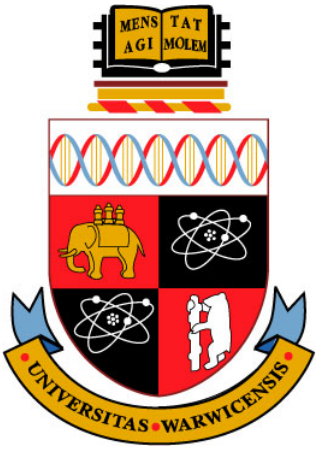


Just In Time Compilation for a High-Level DSL

Nathan Dunne

1604486

3rd Year Dissertation

Supervised by Gihan Mudalige

Department of Computer Science

University of Warwick

2019–20

Abstract

The OP2 Domain Specific Language was created to simplify the process of writing unstructured mesh solver applications for High Performance Computing. This report details the implementation of a new optimisation to the code generation, which re-compiles at run-time when the inputs to the program are known, as well as the benchmarking of this optimisation on a representative example application. For this implementation the only assertion made at run-time was to define input values declared constant as pre-processor literals in the re-compiled code, however further run-time optimisation are also discussed. The finished JIT compilation platform will aid in adding additional optimisations in the future, which could provide speed-up where defining constants could not.

Key Words

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

Acknowledgements

TODO

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 run-time.

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 run-time 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.

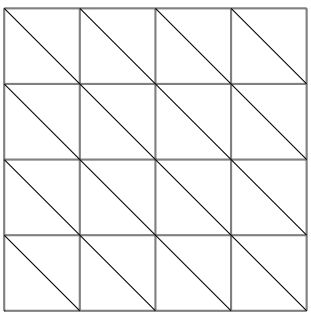


Figure 1: Tri-Structured Mesh

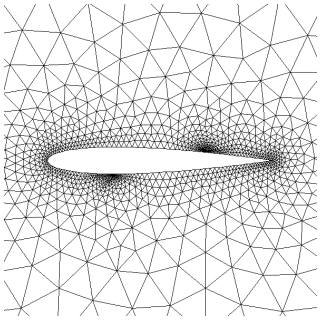


Figure 2: Airfoil Tri-Unstructed Mesh

Unstructed Meshes, such as Figure 2, use connectivity information to specify the mesh topology. The position of elements is highly arbitrary, 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 (p18) details the Implemention and expected results, Section 5 (p39) explains the Testing and Benchmarking of the project, and Section 6 (p49) an Evaluation of the work completed, including Project Management (p51). Lastly, Section **??** (p**??**) contains a discussion on Future Work which could build on top of what was done for this report, and Section 7 (p54) 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* [6], 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:

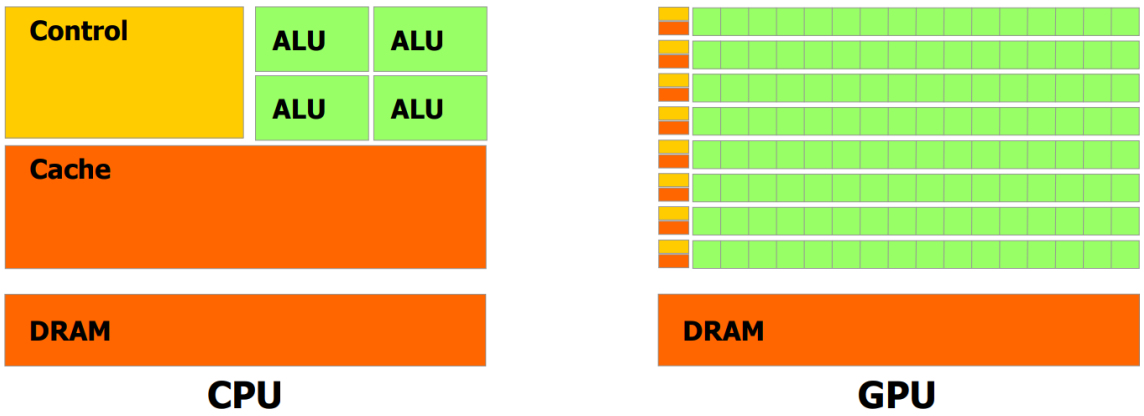


Figure 3: Architecture Comparison. Diagram from [6, p3]

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 [6, p9]. Figure 4 shows an example where 2D identifiers are used.

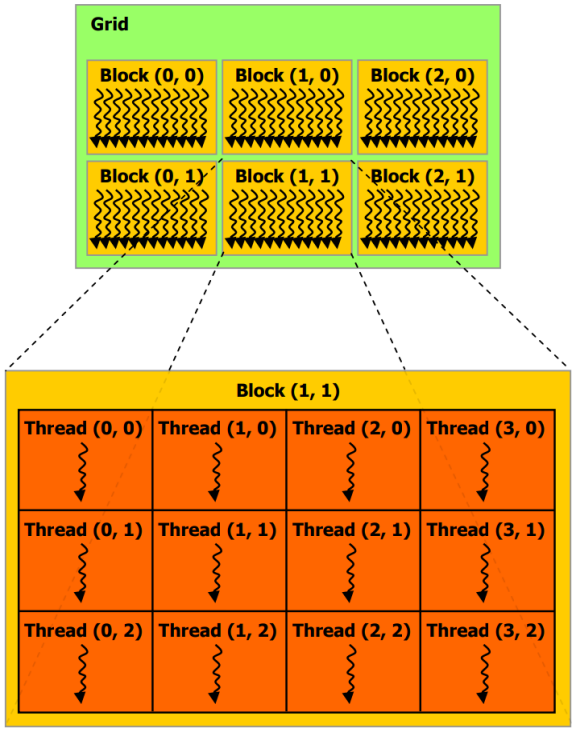


Figure 4: 2D grid of Blocks and Threads. Diagram from [6, p9]

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 [6, 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 [6, 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) [6, 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 [9], for example 8 blocks of 16 $\times$ 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 [15], 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.

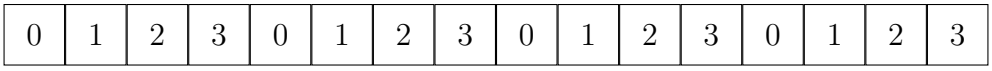
Since this project is focussed on the GPU back-end, and specifically CUDA for NVidia GPUs, the journal article on the design of OP2 for GPU architectures [26] is necessary background material, as it covers a lot of important details from the GPU implementation.
The paper is summerised in the following section:

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.

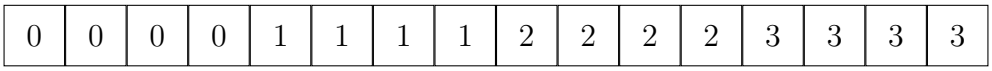
One of the key points made in the paper is on the managing of data dependencies (p1454), and the solutions proposed for avoiding errors when incrementing indirectly referenced array, where two edges end up updating the same node. Solutions

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 [6, 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%.



(a) Array of Structs (AOS) layout



(b) Struct of Arrays (SOA) layout

Figure 5: Data layouts. Diagram reproduced from [26]

The SoA layout is enabled by setting the value of an environment variable: `OP AUTO SOA=1` prior to code generation [14, 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 [14, 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 [13] - a non-linear 2D inviscid airfoil code; Hydra [21] - Rolls Royce's turbomachinery simulator, and Volna [19] - 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.

2.3 Just-In-Time Compilation

Java: The term "Just-In-Time Compilation" is most commonly associated with

the Java Run-time Environment (JRE), as it is integral to the Java Virtual Machine (JVM). In this case, Java compiles code into platform independant "bytecode", then at run-time 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 run-time.

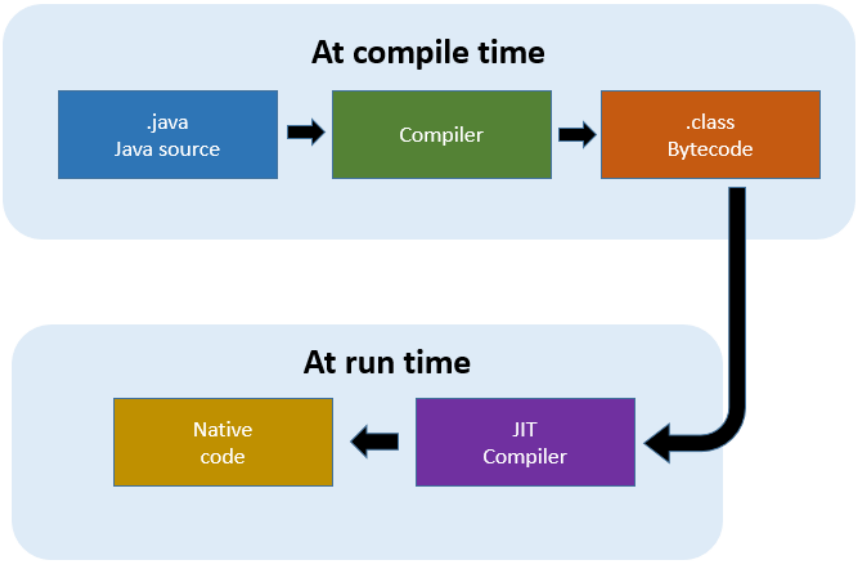


Figure 6: JIT Compilation in the JVM. Diagram from [25]

Chapter 4 of *Java Performance: The Definitive Guide* [27] contains further detail on the JIT compiler.and its impact on the performance of Java.

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 run-time 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.

2.3.1 Related Work

easy::JIT: *easy::JIT* [24] is a library created by Juan Manuel Martinez Caamaño and Serge Guelton of *Quarkslab* [1], and is a closer fit 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, where other C compilers like `gcc` cannot. *easy::JIT* does also differ from this project however, as it utilises code generation at run-time, 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 [12]. The implementation completed for this report will also seek to be compiler agnostic, and therefore will not utilise LLVM.

3 Specification

In order to clearly explain the new system, with the addition that will be made to the OP2 framework, it is important to first describe the exisiting work-flow. The new system model on page 15 may seem complex at first, but understanding the sections that already exisited in the codebase before the start of the project should make it clearer.

3.1 Existing System

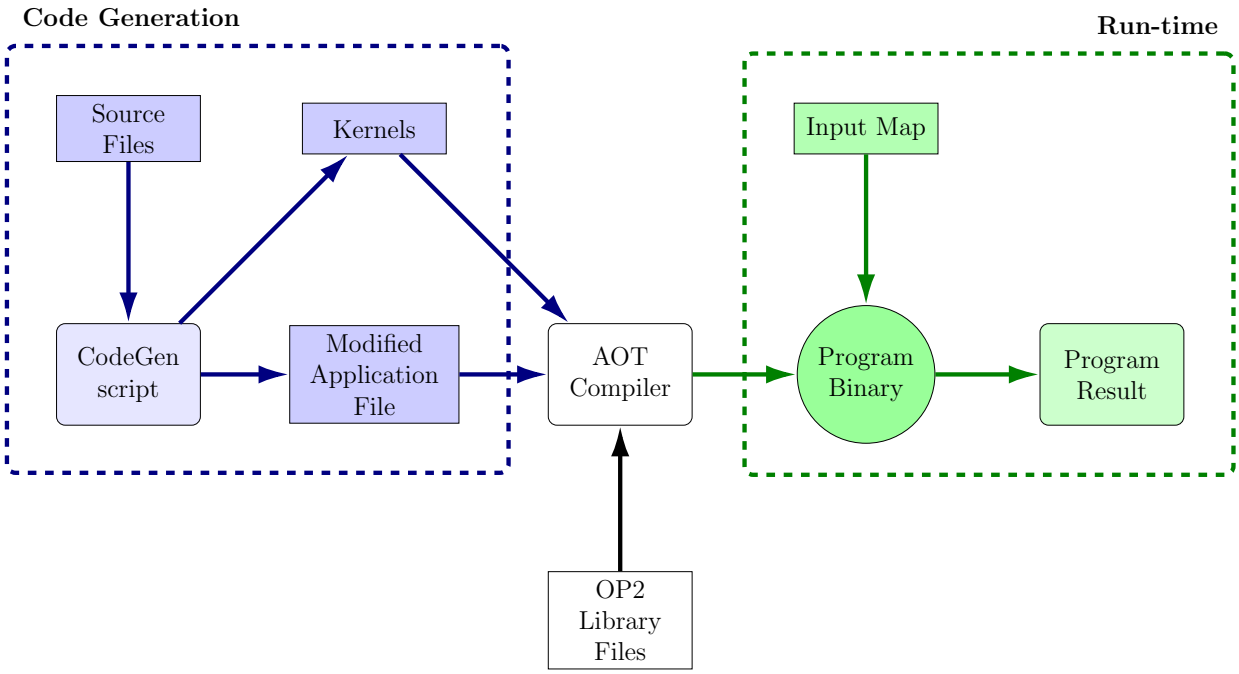


Figure 7: OP2 System Diagram

The pre-exisiting OP2 workflow is shown in Figure 7. In this system the Source Files are contains a master application file, which is a normal C program containing the OP2 API calls for definition of sets, maps, constants, and the structure of the application. The optional C header files containing the operations to be the body of each parallel loop are also part the set of input files.

These Source files form the input to the code generation Python script, from which the output is a modified version of the master appliation file, and a kernel file for each parallel loop. The output is compiled, and linked against the OP2 library for the desired hardware platform to produce an executable Program Binary. The expectation is that this binary will run without error on the target hardware, taking a map as an input to produce the result desired by the application programmer.

The existing system is able to generate optimised code for the target platform from the high level application code, and apply compiler optimisations ahead of time, including optimisations like `-O3` and `--use_fast_math`. It is not, however, able to optimise based on the inputs at all. This is where the implementation for this project comes in: to provide the ability to use the input data when optimising.

3.2 New System

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

The code generation script will need to perform the similar source-to-source translation to the pre-exisiting script, but modified to also generate an altered codebase that will be compiled at run-time, as well as the original code that is compiled ahead of time. These must of course all be valid C files, to be compiled by a normal C compiler. In the case of this project the compiler will be wrapped by the NVidia C Compiler (*nvcc*), as the code generated will be include CUDA. The generated CUDA code must also be valid, and not produce errors when compiled. Finally, the generated code will need to actually invoke the JIT compiler as part of

its execution.

The resulting executable compiled from the generated code must produce an output within some tolerance of the result if the parallel loop iterations are executued in an arbitrary sequential order. OP2 enforces a 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[15, p3], for example through data dependencies. This constraint allows the code generator freedom to not consider the ordering of iterations, and select an ordering based on performance. As with all compilers, correctness will always be a priority over performance.

Outside the code generation script, some OP2 API functions may need to be implemented differently in the OP2 library files, as the functions may need extra information to be stored and retrieved. It is a requirement that the OP2 API itself is not altered by any modifications to the library, so that any and all exisiting programs using the API are able to seemless use the updated version.

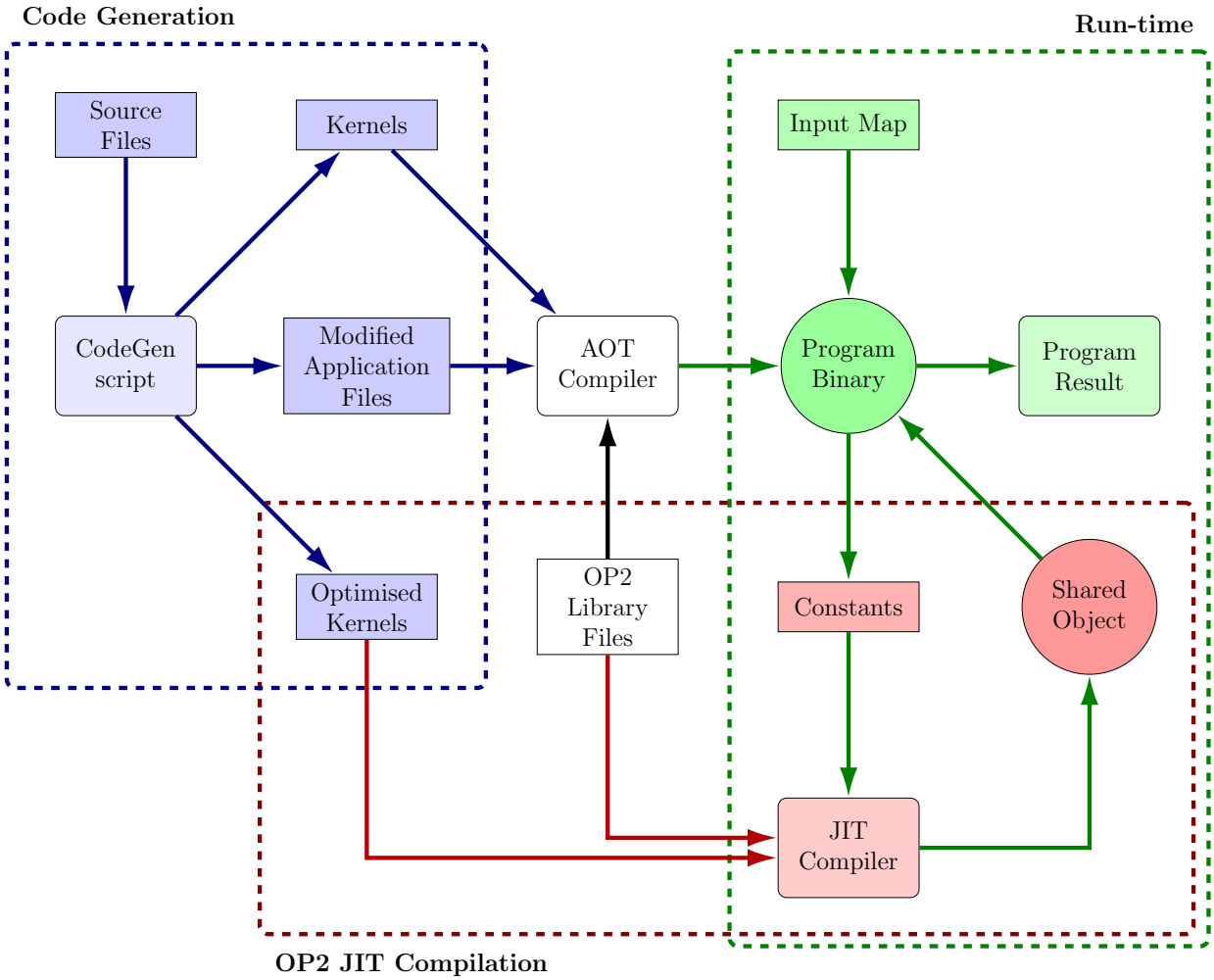
3.3 Run-time Assertions

The application's input will be an unstructured mesh over which to operate, made up of a large amount of data points. The optimisation that will be made for this project is "Constant Definition", which will involve turning values specified in the input as remaining constant into `#define` directives for the C-Preprocessor in the recompiled code. This will make remove the need to store them in memory, and allow them to be read as a literal value, removing the retrieval time when they are required. Other possible optimisations will be discussed in Section 6.1 (Future Work).

3.4 System Model

Figure 8 describes the new workflow of the OP2 library, with 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 an additional set of Optimised Kernels, which contain code that will only be compiled once the constants from the Input Map are known to the program. These Kernel files are not seen by the ahead of time compiler.

Figure 8: OP2 System Diagram with JIT Addition



The JIT compiler also needs to link the Optimised Kernels against the OP2 Library Files, so they must be stored in a location that is also accessible at run-time,

not just when the executable is compiled. This compilation will take place during the execution of the binary, and will therefore make up part of the program's execution duration. It will result in a Shared Object or Dynamically Loaded Library (DLL) file, with a standardised name, which the program can load, and utilise the functions it makes available.

The exported functions will be the recompiled versions of each parallel loop, which as black boxes are equivalent (i.e. they have the same inputs and outputs), however hopefully they are faster to execute than the original versions.

The Kernels compiled Ahead of Time could be altered such that their sole purpose is to invoke the compiler at runtime, then hand off execution to the JIT compiled function. It seems more sensible, however, for them to have the ability to execute the loop body without requiring JIT compilation, as well as being able to re-compiled. Therefore the compiler invokation will be wrapped in a pre-processor conditional, so that the feature can be enabled or disabled using a compiler argument to define a flag, allowing executables with JIT compilation enabled or disabled to be compiled from the same source code.

3.5 Library Modifications

In the OP2 library, the main API function that will need to be modified is:

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

[14, p9]

This function is used to declare a constant value, its dimension, data type, and identifier. Previously this function copied the value to a device symbol, so that when required it could be read from device memory. In this implementation in needs to maintain a de-duplicated list of constants' values and data types, which will be used when the first parallel loop is invoked to generate the head file. At this

point it is known that no more constants can be declared. In the header file each of the constant values will have a `#define` directive making the value available as a literal value.

4 Implementation

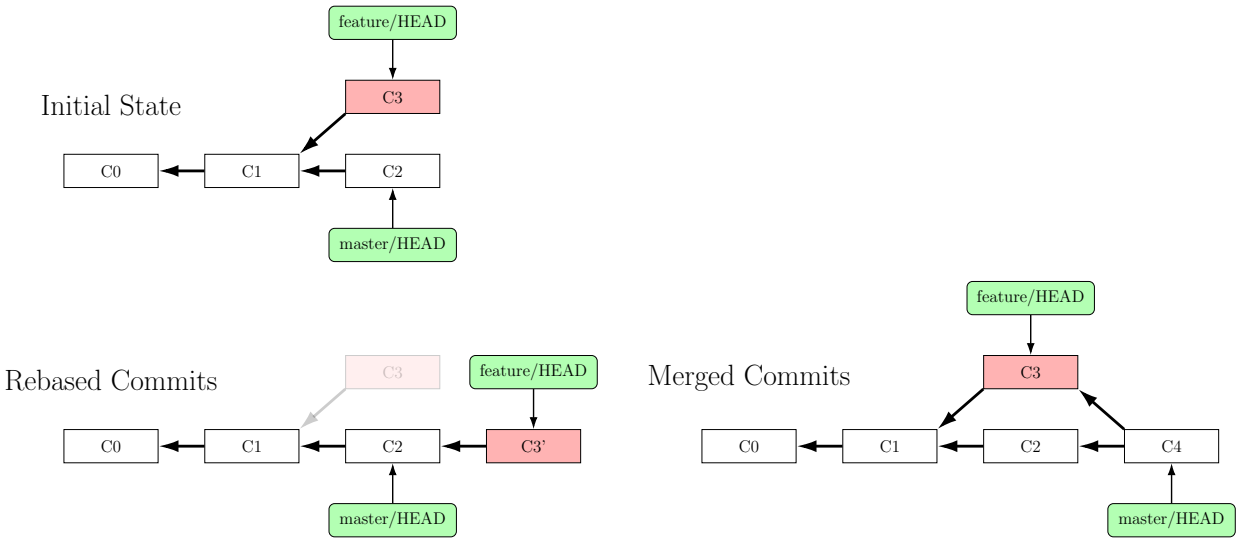
The source code for the OP2 Framework is hosted open source as a GitHub[11] repository. Instructions for obtaining the implementation described in the following section, and for getting started with OP2, are provided be found in Appendix B.

4.1 Git Repository

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

In git terminology, a rebase involves making copies of a branch's commits, and "re-playing" the changes made in them to the top of another branch [17]. In this case, making copies of all commits made to `feature/lazy-execution` and applying them to the latest commit of `master`. The result once any merge conflicts are resolved will be a codebase with all the features of both branches available.

Figure 9: Rebase vs. Merge. Diagram reproduced from [17]



Rebasing is preferable to simply merging for integrating another branch's changes as the result is a linear branch history, rather than creating a diamond. Also, in the case of merge conflicts - where a change has been made in both branches and one needs to be selected - a rebase command will stop at the first conflicting commit and allow the conflict to be resolved [16]. Synchronising using merge would result in receiving all conflicts in one go, which can make it harder to resolve.

The downside of rebasing is it can be harder to recover from an erroneous rebase, than an erroneous merge. This is due to the fact that merges are not destructive, since they do not re-write history in the same way as a rebase. This will not be an issue here however.

The `feature/lazy-execution` branch was created for developing a system to execute parallel loops when values are required, rather than when they are called. This functionality will be achieved using an internal library function:

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

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

Currently this function executes the queued loop as soon as it is invoked, but there is ongoing work into determining when the result of the loop will be needed, and potentially compressing multiple queued actions into fewer at this time. Lazy execution will not be the focus of this project, however the process for invoking parallel loops will be utilised throughout the work done to enable Just-In-Time Compilation for CUDA, so that future efforts towards lazy execution can continue on top of the JIT compilation implementation.

4.2 Code Generation

As described in the Specification before, the majority of the implementation work can be found in a Python code generation script which can be found inthe folder:

`translator/c/python/jit/op2_gen_cuda_jit.py` of the OP2 repository. The code generator for CUDA with JIT compilation is called from another Python script named `op2.py`, which can be found in the parent directory: `translator/c/python/`. This script handles the generation of the Modified Application File, which does not need to change to meet the requirements of this project. The following section will summerise the functionality of `op2.py`, to assist in understanding the context for the CUDA code generation script.

4.2.1 op2.py

`op2.py` is a pre-existing script in the OP2 Framework, which processes application files to gather information to pass to each of the platform specific code generator scripts. It uses Python Regular Expressions to identify OP2 API calls, and ensures certain conditions are met - for example that `op_init` and `op_exit` are called at least once.

It also gathers information about each parallel loop, including the number and type of the parameters, and the details of the indirect data set if the loop is indirect. This stage also includes some error checking, by ensuring types and dimensions are coherent throughout the application.

Once the Application has been analysed, the application file is modified to produce `[-application]_op.cpp`, which is mostly the same as the original application file, but with the addition of `extern` declartions for the function each parallel loop will call: `op_par_loop_[name]`. An implementation for this function will be generated for each hardware platform, including for CUDA with Just in Time compilation - which is the code generator for this project.

Each code generator receives the list of kernel details for each parallel loop as a parameter when it is invoked.

4.2.2 jit/op2_gen_cuda_jit.py

The entry point function for the CUDA JIT code generation is:

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

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

The arguments passed to it from `op2.py` are:

master: The name of the Application 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 element is a map containing many fields describing the kernel.

Where the `kernels` argument serves as the primary input, that the output will be most affected by. The output will be two C source code files, referred to as **kernel files**, for each parallel loop. They will have the following naming scheme:

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

A single **central kernels file** is also generated, which is shared between all parallel loops:

- `cuda/[application]_kernels.cu`

It will contain function definitions required by all loops, or by the master application file; as well as include statements for each of the parallel loops' AOT kernels so they are collated into a single file by the compiler.

The first action of the code generator is to perform a check across all kernels to see if any use the Struct of Arrays data layout [14, p13], or if all are using the

default Array of Structs. Then, a folder `cuda/` is created if it doesn't already exist, and the script will iterate over each kernel, generating both the Ahead-Of-Time (AOT) kernel file, and the Just-In-Time (JIT) kernel file simultaneously. In the System Digram from Section 3 (Figure 8) these files were referred to as "Kernels" and "Optimised Kernels" respectively.

4.2.3 Kernel Files

As mentioned above, the code generator outputs two C source code files for each parallel loop. The following section steps through these kernel files, explaining the purpose of each function that will be generated, and therefore covering a large part of the implementation.

To avoid confusion, Python code that a segment of the implementation will be marked with just a file and line reference, and C code that is an output will be marked *generated by ...* and then a file and line reference.

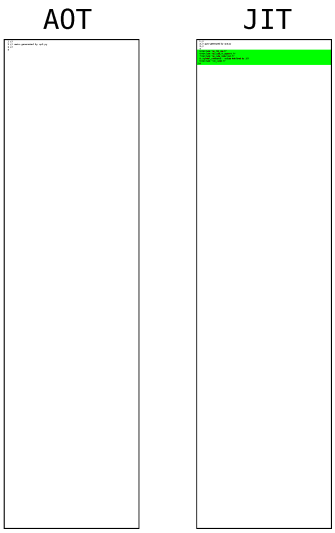
Figures 10-14 show the progression of the two kernel files for an typical parallel loop, and there is a summary on page 33 if the detail is superfluous for your needs.

It may aid in understanding to follow this section with either the translations script, or a set of generated kernel files to hand, since it would not be practical to include full code listings on each page. The inclusion of Figures 10-14 is only for purposes of highlighting the relevant sections of each file, and the generated code in the figures is not intended to be a legible size.

1. JIT includes:

The first piece of C code generated by the Python script is simply the include directives required for the JIT compiled kernel. These are needed for JIT compiled kernels since they will be processed individually by the compiler, so each requires a reference to the OP2 library files. They are not needed by the AOT kernel, as they will be included in the central kernels file, which in turn includes each of the AOT kernel files to produce a single file with all of the AOT kernels and these same `#include` statements:

Figure 10: JIT includes




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

generated by TODO The `jit_const.h` file is also included, which will be generated at run-time (before the compiler is invoked) to contain a `#define` for all input constants, to be handled by the pre-processor.

```
...
//global_constants
#include ‘jit_const.h’
```

generated by TODO

2. User Function:

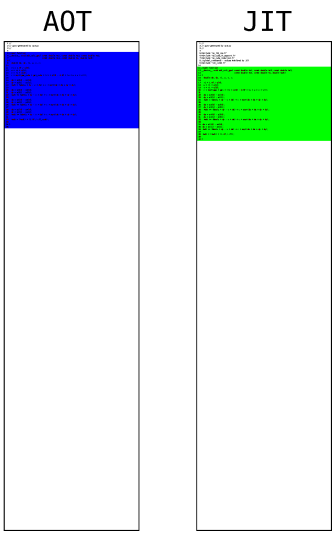
The User Function is the kernel operation specified by the user to be carried out on each iteration of the loop, so this function will run on the device (GPU) on many threads simultanously, performing an action at least once for each set item.

The User Function is given the `__device__` function descriptor, so that it will be compiled for execution on a GPU device, and can only be called from other device code - which will be the next function generated. The whole signature for the function will be:

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

The function body will be pulled from either the application file or a header file, and is checked to ensure it has the correct number of parameters. Any include statements are replaced by the contents of the file, exactly as the pre-processor would.

Figure 11: User Function



Data Layout: If the Struct of Arrays data layout is not requested, the function body will remain largly the same as defined by the application programmer. If it is enabled however there are modifications that need to be made to support this. The code to do this is pulled from the AOT CUDA code generation script: `translator/c/python/aot/op2_gen_cuda_simple.py`.

Since the modifications involve constant values for the stride of different data structures, an attempt to streamline this process using the same constant definition optimisation was made during this project. It was unsuccessful, with a longer discussion on why later in the report.

Optimisations: The constant definition optimisation only needs to be applied to the user function, as it is the only one containing code written by the application developer. Wherever an input constant is referenced it needs to be modified in both the AOT and JIT kernel, but in different ways.

AOT: In the Ahead-Of-Time kernel, which will only be executed if JIT compilation is disabled, 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 constants in the AOT kernel must be replaced with this pattern, using the following lines from the translator script:

```
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. There is no chance of a name collision here since the idenifier will never be allocated memory, only replaced by a literal value.

Multi-Value proved more of a challenge - since values cannot be declared both `__constant__` and defined as external using `extern` [6, p126], which is how they are handled for the sequential JIT implementation.

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 any expression other than a integer literal, this system will run into an issue. As an example, see the result of processing the following statement, where `c_array` is a defined constant with dimension greater than 1:

`int A = c_array[1 + 2]` 1cm⇒ 1cm `int A = op_const_c_array_1 + 2`

If this problem is not solved the most likely outcome is an undefined indentifier error at compile time. In the above example the code will actually compile, but clearly the whole meaning of the statement has changed from the developer's intention, as `op_const_c_array_1` will be replaced by the first value in the array then the literal integer value of 2 will be added to it.

To resolve this, a constant device array is declared in global scope above the top of the function, with the identifier `op_const_[name]`. Each index of the array will be the constant defined for that position. The accesses then can still use the expression for an index, but are modified to instead access the new array, instead of the constant's identifier - so that the meaning of the statement is preserved.

This is only done when an expression index is found and the process becomes necessary, since allocating a new array can take time. If there are no expression accesses, the code will not be generated to handle them.

```
__constant__ int op_const_c_array = { op_const_c_array_1, ...}
 ...
int A = op_const_c_array[1+2]
```

The above is a trivial example, and the actual code is unlikely to be an expression involving only literal values. If it were, then there would be benefit to implementing constant folding to evalute the expression where possible, but this was not done due to the unlikely nature of such code actually being executed.

On the next page is the full Python code for the JIT compiled kernel constants.

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

    # Replace and count all remaining array accesses
    jit_user_function, numFound = re.subn('\\b'+varname+'\[',
                                          'op_const_'+varname+'[',
                                           jit_user_function)

    # At least one expression index was found
    if (numFound > 0):
      if CPP:
        #Line start
        codeline = '__constant__ '           +\
                   consts[nc]['type'][1:-1] +\
                  ' op_const_'              +\
                   varname                  +\
                  '[' + consts[nc]['dim'] + '] = {'

        #Add each constant index to line
        for i in range(0,int(consts[nc]['dim'])):
          codeline += "op_const_"+varname+"_"+str(i)+", "

        # Remove last comma, add closing brace
        codeline = codeline[:-2] + "};"

        #Add array declaration above function
        jit_user_function =  codeline +\
                            '\n\n'    +\
                             jit_user_function
```

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

3. Kernel Function:

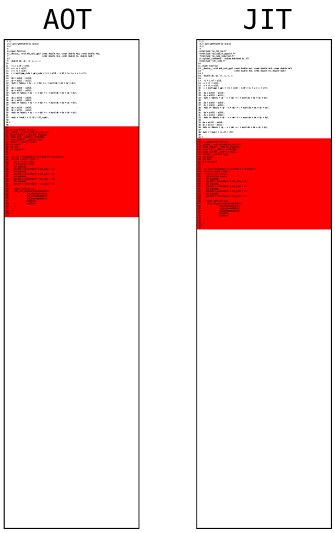
From here onward, all code generated is based only on the kernel descriptor, and does not contain any code written by the application developer.

The kernel function is the same in both files, and is also executed on the GPU across all the parallel threads. 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 purpose of this function is to use the CUDA built in variables `threadIdx.x`, `blockIdx.x`, and `blockDim.x` to determine which section of the workload should be done by each thread. These variables allow a thread to identify iteself from others.

Figure 12: Kernel Function



Indirection: If the loop is indirect, and uses values from another map as indicies, these values need to be read from the inner map in this function, so that the User Function (generated above) can receive all the data it needs without needing to process it. It is possible that the indirect mapping could be optional, in which case the `optflags` argument is checked using a bit comparison to determine if the optional argument was passed or not.

Once this is done, a call is then made to the user function with the parameters it requries, followed by performing any reductions that need to be done. The supported reductions are: sum, maximum, and minimum[14, p11]. Reductions are handled by the `op_reduction` library 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 highlighted in Figure 13.

Figure 13: 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
}
```

Generated by translator/c/python/jit/op2_gen_cuda_jit.py [537-558]

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 the pre-processor flag with identifier `OP2_JIT` has been defined. This allows JIT to be enabled by passing the compiler `-DOP2_JIT` as an argument, and otherwise by default it will be disabled. 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
```

Generated by translator/c/python/jit/op2_gen_cuda_jit.py [522-531]

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 language function linkage, and prevent the compiler from "mangling" the name as it would for C++ code [10]. Following that, the function, which is named `op_par_loop_[name]_rec_execute` ("rec" short for recompiled), will come to reside at the address pointed to by the `[name]_function` function pointer previously referenced in the AOT kernel.

It will be executed after the run-time compiler has been invoked, as the replacement

JIT-compiled host function, and since it makes calls to the kernel and user functions in the same file as iteself, rather than those in the AOT file, the optimisations made to the user function in the JIT kernel are able to be used.

SOA: If the Struct Of Arrays Data layout is enabled, the body of this function will set the stride length for each data structure and copy it to a CUDA device symbol on it's first iteration.

```
if ((OP_kernels[_].count==1) ||
    (direct_[name]_stride_OP2HOST != getSetSizeFromOpArg(&arg_)))
  )
  {
    direct_[name]_stride_OP2HOST = getSetSizeFromOpArg(&arg_);
    cudaMemcpyToSymbol( direct_[name]_stride_OP2CONSTANT,
                        &direct_[name]_stride_OP2HOST,
                        sizeof(int));
  }
```

Generated by translator/c/python/jit/op2_gen_cuda_jit.py [640-654]

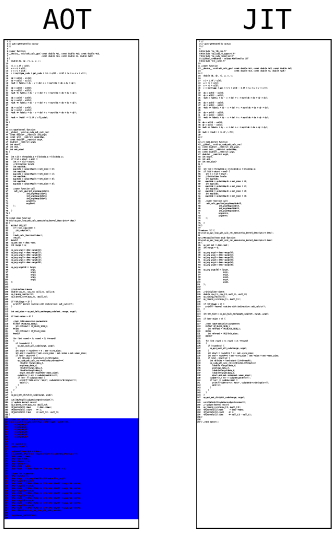
Since these sizes only become available when the function is called and it's arguments are known. It was difficult to replace this symbol copy with a defined constant as each loop would need to have been called at least once before the compilation could be done.

For each loop to execute correctly before JIT compilation, all the input constants would need to be copied as usual so that they can be used on the first iteration of each loop, before the re-compilation is initiated. This is also assuming all loops are executed in turn. It could be the case that a certain loop never gets called, or is only called for the first time half way through an application, in which case any benefit that could be gained from JIT compiling is wasted while waiting for every loop to have been called at least once.

For this reason, the data structure strides remain as a device constant which is copied on the first iteration in both AOT compiled and JIT compiled kernels

5. Loop Function:

Figure 14: Loop Function



The last section to be generated in the kernel files for each parallel loop is the Loop Function, which serves as the entry point for the whole loop operation:

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

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 Host Function acts as the entry point for the JIT compiled kernel.

The purpose of this function is to generate the kernel descriptor for the loop. This is an OP2 data structure that contains the name, operating set, arguments, and execution function of the loop, which is passed as an argument 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.

4.2.4 Kernel Files Summary

To summerise, for every parallel loop two seperate kernel files containing C code are generated, one for Ahead of Time compilation, and one for Just in Time compilation. The code will to be executed when that loop is invoked in the application file, which has been modified to match the function signatures, so the compiler can link the two together.

Figure 15 has been included to clarify the data flow through the two files: starting in the Loop Function at the bottom, which calls the AOT Host Function, where either the re-compiled JIT version is invoked, or the original version is used if JIT compilation is not enabled at when the application is compiled. In the host function the GPU device is invoked, creating many parallel threads, each executing the Kernel and User functions simultaneously.

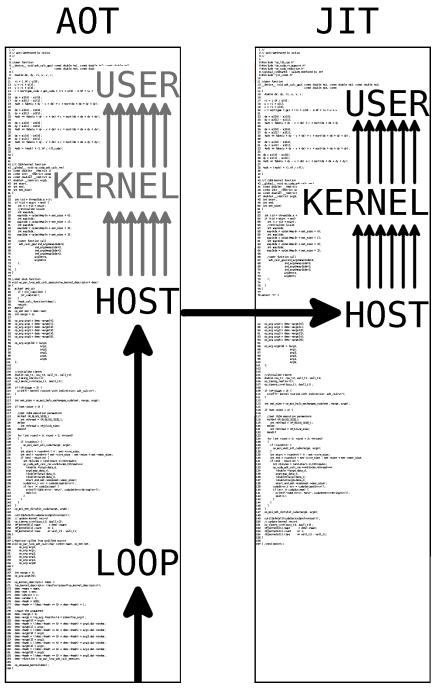


Figure 15: Kernel Flow

The `jit_compile()` function, which is called by the AOT compiled kernel's Host Function when the run-time compilation should happen, has not yet been defined. This will be covered in the next section on the final source file to be generated: the central kernels file.

4.2.5 Central Kernels File

The Central 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 run-time compiler, and declaring constants. It also contains `#include` statements for each of the AOT kernel files, so their contents is imported to make a single file.

When this file is included by the application, the linker will be able to find definitions for each of the parallel loop functions, which were declared extern in the Modified Application File. The compilation process for generated code will be covered further in Section 4.3 on the Makefile.

At the top, the central kernels file includes the requried OP2 library files. It then defines a CUDA constant for each constant the user has defined, which is 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]

```
__constant__ double gam_cuda;
__constant__ double gm1_cuda;
__constant__ double cfl_cuda;
__constant__ double eps_cuda;
__constant__ double mach_cuda;
__constant__ double alpha_cuda;
__constant__ double qinf_cuda[4];
```

example output of previous code segment

Following this, the file contains definitions for two functions. The first is the OP2 API function `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 run-time compiler, load the generated shared object file, and assign a function pointer for each re-compiled loop exported by the DLL so it can be found an executed when required.

1. op_decl_const_char:

This function is an OP2 API function which allows users to declare an input value that will not change over the course of execution. It currently has the following signature, as defined in the OP2 User Guide [14, 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, but only one will be used depeneding on whether JIT compilation is enabled or disabled. The two functions definitions are wrapped with pre-processor conditionals, so that only one of them will be visible to the compiler. As before, the `OP2_JIT` flag being defined is the condition for the JIT functionality to be enabled.

The version of the function for when JIT compilation is disabled is based on the exisiting code generation, and so copies the value passed to it to the corresponding device constant using:

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

The default copy direction for this function is from host memory to device memory, so it does not need to be passed as a parameter.

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 4.2.3 (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 4.2.3 (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.

4.3 Makefile

This implementation relies on GNU Make[23] 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 run-time. 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[5, 4] 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.

4.3.1 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 [13].

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 17a (p44). 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 [18]) 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 4.2; 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 17b 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 17c 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 17c . 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 4.2.5 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 17d, 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 17d . 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 16: 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		$2.484679129111100 \times 10^{-11}\%$	$2.486899575160351 \times 10^{-11}\%$

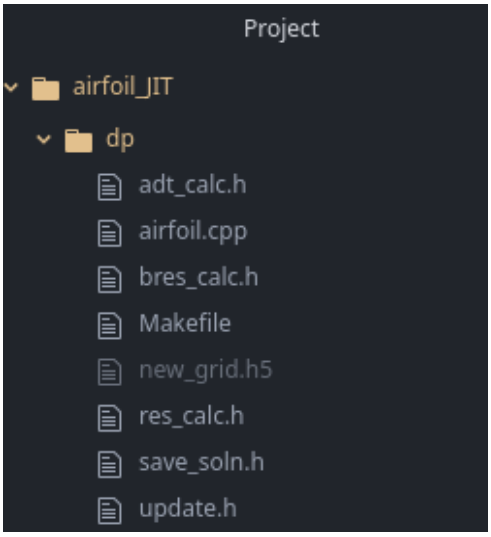
The table shows the result every 100 iterations, as printed to the terminal by the binary, as well as the percentage difference from the exact expected value. Adding a print statement to the

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 17: Files in Application Folder

(a) Input Files



(b) After Code Generation

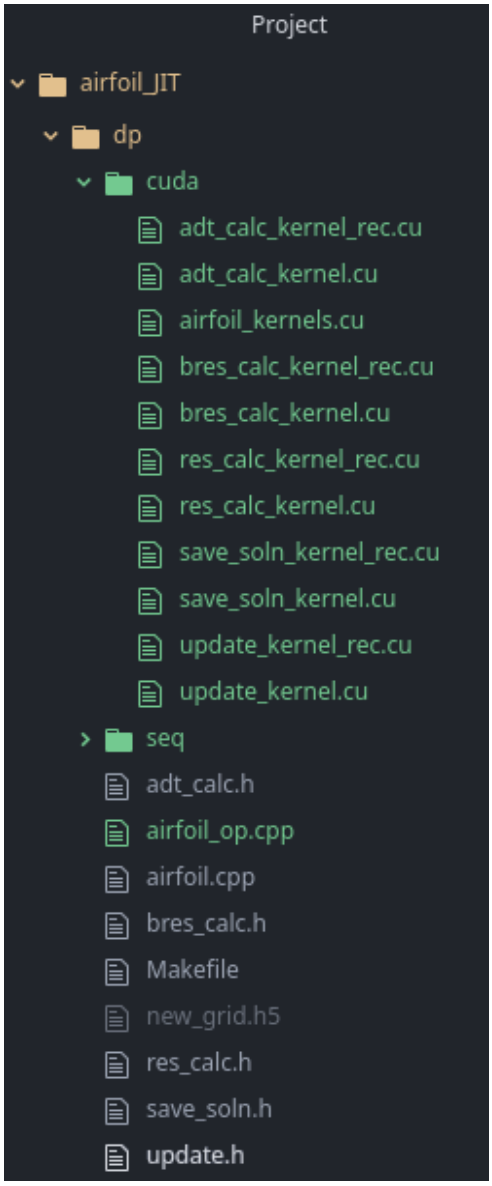
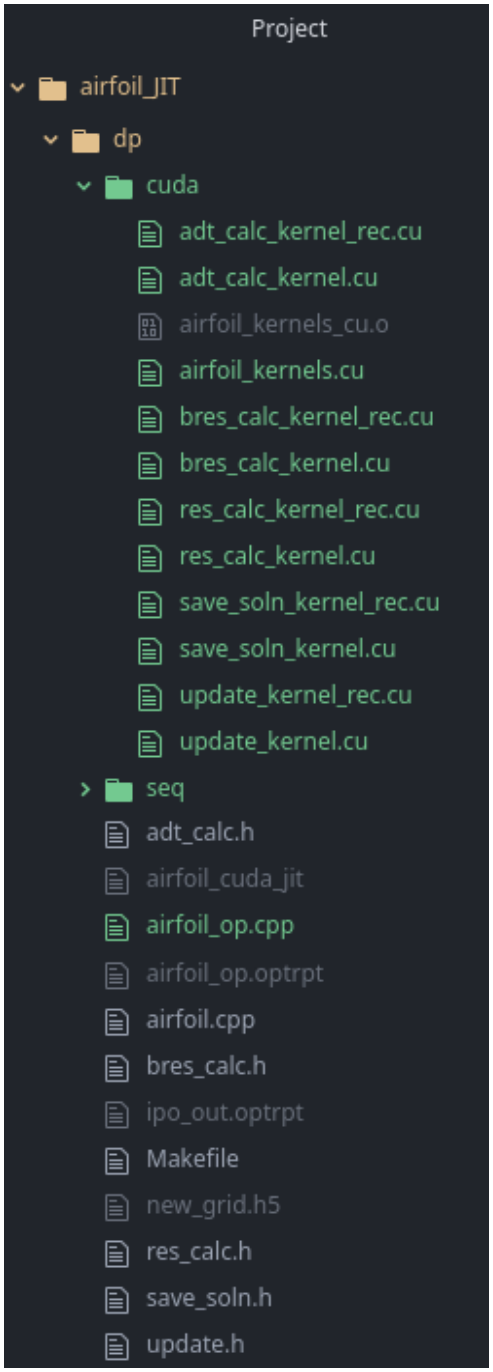
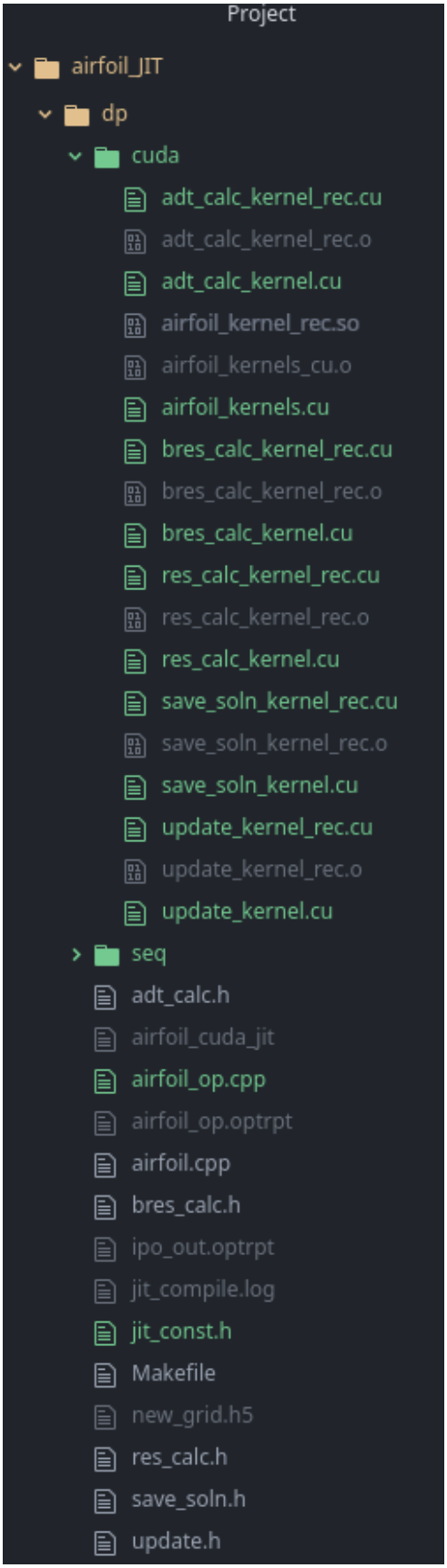


Figure 17: Files in Application Folder

(c) After AOT Compile



(d) After JIT Compile



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 [7] - 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 [8]. 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 [28].

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.

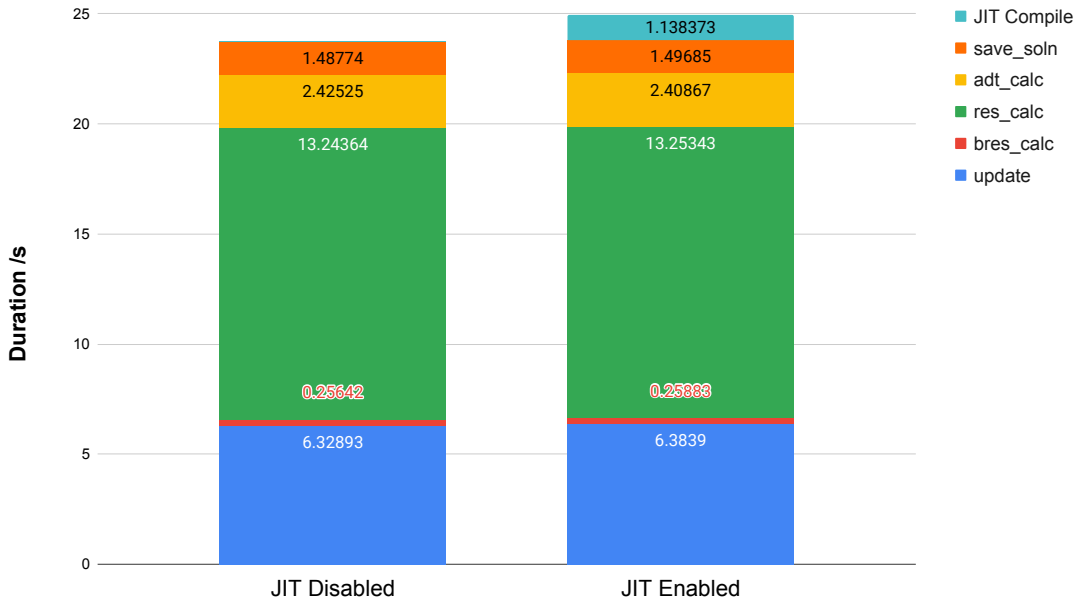
As seen previously in Section 5.2.3 (Just-in-Time Compilation), the time taken for the invocation of the compiler at run-time to complete is also recorded. It is a one-time cost at the start of execution, but still needs to be considered.

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 18 shows the total run-time of both versions, divided sections corresponding to the total wall clock time spent in each of the parallel loops.

Figure 18: run-time 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 18 clearly shows, is that the run-time 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 run-time. 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 run-time for most projects, since CUDA-capable graphics cards have a designated section of device memory for caching constants [6, 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 run-time 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.

6 Evaluation

This project was intended as an investigation, and therefore it can certainly be considered successful, despite not achieving the speed-up that was hoped for at the outset. Through the contributions made to the OP2 project while completing this project, important groundwork has been layed for future contributors to build on top, and implement further optimisations and run-time assertions which might achieve some speedup at run-time.

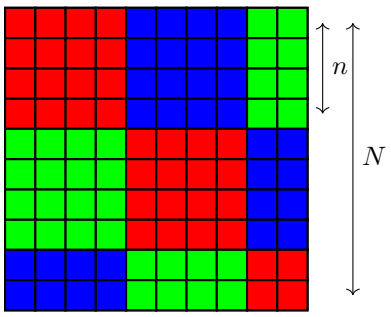
Furthermore, it is useful to discover that the technique of defining constants for the preprocessor does not sufficiently reduce the run-time duration to justify the run-time re-compilation. This will inform future investiagations into what techniques should implemented.

6.1 Future Work

6.1.1 Run-Time Assertions

As previously mentioned, it seems necessary for more to be assertions to be made at run-time in order to produce actual speed-up after JIT compilation. There are a number of possible loop optimisations which could be made, including identifying a loop inside a kernel, and at run-time having the loop bound be hard-coded to remove the need to evaluate the expression of every iteration; or more complex optimisation where two seperate parallel loops might be provably able to be fused into a single loop, but only if the inputs allow for it - meaning this could only be done at run-time.

Figure 19: 2D Loop Tiling



There is also research into applying loop tiling to the generated code, which is dividing the iterations of a loop into sub-regions where both temporal and spatial locality in memory can be exploited.

For example, a loop iterating over a 2D array of size $N \times N$, with Level 1 (L1) cache size of n such that $n < N$ would benefit from dividing the array into squares of size at most $n \times n$, as long as this does not violate any data dependencies in the order of operations (see Figure 19). Doing so prevents values from being

evicted from L1 cache prior to being needed again.

Currently this has only been applied to OPS [22], the precursor to OP2 [20], which supports structured mesh solvers only. There does exist a 2019 paper [**slope**] on automated loop tiling for unstructed meshes, and the issues posed by the need for indirect array accesses. A library provided which demonstrates the technique [3], including a demo using the same *airfoil* application used for this report.

During this project it was suggested that applying loop tiling inside the JIT compilation stage could be a good extension, and would likely provide speedup, however unfortunately there was not sufficient time to reasonably expect the functionality to be finished.

6.1.2 CUDA JIT Compilation

Going in a different direction, the CUDA library does provide an interface for JIT compilation natively, which would allow for re-compilation without requiring a system call to `make` for every loop kernel. System calls can be a significant bottleneck in some cases, and this problem would only compound for applications with a large number of parallel loops. Therefore, using the CUDA JIT compilation system would likely bring down the upfront cost of recompilation. For *airfoil* this re-compile time is very low, it would not have much impact on the results gathered.

Using CUDA's native JIT compilation pipeline would provide the added benefit that a application developer using OP2 would not have to write the Makefile themselves, as currently its contents are not generated by the OP2 code generator, but simply relies on the executable producing an error if a Makefile with the correct target does not exist.

6.1.3 Alternative Hardware Targets

Finally, there are other hardware targets supported by OP2 which may be able to benefit from Just-In-Time compilation, and since the purpose of OP2 is to provide performance on multiple hardware platforms from a single application code any new optimisation which is found to improve performance, should also be ported to other platforms where it might be able to provide benefit. Any users who do not primarly utilise Nvidia GPU hardware should benefit from the JIT compilation optimisation.

6.2 Project Management

This Section serves as a reflection on the project as a whole, and how I believe it went. If this is not of interest the report Conclusion on page 54.

The Gantt chart in figure 20 was produced for the progress report submitted in November, 6 weeks into the project. Having now completed the project I can reflect on how well the timeline was followed, and the successes and challenges of each of the four periods.

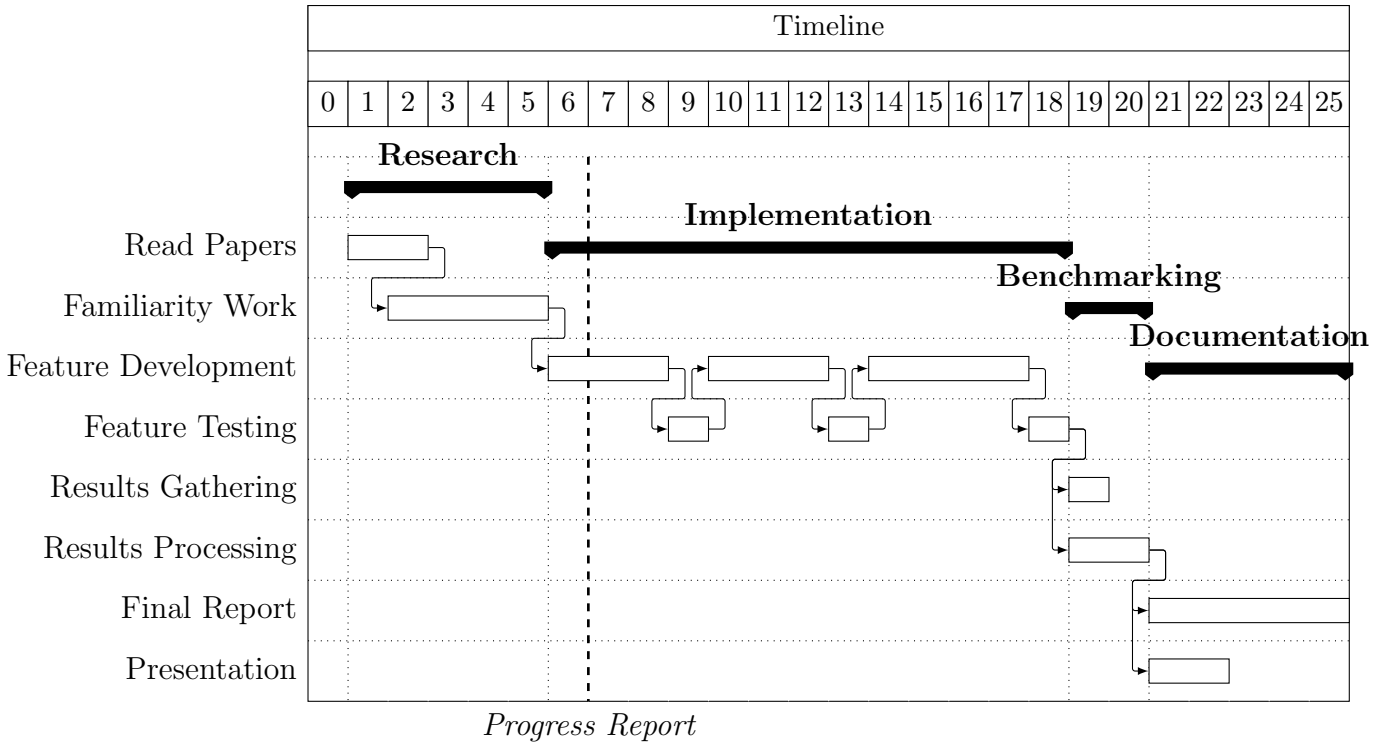


Figure 20: Gantt Diagram produced with Progress Report

6.2.1 Challenges and Reflection

1. Research:

The research section of my project involved reading scientific papers, many produced by contributors to the OP2 framework; and hands-on with CUDA and OP2 trying to build familiarity before the actual implementation began. On reflection, My research was mostly focussed on the existing OP2 work, and many of my sources were from the same authors. Once I had already decided on my approach and begin implementation I discovered some similar work which might have influenced the direction of the project if I had been aware of them earlier on.

2. Implementation:

The Implementation progressed largely as expected, with progress made at a good pace for the time allocated. I did find that while the timetable above listed a total of 3 full weeks for testing, partial solutions were difficult to test fully by as there was no executable generated to ensure the result was correct or perform much benchmarking until towards the end of the implementation.

Instead testing during implementation relied more on comparing expected results from the code generation of airfoil, and modified versions of airfoil, with the actual outputs of the code generation scripts. For some areas of development, I foud it useful to write the code manually into the airfoil files, and figure out how it could work compiling the manually written code, and once it compiled successfully working backwards and modifying the code generator to produce equivalent code that would work for any application.

I had discussed with my supervisor some extension work which could have been a part of the implementation if there was time, such as the Loop Tiling feature mentioned in previous Section 6.1. Since the core functionality of the project was only completed with 2 weeks of planned implementation time left, it was decided that this was an unreasonable goal, and it would be better to thoroughly benchmark the completed functionality than to attempt a complex extension feature and potentially leave it incomplete.

3. Benchmarking:

It was fortunate that the decision to move on to benchmarking instead of pursuing further functionality was taken, as the original plan alloted only a week for gathering results and a second for analysing the results. In reality, the extra 2 weeks that had been allocated Implementation to were also required, as well as part of the time intended for making the project presentation (Week 21. Figure 20).

The delay was mostly due to the desire to use an HPC cluster to gather proper benchmarking data. While the graphics card in my personal laptop was sufficient for validating the code executed correctly, the results would likely have been noisy and inconsistent if not gathered on a dedicated system. Finding a cluster that would allow access to Kepler generation GPUs, and getting approved for access took longer than expected. With the benefit of hindsight it is clear that this process should have begun much earlier in the project, as it was always going to be necessary to have access to an HPC cluster.

Eventually the Cambridge Service for Data-Driven Discovery (CSD3) kindly approved me to use their *Wilkes2* GPU cluster, albeit in the low priority queue. This provided its own challenge, as I often had to wait overnight for results of a submitted job to be provided, or to find out it errored in some way. I was already starting to become familiar with SLURM, the workload manager used to submit jobs, and my knowledge only improved for needing to use it here as well.

4. Documentation:

The last period of work is producing the documentation, which encompasses both creating and giving a presentation on the completed work, and writing this report. The presentation was made using google slides, and I believe it went well, although the demonstration was perhaps not as thourough as it should have been. The report utlilises LaTeX and B, and also was completed well within the alloted time, allowing for sufficient re-drafting.

6.2.2 Tools Selection

I am satisfied with my selection of tools for this project. There was certainly no need to diverge from using GitHub for version control as the rest of the OP2 Framework does, and there have been no issues with using it during this project as I was already very familiar with using `git` prior to starting.

For development, the use of GNU make for AOT compilation is fine enough and very conveniant to combine many commands into a single simple one, however, using it for the JIT compile as well is not an ideal interface for an application developer using OP2.

Lastly, Google Sheets was selected for the presentation because of familiarity, and to ensure changes are automatically saved to a remote in case of loss of data; and LaTeX was used to produce the report.

7 Conclusion

This project was developed as an investigation into a new optimisation for the GPU code generation of the OP2 framework. As part of this investigation, an fully functioning implementation of the technique was designed and completed, and the results benchmarked to determine if the optimisation is able to provide benefit.

The implementation successfuly provides the ability to execute JIT compiled code, and applied an optimisation made based on the inputs of the program, which could only be done at run-time and would not be possible ahead of time: defining the constant values from the input for the pre-processor.

The results from benchmarking were that there was no speedup to the run-time, however it is important to draw the distinction that it was the run-time optimisation of defining of constants which was not able to provide speedup; and not that JIT compilation as a technique does not have potential to speedup.

It is likely that adding loop blocking as a run-time optimisation would produce better results, and adding this feature will have been made easier by the work completed for this project. It is unfortunate that there was not enough time to implement it and benchmark it suffiently as part of this project.

Overall, the project was a successful investigation, which has provided a useful contribution to an open source library. The results gathered will inform and benefit future contributions, and the implementation completed will become part of a framework which provides benefit to many industrial HPC applications.

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] coneoproject. *SLOPE.*

URL: `https://github.com/coneoproject/SLOPE` (visited on 04/16/2020).

[4] NVidia Corporation. *CUDA toolkit.*

URL: `https://developer.nvidia.com/cuda-toolkit` (visited on 04/01/2020).

[5] NVidia Corporation. *NVidia C Compiler.*

URL: `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html` (visited on 04/01/2020).

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

[7] NVidia Corporation. *NVidia GeForce MX250 Specification.*

URL: `https://www.geforce.com/hardware/notebook-gpus/geforce-mx250` (visited on 04/07/2020).

[8] NVidia Corporation. *NVidia Tesla P100 Specification.*

URL: `https://www.nvidia.com/en-gb/data-center/tesla-p100/` (visited on 04/07/2020).

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

[10] *CPP Reference: Language Linkage*. English.
URL: https://en.cppreference.com/w/cpp/language/language_linkage (visited on 04/21/2020).

[11] OP-DSL. *OP2-Common*.
URL: https://github.com/OP-DSL/OP2-Common (visited on 11/05/2019).

[12] 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).

[13] 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).

[14] 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).

[15] 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.

[16] *Git Branchin - Rebasing*. English. Version 2.26.1.
URL: https://git-scm.com/docs/git-rebase (visited on 04/20/2020).

[17] *Git Branching - Rebasing*.
URL: https://git-scm.com/book/en/v2/Git-Branching-Rebasing (visited on 04/20/2020).

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

[19] D. Giles I. Reguly et al. *The Volna-OP2 tsunami code*. https://www.geosci-model-dev.net/11/46 2018.

[20] M.B. Giles I. Reguly G.R. Mudalige et al. *The OPS Domain Specific Abstraction for Multi-Block Structured Grid Computations*. 2014.

[21] 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).

[22] M.B. Giles I.Z. Reguly G.R. Mudalige. "Loop tiling in large-scale stencil codes at run-time with OPS". In: *IEEE Transactions on Parallel and Distributed Systems* (2017).

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

[24] 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.

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

[26] 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).

[27] Scott Oaks. *Java Performance: The Definitive Guide*.

URL: https://www.oreilly.com/library/view/java-performance-the/9781449363512/ch04.html (visited on 04/15/2020).

[28] *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
```

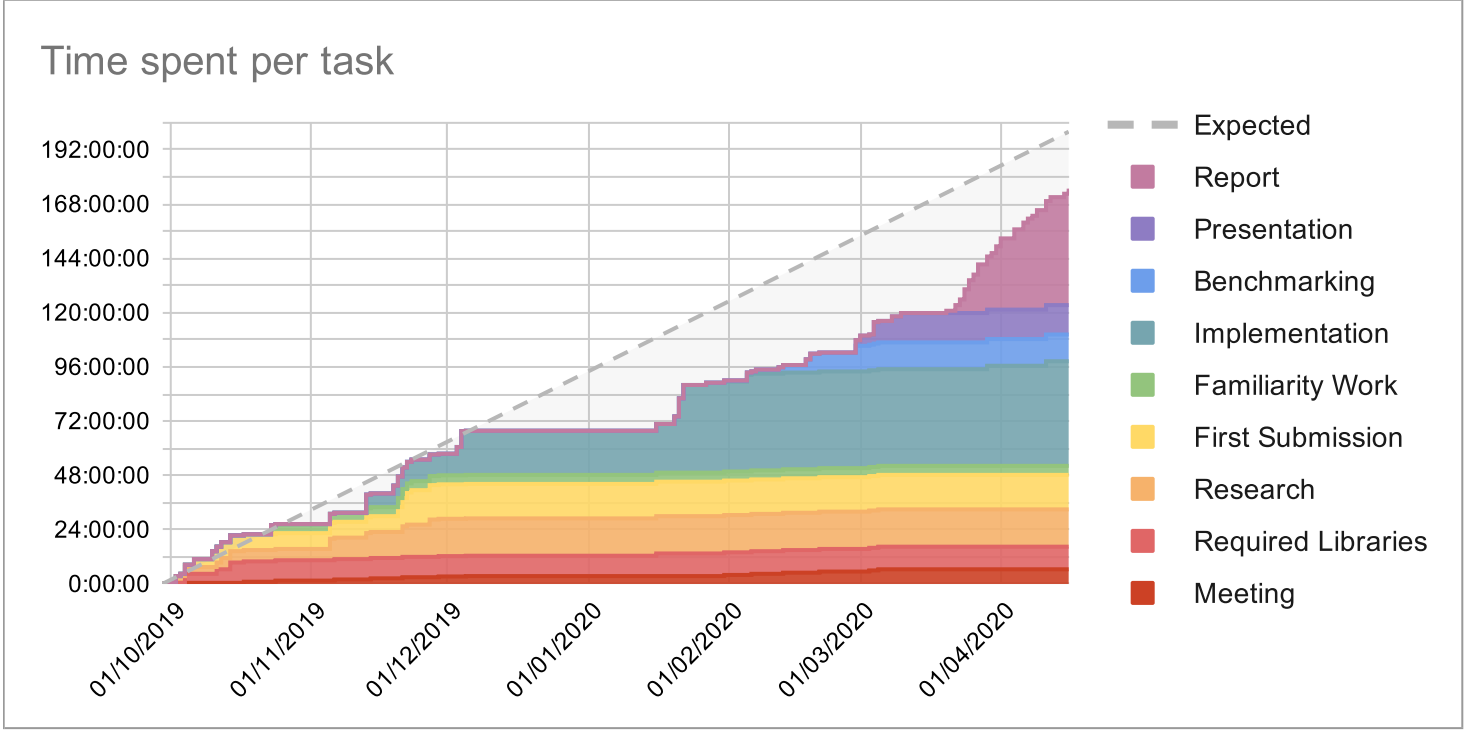
2. **Result:**

```
> ./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

C Time Investment



Date	Expected	Task	Time Spent	Note
29/09/2019	0:00:00		0:00:00	
30/09/2019	0:30:00	Meeting	0:30:00	Initial Meeting, explanation of project and initial work
02/10/2019	2:30:00			
02/10/2019	-0:30:00	Research	3:00:00	Read Papers over summer
03/10/2019	0:30:00			
03/10/2019	-0:30:00	First Submission	1:00:00	Drafted 2 Sections
04/10/2019	0:30:00			
04/10/2019	-3:30:00	Required Libraries	4:00:00	Got ParMetis/ PT-Scotch Sources, Intel Compiler, cuda. Need to build still.
06/10/2019	-1:30:00			
06/10/2019	-4:00:00	First Submission	2:30:00	Gantt Diagram
10/10/2019	0:00:00			
10/10/2019	-3:30:00	First Submission	3:30:00	Draft and submission
11/10/2019	-2:30:00			
11/10/2019	-4:30:00	Required Libraries	2:00:00	Build Libraries and OP2
12/10/2019	-3:30:00			
12/10/2019	-5:30:00	Research	2:00:00	Reading papers
14/10/2019	-3:30:00			
14/10/2019	-6:30:00	Required Libraries	3:00:00	Intel Load Libraries. Matlap eq. Octane to generate meshes. Managed to build and run airfoil_seq, however the numbers don't seem to be correct.
17/10/2019	-3:30:00			
17/10/2019	-4:00:00	Meeting	0:30:00	Skype Meeting, covered issues, next steps (rebase)
23/10/2019	2:00:00			
23/10/2019	-2:00:00	Familiarity Work	04:00:00	Rebase op2, run seq (test PASSED). run mpi found issues.
24/10/2019	-1:00:00			
24/10/2019	-1:30:00	Meeting	0:30:00	Set Timetable for rest of term, resolved issues with Scotch, discussed location for my implementation
05/11/2019	10:30:00			
05/11/2019	6:00:00	Research	4:30:00	cuda tutorial, reading exising translator python files
06/11/2019	7:00:00			

06/11/2019	6:30:00	Meeting	0:30:00	Created remote git branch, set deadline for progress report draft. Discussed progress. Also discussed code guidelines - git clang format
13/11/2019	13:30:00			
13/11/2019	7:30:00	Implementation	6:00:00	Re-did rebase, fixed conflicts. Fixed Makefile for scotch references, and LD_LIBRARY_PATH for HDF5 .so file
13/11/2019	5:30:00	Research	2:00:00	Determining values and origin of paramters to back-end gen functions.
14/11/2019	6:30:00			
14/11/2019	6:00:00	Meeting	0:30:00	
19/11/2019	11:00:00			
19/11/2019	7:30:00	Implementation	3:30:00	Code reading, updated seq_jit with new kernel value initialisation, modified op2.py to call new cuda_jit function
20/11/2019	8:30:00			
20/11/2019	4:30:00	First Submission	4:00:00	Begin Progress report, git cleanup
21/11/2019	5:30:00			
21/11/2019	5:00:00	Meeting	0:30:00	Discuss Progress Report draft
21/11/2019	3:30:00	First Submission	1:30:00	Additions discussed in meeting
21/11/2019	1:30:00	Research	2:00:00	Looked into enqueue_kernel to see how lazy ex is done, also header file creation.
22/11/2019	2:30:00			
22/11/2019	1:45:00	Research	0:45:00	Realised constants are defined by input, understand jit purpose now
22/11/2019	0:00:00	First Submission	1:45:00	Create Flow chart to explain JIT + write explanation
23/11/2019	1:00:00			
23/11/2019	0:00:00	First Submission	1:00:00	Finish Flow chart boxes, wording alterations
27/11/2019	4:00:00			
27/11/2019	1:45:00	Research	2:15:00	Investigate cuda seq generation in existing source, and halo message passing in papers/online. https://www.oerc.ox.ac.uk/sites/default/files/uploads/profile-pages/Gihan/JPDC-OP2.pdf
29/11/2019	3:45:00			
29/11/2019	3:25:00	Meeting	0:20:00	Discussed atomics flag in cuda aot codegen, doesn't change because the choice of codepath is hard coded to not be colouring now.
03/12/2019	7:25:00			
03/12/2019	4:25:00	Implementation	3:00:00	Removed seq codegen and copied jit_include and user_function into file. Soa is Struct of Arrays, and can be forced in type declaration, meaning indicies need to include a "stride".
04/12/2019	5:25:00			
04/12/2019	-1:35:00	Implementation	7:00:00	Continued cuda jit codegen into kernel function files (completed), still errors when building with make - may need to look at const file generation
05/12/2019	-0:35:00			
05/12/2019	-0:45:00	Meeting	0:10:00	Discussed progress with cuda jit, and benchmarking/next steps
16/01/2020	41:15:00			
16/01/2020	39:15:00	Implementation	2:00:00	Makefile, fix cuda function call
16/01/2020	38:15:00	Required Libraries	1:00:00	Update cuda driver
20/01/2020	42:15:00			
20/01/2020	38:55:00	Implementation	3:20:00	Fixing bugs and small mistakes, investigate const for cuda
21/01/2020	39:55:00			

21/01/2020	36:25:00	Implementation	3:30:00	consts with dim1 seem to work using preprocessor #define. multi dim consts tho....
21/01/2020	31:55:00	Implementation	4:30:00	multi dimension constant value reaches kernel, using extern shared memory. Segfault in adt_calc on iteration 235 tho and result seems to be 0 always. Running out of shared memory?
22/01/2020	32:55:00			
22/01/2020	29:25:00	Implementation	3:30:00	Cuda kernels failing to run. Caused by rec/base functions having the same name it seems although unsure exactly why. Restructured codegen to replace func name with name_rec: resolved
22/01/2020	27:00:00	Implementation	2:25:00	decided to break array constants into elements and define? should discuss approach with gihan but it works and completes ~2s slower than aot compiled :/
27/01/2020	32:00:00			
27/01/2020	31:00:00	Implementation	1:00:00	Re-added kernel timers and updated Makefile with JIT/Non-JIT option
31/01/2020	35:00:00			
31/01/2020	34:30:00	Meeting	0:30:00	Discussed completed work, and next steps: Volna translation, Orac Benchmark, Loop Tiling paper
31/01/2020	34:00:00	Benchmarking	0:30:00	Attempted Orac run, encountered issue with Cuda module. Chiron has docs on this but discontinued
05/02/2020	39:00:00			
05/02/2020	38:30:00	Benchmarking	0:30:00	Requested and gained access to Cambridge HPC cluster. No password? Need to speak to Gihan about this
05/02/2020	35:30:00	Implementation	3:00:00	Cloned Volna and MG-CFD for translating. No input data for Volna but found some for MG, and identified issue with translator (#includes going after user defined func). Still seems to be issue with null pointer in compute_step_factor : 30
06/02/2020	36:30:00			
06/02/2020	36:00:00	Meeting	0:30:00	Discussed progress, HPC systems access and term plan. Might need to implement colouring for K80s (Kepler)
07/02/2020	37:00:00			
07/02/2020	36:10:00	Benchmarking	0:50:00	Emailed cam hpc about access. Attempted to generate Volna input data, but issues with supplimentary tool
12/02/2020	41:10:00			
12/02/2020	40:10:00	Benchmarking	1:00:00	Attempting Cam system access, managed to compile but running with slurm indicates an account issue
13/02/2020	41:10:00			
13/02/2020	40:40:00	Benchmarking	0:30:00	Modify Makefile to allow JIT and non-JIT Binaries without rebuilding for comparison
13/02/2020	40:10:00	Meeting	0:30:00	Ran Generated code on Gihan's machine for benchmarking. Observed consistant 6s time lost to compilation
18/02/2020	45:10:00			
18/02/2020	42:40:00	Benchmarking	2:30:00	Added timer wrapper around compilation for comparison. Read about NVRTC - should definitely mention if not implement. Wrestled some more with CAM HPC and managed to submit the jobs. No results yet however
19/02/2020	43:40:00			
19/02/2020	41:10:00	Benchmarking	2:30:00	Got results from Cam system. Unable to build mpi_seq version of OP2 though
21/02/2020	43:10:00			

21/02/2020	42:40:00	Meeting	0:30:00	Disussed next steps, set next meeting for preliminary slides for presentation.
29/02/2020	50:40:00			
29/02/2020	47:40:00	Benchmarking	3:00:00	Get parallel libs onto Cam system, to compile gen_seq OP2, run mpi_genseq airfoil for jit comparison
29/02/2020	45:10:00	Presentation	2:30:00	Colour scheme, basic structure
01/03/2020	46:10:00			
01/03/2020	44:10:00	Presentation	2:00:00	
03/03/2020	46:10:00			
03/03/2020	45:40:00	Meeting	0:30:00	Discussed slides, changes to be made
04/03/2020	46:40:00			
04/03/2020	43:10:00	Presentation	3:30:00	Re-ordered slides,
04/03/2020	42:40:00	Benchmarking	0:30:00	Get Machine specs
04/03/2020	41:10:00	Presentation	1:30:00	Making diagrams
05/03/2020	42:10:00			
05/03/2020	41:40:00	Meeting	0:30:00	Discussed Final slides.
08/03/2020	44:40:00			
08/03/2020	42:40:00	Presentation	2:00:00	Tweaking and Finalising
10/03/2020	44:40:00			
10/03/2020	43:10:00	Presentation	1:30:00	Finalising and giving Presentation
20/03/2020	53:10:00			
20/03/2020	52:10:00	Report	1:00:00	Structure
22/03/2020	54:10:00			
22/03/2020	52:40:00	Report	1:30:00	Intro and Background first draft
22/03/2020	51:40:00	Report	1:00:00	Intro redraft, Background & Motivations
23/03/2020	52:40:00			
23/03/2020	50:10:00	Report	2:30:00	Specification section first draft
24/03/2020	51:10:00			
24/03/2020	48:40:00	Report	2:30:00	Implementation section and tweeks
24/03/2020	46:40:00	Report	2:00:00	Implementation started codeGen breakdown
25/03/2020	47:40:00			
25/03/2020	46:10:00	Report	1:30:00	Continue codegen explanation
25/03/2020	43:40:00	Report	2:30:00	Continue codegen explanation
26/03/2020	44:40:00			
26/03/2020	42:10:00	Report	2:30:00	Continue codegen explanation: Host Function Differences
27/03/2020	43:10:00			
27/03/2020	41:10:00	Report	2:00:00	Finishing kernel file code gen section
27/03/2020	38:40:00	Report	2:30:00	Finished Kernel File section, some tweaks and a long time on one figure
29/03/2020	40:40:00			
29/03/2020	38:40:00	Report	2:00:00	Tweaking figures, User Function section
29/03/2020	37:10:00	Implementation	1:30:00	Adding support for variable index for array constant
30/03/2020	38:10:00			
30/03/2020	36:40:00	Report	1:30:00	Tweaks and redrafting
31/03/2020	37:40:00			
31/03/2020	34:40:00	Report	3:00:00	Redrafting, and Master Kernels File section
01/04/2020	35:40:00			
01/04/2020	34:10:00	Report	1:30:00	Master Kernels File Section, started Makefile section, fixed small bug
01/04/2020	32:10:00	Report	2:00:00	Started Testing section, tested wth icpc and working on gcc
04/04/2020	35:10:00			
04/04/2020	33:40:00	Report	1:30:00	Wrote up Testing Plan, made Figures

04/04/2020	32:10:00	Report	1:30:00	Writing testing results started
04/04/2020	31:10:00	Report	1:00:00	Testing results codegen done
06/04/2020	33:10:00			
06/04/2020	30:10:00	Report	3:00:00	Finished testing section
07/04/2020	31:10:00			
07/04/2020	29:25:00	Report	1:45:00	Benchmarking section done, started results
08/04/2020	30:25:00			
08/04/2020	29:10:00	Report	1:15:00	Finished Benchmarking, start going from start throguh research
09/04/2020	30:10:00			
09/04/2020	27:40:00	Report	2:30:00	Started Research Section
11/04/2020	29:40:00			
11/04/2020	28:10:00	Report	1:30:00	Wrote more research section, discovered issue with SoA codegen
11/04/2020	26:10:00	Implementation	2:00:00	Fixed SoA codgen, attempted to add further optimisation by declaring but proved to be an overcomplication
11/04/2020	25:40:00	Report	0:30:00	Started CUDA research section
12/04/2020	26:40:00			
12/04/2020	24:55:00	Report	1:45:00	Cuda Research Section progress
15/04/2020	27:55:00			
15/04/2020	26:25:00	Report	1:30:00	Related Work research section
16/04/2020	27:25:00			
16/04/2020	25:10:00	Report	2:15:00	Evaluation started