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 hundreds of times greater than conventially available machines are used to solve or approximate complex problems. Such computers have been required for some time to utilise parallelism, in order to find solutions within a reasonable runtime.

Many paradigms for executing parallel workloads have emerged over time. Recently, General Purpose Graphical Processing Units (GPGPUs) have become an increasingly popular hardware architecture, having originally been concieved as specilised hardware for graphical shader calculations. GPU's high number of parallel processing units allow a high degree of parallelism, as well as being able to execute operations usually done by the CPU. Passing work to the GPU device can provide very significant speed-up over sequential execution.

Commonly, a single developer or team is unlikely to have both the necessary expertise in both a niche area of physics with a non-trivial problem to be solved; and also sufficient depth of knowledge in computer science to understand the latest generation of parallel hardware. For this reason the OP2 framework was created: to provide a high level abstraction in which HPC applications can be written, and seperate the application code from the optimisation requirements. OP2 already is able to generate optimised code for a number of backends from an application file.

This report details an investigation into applying a new optimisation to the CUDA code generation in OP2, and the process of 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 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 was selected because it aligned with my interest in High Performance Computing, and previous 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.

1.2 Background Work

In order to become comfortable with the OP2 framework, and provide a useful contribution, it is important to understand the domain of problems for which it was created: Unstructured Mesh Solvers.

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, or prediciting weather patterns. It is usually necessary to discretise such problems, dividing 2D or 3D space into a number of cells. Depending on its structure this mesh can be described as either structured (regular) or unstructured.

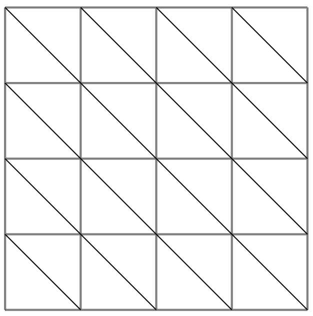


Figure 1: Tri-Structured Mesh

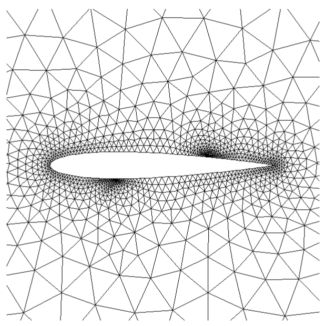


Figure 2: Airfoil Tri-Unstructed Mesh

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, and at every time step re-calculating this value based on the values of cells around it.

Since a structured mesh can be represented using an unstructured mesh, OP2 can support either - however unstructured meshes are more common.

1.3 Report Structure

The rest of this report is structured as follows: In Section 2 (p4) the research done to inform the work in this project is discussed, followed by a Specification in Section 3 (p12). Section 4 (p15) details the Implemention and expected results, Section 5 (p27) explains the Testing and Benchmarking of the project, and Section 6 (p37) an Evaluation of the work completed, including Project Management (p37). Lastly, Section 7 (p37) contains a discussion on Future Work which could build on top of what was done for this report, and Section 8 (p37) an overall Conclusion

2 Research

2.1 CUDA

Since this project will require the automatic generation of CUDA from sequential code, it is important to introduce the NVidia hardware, and the C API which will be utilised. The information is pulled from the *NVidia CUDA C Programming Guide* [5], which will continue to be used for reference throughout this report.

2.1.1 Hardware

GPU hardware is specialised to perform compute-heavy workloads, where the ratio of arithmetic operations to memory operations is high. It achieves this specialisation by devoting more resources to actual data processing, compared to a traditional CPU architecture, which optimises more for flow control and data caching. Figure 3 constrasts the two approaches:

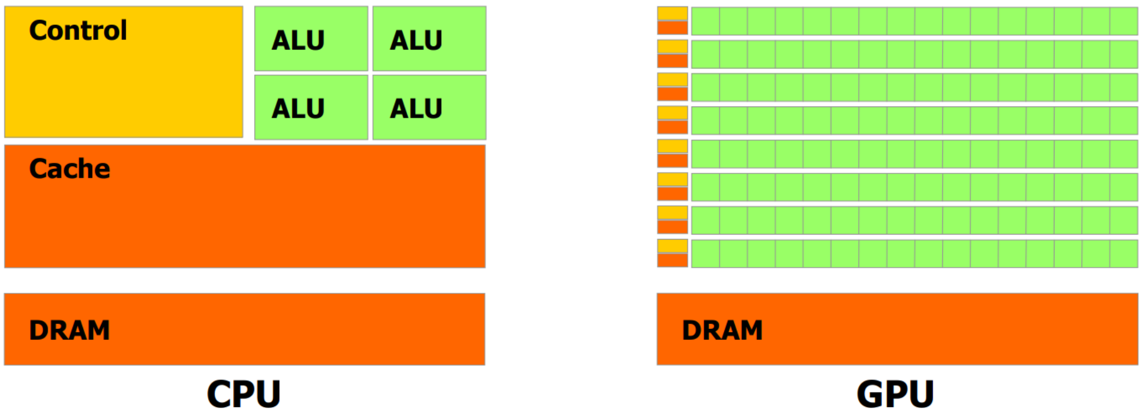


Figure 3: Architecture Comparison. Diagram from [5, p3]

This structure allows GPUs to excel at performing parallel tasks requiring the same operations to be performed on large sets of data, and therefore well suited to the needs of OP2, where a particular function might need to be applied to all edges, or cells, or nodes in a given mesh.

Workloads executed on a GPU are divided among a Grid of Blocks, where each Block contains a number of Threads. To allow for simple mapping from the problem to the thread ID, the Block Identifiers, and Thread Identifiers within each block, can be 1D, 2D or 3D [5, p9]. Figure 4 shows an example where 2D identifiers are used.

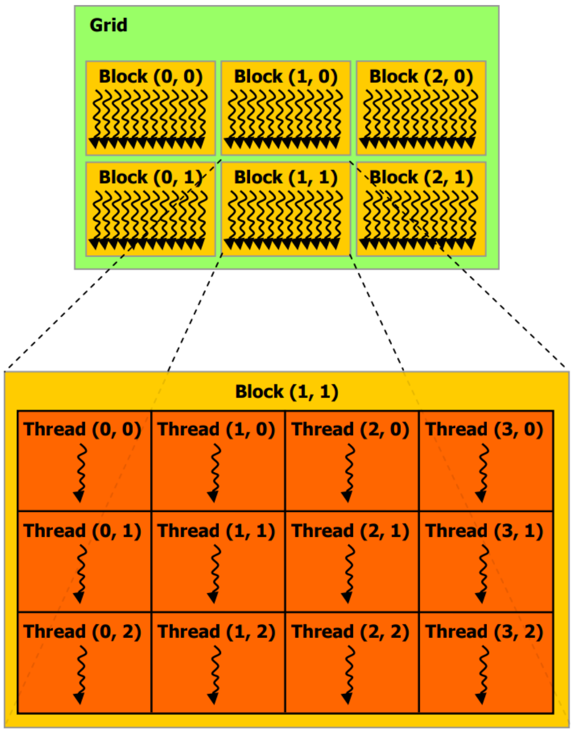


Figure 4: 2D grid of Blocks and Threads. Diagram from [5, p9]

The use of this thread layout will become more clear in the next section, where the invocation of a function to run on a GPU device is described.

2.1.2 Programming Interface

The CUDA C API provides two function type quantifiers that will need to be used in the generated code:

- `__device__`
- `__global__`

Both indicate that the function should be compiled to *PTX*, the CUDA instruction set architecture [5, p15], however the difference is that a function declared `__global__` can be invoked from host (CPU) code, or device (GPU) code; whereas a `__device__` function can only be called from code already executing on the device [5, p81].

Global functions therefore act as a sort of entry point into device code. They are called using new notation which allows the user to specify the requested number of blocks and threads per block:

```
function<<< num_blocks, threads_per_block >>>( [arguments...] )
```

Where the data type of `num_blocks` and `threads_per_block` can be either of `int` (1D), or `dim3` (1D,2D or 3D) [5, p9]. The function body will then be executed `num_blocks` $\times$ `threads_per_block` times. The Kepler Architecture has an upper limit of 2048 total threads per multiprocessor [8], for example 8 blocks of 16×16 threads (2 dimensions of thread IDs), which is a common choice.

Inside the function body, built-in variables can be used to access the thread and block ID of each thread as it executes, and from this determine the work which a certain thread should carry out. Appendix A is a CUDA program written during research to build familiarity which utilises these constructs and ideas, based on an NVidia tutorial.

In the next section, the OP2 library's exisiting code generation script will be discussed, which can produce optimised code executable on a GPU. This code generator will inform some parts of the new code generation script being produced for this report, with the Just-In-Time Compilation functionality as an addition and hopefully an improvement.

2.2 OP2

2.2.1 Exisiting Work

OP2 is an "active library" framework [13], which takes a single application code written using the OP2 Application Programming Interface (API), embedded in either C or Fortran. It uses source-to-source translation to produce multiple solutions, each for different optimised parallel implementations - including CUDA for executing operations on NVidia graphics cards. The generated code is then linked against the OP2 library and compiled to produce an executable for the application, which can run on the desired hardware. It is the extra step of code generation that makes OP2 an "active" library, compared to conventional software libraries.

Since this project is focussed on the GPU back-end, and specifically CUDA for NVidia GPUs, the journal article on the design of OP2 for GPU architectures [20] is necessary background material, as it covers a lot of important details from the GPU implementation.
The paper is summerised in the following section:

1. **Designing OP2 for GPU architectures:**

This article, originally published in the Journal of Parallel and Distributed Computing in 2013, describes the ey design features of the current OP2 library for generating efficient code targeting GPUs based on NVIDIA's Fermi architecture. It is worth noting that Fermi is no longer the latest architecture, and the code generation process has been modified since publication, however the article still provides useful information.

One of the key points made in the paper is on the managing of data dependencies (p1454), and the solutions proposed for avoiding errors when incrementing indirectly referenced array, where two edges end up updating the same node. Solutions include an owner of node data which performs the computation; colouring of edges such that no two edges of the same colour update the same node; and atomic operations. In the implementation for this project, atomic operations, which perform read-modify-write operation on one 32-bit or 64-bit word residing in global or shared memory [5, p96], as used to solve this issue.

The paper also introduces the consideration for data layout in memory. Figure 5 demonstrates the different layouts possible when there are multiple components for each element. The paper concludes that the struct-of-arrays layout reduces the total amount data transferred to and from GPU global memory, in some cases by over 50%.

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

(a) Array of Structs (AOS) layout

0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

(b) Struct of Arrays (SOA) layout

Figure 5: Data layouts. Diagram reproduced from [20]

The SoA layout is enabled by setting the value of an environment variable: `OP AUTO SOA=1` prior to code generation [12, p13].

The existing solution is able to generate optimised CUDA code for a parallel loop; given the set of nodes, edges, cells, or any other collection which it operates over; where the resulting code can assign a set element to a GPU thread where it will be processed by the loop body. It is important to note that the existing implementation for CUDA code generation produces a solution that is compiled entirely ahead of time, i.e. prior to the inputs being known, and therefore is not able to make optimisations based on the mesh input. This project aims to fill this space, and determine if there is benefit to be gained from such optimisations.

Since OP2 enforces that the order in which the function is applied to the members of the set must not affect the final result [12, p4], the consideration for data dependencies when generating code is largely removed, and therfore loop iterations can be scheduled in any order, based on best performance.

2.2.2 OP2 Applications

There are a number of industrial applications that have been implemented using OP2, which would immediately benefit from any optimisation of generated code, including Airfoil [11] - a non-linear 2D inviscid airfoil code; Hydra [16] - Rolls Royce's turbomachinery simulator, and Volna [15] - a finite-volume nonlinear shallow-water equation solver used to simulate tsunamis.

They make use of the abstraction provided by OP2, allowing scientists and engineers to focus on the description of the problem, and seperates the consideration for parallelism and data-movements into the OP2 library and code generation.

A further benefit is that such applications could be ported onto a new generation of hardware which might be developed in the future, with only the OP2 backend library needing to be modified 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.

Later in this report we will see Airfoil used as a benchmark, to determine whether the new optimisation presented in the report is likely to provide benefit to other OP2 applications.

2.2.3 Similar Libraries

2.3 Related Work

2.3.1 Just-In-Time Compilation

1. **Java:**

The term "Just-In-Time Compilation" is most commonly associated with the Java Runtime Environment (JRE), as it is integral to the Java Virtual Machine (JVM). In this case, Java compiles code into platform independant "bytecode", then at runtime this bytecode is compiled again by the JVM into native code, which can be optimised for the machine it will run on. It can also take the program's inputs into account, since they will be known and fixed at runtime.

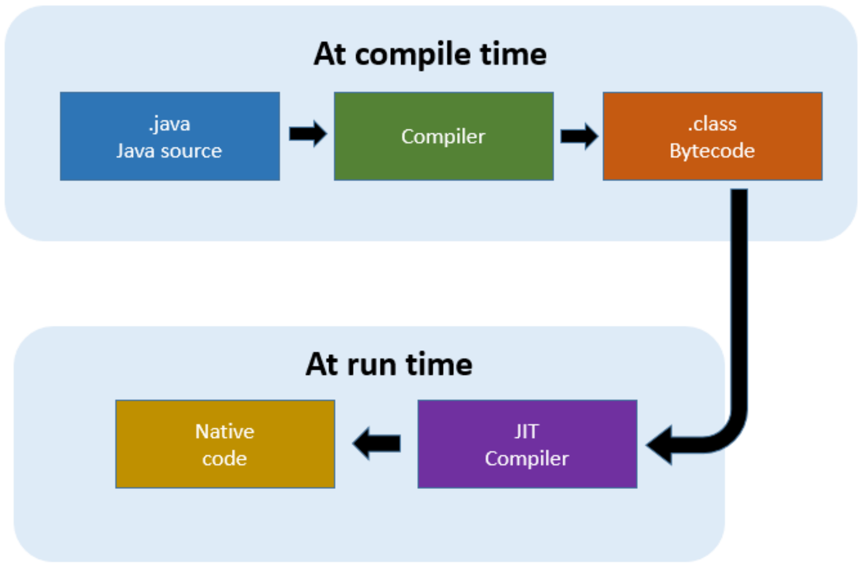


Figure 6: JIT Compilation in the JVM. Diagram from [19]

Chapter 4 of [**javaPerf**] contains further detail on Java Performance and the JIT compiler.

This approach is not exactly equivalent to the approach taken in this report, as will be discussed further in Section 3 (Specification), however the key idea of recompiling code at runtime to obtain performance remains. In the implementation presented here, there is no intermediate code, but rather an alternative source file which is able to utilise assertions made using the input data is compiled and used, in place of equivalent but unoptimsed functions compiled ahead of time.

2. easy::JIT:

easy::JIT [18] is a library created by *Juan Manuel Martinez Caamaño* and *Serge Guelton* of *Quarkslab* [1], and fits closer to the goal of this report. It targets C++ code, and utilises `clang` [2] as the compiler, and therefore can make use of the LLVM's intermediate representation. *easy::JIT* differs does also from this project however, as it utilises code generation at runtime, and a cache of code to ensure this does not need to be done on every execution. The OP2 implementation discussed in this report will generated the majority of the code ahead of time, as this is a slow process.

Applications developed for OP2 are not currently limited to only LLVM-based C compilers, although a translator using LLVM Intermediate Representation to replace the current Python translator is in development [10]. The implementation completed for this report will also seek to be compiler agnostic, and therefore will not utilise LLVM.

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[13, 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

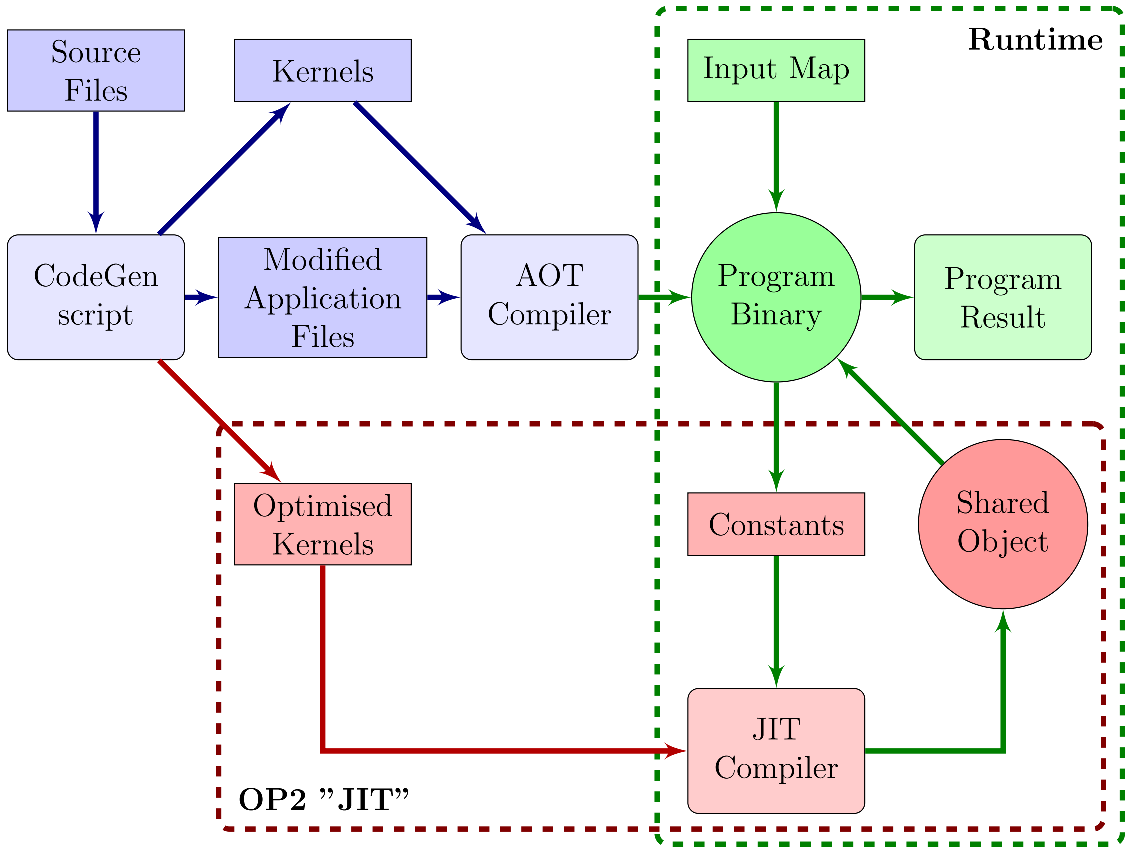


Figure 7: OP2 System Diagram with JIT Addition

Figure 7 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.

The OP2 API function:

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

[12, 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[9]. Instructions for obtaining the implementation completed for this report, and getting started with OP2 can be found in Appendix B.

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 [12, 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

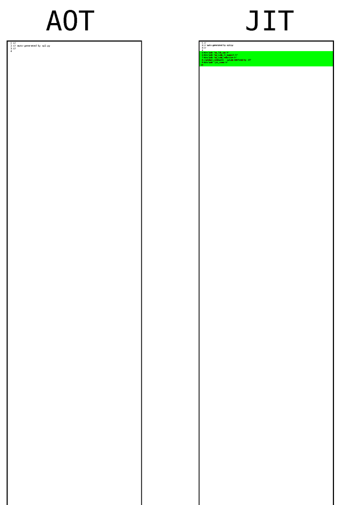
As mentioned above, the code generator creates two files: AOT and JIT for each parallel loop. The following section details the functions generated, and examples in Figures 8-12 show the progression of each file for an example kernel. There is a summary on page 21 if this is not of interest.

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 8: 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:

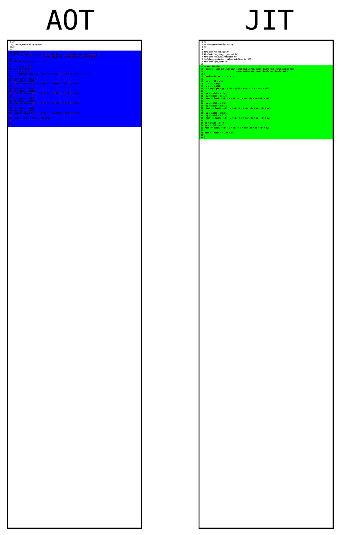
The User Function is the kernel operation specified by the user to be carried out on each iteration of the loop, so this function will run on the device (GPU) at least once for each set item. This function is given the signiture:

`__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 9: 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`[5, 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]

3. **Kernel Function:**

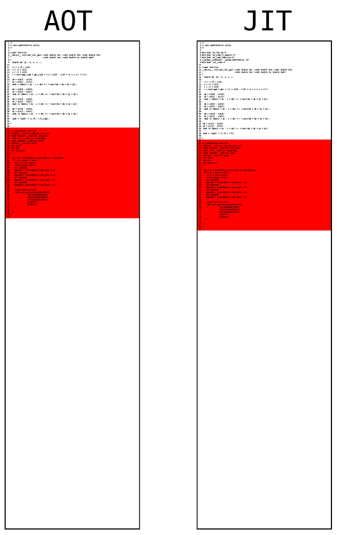
From here onward, all code generated is based only on the kernel descriptor, and not the code that the user wrote for the body of the loop. The kernel function is the same in both files, and is executed on the GPU. It is declared `__global__` so that is exectuted on the device, but can be called from 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 [12, 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[12, p11].

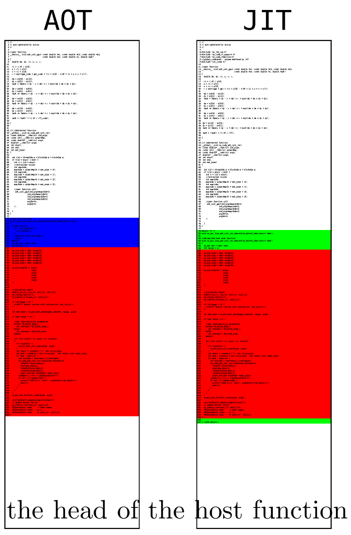
Figure 10: Kernel Function



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 11.

Figure 11: Host Function



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.

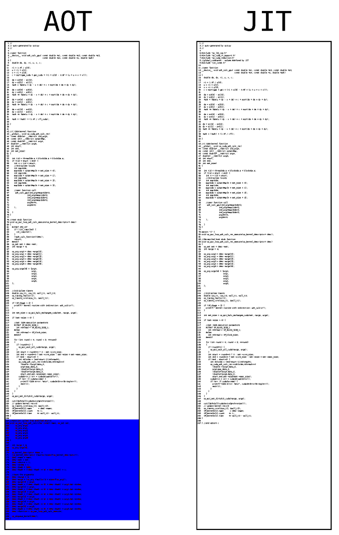
It will be executed after the runtime compiler has been invoked, as the replacement JIT-compiled host function, and make calls to the kernel and user functions in the same file as iteself, rather than those in the AOT file - allowing the optimisations made to be used.

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 12: 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 12).

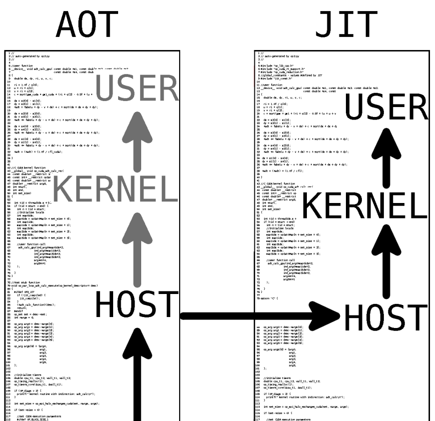
The purpose of this function is to generate the kernel descriptor, then make a call to:

```
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 13 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[12, 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[17] 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 B, 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[4, 3] 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 [11].

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 15a (p32). 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 [14]) 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 15b 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 15c 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 15c . 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 15d, 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 15d . 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 14: Console Output from both binaries

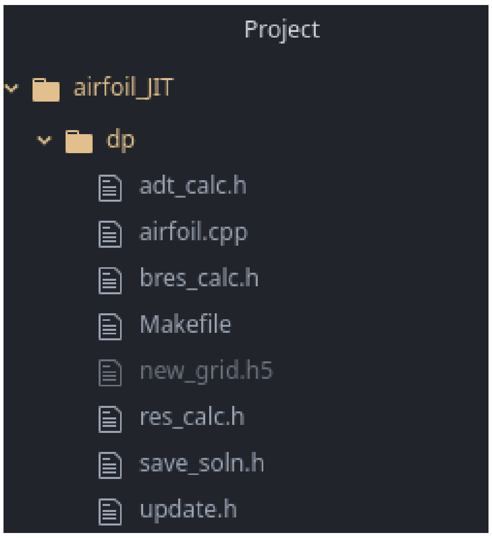
JIT		Enabled	Disabled
Iterations	100	5.02186×10^{-4}	5.02186×10^{-4}
	200	3.41746×10^{-4}	3.41746×10^{-4}
	300	2.63430×10^{-4}	2.63430×10^{-4}
	400	2.16288×10^{-4}	2.16288×10^{-4}
	500	1.84659×10^{-4}	1.84659×10^{-4}
	600	1.60866×10^{-4}	1.60866×10^{-4}
	700	1.42253×10^{-4}	1.42253×10^{-4}
	800	1.27627×10^{-4}	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. Adding a print statement to the generated code for the JIT kernel confirms it is executing the newly compiled functions rather than the originals.

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

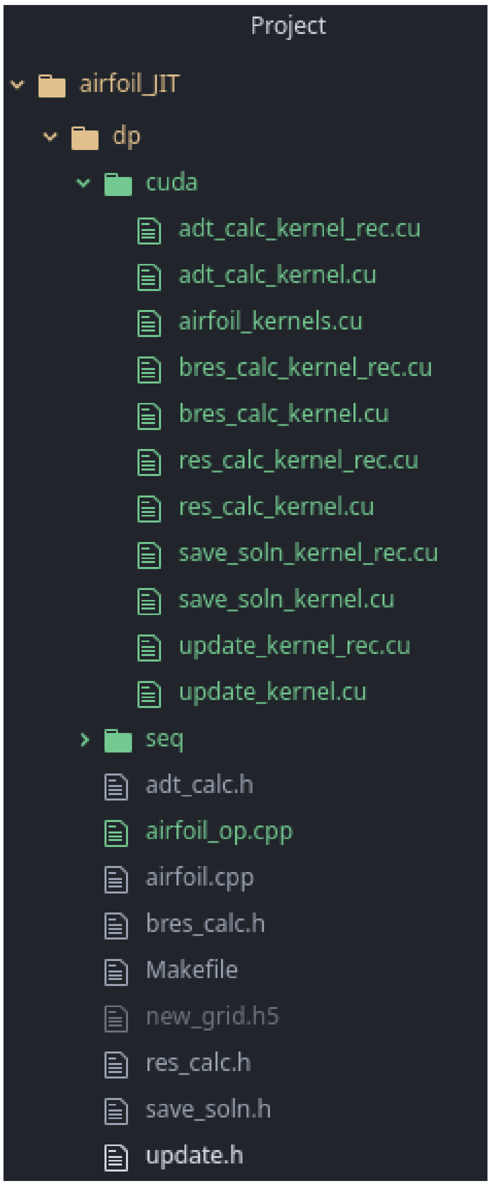
Figure 15: Files in Application Folder

(a) Input Files



```
Project
⌄ airfoil_JIT
  ⌄ dp
      adt_calc.h
      airfoil.cpp
      bres_calc.h
      Makefile
      new_grid.h5
      res_calc.h
      save_soln.h
      update.h
```

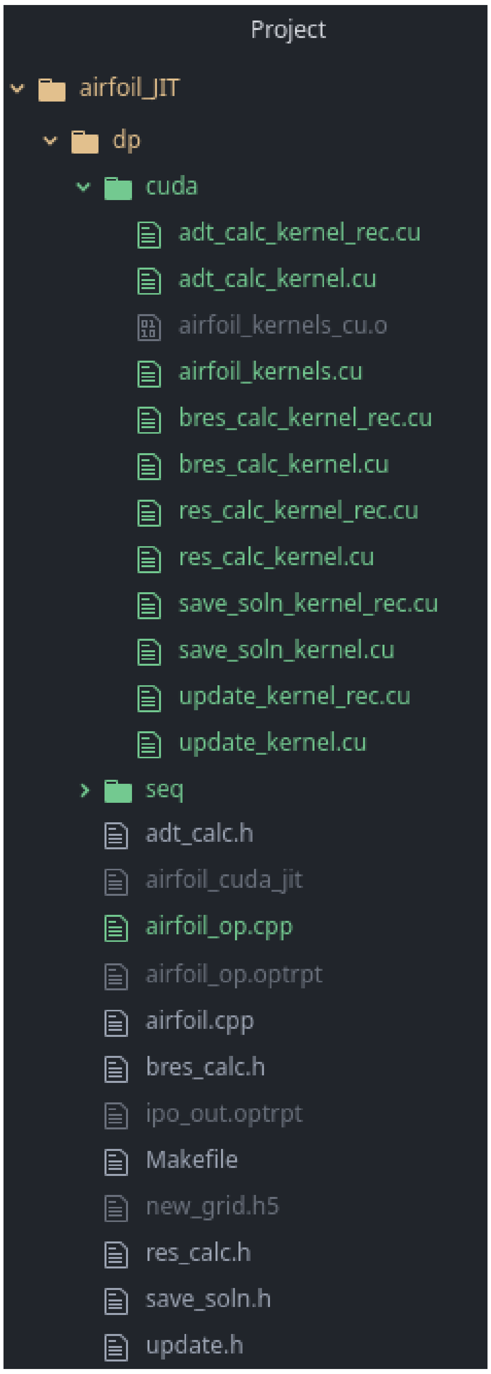
(b) After Code Generation



```
Project
⌄ airfoil_JIT
  ⌄ dp
    ⌄ cuda
        adt_calc_kernel_rec.cu
        adt_calc_kernel.cu
        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_op.cpp
      airfoil.cpp
      bres_calc.h
      Makefile
      new_grid.h5
      res_calc.h
      save_soln.h
      update.h
```

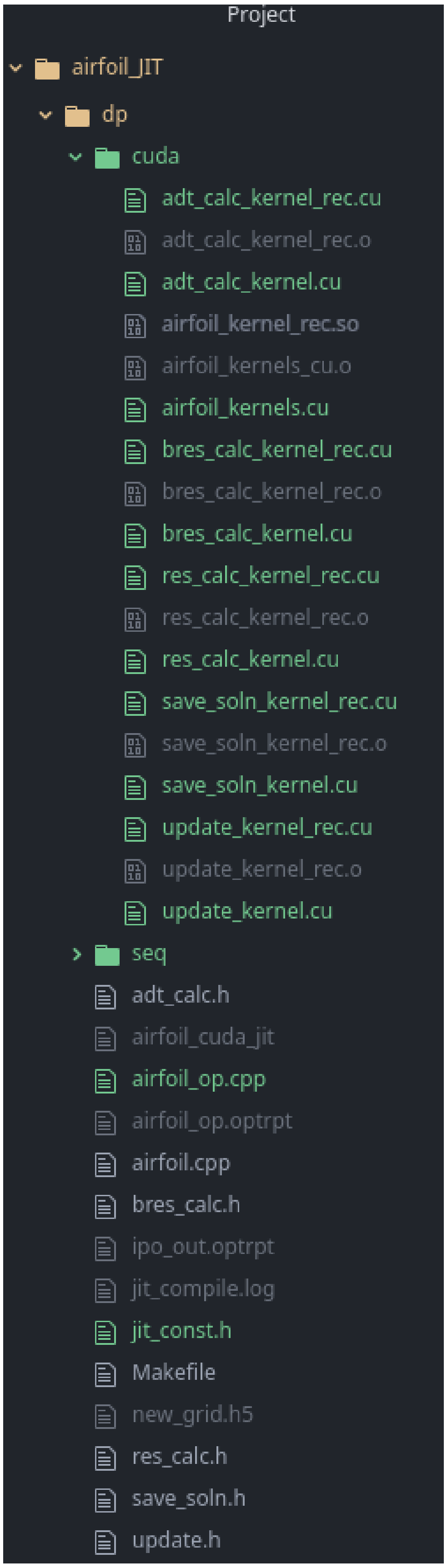
Figure 15: Files in Application Folder

(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

Once the functionality has been confirmed to work as intended, the technique can be benchmarked to determine if there is benefit in using it for run-time efficiency. Testing was done on a personal computer with an NVIDIA GeForce MX250 Graphics Card [6] - and while this is able to execute the CUDA code and ensure it produces the right output, it is not sufficient to gather representative benchmarking data. Using a personal computer system may result in noisy data, from the system scheduling other tasks.

In order to gather better data, access to a supercomputer located in Cambridge, part of the Cambridge Service for Data-Driven Discovery (CSD3), was kindly provided - although workloads for this project were placed in a low priority queue.

The supercomputer named *Wilkes2* was used, which provides 4 NVidia P100 16GB Graphical Processing Units [7]. The translator currently only generates code for a single graphics card, but a possible extension would be to include MPI and divide the workload across multiple GPUs. As with many supercomputer clusters, *Wilkes2* requires jobs to be submitted via SLURM [21].

5.3.1 Benchmarking Strategy

The *airfoil* program is also used for benchmarking, as it is reasonably industrially representative. The input mesh remains the same as in the Testing section, with 721801 nodes, but the number of time steps is upped from 1000 to 10,000 to make any difference more noticable. OP2's internal timing funtions are used to sum the total time spent in each of the parallel loops, which can be compared between the versions with JIT compilation enabled and disabled.

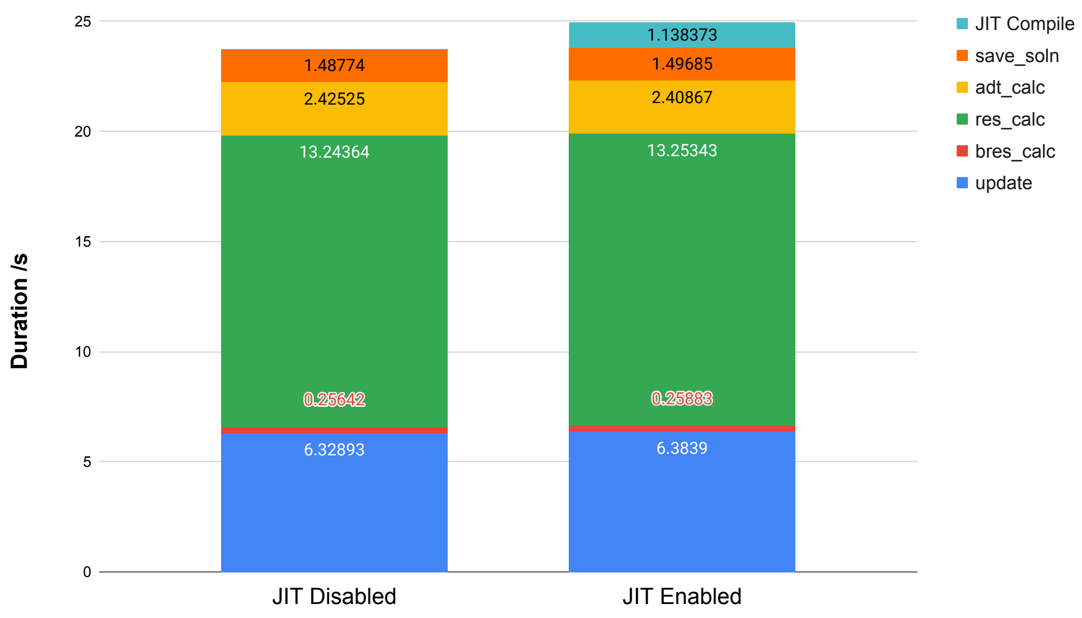
As seen previously in Section 5.2.3 (Just-in-Time Compilation), the time taken for the invocation of the compiler at runtime to complete is also recorded. It is a one-time cost at the start of execution, but still needs to be considered.

Given more time other OP2 applications would also have been used to compare data, however, finding a suitable HPC system and gaining access took a larger portion of the project's duration than expected.

5.3.2 Results

The graph in Figure 16 shows the total runtime of both versions, divided sections corresponding to the total wall clock time spent in each of the parallel loops.

Figure 16: Runtime Divided by Parallel Loop



Values are taken across an average of 10 executions of the binary, to further eliminate any possible noise in the JIT compilation duration.

5.3.3 Analysis

What Figure 16 clearly shows, is that the runtime has not been reduced by using the technique, and indeed is almost the same but with the addition of the time taken to invoke the compiler.

It is important when drawing conclusions from this to remember that there are other assertions that can be made at runtime. Since the only assertion being made is that values declared constant will not change, the time available for optimisation is only the time taken to read constant values from memory, which will not be a significant proportion of the runtime for most projects, since CUDA-capable graphics cards have a designated section of device memory for caching constants [5, p73].

It is true that the constants no longer need to be copied into the device memory, however this was previously only done once at the start of the program.

What the results demonstrate is that more sophisticated optimisations which make use of the inputs being known need to be implemented. Since the JIT Compilation process has now been implemented, this system can continue to be used and improved upon to provide a runtime reduction to the execution of the binary. Even a small improvement to a very large solver, which might run millions of time-step iterations, could quickly re-coup and indeed begin to outweigh the relatively tiny one-time cost of recompilation.

5.3.4 Conclusion

Considering that this project was intended as an investigation, it can certainly be considered successful, despite not achieving the speed-up that was hoped for at the outset. Laying the groundwork for future contributors to build on top of is a worthwhile contribution to the OP2 project, and discovering that the technique of defining constants for the preprocessor is not sufficient to reduce the runtime noticably will inform future investiagations into what techniques should implemented.

6 Evaluation

6.1 Project Management

7 Future Work

8 Conclusion

References

[1] *About Quarkslab.*
URL: `https://quarkslab.com/about/` (visited on 04/15/2020).

[2] *Clang: a C language family frontend for LLVM.*
URL: `https://clang.llvm.org/` (visited on 04/15/2020).

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

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

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

[6] NVidia Corporation. *NVidia GeForce MX250 Specification.*
URL: `https://www.geforce.com/hardware/notebook-gpus/geforce-mx250` (visited on 04/07/2020).

[7] NVidia Corporation. *NVidia Tesla P100 Specification.*
URL: `https://www.nvidia.com/en-gb/data-center/tesla-p100/` (visited on 04/07/2020).

[8] NVidia Corporation. *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Lepler TM GK110.* Version 1.0. 2012.

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

[10] I.Z. Reguly G.D. Balogh G.R. Mudalige et al. "OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling". In: *LLVM Compiler Infrastructure in HPC (LLVM-HPC)* 5 (2018).

[11] 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).

[12] 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).

[13] 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.

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

[15] et al. I. Reguly D. Giles. *The Volna-OP2 tsunami code*. https://www.geosci-model-dev.net/11/4 2018.

[16] C. Bertolli I.Z. Reguly G.R. Mudalige et al. "Acceleration of a Full-scale Industrial CFD Application with OP2". In: *Languages and Compilers for Parallel Computing* (2013), pp. 112–126. URL: `https://people.maths.ox.ac.uk/gilesm/files/OP2-Hydra.pdf` (visited on 04/09/2020).

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

[18] Serge Guelton Juan Manuel Martinez Caamaño. "Easy::Jit: Compiler Assisted Library to Enable Just-in-Time Compilation in C++ Codes". In: *Programming'18 Companion: Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (2018), pp. 49–50.

[19] *Just-in-time (JIT) compiler.*
URL: http://www.cs.sit.kmutt.ac.th/blog/?p=403 (visited on 04/15/2020).

[20] B. Spencer M.B. Giles G.R. Mudalige et al. "Designing OP2 for GPU architectures". In: *Journal of Parallel and Distributed Computing* 73.11 (2013), pp. 1451–1460.
URL: https://www.sciencedirect.com/science/article/pii/S0743731512001694 (visited on 04/09/2020).

[21] *SLURM Documentation.*
URL: https://slurm.schedmd.com/documentation.html (visited on 04/07/2020).

Appendices

A Example CUDA program for vector addition

```
#include<stdio.h>

//Vector Size
#define N 32

//Device Function
__global__ void add(int* a, int* b, int* c)
{
  //perfrom single addition
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
  //store result in c
}

//Generate N random integers, store in a
void random_ints(int* a)
{

  int i;
  for(i=0; i < N; i++)
  {
    a[i] = rand() % 10;
    printf("%02d ", a[i]);
  }
  printf("\n");
}


int main(void)
{
  //Host Arrays
  int *a, *b, *c;
  //Device Arrays
  int *d_a, *d_b, *d_c;

  //Total mem size
  int size = N * sizeof(int);

  //Allocate device mem
  cudaMalloc((void **) &d_a, size);
  cudaMalloc((void **) &d_b, size);
  cudaMalloc((void **) &d_c, size);

  a = (int *)malloc(size); random_ints(a);
  b = (int *)malloc(size); random_ints(b);
  //Allocate and populate a,b

  c = (int *)malloc(size);
  //Allocate c

  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
  //Copy a and b to device memory, store in d_a and d_b

  //Execute in 1 block, N threads
  add<<<1,N>>>(d_a, d_b, d_c);

  //Copy result back from device
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

  for(int i=0; i < N; i++)
  {
    printf("%02d ", c[i]);
  }
  printf("\n");

  //--Free Memory--//
  free(a);
  free(b);
  free(c);

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
  //---------------//

  return 0;
}
```

1. **Compilation:**

```
> nvcc thread_add.cu -o thread_add
```

2. **Result:**

```
> ./thread_add
  03 06 07 05 03 05 06 02 09 01 02 07 00 09 03 06 00 06 02 06 01 08 07 09 02 00 02 03 07 05 09 02
  02 08 09 07 03 06 01 02 09 03 01 09 04 07 08 04 05 00 03 06 01 00 06 03 02 00 06 01 05 05 04 07
  05 14 16 12 06 11 07 04 18 04 03 16 04 16 11 10 05 06 05 12 02 08 13 12 04 00 08 04 12 10 13 09
```

B Getting Started with OP2

Acknowledgements

TODO