: 5.588549s
```

In Figure 11d, which shows the airfoil folder after JIT compilation has completed successfully, we can see that there is now an object file for each of the parallel loops, as well as a new shared object in the `cuda/` folder. Some other miscellaneous files have also been generated, including the compilation log file and the optimisation report from `icpc`.

The JIT compilation log file does not contain any errors, and the expected files have been generated, as can be seen in /Figure 11d . The test is considered **PASSED**.

5.2.4 Output

The final test is that the result of the execution is within tolerance of the expected outcome. This test confirms that the contents of the file not just valid but also correct. The outputs are shown below:

Figure 10: Console Output from both binaries

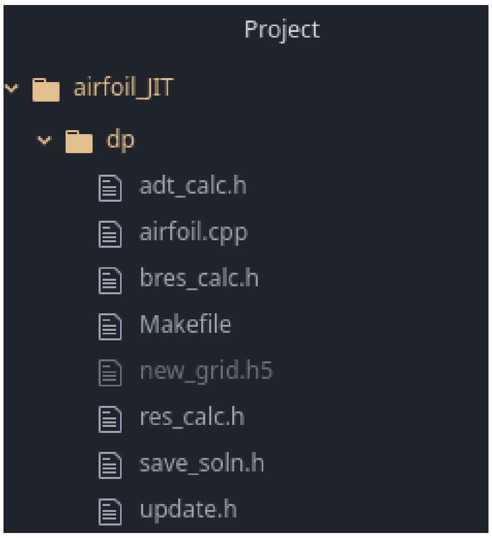
JIT		Enabled	Disabled
Iterations	100	5.02186×10^{-4}	5.02186×10^{-4}
	200	3.41746×10^{-4}	3.41746×10^{-4}
	300	2.63430×10^{-4}	2.63430×10^{-4}
	400	2.16288×10^{-4}	2.16288×10^{-4}
	500	1.84659×10^{-4}	1.84659×10^{-4}
	600	1.60866×10^{-4}	1.60866×10^{-4}
	700	1.42253×10^{-4}	1.42253×10^{-4}
	800	1.27627×10^{-4}	1.27627×10^{-4}
	900	1.15810×10^{-4}	1.15810×10^{-4}
	1000	1.06011×10^{-4}	1.06011×10^{-4}
Accuracy		$2.484679129111100 \times 10^{-11}\%$	$2.486899575160351 \times 10^{-11}\%$

The table shows the result every 100 iterations, as printed to the terminal by the binary, as well as the percentage difference from the exact expected value.

Both outputs are well within the tolerance of $1 \times 10^{-5}\%$. The test is considered **PASSED**.

Figure 11: Files in Application Folder

(a) Input Files



(b) After Code Generation

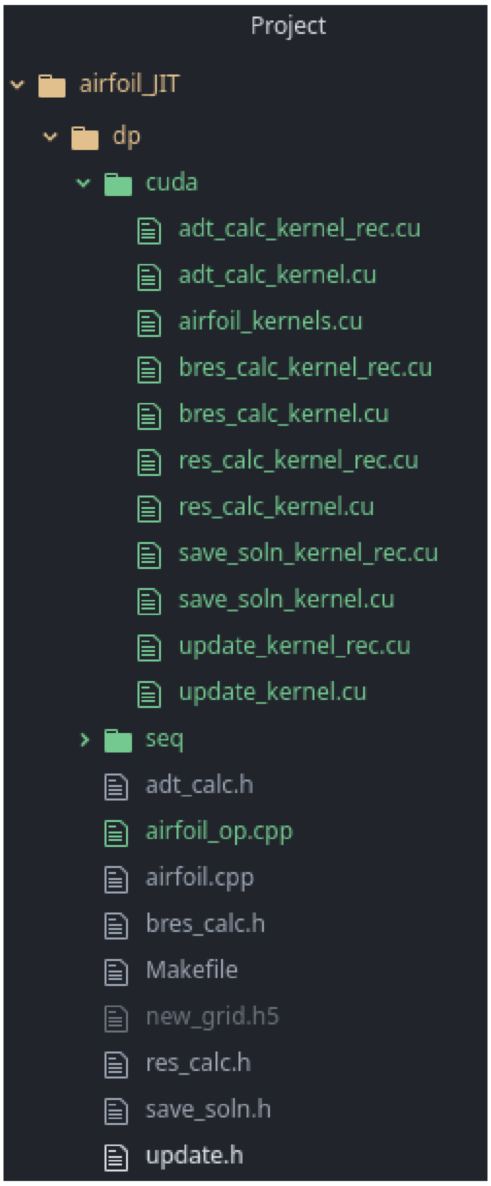
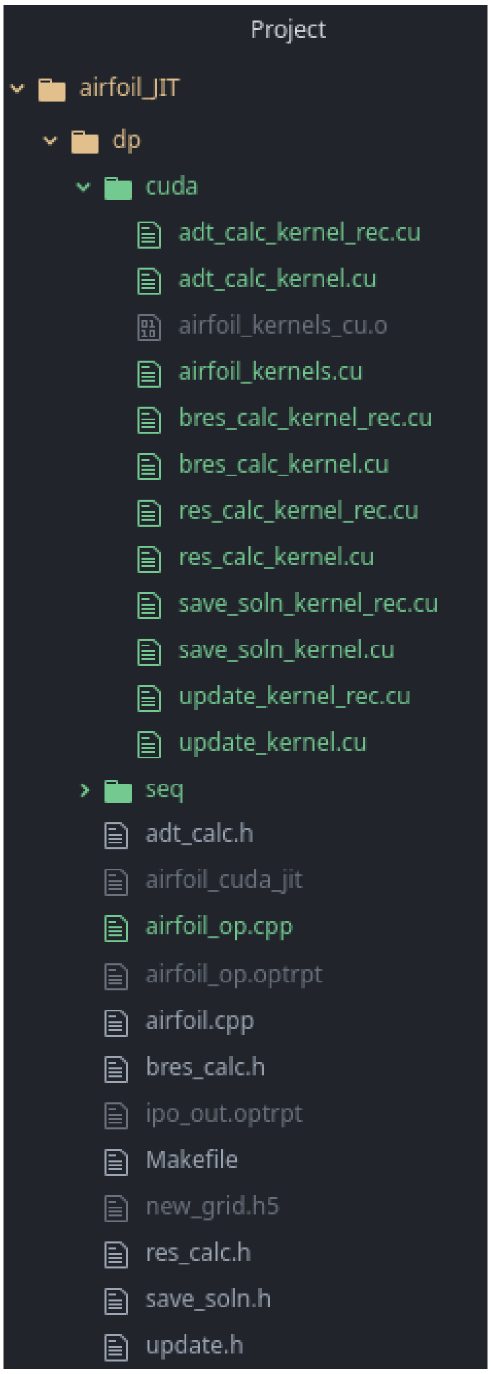


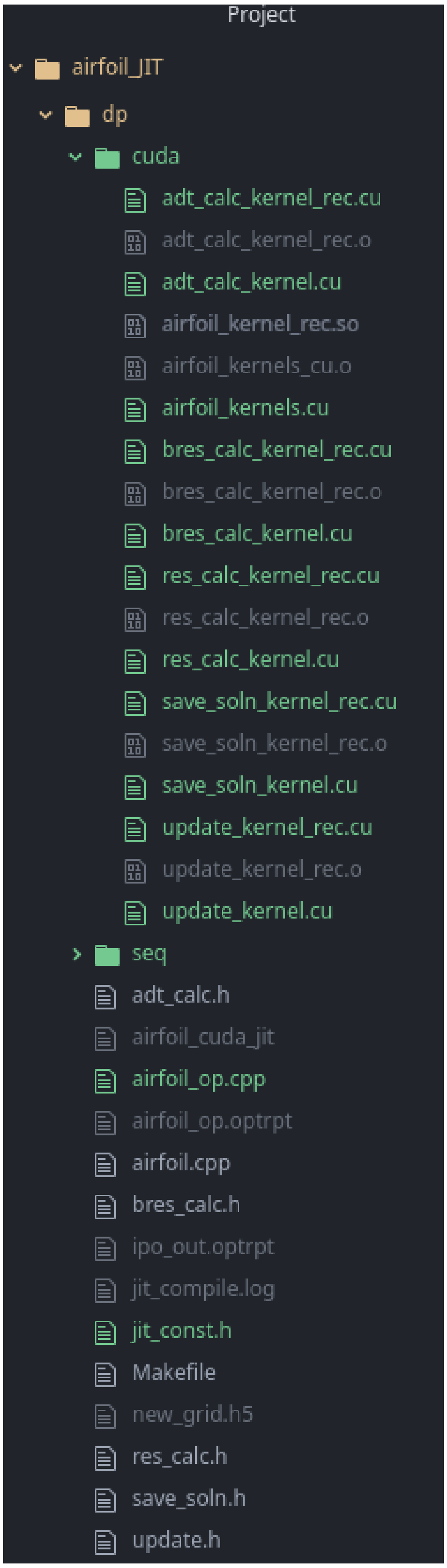
Figure 11: Files in Application Folder

(c) After AOT Compile



```
Project
airfoil_JIT
  dp
    cuda
      adt_calc_kernel_rec.cu
      adt_calc_kernel.cu
      airfoil_kernels_cu.o
      airfoil_kernels.cu
      bres_calc_kernel_rec.cu
      bres_calc_kernel.cu
      res_calc_kernel_rec.cu
      res_calc_kernel.cu
      save_soln_kernel_rec.cu
      save_soln_kernel.cu
      update_kernel_rec.cu
      update_kernel.cu
    seq
    adt_calc.h
    airfoil_cuda_jit
    airfoil_op.cpp
    airfoil_op.optrpt
    airfoil.cpp
    bres_calc.h
    ipo_out.optrpt
    Makefile
    new_grid.h5
    res_calc.h
    save_soln.h
    update.h
```

(d) After JIT Compile



```
Project
airfoil_JIT
  dp
    cuda
      adt_calc_kernel_rec.cu
      adt_calc_kernel_rec.o
      adt_calc_kernel.cu
      airfoil_kernel_rec.so
      airfoil_kernels_cu.o
      airfoil_kernels.cu
      bres_calc_kernel_rec.cu
      bres_calc_kernel_rec.o
      bres_calc_kernel.cu
      res_calc_kernel_rec.cu
      res_calc_kernel_rec.o
      res_calc_kernel.cu
      save_soln_kernel_rec.cu
      save_soln_kernel_rec.o
      save_soln_kernel.cu
      update_kernel_rec.cu
      update_kernel_rec.o
      update_kernel.cu
    seq
    adt_calc.h
    airfoil_cuda_jit
    airfoil_op.cpp
    airfoil_op.optrpt
    airfoil.cpp
    bres_calc.h
    ipo_out.optrpt
    jit_compile.log
    jit_const.h
    Makefile
    new_grid.h5
    res_calc.h
    save_soln.h
    update.h
```

5.3 Benchmarking

6 Evaluation

7 Future Work

8 Conclusion

References

[1] NVidia Corporation. *CUDA toolkit.*
URL: `https://developer.nvidia.com/cuda-toolkit` (visited on 04/01/2020).

[2] NVidia Corporation. *NVidia C Compiler.*
URL: `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html` (visited on 04/01/2020).

[3] NVidia Corporation. *NVidia CUDA C Programming Guide*. English. Version 4.2. NVidia. 160 pp.

[4] OP-DSL. *OP2-Common.* https://github.com/OP-DSL/OP2-Common.

[5] M.B. Giles G.R. Mudalige I. Reguly. *OP2 Airfoil Example*. 2012.
URL: `https://op-dsl.github.io/docs/OP2/airfoil-doc.pdf` (visited on 11/05/2019).

[6] M.B. Giles G.R. Mudalige I. Reguly. *OP2 C++ User Manual.*
URL: `https://op-dsl.github.io/docs/OP2/OP2_Users_Guide.pdf` (visited on 11/05/2019).

[7] M.B. Giles G.R. Mudalige I. Reguly. *OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures*. 2012.

[8] The HDF Group. *HDF5.*
URL: `https://www.hdfgroup.org/` (visited on 04/04/2020).

[9] Free Software Foundation Inc. *GNU Make.*
URL: `https://www.gnu.org/software/make/` (visited on 04/01/2020).

Appendices

A Getting Started with OP2

Acknowledgements

TODO